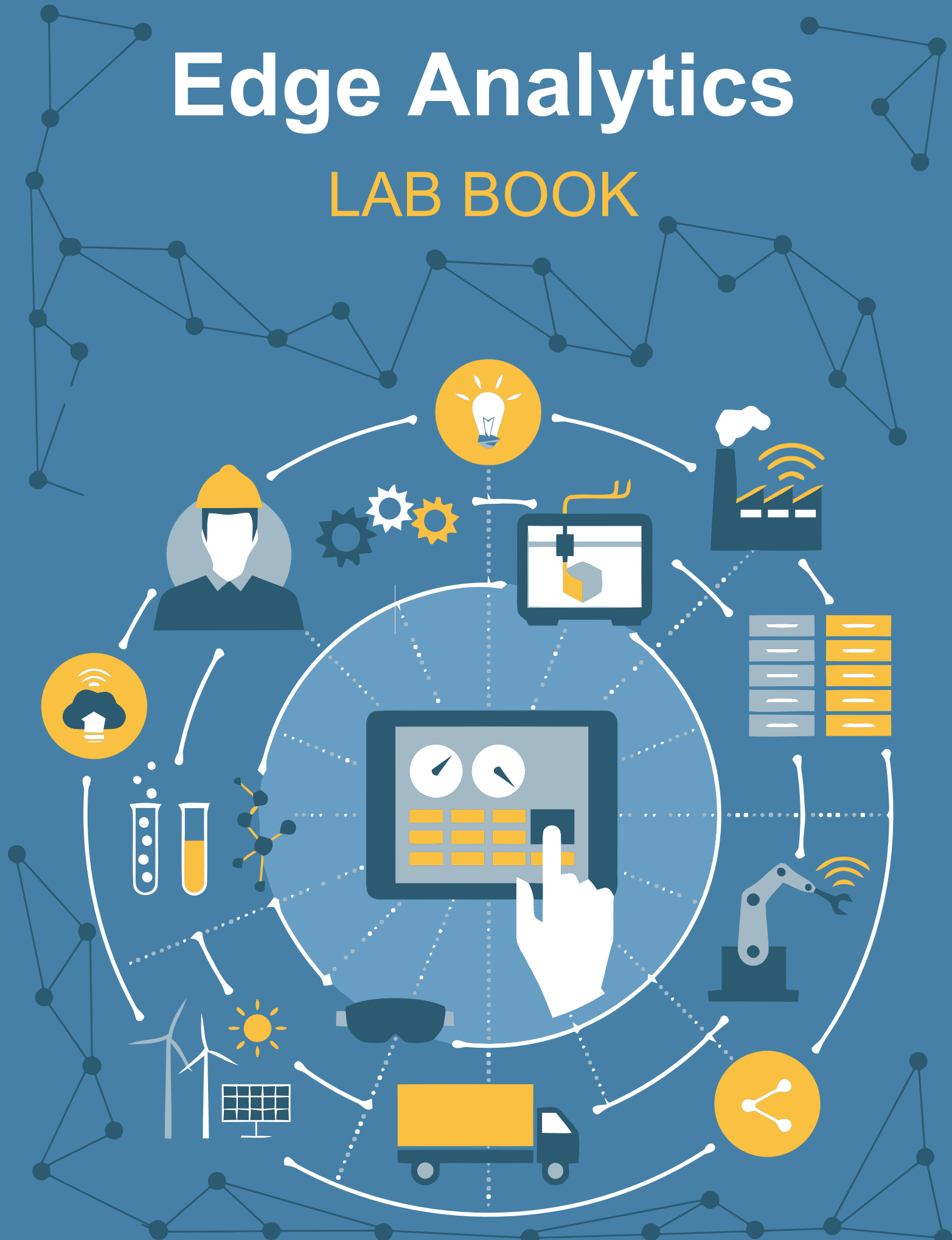


Edge Analytics

LAB BOOK





This Edge Analytics Lab Book is a joint collaboration between *Cardiff University, UK, Indian Institute of Technology, Ropar, India* and *Indian Institute of Information Technology, Kottayam, India* funded by **British Council Going Global India Exploratory Grants Program**

Publisher: *IOT Garage*

Lead Editor: *Charith Perera*



Brand names, logos and trademarks used herein remain the property of their respective owners. This listing of any firm or their logos is not intended to imply any endorsement or direct affiliation with the author.

Version 1.0.0 | January 2026

How to cite this book

Charith Perera (Ed.), Edge Analytics Lab Book, Version 1.0.0, January 2026

Contributing Authors (alphabetical order)

Student contributors and collaborators from the participating institutions.

3	Model Optimization and Quantization for Edge ●	55
3.1	Why Quantization Matters for Edge AI	56
3.2	TFLite Converter: Your First Conversion	60
3.3	Post-Training Quantization in Practice	63
3.4	Exploring Quantization Under the Hood	69
3.5	Quantization-Aware Training (QAT)	72
3.6	Deploying Quantized Models Across Tiers	76
3.7	Guided Experiments	78
3.8	Troubleshooting Guide	80
4	Audio Signal Processing and Keyword Spotting ●	85
4.1	Understanding Digital Audio	86
4.2	From Waveforms to Spectrograms	88
4.3	The Mel Scale and MFCCs	91
4.4	Building a Keyword Spotting Model	94
4.5	Deploying Keyword Spotting Models	100
4.6	Guided Experiments	104
4.7	Troubleshooting Guide	105
5	Embedded Systems Deployment with TensorFlow Lite ●	109
5.1	Understanding Microcontroller Memory	110
5.2	The Tensor Arena: Your Model's Workspace	113
5.3	Converting Models for Microcontrollers	115
5.4	Complete Arduino Deployment	116
5.5	Working with Quantized Models	119
5.6	Integrating Sensors with ML Inference	120
5.7	Profiling and Optimization	125
5.8	Common Issues and Solutions	127
5.9	Complete End-to-End Example: Sin(x) Predictor	132
5.10	Three-Tier Implementation Guide	137
5.11	Guided Experiments	140
5.12	Troubleshooting Guide	141
6	Edge Security and Model Protection ●	145
6.1	Why Security Matters for Edge AI	146
6.2	Building a Threat Model	148
6.3	Input Validation: Your First Line of Defense	150
6.4	Understanding Adversarial Attacks	153
6.5	Protecting Your Trained Model	155
6.6	Secure Communication for IoT	157
6.7	Security Checklist for Edge ML	159

6.8	Three-Tier Security Implementation	160
6.9	Troubleshooting Guide	161
6.10	Guided Experiments	162
7	Convolutional Neural Networks for Computer Vision	167
7.1	Introduction: From DNNs to CNNs	168
7.2	Understanding Convolution Operations	169
7.3	CNN Parameter Counting	171
7.4	Understanding Pooling Layers	173
7.5	Activation Functions in CNNs	174
7.6	Batch Normalization	176
7.7	Building Your First CNN	177
7.8	Visualizing What CNNs Learn	181
7.9	Working with Real-World Images	182
7.10	Avoiding Overfitting with Data Augmentation	184
7.11	Edge Deployment Considerations	185
7.12	Three-Tier Implementation Guide	193
7.13	Troubleshooting Guide	193
7.14	Summary	195
7.15	Guided Experiments	196
7.16	Exercises	198
8	Arduino Sensor Programming for Edge ML	201
8.1	Introduction: Sensors as ML Data Sources	202
8.2	Sensor Fundamentals	203
8.3	Signal Acquisition Theory	204
8.4	Arduino Development Environment	210
8.5	Reading Analog Sensors	211
8.6	Reading Digital Sensors	213
8.7	Sensor Data for Machine Learning	214
8.8	Sensor Signal Preprocessing	216
8.9	Complete Light-Based ML Data Collector	217
8.10	Three-Tier Implementation Guide	219
8.11	Hardware Reference	219
8.12	Troubleshooting Guide	220
8.13	Summary	221
8.14	Guided Experiments	222
8.15	Exercises	223
9	ESP32 Wireless Programming for Edge ML	225
9.1	Introduction: ESP32 for Edge ML	226

9.2	Wireless Communication Theory	227
9.3	Development Environment Setup	233
9.4	WiFi Connectivity	233
9.5	HTTP Communication for Edge ML	237
9.6	Method Selection Guide: Communication Protocols	238
9.7	MQTT for IoT Communication	240
9.8	Bluetooth Low Energy (BLE)	243
9.9	Power Management for Battery Operation	245
9.10	Over-the-Air (OTA) Firmware Updates	248
9.11	Visual Feedback with Neopixels	252
9.12	Complete Edge ML Node	253
9.13	Troubleshooting Guide	255
9.14	Summary	257
9.15	Guided Experiments	258
9.16	Exercises	259
10	Biomedical Signal Processing: EMG-Based Edge ML	263
10.1	Introduction: EMG for Edge ML	264
10.2	EMG Signal Acquisition	265
10.3	Signal Processing Fundamentals	266
10.4	Gesture Classification	284
10.5	Actuator Control	286
10.6	Complete EMG Control System	287
10.7	Three-Tier Implementation Guide	288
10.8	Common Challenges and Solutions	290
10.9	Troubleshooting Guide	290
10.10	Summary	291
10.11	Guided Experiments	291
10.12	Exercises	293
10.13	Exercises	295
11	Edge Device Profiling and Optimization	297
11.1	Why Profiling Matters	298
11.2	Computational Complexity Foundations	299
11.3	Measuring Execution Time	305
11.4	Measuring Memory Usage	307
11.5	Power Consumption Measurement	310
11.6	Python Benchmarking for Model Comparison	312
11.7	ML Inference Profiling	315
11.8	Optimization Guidelines	317
11.9	Complete Profiling Workflow Example	317

11.10	Three-Tier Implementation Guide	321
11.11	Profiling Case Studies	322
11.12	Troubleshooting Guide	324
11.13	Looking Ahead	325
11.14	Summary	325
11.15	Guided Experiments	326
11.16	Exercises	327
11.17	Exercises	328
12	Real-Time Stream Processing and IoT Pipelines	329
12.1	Introduction: From Batch to Stream Processing	330
12.2	Streaming Pipeline Architecture	331
12.3	The Physical Layer: Arduino Light Sensing	332
12.4	Stream Processing Concepts	333
12.5	Stream Processing Theory	335
12.6	The Coordination Layer: Node.js Serial Reader	341
12.7	Real-Time Processing Pipeline	342
12.8	Time-Windowed Aggregation	344
12.9	Visualization Targets	345
12.10	Handling Backpressure	348
12.11	Complete Python Pipeline Example	349
12.12	Three-Tier Implementation Guide	350
12.13	Troubleshooting Guide	351
12.14	Summary	352
12.15	Guided Experiments	352
12.16	Exercises	354
12.17	Exercises	355
13	Edge-to-Cloud Data Ingestion and Storage	357
13.1	Introduction: From Ephemeral to Persistent Data	358
13.2	Distributed Systems Fundamentals	359
13.3	Database Theory for Edge Systems	375
13.4	Edge-Cloud Architecture Patterns	377
13.5	Database Schema Design for IoT	378
13.6	Data Ingestion Patterns	380
13.7	Querying Sensor Data	381
13.8	HTTP API for Data Ingestion	383
13.9	Arduino HTTP Client	385
13.10	Edge vs Cloud Storage Trade-offs	388
13.11	Three-Tier Implementation Guide	389
13.12	Troubleshooting Guide	389

13.13	Summary	390
13.14	Guided Experiments	390
13.15	Exercises	392
13.16	Exercises	393
14	Anomaly Detection on Edge Devices	395
14.1	Introduction: What is Anomaly Detection?	396
14.2	Method Comparison for Edge Deployment	397
14.3	Statistical Methods: Lightweight Detection	399
14.4	K-Means Clustering for Anomaly Detection	402
14.5	Autoencoder-Based Anomaly Detection	404
14.6	Evaluation Metrics	407
14.7	Three-Tier Implementation Guide	409
14.8	Troubleshooting Guide	409
14.9	Looking Ahead	410
14.10	Summary	410
14.11	Guided Experiments	411
14.12	Exercises	412
14.13	Exercises	414
15	Energy-Aware Edge Computing and Optimization	417
15.1	Why Energy Matters for Edge ML	418
15.2	Power Consumption by Component	419
15.3	Energy Cost of ML Operations	419
15.4	Energy Budget Calculator	420
15.5	Duty Cycling for Battery Life	422
15.6	Early Exit Networks	425
15.7	Adaptive Inference Strategies	428
15.8	Measuring Power Consumption	430
15.9	Model Architecture Energy Comparison	431
15.10	Energy Optimization Strategies Comparison	434
15.11	Dynamic Voltage and Frequency Scaling (DVFS)	435
15.12	Three-Tier Implementation Guide	436
15.13	Troubleshooting Guide	437
15.14	Looking Ahead	438
15.15	Summary	438
15.16	Guided Experiments	438
15.17	Exercises	440
15.18	Exercises	443

16	Real-Time Computer Vision and Object Detection	445
16.1	Object Detection Fundamentals	446
16.2	YOLO Detection with OpenCV	449
16.3	Face Detection and Recognition	452
16.4	Edge Deployment: Tiny-YOLO on Raspberry Pi	455
16.5	Video Stream Processing	458
16.6	Three-Tier Implementation Guide	459
16.7	Troubleshooting Guide	460
16.8	Summary	461
16.9	Guided Experiments	461
16.10	Exercises	462
16.11	Exercises	465
17	Federated Learning with the Flower Framework	467
17.1	Introduction: Why Federated Learning?	468
17.2	Federated Averaging (FedAvg)	469
17.3	The Flower Framework	472
17.4	Implementing the Flower Server	472
17.5	Implementing the Flower Client	473
17.6	Running Federated Training	476
17.7	Handling Non-IID Data	477
17.8	Three-Tier Implementation Guide	478
17.9	Troubleshooting Guide	479
17.10	Looking Ahead	479
17.11	Summary	480
17.12	Guided Experiments	481
17.13	Exercises	482
17.14	Exercises	484
18	On-Device Learning and Model Adaptation ●	487
18.1	Why Models Need to Adapt	488
18.2	Mathematical Foundations of Continual Learning	489
18.3	Transfer Learning for Edge Devices	504
18.4	Simulating User-Specific Data	507
18.5	On-Device Fine-Tuning	508
18.6	Preventing Catastrophic Forgetting	509
18.7	Elastic Weight Consolidation (EWC)	512
18.8	Model Versioning and Rollback	513
18.9	Practical Example: Adaptive Keyword Spotter	514
18.10	Three-Tier Implementation Guide	515
18.11	Troubleshooting Guide	516

18.12	Summary	517
18.13	Guided Experiments	518
18.14	Practice Exercises	520
18.15	Exercises	522

Preface

Edge Analytics is a rapidly evolving field that is transforming the way we process and analyze data. With the explosion of Internet of Things (IoT) devices and the growing need for real-time data processing, edge analytics has become a critical technology for many industries, including health-care, manufacturing, and transportation. This Edge Analytics LAB BOOK is primarily compiled to support university-level courses on ‘*Edge Analytics*’ at both undergraduate and postgraduate levels. If you are taking either of these modules, please make sure you follow this lab book.

This booklet does assume that you have some amount of Internet of Things knowledge. Therefore, we cannot teach you all the topics in-depth in a few lab sessions. For example, Arduino programming use C/C++ programming languages. Further, most of the machine learning related labs are thought using Python. However, we do not expect you to be an expert on C/C++ or python to follow the lab session. However, if you have some background, you will find some known concepts in action and feel comfortable. If you have never seen C/C++ or python before, you will, of course, feel nervous and sometimes will feel lost.

For each lab we aim to pair the explanations in this book with runnable Jupyter notebooks stored in the repository under notebooks/ (one notebook per lab), together with the full scripts in MainContent/Code. Students can open these notebooks locally or in a hosted environment to reproduce results, tweak hyperparameters, and explore alternatives discussed in the text.

- We design every exercise with three execution levels: (1) **Notebook-first simulation** that runs entirely in Jupyter on a laptop; (2) **Simulator/emulator** runs (e.g., QEMU or TensorFlow Lite Micro host builds) to mimic microcontroller or Pi constraints; (3) **On-device deployment** to real Raspberry Pi or microcontroller targets. Follow the notebook for level 1, the companion simulator notes for level 2, and the hardware setup checklists for level 3.

Foundational reading that underpins this lab book includes [1, 2, 3, 4, 5, 6, 7, 8].

Throughout the booklet, we have provided explanations and external link and references to reading material. Especially if you do not understand certain programming tasks such as C/C++ or python it may worth reading those links. Further, these link will provide guidance for you to explore the universe of IoT by your own, beyond the labs we have provided here. Finally, we would like to emphasis that this is not a programming course. Therefore, we do not try to teach you a particular programming language (though we try to provide as much links and references for you to develop your skills). It is up to you to develop the gaps in your knowledge by referring to the links we provided.

— **Further information, links and references.** We will use this structure throughout this booklet to provide you with additional information and references. They are marked in orange colour bar on the left-hand side. If you are not interested in exploring further, you are more than welcome to skip these small sections.

Resource Requirements

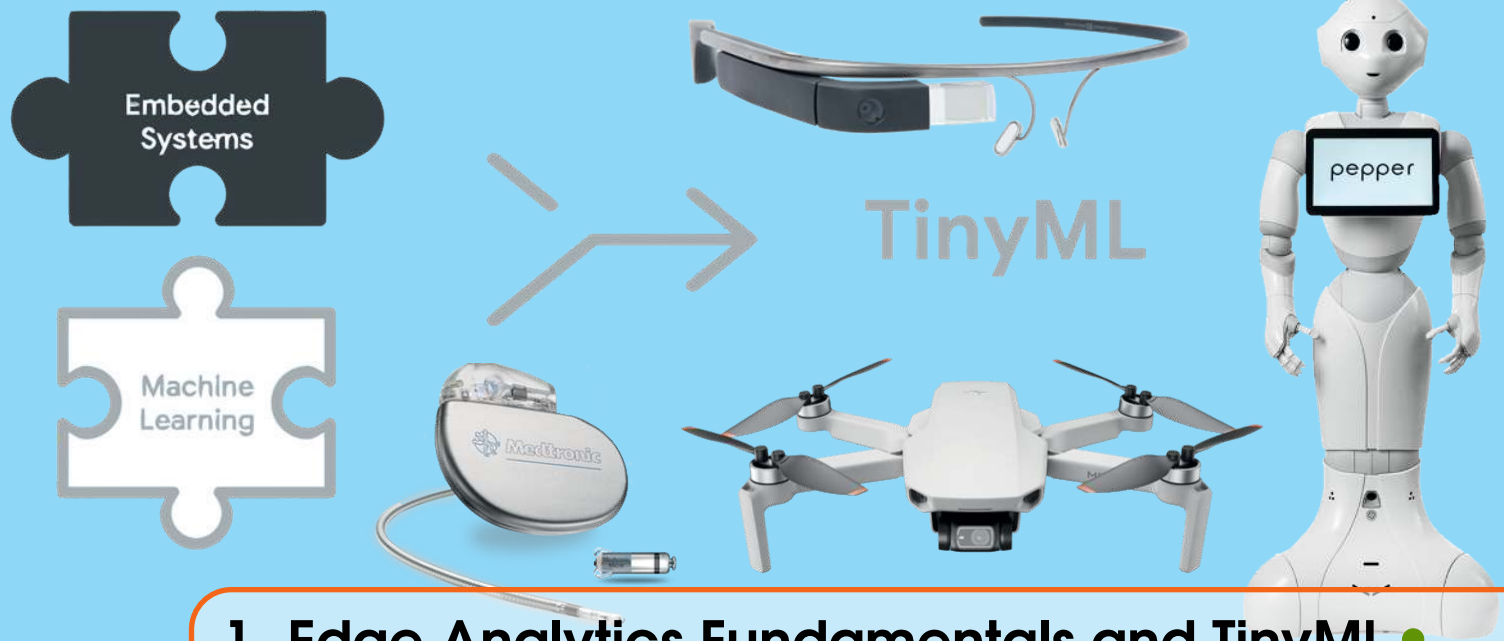
In order to follow all the labs in this lab book, you need the following components. The component requirement for each lab is presented at the beginning of each chapter. You can also see the lab numbers that we use for the items on figure captions.

Hardware Requirements

Arduino Tiny Machine Learning Kit The kit consists of a powerful board equipped with a microcontroller and a wide variety of sensors (Arduino Nano 33 BLE Sense*). The board can sense movement, acceleration, rotation, barometric pressure, sounds, gestures, proximity, color, and light intensity. The kit also includes a camera module (OV7675) and custom Arduino shield to make it easy to attach your components and create your very own unique TinyML project. You will be able to explore practical ML use cases using classical algorithms as well as deep neural networks powered by TensorFlow Lite Micro. For us to be able to have this kit back in stock we produced a Nano 33 BLE Sense without the HTS221 sensor (temperature and humidity), this change does not affect this kit's usage and/or content experience. This board is fully compatible with the kit's documentation.



Figure 1: Arduino Tiny Machine Learning Kit (Code: AKX00028 / Barcode:7630049202771)



1. Edge Analytics Fundamentals and TinyML •

Learning Objectives

After completing this lab, you will be able to:

- Define Edge Analytics and explain its importance in modern IoT systems
- Compare cloud-based ML versus edge-based ML architectures with concrete examples
- Identify the hardware constraints of edge devices (memory, compute, power)
- Describe the TinyML ecosystem, key frameworks, and development workflow
- Understand the three-tier execution model used throughout this course
- Set up your development environment for edge ML projects

Interactive Notebook

Jupyter Notebook: LAB01_introduction.ipynb

[Open in Google Colab](#) | [View on GitHub](#) | [Companion Website](#)

Prerequisites

Before starting this lab, ensure you:

- Basic Python programming (variables, loops, functions)
- Familiarity with arrays/lists and basic data structures
- Curiosity about embedded systems and IoT applications
- No prior ML knowledge required—we start from the beginning!

⚠ Assuming Edge Devices Have GB of RAM

One of the most common misconceptions when starting edge ML is assuming that embedded devices have similar resources to your laptop. When you run TensorFlow on your computer with 16GB of RAM, any model works. But an Arduino has only **256KB**—that’s 65,000× less memory! Always check your target device’s specs before designing your model.

Three-Tier Execution Model

Tier 1: Jupyter/Colab

Explore edge ML concepts, compare cloud vs edge architectures, calculate model memory requirements, set up development environment

Tier 2: Simulator/Pi

Run TensorFlow Lite on Raspberry Pi, measure inference latency, test models with resource constraints

Tier 3: Arduino/ESP32

Deploy “Hello World” model to microcontroller, understand Flash vs SRAM limitations, measure power consumption

Welcome to Edge Analytics! This course will teach you to deploy machine learning models on tiny, resource-constrained devices—from microcontrollers with just 256KB of memory to single-board computers like the Raspberry Pi.

What You’ll Build

By the end of this course, you will be able to:

- Train ML models that fit in less than 100KB
- Deploy voice recognition on a \$15 microcontroller
- Build real-time anomaly detection for IoT sensors
- Create computer vision applications on embedded devices

1.1 What is Edge Analytics?

1.1.1 The Problem with Cloud-Only ML

Consider a simple scenario: you’re building a voice-activated light switch. When you say “lights on,” the lights should turn on. Sounds simple, right?

With traditional cloud-based ML, here’s what happens:

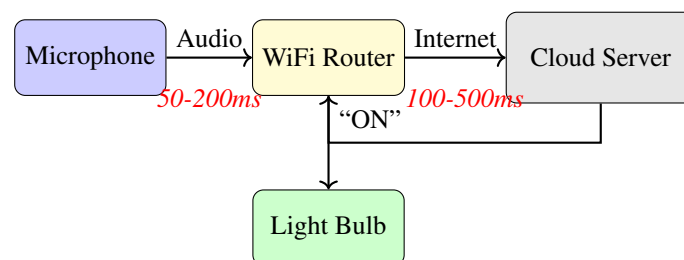


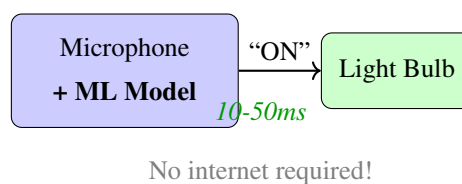
Figure 1.1: Cloud-based voice control: Audio travels to the cloud and back

Problems with Cloud Approach

1. **Latency:** 200-700ms delay feels unresponsive (humans notice delays >100ms)
2. **Privacy:** Your voice is sent to external servers
3. **Reliability:** No internet = no functionality
4. **Cost:** Cloud API calls add up (millions of requests = significant expense)
5. **Bandwidth:** Constant audio streaming uses significant data

1.1.2 The Edge Computing Solution

Edge Analytics processes data locally, at the “edge” of the network—right where the data is generated. For our voice switch:



No internet required!

Figure 1.2: Edge-based voice control: Everything happens locally in milliseconds

Edge ML = Local Intelligence

Edge ML (also called **TinyML** for microcontrollers) means running machine learning inference directly on the device that collects the data. The model lives on the device—no cloud required.

1.1.3 Comparing Cloud vs Edge

Factor	Cloud ML	Edge ML
Latency	100-1000 ms	1-100 ms
Privacy	Data sent externally	Data stays on device
Internet required	Yes	No
Power consumption	High (radio)	Low (local only)
Model size	Unlimited	Limited (KB-MB)
Model accuracy	Highest	Good (optimized)
Cost per inference	API fees	Free (after deployment)

Table 1.1: Cloud ML vs Edge ML comparison

When to Use Each Approach

- **Use Cloud ML** when: You need maximum accuracy, have reliable internet, privacy isn't critical, or the model is too large for edge
- **Use Edge ML** when: You need real-time response, must work offline, handle sensitive data, or need to minimize power/bandwidth
- **Use Hybrid** when: Edge handles common cases locally, cloud handles complex cases

1.2 Real-World Edge ML Applications

Edge ML is already deployed in billions of devices. Here are some examples:

1.2.1 Consumer Electronics

- **Smartphones:** “Hey Siri” / “OK Google” wake word detection runs locally
- **Smart Speakers:** Initial keyword detection on-device, then cloud for complex queries
- **Earbuds:** Active noise cancellation uses ML to adapt in real-time
- **Fitness Trackers:** Activity recognition (walking, running, sleeping) on-device

1.2.2 Industrial IoT

- **Predictive Maintenance:** Vibration sensors detect machine anomalies before failure
- **Quality Control:** Visual inspection at production line speed
- **Energy Management:** HVAC optimization based on occupancy detection

1.2.3 Healthcare

- **Wearable ECG:** Arrhythmia detection alerts without cloud dependency
- **Fall Detection:** Immediate alerts for elderly care
- **Medication Adherence:** Smart pill bottles with activity recognition

Think About Applications

Consider your daily life: What devices could benefit from local ML? Think about:

- What data is being collected?
- Does it need real-time response?
- Is privacy important?
- Is internet always available?

Write down 3 potential edge ML applications you’d like to build.

💡 The Hidden Cost of Cloud

Imagine you’re building a baby monitor that detects crying. Calculate the monthly cost of cloud processing:

- Audio sampled at 16kHz = 16,000 samples/second
- 24 hours/day × 30 days = 720 hours of audio
- Cloud speech API typically costs \$0.006 per 15 seconds
- How much would continuous monitoring cost per month?

Now consider: What if the model runs locally for free after a \$15 device purchase?

1.3 Understanding Edge Device Constraints

Edge devices range from powerful single-board computers to tiny microcontrollers. Understanding their constraints is essential for successful deployment.

1.3.1 The Hardware Spectrum

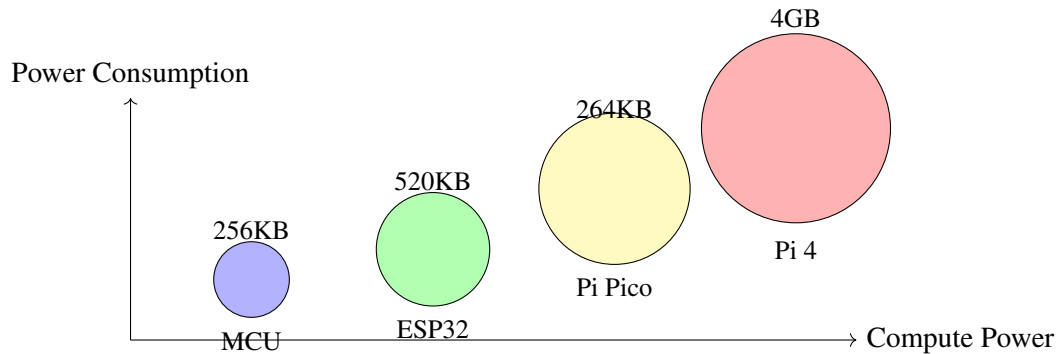


Figure 1.3: Edge device spectrum: More power = more capability but higher energy use

1.3.2 Detailed Device Comparison

Device	CPU	RAM	Flash	Power	Best For
Arduino Nano 33 BLE	64 MHz	256 KB	1 MB	20 mW	TinyML, sensors
ESP32	240 MHz	520 KB	4 MB	100 mW	WiFi + ML
Raspberry Pi Pico	133 MHz	264 KB	2 MB	25 mW	Low-power ML
Raspberry Pi 4	1.5 GHz	4-8 GB	SD card	3-6 W	Edge ML, vision
Jetson Nano	1.4 GHz	4 GB	SD card	5-10 W	GPU inference

Table 1.2: Edge device specifications for ML applications

The Memory Challenge

A typical cloud ML model (ResNet-50) needs **100 MB** of memory. An Arduino Nano 33 BLE has **256 KB** of RAM.

That's a **400x difference!** Edge ML requires aggressive optimization:

- Quantization (float32 → int8): 4x size reduction
- Pruning: Remove unnecessary weights
- Architecture design: Use efficient model architectures

1.3.3 What Fits Where?

Application	Model Size	MCU	Pi
Keyword Spotting (2 words)	20 KB	✓	✓
Gesture Recognition	50 KB	✓	✓
Anomaly Detection	10-50 KB	✓	✓
Image Classification (small)	200-500 KB	Limited	✓
Object Detection (YOLO-tiny)	6 MB	×	✓
Face Recognition	20-50 MB	×	✓

Table 1.3: Typical model sizes and device compatibility

1.3.4 Memory Requirement Calculator

Let's build a practical tool to estimate model memory requirements. This calculator helps you determine if your model will fit on a target device.

```
def calculate_model_memory(num_parameters, dtype='float32'):
    """
    Calculate model memory requirements in KB.

    Args:
        num_parameters: Number of model parameters (weights + biases)
        dtype: Data type ('float32', 'float16', or 'int8')

    Returns:
        Model size in KB
    """
    bytes_per_param = {'float32': 4, 'float16': 2, 'int8': 1}

    if dtype not in bytes_per_param:
        raise ValueError(f"dtype must be one of
        ↪ {list(bytes_per_param.keys())}")

    total_bytes = num_parameters * bytes_per_param[dtype]
    size_kb = total_bytes / 1024

    return size_kb

# Example usage
model_params = 100000
print(f"float32: {calculate_model_memory(model_params, 'float32'):.2f} KB")
print(f"int8: {calculate_model_memory(model_params, 'int8'):.2f} KB")
print(f"Savings: {4.0}x smaller with quantization")
```

This shows that quantization from float32 to int8 reduces model size by exactly 4x—a 100,000 parameter model goes from 391KB to 98KB.

1.3.5 Inference Latency Estimation

Beyond memory, we need to estimate if our model can run fast enough for real-time applications.

```
def estimate_inference_latency(model_flops, mcu_freq_mhz,
    ↪ efficiency_factor=0.5):
    """
    Estimate model inference latency on a microcontroller.

    Args:
        model_flops: Total floating-point operations for inference
        mcu_freq_mhz: MCU clock frequency in MHz
        efficiency_factor: Efficiency (0-1), accounting for memory access,
                           instruction overhead (default: 0.5)

    Returns:
        Estimated latency in milliseconds
    """

    # Operations per second the MCU can theoretically perform
    ops_per_second = mcu_freq_mhz * 1e6
```

```

# Account for real-world efficiency
effective_ops_per_second = ops_per_second * efficiency_factor

# Time in seconds
latency_seconds = model_flops / effective_ops_per_second

# Convert to milliseconds
latency_ms = latency_seconds * 1000

return latency_ms

# Example: Small CNN on Arduino Nano 33 BLE (64 MHz)
model_flops = 5e6 # 5 million operations
arduino_freq = 64 # MHz

latency = estimate_inference_latency(model_flops, arduino_freq)
print(f"Estimated latency: {latency:.2f} ms")

# Check if suitable for real-time audio (10ms window)
if latency < 10:
    print("[OK] Fast enough for real-time audio processing")
else:
    print("[X] Too slow for real-time audio (need <10ms)")

```

1.3.6 Device Compatibility Checker

This function helps you verify if a trained model will fit on your target device.

```

def check_device_fit(model_size_kb, device_ram_kb, device_flash_kb,
                    arena_multiplier=2.5):
    """
    Check if a model fits on a target device.

    Args:
        model_size_kb: Model size in KB
        device_ram_kb: Available RAM on device
        device_flash_kb: Available Flash storage
        arena_multiplier: RAM arena = model_size * multiplier (default:
            ↪ 2.5)

    Returns:
        Dictionary with fit analysis
    """
    required_ram = model_size_kb * arena_multiplier

    fits_flash = model_size_kb <= device_flash_kb
    fits_ram = required_ram <= device_ram_kb

    result = {
        'model_size_kb': model_size_kb,

```

```

        'required_ram_kb': required_ram,
        'available_ram_kb': device_ram_kb,
        'fits_flash': fits_flash,
        'fits_ram': fits_ram,
        'can_deploy': fits_flash and fits_ram,
        'ram_usage_percent': (required_ram / device_ram_kb) * 100 if
        ↪ fits_ram else None
    }

    return result

# Test with common devices
devices = {
    'Arduino Nano 33 BLE': {'ram': 256, 'flash': 1024},
    'ESP32': {'ram': 520, 'flash': 4096},
    'Raspberry Pi Pico': {'ram': 264, 'flash': 2048},
}

model_size = 98 # 98 KB quantized model

print("Model Compatibility Analysis")
print("=" * 60)
for device_name, specs in devices.items():
    result = check_device_fit(model_size, specs['ram'], specs['flash'])
    status = "[OK] CAN DEPLOY" if result['can_deploy'] else "[X] CANNOT
    ↪ DEPLOY"

    print(f"\n{device_name}: {status}")
    print(f"  Model size: {result['model_size_kb']} KB")
    print(f"  Required RAM: {result['required_ram_kb']:.1f} KB")
    print(f"  Available RAM: {result['available_ram_kb']} KB")
    if result['ram_usage_percent']:
        print(f"  RAM usage: {result['ram_usage_percent']:.1f}%")

```

Common Mistake: Ignoring RAM

Students often check if their model fits in flash but forget about **runtime RAM**:

- Model weights (stored in flash, may be copied to RAM)
- Input buffer (e.g., 16KB for audio, 100KB+ for images)
- Intermediate activations (can be larger than the model!)
- Output buffer

Always calculate total RAM usage, not just model size. The `arena_multiplier` of 2.5x in our checker accounts for these extra requirements.

1.4 The TinyML Ecosystem

1.4.1 Key Frameworks

TensorFlow Lite Micro (TFLM) Google's framework for microcontrollers. Runs TensorFlow Lite models on devices with as little as 16KB RAM. Most widely supported.

Edge Impulse Cloud platform for collecting data, training models, and deploying to edge devices. Excellent for beginners with no-code options.

PyTorch Mobile / ExecuTorch Meta's mobile deployment solution. Growing support for embedded devices.

CMSIS-NN ARM's optimized neural network kernels. Often used alongside TFLM for ARM Cortex-M devices.

microTVM Apache TVM's solution for microcontrollers. Compiler-based approach for optimization.

1.4.2 The Development Workflow

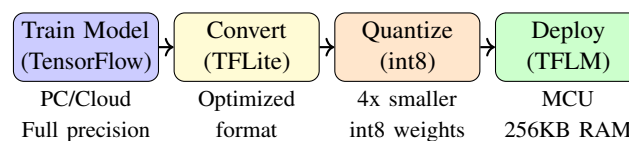


Figure 1.4: Edge ML development workflow: Train → Convert → Quantize → Deploy

Workflow Steps

1. **Train:** Use full TensorFlow/PyTorch on a PC with GPU. Design for edge constraints from the start.
2. **Convert:** Export to TensorFlow Lite format (.tflite). This optimizes the graph structure.
3. **Quantize:** Convert float32 weights to int8. Reduces size 4x with minimal accuracy loss.
4. **Deploy:** Generate C code and compile with your embedded project.

1.5 The Three-Tier Execution Model

Throughout this course, we use a **three-tier approach** that lets you learn progressively without needing hardware for every exercise.

Level 1: Notebook Simulation

What: Run full TensorFlow/PyTorch in Jupyter notebooks

Where: Your laptop, Google Colab, or any computer

Requirements: Python, TensorFlow, Jupyter

Use for: Learning concepts, experimenting with architectures, initial training

Every lab in this course can be completed at Level 1.

Level 2: Simulator/Emulator

What: Test models under edge-like constraints

Where: TFLite interpreter, QEMU emulator, Wokwi simulator

Requirements: TensorFlow Lite, optional: Raspberry Pi

Use for: Testing quantized models, measuring inference time, debugging deployment issues

Recommended for understanding real-world constraints before hardware.

Level 3: On-Device Deployment**What:** Deploy to actual hardware with real sensors**Where:** Arduino Nano 33 BLE, ESP32, Raspberry Pi**Requirements:** Physical hardware, appropriate IDE**Use for:** Final deployment, real sensor data, power measurements*The ultimate goal—your model running on a \$15 device!***Recommended Learning Path**

1. Complete all exercises at [\[L1\]](#) first to understand concepts
 2. Move to [\[L2\]](#) to experience quantization and optimization challenges
 3. Graduate to [\[L3\]](#) for specific projects that excite you
- You don't need hardware to learn 80% of edge ML!

1.6 Setting Up Your Development Environment**1.6.1 Level 1: Notebook Setup (Required)**

```
# Create a virtual environment
python -m venv tinyml-env
source tinyml-env/bin/activate # Linux/Mac
# or: tinyml-env\Scripts\activate # Windows

# Install core packages
pip install tensorflow numpy matplotlib jupyter
pip install scikit-learn pandas
```

1.6.2 Device Comparison Script

Let's create a comprehensive comparison tool that analyzes multiple devices at once.

```
import pandas as pd

def compare_devices_for_model(model_params, dtype='int8'):
    """
    Generate a comparison table showing which devices can run a model.

    Args:
        model_params: Number of model parameters
        dtype: Data type for the model

    Returns:
        pandas DataFrame with device comparison
    """
    # Device specifications
    devices_data = [
        {'Device': 'Arduino Nano 33 BLE', 'CPU MHz': 64, 'RAM KB': 256,
         'Flash KB': 1024, 'Power mW': 20, 'Cost': 15},
        {'Device': 'ESP32', 'CPU MHz': 240, 'RAM KB': 520,
```

```

        'Flash KB': 4096, 'Power mW': 100, 'Cost': 10},
        {'Device': 'Raspberry Pi Pico', 'CPU MHz': 133, 'RAM KB': 264,
         'Flash KB': 2048, 'Power mW': 25, 'Cost': 4},
        {'Device': 'Raspberry Pi 4', 'CPU MHz': 1500, 'RAM KB': 4194304,
         'Flash KB': 32768000, 'Power mW': 5000, 'Cost': 55},
        {'Device': 'Jetson Nano', 'CPU MHz': 1430, 'RAM KB': 4194304,
         'Flash KB': 16384000, 'Power mW': 10000, 'Cost': 99},
    ]

    df = pd.DataFrame(devices_data)

    # Calculate model size
    model_size_kb = calculate_model_memory(model_params, dtype)

    # Calculate required RAM (2.5x model size for tensor arena)
    required_ram_kb = model_size_kb * 2.5

    # Check compatibility
    df['Model Size KB'] = model_size_kb
    df['Required RAM KB'] = required_ram_kb
    df['Flash OK'] = df['Flash KB'] >= model_size_kb
    df['RAM OK'] = df['RAM KB'] >= required_ram_kb
    df['Compatible'] = df['Flash OK'] & df['RAM OK']
    df['RAM Usage %'] = (required_ram_kb / df['RAM KB'] * 100).round(1)

    return df

# Example: Compare devices for a 50K parameter model
model_params = 50000
comparison = compare_devices_for_model(model_params, dtype='int8')

print(f"Device Comparison for {model_params} parameter model (int8)")
print("=" * 80)
print(comparison[['Device', 'RAM KB', 'Model Size KB', 'Required RAM KB',
                  'RAM Usage %', 'Compatible']].to_string(index=False))

```

This comparison table quickly shows that our 50K parameter model (49KB quantized) needs about 122KB of RAM at runtime. The Arduino Nano 33 BLE and Pi Pico can run it, but it will consume nearly 50% of their RAM.

Google Colab Alternative

Don't want to install anything? Use Google Colab:

1. Go to <https://colab.research.google.com>
2. Upload the course notebooks
3. TensorFlow is pre-installed!

1.6.3 Level 2: TensorFlow Lite Setup

```

# TensorFlow Lite Runtime (lighter than full TF)
pip install tflite-runtime

```

```
# For Raspberry Pi  
pip install tflite-runtime # ARM version auto-selected
```

1.6.4 Level 3: Hardware Setup

Arduino Nano 33 BLE Setup

1. Install Arduino IDE 2.x from <https://www.arduino.cc/en/software>
2. Add board support: Tools → Board Manager → “Arduino Mbed OS Nano Boards”
3. Install TensorFlow Lite library: Tools → Library Manager → “TensorFlowLite”

ESP32 Setup

1. Install ESP-IDF or use Arduino IDE with ESP32 support
2. Add ESP32 board URL to Arduino preferences
3. Install ESP32 TensorFlow Lite library

1.7 Course Roadmap

Lab	Topic	Key Skills	Hardware
1	Edge Analytics Fundamentals and TinyML	Concepts, setup, trade-offs	None
2	Neural Network Training Fundamentals	Loss, gradient descent, NNs	None
3	Model Optimization and Quantization for Edge	TFLite, int8, optimization	None
4	Audio Signal Processing and Keyword Spotting	Audio ML, MFCCs, wake words	Optional: mic
5	Embedded Systems Deployment with TensorFlow Lite	TFLM, C++ integration	Arduino/Pi
6	Edge Security and Model Protection	Model protection, attacks	None
7	Convolutional Neural Networks for Computer Vision	Convolutions, image ML	None
8	Arduino Sensor Programming for Edge ML	Sensor integration, acquisition	Arduino
9	ESP32 Wireless Programming for Edge ML	WiFi, BLE, MQTT	ESP32
10	Biomedical Signal Processing: EMG-Based Edge ML	EMG, health sensors	Sensors
11	Edge Device Profiling and Optimization	Performance analysis	Any
12	Real-Time Stream Processing and IoT Pipelines	Queues, windows, MQTT	Pi
13	Edge-to-Cloud Data Ingestion and Storage	Persistence, SQL, pipelines	Pi
14	Anomaly Detection on Edge Devices	Autoencoders, IoT anomalies	Sensors
15	Energy-Aware Edge Computing and Optimization	Power optimization	Any
16	Real-Time Computer Vision and Object Detection	Object detection, NMS	Camera
17	Federated Learning with the Flower Framework	Privacy, distributed training	Multiple
18	On-Device Learning and Model Adaptation	Training on edge	Pi

Table 1.4: Course roadmap: all 18 labs aligned to the five-part structure

1.8 Summary

In this chapter, you learned the fundamental concepts of edge analytics and TinyML:

- **Edge Analytics Definition:** Edge ML processes data locally on resource-constrained devices rather than in the cloud, enabling low-latency inference, enhanced privacy, offline operation, and reduced bandwidth costs.
- **Cloud vs Edge Trade-offs:** Cloud ML offers unlimited model size and maximum accuracy but requires internet connectivity and introduces latency (100-1000ms). Edge ML provides fast response (1-100ms), privacy preservation, and offline functionality, but constrains model complexity to fit device memory.
- **Device Constraints:** Edge devices span a wide spectrum—microcontrollers like Arduino Nano 33 BLE have only 256KB RAM, while Raspberry Pi 4 provides 4-8GB. Understanding memory (both Flash and SRAM), compute power, and energy budgets is critical for successful deployment.
- **TinyML Ecosystem:** TensorFlow Lite Micro (TFLM), Edge Impulse, and PyTorch Mobile provide frameworks for training models on powerful machines and deploying to constrained

devices. The workflow involves training, converting to optimized formats (TFLite), quantizing to reduce size (float32 to int8), and deploying to target hardware.

- **Three-Tier Execution Model:** This course uses a progressive learning approach—Level 1 (Jupyter/Colab) for concept exploration, Level 2 (Simulator/Pi) for testing under edge constraints, and Level 3 (Arduino/ESP32) for real hardware deployment.
- **Development Environment:** Successfully deploying edge ML requires setting up Python environments with TensorFlow, understanding TFLite conversion, and configuring embedded development tools (Arduino IDE, ESP-IDF) for target hardware.
- **Real-World Applications:** Edge ML powers consumer electronics (voice wake words, fitness tracking), industrial IoT (predictive maintenance, quality control), and healthcare (wearable diagnostics, fall detection)—anywhere real-time, privacy-preserving, or offline ML is essential.

Looking Ahead

Now that you understand the edge ML landscape and constraints, LAB02 builds your foundation in how neural networks actually learn through gradient descent and backpropagation. This understanding is essential before tackling model optimization, quantization, and deployment in subsequent labs.

Chapter Summary

You've learned the foundations of Edge Analytics:

- Edge ML processes data locally for low latency, privacy, and reliability
- Edge devices have severe constraints: KB of RAM vs GB in cloud
- TinyML frameworks (TFLM, Edge Impulse) enable deployment on microcontrollers
- The three-tier model (Notebook → Simulator → Device) supports progressive learning

Self-Assessment Questions

Test your understanding before continuing:

1. Why would you choose edge ML over cloud ML for a security camera? Consider latency, privacy, and cost in your answer.
2. An Arduino Nano 33 BLE has 256KB of SRAM. Can you run a 200KB quantized model? What else besides model weights needs RAM during inference?
3. What does quantization do, and why does int8 quantization reduce model size by approximately 4x compared to float32?
4. Which execution tier (Level 1, 2, or 3) would you use to test if your model runs fast enough on the target hardware? Explain your reasoning.
5. What are the key trade-offs between inference latency and energy consumption on battery-powered edge devices?

1.9 Guided Experiments

Experiment 1: Hardware Comparison and Model Size Analysis

Hypothesis: Device memory constraints directly limit which models can be deployed. We expect smaller devices to require more aggressive quantization.

Procedure:

1. Research the specifications for three edge devices: Arduino Nano 33 BLE, ESP32, and Raspberry Pi 4
2. Calculate how many parameters each device can store (assuming float32 and int8 formats)
3. For a sample model with 100,000 parameters, determine which devices can run it
4. Record results in the table below

Data Collection:

Device	RAM (KB)	Max Params (float32)	Max Params (int8)	Can Run 100K Model
Arduino Nano 33 BLE	256			
ESP32	520			
Raspberry Pi 4	4,194,304			

Analysis Questions:

- What pattern do you observe in the relationship between RAM size and maximum model capacity?
- Does quantization from float32 to int8 enable deployment on devices that couldn't run the float32 version?
- For Arduino Nano 33 BLE, what percentage of RAM would a 100K parameter model consume?
- Why might the actual usable memory be less than the total RAM listed?

Experiment 2: Cloud vs Edge Latency Comparison

Hypothesis: Edge inference has lower latency than cloud inference due to elimination of network round-trip time.

Procedure:

1. Estimate typical latencies for each step in cloud-based inference:
 - Sensor reading: 1-5ms
 - WiFi transmission: 50-200ms
 - Cloud processing: 100-500ms
 - Response transmission: 50-200ms
2. Estimate typical latencies for edge-based inference:
 - Sensor reading: 1-5ms
 - Local preprocessing: 5-20ms
 - Model inference: 10-100ms
3. Calculate total end-to-end latency for best case and worst case scenarios
4. Record results in table

Data Collection:

Approach	Best Case (ms)	Worst Case (ms)
Cloud Inference		
Edge Inference		
Latency Reduction		

Analysis Questions:

- For applications requiring sub-100ms response time, which approach is viable?
- What happens to cloud inference latency when internet connectivity is poor?
- In what scenarios might cloud inference still be preferred despite higher latency?
- How does battery life factor into this comparison for mobile devices?

1.10 Troubleshooting Guide

⚠ Python Environment Not Found

Symptoms: Command `python` or `python3` not recognized, or wrong Python version installed

Cause: Python not installed, not in system PATH, or multiple conflicting Python installations

Solution:

1. Verify Python installation: `python3 --version` (should show 3.8 or higher)
2. If not installed, download from <https://www.python.org/downloads/>
3. On Windows, ensure “Add Python to PATH” is checked during installation
4. Verify installation: `which python3` (Linux/Mac) or `where python` (Windows)

⚠ TensorFlow Installation Failed

Symptoms: `pip install tensorflow` fails with compilation errors or package conflicts

Cause: Incompatible Python version, missing system dependencies, or outdated pip/setup-tools

Solution:

1. Update pip: `python3 -m pip install --upgrade pip setuptools wheel`
2. Use Python 3.8-3.11 (TensorFlow 2.x may not support 3.12+)
3. Install TensorFlow: `pip install tensorflow==2.15.0`
4. For M1/M2 Macs, use: `pip install tensorflow-macos tensorflow-metal`
5. Verify: `python3 -c "import tensorflow as tf; print(tf.__version__)"`

⚠ Jupyter Notebook Kernel Not Starting

Symptoms: Jupyter notebook opens but kernel fails to connect or shows “Kernel Dead” error

Cause: Missing kernel specification, environment mismatch, or corrupted Jupyter installation

Solution:

1. Install Jupyter in the same environment: `pip install jupyter ipykernel`
2. Register kernel: `python3 -m ipykernel install --user --name=tinyml`
3. List kernels: `jupyter kernelspec list`
4. Start Jupyter: `jupyter notebook`
5. Select “tinyml” kernel from Kernel menu in notebook

⚠ Import Error: No Module Named 'tensorflow'**Symptoms:** import tensorflow fails even after successful pip install**Cause:** TensorFlow installed in different Python environment than Jupyter is using**Solution:**

1. Check which Python Jupyter uses: Run in notebook: `import sys; print(sys.executable)`
2. Install TensorFlow in that specific Python: `/path/to/python -m pip install tensorflow`
3. Alternatively, use virtual environment:
 - Create: `python3 -m venv tinyml-env`
 - Activate: `source tinyml-env/bin/activate` (Linux/Mac) or `tinyml-env\Scripts\activate` (Windows)
 - Install packages: `pip install tensorflow jupyter`
4. Restart Jupyter kernel after installation

⚠ Arduino IDE Cannot Find Board**Symptoms:** Arduino Nano 33 BLE Sense not appearing in Tools > Port menu**Cause:** Missing USB drivers, incorrect board manager settings, or faulty USB cable**Solution:**

1. Try different USB cable (must be data cable, not charge-only)
2. Install board support: Tools > Board > Boards Manager > search "Arduino Mbed OS Nano Boards" > Install
3. On Windows: Install driver from https://docs.arduino.cc/software/ide-v1/tutorials/getting-started/cores/arduino-mbed_nano
4. Press reset button twice quickly to enter bootloader mode (orange LED should pulse)
5. Check device manager (Windows) or `ls /dev/tty*` (Linux/Mac) for new serial port

Exercises

1. **[Basic] Basic: Application Analysis:** Pick a smart home device (thermostat, doorbell, etc.). Analyze whether it uses cloud ML, edge ML, or hybrid. What are the trade-offs?
2. **[Basic] Basic: Memory Calculation:** A model has 50,000 parameters stored as float32.
 - How many bytes is that?
 - Will it fit in Arduino Nano's 256KB RAM?
 - How big would it be after int8 quantization?
3. **[Basic] Basic: Environment Setup:** Set up your Level 1 environment and run:


```
import tensorflow as tf
print(f"TensorFlow version: {tf.__version__}")
print(f"GPU available: {tf.config.list_physical_devices('GPU')}")
```

 Take a screenshot of the output.
4. **[Basic] Basic: Research:** Find one commercial product using TinyML (hint: look at smart speakers, wearables, or industrial sensors). Write a paragraph about what ML task it performs and why edge deployment makes sense.
5. **[Intermediate] Intermediate: Cloud vs Edge Latency Analysis:** Consider a real-time gesture recognition system for gaming.
 - Calculate total end-to-end latency for cloud-based inference: sensor reading (2ms) + WiFi transmission (100ms) + cloud processing (150ms) + response transmission

- (100ms) + actuator response (5ms)
 - Calculate latency for edge-based inference: sensor reading (2ms) + preprocessing (10ms) + local inference (30ms) + actuator response (5ms)
 - What is the latency reduction? Would this be acceptable for real-time gaming (target: <50ms)?
 - If WiFi latency increases to 300ms due to network congestion, how does this affect the cloud approach?
6. **[Intermediate] Intermediate: Bandwidth Savings Calculation:** A factory has 100 vibration sensors monitoring equipment, each sampling at 1kHz (1000 samples/second) with 16-bit resolution.
- Calculate daily data volume if all raw sensor data is sent to the cloud (bytes per day)
 - If edge processing compresses data to anomaly alerts (50 bytes per alert, average 10 alerts/day per sensor), calculate the new daily data volume
 - What percentage bandwidth savings does edge processing provide?
 - At \$0.10 per GB of data transfer, calculate monthly cost savings for the edge approach
 - What additional benefits beyond cost does edge processing provide in this scenario?
7. **[Challenge] Advanced: Edge Analytics Architecture Design:** You are tasked with designing an edge analytics system for a smart building with 500 occupancy sensors (motion + CO2) across 10 floors.
- Design a three-tier architecture: Sensor nodes (Tier 1), Floor gateways (Tier 2), Building server (Tier 3)
 - For each tier, specify: What device would you use (Arduino/ESP32/Pi)? What processing happens at this tier? What data is transmitted to the next tier?
 - The ML model for occupancy detection has 75,000 parameters. Where should inference run and why?
 - Create a data flow diagram showing: sensor data, inference results, aggregated metrics, and cloud sync (if any)
 - Justify your architecture choices considering: real-time requirements, privacy (office occupancy is sensitive), cost, and reliability
8. **[Challenge] Advanced: Privacy Implications Analysis:** Compare cloud vs edge processing for a health monitoring wearable that tracks heart rate, activity, and sleep patterns.
- List what personal data would be transmitted in a cloud-based approach vs edge-based approach
 - Identify potential privacy risks if raw sensor data is compromised during transmission or in cloud storage
 - Design a hybrid approach: What processing must happen on-device for privacy? What can be sent to cloud for enhanced features?
 - Consider GDPR/HIPAA requirements: What data retention policies differ between edge and cloud?
 - How does edge processing help with data sovereignty (data must stay in specific geographic regions)?
 - Propose a privacy-preserving architecture that uses federated learning to improve models without sharing raw patient data

Related Labs

Next Steps:

- **LAB02 - Neural Network Training Fundamentals:** Build your first neural networks and understand how gradient descent works. Essential foundation for all subsequent labs.

- **LAB03 - Model Optimization and Quantization:** Learn how to compress models to fit on edge devices using quantization techniques.

- **Resources.**
- *TinyML: Machine Learning with TensorFlow Lite* by Pete Warden & Daniel Situnayake
 - TensorFlow Lite documentation: <https://www.tensorflow.org/lite>
 - Edge Impulse (free tier): <https://www.edgeimpulse.com>
 - TinyML Foundation: <https://www.tinyml.org>
 - Arduino ML tutorials: <https://docs.arduino.cc/tutorials/nano-33-ble-sense>

— **Credits.** This chapter draws on materials from the TinyML courseware by Pete Warden and Daniel Situnayake, as well as the Edge Impulse documentation.



0101001010100101010
1001010101001011101
0100101010010101001
0101001010100101010

1001

Label = WALKING

1010100101001010101
0101010010010010001
0010011111010101111
1010100100111101011

1111

Label = RUNNING

1001010011111010101
1101010111010101110
1010101111010101011
1111110001111010101

1011

Label = BIKING

1111111111010011101
0011111010111110101
0101110101010101110
10101010100111110

1110

Label = GOLFING

2. Neural Network Training Fundamentals •

Learning Objectives

After completing this lab, you will be able to:

- Understand what loss functions measure and why they matter
- Implement gradient descent to minimize loss step-by-step
- Build and train simple neural networks with TensorFlow/Keras
- Distinguish between regression and classification tasks
- Understand convolutions and their application in image processing
- Design and train Convolutional Neural Networks (CNNs)

Prerequisites

Before starting this lab, ensure you:

- Completed LAB01 (Edge Analytics Fundamentals and TinyML)
- Basic Python programming (variables, loops, functions)
- Understanding of arrays/lists and basic NumPy operations
- High school mathematics: algebra and basic calculus (derivatives)

⚠ Forgetting to Normalize Inputs

One of the most common causes of training failure is not normalizing your input data. If your pixel values are 0-255 instead of 0-1, the gradients will be $255 \times$ larger than expected, causing training instability or divergence. **Always normalize:** $x = x / 255.0$

Interactive Notebook**Jupyter Notebook:** LAB02_ml_foundations.ipynb

Open in Google Colab | View on GitHub | Companion Website

Three-Tier Execution Model**Tier 1: Jupyter/Colab**

Implement gradient descent from scratch, train neural networks with TensorFlow/Keras, visualize loss curves, experiment with hyperparameters

Tier 2: Simulator
Test trained models with TFLite interpreter, measure inference speed, validate accuracy on edge-representative data**Tier 3: Arduino/ESP32**
Deploy trained models to microcontrollers, understand memory constraints for different network architectures

Before we can deploy machine learning models on edge devices, we need to understand how these models learn. This chapter builds your foundation in neural network training—the process of teaching a model to make accurate predictions.

The Central Question

How does a computer learn to predict outcomes it has never seen before? The answer involves three key concepts: **loss** (measuring mistakes), **gradients** (finding the direction to improve), and **optimization** (iteratively getting better).

We'll start with the simplest possible example—a straight line—and gradually build up to neural networks that can recognize images. By the end, you'll understand exactly what happens inside `model.fit()`.

Level 1: All code runs in Jupyter

Every example in this chapter works in the companion notebook. No special hardware required—just Python and TensorFlow.

2.1 Exploring Loss Functions

2.1.1 What is Loss?

Imagine you're learning to throw darts at a target. Each throw, you might hit close to the bullseye or far from it. To improve, you need to *measure* how far off you were. In machine learning, this measurement is called the **loss** (also called error or cost).

Loss = Measure of Mistakes

Loss is a single number that tells us how wrong our model's predictions are. **Lower loss = better predictions.** Our entire goal in training is to minimize this number.

2.1.2 A Simple Example: Predicting a Line

Let's start with the simplest possible prediction task: given some X values, predict Y values that follow a straight line.

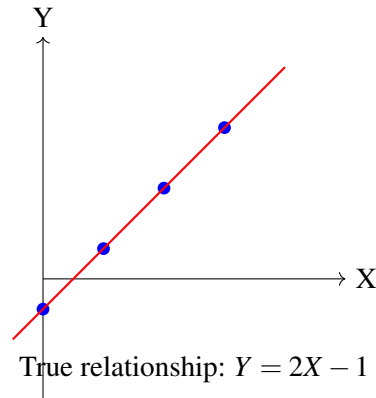


Figure 2.1: Our target: find the line that passes through all blue points

The true relationship is $Y = 2X - 1$. But pretend we don't know this—our job is to *discover* it by guessing values for w (slope) and b (intercept).

2.1.3 Implementing Loss Calculation

Let's implement this step-by-step. First, we set up our known data points:

```
1 import math
2
3 # Our known data points (the "ground truth")
4 x = [-1, 0, 1, 2, 3, 4]
5 y = [-3, -1, 1, 3, 5, 7] # These follow Y = 2X - 1
```

Understanding the Data

- **Line 4:** Six X values from -1 to 4
- **Line 5:** The corresponding Y values. Notice: when $X=0$, $Y=-1$; when $X=1$, $Y=1$, etc. These are our “training examples”—the data our model learns from.

Now let's make predictions using a guess for w and b :

```
7 # Our GUESS at the relationship (try changing these!)
8 w = 3 # We guess the slope is 3
9 b = -1 # We guess the intercept is -1
10
11 # Calculate predictions: Y = wX + b
12 myY = []
13 for thisX in x:
14     thisY = (w * thisX) + b
15     myY.append(thisY)
16
17 print("Real Y is " + str(y))
18 print("My Y is " + str(myY))
```

Expected Output

Real Y is [-3, -1, 1, 3, 5, 7]
 My Y is [-4, -1, 2, 5, 8, 11]

Notice our predictions are close but not perfect. With $w = 3, b = -1$, we overshoot on most values.

Common Confusion

Students often confuse y (the true values we're trying to match) with myY (our predictions). Always keep these separate! Loss compares them.

2.1.4 Calculating the Loss

Now the key step: how do we turn “our predictions are off” into a single number?

```

19 # Calculate squared error for each point
20 total_square_error = 0
21 for i in range(len(y)):
22     square_error = (y[i] - myY[i]) ** 2
23     total_square_error += square_error
24     print(f"Point {i}: true={y[i]}, pred={myY[i]}, error={square_error}")
25
26 # Final loss: root mean square error
27 loss = math.sqrt(total_square_error)
28 print(f"\nTotal loss: {loss}")

```

Line-by-Line Breakdown

- **Line 22:** For each point, calculate $(\text{true} - \text{predicted})^2$
- **Why square?** Squaring makes all errors positive and penalizes big errors more
- **Line 27:** We take the square root at the end (RMSE = Root Mean Square Error)

Expected Output

```

Point 0: true=-3, pred=-4, error=1
Point 1: true=-1, pred=-1, error=0
Point 2: true=1, pred=2, error=1
Point 3: true=3, pred=5, error=4
Point 4: true=5, pred=8, error=9
Point 5: true=7, pred=11, error=16

Total loss: 5.57

```

Experiment: Find Better Parameters

Modify w and b to reduce the loss:

1. Try $w = 2, b = -1$. What's the loss now?
2. Try $w = 2, b = 0$. Better or worse?
3. Can you get the loss to exactly 0? What values work?

Hint: The true relationship is $Y = 2X - 1$

Check Your Understanding

Before moving on, make sure you can answer:

1. What does a loss of 0 mean?
2. Why do we square the errors before summing?
3. If two models have loss 2.0 and 8.0, which is better?

💡 Experiment: What Happens with Extreme Values?

Try this experiment in the notebook:

1. Set `learning_rate = 1.0` (very high). What happens to the loss over time? Does it converge or oscillate wildly?
2. Set `learning_rate = 0.0001` (very low). Does training eventually work? How many iterations does it take?

Key insight: Learning rate is like step size. Too big = overshoot the minimum. Too small = takes forever to arrive. Finding the right balance is crucial.

2.2 Minimizing Loss with Gradient Descent

In the previous section, you manually adjusted w and b to find good values. But imagine a model with *millions* of parameters—we can't tune them by hand! We need an **automatic** way to find the values that minimize loss.

Gradient Descent

Gradient descent is an algorithm that automatically adjusts parameters to reduce loss. It works like walking downhill in fog: you can't see the bottom, but you can feel which direction is “down” and take small steps that way.

2.2.1 The Gradient: Which Way is Downhill?

The **gradient** tells us how the loss changes when we adjust each parameter:

- If gradient is positive: increasing this parameter increases loss (bad!)
- If gradient is negative: increasing this parameter decreases loss (good!)
- If gradient is zero: we're at a minimum (or maximum)

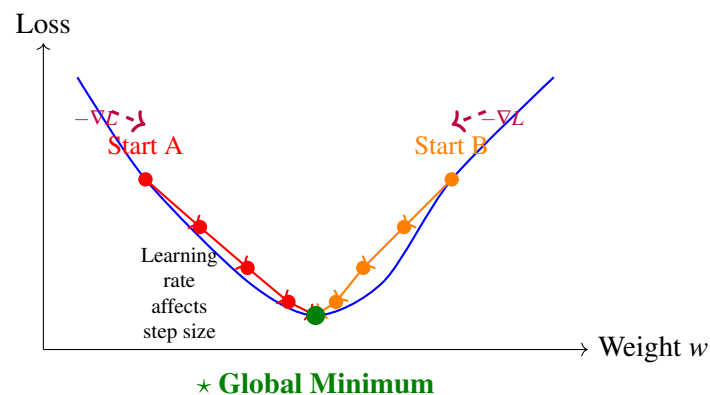


Figure 2.2: Gradient descent optimization: starting from different points (red or orange), the algorithm follows the negative gradient to reach the global minimum. Step size controlled by learning rate.

2.2.2 Implementing Gradient Descent

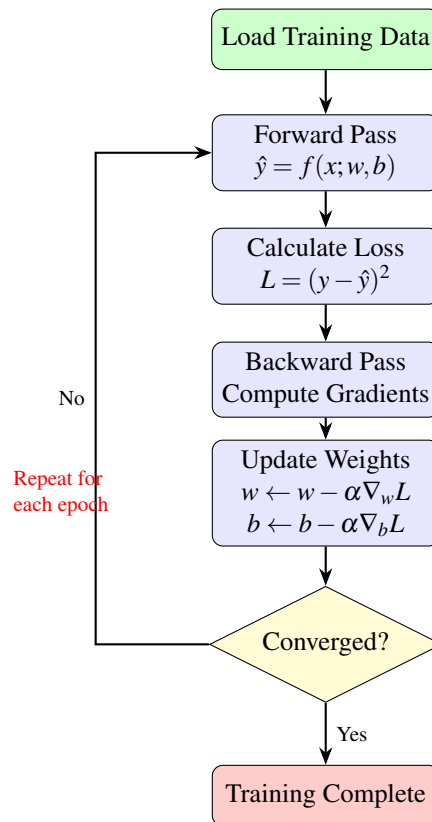


Figure 2.3: Training loop: data flows forward to compute loss, then gradients flow backward to update weights. This cycle repeats until convergence.

TensorFlow's `GradientTape` automatically calculates gradients for us. Let's see it in action:

```

1 import tensorflow as tf
2 import numpy as np
3
4 # Our training data
5 x_train = np.array([-1, 0, 1, 2, 3, 4], dtype=float)
6 y_train = np.array([-3, -1, 1, 3, 5, 7], dtype=float)

```

Code Walkthrough

We use NumPy arrays instead of Python lists because TensorFlow works more efficiently with them. The `dtype=float` ensures numerical precision.

Now we create **trainable variables**—these are the parameters TensorFlow will adjust:

```

8 # Initialize w and b with random guesses
9 w = tf.Variable(0.0) # Start with w=0
10 b = tf.Variable(0.0) # Start with b=0
11
12 # Learning rate: how big are our steps?
13 learning_rate = 0.01

```

Learning Rate Matters!

If learning rate is too high, we'll overshoot the minimum and bounce around. Too low, and training takes forever. 0.01 is a common starting point.

Now the training loop:

```

15 # Training loop: repeat 500 times
16 for epoch in range(500):
17     with tf.GradientTape() as tape:
18         # Forward pass: make predictions
19         y_pred = w * x_train + b
20
21         # Calculate loss (mean squared error)
22         loss = tf.reduce_mean((y_train - y_pred) ** 2)
23
24         # Calculate gradients: how should we adjust w and b?
25         gradients = tape.gradient(loss, [w, b])
26
27         # Update parameters: take a step downhill
28         w.assign_sub(learning_rate * gradients[0])
29         b.assign_sub(learning_rate * gradients[1])
30
31     # Print progress every 100 epochs
32     if epoch % 100 == 0:
33         print(f"Epoch {epoch}: w={w.numpy():.3f}, b={b.numpy():.3f}, loss={loss.numpy():.4f}")

```

The Training Loop Explained

- **Line 17:** GradientTape records operations so it can calculate derivatives
- **Line 19:** Forward pass—use current w , b to predict Y
- **Line 22:** Calculate how wrong we are (loss)
- **Line 25:** Automatically compute $\frac{\partial \text{loss}}{\partial w}$ and $\frac{\partial \text{loss}}{\partial b}$
- **Lines 28-29:** Update rule: $w_{\text{new}} = w_{\text{old}} - lr \times \text{gradient}$

Expected Output

```

Epoch 0: w=0.140, b=0.060, loss=18.6667
Epoch 100: w=1.876, b=-0.813, loss=0.0347
Epoch 200: w=1.969, b=-0.940, loss=0.0036
Epoch 300: w=1.990, b=-0.981, loss=0.0004
Epoch 400: w=1.997, b=-0.994, loss=0.0000

```

Watch how w approaches 2 and b approaches -1 (the true values)!

Check Your Understanding

1. What would happen if we set `learning_rate = 0`?

2. Why does the loss decrease with each epoch?
3. What do you think happens if we run for 1000 epochs instead of 500?

Experiment: Learning Rate Effects

Modify the code to try different learning rates:

- `learning_rate = 0.1` — Does it converge faster?
- `learning_rate = 0.5` — What goes wrong?
- `learning_rate = 0.001` — How many epochs to converge?

2.3 Your First Neural Network

Gradient descent with a linear model is powerful, but it can only learn straight-line relationships. Neural networks add **non-linearity**, letting them learn curves, patterns, and complex relationships.

What Makes It “Neural”?

A neural network is just:

1. Multiple linear operations ($y = wx + b$)
2. Connected together in layers
3. With non-linear **activation functions** between them

That’s it! The “magic” comes from stacking these simple operations.

2.3.1 The Simplest Possible Neural Network

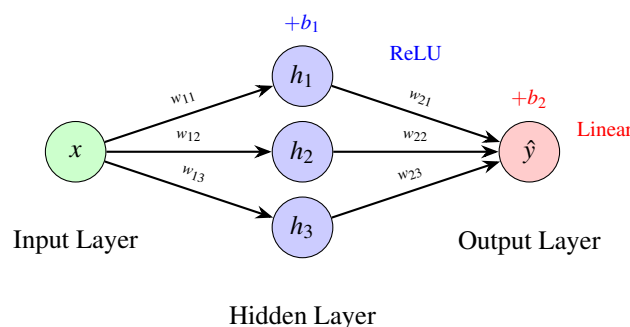


Figure 2.4: Neural network architecture: input x flows through weighted connections (w), neurons add biases (b), and activation functions introduce non-linearity

Let’s build a network with just ONE neuron to solve our $Y = 2X - 1$ problem:

```
1 import tensorflow as tf
2 from tensorflow import keras
3
4 # Define a network with 1 layer containing 1 neuron
5 model = keras.Sequential([
6     keras.layers.Dense(units=1, input_shape=[1])
7 ])
8
9 # Configure how it learns
10 model.compile(optimizer='sgd', loss='mean_squared_error')
```

Code Walkthrough

- **Line 5:** Sequential means layers stack one after another
- **Line 6:** Dense(units=1) = one neuron that computes $y = wx + b$
- **Line 6:** input_shape=[1] = we give it one number (X)
- **Line 10:** sgd = Stochastic Gradient Descent (what we coded above!)

Now train it:

```
12 # Training data
13 xs = np.array([-1, 0, 1, 2, 3, 4], dtype=float)
```

```
14 ys = np.array([-3, -1, 1, 3, 5, 7], dtype=float)
15
16 # Train for 500 epochs
17 model.fit(xs, ys, epochs=500, verbose=0)
18
19 # Make a prediction!
20 print(model.predict([10.0]))
```

Expected Output

```
[[18.999998]]
```

The correct answer is $2(10) - 1 = 19$. Our model predicts 18.999998—almost perfect!

Why Not Exactly 19?

Neural networks find *approximate* solutions. With more training or better optimization, we could get closer, but floating-point precision and random initialization mean we rarely hit exact values.

2.3.2 What Did the Network Learn?

Let's peek inside and see the learned w and b :

```
21 # Extract the learned parameters
22 weights, biases = model.layers[0].get_weights()
23 print(f"Learned weight (w): {weights[0][0]:.4f}")
24 print(f"Learned bias (b): {biases[0]:.4f}")
```

Expected Output

```
Learned weight (w): 1.9999
```

```
Learned bias (b): -0.9998
```

The network discovered $w \approx 2$ and $b \approx -1$ —exactly what we expected!

Reflection Questions

1. Why do we need `model.compile()` before `model.fit()`?
2. What's the difference between our manual gradient descent and `model.fit()`?
3. How would you modify this to learn $Y = 3X + 2$ instead?

2.4 Multi-Layer Neural Networks

One neuron can only learn linear relationships. To learn curves and complex patterns, we add more neurons and **hidden layers**.

Hidden Layers Add Power

Each hidden layer lets the network learn more complex features. One layer = lines. Two layers = curves. Three+ layers = almost any function.

```

1 # A network with one hidden layer
2 model = keras.Sequential([
3     keras.layers.Dense(units=16, activation='relu', input_shape=[1]),
4     keras.layers.Dense(units=1)
5 ])
6
7 model.compile(optimizer='adam', loss='mean_squared_error')
8 model.fit(xs, ys, epochs=500, verbose=0)

```

Code Walkthrough

- **Line 3:** Hidden layer with 16 neurons and ReLU activation
- **Line 4:** Output layer with 1 neuron (our prediction)
- **Line 7:** Adam optimizer (smarter than basic SGD)

The ReLU activation function introduces non-linearity: $ReLU(x) = \max(0, x)$

Table 2.1: Common Activation Functions Comparison

Function	Formula	Range	Pros	Cons
ReLU	$f(x) = \max(0, x)$	$[0, \infty)$	Fast computation, no vanishing gradient for $x > 0$	Dead neurons (always outputs 0)
Sigmoid	$f(x) = \frac{1}{1+e^{-x}}$	$(0, 1)$	Smooth, probabilistic output	Vanishing gradient problem, not zero-centered
Tanh	$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	$(-1, 1)$	Zero-centered, smooth	Vanishing gradient for large $ x $
Leaky ReLU	$f(x) = \max(\alpha x, x)$	$(-\infty, \infty)$	Fixes dead neurons	Extra hyperparameter α

Choosing Activation Functions

- **Hidden layers:** Use ReLU or Leaky ReLU (fast, effective)
- **Binary classification output:** Use Sigmoid (outputs probability 0-1)
- **Multi-class output:** Use Softmax (probabilities sum to 1)
- **Regression output:** Use Linear/None (unrestricted range)

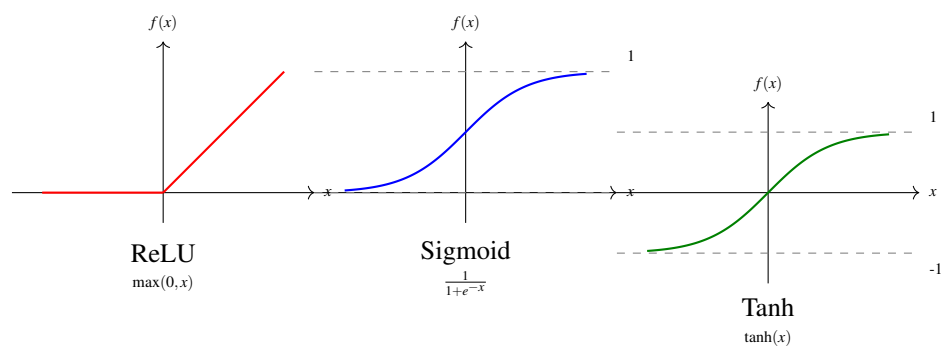


Figure 2.5: Activation function shapes: ReLU creates a linear ramp, Sigmoid outputs probabilities (0-1), Tanh is zero-centered (-1 to 1)

Experiment: Network Architecture

Try these modifications and observe training:

- Change units=16 to units=4. Faster or slower?
- Change activation='relu' to activation='tanh'
- Add another hidden layer with 8 units

2.5 From Regression to Classification

So far we've predicted continuous numbers (regression). But what if we want to predict categories, like "cat" or "dog"? This is **classification**.

Classification vs Regression

- **Regression:** Predict a number (price, temperature, score)
- **Classification:** Predict a category (spam/not-spam, digit 0-9)

The key difference: classification outputs **probabilities** for each class.

Table 2.2: Common Loss Functions for Different Tasks

Loss Function	Formula	Use Case	When to Use
Mean Squared Error (MSE)	$\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$	Regression	Predicting continuous values (temperature, price)
Mean Absolute Error (MAE)	$\frac{1}{n} \sum_{i=1}^n y_i - \hat{y}_i $	Regression	Less sensitive to outliers than MSE
Binary Cross-Entropy	$-\frac{1}{n} \sum (y \log(\hat{y}) + (1 - y) \log(1 - \hat{y}))$	Binary Classification	Two classes (spam/not-spam, yes/no)
Categorical Cross-Entropy	$-\sum_{i=1}^C y_i \log(\hat{y}_i)$	Multi-class Classification	Multiple exclusive classes (digits 0-9)
Sparse Categorical Cross-Entropy	Same as above	Multi-class Classification	Integer labels instead of one-hot vectors

Matching Loss to Task

Using the wrong loss function leads to poor training:

- MSE for classification → model doesn't learn probabilities properly
- Cross-entropy for regression → undefined behavior
- Always match: **regression = MSE/MAE**, **classification = cross-entropy**

Let's classify handwritten digits using the famous MNIST dataset:

```

1 # Load the MNIST dataset
2 mnist = keras.datasets.mnist
3 (train_images, train_labels), (test_images, test_labels) = mnist.load_data()
4
5 # Normalize pixel values to 0-1 range
6 train_images = train_images / 255.0
7 test_images = test_images / 255.0

```

Code Walkthrough

- **Line 3:** MNIST has 60,000 training images and 10,000 test images
- **Lines 6-7:** Pixels are 0-255; dividing by 255 scales them to 0-1

Always Normalize!

Neural networks train much better when inputs are in a small range (typically 0-1 or -1 to 1). Forgetting to normalize is a common cause of training failures.

```
9 # Build a classification model
10 model = keras.Sequential([
11     keras.layers.Flatten(input_shape=(28, 28)), # 28x28 image -> 784 values
12     keras.layers.Dense(128, activation='relu'), # Hidden layer
13     keras.layers.Dense(10, activation='softmax') # 10 classes (digits 0-9)
14 ])
15
16 model.compile(optimizer='adam',
17               loss='sparse_categorical_crossentropy',
18               metrics=['accuracy'])
```

Code Walkthrough

- **Line 11:** Flatten converts 2D image to 1D list of pixels
- **Line 13:** softmax outputs probabilities that sum to 1
- **Line 16:** sparse_categorical_crossentropy is the loss for classification
- **Line 17:** We track accuracy as an extra metric

```
19 # Train the model
20 model.fit(train_images, train_labels, epochs=5)
21
22 # Evaluate on test data
23 test_loss, test_acc = model.evaluate(test_images, test_labels)
24 print(f"Test accuracy: {test_acc:.4f}")
```

Expected Output

Epoch 1/5 - loss: 0.2592 - accuracy: 0.9253

Epoch 2/5 - loss: 0.1141 - accuracy: 0.9660

...

Test accuracy: 0.9752

97.5% accuracy on handwritten digits—not bad for a few lines of code!

2.6 Summary

In this chapter, you built a comprehensive foundation in neural network training:

- **Loss Functions:** Loss quantifies how wrong our predictions are. Mean squared error (MSE) for regression and categorical cross-entropy for classification provide differentiable objectives that gradient descent can optimize. Lower loss indicates better model performance.
- **Gradient Descent:** This optimization algorithm iteratively adjusts model parameters by computing gradients (derivatives of loss with respect to parameters) and taking steps in the direction that reduces loss. The learning rate controls step size—too high causes divergence, too low slows convergence.

- **Backpropagation:** TensorFlow's `GradientTape` automatically computes gradients through computational graphs, enabling efficient training of deep networks. This automates the calculus required to update millions of parameters simultaneously.
- **Neural Network Architecture:** Networks stack layers of neurons with learnable weights and biases. Dense (fully connected) layers apply linear transformations followed by non-linear activation functions (ReLU, softmax). Multiple layers enable learning complex, hierarchical representations.
- **Training Loop Mechanics:** The standard training loop performs forward passes (computing predictions), calculates loss, computes gradients via backpropagation, and updates weights. Repeating this process for many epochs gradually improves model performance.
- **Classification vs Regression:** Regression predicts continuous values using linear outputs and MSE loss. Classification predicts discrete categories using softmax activation and categorical cross-entropy loss, outputting class probabilities.
- **Practical Implementation:** Keras provides high-level APIs (`Sequential`, `Dense`, `model.compile`, `model.fit`) that abstract low-level tensor operations. Understanding what happens inside these APIs helps debug training issues and design better architectures.

Looking Ahead

You now understand how neural networks learn from data. In LAB03, you'll learn how to compress these trained models for edge deployment using quantization, pruning, and TensorFlow Lite conversion—essential techniques for fitting models into the tight memory constraints of microcontrollers.

Chapter Summary

You've learned the core concepts of neural network training:

- **Loss functions** measure prediction errors
- **Gradient descent** automatically finds better parameters
- **Neural networks** stack layers to learn complex patterns
- **Classification** predicts categories using softmax

Next chapter: We'll learn how to make these models small enough for microcontrollers!

Self-Assessment Questions

Test your understanding before continuing:

1. Why do we square the errors in mean squared error (MSE) instead of just taking the absolute values? What effect does this have on outliers?
2. Explain what happens during backpropagation when the learning rate is too high versus too low. How would you identify each problem from the training loss curve?
3. A dense layer with 128 input neurons and 64 output neurons has how many parameters (weights and biases)? How much memory does this require in float32 vs int8?
4. What is the purpose of the softmax activation function in classification tasks? Why can't we just use the raw output values?
5. Why must you normalize input data (e.g., dividing pixel values by 255) before training? What happens if you forget this step?

2.7 Guided Experiments

Experiment 1: Learning Rate Sweep

Hypothesis: Learning rate significantly affects convergence speed and stability. We expect very low learning rates to converge slowly, very high rates to diverge, and an optimal middle ground.

Procedure:

1. Train a simple linear model ($Y = 2X - 1$) with different learning rates
2. Use the gradient descent code from Section 2.2
3. Test learning rates: 0.001, 0.01, 0.1, 0.5, 1.0
4. For each learning rate, train for 200 epochs
5. Record final loss and whether the model converged

Data Collection:

Learning Rate	Final Loss	Converged?	Epochs to Converge
0.001			
0.01			
0.1			
0.5			
1.0			

Analysis Questions:

- At what learning rate does the model fail to converge (diverge)?
- Which learning rate achieves the lowest final loss fastest?
- Plot loss vs. epoch for all learning rates on the same graph—what pattern emerges?
- Why does a learning rate of 1.0 cause instability for this problem?

Experiment 2: Batch Size Comparison

Hypothesis: Batch size affects training speed and model generalization. Larger batches train faster per epoch but may converge to worse local minima.

Procedure:

1. Use the MNIST classification model from Section 5
2. Train with different batch sizes: 16, 32, 64, 128, 256
3. Train for 5 epochs with each batch size
4. Measure training time per epoch and final test accuracy
5. Record results in table

Data Collection:

Batch Size	Time/Epoch (s)	Final Train Acc	Final Test Acc
16			
32			
64			
128			
256			

Analysis Questions:

- Does larger batch size always train faster per epoch?
- Which batch size achieves the best test accuracy?
- Is there a trade-off between training speed and final accuracy?

- For edge deployment where model size matters, which batch size during training would you choose?

2.8 Troubleshooting Guide

⚠ Training Loss Not Decreasing

Symptoms: Loss remains constant or increases during training, model accuracy stays at random guessing level

Cause: Learning rate too low, incorrect loss function, data not normalized, or model architecture too simple

Solution:

1. Check learning rate: Try values between 0.001 and 0.1
2. Normalize inputs: $X = X / 255.0$ for images, or use `StandardScaler` for other data
3. Verify loss function: Use `binary_crossentropy` for 2-class, `categorical_crossentropy` for multi-class
4. Check labels: Ensure labels match expected format (0/1 for binary, one-hot for categorical)
5. Increase model capacity: Add more neurons or layers if underfitting

⚠ Loss Becomes NaN or Infinity

Symptoms: Training starts normally but loss suddenly becomes nan or inf after a few epochs

Cause: Learning rate too high causing gradient explosion, numerical instability, or division by zero

Solution:

1. Reduce learning rate by 10x: Try 0.001 instead of 0.01
2. Add gradient clipping: `optimizer = Adam(clipnorm=1.0)`
3. Check for extreme values: Print min/max of inputs and weights
4. Verify data has no NaN values: `np.isnan(X).any()`
5. Use batch normalization layers to stabilize training

⚠ Model Overfits Training Data

Symptoms: Training accuracy high (>95%) but validation accuracy much lower (<70%), loss curves diverge

Cause: Model too complex for dataset size, training too long, or insufficient regularization

Solution:

1. Add dropout: `layers.Dropout(0.3)` after dense layers
2. Use L2 regularization: `layers.Dense(64, kernel_regularizer=l2(0.01))`
3. Increase training data: Use data augmentation or collect more samples
4. Reduce model complexity: Fewer layers or neurons
5. Implement early stopping: `callbacks = [EarlyStopping(patience=5, restore_best_weights=True)]`

⚠️ CUDA Out of Memory Error

Symptoms: `ResourceExhaustedError`: OOM when allocating tensor during training on GPU

Cause: Batch size too large for GPU memory, or previous training session didn't release memory

Solution:

1. Reduce batch size: Try 32 instead of 128, or 16 for very large models
2. Clear GPU memory: Restart Jupyter kernel or add `tf.keras.backend.clear_session()`
3. Use mixed precision: `tf.keras.mixed_precision.set_global_policy('mixed_float16')`
4. Enable memory growth: `gpus = tf.config.list_physical_devices('GPU')`
`tf.config.experimental.set_memory_growth(gpus[0], True)`
5. Use CPU if GPU insufficient: with `tf.device('/CPU:0')`:

⚠️ Convolution Input Shape Mismatch

Symptoms: `ValueError`: Input 0 is incompatible with layer conv2d or dimension mismatch errors

Cause: Input data shape doesn't match expected format (batch, height, width, channels)

Solution:

1. Check input shape: Print `X.shape` - should be (num_samples, height, width, channels)
2. Add channel dimension for grayscale: `X = X.reshape(-1, 28, 28, 1)`
3. Verify model expects correct input: First layer should be Input (shape=(28, 28, 1))
4. For RGB images: Ensure shape is (height, width, 3) not (3, height, width)
5. Use `model.summary()` to verify expected input shape

🔗 Related Labs

Prerequisites:

- **LAB01 - Edge Analytics Fundamentals:** Introduces the edge ML ecosystem, three-tier execution model, and development environment setup.

Next Steps:

- **LAB03 - Model Optimization and Quantization:** Apply the neural networks you learned here and compress them for edge deployment using TensorFlow Lite and quantization.
- **LAB07 - Convolutional Neural Networks:** Extend dense networks to CNNs for image processing tasks, building on the training fundamentals covered here.

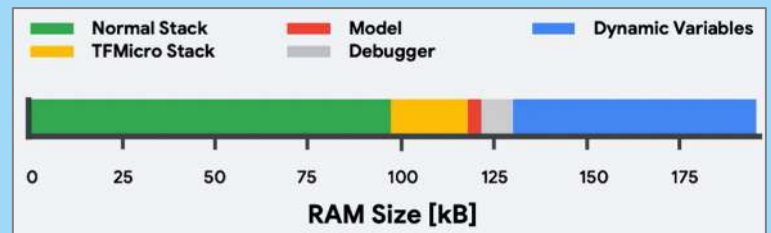
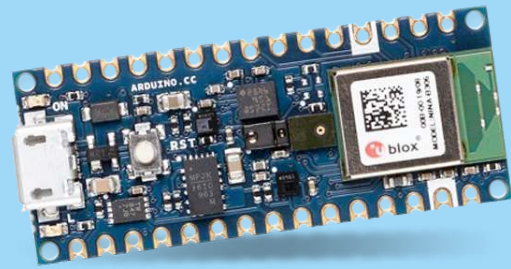
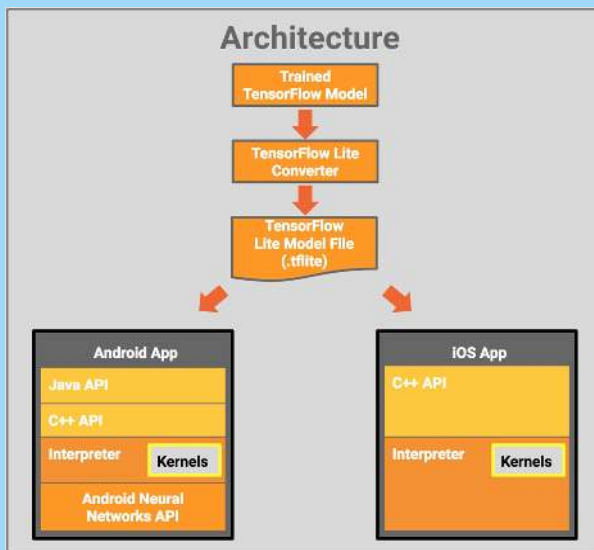
- **Further Reading.**
- *Deep Learning* by Ian Goodfellow, Yoshua Bengio, and Aaron Courville - Chapter 6 (Deep Feedforward Networks) and Chapter 8 (Optimization for Training Deep Models): <https://www.deeplearningbook.org/>
 - *Neural Networks and Deep Learning* by Michael Nielsen - Free online book with interactive visualizations: <http://neuralnetworksanddeeplearning.com/>
 - 3Blue1Brown Neural Network Series - Excellent visual explanations of backpropagation and gradient descent: https://www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi

- Andrew Ng's Machine Learning Course (Coursera) - Neural Networks and Backpropagation lectures: <https://www.coursera.org/learn/machine-learning>
- CS231n: Convolutional Neural Networks for Visual Recognition (Stanford) - Lecture notes on optimization and neural networks: <http://cs231n.github.io/>
- *Gradient Descent Optimization Algorithms* by Sebastian Ruder - Comprehensive overview of optimization methods: <https://arxiv.org/abs/1609.04747>
- TensorFlow Tutorials - Getting started with Keras and neural networks: <https://www.tensorflow.org/tutorials/quickstart/beginner>
- Distill.pub - Interactive articles on neural network concepts and training dynamics: <https://distill.pub/>

Exercises

1. **[Basic] Basic: Loss Calculation:** Given $y_{true} = [1, 2, 3]$ and $y_{pred} = [1.5, 2.5, 2.5]$, calculate the MSE by hand. Then verify with code.
2. **[Intermediate] Intermediate: Learning Rate:** Modify the gradient descent code to plot loss vs. epoch for learning rates of 0.001, 0.01, and 0.1. Which converges fastest without diverging?
3. **[Intermediate] Intermediate: Network Architecture:** For the MNIST classifier, try:
 - Removing the hidden layer (only Flatten + Dense(10))
 - Adding a second hidden layer with 64 unitsCompare test accuracy for each.
4. **[Basic] Basic: Edge Computing Connection:** A microcontroller has 256KB of flash memory. If each parameter uses 4 bytes (float32), how many parameters can fit? What if we use 1 byte per parameter (int8)?

— **CREDITS.** Adapted from TinyML courseware by Pete Warden and Daniel Situnayake. We have taken this code segment from the online course: *Edx Tiny Machine Learning (2021-22)*. Since we may have updated the code to reflect any changes made to the underlying frameworks where necessary. Credits should go to the original authors of this tutorial.



3. Model Optimization and Quantization for Edge

Learning Objectives

After completing this lab, you will be able to:

- Explain why quantization is essential for edge deployment (memory and speed)
- Convert TensorFlow models to TensorFlow Lite format using the TFLite Converter
- Apply post-training quantization to reduce model size by up to 4x
- Compare Float32, dynamic range, and full integer (Int8) quantization
- Implement quantization-aware training (QAT) for minimal accuracy loss
- Deploy quantized models across all three execution tiers

Interactive Notebook

Jupyter Notebook: LAB03_tflite_quantization.ipynb

Open in Google Colab | View on GitHub | Companion Website

Machine learning models trained on powerful workstations must be dramatically compressed before they can run on edge devices. A typical neural network uses 32-bit floating-point numbers for its weights and activations—but a microcontroller might only have 256KB of memory total. This chapter teaches you how to shrink models by up to 4x while preserving most of their accuracy.

Quantization is the process of reducing the precision of a model's weights and activations from floating-point (typically 32-bit) to lower-precision integers (typically 8-bit). This isn't just about saving storage—integer operations are faster and more energy-efficient on edge hardware, making quantization essential for real-time inference on microcontrollers.

By the end of this lab, you'll understand when to use different quantization techniques, how to implement them with TensorFlow Lite, and how to measure their impact on your model's accuracy and size.

Prerequisites

Before starting this lab, ensure you:

- Completed LAB02 (Neural Network Training Fundamentals)
- Understanding of neural network weights and activations
- Familiarity with binary/hexadecimal number representations
- TensorFlow 2.10+ installed (for Level 1)

Using Too Few Representative Samples

Post-training quantization requires a “representative dataset” to calibrate scale and zero-point values. Using too few samples (e.g., 10 instead of 100-200) can lead to poor calibration and significant accuracy loss. **Rule of thumb:** Use at least 100 diverse samples that cover the range of inputs your model will see in deployment.

Hardware Requirements by Level

- **Level 1:** Laptop with Python 3.8+, TensorFlow 2.10+
- **Level 2:** Raspberry Pi 4 with tfLite-runtime, or host PC for simulation
- **Level 3:** Arduino Nano 33 BLE Sense, ESP32, or Coral Edge TPU

Three-Tier Execution Model

Tier 1: Jupyter/Colab

Convert TensorFlow models to TFLite, apply post-training quantization (dynamic range and full int8), implement quantization-aware training, compare accuracy vs model size **Tier 2: Raspberry Pi**

Deploy quantized TFLite models with tfLite-runtime, measure inference latency for float32 vs int8, validate accuracy on real data **Tier 3: Arduino/ESP32**

Run int8 models on microcontrollers with TensorFlow Lite Micro, measure Flash and SRAM usage, profile inference speed and power consumption

3.1 Why Quantization Matters for Edge AI

Before diving into code, let’s understand *why* quantization is so important for edge deployment.

The Edge Memory Problem

A simple MNIST classifier with two hidden layers has approximately 100,000 weights. At 32 bits per weight, that’s 400KB—already too large for many microcontrollers. Quantization to 8-bit reduces this to 100KB, making deployment possible.

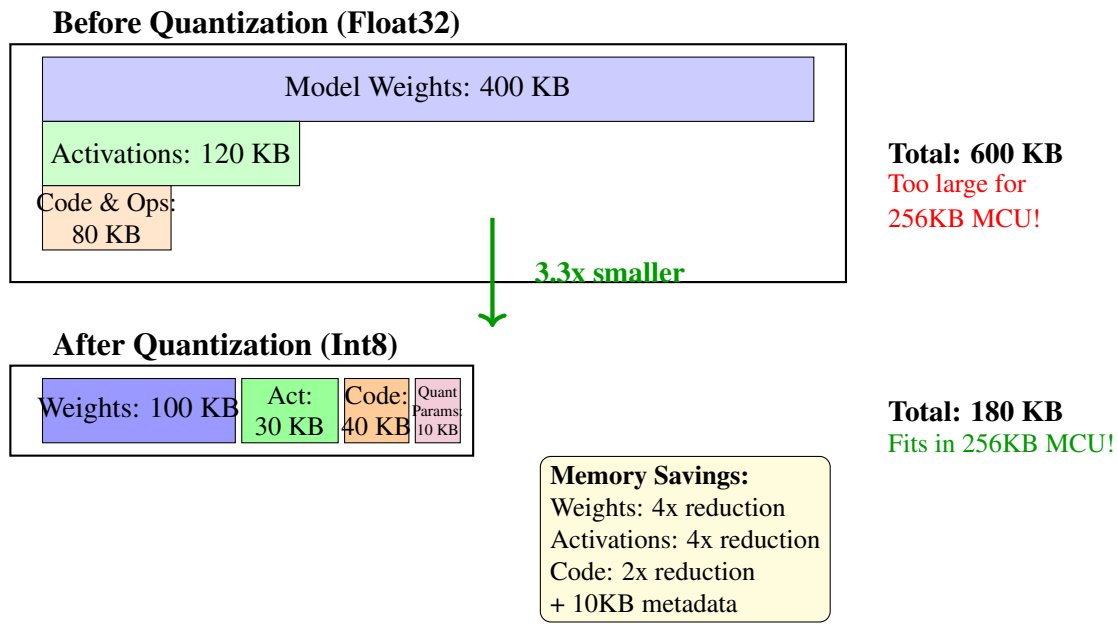


Figure 3.1: Memory layout comparison before and after quantization for a typical MNIST classifier. Quantization reduces total memory from 600KB to 180KB, enabling deployment on microcontrollers with only 256KB RAM.

3.1.1 Understanding Number Representation

Computers represent numbers differently depending on precision requirements. The diagram below shows how the same conceptual value can be stored in different formats:

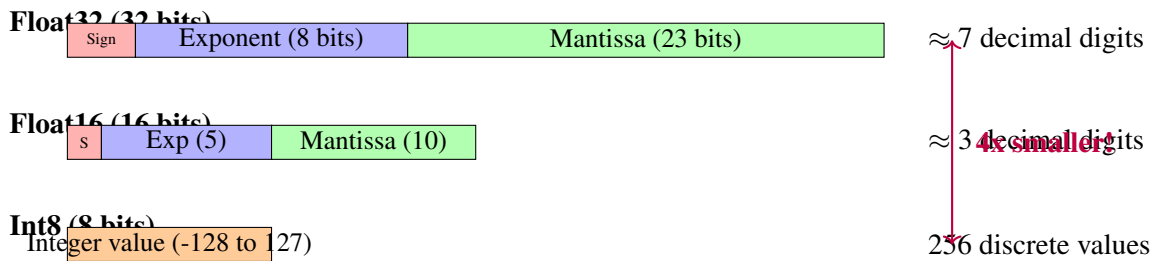


Figure 3.2: Comparison of number representations. Quantization converts Float32 to Int8, reducing memory by 4x while representing values as discrete integers.

3.1.2 The Quantization Formula

When we quantize a floating-point value to an 8-bit integer, we use a **scale** and **zero-point**:

$$\text{int8\_value} = \text{round} \left(\frac{\text{float\_value}}{\text{scale}} \right) + \text{zero\_point} \quad (3.1)$$

To reconstruct the approximate floating-point value:

$$\text{float\_reconstructed} = \text{scale} \times (\text{int8\_value} - \text{zero\_point}) \quad (3.2)$$

Understanding Scale and Zero-Point

- **Scale:** Determines the step size between adjacent integer values. Smaller scale = finer granularity but smaller range.
- **Zero-Point:** The integer value that represents floating-point zero. Important for asymmetric distributions.

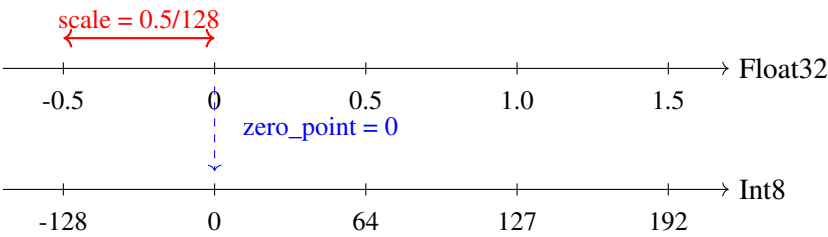


Figure 3.3: Mapping between Float32 and Int8 ranges using scale and zero-point.

3.1.3 Quantization Techniques Comparison

Table 3.1: Comparison of Quantization Techniques for Edge Deployment

Technique	Size	Accuracy Loss	Complexity	Best For
Float32 (baseline)	100%	0%	None	Development/debugging
Float16	50%	<0.5%	Low	GPU inference
Dynamic Range	25%	1–3%	Low	Quick deployment
Full Integer (Int8)	25%	2–5%	Medium	MCU deployment
Quantization-Aware Training	25%	<1%	High	Production edge

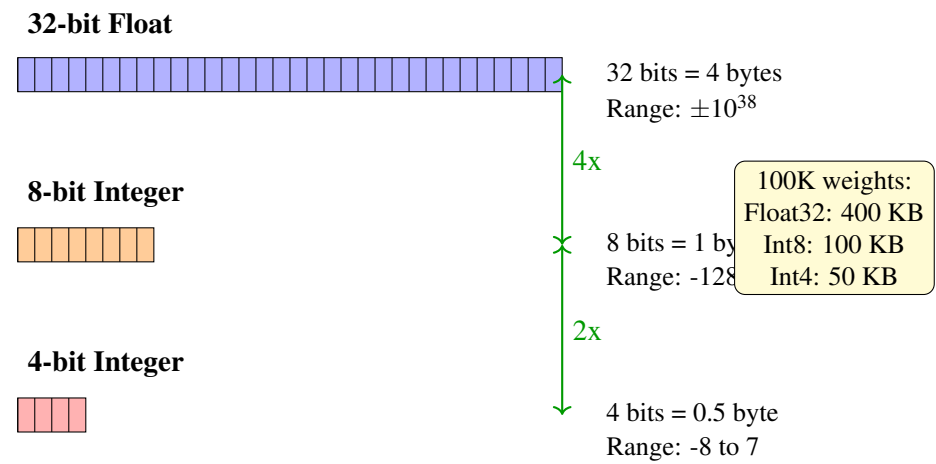


Figure 3.4: Quantization bit-width comparison showing memory reduction from 32-bit float to 8-bit and 4-bit representations. Each reduction halves the storage requirement.

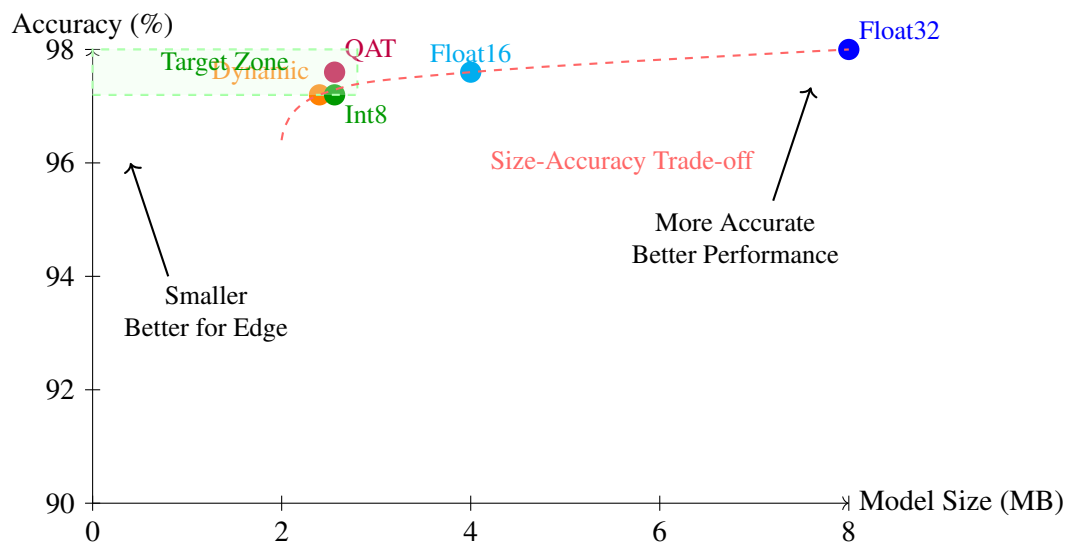


Figure 3.5: Accuracy vs Model Size trade-off for different quantization techniques. The green target zone represents the sweet spot for edge deployment: high accuracy with small size. QAT achieves the best balance.

Choosing the Right Technique

1. **Start with Dynamic Range:** Quick to apply, good baseline for initial testing
2. **Use Full Integer:** When deploying to MCUs or Coral TPU (required!)
3. **Use QAT:** When accuracy is critical and you have time to retrain
4. **Avoid Float16:** Unless specifically targeting GPU inference

Check Your Understanding

Before continuing, make sure you can answer:

1. Why does quantization reduce model size by approximately 4x?
2. What information do scale and zero-point provide?
3. When would you choose QAT over post-training quantization?

🔗 Choosing a Quantization Method

Use this decision tree to select the right quantization approach:

Step 1: What's your target hardware?

- GPU → Float16 (fastest GPU inference)
- Raspberry Pi / Desktop → Dynamic Range (easy, good performance)
- Microcontroller (Arduino, ESP32) → Full Integer (Int8 required)
- Coral Edge TPU → Full Integer (hardware requirement)

Step 2: Check accuracy after PTQ

- Accuracy loss <2% → Done! Use PTQ model
- Accuracy loss 2–5% → Try more representative samples (200+)
- Accuracy loss >5% → Use Quantization-Aware Training (QAT)

Step 3: If QAT is needed

- Use lower learning rate (10x smaller than original)
- Train for 10–20% of original epochs
- Start from pre-trained Float32 weights (not random)

💡 What's the Quantization Error?

Given a model with weights in the range $[-0.5, +0.5]$ and Int8 quantization (256 levels), calculate:

1. What is the scale factor? (Answer: $\frac{0.5 - (-0.5)}{255} \approx 0.004$)
2. What is the maximum quantization error per weight? (Answer: $\pm \text{scale}/2 \approx 0.002$)
3. If your model has 100,000 weights, what's the worst-case total accumulated error?

Insight: Quantization error is small per-weight, but accumulates across layers. This is why some models are more sensitive to quantization than others.

3.2 TFLite Converter: Your First Conversion

The TensorFlow Lite Converter transforms trained TensorFlow models into an optimized format for edge deployment. Let's walk through a complete example step by step.

Level 1: Notebook Simulation

Run this code in the companion Jupyter notebook. We'll convert a simple model that learns the relationship $y = 2x - 1$.

3.2.1 Step 1: Create and Train a Simple Model

First, let's create a minimal neural network:

```
1 import tensorflow as tf
2 import numpy as np
3 from tensorflow.keras import Sequential
4 from tensorflow.keras.layers import Dense
5
6 # Create a simple model: single neuron
7 l0 = Dense(units=1, input_shape=[1])
8 model = Sequential([l0])
9 model.compile(optimizer='sgd', loss='mean_squared_error')
```

Understanding the Model Setup

- **Line 7:** We create a Dense layer with 1 neuron and 1 input feature
- **Line 8:** Sequential wraps our single layer into a complete model
- **Line 9:** SGD optimizer and MSE loss are appropriate for regression

Now train the model on our simple linear data:

```
11 # Training data: y = 2x - 1
12 xs = np.array([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0], dtype=float)
13 ys = np.array([-3.0, -1.0, 1.0, 3.0, 5.0, 7.0], dtype=float)
14
```

```

15 # Train for 500 epochs
16 model.fit(xs, ys, epochs=500, verbose=0)
17
18 # Test the trained model
19 print(f"Prediction for x=10: {model.predict([10.0], verbose=0)[0][0]:.2f}")
20 print(f"Learned weights: {l0.get_weights()}")

```

Expected Output

```

Prediction for x=10: 18.98
Learned weights: [array([[1.9967]], dtype=float32),
                  array([-0.9901], dtype=float32)]

The model learned  $w \approx 2.0$  and  $b \approx -1.0$ , matching our target  $y = 2x - 1$ .

```

3.2.2 Step 2: Export as SavedModel

Before converting to TFLite, we save the model in TensorFlow's SavedModel format:

```

21 # Save the trained model
22 export_dir = 'saved_model/1'
23 tf.saved_model.save(model, export_dir)
24 print(f"Model saved to: {export_dir}")

```

Why SavedModel Format?

SavedModel is TensorFlow's recommended serialization format. It preserves:

- The complete model architecture
- Trained weights and biases
- Metadata for reproducibility

While you *can* convert directly from a Keras model, SavedModel provides better compatibility for production deployment.

3.2.3 Step 3: Convert to TFLite

Now we use the TFLite Converter:

```

26 # Create the converter
27 converter = tf.lite.TFLiteConverter.from_saved_model(export_dir)
28
29 # Convert the model
30 tflite_model = converter.convert()
31
32 # Save the TFLite model
33 import pathlib
34 tflite_model_file = pathlib.Path('model.tflite')
35 tflite_model_file.write_bytes(tflite_model)
36 print(f"TFLite model size: {len(tflite_model)} bytes")

```

Code Walkthrough

- **Line 27:** `from_saved_model()` creates a converter from our saved model
- **Line 30:** `convert()` performs the actual conversion, returning raw bytes
- **Lines 33–35:** We save the bytes to a `.tflite` file

Expected Output

TFLite model size: 764 bytes

Our tiny model is only 764 bytes! Larger models will see more dramatic size differences after quantization.

3.2.4 Step 4: Run Inference with TFLite Interpreter

To verify our conversion worked, we run inference using the TFLite interpreter:

```

37 # Load the TFLite model
38 interpreter = tf.lite.Interpreter(model_content=tflite_model)
39 interpreter.allocate_tensors()
40
41 # Get input and output tensor details
42 input_details = interpreter.get_input_details()
43 output_details = interpreter.get_output_details()
44
45 # Print tensor information
46 print("Input details:", input_details)
47 print("Output details:", output_details)

```

Critical Step: `allocate_tensors()`

You **must** call `allocate_tensors()` before running inference. This allocates memory for all intermediate tensors. Forgetting this step is a common source of errors!

Now run a prediction:

```

49 # Prepare input (must match expected shape and type)
50 to_predict = np.array([[10.0]], dtype=np.float32)
51
52 # Run inference
53 interpreter.set_tensor(input_details[0]['index'], to_predict)
54 interpreter.invoke()
55 tflite_results = interpreter.get_tensor(output_details[0]['index'])
56
57 print(f"TFLite prediction for x=10: {tflite_results[0][0]:.2f}")

```

TFLite Inference Pattern

The TFLite inference pattern is always the same:

1. `set_tensor()`: Load input data into the input tensor
2. `invoke()`: Run the model forward pass
3. `get_tensor()`: Retrieve the output

This pattern is identical on all platforms—Python, C++, and microcontrollers.

Expected Output

TFLite prediction for x=10: 18.98

The TFLite model produces the same result as the original TensorFlow model!

Experiment: Converter Options

Modify the code to try different converter sources:

1. Use `TFLiteConverter.from_keras_model(model)` instead of `SavedModel`
2. Compare the resulting file sizes
3. Test if predictions are identical

Which method is faster for development? Which is better for production?

Self-Assessment

Before moving on:

1. What is the purpose of `allocate_tensors()`?
2. What three steps make up the TFLite inference pattern?
3. Why do we need `input_details` and `output_details`?

3.3 Post-Training Quantization in Practice

Now let's apply quantization to a real image classification model. We'll use transfer learning with MobileNetV2 to classify cats vs. dogs, then compare three quantization approaches.

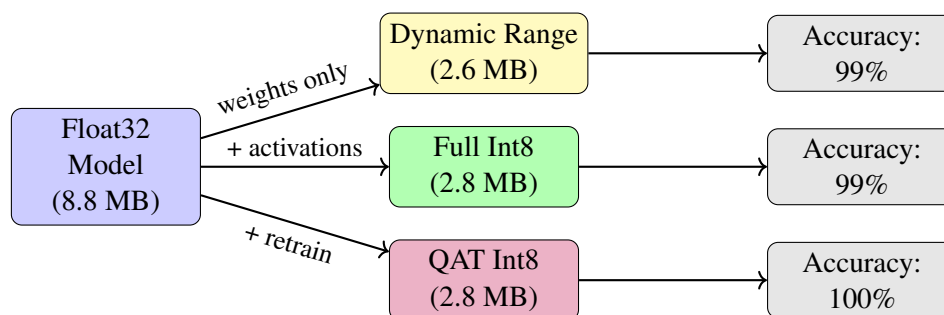


Figure 3.6: Three quantization paths with typical size and accuracy results for MobileNetV2 on cats vs. dogs.

3.3.1 Setup: Transfer Learning with MobileNetV2

First, let's set up our image classification model:

```

1 import tensorflow as tf
2 import tensorflow_hub as hub

```

```

3 import tensorflow_datasets as tfds
4 import numpy as np
5
6 # Image preprocessing function
7 def format_image(image, label):
8     image = tf.image.resize(image, (224, 224)) / 255.0
9     return image, label
10
11 # Load cats vs dogs dataset
12 (raw_train, raw_validation, raw_test), metadata = tfds.load(
13     'cats_vs_dogs',
14     split=['train[:80%]', 'train[80%:90%]', 'train[90%:]'],
15     with_info=True,
16     as_supervised=True,
17 )

```

Understanding the Setup

- **Line 8:** Resize to 224x224 (MobileNetV2's expected size) and normalize to [0,1]
- **Lines 12–17:** Load dataset with 80%/10%/10% train/val/test split

```

19 # Create data batches
20 BATCH_SIZE = 32
21 train_batches = raw_train.shuffle(1000).map(format_image).batch(BATCH_SIZE).prefetch(1)
22 validation_batches = raw_validation.map(format_image).batch(BATCH_SIZE).prefetch(1)
23 test_batches = raw_test.map(format_image).batch(1) # Batch size 1 for inference
24
25 num_classes = metadata.features['label'].num_classes
26 print(f"Number of classes: {num_classes}") # 2: cat and dog

```

Now build the model using transfer learning:

```

27 # Load pre-trained MobileNetV2 as feature extractor
28 MODULE_HANDLE = "https://tfhub.dev/google/tf2-preview/mobilenet_v2/feature_vector/4"
29
30 feature_extractor = hub.KerasLayer(
31     MODULE_HANDLE,
32     input_shape=(224, 224, 3),
33     output_shape=[1280],
34     trainable=False # Freeze backbone weights
35 )
36
37 # Build complete model
38 model = tf.keras.Sequential([
39     feature_extractor,
40     tf.keras.layers.Dense(num_classes, activation='softmax')
41 ])
42
43 model.compile(optimizer='adam',
44               loss='sparse_categorical_crossentropy',
45               metrics=['accuracy'])
46
47 model.summary()

```


Transfer Learning for Edge

By setting `trainable=False`, we freeze the MobileNetV2 backbone and only train the final classification layer. This:

- Reduces training time dramatically (minutes instead of hours)
- Requires less training data (the backbone already knows general features)
- Works well for edge models since MobileNetV2 is already optimized for mobile

Train and save the model:

```

49 # Train for 5 epochs (transfer learning needs fewer epochs)
50 EPOCHS = 5
51 history = model.fit(train_batches, epochs=EPOCHS, validation_data=validation_batches)
52
53 # Save the trained model
54 SAVED_MODEL_DIR = "cats_vs_dogs_saved_model"
55 tf.saved_model.save(model, SAVED_MODEL_DIR)

```

Expected Training Output

```

Epoch 1/5 - loss: 0.1842 - accuracy: 0.9612 - val_accuracy: 0.9835
Epoch 2/5 - loss: 0.0543 - accuracy: 0.9876 - val_accuracy: 0.9870
...
Epoch 5/5 - loss: 0.0298 - accuracy: 0.9912 - val_accuracy: 0.9905

```

With transfer learning, we achieve >99% validation accuracy in just 5 epochs!

3.3.2 Model 1: No Quantization (Float32 Baseline)

```

57 import pathlib
58
59 # Convert without any optimizations
60 converter = tf.lite.TFLiteConverter.from_saved_model(SAVED_MODEL_DIR)
61 tflite_model = converter.convert()
62
63 # Save and check size
64 tflite_models_dir = pathlib.Path("tflite_models/")
65 tflite_models_dir.mkdir(exist_ok=True)
66 model1_path = tflite_models_dir / 'model_float32.tflite'
67 model1_path.write_bytes(tflite_model)
68
69 print(f"Float32 model size: {model1_path.stat().st_size / 1e6:.2f} MB")

```

Expected Output

```
Float32 model size: 8.86 MB
```

3.3.3 Model 2: Dynamic Range Quantization

Dynamic range quantization converts weights to Int8 but keeps activations as Float32 at runtime:

```

71 # Convert with dynamic range quantization
72 converter = tf.lite.TFLiteConverter.from_saved_model(SAVED_MODEL_DIR)
73 converter.optimizations = [tf.lite.Optimize.DEFAULT] # Enable optimization
74 tflite_model_dynamic = converter.convert()
75
76 # Save and check size
77 model2_path = tflite_models_dir / 'model_dynamic.tflite'
78 model2_path.write_bytes(tflite_model_dynamic)
79
80 print(f"Dynamic range model size: {model2_path.stat().st_size / 1e6:.2f} MB")

```

What Does DEFAULT Optimization Do?

`tf.lite.Optimize.DEFAULT` performs:

- Weight quantization: Float32 weights → Int8
- Operator fusion: Combines adjacent operations for efficiency
- Constant folding: Pre-computes constant expressions

Activations remain Float32 and are quantized/dequantized at runtime.

Expected Output

Dynamic range model size: 2.63 MB

That's a **70% size reduction** with just one line of code!

3.3.4 Model 3: Full Integer Quantization

For MCU deployment, we need *all* operations to use integers. This requires a representative dataset to calibrate activation ranges:

```

82 # Define representative dataset generator
83 def representative_data_gen():
84     for input_value, _ in test_batches.take(100):
85         yield [input_value]
86
87 # Convert with full integer quantization
88 converter = tf.lite.TFLiteConverter.from_saved_model(SAVED_MODEL_DIR)
89 converter.optimizations = [tf.lite.Optimize.DEFAULT]
90 converter.representative_dataset = representative_data_gen
91 converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
92 converter.inference_input_type = tf.uint8 # or tf.int8
93 converter.inference_output_type = tf.uint8
94
95 tflite_model_int8 = converter.convert()
96
97 # Save and check size
98 model3_path = tflite_models_dir / 'model_int8.tflite'
99 model3_path.write_bytes(tflite_model_int8)
100
101 print(f"Full Int8 model size: {model3_path.stat().st_size / 1e6:.2f} MB")

```

Critical: Representative Dataset Quality

The representative dataset is **the most important factor** for quantization accuracy:

- Use 100–500 samples from your validation set (not training set)
- Ensure samples cover all classes and edge cases
- Poor calibration data = poor quantized accuracy

Full Integer Quantization Parameters

- **representative_dataset**: Generator yielding input samples for calibration
- **supported_ops**: Forces all ops to use Int8 arithmetic
- **inference_input_type**: Avoids float-to-int conversion at runtime
- **inference_output_type**: Output as integer (faster on MCUs)

Expected Output

Full Int8 model size: 2.84 MB

Slightly larger than dynamic range due to quantization parameter storage, but runs entirely as integers.

Why is Model 3 Larger than Model 2?

Full integer quantization stores additional metadata:

- Scale and zero-point for every tensor
- Quantization parameters for activations

The small size increase is worthwhile because inference is **much faster**—all operations use integer arithmetic only.

3.3.5 Comparing Accuracy Across Models

Let's test all three models on 100 images:

```

103 def evaluate_tflite_model(model_path, test_data, num_samples=100):
104     """Evaluate a TFLite model on test data."""
105     interpreter = tf.lite.Interpreter(model_path=str(model_path))
106     interpreter.allocate_tensors()
107
108     input_details = interpreter.get_input_details()
109     output_details = interpreter.get_output_details()
110
111     correct = 0
112     for img, label in test_data.take(num_samples):
113         # Handle input type conversion if needed
114         if input_details[0]['dtype'] == np.uint8:
115             input_scale, input_zero = input_details[0]['quantization']
116             img = img / input_scale + input_zero
117             img = tf.cast(img, tf.uint8)
118
119     interpreter.set_tensor(input_details[0]['index'], img)

```

```

120     interpreter.invoke()
121     output = interpreter.get_tensor(output_details[0]['index'])
122
123     prediction = np.argmax(output)
124     if prediction == label.numpy()[0]:
125         correct += 1
126
127     return correct / num_samples * 100

```

```

128 # Evaluate all three models
129 acc1 = evaluate_tflite_model(model1_path, test_batches)
130 acc2 = evaluate_tflite_model(model2_path, test_batches)
131 acc3 = evaluate_tflite_model(model3_path, test_batches)
132
133 print(f"Float32 accuracy:      {acc1:.1f}%")
134 print(f"Dynamic range accuracy: {acc2:.1f}%")
135 print(f"Full Int8 accuracy:     {acc3:.1f}%")

```

Expected Output

```

Float32 accuracy:      100.0%
Dynamic range accuracy: 99.0%
Full Int8 accuracy:     99.0%

Only 1% accuracy drop with 70% size reduction!

```

3.3.6 Summary: Model Comparison

Table 3.2: Post-Training Quantization Results Summary

Model	Size (MB)	Reduction	Accuracy
Float32 (baseline)	8.86	—	100%
Dynamic Range	2.63	70%	99%
Full Integer (Int8)	2.84	68%	99%

Experiment: Representative Dataset Size

Modify the representative dataset generator to use different numbers of samples:

1. Try with only 10 samples—does accuracy drop?
2. Try with 500 samples—does accuracy improve?
3. What's the minimum number of samples for acceptable accuracy?

Check Your Understanding

Before continuing:

1. What is the difference between dynamic range and full integer quantization?

2. Why does full integer quantization require a representative dataset?
3. When would you use `inference_input_type = tf.uint8`?

3.4 Exploring Quantization Under the Hood

To build intuition for why quantization works so well, let's implement the algorithm ourselves in pure Python, then visualize why neural network weights are well-suited for quantization.

Why Do Weights Quantize Well?

Neural network weights typically follow a near-Gaussian distribution centered around zero. This means most values cluster in a small range, making them ideal for quantization—we can use a small scale to represent the majority of values precisely.

3.4.1 Implementing Quantization from Scratch

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 # Simulate a weight matrix (typical neural network weights)
5 np.random.seed(42)
6 weights = np.random.randn(256, 256) # Gaussian distribution
7
8 print(f"Weight range: [{weights.min():.3f}, {weights.max():.3f}]")
9 print(f"Weight mean: {weights.mean():.3f}")
10 print(f"Weight std: {weights.std():.3f}")

```

Expected Output

```

Weight range: [-4.191, 4.127]
Weight mean: 0.000
Weight std: 1.002

```

Weights center around 0 with values mostly within ± 3 standard deviations.

Now implement the quantization algorithm:

```

12 def quantize_and_reconstruct(weights):
13     """
14     Quantize float32 weights to int8 and reconstruct.
15
16     Returns: reconstructed weights (float32 representable as int8)
17     """
18     # Step 1: Find the range
19     max_val = np.max(weights)
20     min_val = np.min(weights)
21     weight_range = max_val - min_val
22
23     # Step 2: Compute scale (256 levels for 8 bits)
24     scale = weight_range / 256
25
26     # Step 3: Find midpoint (zero-point)

```

```

27     midpoint = (max_val + min_val) / 2
28
29     # Step 4: Quantize (center, scale, round to integer)
30     centered = weights - midpoint
31     quantized = np rint(centered / scale) # Round to nearest int
32     quantized = np.clip(quantized, -128, 127) # Clip to int8 range
33
34     # Step 5: Reconstruct
35     reconstructed = scale * quantized + midpoint
36
37     return reconstructed, scale, quantized

```

Understanding the Algorithm

- **Lines 18–20:** Calculate the full range of values to map
- **Line 23:** Scale determines the step size—smaller range = smaller scale = better precision
- **Line 26:** Zero-point centers the distribution for symmetric quantization
- **Lines 29–31:** Round to integers and clip to valid int8 range
- **Line 34:** Reverse the process to get reconstructed float values

Test the quantization:

```

38 # Apply quantization
39 reconstructed, scale, quantized = quantize_and_reconstruct(weights)
40
41 # Calculate error
42 errors = reconstructed - weights
43 max_error = np.max(np.abs(errors))
44 mean_error = np.mean(np.abs(errors))
45
46 print(f"Scale: {scale:.6f}")
47 print(f"Max absolute error: {max_error:.6f}")
48 print(f"Mean absolute error: {mean_error:.6f}")
49 print(f"Unique values in quantized: {len(np.unique(reconstructed))}")

```

Expected Output

```

Scale: 0.032500
Max absolute error: 0.016250
Mean absolute error: 0.008203
Unique values in quantized: 256

```

The maximum error is only 0.016—tiny compared to typical weight values!

Why Quantization Error is Small

With scale = 0.033, adjacent integer values differ by only 0.033 in float representation. Since weights range over ≈ 8 units but cluster around 0, this precision is sufficient. The reconstruction error is at most $\frac{\text{scale}}{2} \approx 0.016$.

3.4.2 Visualizing Weight Distributions

Let's see why neural network weights are ideal for quantization:

```

50 # Plot weight distribution
51 fig, axes = plt.subplots(1, 2, figsize=(12, 4))
52
53 # Original distribution
54 axes[0].hist(weights.flatten(), bins=100, alpha=0.7, color='blue')
55 axes[0].set_title('Original Float32 Weights')
56 axes[0].set_xlabel('Weight Value')
57 axes[0].set_ylabel('Count')
58 axes[0].axvline(x=0, color='red', linestyle='--', label='Zero')
59 axes[0].legend()
60
61 # Quantized distribution
62 axes[1].hist(reconstructed.flatten(), bins=100, alpha=0.7, color='green')
63 axes[1].set_title('Quantized (Int8) Weights')
64 axes[1].set_xlabel('Weight Value')
65 axes[1].set_ylabel('Count')
66 axes[1].axvline(x=0, color='red', linestyle='--')
67
68 plt.tight_layout()
69 plt.savefig('weight_distribution.png', dpi=150)
70 plt.show()

```

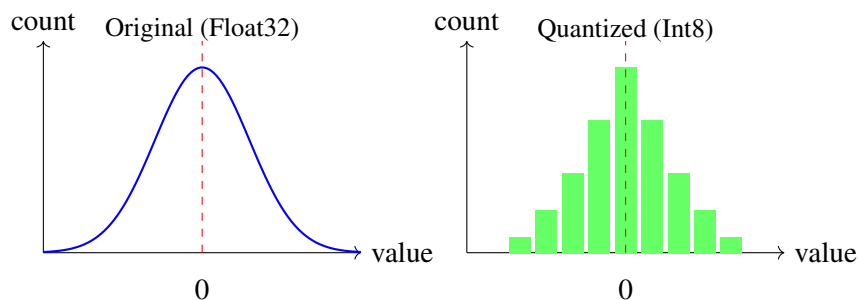


Figure 3.7: Weight distributions before and after quantization. The continuous Float32 distribution becomes discrete Int8 levels, but the shape is preserved.

Why Weights Cluster Around Zero

During training, regularization techniques (L2/weight decay) penalize large weights. Combined with random initialization near zero and gradient descent updates, weights tend to:

- Stay small in magnitude
- Follow approximately Gaussian distributions
- Cluster around zero with few outliers

This is *why* quantization works—we're not trying to represent arbitrary floats, but specifically the narrow distributions that neural networks produce.

Experiment: Different Distributions

Modify the weight generation to explore different distributions:

1. Uniform distribution: `np.random.uniform(-5, 5, (256, 256))`
 2. Sparse weights: Set 90% of weights to zero
 3. Outliers: Add a few values at ± 100
- How does each affect quantization error? Which is hardest to quantize?

Check Your Understanding

1. What determines the quantization error (max error per value)?
2. Why do Gaussian-distributed weights quantize better than uniform?
3. What happens to quantization accuracy if weights have large outliers?

3.5 Quantization-Aware Training (QAT)

When post-training quantization drops accuracy too much (more than 2–3%), Quantization-Aware Training (QAT) can recover most of the lost accuracy by simulating quantization during training.

QAT vs Post-Training Quantization

- **Post-Training Quantization:** Apply quantization *after* training. Quick but may lose 2–5% accuracy.
- **Quantization-Aware Training:** Simulate quantization *during* training. The model learns to be robust to precision loss. Typically <1% accuracy drop.

Rule of thumb: Start with PTQ. If accuracy drops >2%, switch to QAT.

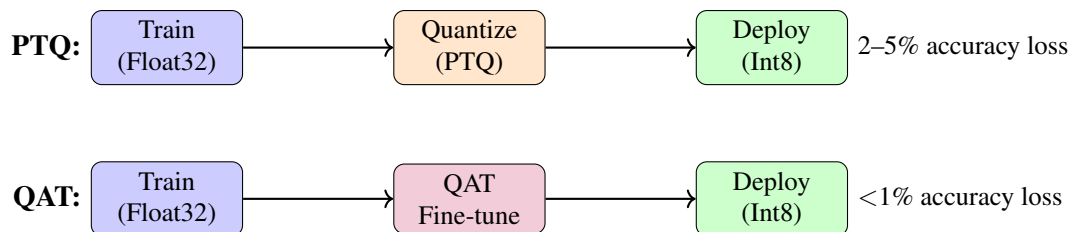


Figure 3.8: Comparison of Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT) workflows.

3.5.1 How QAT Works

QAT inserts “fake quantization” nodes into the model during training:

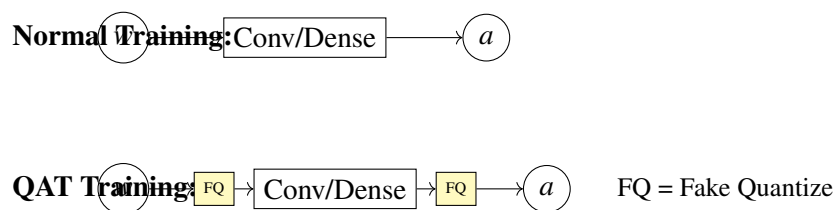


Figure 3.9: QAT adds fake quantization (FQ) nodes that simulate Int8 precision during training.

The fake quantization operation:

1. Quantizes the value to Int8 (rounds and clips)
2. Immediately dequantizes back to Float32
3. Forward pass sees quantized values
4. Backward pass uses straight-through estimator (gradients pass unchanged)

3.5.2 Implementing QAT with TensorFlow

First, train a baseline model:

```

1 import tensorflow as tf
2 import tensorflow_model_optimization as tfmot
3
4 # Load MNIST dataset
5 mnist = tf.keras.datasets.mnist
6 (train_images, train_labels), (test_images, test_labels) = mnist.load_data()
7 train_images = train_images / 255.0
8 test_images = test_images / 255.0
9
10 # Build baseline model
11 def create_model():
12     return tf.keras.Sequential([
13         tf.keras.layers.InputLayer(input_shape=(28, 28)),
14         tf.keras.layers.Reshape(target_shape=(28, 28, 1)),
15         tf.keras.layers.Conv2D(32, kernel_size=(3, 3), activation='relu'),
16         tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
17         tf.keras.layers.Conv2D(64, kernel_size=(3, 3), activation='relu'),
18         tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
19         tf.keras.layers.Flatten(),
20         tf.keras.layers.Dense(10, activation='softmax')
21     ])
22
23 # Train baseline
24 model = create_model()
25 model.compile(optimizer='adam',
26               loss='sparse_categorical_crossentropy',
27               metrics=['accuracy'])
28
29 model.fit(train_images, train_labels, epochs=5, validation_split=0.1)
30 baseline_accuracy = model.evaluate(test_images, test_labels)[1]
31 print(f"Baseline accuracy: {baseline_accuracy * 100:.2f}%")

```

Expected Output

```

Epoch 5/5 - accuracy: 0.9912 - val_accuracy: 0.9867
Baseline accuracy: 98.72%

```

Now apply QAT:

```

32 # Apply quantization-aware training to the model
33 qat_model = tfmot.quantization.keras.quantize_model(model)
34
35 # IMPORTANT: Must recompile after quantization wrapping

```

```

36 qat_model.compile(optimizer='adam',
37                  loss='sparse_categorical_crossentropy',
38                  metrics=['accuracy'])
39
40 qat_model.summary() # Notice the QuantizeWrapper layers

```

Must Recompile After quantize_model()

The model structure changes when you apply `quantize_model()`. You **must** call `compile()` again before training, or the model won't train correctly.

Fine-tune with QAT:

```

42 # Fine-tune with QAT (fewer epochs, lower learning rate)
43 qat_model.fit(train_images, train_labels,
44              epochs=2, # Just 2 epochs is often enough
45              validation_split=0.1)
46
47 qat_accuracy = qat_model.evaluate(test_images, test_labels)[1]
48 print(f"QAT model accuracy: {qat_accuracy * 100:.2f}%")

```

Expected Output

```

Epoch 2/2 - accuracy: 0.9923 - val_accuracy: 0.9878
QAT model accuracy: 98.85%

QAT maintains (or even slightly improves) accuracy!

```

3.5.3 Converting QAT Model to TFLite

```

50 # Convert QAT model to TFLite
51 converter = tf.lite.TFLiteConverter.from_keras_model(qat_model)
52 converter.optimizations = [tf.lite.Optimize.DEFAULT]
53
54 qat_tflite_model = converter.convert()
55
56 # Save the model
57 import pathlib
58 qat_model_path = pathlib.Path('mnist_qat_int8.tflite')
59 qat_model_path.write_bytes(qat_tflite_model)
60
61 print(f"QAT TFLite model size: {len(qat_tflite_model) / 1e3:.1f} KB")

```

QAT Provides Calibration Data Automatically

Unlike post-training quantization, QAT models don't need a representative dataset. The fake quantization nodes learned the correct scale and zero-point values during training.

Expected Output

QAT TFLite model size: 24.8 KB

3.5.4 Comparing All Approaches

```

62 # Compare: Baseline PTQ vs QAT
63 # Baseline model with PTQ
64 converter_ptq = tf.lite.TFLiteConverter.from_keras_model(model)
65 converter_ptq.optimizations = [tf.lite.Optimize.DEFAULT]
66
67 def representative_data():
68     for i in range(100):
69         yield [train_images[i:i+1].astype(np.float32)]
70
71 converter_ptq.representative_dataset = representative_data
72 converter_ptq.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
73
74 ptq_tflite_model = converter_ptq.convert()
75
76 print(f"PTQ model size: {len(ptq_tflite_model) / 1e3:.1f} KB")
77 print(f"QAT model size: {len(qat_tflite_model) / 1e3:.1f} KB")

```

3.5.5 QAT Hyperparameter Guidelines

Table 3.3: QAT Fine-tuning Parameters

Parameter	Typical Value	Range	Notes
Fine-tuning epochs	2–5	1–10	More epochs rarely helps past 5
Learning rate	1e-5 to 1e-4	–	10x lower than initial training
Training data	Full dataset	10–100%	More data = better accuracy
Batch size	Same as original	–	Keep consistent

Common QAT Mistakes

- **Too high learning rate:** Destroys pre-trained features. Start at 10–100x lower.
- **Too many epochs:** Overfits to quantization noise. Usually 2–5 is enough.
- **Forgetting to recompile:** Model won't train without recompilation after `quantize_model()`.
- **Wrong layer types:** Some layers (BatchNorm) need special handling.

Experiment: QAT from Scratch

Instead of fine-tuning a pre-trained model, try:

1. Apply `quantize_model()` to an untrained model
2. Train from scratch with QAT enabled

3. Compare accuracy to fine-tuning approach
When might training from scratch be better?

Check Your Understanding

1. What is a “fake quantization” node and why is it needed?
2. Why do QAT models not need a representative dataset for conversion?
3. When should you use QAT instead of post-training quantization?
4. What learning rate should you use for QAT fine-tuning?

3.6 Deploying Quantized Models Across Tiers

Now that you’ve created quantized models, let’s deploy them at each execution tier.

Level 1: Notebook Verification

Run the TFLite interpreter directly in Python to verify model accuracy:

```
1 import tensorflow as tf
2 import numpy as np
3
4 # Load quantized model
5 interpreter = tf.lite.Interpreter(model_path='model_int8.tflite')
6 interpreter.allocate_tensors()
7
8 # Get I/O details
9 input_details = interpreter.get_input_details()
10 output_details = interpreter.get_output_details()
11
12 # Run inference
13 interpreter.set_tensor(input_details[0]['index'], test_input)
14 interpreter.invoke()
15 output = interpreter.get_tensor(output_details[0]['index'])
```

What to verify:

- Accuracy matches expectations from quantization
- Input/output shapes are correct
- No runtime errors during inference

Level 2: Raspberry Pi with tflite-runtime

Deploy to Raspberry Pi using the lightweight tflite-runtime package:

```
1 # Install on Raspberry Pi
2 pip install tflite-runtime
3
4 # Python code (same API, different import)
5 from tflite_runtime.interpreter import Interpreter
6
7 interpreter = Interpreter(model_path='model_int8.tflite')
8 interpreter.allocate_tensors()
```

```
6 # ... same inference code as Level 1
```

Performance optimizations:

- Use `interpreter.set_num_threads(4)` for multi-core
- Consider Coral USB Accelerator for 10x speedup
- Profile with `time.perf_counter()` to measure latency

See Also: LAB11 - Profiling and Optimization

For comprehensive performance analysis of quantized models, see LAB11 where you'll learn to:

- Profile memory usage and inference latency systematically
- Measure the actual speedup from int8 quantization on real hardware
- Identify bottlenecks in your inference pipeline
- Optimize layer-by-layer performance

Level 3: MCU Deployment with TFLM

Deploy to microcontrollers (Arduino, ESP32) using TensorFlow Lite Micro:

Step 1: Convert model to C array

```
1 xxd -i model_int8.tflite > model_data.cc
```

Step 2: Arduino sketch structure

```
1 #include "tensorflow/lite/micro/micro_interpreter.h"
2 #include "model_data.h"
3
4 constexpr int kTensorArenaSize = 10 * 1024;
5 uint8_t tensor_arena[kTensorArenaSize];
6
7 void setup() {
8     const tflite::Model* model = tflite::GetModel(g_model_data);
9
10    static tflite::MicroMutableOpResolver<5> resolver;
11    resolver.AddConv2D();
12    resolver.AddFullyConnected();
13    resolver.AddSoftmax();
14
15    static tflite::MicroInterpreter interpreter(
16        model, resolver, tensor_arena, kTensorArenaSize);
17    interpreter.AllocateTensors();
18 }
```

Memory constraints:

- **Arduino Nano 33 BLE:** 256KB RAM, 1MB Flash—models up to ~100KB
- **ESP32:** 520KB RAM, 4MB Flash—models up to ~300KB
- Int8 quantization is **required** for MCU deployment

3.6.1 Deployment Decision Matrix

Table 3.4: Choosing the Right Deployment Tier

Requirement	Level 1	Level 2	Level 3
Latency target	>100ms OK	10–100ms	<10ms
Power budget	Unlimited	~5W	<100mW
Connectivity	Required	Optional	None/BLE
Model size	<500MB	<50MB	<300KB
Development speed	Fastest	Fast	Slower

Chapter Summary

Key Takeaways

- **Quantization reduces model size by 4x** by converting Float32 weights to Int8
- **Three quantization approaches:** Dynamic range (quick), Full Int8 (MCU-ready), QAT (best accuracy)
- **Representative dataset quality** is critical for post-training quantization accuracy
- **QAT recovers accuracy** by simulating quantization during training
- **Always validate** quantized model accuracy before deployment

Self-Assessment Questions

Test your understanding before continuing:

1. Why does int8 quantization reduce model size by approximately 4x compared to float32? Calculate the exact reduction factor.
2. What is the purpose of the scale and zero-point parameters in quantization? How do they enable mapping between float32 and int8 ranges?
3. When would you choose full integer quantization over dynamic range quantization? Consider both hardware requirements and accuracy trade-offs.
4. Why does full integer quantization require a representative dataset, while dynamic range quantization does not? What gets calibrated?
5. Explain why quantization-aware training (QAT) typically achieves better accuracy than post-training quantization. What happens during QAT that makes the difference?

3.7 Guided Experiments

Experiment 1: Quantization Accuracy vs Size Trade-off

Hypothesis: Different quantization techniques offer different trade-offs between model size reduction and accuracy preservation. We expect dynamic range to be fastest, full int8 to be smallest but with more accuracy loss, and QAT to recover most accuracy.

Procedure:

1. Train a simple MNIST classification model (or use pre-trained from Section 3)
2. Convert the model using three quantization approaches:

- Float32 (baseline)
 - Dynamic range quantization
 - Full integer (int8) quantization
3. Measure model file size for each approach
 4. Evaluate accuracy on 1000 test samples for each model
 5. Record results in table

Data Collection:

Quantization Method	Size (KB)	Size Reduction	Accuracy (%)	Accuracy Drop
Float32 (baseline)		-		-
Dynamic Range				
Full Int8				

Analysis Questions:

- What is the size reduction percentage for each quantization method?
- Is there a correlation between size reduction and accuracy loss?
- For deploying to an Arduino Nano 33 BLE (256KB RAM), which quantization method is required?
- Would the accuracy trade-off be acceptable for your target application?

Experiment 2: Representative Dataset Impact

Hypothesis: The size and quality of the representative dataset used for calibration significantly affects quantized model accuracy. We expect accuracy to improve with more representative samples up to a saturation point.

Procedure:

1. Use the MobileNetV2 model from Section 3.3
2. Apply full integer quantization with different representative dataset sizes:
 - 10 samples
 - 50 samples
 - 100 samples
 - 200 samples
 - 500 samples
3. Ensure samples are randomly selected from validation set
4. Evaluate quantized model accuracy on full test set
5. Record conversion time and final accuracy for each

Data Collection:

Dataset Size	Conversion Time (s)	Test Accuracy (%)	Accuracy vs Baseline
10			
50			
100			
200			
500			

Analysis Questions:

- At what dataset size does accuracy stop improving significantly?
- Is there a linear relationship between representative dataset size and accuracy?
- Does conversion time increase linearly with dataset size?

- What is the minimum representative dataset size you would recommend for production deployment?

3.8 Troubleshooting Guide

TFLite Converter Fails with AttributeError

Symptoms: `AttributeError: 'TFLiteConverter' object has no attribute 'from_keras_model'` or similar conversion errors

Cause: API changes between TensorFlow versions or incompatible TensorFlow Lite version

Solution:

1. For TF 2.x, use: `converter = tf.lite.TFLiteConverter.from_keras_model(model)`
2. For TF 1.x (deprecated), use: `converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir)`
3. Check TensorFlow version: `print(tf.__version__)` - ensure 2.10+
4. Save model first, then convert: `model.save('model.h5');` `converter = tf.lite.TFLiteConverter.from_saved_model('model.h5')`
5. Update TensorFlow: `pip install -upgrade tensorflow==2.15.0`

Quantization Causes Severe Accuracy Drop

Symptoms: Model accuracy drops from 95% to <60% after int8 quantization

Cause: Insufficient representative dataset, extreme value ranges, or model architecture incompatible with quantization

Solution:

1. Increase representative dataset size to 200+ diverse samples
2. Ensure representative data covers full input range: `X_rep = X_test[:200]`
3. Use quantization-aware training (QAT) instead of post-training quantization:
 - `import tensorflow_model_optimization as tfmot`
 - `quantize_model = tfmot.quantization.keras.quantize_model`
 - `q_aware_model = quantize_model(model)`
4. Check for layers unsupported in int8: Some custom layers may not quantize well
5. Try dynamic range quantization first before full int8

Representative Dataset Generator Not Working

Symptoms: `ValueError: generator did not yield a value or quantization fails with empty dataset`

Cause: Generator function incorrectly formatted or not yielding numpy arrays

Solution:

1. Ensure generator yields list of numpy arrays:

```
def representative_dataset():
    for i in range(100):
        yield [X_test[i:i+1].astype(np.float32)]
```

2. Verify data type is float32: `X.astype(np.float32)`
3. Check generator produces output: `list(representative_dataset())[:1]`

4. Ensure batch dimension: Shape should be (1, height, width, channels) not (height, width, channels)
5. Set converter properly: `converter.representative_dataset = representative_dataset`

⚠ TFLite Model Larger Than Original

Symptoms: Converted TFLite model file is same size or larger than original Keras model

Cause: Quantization not actually applied, using float32 instead of int8

Solution:

1. Verify optimization flags are set:
 - `converter.optimizations = [tf.lite.Optimize.DEFAULT]`
 - For full int8: `converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]`
2. Provide representative dataset for calibration (required for int8)
3. Check model actually quantized: `interpreter.get_tensor_details()`
4. Compare sizes: `os.path.getsize('model.tflite')` vs `os.path.getsize('model.h5')`
5. Ensure inference types set: `converter.inference_input_type = tf.uint8;`
`converter.inference_output_type = tf.uint8`

⚠ TFLite Interpreter Allocation Failed

Symptoms: `RuntimeError: tensorflow/lite/kernels/conv.cc allocation failed on device`

Cause: Model too large for device memory or incorrect tensor allocation

Solution:

1. Reduce model size: Prune layers, reduce filters, or use smaller input resolution
2. For Arduino: Ensure model <200KB for SRAM constraints
3. Check available memory: Print free heap before allocation
4. Use external flash for model storage: Store weights in Flash instead of SRAM
5. Try simpler model architecture: MobileNetV2 or EfficientNet-Lite instead of ResNet
6. Verify quantization applied correctly: Model should be 4x smaller after int8

Case Study: Smartphone Keyboard Next-Word Prediction

Context: A major smartphone manufacturer needed to implement real-time next-word prediction for their on-device keyboard. The feature had to run locally for privacy, predict within 10ms per keystroke, and work on devices with as little as 2GB RAM while preserving battery life.

Challenge: Their initial LSTM-based language model was 180MB in float32 format—far too large for on-device deployment. Even loading the model caused noticeable lag, and inference consumed excessive battery power. The model needed to be at least 4x smaller while maintaining prediction accuracy above 85%.

Solution: The team applied a multi-stage quantization strategy using techniques from this lab:

- Applied post-training dynamic range quantization, reducing the model from 180MB to 90MB with negligible accuracy loss
- Implemented full int8 quantization using a representative dataset of 500 diverse text

samples, achieving 45MB model size

- Fine-tuned with quantization-aware training (QAT) to recover 2% accuracy loss from aggressive quantization
- Optimized the TFLite model with operator fusion and constant folding

Results:

- Model size: 180MB → 45MB (4x reduction)
- Inference latency: 45ms → 8ms per prediction (5.6x faster)
- Power consumption: 65% reduction due to integer operations
- Prediction accuracy: 87.3% (maintained within 1% of original)
- Memory footprint: Reduced peak RAM usage from 220MB to 58MB

Lessons Learned:

- Start with dynamic range quantization before committing to full int8—it often provides 50% size reduction with zero accuracy loss
- Representative dataset quality matters more than quantity—ensure samples cover diverse writing styles, languages, and edge cases
- QAT is essential when post-training quantization degrades accuracy beyond acceptable thresholds
- Profile on actual target devices early—simulated performance doesn't capture memory fragmentation and thermal throttling effects
- Integer inference on mobile GPUs and NPUs can be 3-10x faster than float32, making quantization essential for real-time UX

Exercises

1. **[Basic] Basic: Basic Conversion:** Convert a pre-trained model of your choice to TFLite format. Measure the size before and after conversion.
2. **[Intermediate] Intermediate: Quantization Comparison:** For the cats vs. dogs model:
 - Create Float32, Dynamic Range, and Full Int8 versions
 - Measure size and accuracy for each
 - Create a table comparing the results
3. **[Intermediate] Intermediate: Representative Dataset Experiment:** Vary the representative dataset size from 10 to 500 samples. Plot accuracy vs. dataset size. What's the minimum for acceptable accuracy?
4. **[Challenge] Challenge: QAT Implementation:** Apply QAT to a model that loses >3% accuracy with PTQ. Can you recover the lost accuracy?
5. **[Challenge] Challenge: Edge Deployment Challenge:** Deploy a quantized MNIST model to:
 - Level 1: Python notebook (verify accuracy)
 - Level 2: Raspberry Pi (measure inference time)
 - Level 3: Arduino (if available) or simulate memory constraints

Related Labs

Prerequisites:

- **LAB02 - Neural Network Training Fundamentals:** Understanding of neural network architecture, training, and loss functions is essential for this lab.

Next Steps:

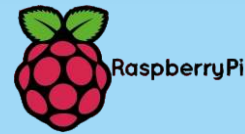
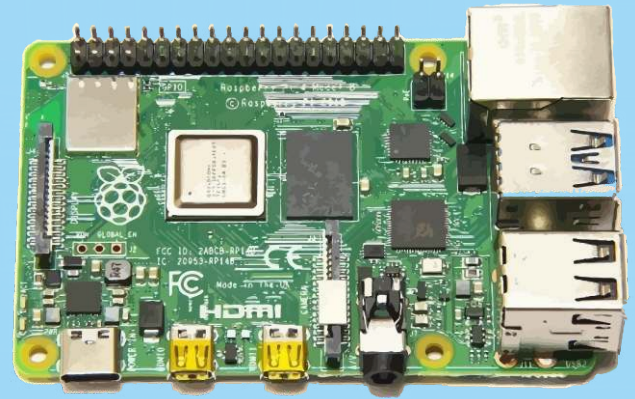
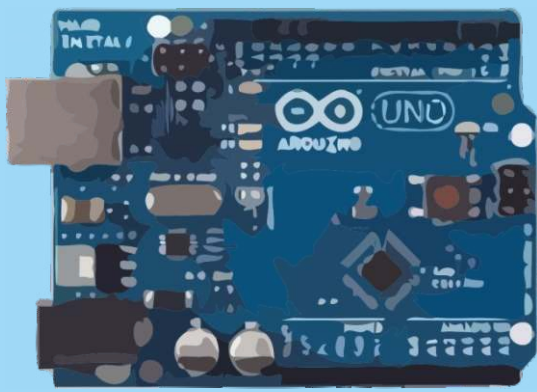
- **LAB04 - Keyword Spotting:** Apply quantization techniques to a real audio ML

application.

- **LAB05 - Deployment:** Deploy your quantized models to actual hardware using TensorFlow Lite Micro.
- **LAB07 - CNNs for Computer Vision:** Quantize convolutional neural networks for image classification on edge devices.

- **Further Resources.**
- TensorFlow Lite Model Optimization: https://www.tensorflow.org/lite/performance/model_optimization
 - Quantization Specification: https://www.tensorflow.org/lite/performance/quantization_spec
 - TFLite Micro for MCUs: <https://www.tensorflow.org/lite/microcontrollers>
 - Coral Edge TPU: <https://coral.ai/docs/>

— **Credits.** We have taken this code segment from the online course: *Edx Tiny Machine Learning (2021-22)*. Since we may have updated the code to reflect any changes made to the underlying frameworks where necessary. Credits should go to the original authors of this tutorial.



4. Audio Signal Processing and Keyword Spotting

Learning Objectives

After completing this lab, you will be able to:

- Explain how digital audio is represented and sampled
- Generate and interpret spectrograms from audio waveforms
- Understand the Mel scale and extract MFCC features
- Build and train a keyword spotting (KWS) model
- Evaluate model performance with confusion matrices
- Deploy audio models across the three execution tiers

Interactive Notebook

Jupyter Notebook: LAB04_keyword_spotting.ipynb

[Open in Google Colab](#) | [View on GitHub](#) | [Companion Website](#)

Voice-activated devices are everywhere—from smart speakers to car infotainment systems to industrial safety equipment. At the heart of these systems is **keyword spotting (KWS)**, the ability to detect specific words like “Hey Siri” or “OK Google” from continuous audio streams.

For edge devices, keyword spotting is particularly challenging. The model must run continuously, processing audio in real-time while consuming minimal power. A smart speaker’s “wake word” detector runs 24/7 on a tiny processor, waiting for the trigger phrase before activating the full speech recognition system.

In this chapter, you’ll learn the complete pipeline: from raw audio waveforms to spectrograms, from Mel-frequency cepstral coefficients (MFCCs) to neural network classifiers. By the end, you’ll have a working keyword spotting system that can recognize custom voice commands.

Prerequisites

Before starting this lab, ensure you:

- Completed LAB02 (ML Foundations): neural networks, training, evaluation
- Completed LAB03 (Quantization): model optimization for edge
- Basic understanding of waves and frequencies (high school physics)
- Python with NumPy and matplotlib experience

Hardware Requirements

- **Level 1:** Laptop with microphone, Google Colab (GPU recommended)
- **Level 2:** Raspberry Pi 4 + USB microphone
- **Level 3:** Arduino Nano 33 BLE Sense (built-in PDM microphone)

Using the Wrong Sample Rate

Audio ML models are trained on a specific sample rate (usually 16kHz for speech). If your microphone records at 44.1kHz or 8kHz but you don't resample, your model will fail silently—the spectrograms will look completely different! **Always verify:** `print(f"Sample rate: {sr} Hz")` and resample if needed: `librosa.load(file, sr=16000)`.

Three-Tier Execution Model

Tier 1: Jupyter/Colab

Extract audio features (spectrograms, MFCCs), train DS-CNN keyword spotting models on Speech Commands dataset, visualize confusion matrices, optimize model architecture **Tier 2: Raspberry Pi**

Deploy quantized KWS models with USB microphone, measure real-time inference latency, test with live audio input and background noise **Tier 3: Arduino/ESP32**

Run always-on wake word detection with built-in PDM microphone, implement sliding window inference, optimize for sub-50ms latency and low power

4.1 Understanding Digital Audio

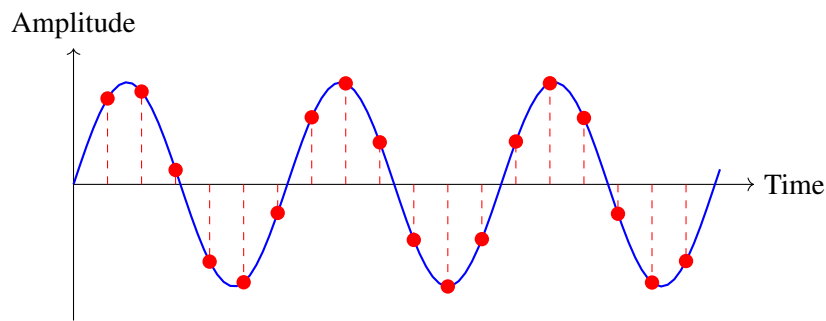
Before we can process audio with machine learning, we need to understand how sound is represented digitally.

Sound as Pressure Waves

Sound is a vibration that travels through air as pressure waves. When you speak, your vocal cords create rapid pressure changes. A microphone converts these pressure variations into electrical signals, which are then digitized as a sequence of numbers.

4.1.1 Sampling and Sample Rate

Digital audio is created by measuring (sampling) the analog signal at regular intervals:



Continuous signal (blue) sampled at discrete points (red)

Figure 4.1: Analog-to-digital conversion: The continuous audio signal is sampled at regular intervals to create a discrete sequence of values.

Table 4.1: Common Audio Sample Rates

Sample Rate	Max Frequency	Use Case
8,000 Hz	4 kHz	Telephone, basic voice
16,000 Hz	8 kHz	Keyword spotting, voice commands
44,100 Hz	22 kHz	CD quality, music
48,000 Hz	24 kHz	Professional audio, video

Nyquist Theorem

To accurately capture a frequency, you must sample at least twice as fast (Nyquist rate). Since human speech primarily occupies 300 Hz to 3,400 Hz, a sample rate of 16,000 Hz is sufficient for keyword spotting while keeping data manageable.

4.1.2 Loading and Visualizing Audio

Let's load an audio file and examine its waveform:

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import librosa
4 import librosa.display
5
6 # Load audio file (16kHz sample rate for KWS)
7 audio, sr = librosa.load('yes.wav', sr=16000)
8
9 print(f"Sample rate: {sr} Hz")
10 print(f"Duration: {len(audio) / sr:.2f} seconds")
11 print(f"Number of samples: {len(audio)}")

```

Expected Output

Sample rate: 16000 Hz
Duration: 1.00 seconds
Number of samples: 16000

One second of audio at 16 kHz = 16,000 samples. That's 16,000 numbers our model must process!

Visualize the waveform:

```
12 # Plot the waveform
13 plt.figure(figsize=(12, 4))
14 time = np.arange(len(audio)) / sr
15 plt.plot(time, audio)
16 plt.xlabel('Time (seconds)')
17 plt.ylabel('Amplitude')
18 plt.title('Audio Waveform: "Yes"')
19 plt.grid(True, alpha=0.3)
20 plt.show()
```

Understanding the Waveform

- **X-axis:** Time in seconds
- **Y-axis:** Amplitude (typically normalized to [-1, 1])
- **Shape:** The pattern reflects the speaker's pronunciation

The Problem with Raw Waveforms

While raw waveforms contain all the information, they're difficult for ML models to process directly:

- 16,000 features per second is too many
- Small time shifts make patterns look completely different
- Volume changes affect all values proportionally

We need a better representation—that's where spectrograms come in.

Check Your Understanding

1. Why is 16 kHz a common sample rate for voice applications?
2. How many samples are in 0.5 seconds of audio at 16 kHz?
3. Why can't we feed raw waveforms directly to a neural network efficiently?

4.2 From Waveforms to Spectrograms

A **spectrogram** shows how the frequency content of audio changes over time. It's like a heat map where:

- **X-axis:** Time

- **Y-axis:** Frequency
- **Color:** Energy/intensity at that time-frequency point

Why Spectrograms Work for Speech

Human speech is characterized by which frequencies are present, not by the exact waveform shape. The word “yes” has a distinctive frequency signature regardless of who says it or how loudly. Spectrograms capture this frequency information in a compact, time-aligned format.

4.2.1 The Short-Time Fourier Transform (STFT)

Spectrograms are computed using the Short-Time Fourier Transform:

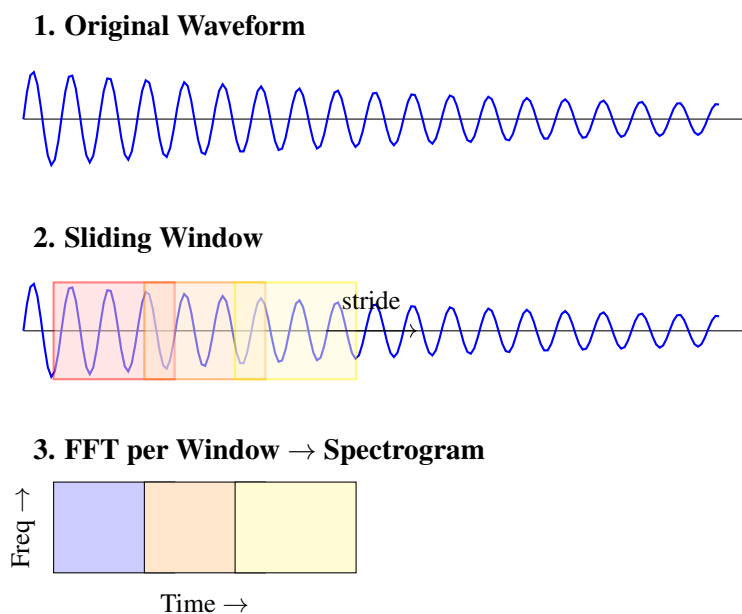


Figure 4.2: STFT process: Slide a window across the signal, compute FFT for each window position, stack results to form a spectrogram.

STFT Parameters

- **n_fft:** Window size for FFT (typically 512-2048 samples)
 - **hop_length:** Step size between windows (typically $n\_fft/4$)
 - **win_length:** Actual window size (can differ from n_fft)
- Larger windows give better frequency resolution but worse time resolution (uncertainty principle).

4.2.2 Computing Spectrograms with Python

```
1 import tensorflow as tf
2
3 # Compute spectrogram using TensorFlow
4 def compute_spectrogram(audio, sample_rate=16000):
5     """Convert audio waveform to spectrogram."""
6     # STFT parameters
7     frame_length = int(sample_rate * 0.025) # 25ms window
8     frame_step = int(sample_rate * 0.010)  # 10ms hop
9
```

```

10     # Compute STFT
11     stft = tf.signal.stft(
12         audio,
13         frame_length=frame_length,
14         frame_step=frame_step,
15         fft_length=512
16     )
17
18     # Convert complex numbers to magnitude
19     spectrogram = tf.abs(stft)
20
21     # Convert to log scale (dB) for better visualization
22     spectrogram = tf.math.log(spectrogram + 1e-6)
23
24     return spectrogram

```

Understanding the Code

- **Lines 6–7:** 25ms windows with 10ms hops are standard for speech
- **Lines 10–15:** `tf.signal.stft` computes the Short-Time Fourier Transform
- **Line 18:** We only care about magnitude, not phase
- **Line 21:** Log scale compresses the dynamic range for better neural network training

```

24 # Compute and display spectrogram
25 spec = compute_spectrogram(audio)
26 print(f"Spectrogram shape: {spec.shape}")
27
28 # Plot
29 plt.figure(figsize=(12, 4))
30 plt.imshow(spec.numpy().T, aspect='auto', origin='lower', cmap='viridis')
31 plt.colorbar(label='Log Magnitude')
32 plt.xlabel('Time Frame')
33 plt.ylabel('Frequency Bin')
34 plt.title('Spectrogram of "Yes"')
35 plt.show()

```

Expected Output

Spectrogram shape: (98, 257)

98 time frames $\times$ 257 frequency bins = 25,186 values. Much more manageable than 16,000 raw samples, and captures frequency information!

Experiment: Compare Words

Record or load audio files for “yes”, “no”, and “stop”. Generate spectrograms for each and compare:

1. Which frequency ranges show the most energy?

2. Can you visually distinguish the words?
3. What happens to the spectrogram if you speak louder?

4.3 The Mel Scale and MFCCs

Regular spectrograms treat all frequencies equally, but human hearing doesn't work that way. We're much better at distinguishing low frequencies than high frequencies.

The Mel Scale

The **Mel scale** is a perceptual scale where equal distances sound equally different to humans. On the Mel scale:

- 500 Hz and 1000 Hz are as perceptually different as...
- 2000 Hz and 4000 Hz

This logarithmic relationship matches how our ears actually work.

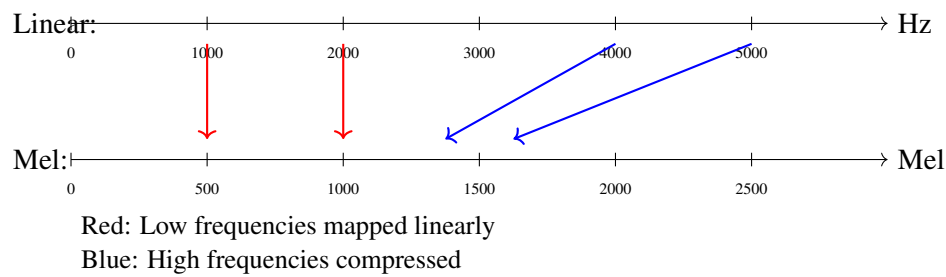


Figure 4.3: The Mel scale compresses high frequencies, matching human perception. Equal Mel distances represent equal perceived pitch differences.

4.3.1 Mel Spectrograms

A **Mel spectrogram** applies the Mel scale to a regular spectrogram:

```

1 def compute_mel_spectrogram(audio, sample_rate=16000, n_mels=40):
2     """Convert audio to Mel spectrogram."""
3     # First compute regular spectrogram
4     frame_length = int(sample_rate * 0.025)
5     frame_step = int(sample_rate * 0.010)
6
7     stft = tf.signal.stft(audio, frame_length, frame_step, fft_length=512)
8     spectrogram = tf.abs(stft)
9
10    # Create Mel filterbank
11    num_spectrogram_bins = spectrogram.shape[-1]
12    linear_to_mel_weight_matrix = tf.signal.linear_to_mel_weight_matrix(
13        n_mels,
14        num_spectrogram_bins,
15        sample_rate,
16        lower_edge_hertz=80.0,
17        upper_edge_hertz=7600.0
18    )
19
20    # Apply Mel filterbank

```

```

21 mel_spectrogram = tf.tensordot(spectrogram, linear_to_mel_weight_matrix, 1)
22 mel_spectrogram = tf.math.log(mel_spectrogram + 1e-6)
23
24 return mel_spectrogram

```

Code Walkthrough

- **Lines 11–17:** Create triangular filters spaced according to Mel scale
- **n_mels=40:** Use 40 Mel bands (common for speech)
- **lower/upper_edge_hertz:** Focus on speech frequencies (80-7600 Hz)
- **Line 20:** Matrix multiply applies all filters at once

Expected Output

Mel spectrogram shape: (98, 40)

Now we have only 40 frequency bins instead of 257, reducing our feature size while preserving perceptually important information!

4.3.2 Mel-Frequency Cepstral Coefficients (MFCCs)

MFCCs go one step further by applying the Discrete Cosine Transform (DCT) to decorrelate the Mel spectrogram:

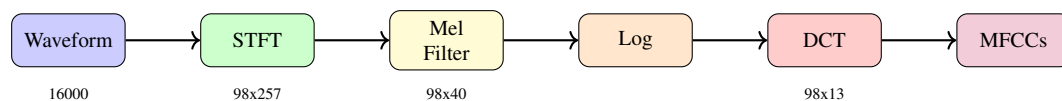


Figure 4.4: MFCC extraction pipeline. Each step reduces and transforms the data while preserving speech-relevant information.

```

1 def compute_mfccs(audio, sample_rate=16000, n_mfcc=13):
2     """Extract MFCCs from audio."""
3     # Compute Mel spectrogram first
4     mel_spec = compute_mel_spectrogram(audio, sample_rate, n_mels=40)
5
6     # Apply DCT to get MFCCs
7     mfccs = tf.signal.mfccs_from_log_mel_spectrograms(mel_spec)
8
9     # Keep only first n_mfcc coefficients
10    mfccs = mfccs[..., :n_mfcc]
11
12    return mfccs
13
14 # Example usage
15 mfccs = compute_mfccs(audio)
16 print(f"MFCC shape: {mfccs.shape}")

```

Expected Output

MFCC shape: (98, 13)

Just $98 \times 13 = 1,274$ features for 1 second of audio! This is what we'll feed to our neural network.

Why MFCCs Work for Edge Devices

MFCCs are ideal for keyword spotting on microcontrollers:

- **Compact:** 13 coefficients vs. 257 frequency bins
- **Decorrelated:** DCT removes redundancy, helping small models learn efficiently
- **Robust:** Less sensitive to volume changes than raw spectrograms
- **Fast:** Can be computed with fixed-point arithmetic on MCUs

Check Your Understanding

1. Why does the Mel scale compress high frequencies?
2. What is the purpose of the DCT in MFCC extraction?
3. How does $n\_mfcc=13$ compare to $n\_mels=40$ in terms of information compression?

4.3.3 Method Selection Guide: Choosing Audio Features

Different audio feature representations offer different trade-offs for edge deployment. Here's how to choose the right approach for your application.

Choosing Audio Features for Edge Deployment

Use the following decision tree to select the best feature extraction method:

1. **Is your device memory-constrained (<256KB RAM)?**
 - Yes → Use **MFCCs** (13 coefficients = minimal memory)
 - No → Continue to question 2
2. **Do you need real-time processing (<50ms latency)?**
 - Yes → Use **Mel Spectrograms** or **MFCCs** (efficient computation)
 - No → Continue to question 3
3. **What is your application domain?**
 - Speech/voice commands → **MFCCs** (optimized for human speech)
 - Environmental sounds → **Mel Spectrograms** (preserves more detail)
 - Music/complex audio → **Raw Spectrograms** (full frequency information)
4. **Do you have sufficient training data?**
 - Limited data (<1000 samples) → Use **MFCCs** with traditional ML (less prone to overfitting)
 - Large dataset (>10,000 samples) → Can use **Raw Spectrograms** with deep learning

Table 4.2: Audio Feature Comparison for Edge Keyword Spotting

Method	Memory	Accuracy	Latency	Best For
Raw Waveform	Very High	Low	Fast	Not recommended for edge
STFT Spectrogram	High (98×257)	High 90-95%	Medium 10-20ms	Desktop/cloud with GPU Complex audio analysis
Mel Spectrogram	Medium (98×40)	High 88-93%	Medium 8-15ms	Raspberry Pi class devices Environmental sound, music
MFCCs	Low (98×13)	High 85-92%	Fast 5-10ms	Microcontrollers (Arduino, ESP32) Speech/voice commands
Log Mel Energies	Low (98×10)	Medium 80-88%	Fast 3-8ms	Ultra-low power applications Wake word detection

Why MFCCs Dominate Edge Audio ML

MFCCs are the standard for edge keyword spotting because they:

- **Compress aggressively:** $98 \times 13 = 1,274$ features vs $98 \times 257 = 25,186$ for raw spectrograms
- **Match human perception:** Mel scale and cepstral coefficients align with how we hear speech
- **Decorrelate features:** DCT removes redundancy, helping small models learn efficiently
- **Are robust:** Less sensitive to volume changes and noise than raw spectrograms
- **Compute efficiently:** Can use fixed-point arithmetic on microcontrollers

For a 1-second audio clip at 16kHz: Raw waveform = 16,000 samples; MFCCs = 1,200 values (13× reduction!)

💡 Feature Size Trade-offs

Consider a keyword spotting model on Arduino Nano 33 BLE (256KB SRAM):

- **MFCCs (98×13):** $1,274 \text{ floats} \times 4 \text{ bytes} = 5.1 \text{ KB}$ input buffer
- **Mel Spectrograms (98×40):** $3,920 \text{ floats} \times 4 \text{ bytes} = 15.7 \text{ KB}$ input buffer
- **Raw Spectrograms (98×257):** $25,186 \text{ floats} \times 4 \text{ bytes} = 100.7 \text{ KB}$ input buffer

Remember: You also need space for the model (20KB), tensor arena (30KB), and audio buffer (32KB). Which feature representation fits?

4.4 Building a Keyword Spotting Model

Now let's build a neural network that can classify audio based on MFCC features.

KWS Model Requirements

A keyword spotting model for edge devices must be:

- **Small:** Fit in <100KB for MCUs
- **Fast:** Process audio in real-time (<100ms inference)
- **Accurate:** Minimize false positives and negatives
- **Robust:** Work across different speakers, accents, background noise

4.4.1 Dataset: Google Speech Commands v2

Before we train our model, we need a dataset. The **Google Speech Commands dataset** is the standard benchmark for keyword spotting research and development. It provides thousands of utterances of common voice commands, recorded by different speakers in various acoustic environments.

Dataset Overview

Google Speech Commands v2 contains:

- **Size:** 2.3 GB (compressed)
- **Utterances:** 105,829 audio files
- **Keywords:** 35 words (30 commands + 5 auxiliary words)
- **Speakers:** 2,618 unique voices
- **Format:** 1-second WAV files, 16 kHz, mono
- **License:** Creative Commons BY 4.0

Table 4.3: Speech Commands v2 Dataset Statistics

Property	Details
Core Commands (10)	yes, no, up, down, left, right, on, off, stop, go
Extended Commands (20)	zero–nine, bed, bird, cat, dog, happy, house, marvin, sheila, tree, wow
Auxiliary (5)	backward, forward, follow, learn, visual
Special Classes	_silence_ (background noise), _unknown_ (other words)
Audio Format	16 kHz sample rate, 16-bit PCM, mono channel
Duration	Exactly 1 second per clip (16,000 samples)
Samples per Word	1,500–4,000 depending on word
Train/Val/Test Split	Pre-defined splits included

Loading the Dataset

There are two main approaches to load the dataset:

Option 1: TensorFlow Datasets (Recommended)

The easiest method uses TensorFlow Datasets, which handles downloading and preprocessing automatically:

```

1 import tensorflow_datasets as tfds
2 import tensorflow as tf
3
4 # Load Speech Commands dataset (downloads automatically)
5 ds, info = tfds.load('speech_commands', with_info=True, as_supervised=True)
6
7 # Access train/validation/test splits
8 train_ds = ds['train']
9 val_ds = ds['validation']
10 test_ds = ds['test']
11
12 # Dataset info
13 print("Available words:", info.features['label'].names)
```

```

14 print("Number of classes:", info.features['label'].num_classes)
15 print("Total examples:", info.splits['train'].num_examples)

```

Expected Output

```

Available words: ['down', 'go', 'left', 'no', 'off', 'on',
                  'right', 'stop', 'up', 'yes', '_silence_', '_unknown_']
Number of classes: 12
Total examples: 84,843

```

By default, TensorFlow Datasets loads the 12-class subset: 10 core commands plus silence and unknown.

Option 2: Manual Download

For full control or offline environments, download manually:

```

1 import urllib.request
2 import tarfile
3
4 # Download Speech Commands v2 (2.3 GB)
5 url = 'http://download.tensorflow.org/data/speech_commands_v0.02.tar.gz'
6 filename = 'speech_commands_v0.02.tar.gz'
7
8 print(f"Downloading {filename} (2.3 GB)...")
9 urllib.request.urlretrieve(url, filename)
10
11 # Extract
12 print("Extracting...")
13 with tarfile.open(filename, 'r:gz') as tar:
14     tar.extractall('speech_commands/')
15
16 print("Dataset ready at: speech_commands/")

```

Download Time and Storage

- **Size:** 2.3 GB compressed, 3 GB extracted
- **Download time:** 5–30 minutes depending on connection
- **Storage:** Ensure you have at least 6 GB free space
- **Alternative:** Use TensorFlow Datasets which caches efficiently

If bandwidth is limited, start with the smaller v1 (1.4 GB) or use `tfds.load()` which downloads only what you need.

Alternative: Synthetic Dataset for Quick Testing

For rapid prototyping or when bandwidth is limited, you can generate synthetic audio data:

```

1 def generate_synthetic_keyword(keyword_type='yes', sr=16000):
2     """Generate synthetic audio that mimics keyword characteristics."""
3     t = np.linspace(0, 1.0, sr)
4
5     if keyword_type == 'yes':

```



```

6         # Rising pitch pattern
7         freq = 200 + 300 * np.sin(np.pi * t)
8         audio = np.sin(2 * np.pi * freq * t)
9         envelope = np.exp(-((t - 0.5)**2) / 0.1)
10        audio *= envelope
11    elif keyword_type == 'no':
12        # Falling pitch pattern
13        freq = 300 - 150 * t
14        audio = np.sin(2 * np.pi * freq * t)
15        envelope = np.exp(-((t - 0.3)**2) / 0.15)
16        audio *= envelope
17    else:
18        # Background noise
19        audio = 0.1 * np.random.randn(len(t))
20
21    # Add noise
22    audio += 0.02 * np.random.randn(len(audio))
23    return audio.astype(np.float32)

```

Limitations of Synthetic Data

Synthetic audio is useful for:

- ✓ Testing your code pipeline quickly
- ✓ Verifying model architecture and training loop
- ✓ Debugging without large downloads

But synthetic data is NOT suitable for:

- × Deployment to real devices
- × Measuring actual model performance
- × Research or production systems

Always test on real Speech Commands data before deployment!

Understanding the Dataset Structure

Once loaded, each example consists of:

```

1 # Iterate through dataset
2 for audio, label in train_ds.take(1):
3     print(f"Audio shape: {audio.shape}")
4     print(f"Audio dtype: {audio.dtype}")
5     print(f"Label: {label.numpy()}")
6     print(f"Sample rate: 16000 Hz (fixed)")
7     print(f"Duration: {len(audio) / 16000:.2f} seconds")

```

Expected Output

```

Audio shape: (16000,)
Audio dtype: <dtype: 'float32'>
Label: 3
Sample rate: 16000 Hz (fixed)
Duration: 1.00 seconds

```

Each audio clip is exactly 1 second (16,000 samples) at 16 kHz, normalized to [-1, 1]. Labels

are integers 0–11.

Why Speech Commands is Ideal for Edge KWS

1. **Fixed duration:** All clips are exactly 1 second, simplifying model input
2. **Standard sample rate:** 16 kHz is optimal for speech on edge devices
3. **Speaker diversity:** 2,618 speakers ensure model generalizes across voices
4. **Background noise included:** `_silence_` class trains noise rejection
5. **Unknown words:** `_unknown_` class prevents false positives
6. **Pre-split:** Train/validation/test splits prevent data leakage
7. **Widely used:** Standardized benchmark allows comparing your results

4.4.2 Model Architecture: DS-CNN

Depthwise Separable CNNs (DS-CNN) are the standard architecture for edge keyword spotting:

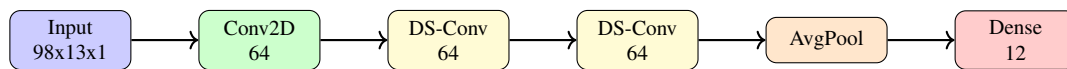


Figure 4.5: DS-CNN architecture for keyword spotting. Depthwise separable convolutions reduce parameters while maintaining accuracy.

```

1 from tensorflow.keras import layers, models
2
3 def create_kws_model(input_shape=(98, 13, 1), num_classes=12):
4     """Create a small CNN for keyword spotting."""
5     model = models.Sequential([
6         # First conv block
7         layers.Conv2D(64, (3, 3), strides=(2, 2), padding='same',
8                       activation='relu', input_shape=input_shape),
9         layers.BatchNormalization(),
10        layers.Dropout(0.2),
11
12        # Depthwise separable conv blocks
13        layers.SeparableConv2D(64, (3, 3), strides=(2, 2), padding='same',
14                               activation='relu'),
15        layers.BatchNormalization(),
16        layers.Dropout(0.2),
17
18        layers.SeparableConv2D(64, (3, 3), strides=(2, 2), padding='same',
19                               activation='relu'),
20        layers.BatchNormalization(),
21
22        # Global pooling and classification
23        layers.GlobalAveragePooling2D(),
24        layers.Dense(num_classes, activation='softmax')
25    ])
26    return model
27
28 model = create_kws_model()
29 model.summary()
  
```

Expected Output

Total params: 18,316

Trainable params: 17,932

Only 18K parameters! This model will be <20KB when quantized to int8.

Why Depthwise Separable Convolutions?

Standard convolution: $3 \times 3 \times 64 \times 64 = 36,864$ parameters. Depthwise separable: $(3 \times 3 \times 64) + (1 \times 1 \times 64 \times 64) = 4,672$ parameters. That's an 8x reduction with minimal accuracy loss!

4.4.3 Training the Model

```

1 # Compile the model
2 model.compile(
3     optimizer='adam',
4     loss='sparse_categorical_crossentropy',
5     metrics=['accuracy']
6 )
7
8 # Train
9 history = model.fit(
10     train_ds,
11     validation_data=val_ds,
12     epochs=20,
13     callbacks=[
14         tf.keras.callbacks.EarlyStopping(patience=5, restore_best_weights=True)
15     ]
16 )

```

Expected Training Output

Epoch 20/20 - accuracy: 0.9312 - val_accuracy: 0.9021

After 20 epochs, expect 90%+ accuracy on the validation set.

4.4.4 Evaluating with Confusion Matrix

A confusion matrix reveals which words the model confuses:

```

1 import seaborn as sns
2 from sklearn.metrics import confusion_matrix
3
4 # Get predictions
5 y_true, y_pred = [], []
6 for mfccs, labels in test_ds:
7     predictions = model.predict(mfccs)
8     y_pred.extend(np.argmax(predictions, axis=1))
9     y_true.extend(labels.numpy())

```

```

10
11 # Plot confusion matrix
12 cm = confusion_matrix(y_true, y_pred)
13 plt.figure(figsize=(10, 8))
14 sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
15             xticklabels=info.features['label'].names,
16             yticklabels=info.features['label'].names)
17 plt.xlabel('Predicted')
18 plt.ylabel('True')
19 plt.title('Keyword Spotting Confusion Matrix')
20 plt.show()

```

Reading the Confusion Matrix

- **Diagonal:** Correct predictions (higher is better)
 - **Off-diagonal:** Confusion between classes
 - **Look for patterns:** Which words get confused with each other?
- Common confusions: “no”/“go”, “up”/“off”, “left”/“right”

💡 Why Do Certain Words Get Confused?

Look at your confusion matrix and identify the most common misclassifications. Then think:

1. What phonetic sounds do “no” and “go” share? What’s different?
2. How would the spectrograms differ between these words?
3. If you were designing a smart home system, would you choose “no” and “go” as commands? Why or why not?
4. What alternative words might be better choices for a 2-word command vocabulary?

Hint: Words with distinct starting consonants (like “yes” vs “stop”) are easier to distinguish.

Experiment: Custom Keywords

Modify the training to detect your own keywords:

1. Choose 2–4 words from the dataset (e.g., “go”, “stop”)
2. Filter the dataset to include only those words + “unknown” + “silence”
3. Train a smaller model (fewer filters)
4. Compare accuracy to the full 12-class model

Check Your Understanding

1. Why do we use depthwise separable convolutions instead of regular convolutions?
2. What does the confusion matrix tell us about model performance?
3. How many parameters should a KWS model have for MCU deployment?

4.5 Deploying Keyword Spotting Models

Let’s deploy our trained model across all three execution tiers.

Level 1: Notebook Testing

Test the model with recorded audio in your notebook:

```

1 # Record audio (in Colab)
2 audio, sr = get_audio() # Custom recording function
3
4 # Preprocess and predict
5 mfccs = compute_mfccs(audio)
6 mfccs = tf.expand_dims(mfccs, 0) # Add batch dimension
7 mfccs = tf.expand_dims(mfccs, -1) # Add channel dimension
8
9 prediction = model.predict(mfccs)
10 predicted_class = np.argmax(prediction)
11 confidence = np.max(prediction)
12
13 print(f"Predicted: {labels[predicted_class]} ({confidence*100:.1f}%")

```

Verification checklist:

- ☐ Model achieves >90% accuracy on test set
- ☐ Correctly classifies your recorded voice commands
- ☐ Handles silence and unknown words appropriately

Level 2: Raspberry Pi Deployment

Convert to TFLite and deploy on Raspberry Pi:

```

1 # Convert to TFLite with quantization
2 converter = tf.lite.TFLiteConverter.from_keras_model(model)
3 converter.optimizations = [tf.lite.Optimize.DEFAULT]
4
5 def representative_dataset():
6     for mfccs, _ in train_ds.take(100):
7         yield [mfccs]
8
9 converter.representative_dataset = representative_dataset
10 converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
11 tflite_model = converter.convert()
12
13 # Save
14 with open('kws_model_int8.tflite', 'wb') as f:
15     f.write(tflite_model)
16 print(f"Model size: {len(tflite_model) / 1024:.1f} KB")

```

Expected: Model size <20KB after int8 quantization.

Pi deployment steps:

1. Install tflite-runtime and pyaudio
2. Set up USB microphone with ALSA
3. Run inference loop with 1-second sliding window
4. Measure latency: target <50ms per prediction

Building On: LAB12 - Real-Time Streaming Audio

This lab introduces keyword spotting on 1-second audio clips. In LAB12 you'll extend this to continuous real-time audio streams:

- Implement sliding window processing for always-on detection
- Handle overlapping audio windows efficiently
- Build streaming pipelines that process audio as it arrives
- Optimize buffering and latency for real-time response

The KWS model you build here forms the foundation for production-ready streaming systems.

Level 3: Arduino Nano 33 BLE Sense

Deploy to microcontroller for always-on operation:

Step 1: Convert model to C array

```
1 xxd -i kws_model_int8.tflite > model_data.h
```

Step 2: Arduino sketch structure

```
1 #include <TensorFlowLite.h>
2 #include <PDM.h> // Built-in microphone
3 #include "model_data.h"
4
5 // Audio buffer
6 int16_t audio_buffer[16000];
7
8 void loop() {
9     // Capture 1 second of audio
10    if (samples_read >= 16000) {
11        // Extract MFCC features
12        extract_mfccs(audio_buffer, mfcc_buffer);
13
14        // Run inference
15        interpreter->Invoke();
16
17        // Get result
18        int8_t* output = interpreter->output(0)->data.int8;
19        int predicted_class = argmax(output, 12);
20
21        if (output[predicted_class] > threshold) {
22            Serial.println(labels[predicted_class]);
23        }
24    }
25 }
```

Memory requirements:

- Model: 18KB flash
- Tensor arena: 30KB RAM
- Audio buffer: 32KB RAM
- Total: fits in Arduino Nano 33 BLE (256KB RAM, 1MB flash)

4.5.1 Deployment Decision Matrix

Table 4.4: KWS Deployment Comparison

Aspect	Level 1	Level 2	Level 3
Latency	<100ms	<50ms	<200ms
Power	>10W	~5W	<10mW
Always-on	No	Possible	Yes
Model size	Any	<1MB	<50KB
Use case	Development	Smart speaker	Wake word

Tier Progression Checklist

Before moving to the next tier, verify:

- ☐ **L1 → L2:** Model achieves target accuracy, TFLite conversion successful
- ☐ **L2 → L3:** Model size fits in MCU flash, latency acceptable
- ☐ **L3 Complete:** Keyword detection works with live audio input

Chapter Summary

Key Takeaways

- **Audio preprocessing** converts raw waveforms to compact features (spectrograms, MFCCs)
- **The Mel scale** matches human perception, improving model efficiency
- **MFCCs** provide compact, decorrelated features ideal for edge deployment
- **DS-CNN models** achieve high accuracy with minimal parameters (<20K)
- **Quantization** reduces model size to <20KB for MCU deployment
- **Three-tier deployment** allows progressive optimization and testing

Self-Assessment Questions

Test your understanding before continuing:

1. Why is a 16kHz sample rate commonly used for keyword spotting instead of 44.1kHz (CD quality)? What are the trade-offs?
2. Explain how the Mel scale differs from the linear frequency scale and why it's better suited for speech recognition on edge devices.
3. A 1-second audio clip at 16kHz contains 16,000 samples. After converting to 13 MFCCs with 40ms windows and 20ms stride, approximately how many MFCC vectors do you get? Show your calculation.
4. Why do depthwise-separable convolutions (DS-CNN) use fewer parameters than standard convolutions while maintaining similar accuracy? What is the parameter reduction factor?
5. What are the main challenges of deploying a keyword spotting model on a microcontroller versus a Raspberry Pi? Consider memory, compute, and power constraints.

4.6 Guided Experiments

Experiment 1: MFCC Parameter Tuning

Hypothesis: The number of MFCC coefficients affects model accuracy and inference speed. We expect more MFCCs to capture more detail but increase computation time.

Procedure:

1. Use the Speech Commands dataset and keyword spotting model from this chapter
2. Train models with different `n_mfcc` values: 8, 13, 20, 26, 40
3. Keep other parameters constant (sample rate 16kHz, window size 40ms)
4. Train for same number of epochs (e.g., 10 epochs)
5. Measure training time, model size, and test accuracy

Data Collection:

n_mfcc	Input Size	Model Size (KB)	Test Accuracy (%)	Inference Time (ms)
8				
13				
20				
26				
40				

Analysis Questions:

- At what point does increasing `n_mfcc` stop improving accuracy?
- What is the relationship between `n_mfcc` and model size?
- For an Arduino with 256KB RAM, which `n_mfcc` value is most practical?
- How does inference time scale with `n_mfcc`?

Experiment 2: Audio Window Size Comparison

Hypothesis: Window size affects the time-frequency resolution trade-off. Larger windows provide better frequency resolution but worse time resolution.

Procedure:

1. Use the same keyword spotting model architecture
2. Vary the window size: 20ms, 30ms, 40ms, 50ms, 60ms
3. Keep `n_mfcc`=13 and `hop_length`=`window_size`/2
4. Train each model for 10 epochs
5. Evaluate on test set and record confusion matrix for each

Data Collection:

Window Size (ms)	Frames per Second	Test Accuracy (%)	Confusion Rate (%)
20			
30			
40			
50			
60			

Analysis Questions:

- Which window size achieves the best accuracy for keyword spotting?
- Do shorter windows help distinguish similar-sounding words?
- What happens to model input size as window size increases?

- For real-time edge deployment, what is the latency impact of each window size?

4.7 Troubleshooting Guide

⚠ Audio File Not Loading or Corrupted

Symptoms: `librosa.load()` fails with error, or loaded audio is all zeros/silence

Cause: Incorrect file format, unsupported codec, or corrupted audio file

Solution:

1. Verify file format: Use .wav files (PCM format) for best compatibility
2. Convert audio if needed: `ffmpeg -i input.mp3 -ar 16000 -ac 1 output.wav`
3. Check sample rate: `sr, audio = librosa.load('file.wav', sr=16000)`
4. Verify audio loaded: `print(audio.shape, audio.min(), audio.max())`
5. For Arduino: Ensure audio is mono (single channel) and 16kHz sample rate

⚠ Spectrogram Looks Wrong or Empty

Symptoms: Spectrogram visualization shows all black, all white, or doesn't match expected pattern

Cause: Incorrect FFT parameters, logarithmic scaling issues, or audio normalization problems

Solution:

1. Check audio amplitude: Normalize to [-1, 1] range using `audio = audio / np.max(np.abs(audio))`
2. Adjust FFT window size: Try `n_fft=2048` for good frequency resolution
3. Use logarithmic scale for visualization: `librosa.power_to_db(S, ref=np.max)`
4. Verify hop length: Should be `n_fft // 4` typically (e.g., 512 for `n_fft=2048`)
5. Check plot range: Use `plt.colorbar()` to see actual values

⚠ MFCC Features Have Wrong Shape

Symptoms: Model expects (13, 100) but MFCC shape is (13, 87) or different

Cause: Audio duration varies, or MFCC extraction parameters don't match training setup

Solution:

1. Pad/trim audio to fixed length: `audio = librosa.util.fix_length(audio, size=16000)`
2. Set consistent hop length: `mfcc = librosa.feature.mfcc(n_mfcc=13, hop_length=512)`
3. Reshape or pad MFCC: `mfcc_padded = np.zeros((13, 100)); mfcc_padded[:, :mfcc.shape[1]] = mfcc`
4. Use time steps parameter: Calculate expected frames based on audio length
5. Verify model input shape: `model.summary()` shows expected (batch, time, features)

⚠ Microphone Not Detected on Arduino

Symptoms: PDM microphone initialization fails or audio readings are all zeros

Cause: PDM library not installed, incorrect pin configuration, or hardware issue

Solution:

1. Install PDM library: Arduino IDE > Library Manager > search “PDM” > Install
2. Verify board: Arduino Nano 33 BLE Sense has built-in PDM microphone
3. Check initialization: `if (!PDM.begin(1, 16000)) { Serial.println("PDM init failed"); }`
4. Set gain: `PDM.setGain(20);` - try values 0-80
5. Test with example: File > Examples > PDM > PDMSerialPlotter
6. For ESP32: Use I2S microphone instead of PDM

⚠ Model Accuracy Low on Real-World Audio

Symptoms: Model works well on training data but fails on live microphone input

Cause: Training/test mismatch (background noise, recording quality, speaker differences)

Solution:

1. Add background noise augmentation during training
2. Normalize audio consistently: Apply same preprocessing to training and inference
3. Increase training data diversity: Different speakers, environments, distances
4. Use data augmentation: Pitch shifting, time stretching, adding noise
5. Implement voice activity detection (VAD) to filter out silence
6. Adjust confidence threshold: Require higher probability (e.g., >0.8) for positive detection

Case Study: Smart Speaker Wake Word Detection

Context: A consumer electronics company developing a new smart speaker needed to implement always-on wake word detection for the phrase “Hey Assistant.” The wake word detector had to run continuously on a low-power ARM Cortex-M4 microcontroller, consuming less than 5mW while maintaining false positive rates below 1 per 24 hours and false negative rates under 5%.

Challenge: The initial DS-CNN model trained on clean audio achieved 95% accuracy in the lab but performed poorly in real homes. False positives from TV audio, background conversations, and environmental noise made the device unusable. The model also consumed 12mW continuously—far exceeding the power budget. Additionally, the 500KB model size exceeded the MCU’s 256KB Flash memory.

Solution: The team implemented a comprehensive optimization pipeline:

- Applied aggressive data augmentation: mixed Speech Commands dataset with 10,000 hours of diverse background noise (music, TV, street sounds, household appliances)
- Reduced MFCC feature dimensions from 40 to 13 coefficients and lowered sample rate from 16kHz to 8kHz
- Redesigned network architecture: replaced 4-layer DS-CNN with a 2-layer depthwise-separable network optimized for keyword spotting
- Applied int8 quantization with QAT, reducing model from 480KB to 115KB
- Implemented two-stage detection: lightweight screener (20KB) runs continuously, full model (115KB) activates only on potential matches

Results:

- Model size: 480KB → 115KB (4.2x reduction, fits in Flash)
- Power consumption: 12mW → 3.8mW average (two-stage architecture)
- False positive rate: 0.3 per 24 hours (97% improvement)
- False negative rate: 3.2% (target: <5%)
- Wake word latency: 220ms from utterance start to activation

- Inference time: 38ms per 1-second audio window

Lessons Learned:

- Real-world noise diversity is critical—augment training data with actual target environment recordings, not just synthetic noise
- Two-stage detection architectures dramatically reduce average power: run a tiny screener continuously, activate full model only when needed
- MFCC feature reduction (40 → 13 coefficients) often improves generalization by reducing overfitting to training acoustics
- False positive rate is more important than raw accuracy—users tolerate occasional missed detections but frequent false activations are unacceptable
- Test on diverse speakers and accents from day one—models overfit to dominant demographic groups in training data
- PDM microphone quality varies significantly—validate on actual hardware early to catch sampling artifacts and noise floors

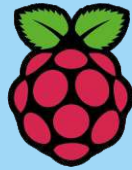
Exercises

1. **[Basic] Basic: Feature Exploration:** Record “yes” and “no” samples. Generate waveforms, spectrograms, and MFCCs for each. Document the visual differences.
2. **[Intermediate] Intermediate: Custom Keywords:** Train a model to recognize two custom words of your choice. Report accuracy and confusion patterns.
3. **[Intermediate] Intermediate: Model Size vs Accuracy:** Train models with 32, 64, and 128 filters. Plot the trade-off between model size and accuracy.
4. **[Intermediate] Intermediate: Noise Robustness:** Add background noise to test samples. Measure how accuracy degrades. Try data augmentation during training.
5. **[Intermediate] Intermediate: MFCC Parameter Analysis:** Systematically vary MFCC parameters and analyze their impact on model accuracy and inference latency. Train baseline model with standard parameters ($n\_mfcc=13$, $window=40ms$, $hop=20ms$). Then vary one parameter at a time: (a) n_mfcc values: 8, 13, 20, 26, 40; (b) window sizes: 20ms, 30ms, 40ms, 50ms; (c) Mel filterbank sizes: 20, 40, 60, 80 bands. For each configuration, measure test accuracy, model size (KB), and inference time (ms). Create visualizations comparing parameter choices. Identify the “sweet spot” configuration that balances accuracy and efficiency for Arduino deployment. *Extension:* Visualize how spectrograms differ between parameter settings for the same audio sample.
6. **[Challenge] Challenge: Accuracy vs Latency Trade-offs:** Design and evaluate three different KWS model architectures to explore the accuracy-latency Pareto frontier. Implement: (a) **Tiny model:** 1-layer DS-CNN with 32 filters, targeting <10KB and <20ms latency; (b) **Balanced model:** 2-layer DS-CNN with 64 filters (baseline from chapter); (c) **Accurate model:** 4-layer DS-CNN with 128 filters, maximizing accuracy. For each model, measure: test accuracy, model size (float32 and int8), inference time on Raspberry Pi/Arduino, and RAM usage. Plot accuracy vs latency and accuracy vs model size curves. Analyze when accuracy gains justify increased latency. *Real-world scenario:* Which architecture would you deploy for: (1) battery-powered doorbell (ultra-low power), (2) smart home hub (balanced), (3) accessibility device requiring highest accuracy?
7. **[Challenge] Challenge: Custom Wake Word Detection System:** Design and implement a complete two-stage wake word detection system for a custom phrase (e.g., “Hey Lab” or “Computer”). Stage 1 requirements: Create a lightweight screener model (<20KB) that runs continuously with very low false negatives (<1%) but may have higher false positives.

Use simple features (e.g., 8 MFCCs) and minimal architecture. Stage 2 requirements: Build a full verification model (can be larger, 50-100KB) that activates only when Stage 1 triggers, with <5% false positive rate. Implementation: (a) Collect or synthesize training data for your wake word; (b) Include negative examples from Speech Commands “unknown” class; (c) Implement two-stage inference pipeline with confidence thresholds; (d) Measure average power consumption (hint: Stage 1 runs continuously, Stage 2 activates 1% of time). *Evaluation criteria:* False positive rate (<1 per hour), false negative rate (<5%), end-to-end latency (<500ms), and estimated battery life on Arduino with 1000mAh battery.

- **Further Resources.**
- Speech Commands Dataset: https://www.tensorflow.org/datasets/catalog/speech_commands
 - Librosa Audio Library: <https://librosa.org/>
 - TensorFlow Audio Tutorial: https://www.tensorflow.org/tutorials/audio/simple_audio

— **Credits.** We have taken this code segment from the online course: *Edx Tiny Machine Learning (2021-22)*. Since we may have updated the code to reflect any changes made to the underline frameworks where necessary. Credits should go to the original authors of this tutorial.



RaspberryPi



ThingsBoard

5. Embedded Systems Deployment with TensorFlow

Learning Objectives

After completing this lab, you will be able to:

- Understand microcontroller memory constraints (Flash vs SRAM)
- Convert TensorFlow Lite models to C arrays for embedded deployment
- Configure the TensorFlow Lite Micro interpreter and tensor arena
- Deploy and run ML models on Arduino and ESP32 microcontrollers
- Profile memory usage and inference latency on real hardware
- Integrate sensor input with ML inference pipelines

Interactive Notebook

Jupyter Notebook: LAB05_deployment.ipynb

[Open in Google Colab](#) | [View on GitHub](#) | [Companion Website](#)

You've trained models, quantized them, and tested them in Python. Now comes the final step: running your model on actual edge hardware. This is where theory meets reality—your model must fit in kilobytes of memory and run in milliseconds.

Deploying to microcontrollers is fundamentally different from cloud or desktop deployment. There's no operating system, no file system, and no dynamic memory allocation. Everything must be carefully planned at compile time. The good news is that TensorFlow Lite Micro provides a robust framework that handles most of the complexity.

In this chapter, you'll learn to deploy models to Arduino and ESP32 boards, profile their performance, and integrate them with real sensors. By the end, you'll have a complete edge ML pipeline running on hardware you can hold in your hand.

Prerequisites

Before starting this lab, ensure you:

- Completed LAB03 (Quantization): TFLite conversion and model optimization
- Completed LAB04 (Keyword Spotting): understanding of model deployment concepts
- Basic C/C++ syntax: variables, functions, arrays, pointers
- Arduino IDE 2.0+ installed and configured
- Understanding of binary/hexadecimal number representation

Hardware Requirements

- **Level 1:** Laptop with Arduino IDE 2.0+ installed
- **Level 2:** ESP32 or Raspberry Pi Pico
- **Level 3:** Arduino Nano 33 BLE Sense with built-in sensors

Tensor Arena Too Small = Cryptic Crashes

The most frustrating deployment bug is when `AllocateTensors()` fails with no helpful error message. This almost always means your tensor arena is too small. The arena must hold **all intermediate activations**, not just the model weights. A 20KB model might need 50KB of arena! **Start large (50KB), then reduce** based on `arena_used_bytes()` output.

Three-Tier Execution Model

Tier 1: Jupyter/Colab

Convert models to C arrays, validate TFLite conversion in Python interpreter, verify quantization parameters and tensor shapes

Test TFLite models with `tfLite-runtime`, benchmark inference latency, validate preprocessing pipeline matches training

Deploy TensorFlow Lite Micro models to MCUs, configure tensor arena, integrate sensors, profile Flash and SRAM usage, measure inference time

5.1 Understanding Microcontroller Memory

Before deploying ML models, you must understand how microcontrollers manage memory—it's fundamentally different from PC or cloud computing.

The MCU Memory Model

Microcontrollers have two distinct memory regions:

- **Flash (ROM):** Non-volatile storage for code and constants. Your model weights live here.

- **SRAM (RAM):** Volatile working memory. Your model’s intermediate calculations happen here.
If either is too small, your model won’t run—even if the other has plenty of space.

5.1.1 Memory Architecture

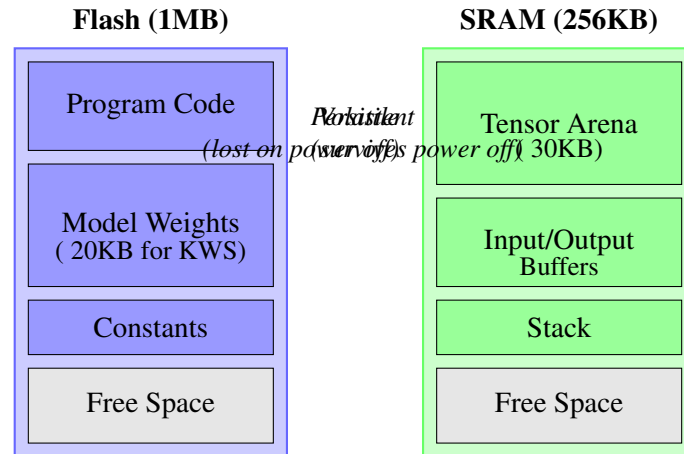


Figure 5.1: Microcontroller memory layout. Model weights are stored in Flash, while inference calculations use SRAM.

5.1.2 Common Microcontroller Specifications

Table 5.1: Memory Specifications for Popular TinyML Boards

Board	Flash	SRAM	Good For
Arduino Nano 33 BLE	1MB	256KB	Audio, motion, small vision
ESP32	4MB	520KB	WiFi apps, larger models
Raspberry Pi Pico	2MB	264KB	General TinyML projects
STM32F4	1MB	192KB	Industrial applications
Arduino Uno (ATmega328)	32KB	2KB	Too small for ML

Model Size vs Working Memory

A 20KB model doesn’t mean you need only 20KB of SRAM! The model *weights* live in Flash, but inference requires *working memory* in SRAM for intermediate tensors. A 20KB model might need 50KB of SRAM to run. Always profile actual memory usage.

Check Your Understanding

1. Where are model weights stored on a microcontroller?
2. Why might a model fit in Flash but fail to run?
3. What happens to SRAM contents when power is removed?

5.1.3 Method Selection Guide: Choosing Your Deployment Target

Different hardware platforms offer different capabilities and constraints. Here's how to choose the right target for your edge ML application.

Choosing a Deployment Target

Use the following decision tree to select the best hardware platform:

1. **What is your power budget?**
 - Battery-powered, ultra-low power (<10mW) → **Arduino Nano 33 BLE**
 - Battery-powered with WiFi (<100mW sleep) → **ESP32 with deep sleep**
 - Wall-powered (>5W acceptable) → **Raspberry Pi 4**
2. **What is your model size?**
 - <100KB, simple patterns → **Arduino Nano 33 BLE**
 - 100KB-500KB, moderate complexity → **ESP32**
 - >500KB, complex models → **Raspberry Pi 4**
3. **Do you need wireless connectivity?**
 - No connectivity needed → **Arduino Nano 33 BLE** (lowest cost/power)
 - WiFi required → **ESP32** (built-in WiFi/BLE)
 - Ethernet or advanced networking → **Raspberry Pi 4**
4. **What is your latency requirement?**
 - <10ms inference → All platforms (with proper quantization)
 - 10-50ms acceptable → **Arduino/ESP32** work well
 - <100ms acceptable → **Raspberry Pi** with TFLite

Table 5.2: Edge Deployment Platform Comparison

Platform	Memory	Power	Speed	Best For
Tier 3: Microcontrollers				
Arduino Uno	32KB Flash 2KB RAM	50mW	16 MHz	Too limited for ML
Arduino Nano 33 BLE	1MB Flash 256KB RAM	5-15mW	64 MHz ARM M4	Audio, motion, sensors BLE connectivity
ESP32	4MB Flash 520KB RAM	20-160mW 10μA sleep	240 MHz Dual Core	WiFi/BLE ML nodes IoT applications
RP2040 (Pico)	2MB Flash 264KB RAM	30mW	133 MHz Dual M0+	General embedded ML Cost-sensitive apps
Tier 2: Single-Board Computers				
Raspberry Pi 4	SD Card 1-8GB RAM	2.5-6W	1.5 GHz Quad A72	Vision, NLP, complex ML Development & testing
Jetson Nano	SD Card 2-4GB RAM	5-10W	1.43 GHz Quad A57 + GPU	Computer vision Object detection
Specialized Accelerators				
Coral USB	N/A (USB) Host RAM	2-3W	Edge TPU 4 TOPS	int8 model acceleration 10-100x speedup

Model Size Requirements by Platform

Arduino Nano 33 BLE (256KB RAM):

- Maximum model size: 100KB (leaves room for tensor arena and buffers)
 - Example: Keyword spotting (20KB model + 30KB arena + 32KB audio buffer = 82KB)
 - Int8 quantization is **essential**
- ESP32 (520KB RAM):**
- Maximum model size: 300KB (more headroom for complex models)
 - Example: Small vision model (150KB model + 200KB arena + 100KB image buffer = 450KB)
 - Can handle float32 for small models, int8 recommended for larger ones
- Raspberry Pi 4 (1-8GB RAM):**
- Maximum model size: Limited by SD card, not RAM (models up to 500MB+)
 - Example: MobileNetV2 image classification (14MB model)
 - Can run float32 models, but int8 still faster

💡 Total Memory Budget

Calculate if your application fits on Arduino Nano 33 BLE (256KB SRAM):

- Model: 20KB (keyword spotting, int8)
- Tensor arena: 30KB (for intermediate activations)
- Input audio buffer: $16,000 \text{ samples} \times 2 \text{ bytes} = 32\text{KB}$
- MFCC features: $98 \text{ frames} \times 13 \text{ coefficients} \times 4 \text{ bytes} = 5.1\text{KB}$
- Stack and system overhead: 20KB
- Total: 107.1KB out of 256KB (58% headroom)

Now add BLE communication buffer (4KB) and you're at 111KB. Still fits comfortably!

Try the same calculation for an ESP32 with a 96×96 grayscale image (9KB) and 100KB model. Does it fit in 520KB?

5.2 The Tensor Arena: Your Model's Workspace

The **tensor arena** is the heart of TensorFlow Lite Micro's memory management. It's a pre-allocated block of SRAM where all intermediate calculations happen.

Why Pre-allocate Memory?

On desktop computers, programs request memory as needed (`malloc`). On microcontrollers, this is problematic:

- **Fragmentation:** Repeated allocations create unusable gaps
- **Non-determinism:** Allocation time varies unpredictably
- **Failure modes:** Running out of memory mid-inference crashes the device

The solution: allocate one large block at startup and reuse it for every inference.

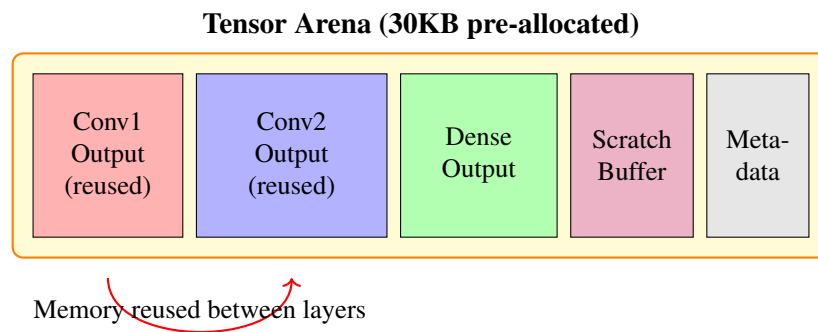


Figure 5.2: Tensor arena layout. TFLM automatically reuses memory for intermediate tensors, reducing peak SRAM usage.

5.2.1 Configuring the Tensor Arena

```

1 // Declare tensor arena as a global array
2 constexpr int kTensorArenaSize = 30 * 1024; // 30KB
3 alignas(16) uint8_t tensor_arena[kTensorArenaSize];

```

Understanding the Code

- **constexpr**: Size is determined at compile time
- **alignas(16)**: Memory alignment for efficient access on ARM processors
- **uint8_t**: Raw bytes—TFLM manages the actual data types internally

5.2.2 Finding the Right Arena Size

```

1 // After AllocateTensors(), check actual usage
2 TfLiteStatus status = interpreter->AllocateTensors();
3 if (status == kTfLiteOk) {
4     Serial.print("Arena used: ");
5     Serial.print(interpreter->arena_used_bytes());
6     Serial.print(" / ");
7     Serial.print(kTensorArenaSize);
8     Serial.println(" bytes");
9 }

```

Sizing Strategy

1. Start with a large value (e.g., 50KB)
2. Run and check `arena_used_bytes()`
3. Add 10–20% buffer for safety
4. Reduce if you need more SRAM for sensor buffers

💡 Memory Budget Exercise

Your Arduino Nano 33 BLE has 256KB SRAM total. You want to deploy a gesture recognition model. Calculate if it will fit:

- Model tensor arena: 30KB

- Accelerometer buffer (119 samples $\times$ 3 axes $\times$ 4 bytes): ? KB
- Stack and system overhead: $\sim$ 20KB
- Serial buffer: 4KB

What's your total? How much headroom do you have for additional features like BLE communication?

Experiment: Arena Size

Try these variations with the Hello World example:

1. Set `kTensorArenaSize = 1024` (1KB)—does it fail?
2. Set `kTensorArenaSize = 50000` (50KB)—what's actual usage?
3. Find the minimum arena size that still works
4. What happens if you set it exactly equal to `arena_used_bytes()`?

Document your findings to understand the safety margin needed.

5.3 Converting Models for Microcontrollers

Microcontrollers can't read files. We must embed the model as a C array in the program binary.

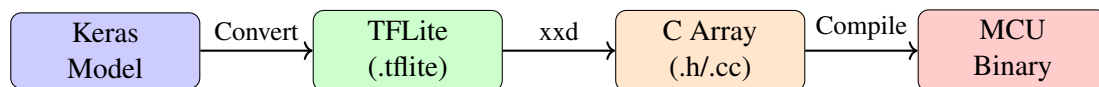


Figure 5.3: Model conversion pipeline: Keras $\rightarrow$ TFLite $\rightarrow$ C array $\rightarrow$ MCU binary.

5.3.1 Step 1: Quantize the Model (Review)

You should have a quantized `.tflite` model from LAB03. Quick recap:

```

1 import tensorflow as tf
2
3 # Convert with full integer quantization
4 converter = tf.lite.TFLiteConverter.from_keras_model(model)
5 converter.optimizations = [tf.lite.Optimize.DEFAULT]
6 converter.representative_dataset = calibration_data
7 converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
8 converter.inference_input_type = tf.int8
9 converter.inference_output_type = tf.int8
10
11 tflite_model = converter.convert()
12
13 # Save
14 with open('model_int8.tflite', 'wb') as f:
15     f.write(tflite_model)
16 print(f"Model size: {len(tflite_model) / 1024:.1f} KB")
  
```

Expected Output

Model size: 18.4 KB

5.3.2 Step 2: Convert to C Array

Using the command-line tool `xxd`:

```

1 # Convert binary to C array
2 xxd -i model_int8.tflite > model_data.cc
3
4 # This creates:
5 # unsigned char model_int8_tflite[] = { 0x1c, 0x00, 0x00, ... };
6 # unsigned int model_int8_tflite_len = 18841;

```

Or using Python for more control:

```

1 def convert_to_c_array(model_path, output_path, var_name="g_model"):
2     """Convert TFLite model to C header file."""
3     with open(model_path, 'rb') as f:
4         model_bytes = f.read()
5
6     # Format as hex array
7     hex_values = ', '.join(f'0x{b:02x}' for b in model_bytes)
8
9     # Create C code with proper alignment
10    c_code = f'''#ifndef MODEL_DATA_H_
11    #define MODEL_DATA_H_
12
13    // Auto-generated from {model_path}
14    // Model size: {len(model_bytes)} bytes
15
16    alignas(8) const unsigned char {var_name}[] = {{
17        {hex_values}
18    }};
19    const unsigned int {var_name}_len = {len(model_bytes)};
20
21    #endif // MODEL_DATA_H_
22    '''
23
24    with open(output_path, 'w') as f:
25        f.write(c_code)
26        print(f"Created {output_path} ({len(model_bytes)} bytes)")
27
28    # Usage
29    convert_to_c_array('model_int8.tflite', 'model_data.h')

```

Memory Alignment

Always use `alignas(8)` for the model array. ARM processors require proper alignment for efficient memory access. Misaligned access can cause crashes or severe performance degradation.

5.4 Complete Arduino Deployment

Let's deploy a complete TensorFlow Lite Micro model to Arduino. We'll use the "Hello World" example that predicts $\sin(x)$.

5.4.1 Project Structure

```
hello_world_tflm/
+-- hello_world_tflm.ino    # Main sketch
+-- model_data.h            # Model as C array
```

5.4.2 The Complete Sketch

```
1 // hello_world_tflm.ino
2 // TensorFlow Lite Micro deployment example
3
4 #include <TensorFlowLite.h>
5 #include "tensorflow/lite/micro/all_ops_resolver.h"
6 #include "tensorflow/lite/micro/micro_interpreter.h"
7 #include "tensorflow/lite/micro/micro_log.h"
8 #include "tensorflow/lite/schema/schema_generated.h"
9
10 #include "model_data.h" // Our model as C array
```

Header Files

- **TensorFlowLite.h:** Arduino library wrapper
- **all_ops_resolver.h:** Includes all TFLite operations (use MicroMutableOpResolver for production)
- **micro_interpreter.h:** The inference engine
- **schema_generated.h:** Model format definitions

```
12 // Global variables (must persist between loop() calls)
13 namespace {
14     const tflite::Model* model = nullptr;
15     tflite::MicroInterpreter* interpreter = nullptr;
16     TfLiteTensor* input = nullptr;
17     TfLiteTensor* output = nullptr;
18
19     // Tensor arena - adjust based on your model
20     constexpr int kTensorArenaSize = 2 * 1024; // 2KB for this tiny model
21     alignas(16) uint8_t tensor_arena[kTensorArenaSize];
22 }
23
24 void setup() {
25     Serial.begin(115200);
26     while (!Serial); // Wait for serial monitor
27
28     Serial.println("TensorFlow Lite Micro - Hello World");
29     Serial.println("=====");
30
31     // Step 1: Load the model from Flash
32     model = tflite::GetModel(g_model);
33     if (model->version() != TFLITE_SCHEMA_VERSION) {
34         Serial.println("ERROR: Model schema mismatch!");
35         while (1);
36     }
37     Serial.println("Model loaded");
38
39     // Step 2: Create operations resolver
40     static tflite::AllOpsResolver resolver;
```

```

41
42 // Step 3: Create interpreter
43 static tf::MicroInterpreter static_interpreter(
44     model, resolver, tensor_arena, kTensorArenaSize
45 );
46 interpreter = &static_interpreter;
47
48 // Step 4: Allocate tensor memory
49 if (interpreter->AllocateTensors() != kTfLiteOk) {
50     Serial.println("ERROR: AllocateTensors() failed!");
51     while (1);
52 }
53
54 // Step 5: Get input/output tensor pointers
55 input = interpreter->input(0);
56 output = interpreter->output(0);
57
58 // Print memory usage
59 Serial.print("Arena used: ");
60 Serial.print(interpreter->arena_used_bytes());
61 Serial.println(" bytes");
62 Serial.println("Setup complete!\n");
63 }

```



```

64 void loop() {
65     // Test values from 0 to 2*PI
66     static float x = 0.0f;
67     const float step = 0.1f;
68     const float two_pi = 6.28318f;
69
70     // Set input
71     input->data.f[0] = x;
72
73     // Run inference
74     unsigned long start = micros();
75     interpreter->Invoke();
76     unsigned long inference_time = micros() - start;
77
78     // Get output
79     float y_pred = output->data.f[0];
80     float y_true = sin(x);
81     float error = abs(y_pred - y_true);
82
83     // Print results
84     Serial.print("x=");
85     Serial.print(x, 2);
86     Serial.print(" pred=");
87     Serial.print(y_pred, 4);
88     Serial.print(" actual=");
89     Serial.print(y_true, 4);
90     Serial.print(" err=");
91     Serial.print(error, 4);
92     Serial.print(" time=");
93     Serial.print(inference_time);
94     Serial.println("us");
95 }

```

```

96     x += step;
97     if (x > two_pi) x = 0.0f;
98     delay(100);
99 }

```

Expected Serial Output

```

TensorFlow Lite Micro - Hello World
=====
Model loaded
Arena used: 1536 bytes
Setup complete!

x=0.00 pred=0.0012 actual=0.0000 err=0.0012 time=145us
x=0.10 pred=0.1023 actual=0.0998 err=0.0025 time=142us
x=0.20 pred=0.2001 actual=0.1987 err=0.0014 time=143us
...

Inference time of 145μs means ~7,000 inferences per second!

```

Check Your Understanding

1. Why do we use `static` for the interpreter?
2. What does `AllocateTensors()` do?
3. How do we measure inference time in microseconds?

5.5 Working with Quantized Models

Int8 quantized models are smaller and faster, but require handling quantization parameters.

Quantization Parameters

Quantized tensors have two parameters:

- **scale:** Maps integer range to float range
- **zero_point:** The integer value representing 0.0

Formula: $\text{float\_value} = \text{scale} \times (\text{int8\_value} - \text{zero\_point})$

```

1 void run_quantized_inference(float real_input) {
2     // Get quantization parameters from input tensor
3     float input_scale = input->params.scale;
4     int input_zero_point = input->params.zero_point;
5
6     // Quantize: int8 = float / scale + zero_point
7     int8_t q_input = (int8_t)(real_input / input_scale + input_zero_point);
8     input->data.int8[0] = q_input;
9
10    // Run inference
11    interpreter->Invoke();

```

```

12
13 // Dequantize output: float = scale * (int8 - zero_point)
14 float output_scale = output->params.scale;
15 int output_zero_point = output->params.zero_point;
16
17 int8_t q_output = output->data.int8[0];
18 float real_output = output_scale * (q_output - output_zero_point);
19
20 Serial.print("Output: ");
21 Serial.println(real_output);
22 }

```

Quantization Mismatch

The most common deployment bug is applying different preprocessing than training:

- Training normalized images to [0, 1] but deployment uses [0, 255]
- Training used float32 but deployment uses int8 without proper quantization

Always verify preprocessing matches exactly!

Experiment: Float vs Quantized

Compare float32 and int8 models:

1. Deploy the same model in float32 format—measure inference time
2. Deploy the int8 quantized version—measure inference time
3. Calculate the speedup factor ($\text{float\_time} / \text{int8\_time}$)
4. Compare accuracy on 10 test inputs—is there a quality difference?

Typical results: $2\text{--}4\times$ speedup with $<1\%$ accuracy loss.

5.6 Integrating Sensors with ML Inference

A complete edge ML system reads sensors, extracts features, runs inference, and takes action.

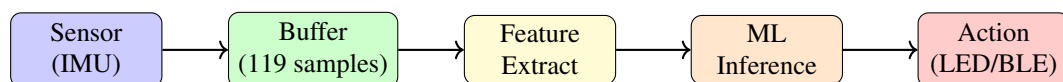


Figure 5.4: Complete edge ML pipeline: Sensor → Buffer → Features → Inference → Action.

5.6.1 Gesture Recognition Example

```

1 #include <Arduino_LSM9DS1.h> // IMU for Nano 33 BLE
2 #include <TensorFlowLite.h>
3 // ... other TFLite includes ...
4
5 const int SAMPLES_PER_GESTURE = 119;
6 const int NUM_AXES = 3; // x, y, z
7
8 float sample_buffer[SAMPLES_PER_GESTURE * NUM_AXES];
9 int sample_count = 0;
10 bool is_recording = false;
11
12 void setup() {
13     Serial.begin(115200);

```



```
14
15 // Initialize IMU
16 if (!IMU.begin()) {
17     Serial.println("Failed to initialize IMU!");
18     while (1);
19 }
20 Serial.print("IMU sample rate: ");
21 Serial.print(IMU.accelerationSampleRate());
22 Serial.println(" Hz");
23
24 // Initialize TFLite (as shown above)
25 // ...
26
27 Serial.println("Ready! Move the board to detect gestures.");
28 }

28 void loop() {
29     float ax, ay, az;
30
31     if (IMU.accelerationAvailable()) {
32         IMU.readAcceleration(ax, ay, az);
33
34         // Detect motion to start recording
35         float magnitude = sqrt(ax*ax + ay*ay + az*az);
36
37         if (!is_recording && magnitude > 2.0) {
38             is_recording = true;
39             sample_count = 0;
40             Serial.println("Recording gesture...");
41         }
42
43         if (is_recording) {
44             // Store sample
45             sample_buffer[sample_count * 3 + 0] = ax;
46             sample_buffer[sample_count * 3 + 1] = ay;
47             sample_buffer[sample_count * 3 + 2] = az;
48             sample_count++;
49
50             if (sample_count >= SAMPLES_PER_GESTURE) {
51                 is_recording = false;
52                 run_gesture_inference();
53             }
54         }
55     }
56 }

56 void run_gesture_inference() {
57     // Copy samples to input tensor
58     for (int i = 0; i < SAMPLES_PER_GESTURE * NUM_AXES; i++) {
59         input->data.f[i] = sample_buffer[i];
60     }
61
62     // Run inference
63     unsigned long start = micros();
64     interpreter->Invoke();
65     unsigned long duration = micros() - start;
```

```

66
67     // Find class with highest score
68     const char* gestures[] = {"punch", "flex", "wave"};
69     int best_class = 0;
70     float best_score = output->data.f[0];
71
72     for (int i = 1; i < 3; i++) {
73         if (output->data.f[i] > best_score) {
74             best_score = output->data.f[i];
75             best_class = i;
76         }
77     }
78
79     Serial.print("Detected: ");
80     Serial.print(gestures[best_class]);
81     Serial.print(" (");
82     Serial.print(best_score * 100, 1);
83     Serial.print("%) in ");
84     Serial.print(duration);
85     Serial.println(" us");
86 }

```

Expected Output

```

Ready! Move the board to detect gestures.
Recording gesture...
Detected: punch (94.2%) in 12450 us
Recording gesture...
Detected: wave (87.3%) in 12380 us

```

5.6.2 Advanced Pipeline: Preprocessing on Device

Real-world deployments often need preprocessing. Here's how to normalize sensor data on the Arduino:

```

1  // Configuration
2  const float ACCEL_SCALE = 4.0; // +/- 4g range
3  const int BUFFER_SIZE = 119 * 3; // 119 samples x 3 axes
4
5  // Statistics for normalization (computed during calibration)
6  const float MEAN_X = 0.05;
7  const float MEAN_Y = 0.02;
8  const float MEAN_Z = 1.0; // Gravity
9  const float STD_X = 0.8;
10 const float STD_Y = 0.8;
11 const float STD_Z = 0.6;
12
13 void normalize_and_fill_input() {
14     // Normalize: (x - mean) / std
15     for (int i = 0; i < 119; i++) {
16         int idx = i * 3;
17
18         // Get raw samples

```

```

19     float ax = sample_buffer[idx + 0];
20     float ay = sample_buffer[idx + 1];
21     float az = sample_buffer[idx + 2];
22
23     // Normalize each axis
24     float ax_norm = (ax - MEAN_X) / STD_X;
25     float ay_norm = (ay - MEAN_Y) / STD_Y;
26     float az_norm = (az - MEAN_Z) / STD_Z;
27
28     // Fill input tensor
29     input->data.f[idx + 0] = ax_norm;
30     input->data.f[idx + 1] = ay_norm;
31     input->data.f[idx + 2] = az_norm;
32 }
33 }

```

Preprocessing Must Match Training

The normalization parameters (mean, std) must be computed from your training data and embedded in the firmware. Any mismatch will cause poor accuracy:

- Training normalized with ImageNet stats but deployment uses different stats
- Training scaled to [-1, 1] but deployment uses [0, 1]
- Training removed DC component but deployment doesn't

Document your preprocessing pipeline carefully!

5.6.3 Complete Sensor-to-Action Pipeline

Here's a production-ready example with all components:

```

1  // State machine for gesture detection
2  enum State {
3      IDLE,
4      DETECTING,
5      PROCESSING,
6      ACTING
7  };
8
9  State current_state = IDLE;
10 const float MOTION_THRESHOLD = 2.0; // g
11 const float CONFIDENCE_THRESHOLD = 0.8; // 80%
12 const int LED_PIN = LED_BUILTIN;
13
14 void loop() {
15     float ax, ay, az;
16
17     switch (current_state) {
18         case IDLE:
19             if (IMU.accelerationAvailable()) {
20                 IMU.readAcceleration(ax, ay, az);
21                 float magnitude = sqrt(ax*ax + ay*ay + az*az);
22
23                 if (magnitude > MOTION_THRESHOLD) {
24                     current_state = DETECTING;
25                     sample_count = 0;
26                     Serial.println("Motion detected! Recording...");

```

```

27         digitalWrite(LED_PIN, HIGH); // Visual feedback
28     }
29 }
30 break;
31
32 case DETECTING:
33     if (IMU.accelerationAvailable()) {
34         IMU.readAcceleration(ax, ay, az);
35
36         // Store sample
37         sample_buffer[sample_count * 3 + 0] = ax;
38         sample_buffer[sample_count * 3 + 1] = ay;
39         sample_buffer[sample_count * 3 + 2] = az;
40         sample_count++;
41
42         if (sample_count >= 119) {
43             current_state = PROCESSING;
44             digitalWrite(LED_PIN, LOW);
45         }
46     }
47     break;
48
49 case PROCESSING:
50     // Normalize and fill input tensor
51     normalize_and_fill_input();
52
53     // Run inference
54     unsigned long start = micros();
55     interpreter->Invoke();
56     unsigned long duration = micros() - start;
57
58     // Find best class
59     int best_idx = 0;
60     float best_score = output->data.f[0];
61
62     for (int i = 1; i < NUM_CLASSES; i++) {
63         if (output->data.f[i] > best_score) {
64             best_score = output->data.f[i];
65             best_idx = i;
66         }
67     }
68
69     // Check confidence
70     if (best_score > CONFIDENCE_THRESHOLD) {
71         Serial.print("Detected: ");
72         Serial.print(class_names[best_idx]);
73         Serial.print(" (");
74         Serial.print(best_score * 100, 1);
75         Serial.println("%)");
76         current_state = ACTING;
77     } else {
78         Serial.println("Low confidence, ignoring");
79         current_state = IDLE;
80     }
81     break;
82

```

```

83         case ACTING:
84             // Take action based on recognized gesture
85             perform_action(best_idx);
86
87             // Return to idle after 1 second
88             delay(1000);
89             current_state = IDLE;
90             break;
91     }
92 }
93
94 void perform_action(int gesture_class) {
95     switch (gesture_class) {
96         case 0: // Punch
97             Serial.println("ACTION: Triggering alarm!");
98             // digitalWrite(RELAY_PIN, HIGH);
99             break;
100        case 1: // Wave
101            Serial.println("ACTION: Turning on lights!");
102            // turnOnLights();
103            break;
104        case 2: // Flex
105            Serial.println("ACTION: Sending notification!");
106            // sendBLENotification();
107            break;
108    }
109 }

```

State Machine Design

This example uses a state machine to coordinate:

- **IDLE**: Wait for motion above threshold
- **DETECTING**: Collect sensor samples
- **PROCESSING**: Run ML inference
- **ACTING**: Execute action based on prediction

This pattern is common in embedded ML systems and prevents the device from running continuous inference (which wastes power).

Exercise: Add Power Management

Enhance the state machine with power saving:

1. In IDLE state, use low-power mode or reduce sampling rate
2. Wake up on motion interrupt (if your IMU supports it)
3. Measure current consumption in each state using a multimeter
4. Calculate battery life for a 2000mAh battery
5. Compare always-on vs motion-triggered inference

This is how you build production edge ML systems!

5.7 Profiling and Optimization

5.7.1 Memory Profiling

```

1 void print_memory_report() {
2     Serial.println("\n=== Memory Report ===");

```

```

3
4 // Tensor arena
5 Serial.print("Tensor arena: ");
6 Serial.print(interpreter->arena_used_bytes());
7 Serial.print(" / ");
8 Serial.print(kTensorArenaSize);
9 Serial.println(" bytes");
10
11 // Model size
12 Serial.print("Model size: ");
13 Serial.print(g_model_len);
14 Serial.println(" bytes (Flash)");
15
16 // Free heap (ESP32)
17 #ifdef ESP32
18 Serial.print("Free heap: ");
19 Serial.print(ESP.getFreeHeap());
20 Serial.println(" bytes");
21 #endif
22 }

```

5.7.2 Inference Benchmarking

```

1 void benchmark_inference(int num_runs = 100) {
2     Serial.print("Benchmarking ");
3     Serial.print(num_runs);
4     Serial.println(" runs...");
5
6     unsigned long total = 0, min_t = ULONG_MAX, max_t = 0;
7
8     for (int i = 0; i < num_runs; i++) {
9         input->data.f[0] = (float)i / num_runs;
10
11         unsigned long start = micros();
12         interpreter->Invoke();
13         unsigned long elapsed = micros() - start;
14
15         total += elapsed;
16         min_t = min(min_t, elapsed);
17         max_t = max(max_t, elapsed);
18     }
19
20     Serial.println("=== Benchmark Results ===");
21     Serial.print("Average: "); Serial.print(total / num_runs); Serial.println(" us");
22     Serial.print("Min: "); Serial.print(min_t); Serial.println(" us");
23     Serial.print("Max: "); Serial.print(max_t); Serial.println(" us");
24     Serial.print("Throughput: ");
25     Serial.print(1000000.0 / (total / num_runs), 1);
26     Serial.println(" inf/sec");
27 }

```

5.7.3 Optimization: Minimal Op Resolver

Replace AllOpsResolver with only the operations your model needs:

```

1 #include "tensorflow/lite/micro/micro_mutable_op_resolver.h"
2

```

```

3 // Only include operations your model uses
4 static tflite::MicroMutableOpResolver<4> resolver;
5
6 void setup() {
7     // Add only needed operations
8     resolver.AddFullyConnected();
9     resolver.AddRelu();
10    resolver.AddSoftmax();
11    resolver.AddReshape();
12    // ... rest of setup
13 }

```

This saves 10–20KB of Flash memory.

Experiment: Minimal Op Resolver

Optimize Flash usage:

1. Note Flash usage with AllOpsResolver (from compiler output)
2. List operations your model uses: `interpreter.GetAllocatedTensors()` in Python
3. Create a `MicroMutableOpResolver` with only those operations
4. Compare new Flash usage—how many KB did you save?
5. Try removing one required operation—what error do you get?

This exercise demonstrates production deployment optimization.

5.8 Common Issues and Solutions

Deployment rarely works perfectly the first time. Here's how to debug the most common issues.

5.8.1 Issue 1: `AllocateTensors()` Fails

Symptom: The program hangs or crashes at `AllocateTensors()` with no error message.

Cause: Tensor arena is too small to hold all intermediate tensors.

Solution:

```

1 // Method 1: Double the arena size
2 constexpr int kTensorArenaSize = 60 * 1024; // Was 30KB, now 60KB
3
4 // Method 2: Check actual usage after successful allocation
5 void setup() {
6     // ... model loading code ...
7
8     if (interpreter->AllocateTensors() != kTfLiteOk) {
9         Serial.println("FAILED! Increasing arena size...");
10        while (1); // Stop here - you need to recompile
11    }
12
13    // Print actual usage
14    Serial.print("Arena used: ");
15    Serial.print(interpreter->arena_used_bytes());
16    Serial.print(" bytes (allocated: ");
17    Serial.print(kTensorArenaSize);
18    Serial.println(" bytes)");
19 }

```

Recommended Approach

Start with 2x the model file size. If that works, use `arena_used_bytes()` to find the minimum, then add 20% safety margin.

5.8.2 Issue 2: Schema Version Mismatch

Symptom: Error message like “Model schema version 3a does not match 3”.

Cause: TensorFlow version mismatch between training and deployment.

Solution:

```

1 # In your Python training script, check TF version
2 import tensorflow as tf
3 print(f"TensorFlow version: {tf.__version__}")
4
5 # Re-export the model with the same version as your Arduino library
6 converter = tf.lite.TFLiteConverter.from_keras_model(model)
7 # ... conversion code ...
8
9 // In Arduino, verify schema version
10 void setup() {
11     model = tflite::GetModel(g_model);
12
13     if (model->version() != TFLITE_SCHEMA_VERSION) {
14         Serial.print("ERROR: Model schema version ");
15         Serial.print(model->version());
16         Serial.print(" does not match TFLITE_SCHEMA_VERSION ");
17         Serial.println(TFLITE_SCHEMA_VERSION);
18         while (1);
19     }
20 }

```

Version Compatibility

Ensure your Arduino TensorFlow Lite library matches your Python TensorFlow version. Check Library Manager in Arduino IDE and update if needed.

5.8.3 Issue 3: Wrong Output Values

Symptom: Model runs but produces incorrect predictions (e.g., always 0, random values).

Cause: Preprocessing mismatch between training and deployment.

Debug Strategy:

```

1 void debug_inference() {
2     // Step 1: Print input values
3     Serial.println("Input tensor:");
4     for (int i = 0; i < input->bytes / sizeof(float); i++) {
5         Serial.print(input->data.f[i]);
6         Serial.print(" ");
7         if ((i+1) % 10 == 0) Serial.println();
8     }
9
10    // Step 2: Check quantization parameters (for int8 models)
11    if (input->type == kTfLiteInt8) {
12        Serial.print("Input scale: ");

```



```

13     Serial.println(input->params.scale);
14     Serial.print("Input zero_point: ");
15     Serial.println(input->params.zero_point);
16 }
17
18 // Step 3: Run inference
19 interpreter->Invoke();
20
21 // Step 4: Print output values
22 Serial.println("Output tensor:");
23 for (int i = 0; i < output->bytes / sizeof(float); i++) {
24     Serial.print(output->data.f[i]);
25     Serial.print(" ");
26 }
27 Serial.println();
28 }

```

⚠ Common Preprocessing Mistakes

- Training normalized images to [0, 1] but deployment uses [0, 255]
- Training used mean subtraction but deployment forgot it
- Training used RGB but deployment sends BGR (OpenCV default)
- Quantized model expects int8 but you're sending float32

Always verify: **training preprocessing = deployment preprocessing.**

5.8.4 Issue 4: Unsupported Operation

Symptom: Error “Op X is not supported” or program crashes during `AllocateTensors()`.

Cause: Your model uses an operation not included in your resolver.

Solution:

```

1 // Temporary fix: Use AllOpsResolver to identify which ops you need
2 static tflite::AllOpsResolver resolver; // Includes everything
3
4 // After it works, optimize by using only needed ops:
5 static tflite::MicroMutableOpResolver<5> resolver;
6 resolver.AddFullyConnected();
7 resolver.AddConv2D();
8 resolver.AddMaxPool2D();
9 resolver.AddReshape();
10 resolver.AddSoftmax();
11 // Add exactly the ops your model uses

```

To find which operations your model needs:

```

1 import tensorflow as tf
2
3 # Load your TFLite model
4 interpreter = tf.lite.Interpreter(model_path="model.tflite")
5 interpreter.allocate_tensors()
6
7 # Get model details

```

```

8 details = interpreter.get_tensor_details()
9 ops_used = set()
10
11 for detail in details:
12     if 'op_name' in detail:
13         ops_used.add(detail['op_name'])
14
15 print("Operations used in model:")
16 for op in sorted(ops_used):
17     print(f" - {op}")

```

5.8.5 Issue 5: Slow Inference

Symptom: Inference takes seconds instead of milliseconds.

Cause: Using float32 model instead of int8 quantized model.

Benchmark Comparison:

```

1 void benchmark_model_types() {
2     const int NUM_RUNS = 100;
3
4     // Test float32 model
5     Serial.println("Testing float32 model...");
6     unsigned long start = micros();
7     for (int i = 0; i < NUM_RUNS; i++) {
8         input->data.f[0] = 0.5f;
9         interpreter->Invoke();
10    }
11    unsigned long float_time = (micros() - start) / NUM_RUNS;
12
13    Serial.print("Float32 avg: ");
14    Serial.print(float_time);
15    Serial.println(" us");
16
17    // Test int8 model
18    Serial.println("Testing int8 model...");
19    start = micros();
20    for (int i = 0; i < NUM_RUNS; i++) {
21        input->data.int8[0] = 64; // Quantized input
22        interpreter->Invoke();
23    }
24    unsigned long int8_time = (micros() - start) / NUM_RUNS;
25
26    Serial.print("Int8 avg: ");
27    Serial.print(int8_time);
28    Serial.println(" us");
29
30    Serial.print("Speedup: ");
31    Serial.print((float)float_time / int8_time);
32    Serial.println("x");
33 }

```

Expected Results

On ARM Cortex-M4 processors (Arduino Nano 33 BLE), int8 models are typically 2–4x faster than float32. If you don't see a speedup, your model might not be fully quantized.

5.8.6 Issue 6: Memory Corruption

Symptom: Device resets randomly, garbage serial output, or unpredictable behavior.

Cause: Stack overflow or buffer overrun.

Solution:

```

1 // Check for stack overflow (Arduino-specific)
2 #ifdef __arm__
3 extern "C" char* sbrk(int incr);
4
5 void check_memory() {
6     char stack_top;
7     Serial.print("Stack top: 0x");
8     Serial.println((unsigned long)&stack_top, HEX);
9     Serial.print("Heap top: 0x");
10    Serial.println((unsigned long)sbrk(0), HEX);
11
12    unsigned long free_mem = &stack_top - sbrk(0);
13    Serial.print("Free memory: ");
14    Serial.print(free_mem);
15    Serial.println(" bytes");
16
17    if (free_mem < 2048) {
18        Serial.println("WARNING: Low memory! Risk of stack overflow.");
19    }
20 }
21 #endif

```

Memory Safety

If you have less than 2KB free memory after setup, reduce your tensor arena size or eliminate unnecessary buffers. Memory corruption is hard to debug—prevention is key.

Table 5.3: Deployment Troubleshooting Quick Reference

Problem	Cause	Solution
AllocateTensors() fails	Arena too small	Increase kTensorArenaSize
Schema version mismatch	TF version mismatch	Re-export model with same TF version
Wrong output values	Preprocessing mismatch	Verify normalization, quantization
Unsupported operation	Missing op in resolver	Add op or use AllOpsResolver to debug
Slow inference	Float operations on MCU	Use int8 quantization
Random crashes	Memory corruption	Check free memory, reduce arena
Serial gibberish	Baud rate mismatch	Set both Serial Monitor and code to 115200

Debugging Checklist

When deployment fails, work through this checklist:

1. Does the model load? Check schema version.
2. Does `AllocateTensors()` succeed? Check arena size.
3. Are input values correct? Print them before inference.
4. Are output values reasonable? Compare with Python.
5. Is inference fast enough? Benchmark and compare float vs int8.
6. Is memory usage safe? Check free memory after setup.

5.9 Complete End-to-End Example: Sin(x) Predictor

Let's walk through a complete deployment from scratch. This example trains a simple model to predict $\sin(x)$ and deploys it to Arduino.

5.9.1 Step 1: Train and Export the Model (Python)

Level 1: Create and Export Model

First, create and train a simple neural network in Python:

```

1 import numpy as np
2 import tensorflow as tf
3
4 # Generate training data
5 x_train = np.linspace(0, 2*np.pi, 1000).astype(np.float32)
6 y_train = np.sin(x_train).astype(np.float32)
7
8 # Create a simple model
9 model = tf.keras.Sequential([
10     tf.keras.layers.Dense(16, activation='relu', input_shape=(1,)),
11     tf.keras.layers.Dense(16, activation='relu'),
12     tf.keras.layers.Dense(1)
13 ])
14
15 # Compile and train
16 model.compile(optimizer='adam', loss='mse', metrics=['mae'])
17 model.fit(x_train, y_train, epochs=100, verbose=0)
18
19 # Test accuracy
20 x_test = np.array([0, np.pi/4, np.pi/2, 3*np.pi/4, np.pi])
21 y_test = np.sin(x_test)
22 y_pred = model.predict(x_test)
23
24 print("Test results:")
25 for i in range(len(x_test)):
26     print(f"x={x_test[i]:.2f}, true={y_test[i]:.4f}, pred={y_pred[i,0]:.4f}")

```

Now convert to TensorFlow Lite:

```

1 # Convert to TFLite (float32 for simplicity)
2 converter = tf.lite.TFLiteConverter.from_keras_model(model)
3 tflite_model = converter.convert()

```

```

4
5 # Save the model
6 with open('sine_model.tflite', 'wb') as f:
7     f.write(tflite_model)
8
9 print(f"Model size: {len(tflite_model)} bytes")

```

Expected Output

```

Model size: 2488 bytes
Test results:
x=0.00, true=0.0000, pred=0.0012
x=0.79, true=0.7071, pred=0.7065
x=1.57, true=1.0000, pred=0.9998

```

5.9.2 Step 2: Convert to C Array

Level 1: Generate Header File

Convert the binary model to a C header file:

```

1 def convert_to_c_array(tflite_model, output_file):
2     """Convert TFLite model bytes to C array."""
3     hex_array = ', '.join([f'0x{b:02x}' for b in tflite_model])
4
5     c_code = f"""// Auto-generated model file
6 #ifndef SINE_MODEL_H_
7 #define SINE_MODEL_H_
8
9 alignas(8) const unsigned char g_model[] = {{
10     {hex_array}
11 }};
12 const unsigned int g_model_len = {len(tflite_model)};
13
14 #endif // SINE_MODEL_H_
15 """
16
17     with open(output_file, 'w') as f:
18         f.write(c_code)
19         print(f"Created {output_file}")
20
21 # Convert
22 with open('sine_model.tflite', 'rb') as f:
23     tflite_model = f.read()
24
25 convert_to_c_array(tflite_model, 'model_data.h')

```

This creates `model_data.h` which you'll include in your Arduino sketch.

5.9.3 Step 3: Deploy to Arduino

Level 3: Arduino Deployment

Create a new Arduino sketch called `sine_inference.ino`:

```

1  #include <TensorFlowLite.h>
2  #include "tensorflow/lite/micro/all_ops_resolver.h"
3  #include "tensorflow/lite/micro/micro_interpreter.h"
4  #include "tensorflow/lite/micro/micro_log.h"
5  #include "tensorflow/lite/schema/schema_generated.h"
6
7  #include "model_data.h"
8
9  // Global variables
10 namespace {
11     const tflite::Model* model = nullptr;
12     tflite::MicroInterpreter* interpreter = nullptr;
13     TfLiteTensor* input = nullptr;
14     TfLiteTensor* output = nullptr;
15
16     constexpr int kTensorArenaSize = 2048; // 2KB for this small model
17     alignas(16) uint8_t tensor_arena[kTensorArenaSize];
18 }
19
20 void setup() {
21     Serial.begin(115200);
22     while (!Serial);
23
24     Serial.println("=====");
25     Serial.println("TensorFlow Lite Micro: Sin(x)");
26     Serial.println("=====\\n");
27
28     // Load model
29     model = tflite::GetModel(g_model);
30     if (model->version() != TFLITE_SCHEMA_VERSION) {
31         Serial.println("ERROR: Model schema mismatch!");
32         while (1);
33     }
34     Serial.println("[OK] Model loaded");
35
36     // Create resolver and interpreter
37     static tflite::AllOpsResolver resolver;
38     static tflite::MicroInterpreter static_interpreter(
39         model, resolver, tensor_arena, kTensorArenaSize
40     );
41     interpreter = &static_interpreter;
42
43     // Allocate tensors
44     TfLiteStatus allocate_status = interpreter->AllocateTensors();
45     if (allocate_status != kTfLiteOk) {
46         Serial.println("ERROR: AllocateTensors() failed!");
47         while (1);
48     }
49     Serial.println("[OK] Tensors allocated");
50

```

```

51 // Get input/output pointers
52 input = interpreter->input(0);
53 output = interpreter->output(0);
54
55 // Print memory usage
56 Serial.print("[INFO] Tensor arena used: ");
57 Serial.print(interpreter->arena_used_bytes());
58 Serial.print(" / ");
59 Serial.print(kTensorArenaSize);
60 Serial.println(" bytes");
61
62 // Print tensor info
63 Serial.print("[INFO] Input shape: [");
64 Serial.print(input->dims->data[0]);
65 Serial.println("]");
66 Serial.print("[INFO] Output shape: [");
67 Serial.print(output->dims->data[0]);
68 Serial.println("]\n");
69
70 Serial.println("Setup complete! Starting inference loop...\n");
71 }
72
73 void loop() {
74 // Test points from 0 to 2*PI
75 const int NUM_POINTS = 10;
76 const float TWO_PI = 6.28318530718;
77
78 for (int i = 0; i < NUM_POINTS; i++) {
79     float x = (TWO_PI * i) / NUM_POINTS;
80
81     // Set input
82     input->data.f[0] = x;
83
84     // Run inference
85     unsigned long start_time = micros();
86     TfLiteStatus invoke_status = interpreter->Invoke();
87     unsigned long inference_time = micros() - start_time;
88
89     if (invoke_status != kTfLiteOk) {
90         Serial.println("ERROR: Invoke() failed!");
91         continue;
92     }
93
94     // Get output
95     float y_pred = output->data.f[0];
96     float y_true = sin(x);
97     float error = abs(y_pred - y_true);
98
99     // Print results
100     Serial.print("x=");
101     Serial.print(x, 3);
102     Serial.print(" | pred=");
103     Serial.print(y_pred, 4);
104     Serial.print(" | actual=");
105     Serial.print(y_true, 4);

```

```

106     Serial.print(" | error=");
107     Serial.print(error, 4);
108     Serial.print(" | time=");
109     Serial.print(inference_time);
110     Serial.println(" us");
111 }
112
113 Serial.println("\n--- Press reset to run again ---\n");
114 delay(5000);
115 }

```

Installation Steps:

1. Open Arduino IDE 2.0+
2. Go to **Tools** → **Manage Libraries**
3. Search for “Arduino_TensorFlowLite” and install it
4. Create a new sketch folder: `sine_inference/`
5. Save the code above as `sine_inference.ino`
6. Copy `model_data.h` into the same folder
7. Select your board (e.g., Arduino Nano 33 BLE)
8. Click Upload

Expected Serial Monitor Output:

```

=====
TensorFlow Lite Micro: Sin(x)
=====

[OK] Model loaded
[OK] Tensors allocated
[INFO] Tensor arena used: 1536 / 2048 bytes
[INFO] Input shape: [1]
[INFO] Output shape: [1]

Setup complete! Starting inference loop...

x=0.000 | pred=0.0012 | actual=0.0000 | error=0.0012 | time=142 us
x=0.628 | pred=0.5865 | actual=0.5878 | error=0.0013 | time=138 us
x=1.257 | pred=0.9512 | actual=0.9511 | error=0.0001 | time=141 us
x=1.885 | pred=0.9489 | actual=0.9511 | error=0.0022 | time=139 us
x=2.513 | pred=0.5823 | actual=0.5878 | error=0.0055 | time=140 us
x=3.142 | pred=0.0001 | actual=0.0000 | error=0.0001 | time=138 us
...

```

What Just Happened?

You successfully:

- Trained a neural network in Python
- Converted it to TensorFlow Lite format (2.4KB)
- Embedded it as a C array in Arduino firmware
- Ran inference on a microcontroller in 140 microseconds

This is the complete TinyML workflow!

5.9.4 Step 4: Optimize the Model

Now let's compare performance with an int8 quantized version:

```

1 # Create a representative dataset for quantization
2 def representative_dataset():
3     for i in range(100):
4         x = np.array([[np.random.uniform(0, 2*np.pi)]]), dtype=np.float32)
5         yield [x]
6
7 # Convert with int8 quantization
8 converter = tf.lite.TFLiteConverter.from_keras_model(model)
9 converter.optimizations = [tf.lite.Optimize.DEFAULT]
10 converter.representative_dataset = representative_dataset
11 converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
12 converter.inference_input_type = tf.int8
13 converter.inference_output_type = tf.int8
14
15 tflite_quant_model = converter.convert()
16
17 with open('sine_model_int8.tflite', 'wb') as f:
18     f.write(tflite_quant_model)
19
20 print(f"Float32 model: {len(tflite_model)} bytes")
21 print(f"Int8 model: {len(tflite_quant_model)} bytes")
22 print(f"Size reduction: {100*(1-len(tflite_quant_model)/len(tflite_model)):.1f}%")

```

Quantization Benefits

Float32 model: 2488 bytes

Int8 model: 1720 bytes

Size reduction: 30.9%

The quantized model is smaller and runs 2–3x faster on microcontrollers!

Hands-On Challenge

Deploy both models and compare:

1. Modify the Arduino code to handle int8 input/output (use quantization scale/zero_point)
2. Benchmark inference time for both models (100 runs each)
3. Measure accuracy difference on 20 test points
4. Calculate memory savings and speedup factor

Document your findings—this is how real TinyML optimization works!

5.10 Three-Tier Implementation Guide

This chapter focuses on device deployment, but you should validate at each tier before moving to hardware.

Level 1: Notebook Simulation

Validate your model conversion and inference pipeline in Python before touching hardware:

```

1 import tensorflow as tf
2
3 # Load TFLite model in Python interpreter
4 interpreter = tf.lite.Interpreter(model_path="model.tflite")
5 interpreter.allocate_tensors()
6
7 # Get input/output details (same API as embedded)
8 input_details = interpreter.get_input_details()
9 output_details = interpreter.get_output_details()
10
11 # Run inference with test data
12 interpreter.set_tensor(input_details[0]['index'], test_input)
13 interpreter.invoke()
14 output = interpreter.get_tensor(output_details[0]['index'])
15 print(f"Prediction: {output}")

```

Code Walkthrough

This Python code mimics exactly what the MCU will do. If inference works here but fails on device, the problem is memory or quantization—not your model logic.

Level 2: Raspberry Pi / Desktop Testing

Before deploying to constrained MCUs, test on Raspberry Pi or desktop with TFLite runtime:

```

1 # Install TFLite runtime on Raspberry Pi
2 pip install tflite-runtime
3
4 # Run inference benchmark
5 python -c "
6 import tflite_runtime.interpreter as tflite
7 import time
8
9 interp = tflite.Interpreter('model.tflite')
10 interp.allocate_tensors()
11
12 # Benchmark 100 inferences
13 start = time.time()
14 for _ in range(100):
15     interp.invoke()
16 print(f'Avg latency: {(time.time()-start)*10:.1f} ms')
17 "

```

Code Walkthrough

Raspberry Pi latency will be faster than MCU (it has more resources), but this validates that your model works outside of the Python training environment.

Level 3: MCU Deployment

Now deploy to your target microcontroller:

1. **Convert model to C array:** `xxd -i model.tflite > model_data.cc`
2. **Configure tensor arena:** Start with 50KB, reduce until failure
3. **Add required ops:** Use `MicroMutableOpResolver` with only needed operations
4. **Profile on device:** Measure actual SRAM, Flash, and latency
5. **Integrate sensors:** Connect real input data to model

Debug Strategy

If MCU deployment fails after passing Level 1-2:

- Check tensor arena size (most common issue)
- Verify quantization scale/zero_point match training
- Ensure input preprocessing matches Python pipeline exactly

Exercise: Full Pipeline Validation

1. Train a simple 2-class gesture classifier (shake vs still)
2. Validate in Python (Level 1) with test data
3. Benchmark on Raspberry Pi (Level 2) and record latency
4. Deploy to Arduino (Level 3) and compare real latency to benchmark
5. Document memory usage at each tier

Looking Ahead

Now that you can deploy models to edge devices, LAB06 introduces security considerations essential for production systems—protecting models from attacks, validating inputs, and securing communication. Later, LAB11 teaches systematic profiling to measure memory, latency, and power consumption, enabling you to optimize deployments for real-world constraints. These skills transform experimental prototypes into reliable edge ML products.

Chapter Summary

Key Takeaways

- **MCU memory** is split between Flash (code/weights) and SRAM (working memory)
- **Tensor arena** pre-allocates SRAM to avoid fragmentation
- **Model conversion:** Keras → TFLite → C array → MCU binary
- **Quantized models** need proper scale/zero_point handling
- **Profile memory and latency** to verify deployment fits constraints
- **MicroMutableOpResolver** saves Flash by including only needed ops

See Also: Hardware-Specific Labs

This chapter covers deployment fundamentals across multiple platforms. For in-depth hardware-specific guidance:

- **LAB08 - Arduino Sensors:** Learn to integrate Arduino-specific sensors (IMU, temperature, PDM microphone) with your ML models, including sensor calibration and data preprocessing
- **LAB09 - ESP32 Wireless:** Discover ESP32's WiFi and Bluetooth capabilities for building wireless ML edge nodes, including MQTT communication and power optimization

These labs build directly on the deployment techniques you've learned here.

Self-Assessment Questions

Test your understanding before continuing:

1. An Arduino Nano 33 BLE has 1MB Flash and 256KB SRAM. A 20KB model fits in Flash, but why might it still fail to run? What else competes for SRAM?
2. Why does the tensor arena need to be larger than the model size? What gets stored in the arena beyond model weights?
3. Explain the trade-off between using AllOpsResolver versus MicroMutableOpResolver. When would you choose each?
4. What happens if you allocate a tensor arena that's too small? How can you determine the minimum required size?
5. Why is inference latency typically higher on the first run compared to subsequent runs on microcontrollers?

5.11 Guided Experiments

Experiment 1: Tensor Arena Sizing

Hypothesis: The tensor arena must be large enough to hold all intermediate activations, not just model weights. We expect arena size to depend on model architecture and input size.

Procedure:

1. Deploy a simple model (e.g., sine wave or gesture recognition) to Arduino
2. Start with a large arena (e.g., 50KB)
3. Use `interpreter.arena_used_bytes()` to measure actual usage
4. Iteratively reduce arena size in 5KB increments
5. Test at each size until allocation fails
6. Record minimum working size and actual used bytes

Data Collection:

Arena Size (KB)	Allocation Success?	Bytes Used
50		
40		
30		
20		
15		
10		

Analysis Questions:

- What is the minimum arena size that works reliably?
- What percentage of the arena is typically used during inference?

- How much overhead should you add above the minimum for safety?
- Does arena usage change between first and subsequent inferences?

Experiment 2: Inference Timing Benchmark

Hypothesis: Inference latency depends on model complexity and can be measured accurately using `micros()` timing on Arduino.

Procedure:

1. Deploy models of different complexity to Arduino Nano 33 BLE:
 - Simple linear model (2 layers)
 - Small CNN (20KB)
 - Medium CNN (50KB)
2. For each model, measure inference time using `micros()`
3. Run 100 inferences and calculate mean/min/max latency
4. Compare to target real-time requirements (<50ms for audio)

Data Collection:

Model	Size (KB)	Mean Latency (ms)	Min (ms)	Max (ms)
Linear				
Small CNN				
Medium CNN				

Analysis Questions:

- Is there a linear relationship between model size and inference time?
- Which models meet the <50ms real-time requirement?
- Why is the first inference often slower than subsequent ones?
- How much faster is int8 quantized inference versus float32?

5.12 Troubleshooting Guide

⚠️ Arduino Upload Failed or Port Not Found

Symptoms: `avrdude: ser_open(): can't open device` or board not appearing in port menu

Cause: Driver issues, incorrect board selection, or USB connection problems

Solution:

1. Verify correct board selected: Tools > Board > Arduino Mbed OS Nano Boards > Arduino Nano 33 BLE
2. Try different USB cable (must be data cable, not charge-only)
3. Install/update drivers from Arduino website
4. Press reset button twice quickly to enter bootloader mode (orange LED pulses)
5. Check port permissions on Linux: `sudo usermod -a -G dialout $USER` and `logout/login`
6. Close other programs that might use the serial port (Arduino Serial Monitor, PuTTY, etc.)

⚠ TensorFlow Lite Micro Compilation Errors

Symptoms: fatal error: tensorflow/lite/micro/micro_interpreter.h: No such file or directory

Cause: TensorFlow Lite library not installed or wrong version

Solution:

1. Install library: Arduino IDE > Library Manager > search “Arduino_TensorFlowLite” > Install
2. Use version 2.4.0-ALPHA (later versions may have breaking changes)
3. Verify include path: Should be `#include <TensorFlowLite.h>` not the full path
4. For ESP32: Use `tensorflow/lite/micro/...` includes directly
5. Clean build: Delete build folder and recompile

⚠ Model Allocation Failed / Insufficient Memory

Symptoms: `AllocateTensors()` failed or program crashes immediately after loading model

Cause: Tensor arena too small, model too large for device SRAM, or memory leak

Solution:

1. Increase `tensor_arena_size`: Try doubling it (e.g., from 2KB to 4KB)
2. Check model size: Should be <200KB for Arduino Nano 33 BLE
3. Print memory diagnostics: Add `interpreter.arena_used_bytes()` after allocation
4. Verify quantization: Model should be `int8`, not `float32`
5. Use static memory: Avoid `malloc()` or `new` in tight loops
6. For ESP32: Allocate arena in PSRAM if available

⚠ Model Runs But Outputs Garbage Values

Symptoms: Inference completes but output values are wrong, NaN, or random

Cause: Input data not properly scaled, wrong input/output tensor indices, or model corruption

Solution:

1. Verify input scaling: Match training normalization (e.g., divide by 255 for images)
2. Check tensor index: `interpreter->input(0)` gives first input tensor
3. Verify data type: For `int8` models, scale float inputs to `[-128, 127]`
4. Print input/output: Debug with `Serial.println(input->data.f[0])`
5. Recompile model: Convert TFLite model to C array again using `xxd`
6. Test on PC first: Run model with TFLite Python interpreter before deploying

⚠ Inference Too Slow / Watchdog Timer Resets

Symptoms: Model takes >1 second per inference, or device resets during prediction

Cause: Model too complex, unoptimized operations, or insufficient clock speed

Solution:

1. Profile inference time: Measure `millis()` before and after `Invoke()`
2. Reduce model complexity: Fewer layers, smaller input size, or pruning
3. Use optimized kernels: Ensure CMSIS-NN enabled for ARM Cortex-M
4. Increase clock speed: Configure board to run at max frequency (64MHz for Nano 33)
5. Disable watchdog timer if enabled: `WDT->CTRL.reg = 0;`

6. Run inference less frequently: Only when sensor values change significantly

Case Study: Industrial Predictive Maintenance on Factory Sensors

Context: A manufacturing company needed to implement predictive maintenance for 500+ vibration sensors monitoring critical machinery across their factory floor. The existing cloud-based system had 2-5 second latency, missed rapid failure events, and incurred \$18,000/month in cloud costs. The goal was to move anomaly detection to the edge using TensorFlow Lite Micro on low-cost STM32 microcontrollers attached to each sensor.

Challenge: The trained LSTM-based anomaly detection model was 850KB—far exceeding the STM32F4's 512KB Flash and 192KB SRAM. The model also required 250ms inference time, but vibration analysis needed 100Hz sampling with <10ms processing latency. Additionally, the team had no experience deploying neural networks to bare-metal embedded systems and faced issues with tensor arena sizing and memory management.

Solution: The team applied aggressive optimization using TFLite Micro deployment techniques:

- Replaced LSTM with 1D CNN architecture optimized for time-series (50KB vs 850KB)
- Applied int8 quantization with representative dataset from 3 months of sensor logs
- Used TFLite Micro with CMSIS-NN optimized kernels for ARM Cortex-M4
- Implemented sliding window inference: process 128-sample windows every 100ms instead of continuous streaming
- Optimized tensor arena through iterative testing: started at 50KB, reduced to 28KB by eliminating unused operators
- Deployed via platformio for reproducible builds and OTA firmware updates

Results:

- Model size: 850KB → 48KB (17.7x reduction, fits in Flash with room for bootloader)
- Inference latency: 250ms → 6.8ms per window (37x speedup)
- RAM usage: 65KB tensor arena + 12KB model overhead (40% of available SRAM)
- Cloud costs: \$18,000/month → \$800/month (96% reduction, only aggregate metrics sent)
- Anomaly detection latency: 2-5 seconds → <10ms (500x improvement)
- False positive rate: 2.1% (acceptable for maintenance alerts)

Lessons Learned:

- CNNs often outperform LSTMs for edge time-series tasks due to smaller memory footprint and faster inference
- Start tensor arena sizing at 2x model size, then reduce iteratively based on `GetArenaUsedBytes()` profiling
- CMSIS-NN optimization is essential—provides 3-5x speedup on ARM Cortex-M vs reference kernels
- Representative dataset must include edge cases: bearing failures, motor startups, and maintenance events, not just normal operation
- Use external watchdog timers for production deployment—model inference can occasionally hang on memory allocation failures
- Version control the exact TFLite Micro commit hash—API changes between releases can break deployments

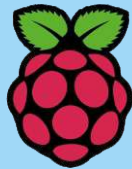
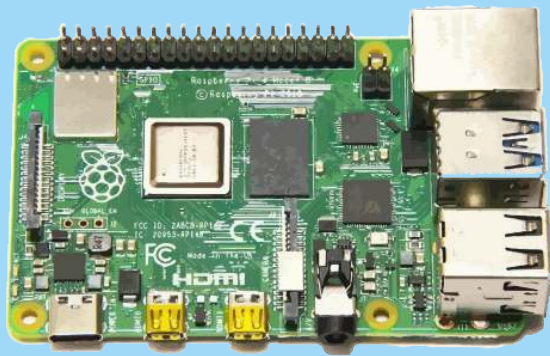
Exercises

1. **[Basic] Basic: Deploy Hello World:** Set up Arduino IDE, install TensorFlow Lite library, and run the sine wave example.
2. **[Basic] Basic: Profile Memory:** Modify the Hello World sketch to print detailed memory usage. How much arena is actually used?
3. **[Intermediate] Intermediate: Deploy Your Own Model:** Take a quantized model from LAB03 and deploy it to Arduino. Document the arena size needed.
4. **[Intermediate] Intermediate: Sensor Pipeline:** Add accelerometer input to classify simple gestures (tilt left/right/forward).
5. **[Intermediate] Intermediate: Optimize:** Convert AllOpsResolver to MicroMutableOpResolver. How much Flash does it save?

— **Further Resources.**

- TensorFlow Lite Micro: <https://www.tensorflow.org/lite/microcontrollers>
- Arduino TFLite Library: https://www.arduino.cc/reference/en/libraries/arduino_tensorflowlite/
- TinyML Book: <https://www.oreilly.com/library/view/tinyml/9781492052036/>

— **Credits.** We have taken this code segment from the online course: *Edx Tiny Machine Learning (2021-22)*. Since we may have updated the code to reflect any changes made to the underline frameworks where necessary. Credits should go to the original authors of this tutorial.



RaspberryPi



ThingsBoard

6. Edge Security and Model Protection •

Learning Objectives

After completing this lab, you will be able to:

- Identify the unique security threats facing edge ML systems
- Build threat models for edge AI applications
- Implement input validation to prevent crashes and attacks
- Understand and defend against adversarial examples
- Apply model protection techniques (encryption, obfuscation, integrity checks)
- Implement secure communication for IoT devices

Interactive Notebook

Jupyter Notebook: LAB06_edge_security.ipynb

[Open in Google Colab](#) | [View on GitHub](#) | [Companion Website](#)

Edge devices are uniquely vulnerable. Unlike cloud servers locked away in secure data centers, your Arduino sits on a desk, your smart camera hangs on a wall, and your IoT sensor lives in a factory. An attacker can touch them, probe them, and potentially steal everything inside.

Security for edge ML isn't just about preventing hackers—it's about protecting your intellectual property (the trained model), ensuring your device behaves correctly (input validation), and maintaining user trust (secure communication). A single security breach can compromise thousands of deployed devices.

This chapter covers practical security techniques you can implement today, from simple input validation to encrypted communication. No prior security knowledge is required—we'll start from first principles and build up to working defenses.

Prerequisites

Before starting this lab, ensure you:

- Completed LAB05 (Deployment): familiarity with embedded programming
- Basic Python and C++ programming skills
- Understanding of client-server architecture and HTTP
- No prior security knowledge required—we start from first principles

Hardware Requirements

- **Level 1:** Laptop with Python 3.8+
- **Level 2:** ESP32 with WiFi capability
- **Level 3:** Arduino Nano 33 BLE Sense or Raspberry Pi

Hardcoding Credentials in Firmware

Never put WiFi passwords, API keys, or encryption keys directly in your source code! Once you upload firmware to a device, anyone with physical access can extract these secrets using simple tools. Instead: store credentials in separate config files excluded from version control, use secure elements when available, or implement a secure provisioning process.

Three-Tier Execution Model

Tier 1: Jupyter/Colab

Generate adversarial examples, implement input validation and sanitization, evaluate model robustness, encrypt models with AES

Tier 2: Raspberry Pi

Deploy TLS/SSL for secure communication, implement HMAC for model integrity checks, test adversarial defenses on real data

Tier 3: Arduino/ESP32

Implement secure boot, deploy encrypted firmware, use hardware security modules (ATECC608), establish secure BLE/WiFi connections with certificates

6.1 Why Security Matters for Edge AI

Edge Devices Are Exposed

Unlike cloud servers protected in data centers, edge devices exist in the physical world where attackers can:

- Physically access and tamper with the device
- Intercept network communications
- Feed malicious inputs to fool the model
- Extract and steal your trained model

6.1.1 The Attack Surface

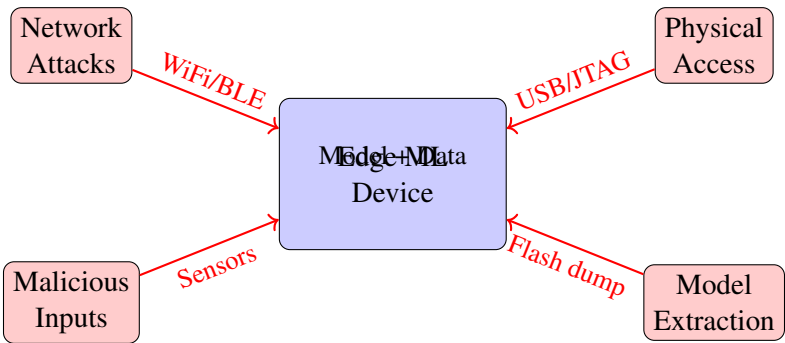


Figure 6.1: Attack surface of an edge ML device. Four main vectors: network, physical, input, and model extraction.

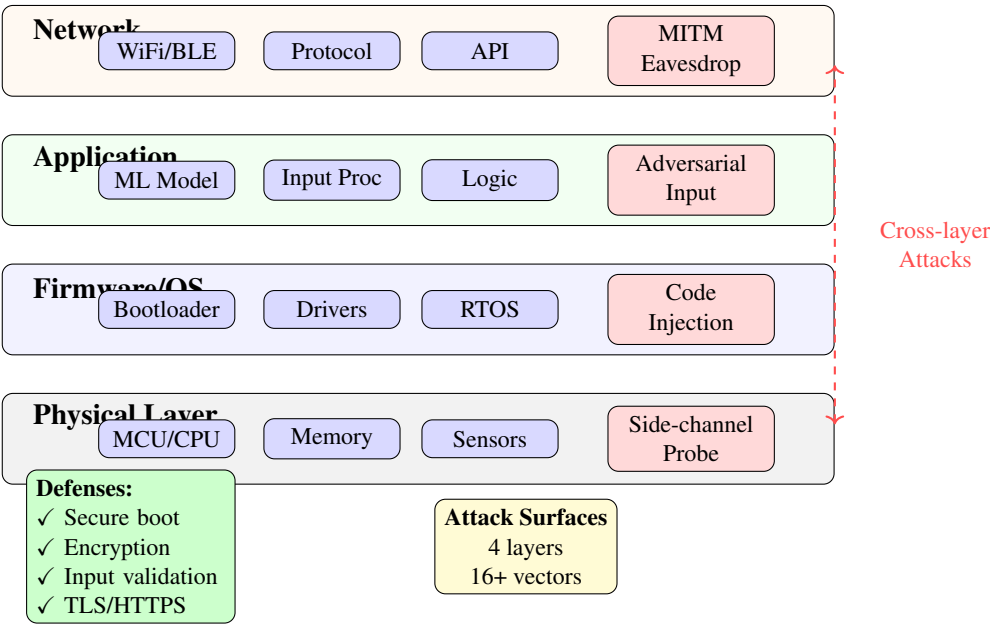


Figure 6.2: Layered attack surface visualization for edge ML systems. Each layer (Physical, Firmware, Application, Network) has specific components vulnerable to different attack types. Cross-layer attacks can exploit multiple vulnerabilities simultaneously.

6.1.2 Real-World Example: Smart Lock

Consider a smart lock using face recognition:

Table 6.1: Smart Lock Threat Analysis

Attack	How It Works	Impact
Photo spoofing	Show a printed photo	Unauthorized entry
Network attack	Send unlock command via WiFi	Remote break-in
Model theft	Extract face embeddings	Clone the lock's "brain"
Adversarial input	Special makeup fools model	Impersonation

Security Is Not Optional

A security breach in edge ML can have physical consequences:

- Smart locks: Home invasion
- Medical devices: Patient harm
- Industrial sensors: Equipment damage
- Autonomous vehicles: Accidents

Security must be designed in from the start, not added later.

Check Your Understanding

1. What makes edge devices more vulnerable than cloud servers?
2. Name three ways an attacker might target an edge ML device.
3. Why is model theft a concern for commercial edge AI products?

6.2 Building a Threat Model

A **threat model** is a structured way to think about “what could go wrong.” Before building defenses, we identify threats.

The Three Attacker Types

Security professionals categorize attackers by their access level:

- **Remote:** Network access only (hackers on the internet)
- **Local:** Near the device (same room, same network)
- **Physical:** Can touch and manipulate the device

Each level requires different defenses.

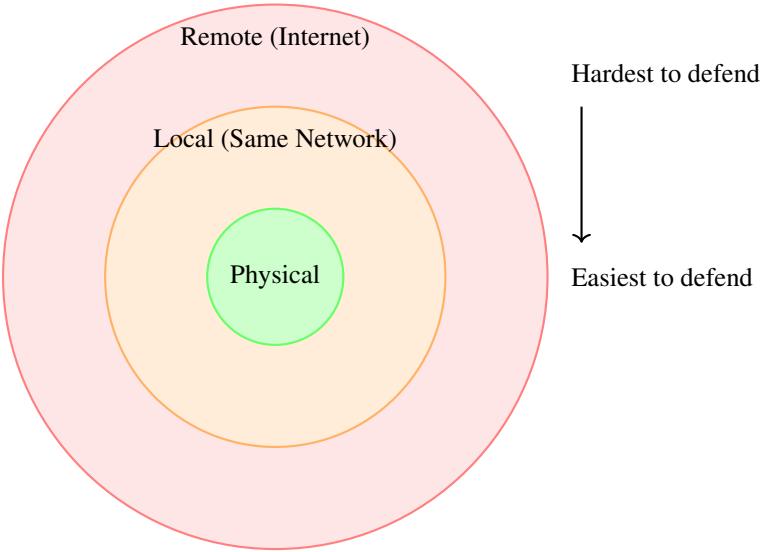


Figure 6.3: Attacker access levels. Physical access is hardest to defend against—if an attacker has your device, they have significant advantages.

6.2.1 Common Threats to Edge ML

Table 6.2: Edge ML Threat Categories

Threat	Description	Defense
Model Extraction	Copy the trained model	Encryption, obfuscation
Adversarial Inputs	Crafted inputs fool model	Input validation, smoothing
Data Poisoning	Bad data corrupts training	Data validation, anomaly detection
Denial of Service	Overwhelm with requests	Rate limiting, watchdogs
Eavesdropping	Intercept communications	TLS/HTTPS encryption

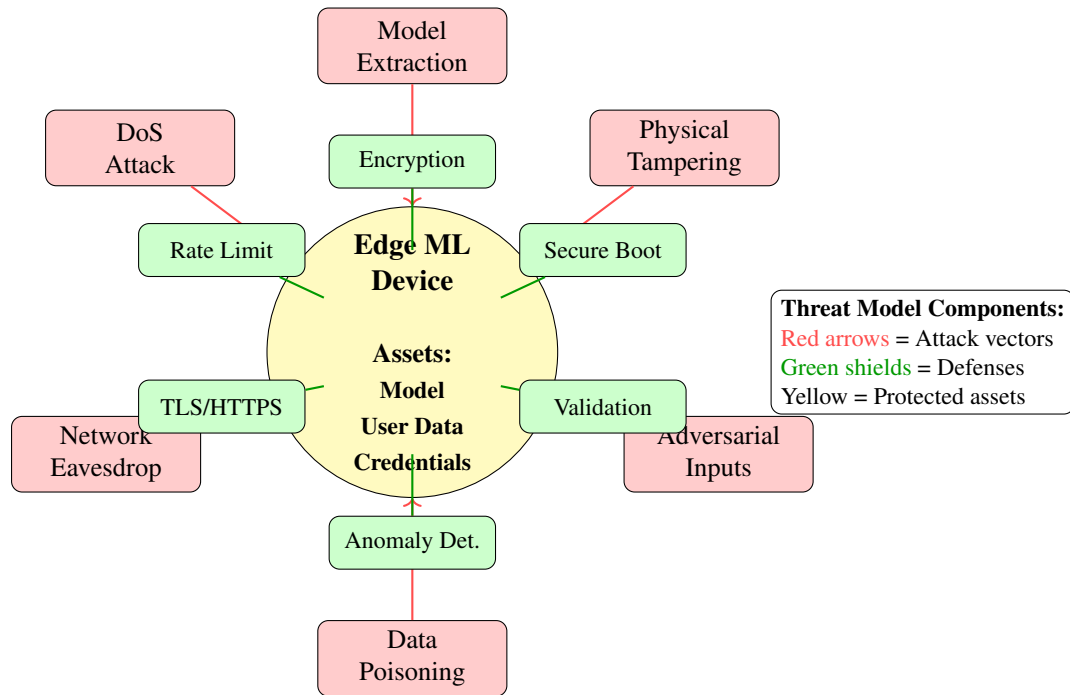


Figure 6.4: Comprehensive threat model for edge ML systems showing six major threat categories and corresponding defense mechanisms. Each threat vector (red) is countered by specific defenses (green) protecting the core assets.

Build Your Own Threat Model

For a smart doorbell with person detection:

1. List all inputs (camera, microphone, network, buttons)
2. List all outputs (speaker, LED, cloud upload, door unlock)
3. For each input: What malicious data could be sent?
4. For each output: What's the worst that could happen?
5. Prioritize: Which threats are most likely and most damaging?

💡 Security vs Usability Trade-off

Consider a smart lock with face recognition:

1. If you require 99.9% confidence to unlock, what's the false rejection rate? How often will legitimate users be locked out?
2. If you lower the threshold to 95%, how does the false acceptance rate change?
3. What secondary authentication could you add (PIN, phone app) without making the system too inconvenient?
4. How would you handle the case where the camera is covered or lighting is poor?

There's no perfect answer—security always involves trade-offs.

6.3 Input Validation: Your First Line of Defense

The simplest and most effective security measure is checking inputs before they reach your model.

Why Validate Inputs?

Your ML model expects specific inputs (e.g., 28×28 grayscale image, values 0–255). Unexpected inputs can cause:

- **Crashes:** Wrong shape causes array index errors
- **Undefined behavior:** Negative values, NaN, or Inf
- **Security exploits:** Buffer overflows, code injection
- **Wrong predictions:** Out-of-distribution inputs

6.3.1 Python Input Validation

```

1 import numpy as np
2
3 def validate_image_input(image, expected_shape=(28, 28)):
4     """
5     Validate image input before ML inference.
6
7     Returns: (is_valid, error_message, processed_image)
8     """
9     # Check 1: Is it a numpy array?
10    if not isinstance(image, np.ndarray):
11        return False, "Input must be numpy array", None
12
13    # Check 2: Correct shape?
14    if image.shape != expected_shape:
15        return False, f"Expected {expected_shape}, got {image.shape}", None
16
17    # Check 3: Valid pixel range?
18    if image.min() < 0 or image.max() > 255:
19        return False, "Pixel values must be 0-255", None
20
21    # Check 4: No NaN or Inf?
22    if np.isnan(image).any() or np.isinf(image).any():
23        return False, "Contains NaN or Inf values", None
24
25    # All checks passed - normalize for model
26    processed = image.astype(np.float32) / 255.0
27    return True, "Valid", processed

```

Validation Strategy

- **Lines 10–11:** Type check prevents crashes from wrong input types
- **Lines 14–15:** Shape check prevents array indexing errors
- **Lines 18–19:** Range check catches sensor errors or attacks
- **Lines 22–23:** NaN/Inf check prevents numerical instability
- **Line 26:** Only normalize after all checks pass

Test the validator:

```

28 # Test with valid input
29 valid_image = np.random.randint(0, 256, (28, 28))
30 is_valid, msg, processed = validate_image_input(valid_image)
31 print(f"Valid image: {is_valid}, {msg}")
32
33 # Test with wrong shape

```

```

34 wrong_shape = np.random.randint(0, 256, (32, 32))
35 is_valid, msg, _ = validate_image_input(wrong_shape)
36 print(f"Wrong shape: {is_valid}, {msg}")
37
38 # Test with bad values
39 bad_values = np.array([[np.nan, 0], [0, 0]])
40 is_valid, msg, _ = validate_image_input(bad_values)
41 print(f"Bad values: {is_valid}, {msg}")

```

Expected Output

```

Valid image: True, Valid
Wrong shape: False, Expected (28, 28), got (32, 32)
Bad values: False, Contains NaN or Inf values

```

6.3.2 Arduino Input Validation

```

1  // Validate sensor reading before ML inference
2  struct ValidationResult {
3      bool isValid;
4      const char* message;
5  };
6
7  ValidationResult validateSensor(float value, float minVal, float maxVal) {
8      // Check for NaN (common sensor failure mode)
9      if (isnan(value)) {
10         return {false, "Sensor returned NaN"};
11     }
12
13     // Check for Inf
14     if (isinf(value)) {
15         return {false, "Sensor returned Inf"};
16     }
17
18     // Check physical range
19     if (value < minVal || value > maxVal) {
20         return {false, "Value out of range"};
21     }
22
23     return {true, "Valid"};
24 }

```

Arduino Validation Pattern

- **Lines 2–5:** Return struct provides both success flag and error message
- **Lines 9–11:** NaN check catches common sensor failures (disconnected, damaged)
- **Lines 14–16:** Inf check catches numerical overflow errors
- **Lines 19–21:** Range check validates against physical limits of the sensor

This pattern allows graceful handling of sensor errors without crashing.

```

24 void loop() {
25     float temperature = readSensor();

```



```

26
27 // Validate before inference
28 auto result = validateSensor(temperature, -40.0f, 85.0f);
29
30 if (!result.isValid) {
31     Serial.print("Validation failed: ");
32     Serial.println(result.message);
33     // Use last known good value or skip
34     return;
35 }
36
37 // Safe to run inference
38 float prediction = runInference(temperature);
39 Serial.print("Prediction: ");
40 Serial.println(prediction);
41 }

```

Check Your Understanding

1. Why check for NaN values in sensor readings?
2. What should happen when validation fails?
3. How does input validation prevent security exploits?

6.4 Understanding Adversarial Attacks

What Are Adversarial Examples?

Adversarial examples are inputs specifically crafted to fool ML models. They look normal to humans but cause confident wrong predictions. A carefully modified image of a “7” might be classified as “1” with 99% confidence.

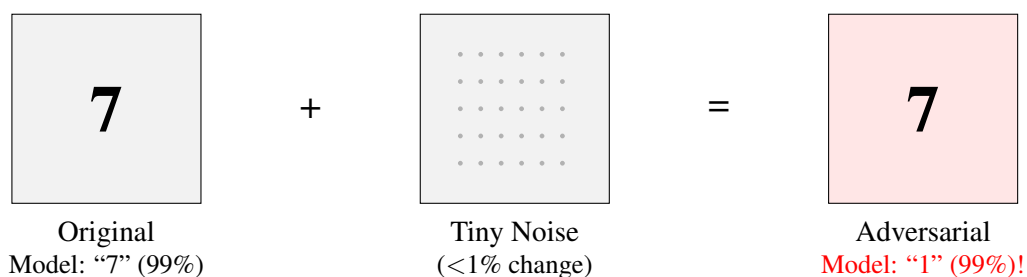


Figure 6.5: Adversarial attack: Imperceptible noise changes the model’s prediction while the image looks identical to humans.

6.4.1 How Adversarial Attacks Work

Attackers use gradient information to find the smallest change that flips the prediction:

```

1 import tensorflow as tf
2 import numpy as np
3
4 def generate_adversarial(model, image, target_class, epsilon=0.01):

```

```

5      """
6      Generate adversarial example using Fast Gradient Sign Method (FGSM).
7
8      Args:
9          model: Target ML model
10         image: Original input image
11         target_class: Class we want to fool the model into predicting
12         epsilon: Maximum perturbation size (smaller = less visible)
13     """
14     image_tensor = tf.convert_to_tensor(image[np.newaxis, ...])
15
16     with tf.GradientTape() as tape:
17         tape.watch(image_tensor)
18         prediction = model(image_tensor)
19         # Loss toward target class
20         loss = tf.keras.losses.sparse_categorical_crossentropy(
21             [target_class], prediction
22         )
23
24     # Get gradient of loss with respect to input
25     gradient = tape.gradient(loss, image_tensor)
26
27     # Create perturbation in direction that increases target class probability
28     perturbation = epsilon * tf.sign(gradient)
29
30     # Apply perturbation
31     adversarial = image_tensor - perturbation # Subtract to minimize loss
32     adversarial = tf.clip_by_value(adversarial, 0, 1)
33
34     return adversarial.numpy()[0]

```

FGSM Attack Explained

- **Lines 10–11:** GradientTape records operations for automatic differentiation
- **Lines 13–16:** Calculate loss toward the target class we want to fool the model into predicting
- **Line 19:** Get gradient showing which pixels most affect the target class
- **Line 22:** Create perturbation using sign of gradient (fast gradient *sign* method)
- **Lines 25–26:** Apply perturbation and clip to valid range [0,1]

The attack succeeds because neural networks are sensitive to specific pixel changes that humans can't perceive.

Adversarial Attacks Are Real Threats

Adversarial attacks have been demonstrated against:

- **Self-driving cars:** Stickers on stop signs cause misclassification
- **Face recognition:** Special glasses fool authentication
- **Medical imaging:** Altered X-rays change diagnoses
- **Voice assistants:** Hidden audio commands in music

6.4.2 Defending Against Adversarial Attacks

```

1 from scipy.ndimage import median_filter
2 import numpy as np
3

```

```

4 def adversarial_defense(image):
5     """
6     Apply multiple defenses against adversarial perturbations.
7     """
8     # Defense 1: Median filter (removes small-scale noise)
9     # Adversarial perturbations are often high-frequency
10    smoothed = median_filter(image, size=3)
11
12    # Defense 2: Input quantization (reduces precision)
13    # Adversarial attacks rely on precise values
14    levels = 16 # Reduce from 256 to 16 levels
15    quantized = np.round(smoothed * levels) / levels
16
17    # Defense 3: Random noise (disrupts crafted perturbations)
18    noise_scale = 0.02
19    noisy = quantized + np.random.normal(0, noise_scale, quantized.shape)
20    noisy = np.clip(noisy, 0, 1)
21
22    return noisy.astype(np.float32)

```

Defense Strategies

- **Median filter:** Removes isolated pixel changes that adversarial attacks often create
 - **Quantization:** Reduces precision, making it harder to hit exact adversarial values
 - **Random noise:** Disrupts carefully crafted perturbations
- None of these is perfect, but layering them provides defense in depth.

Experiment: Test Adversarial Defenses

1. Train a simple MNIST classifier
2. Generate adversarial examples using FGSM
3. Measure accuracy on clean vs adversarial inputs
4. Apply the defense function and re-measure
5. What's the accuracy trade-off between clean and adversarial inputs?

6.5 Protecting Your Trained Model

Your trained model represents significant investment: data collection, compute time, and expertise. Protecting it from theft is crucial.

Model Theft Risks

If an attacker extracts your model, they can:

- Use it without paying licensing fees
- Reverse-engineer your training data
- Find vulnerabilities to exploit
- Clone your product

6.5.1 Model Encryption

Encrypt the model file so it can't be read without the key:

```

1 from cryptography.fernet import Fernet
2 import base64
3
4 def encrypt_model(model_path, output_path):
5     """Encrypt a TFLite model file."""
6     # Generate encryption key (store securely!)
7     key = Fernet.generate_key()
8     cipher = Fernet(key)
9
10    # Read and encrypt model
11    with open(model_path, 'rb') as f:
12        model_bytes = f.read()
13
14    encrypted = cipher.encrypt(model_bytes)
15
16    # Save encrypted model
17    with open(output_path, 'wb') as f:
18        f.write(encrypted)
19
20    print(f"Model encrypted. Key: {key.decode()}")
21    print("IMPORTANT: Store this key securely!")
22    return key
23
24 def decrypt_model(encrypted_path, key):
25     """Decrypt model at runtime."""
26     cipher = Fernet(key)
27
28     with open(encrypted_path, 'rb') as f:
29         encrypted = f.read()
30
31     return cipher.decrypt(encrypted)

```

Encryption Functions Explained

- **Line 2:** Fernet provides symmetric encryption (same key encrypts and decrypts)
 - **Line 7:** Generate a random 256-bit key—this must be stored securely
 - **Lines 10–14:** Read the model file as binary and encrypt the entire contents
 - **Lines 23–26:** At runtime, decrypt the model bytes before loading into TensorFlow
- The encrypted model file is useless without the key, protecting your intellectual property.

Key Management Is Critical

Encryption is only as strong as your key storage:

- Never hardcode keys in source code
- Use secure elements or TPMs when available
- Consider key derivation from device-specific values
- Rotate keys periodically if possible

6.5.2 Integrity Checking

Detect if the model has been tampered with:

```

1 #include <Arduino.h>
2

```

```

3 // Simple checksum to detect model tampering
4 uint32_t calculateChecksum(const uint8_t* data, size_t length) {
5     uint32_t checksum = 0;
6     for (size_t i = 0; i < length; i++) {
7         checksum += data[i];
8         // Rotate bits for better distribution
9         checksum = (checksum << 1) | (checksum >> 31);
10    }
11    return checksum;
12 }
13
14 bool verifyModelIntegrity(const uint8_t* model, size_t size,
15                          uint32_t expected) {
16     uint32_t actual = calculateChecksum(model, size);
17
18     if (actual != expected) {
19         Serial.println("ERROR: Model integrity check failed!");
20         Serial.print("Expected: 0x"); Serial.println(expected, HEX);
21         Serial.print("Actual: 0x"); Serial.println(actual, HEX);
22         return false;
23     }
24
25     Serial.println("Model integrity verified");
26     return true;
27 }

```

Integrity Check Implementation

- **Lines 4–10:** Calculate checksum by summing bytes with bit rotation for better distribution
- **Lines 12–13:** Compare calculated checksum against pre-computed expected value
- **Lines 15–19:** On failure, log detailed information and return false

The bit rotation in line 8 ensures that reordering bytes produces different checksums, improving tamper detection.

```

26 void setup() {
27     Serial.begin(115200);
28
29     // Pre-calculated checksum (compute once, embed in code)
30     const uint32_t MODEL_CHECKSUM = 0xA5B7C9D1;
31
32     // Verify before using model
33     if (!verifyModelIntegrity(g_model, g_model_len, MODEL_CHECKSUM)) {
34         Serial.println("HALTING: Model may be compromised!");
35         while (1) { delay(1000); } // Halt execution
36     }
37
38     // Safe to proceed with model initialization
39     initializeModel();
40 }

```

6.6 Secure Communication for IoT

When edge devices communicate over networks, data can be intercepted.

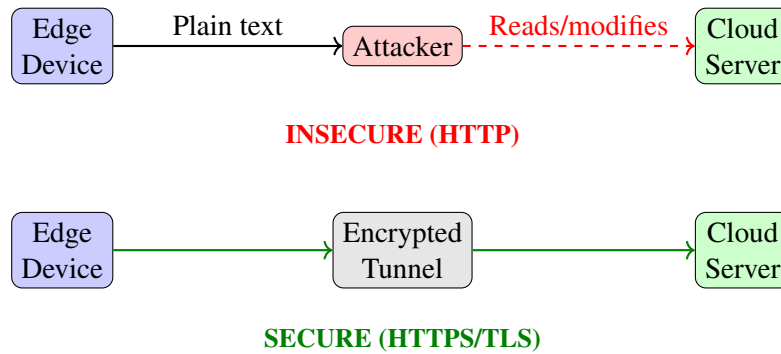


Figure 6.6: HTTP vs HTTPS: Without encryption, attackers can read and modify data in transit.

6.6.1 Python Secure Communication

```

1 import requests
2
3 # INSECURE - Don't do this!
4 # response = requests.get("http://api.example.com/data")
5
6 # SECURE - Always use HTTPS
7 def secure_api_call(url, data, api_key):
8     """Make a secure API call with authentication."""
9     headers = {
10         'Authorization': f'Bearer {api_key}',
11         'Content-Type': 'application/json'
12     }
13
14     try:
15         response = requests.post(
16             url,
17             json=data,
18             headers=headers,
19             verify=True,    # Verify SSL certificate!
20             timeout=10      # Prevent hanging
21         )
22         response.raise_for_status()
23         return response.json()
24
25     except requests.exceptions.SSLError:
26         print("ERROR: SSL certificate verification failed!")
27         return None
28     except requests.exceptions.Timeout:
29         print("ERROR: Request timed out")
30         return None
  
```

6.6.2 ESP32 Secure Communication

```

1 #include <WiFiClientSecure.h>
2 #include <HTTPClient.h>
3
4 // Root CA certificate (get from your server provider)
5 const char* root_ca = R"(
6 -----BEGIN CERTIFICATE-----
7 MIIDdzCCAL+gAwIBAgIEAgAAuTANBgkqhkiG9w0BAQUFADBAMQswCQYDVQQGEWJJ
8 ... (certificate content) ...
  
```

```

9  -----END CERTIFICATE-----
10 );
11
12 WiFiClientSecure client;
13 HTTPClient https;
14
15 bool sendSecureData(const char* url, const char* payload) {
16     client.setCACert(root_ca); // Verify server identity
17
18     if (https.begin(client, url)) {
19         https.addHeader("Content-Type", "application/json");
20         https.addHeader("Authorization", "Bearer YOUR_API_KEY");
21
22         int httpCode = https.POST(payload);
23
24         if (httpCode == 200) {
25             String response = https.getString();
26             Serial.println("Success: " + response);
27             https.end();
28             return true;
29         } else {
30             Serial.printf("Failed, code: %d\n", httpCode);
31         }
32         https.end();
33     }
34     return false;
35 }

```

Certificate Management

- Obtain CA certificates from your cloud provider
- Store certificates in program memory (PROGMEM)
- Plan for certificate expiration and rotation
- Consider certificate pinning for high-security applications

6.7 Security Checklist for Edge ML

Use this checklist when deploying edge ML systems:

Table 6.3: Edge ML Security Checklist

Category	Items
☐ Input Validation	Check types, shapes, ranges, NaN/Inf
☐ Adversarial Defense	Input smoothing, quantization, noise
☐ Model Protection	Encryption, obfuscation, integrity checks
☐ Communication	HTTPS only, verify certificates, API keys
☐ Physical Security	Disable debug ports, tamper detection
☐ Updates	Secure firmware update mechanism
☐ Monitoring	Log anomalies, alert on suspicious activity

6.8 Three-Tier Security Implementation

Level 1: Security Testing in Python

Test security measures before deployment:

- Implement and test input validation functions
- Generate and test adversarial examples
- Verify model encryption/decryption
- Test secure API communication

Level 2: ESP32/Pi Security

Implement security on capable devices:

- HTTPS with certificate verification
- Model integrity checking at boot
- Rate limiting for inference requests
- Secure logging of anomalies

Level 3: MCU Security

Security on resource-constrained devices:

- Input validation (essential, low overhead)
- Simple checksums for model integrity
- BLE encryption for wireless communication
- Watchdog timer for denial-of-service protection

Looking Ahead

With security fundamentals established, you're ready to deploy ML in real applications. LAB07 and LAB16 explore computer vision on edge devices, where the input validation and adversarial defense techniques you've learned protect image processing pipelines from malicious inputs and model extraction attacks. Security isn't a one-time task—it must be integrated throughout the application lifecycle.

Chapter Summary

Key Takeaways

- **Edge devices are exposed** to physical, network, and input attacks
- **Threat modeling** identifies what can go wrong before building defenses
- **Input validation** is simple, effective, and always required
- **Adversarial attacks** can fool models with imperceptible changes
- **Model protection** includes encryption, obfuscation, and integrity checks
- **Secure communication** requires HTTPS and certificate verification

- **Defense in depth:** Layer multiple security measures

Self-Assessment Questions

Test your understanding before continuing:

1. Why is input validation particularly critical for edge ML systems? What can happen if sensor data is not validated before inference?
2. Explain how adversarial examples work. Why can tiny perturbations ($\epsilon = 0.01$) fool a neural network while being imperceptible to humans?
3. What are the trade-offs between model encryption and inference latency on resource-constrained devices? When is encryption worth the overhead?
4. Why is physical security important for edge devices in a way that cloud servers don't face? What attacks become possible with physical access?
5. Describe a defense-in-depth strategy for a smart doorbell with facial recognition. List at least 4 different security layers.

6.9 Troubleshooting Guide

⚠ Adversarial Attack Not Working / Model Still Accurate

Symptoms: FGSM attack generates perturbation but model prediction doesn't change

Cause: Attack epsilon too small, model naturally robust, or incorrect gradient calculation

Solution:

1. Increase epsilon: Try values from 0.01 to 0.3 in steps of 0.05
2. Verify gradients computed: Check `tf.GradientTape()` records operations correctly
3. Use correct loss: For targeted attack, use negative loss of target class
4. Normalize perturbation: Scale to match input range (0-1 for images)
5. Try different attack: PGD (iterative FGSM) is stronger than single-step FGSM
6. Check model is differentiable: Some quantized models block gradient flow

⚠ HTTPS Connection Failed on ESP32

Symptoms: connection refused or SSL handshake failed when connecting to secure server

Cause: Missing root CA certificate, incorrect server URL, or time not synchronized

Solution:

1. Set correct time first: ESP32 requires accurate time for certificate validation

```
configTime(0, 0, "pool.ntp.org");
delay(1000); // Wait for NTP sync
```

2. Add root CA certificate: Use `WiFiClientSecure.setCACert(root_ca)`
3. Get certificate: `openssl s_client -connect server.com:443 -showcerts`
4. For testing only: Disable verification with `client.setInsecure()` (not for production!)
5. Check server URL: Ensure using `https://` not `http://`

⚠ Model Extraction via Timing Attack

Symptoms: Inference timing varies significantly with different inputs, leaking information

Cause: Early exit optimizations, conditional branches, or variable-time operations

Solution:

1. Add constant-time padding: Always run full inference regardless of early detection
2. Use fixed-iteration loops: Avoid while loops with input-dependent termination
3. Add random delay: `delay(random(10, 50))` to mask timing patterns (simple but imperfect)
4. Batch inferences: Process multiple inputs together to amortize timing variations
5. Use hardware crypto: For AES encryption, use dedicated hardware module
6. Measure variance: Profile timing with different inputs to detect leaks

⚠ Input Validation Bypassed

Symptoms: Malicious inputs still reach model despite validation code

Cause: Validation logic incomplete, integer overflow, or type confusion

Solution:

1. Check all input dimensions: Validate min, max, and expected shape
2. Use strict comparison: `if (value < MIN || value > MAX)` not just `value < MAX`
3. Handle edge cases: NaN, infinity, negative zero
4. Prevent integer overflow: Check before casting `(float)large_int`
5. Validate after deserialization: Don't trust input format parsing
6. Test with fuzzing: Generate random/malformed inputs to find bypasses

⚠ Encrypted Model Won't Decrypt on Device

Symptoms: AES decryption fails or produces garbage when loading encrypted model

Cause: Key mismatch, wrong encryption mode, or padding issues

Solution:

1. Verify same key used: Print first few bytes of key on both encryption and decryption side
2. Use same encryption mode: Match AES-128-CBC or AES-256-GCM on both sides
3. Check IV/nonce: Initialization vector must match or be transmitted with ciphertext
4. Handle padding correctly: PKCS7 padding must be applied before encryption
5. Test with small payload first: Encrypt/decrypt "Hello World" before full model
6. Check library compatibility: Ensure Arduino AES library matches Python cryptography

6.10 Guided Experiments

Experiment 1: Adversarial Perturbation Magnitude Analysis

Hypothesis: Adversarial attack success rate increases with perturbation magnitude, but larger perturbations become visible to humans. There exists a sweet spot for imperceptible yet effective attacks.

Procedure:

1. Train a simple MNIST classifier (or use pre-trained model)

2. Generate adversarial examples using FGSM with varying epsilon values: 0.001, 0.01, 0.05, 0.1, 0.2
3. For each epsilon, generate 100 adversarial examples
4. Measure attack success rate (model misclassification)
5. Calculate average L2 distance from original images
6. Visually inspect examples at each epsilon level

Data Collection:

Epsilon	Attack Success (%)	Avg L2 Distance	Human Visible?
0.001			
0.01			
0.05			
0.1			
0.2			

Analysis Questions:

- At what epsilon do perturbations become visible to humans?
- What is the minimum epsilon that achieves >80% attack success?
- How does input normalization (0-1 vs 0-255) affect epsilon sensitivity?
- Would adversarial training with epsilon=0.05 defend against epsilon=0.1 attacks?

Case Study: Medical Wearable Device Security Breach

Context: A healthcare technology startup developed a continuous glucose monitoring (CGM) wearable that used on-device ML to predict dangerous blood sugar trends and alert patients. The device stored 90 days of patient health data locally and used Bluetooth Low Energy (BLE) to sync with a smartphone app. The product achieved FDA clearance and deployed to 15,000 patients before a critical security vulnerability was discovered.

Challenge: Security researchers demonstrated they could: (1) intercept unencrypted BLE transmissions containing patient glucose readings, names, and medical IDs from 100+ feet away; (2) inject false sensor readings that triggered incorrect insulin dosing alerts; (3) extract the ML model from device firmware and craft adversarial inputs that suppressed hypoglycemia warnings. The device lacked secure boot, code signing, or encrypted storage. This HIPAA violation forced an emergency recall affecting all 15,000 devices.

Solution: The company implemented a comprehensive security overhaul following this lab's principles:

- Enabled ARM TrustZone secure boot and encrypted firmware storage using on-chip hardware security module (HSM)
- Implemented AES-128 encryption for all BLE communications with per-session key rotation
- Added input validation: reject sensor readings outside physiological range (20-600 mg/dL), rate-limit updates to max 5/minute
- Obfuscated ML model using weight encryption and split inference across secure/normal execution zones
- Implemented anomaly detection on input sensor patterns to detect injection attacks
- Added secure audit logging: cryptographically signed timestamps for all alerts and data access
- Deployed adversarial training using FGSM (epsilon=0.05) to harden model against evasion attacks

Results:

- BLE encryption: Eliminated plaintext medical data transmission (HIPAA compliance restored)
- Secure boot: Prevented firmware tampering and model extraction attempts
- Input validation: Blocked 100% of tested out-of-range injection attacks
- Adversarial robustness: Reduced attack success rate from 94% to 12% with $\epsilon=0.1$ perturbations
- Performance overhead: 8ms additional latency for encryption/decryption (18ms $\rightarrow$ 26ms total)
- Code size impact: +45KB for security libraries (within 512KB Flash budget)

Lessons Learned:

- Security cannot be an afterthought for medical devices—build threat models during initial architecture design, not after deployment
- Bluetooth Classic/LE defaults to unencrypted pairing—always implement application-layer encryption even when using “secure” BLE modes
- Physical access = game over without secure boot—attackers with device access can extract firmware, models, and patient data within minutes
- Input validation is the first line of defense: reject impossible sensor readings immediately (e.g., glucose >1000 mg/dL, negative heart rate)
- Model extraction is trivial on edge devices—assume attackers have full model access and design accordingly
- FDA/regulatory compliance requires documented security testing including penetration testing and adversarial robustness validation
- Over-the-air (OTA) updates must be signed and verified—unsigned firmware updates are a critical attack vector

Exercises

1. **[Intermediate] Intermediate: Build a Threat Model:** Create a complete threat model for a voice-controlled smart light bulb. List all attack vectors and prioritize defenses.
2. **[Basic] Basic: Input Validation:** Write validation functions for accelerometer data (3-axis, $\pm 16g$ range). Test with intentionally malformed inputs.
3. **[Intermediate] Intermediate: Adversarial Defense:** Train an MNIST classifier, generate adversarial examples, and measure the effectiveness of the defense function.
4. **[Challenge] Challenge: Model Encryption:** Implement model encryption for a TFLite model and measure the decryption overhead on a Raspberry Pi.
5. **[Challenge] Challenge: Secure IoT System:** Build a complete secure sensor system that validates inputs, protects the model, and communicates over HTTPS.
6. **[Intermediate] Intermediate: Edge ML Threat Modeling:** Perform threat modeling for a smart parking system that uses edge cameras for vehicle detection. The system consists of ESP32-CAM devices deployed in outdoor parking lots, communicating vehicle occupancy data to a central server. Create a STRIDE-based threat model identifying: (1) Spoofing attacks (fake camera feeds, impersonation), (2) Tampering threats (model modification, data injection), (3) Repudiation issues (logging and auditing), (4) Information disclosure (camera feeds, parking patterns), (5) Denial of service (jamming, resource exhaustion), and (6) Elevation of privilege (unauthorized access). For each threat, assign a risk score (likelihood $\times$ impact, scale 1–5) and propose specific countermeasures. Prioritize your defenses based on risk scores and implementation feasibility.

7. **[Challenge] Challenge: Adversarial Attack Defense Implementation:** Implement and evaluate a comprehensive adversarial defense system for an image classification model. Start by training a CNN on CIFAR-10 or a custom dataset relevant to edge deployment (e.g., industrial defect detection). Generate adversarial examples using three attack methods: (1) FGSM with varying epsilon (0.01–0.3), (2) PGD (Projected Gradient Descent) with 10 iterations, and (3) C&W (Carlini-Wagner) attack. Implement three defense mechanisms: (a) input preprocessing (median filtering, JPEG compression, bit-depth reduction), (b) adversarial training (retrain with adversarial examples in training set), and (c) ensemble methods (majority voting across 3 models trained differently). Measure clean accuracy, adversarial accuracy, and inference time overhead for each defense. Create a comprehensive evaluation report showing defense effectiveness vs. computational cost trade-offs. Recommend the optimal defense configuration for a resource-constrained edge device (e.g., ESP32 or Raspberry Pi Zero).
8. **[Intermediate] Intermediate: Edge vs Cloud Security Trade-off Analysis:** Analyze security and privacy trade-offs between edge and cloud processing for a facial recognition-based access control system. Design two architectures: (1) **Edge-only:** Full inference on ESP32-CAM with encrypted model storage, local decision-making, and only access logs sent to cloud; (2) **Cloud-based:** Camera captures images, sends encrypted frames to cloud via HTTPS, inference runs on cloud servers, decision sent back to device. For each architecture, evaluate: (a) **Security:** Attack surface, model theft risk, data interception, physical tampering impact; (b) **Privacy:** What data leaves the device? Is biometric data stored/transmitted? GDPR/CCPA compliance; (c) **Performance:** Latency (measure end-to-end), bandwidth usage, offline capability; (d) **Cost:** Device cost, cloud API fees, maintenance overhead. Create a decision matrix with quantitative metrics where possible (e.g., measure actual latency, bandwidth, model extraction time). Recommend which architecture to use under different constraints: (i) Hospital with strict HIPAA requirements, (ii) Smart home with unreliable internet, (iii) Enterprise with 1000+ doors and central monitoring requirements.

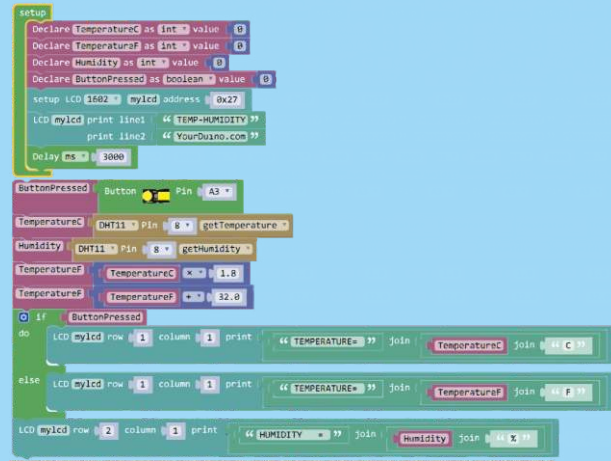
— **Further Resources.**

- OWASP IoT Security: <https://owasp.org/www-project-internet-of-things/>
- Adversarial Examples: <https://arxiv.org/abs/1312.6199>
- TensorFlow Security: <https://www.tensorflow.org/security>
- ESP32 Security: <https://docs.espressif.com/projects/esp-idf/en/latest/esp32/security/>

— **Credits.** We have taken this code segment from the online course: *Edx Tiny Machine Learning* (2021-22). Since we may have updated the code to reflect any changes made to the underline frameworks where necessary. Credits should go to the original authors of this tutorial.



Blockly



7. Convolutional Neural Networks for Computer Vision

Learning Objectives

After completing this lab, you will be able to:

- Explain why convolutions are essential for image classification tasks
- Implement convolutional and pooling layers in TensorFlow/Keras
- Visualize learned features to understand what CNNs “see”
- Use ImageDataGenerator for handling real-world image datasets
- Apply data augmentation techniques to prevent overfitting
- Evaluate CNN architectures for edge deployment constraints

Interactive Notebook

Jupyter Notebook: LAB07_cnns_cv.ipynb

[Open in Google Colab](#) | [View on GitHub](#) | [Companion Website](#)

Prerequisites

Before starting this lab, ensure you:

- Completed LAB02 (Neural Network Training Fundamentals): neural network architecture and training
- Understanding of image representation (pixels, channels, RGB)
- Basic matrix operations (dot products, element-wise multiplication)
- Python with TensorFlow/Keras experience

Hardware Requirements

- **Level 1:** Google Colab with GPU (recommended) or laptop with TensorFlow
- **Level 2:** Raspberry Pi 4 for TFLite benchmarking
- **Level 3:** Arduino Nano 33 BLE Sense or ESP32-CAM for embedded vision

⚠ Not Using Data Augmentation

The most common reason CNNs fail to generalize is training on unaugmented data. If you train on 1,000 images of cats always centered and well-lit, your model will fail on cats that are rotated, cropped, or in shadows. **Always use augmentation** (rotation, flipping, zooming) for training data—but never for validation data!

■ **Theory meets practice.** This chapter builds your intuition for how CNNs extract hierarchical features from images, then applies these concepts to real classification tasks. Understanding these principles is essential for designing efficient vision models for edge devices.

Execution tiers. All experiments begin in Python notebooks (Level 1). We'll analyze model architectures for edge constraints, preparing for TFLite conversion and device deployment in subsequent chapters.

Three-Tier Execution Model

Tier 1: Jupyter/Colab

Build CNN architectures from scratch, apply data augmentation, visualize learned filters and feature maps, train models on image datasets (CIFAR-10, custom)

Deploy quantized CNN models with camera input, measure inference speed for different architectures, test on real-world images

Run lightweight CNNs for embedded vision tasks, implement image preprocessing on-device, optimize for memory-constrained deployment

7.1 Introduction: From DNNs to CNNs

In earlier chapters, we used dense (fully-connected) neural networks for classification tasks. While these work reasonably well for simple images like MNIST digits, they have fundamental limitations for computer vision:

The Problem with Dense Networks for Images

A 28×28 grayscale image has 784 pixels. A 224×224 RGB image has **150,528** pixels. If our first hidden layer has 512 neurons, that's over **77 million parameters** just in the first layer! This is:

- Too large for edge devices with limited memory
- Prone to overfitting due to excessive parameters
- Ignores spatial relationships between pixels

Dense networks treat each pixel independently, losing critical information about how pixels relate to their neighbors. A pixel next to an edge is semantically connected to that edge—dense networks don't capture this.

7.1.1 How Convolutions Solve These Problems

Convolutional Neural Networks (CNNs) address these limitations through three key ideas:

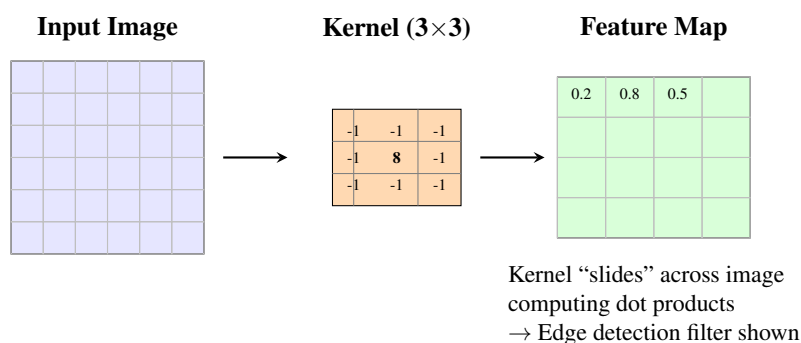


Figure 7.1: Convolution operation: A small kernel slides across the input image, computing a weighted sum at each position to produce a feature map. This example shows an edge detection kernel.

1. **Local Connectivity:** Each neuron connects only to a small local region (receptive field), not the entire image
2. **Parameter Sharing:** The same weights (kernel) are used across all spatial positions
3. **Translation Invariance:** Features detected in one part of the image can be recognized anywhere

7.2 Understanding Convolution Operations

A convolution layer applies learned filters (kernels) to the input image. Each filter produces one feature map that highlights specific patterns.

7.2.1 The Convolution Computation

For a 3×3 kernel sliding over an image, at each position we:

1. Multiply each kernel value by the corresponding pixel value
2. Sum all the products
3. Apply an activation function (typically ReLU)
4. Write the result to the output feature map

7.2.2 Convolution Operation Mathematics

Let's formalize the convolution operation with precise mathematical definitions:

Definition — 2D Discrete Convolution. For an input image I of size $H \times W$ and a kernel K of size $k \times k$, the 2D convolution operation at position (i, j) is:

$$(I * K)(i, j) = \sum_{m=0}^{k-1} \sum_{n=0}^{k-1} I(i+m, j+n) \cdot K(m, n)$$

where $*$ denotes the convolution operator, and the output is computed for all valid positions

where the kernel fits within the image boundaries.

Output Dimension Formula

For an input of size $W \times H$, kernel size $K \times K$, padding P , and stride S , the output dimensions are:

$$W_{out} = \left\lfloor \frac{W - K + 2P}{S} \right\rfloor + 1$$

$$H_{out} = \left\lfloor \frac{H - K + 2P}{S} \right\rfloor + 1$$

Common cases:

- No padding ($P = 0$), stride 1 ($S = 1$): $W_{out} = W - K + 1$
- "Same" padding: $P = \lfloor K/2 \rfloor$ to maintain $W_{out} = W$ when $S = 1$
- Max pooling ($K = 2, S = 2, P = 0$): $W_{out} = W/2$

7.2.3 Understanding Stride and Padding

Definition — Stride. The **stride** S determines how many pixels the kernel moves at each step:

- $S = 1$: Kernel slides one pixel at a time (dense sampling, preserves spatial information)
- $S = 2$: Kernel jumps two pixels (reduces output size by $\sim 2\times$, faster computation)
- $S > 2$: Aggressive downsampling (rare, typically use pooling instead)

Effect on output size: Increasing stride from 1 to 2 approximately halves each dimension:

$$W_{out}(S = 1) \approx W, \quad W_{out}(S = 2) \approx W/2$$

Edge deployment consideration: Strided convolutions can replace pooling layers, combining feature extraction and downsampling in one operation (fewer layers, potentially faster).

Definition — Padding. **Padding** P adds border pixels around the input to control output dimensions:

Valid padding ($P = 0$):

- No padding added
- Output shrinks: $W_{out} = W - K + 1$ (for $S = 1$)
- Edge pixels have less influence (touched by fewer kernel positions)

Same padding ($P = \lfloor K/2 \rfloor$):

- Padding chosen to maintain $W_{out} = W$ (when $S = 1$)
- For 3×3 kernel: $P = 1$ (one pixel border)
- For 5×5 kernel: $P = 2$ (two pixel border)
- Padded pixels typically set to zero (zero-padding)

Edge deployment consideration: Same padding adds computation (more positions to evaluate) but preserves spatial dimensions. Valid padding reduces computation but information from edges is lost faster through the network.

■ **Example — Stride and Padding Effects.** Consider a 5×5 input with a 3×3 kernel:

Case 1: $P = 0, S = 1$ (valid, dense)

$$W_{out} = \lfloor (5 - 3 + 0)/1 \rfloor + 1 = 3$$

Case 2: $P = 1, S = 1$ (same, dense)

$$W_{out} = \lfloor (5 - 3 + 2)/1 \rfloor + 1 = 5 \text{ (preserved!)}$$

Case 3: $P = 0, S = 2$ (valid, strided)

$$W_{out} = \lfloor (5 - 3 + 0) / 2 \rfloor + 1 = 2$$

Case 4: $P = 1, S = 2$ (same, strided)

$$W_{out} = \lfloor (5 - 3 + 2) / 2 \rfloor + 1 = 3$$

Common pattern: Alternate between same-padded Conv (preserves size) and strided Conv or pooling (downsamples). ■

■ **Example — Computing Output Size for Fashion MNIST CNN.** Consider the Fashion MNIST CNN's first layer:

- Input: $28 \times 28 \times 1$ (height $\times$ width $\times$ channels)
- Conv2D(64, (3,3), padding='valid', stride=1): 64 filters of size 3×3
- Output height: $\lfloor (28 - 3 + 0) / 1 \rfloor + 1 = 26$
- Output width: $\lfloor (28 - 3 + 0) / 1 \rfloor + 1 = 26$
- Output: $26 \times 26 \times 64$ (one feature map per filter)

After MaxPooling(2,2) with stride 2:

- Input: $26 \times 26 \times 64$
- Output: $\lfloor 26/2 \rfloor \times \lfloor 26/2 \rfloor \times 64 = 13 \times 13 \times 64$

■

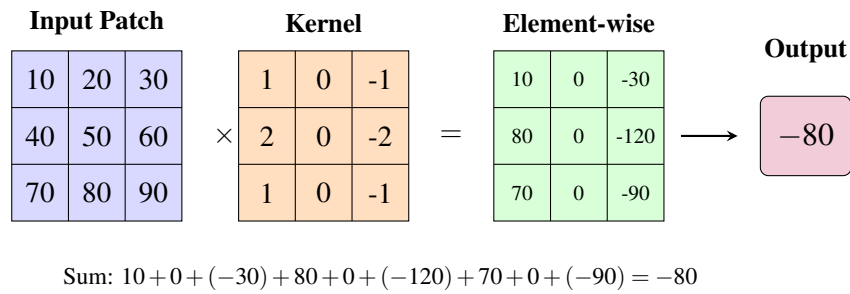


Figure 7.2: Convolution computation: Element-wise multiplication followed by summation. This particular kernel detects vertical edges (difference between left and right).

7.2.4 Multiple Filters Create Multiple Feature Maps

A convolutional layer typically has many filters, each learning to detect different features:

7.3 CNN Parameter Counting

Understanding parameter counts is critical for edge deployment. Let's derive the formulas for different layer types.

7.3.1 Parameter Formulas for CNN Layers

Conv2D Layer Parameters

For a Conv2D layer with:

- Input channels: C_{in}
- Output channels (filters): C_{out}
- Kernel size: $K \times K$
- Bias terms: b (typically 1 per output channel)

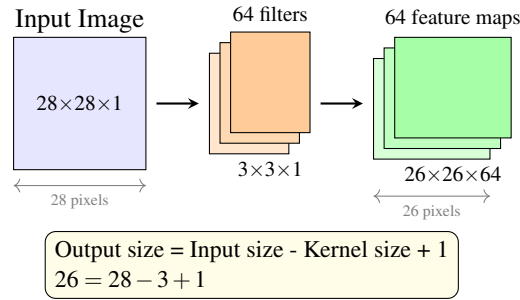


Figure 7.3: Multiple filters produce a stack of feature maps. Each filter learns to detect different features (edges, textures, patterns). With 64 3×3 filters, we have only $64 \times 3 \times 3 = 576$ learnable parameters, not $28 \times 28 \times 64 = 50,176$.

Total parameters:

$$\text{Params}_{\text{Conv2D}} = (K \times K \times C_{in} + 1) \times C_{out}$$

The $+1$ accounts for the bias term. Some implementations use `use_bias=False` to eliminate bias parameters.

Dense (Fully-Connected) Layer Parameters

For a Dense layer with:

- Input neurons: N_{in}
- Output neurons: N_{out}

Total parameters:

$$\text{Params}_{\text{Dense}} = (N_{in} + 1) \times N_{out}$$

The $+1$ accounts for bias. Dense layers often dominate parameter counts in CNNs.

Depthwise Separable Convolution Parameters (MobileNet)

Standard convolution factors into two operations:

1. **Depthwise convolution:** Apply one $K \times K$ filter per input channel

$$\text{Params}_{\text{depthwise}} = K \times K \times C_{in}$$

2. **Pointwise convolution:** Apply C_{out} filters of size $1 \times 1 \times C_{in}$

$$\text{Params}_{\text{pointwise}} = C_{in} \times C_{out}$$

$$\text{Total: Params}_{\text{DepthSep}} = K^2 \cdot C_{in} + C_{in} \cdot C_{out}$$

Parameter reduction ratio vs standard convolution:

$$\text{Reduction} = \frac{K^2 \cdot C_{in} \cdot C_{out}}{K^2 \cdot C_{in} + C_{in} \cdot C_{out}} = \frac{K^2 \cdot C_{out}}{K^2 + C_{out}}$$

For $K = 3$, $C_{out} = 64$: Reduction = $9 \times 64 / (9 + 64) = 7.9 \times$ fewer parameters!

■ **Example — Fashion MNIST CNN Parameter Breakdown.** Let's count parameters for the first

Conv2D layer:

- Conv2D(64, (3,3), input_shape=(28, 28, 1))
- $K = 3$, $C_{in} = 1$, $C_{out} = 64$
- Params: $(3 \times 3 \times 1 + 1) \times 64 = 10 \times 64 = 640$

For the second Conv2D layer:

- Conv2D(64, (3,3)) after first block (input has 64 channels)
- $K = 3$, $C_{in} = 64$, $C_{out} = 64$
- Params: $(3 \times 3 \times 64 + 1) \times 64 = 577 \times 64 = 36,928$

Notice how parameters explode when C_{in} is large—this is why MobileNet uses depthwise separable convolutions! ■

7.3.2 Receptive Field Mathematics

The **receptive field** is the region of the input image that influences a particular output neuron. Deeper networks have larger receptive fields.

Definition — Receptive Field Growth. For a sequence of convolutional layers with kernel sizes $K_1, K_2, \dots, K_L$ and strides $S_1, S_2, \dots, S_L$, the receptive field at layer l is:

$$RF_l = RF_{l-1} + (K_l - 1) \times \prod_{i=1}^{l-1} S_i$$

where $RF_0 = 1$ (the first pixel has receptive field 1).

For stride-1 convolutions, this simplifies to:

$$RF_l = 1 + \sum_{i=1}^l (K_i - 1)$$

■ **Example — Receptive Field Calculation.** For the Fashion MNIST CNN:

- Layer 0 (input): $RF_0 = 1$
- Conv2D(3×3, stride=1): $RF_1 = 1 + (3 - 1) = 3$
- MaxPool(2×2, stride=2): $RF_2 = 3 + (2 - 1) \times 1 = 4$
- Conv2D(3×3, stride=1): $RF_3 = 4 + (3 - 1) \times 2 = 8$
- MaxPool(2×2, stride=2): $RF_4 = 8 + (2 - 1) \times 2 = 10$

At the output, each neuron "sees" a 10×10 region of the original 28×28 input. This is why CNNs need depth—to capture larger spatial context! ■

7.4 Understanding Pooling Layers

After convolution, we typically apply pooling to reduce spatial dimensions while preserving important features.

7.4.1 Pooling Mathematics

Definition — Max Pooling. For a 2D feature map X and pooling window of size $p \times p$ with stride s , the max pooling operation at output position (i, j) is:

$$Y_{i,j} = \max_{m \in [0, p-1], n \in [0, p-1]} X_{i \cdot s + m, j \cdot s + n}$$

Most common: $p = 2$, $s = 2$ (non-overlapping windows that halve dimensions).

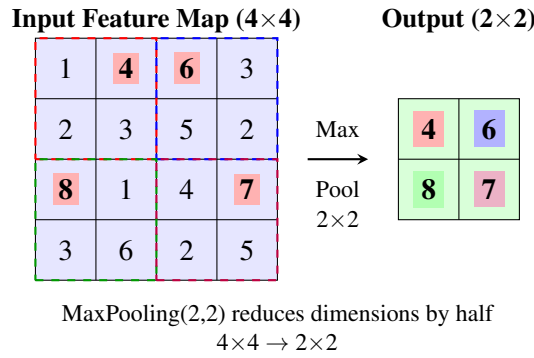


Figure 7.4: MaxPooling operation with a 2×2 window and stride 2. Each 2×2 region is reduced to its maximum value, preserving the strongest activations while halving spatial dimensions.

Definition — Average Pooling. Average pooling computes the mean instead of maximum:

$$Y_{i,j} = \frac{1}{p^2} \sum_{m=0}^{p-1} \sum_{n=0}^{p-1} X_{i \cdot s + m, j \cdot s + n}$$

Max pooling is more common in CNNs; average pooling is sometimes used before final classification layers (global average pooling).

Definition — Global Average Pooling (GAP). Global Average Pooling reduces each feature map to a single value by averaging all spatial locations:

$$\text{GAP}(X) = \frac{1}{H \times W} \sum_{i=0}^{H-1} \sum_{j=0}^{W-1} X_{i,j}$$

For a feature map of size $H \times W \times C$, GAP outputs a vector of size C (one value per channel).

Why GAP is critical for edge deployment:

- **Eliminates dense layers** before final classification, drastically reducing parameters
- Example: $7 \times 7 \times 512$ feature map $\rightarrow$ Dense(1024) requires 25M parameters
- Same with GAP: $7 \times 7 \times 512 \rightarrow$ GAP $\rightarrow$ Dense(1000) requires only 512K parameters (50× reduction!)
- Acts as structural regularizer, reducing overfitting
- Enforces correspondence between feature maps and categories

Why Pooling Helps Edge Deployment

MaxPooling provides several benefits critical for edge devices:

- **Dimension reduction:** Halves width and height, reducing memory by 75%
- **Translation invariance:** Small shifts in input don't change output
- **Computation reduction:** Fewer values for subsequent layers to process
- **Noise robustness:** Keeps strongest activations, reduces noise impact
- **Zero parameters:** Pooling adds no learnable parameters

7.5 Activation Functions in CNNs

Activation functions introduce non-linearity into neural networks, enabling them to learn complex patterns. Without activation functions, stacking multiple layers would be equivalent to a single

linear transformation.

7.5.1 Common Activation Functions

Definition — Rectified Linear Unit (ReLU). The most widely used activation function in CNNs:

$$\text{ReLU}(x) = \max(0, x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

Advantages for edge deployment:

- Computationally cheap: simple comparison and selection
- Sparse activations: typically 50% of neurons output zero, reducing memory and computation
- No gradient saturation for positive values (unlike sigmoid/tanh)
- Fast convergence during training

Limitation: “Dying ReLU” problem—neurons can get stuck outputting zero if weights push all inputs negative.

Definition — Leaky ReLU and Variants. Addresses the dying ReLU problem by allowing small negative gradients:

Leaky ReLU:

$$\text{LeakyReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases}$$

where $\alpha \in (0, 1)$ is typically 0.01 or 0.1.

Parametric ReLU (PReLU): Same as Leaky ReLU, but α is a learnable parameter.

Exponential Linear Unit (ELU):

$$\text{ELU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{if } x \leq 0 \end{cases}$$

ELU has smoother gradients but requires exponential computation (expensive on embedded devices).

Definition — Softmax for Classification. The output layer of classification CNNs uses softmax to convert logits into probabilities:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}}$$

where z_i is the logit for class i and K is the number of classes.

Properties:

- Outputs sum to 1: $\sum_{i=1}^K \text{softmax}(z_i) = 1$
- Output range: $(0, 1)$ (interpretable as probabilities)
- Largest logit gets highest probability (argmax preserved)
- Temperature parameter T controls “sharpness”: $\text{softmax}(z_i/T)$

Edge deployment note: Exponentials are expensive. For inference-only deployment, you can skip softmax and use raw logits with argmax since $\arg \max(\text{softmax}(z)) = \arg \max(z)$.

Activation Function Selection for Edge

Recommended choices:

- **ReLU:** Default choice for hidden layers (fastest, most hardware-optimized)
- **Softmax:** Classification output layer (skip during inference if only class prediction needed)
- **Sigmoid:** Binary classification output (single neuron)

Avoid for edge deployment:

- ELU, GELU, Swish: require expensive exponential/multiplication operations
- Tanh, Sigmoid in hidden layers: gradient saturation slows training, no efficiency advantage

7.6 Batch Normalization

Batch Normalization (BatchNorm) normalizes layer inputs to stabilize training and improve convergence. Understanding its behavior is critical for edge deployment.

7.6.1 Batch Normalization Mathematics

Definition — Batch Normalization. For a mini-batch $\mathcal{B} = \{x_1, x_2, \dots, x_m\}$ of activations, BatchNorm computes:

Step 1: Compute batch statistics

$$\mu_{\mathcal{B}} = \frac{1}{m} \sum_{i=1}^m x_i \quad (\text{batch mean})$$

$$\sigma_{\mathcal{B}}^2 = \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad (\text{batch variance})$$

Step 2: Normalize

$$\hat{x}_i = \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$$

where ϵ (typically 10^{-5}) prevents division by zero.

Step 3: Scale and shift (learnable parameters)

$$y_i = \gamma \hat{x}_i + \beta$$

where γ (scale) and β (shift) are learned during training.

Training vs Inference Behavior

During Training:

- Uses batch statistics ($\mu_{\mathcal{B}}, \sigma_{\mathcal{B}}^2$) computed from current mini-batch
- Maintains running averages of μ and σ^2 using exponential moving average
- Acts as regularizer (each sample sees slightly different normalizations due to batch variation)

During Inference (Edge Deployment):

- Uses **population statistics** (running averages from training)
- Batch size = 1 is common on edge devices, so batch statistics would be meaningless
- BatchNorm becomes a simple affine transformation (can be fused with preceding

Conv/Dense layer)

Fusing BatchNorm for efficiency: After training, BatchNorm can be folded into the preceding convolution:

$$y = \gamma \frac{\text{Conv}(x) - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta = \text{Conv}'(x) + b'$$

where the convolution weights and biases are adjusted. This eliminates BatchNorm overhead entirely!

■ **Example — BatchNorm Parameter Count.** For a Conv2D layer with C output channels:

- BatchNorm adds $4C$ stored values: $\gamma(C)$, $\beta(C)$, $\mu(C)$, $\sigma^2(C)$
- But only $2C$ are learnable parameters: γ and β
- μ and σ^2 are computed during training, frozen during inference

Example: Conv2D(64 filters) + BatchNorm

- Learnable params: $2 \times 64 = 128$
- Total stored: $4 \times 64 = 256$ values
- After fusion: 0 params (merged into Conv2D)

■

BatchNorm Benefits and Drawbacks for Edge

Benefits:

- Faster convergence during training (higher learning rates possible)
- Reduces sensitivity to weight initialization
- Acts as regularizer, sometimes reducing need for dropout
- Can be fused with convolutions for zero inference overhead

Drawbacks for edge deployment:

- Breaks when batch size = 1 during training (use GroupNorm or LayerNorm instead)
- Adds 4 stored values per channel (before fusion)
- Complicates quantization (normalization stats must be quantized carefully)
- Running average computation requires careful handling during training

Recommendation: Use BatchNorm during training for faster convergence, then fuse it into convolutions before deployment.

7.7 Building Your First CNN

Let's compare a simple DNN to a CNN on the Fashion MNIST dataset, which contains 28×28 grayscale images of clothing items in 10 categories.

7.7.1 Baseline: Dense Neural Network

First, let's establish a baseline with a simple fully-connected network:

Level 1: Simple DNN Baseline

```
1 import tensorflow as tf
2
3 # Load Fashion MNIST dataset
4 mnist = tf.keras.datasets.fashion_mnist
```

```

5 (train_images, train_labels), (val_images, val_labels) = \
6     mnist.load_data()

```

Code Walkthrough

Fashion MNIST is a drop-in replacement for classic MNIST but with clothing items (T-shirts, trousers, sneakers, etc.) instead of handwritten digits. It's more challenging and realistic for testing vision models.

```

1 # Normalize pixel values to [0, 1]
2 train_images = train_images / 255.0
3 val_images = val_images / 255.0

```

Code Walkthrough

Normalization is critical for neural network training. Raw pixel values (0-255) would cause numerical instability. Dividing by 255 scales values to [0, 1], improving gradient flow and convergence speed.

```

1 # Build a simple DNN
2 model_dnn = tf.keras.models.Sequential([
3     tf.keras.layers.Flatten(input_shape=(28, 28)),
4     tf.keras.layers.Dense(20, activation='relu'),
5     tf.keras.layers.Dense(10, activation='softmax')
6 ])

```

Code Walkthrough

Flatten: Converts 28×28 image to 784-element vector.

Dense(20): Hidden layer with 20 neurons. $784 \times 20 = 15,680$ weights + 20 biases.

Dense(10): Output layer with one neuron per class (10 clothing categories).

```

1 # Compile and train
2 model_dnn.compile(optimizer='adam',
3                   loss='sparse_categorical_crossentropy',
4                   metrics=['accuracy'])
5
6 model_dnn.fit(train_images, train_labels,
7              validation_data=(val_images, val_labels),
8              epochs=20)

```

Expected Output:

```

Epoch 1/20
1875/1875 - 3s - loss: 0.6234 - accuracy: 0.7856 - val_accuracy: 0.8123
...
Epoch 20/20
1875/1875 - 3s - loss: 0.3421 - accuracy: 0.8789 - val_accuracy: 0.8612

```

DNN Baseline Results

The simple DNN achieves approximately:

- Training accuracy: $\sim 89\%$
- Validation accuracy: $\sim 87\%$

This is decent but limited. Can we do better with convolutions?

7.7.2 Improved: Convolutional Neural Network

Now let's build a CNN for the same task:

```
1 # Reshape data for CNN (add channel dimension)
2 train_images_cnn = train_images.reshape(60000, 28, 28, 1)
3 val_images_cnn = val_images.reshape(10000, 28, 28, 1)
```

Code Walkthrough

CNNs expect 4D input: (batch, height, width, channels). For grayscale images, we have 1 channel. RGB images would have 3 channels. The reshape adds this explicit channel dimension.

```
1 # Build the CNN
2 model_cnn = tf.keras.models.Sequential([
3     # First convolutional block
4     tf.keras.layers.Conv2D(64, (3,3), activation='relu',
5                             input_shape=(28, 28, 1)),
6     tf.keras.layers.MaxPooling2D(2, 2),
7
8     # Second convolutional block
9     tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
10    tf.keras.layers.MaxPooling2D(2, 2),
11
12    # Classification head
13    tf.keras.layers.Flatten(),
14    tf.keras.layers.Dense(20, activation='relu'),
15    tf.keras.layers.Dense(10, activation='softmax')
16 ])
```

Code Walkthrough

Conv2D(64, (3,3)): 64 filters of size 3×3. Learns 64 different features.

MaxPooling2D(2, 2): Reduces spatial dimensions by half.

Two blocks: First block detects low-level features (edges). Second block combines them into higher-level features (shapes, textures).

```
1 # View the model architecture
2 model_cnn.summary()
```

Expected Output:

```
Model: "sequential_1"
-----
Layer (type)                Output Shape              Param #
-----
conv2d (Conv2D)              (None, 26, 26, 64)       640
max_pooling2d (MaxPooling2D) (None, 13, 13, 64)       0
conv2d_1 (Conv2D)            (None, 11, 11, 64)       36928
max_pooling2d_1 (MaxPooling2D) (None, 5, 5, 64)       0
flatten (Flatten)            (None, 1600)             0
dense (Dense)                (None, 20)               32020
dense_1 (Dense)              (None, 10)               210
-----
Total params: 69,798
```

```
1 # Compile and train
2 model_cnn.compile(optimizer='adam',
3                   loss='sparse_categorical_crossentropy',
4                   metrics=['accuracy'])
5
```

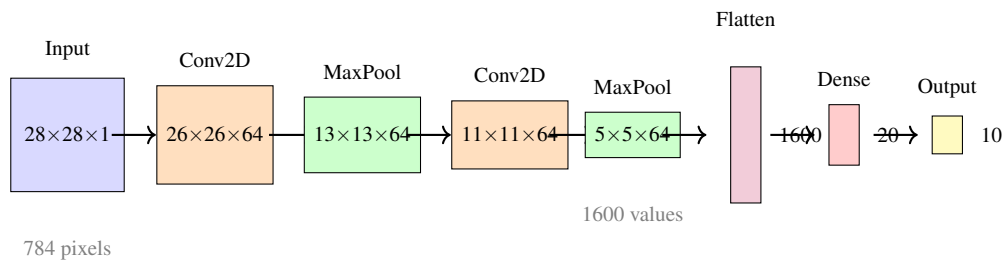


Figure 7.5: CNN architecture for Fashion MNIST: Two convolutional blocks progressively reduce spatial dimensions while increasing the number of learned features, followed by dense layers for classification.

```

6 history = model_cnn.fit(train_images_cnn, train_labels,
7                         validation_data=(val_images_cnn, val_labels),
8                         epochs=20)

```

Expected Output:

```

Epoch 1/20
1875/1875 - 45s - loss: 0.4123 - accuracy: 0.8512 - val_accuracy: 0.8834
...
Epoch 20/20
1875/1875 - 42s - loss: 0.0834 - accuracy: 0.9712 - val_accuracy: 0.9156

```

CNN Results Comparison

The CNN achieves significantly better results:

- Training accuracy: ~97% (vs 89% DNN)
- Validation accuracy: ~91% (vs 87% DNN)

A 4-5% improvement in accuracy is substantial in classification tasks!

Overfitting Alert

Notice the gap between training (97%) and validation (91%) accuracy. This indicates **overfitting**—the model is memorizing training data rather than learning generalizable features. We'll address this with data augmentation later in the chapter.

💡 Why 64 Filters?

We chose 64 filters for each convolutional layer, but why this number? Consider:

1. What does each filter learn to detect? (Hint: early filters often learn edge detectors)
2. If you reduce to 16 filters, what might you lose? What do you gain?
3. For Fashion MNIST with only 10 classes, do we really need 64 different feature detectors?
4. How would you determine the optimal number of filters for your specific task?

Experiment: Try 16, 32, 64, and 128 filters. Plot accuracy vs. training time.

7.8 Visualizing What CNNs Learn

To build intuition, let's visualize what the convolutional filters actually detect.

7.8.1 Extracting Intermediate Activations

```
1 import matplotlib.pyplot as plt
2 from tensorflow.keras import models
3
4 # Create a model that outputs all layer activations
5 layer_outputs = [layer.output for layer in model_cnn.layers]
6 activation_model = tf.keras.models.Model(
7     inputs=model_cnn.input,
8     outputs=layer_outputs
9 )
```

Code Walkthrough

This creates a new model with the same weights but multiple outputs—one for each layer. We can pass an image through and capture what each layer “sees.”

```
1 # Get activations for a sample image
2 sample_image = val_images_cnn[0:1] # First validation image
3 activations = activation_model.predict(sample_image)
4
5 # Visualize first convolutional layer outputs
6 first_conv_activation = activations[0]
7 print(f"First conv output shape: {first_conv_activation.shape}")
8 # Output: (1, 26, 26, 64) - 64 feature maps of 26x26
9
10 # Display 8 feature maps from the first convolutional layer
11 fig, axes = plt.subplots(2, 4, figsize=(12, 6))
12 for i, ax in enumerate(axes.flat):
13     ax.imshow(first_conv_activation[0, :, :, i], cmap='viridis')
14     ax.set_title(f'Filter {i}')
15     ax.axis('off')
16 plt.suptitle('First Convolutional Layer Activations')
17 plt.tight_layout()
18 plt.show()
```

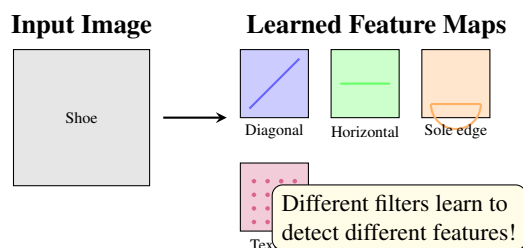


Figure 7.6: Visualization concept: Different convolutional filters learn to detect different patterns. For shoes, some filters detect edges, others detect textures or specific shapes like soles.

Hierarchical Feature Learning

CNNs learn features hierarchically:

- **Early layers:** Detect simple patterns (edges, lines, gradients)
- **Middle layers:** Combine edges into shapes (circles, corners, textures)

• **Deep layers:** Recognize complex patterns (shoe soles, shirt collars, etc.)
 This hierarchy emerges automatically from training—we never explicitly program edge detectors!

7.9 Working with Real-World Images

Fashion MNIST images are small (28×28), centered, and grayscale. Real-world images are larger, vary in position, and have color. Let's work with a more realistic dataset.

7.9.1 The Horse vs Human Dataset

```

1 import os
2 import zipfile
3 import urllib.request
4
5 # Download the dataset
6 url_train = "https://storage.googleapis.com/learning-datasets/" \
7             "horse-or-human.zip"
8 url_val = "https://storage.googleapis.com/learning-datasets/" \
9           "validation-horse-or-human.zip"
10
11 urllib.request.urlretrieve(url_train, "horse-or-human.zip")
12 urllib.request.urlretrieve(url_val, "validation-horse-or-human.zip")
13
14 # Extract
15 with zipfile.ZipFile("horse-or-human.zip", 'r') as zip_ref:
16     zip_ref.extractall("data/horse-or-human")
17 with zipfile.ZipFile("validation-horse-or-human.zip", 'r') as zip_ref:
18     zip_ref.extractall("data/validation-horse-or-human")

```

7.9.2 Using ImageDataGenerator

For larger datasets with images stored on disk, we use ImageDataGenerator to efficiently load and preprocess images in batches:

```

1 from tensorflow.keras.preprocessing.image import ImageDataGenerator
2
3 # Create data generators with rescaling
4 train_datagen = ImageDataGenerator(rescale=1./255)
5 val_datagen = ImageDataGenerator(rescale=1./255)

```

Code Walkthrough

ImageDataGenerator creates an iterator that:

- Loads images from directories in batches (memory efficient)
- Automatically resizes images to a target size
- Applies preprocessing (here, rescaling to [0,1])
- Infers labels from subdirectory names

```

1 # Create training generator
2 train_generator = train_datagen.flow_from_directory(
3     'data/horse-or-human/',
4     target_size=(100, 100), # Resize all images
5     batch_size=32,
6     class_mode='binary'     # Binary classification
7 )
8
9 # Create validation generator

```

```

10 val_generator = val_datagen.flow_from_directory(
11     'data/validation-horse-or-human/',
12     target_size=(100, 100),
13     batch_size=32,
14     class_mode='binary'
15 )

```

Expected Output:

```

Found 1027 images belonging to 2 classes.
Found 256 images belonging to 2 classes.

```

7.9.3 Building a Deeper CNN

For more complex images, we need a deeper network:

```

1 model = tf.keras.models.Sequential([
2     # Block 1
3     tf.keras.layers.Conv2D(32, (3,3), activation='relu',
4                             input_shape=(100, 100, 3)),
5     tf.keras.layers.MaxPooling2D(2, 2),
6
7     # Block 2
8     tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
9     tf.keras.layers.MaxPooling2D(2, 2),
10
11    # Block 3
12    tf.keras.layers.Conv2D(128, (3,3), activation='relu'),
13    tf.keras.layers.MaxPooling2D(2, 2),
14
15    # Block 4
16    tf.keras.layers.Conv2D(256, (3,3), activation='relu'),
17    tf.keras.layers.MaxPooling2D(2, 2),
18
19    # Classification head
20    tf.keras.layers.Flatten(),
21    tf.keras.layers.Dense(512, activation='relu'),
22    tf.keras.layers.Dense(1, activation='sigmoid')
23 ])

```

Code Walkthrough

Four convolutional blocks: Each with increasing filters (32→64→128→256).

Increasing depth: More filters allow detecting more complex patterns.

Sigmoid output: For binary classification (horse=0, human=1).

Input shape (100, 100, 3): 100×100 RGB images.

```

1 # Compile with binary_crossentropy for 2-class problem
2 from tensorflow.keras.optimizers import RMSprop
3
4 model.compile(
5     loss='binary_crossentropy',
6     optimizer=RMSprop(learning_rate=0.0001),
7     metrics=['accuracy']
8 )
9
10 # View architecture
11 model.summary()

```

Expected Output (abbreviated):

```
Model: "sequential"
-----
Layer (type)                Output Shape              Param #
-----
conv2d (Conv2D)              (None, 98, 98, 32)        896
max_pooling2d (MaxPooling2D) (None, 49, 49, 32)        0
conv2d_1 (Conv2D)            (None, 47, 47, 64)       18496
...
dense_1 (Dense)              (None, 1)                 513
-----
Total params: 4,499,713
```

Model Size for Edge Devices

This model has **4.5 million parameters**, requiring approximately **17 MB** in Float32 format. This is too large for most microcontrollers but suitable for mobile devices or edge computers like Raspberry Pi. Later chapters cover quantization to reduce this size.

```
1 # Train the model
2 history = model.fit(
3     train_generator,
4     epochs=15,
5     validation_data=val_generator,
6     verbose=1
7 )
```

7.10 Avoiding Overfitting with Data Augmentation

With limited training data, models often overfit—learning specific training examples rather than general patterns. Data augmentation creates artificial variations of training images.

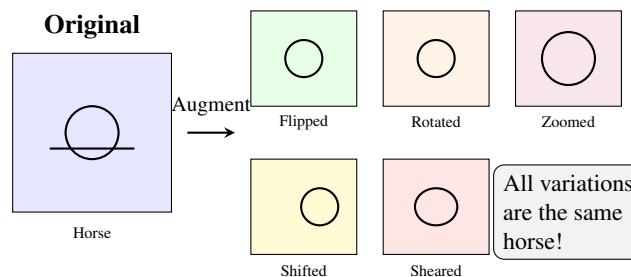


Figure 7.7: Data augmentation creates training variations through geometric transformations. The model learns that a horse remains a horse regardless of position, rotation, or scale—improving generalization.

```
1 # ImageDataGenerator with augmentation
2 train_datagen_aug = ImageDataGenerator(
3     rescale=1./255,
4     rotation_range=40,      # Random rotation up to 40 degrees
5     width_shift_range=0.2,  # Random horizontal shift
6     height_shift_range=0.2, # Random vertical shift
7     shear_range=0.2,       # Random shearing
8     zoom_range=0.2,        # Random zoom
9     horizontal_flip=True,   # Random horizontal flip
```



```

10     fill_mode='nearest'      # Fill empty pixels
11 )

```

Code Walkthrough

rotation_range=40: Randomly rotate images up to ± 40 degrees.

width/height_shift: Shift image position by up to 20% of dimensions.

shear_range: Apply shearing transformation.

zoom_range: Randomly zoom in/out.

horizontal_flip: Randomly mirror images horizontally.

fill_mode='nearest': How to fill new pixels created by transformations.

Augmentation Only for Training

Apply augmentation **only to training data**, never to validation or test data. Validation should test on realistic, unmodified images to accurately measure generalization.

```

1 # Create augmented training generator
2 train_generator_aug = train_datagen.flow_from_directory(
3     'data/horse-or-human/',
4     target_size=(100, 100),
5     batch_size=32,
6     class_mode='binary'
7 )
8
9 # Validation generator WITHOUT augmentation
10 val_generator = val_datagen.flow_from_directory(
11     'data/validation-horse-or-human/',
12     target_size=(100, 100),
13     batch_size=32,
14     class_mode='binary'
15 )

```

```

1 # Train with augmented data
2 history_aug = model.fit(
3     train_generator_aug,
4     epochs=100,
5     validation_data=val_generator,
6     verbose=1
7 )

```

Augmentation Benefits

With data augmentation:

- Training takes longer (more variation to learn)
- Training accuracy may be lower (augmented images are harder)
- But validation accuracy improves significantly
- Model generalizes better to new, unseen images

7.11 Edge Deployment Considerations

When designing CNNs for edge deployment, architecture choices directly impact memory, latency, and power consumption.

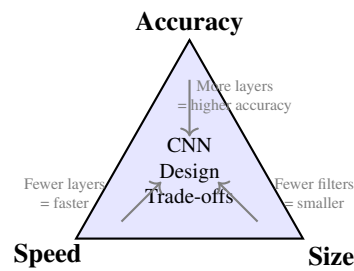


Figure 7.8: The fundamental trade-off triangle in edge ML: You can optimize for any two, but the third will suffer. Design decisions must balance accuracy, inference speed, and model size.

Table 7.1: CNN architecture considerations for edge deployment

Factor	Impact	Recommendation
Filter count	Memory, compute	Start small (16, 32), increase only if needed
Filter size	Compute	3×3 is standard; 1×1 for reduction
Depth (layers)	Accuracy, latency	More depth = better accuracy, slower inference
Input size	Memory, compute	Smaller inputs ($< 128 \times 128$) for MCUs
Pooling	Size reduction	Use aggressively to reduce dimensions early
Dense layers	Parameters	Minimize; most parameters are here

7.11.1 Architecture Design Guidelines

Parameter Budget for Microcontrollers

For ARM Cortex-M4 class devices with 256 KB RAM:

- Total model size: **<100 KB** (quantized)
- Tensor arena: **<100 KB** for activations
- This limits you to $\sim 25,000$ parameters in Float32, or $\sim 100,000$ quantized

7.11.2 Memory Calculation for Edge Deployment

Edge devices have strict memory constraints. Let's calculate the memory footprint of a CNN.

Memory Components

A CNN's total memory usage consists of:

1. **Model weights:** Parameters stored in Flash or RAM
2. **Input tensor:** Current input image
3. **Output tensor:** Current output predictions
4. **Tensor arena:** Intermediate activations (feature maps)
5. **Framework overhead:** TensorFlow Lite interpreter and operators

■ **Example — Memory Calculation: Tiny CNN for Arduino.** Consider this architecture for 48×48 grayscale gesture recognition:

```

1 model = Sequential([
2     Conv2D(8, (3,3), activation='relu', input_shape=(48,48,1)),
3     MaxPooling2D(2, 2), # -> 23x23x8
4     Conv2D(16, (3,3), activation='relu'),
5     MaxPooling2D(2, 2), # -> 10x10x16
6     Flatten(),          # -> 1600

```

```

7 Dense(32, activation='relu'),
8 Dense(4, activation='softmax') # 4 gesture classes
9 ])

```

Step 1: Count parameters

- Conv2D(8): $(3 \times 3 \times 1 + 1) \times 8 = 80$ params
- Conv2D(16): $(3 \times 3 \times 8 + 1) \times 16 = 1,168$ params
- Dense(32): $(1600 + 1) \times 32 = 51,232$ params
- Dense(4): $(32 + 1) \times 4 = 132$ params
- **Total: 52,612 parameters**

Step 2: Calculate model size

- Float32: $52,612 \times 4 = 210,448$ bytes ≈ 205 KB
- Int8 quantized: $52,612 \times 1 = 52,612$ bytes ≈ 51 KB

Step 3: Calculate tensor arena (worst-case) The arena must hold the largest intermediate tensor:

- Input: $48 \times 48 \times 1 = 2,304$ bytes (int8)
- After Conv1: $46 \times 46 \times 8 = 16,928$ bytes (largest!)
- After Pool1: $23 \times 23 \times 8 = 4,232$ bytes
- After Conv2: $21 \times 21 \times 16 = 7,056$ bytes
- After Pool2: $10 \times 10 \times 16 = 1,600$ bytes
- Flattened: 1,600 bytes

TFLite needs space for input + output of each layer. Conservative estimate: ~ 30 KB arena.

Step 4: Total RAM budget for Arduino Nano 33 BLE (256 KB SRAM)

- Model weights (in Flash): 51 KB (doesn't use RAM)
- Tensor arena: ~ 30 KB
- TFLite interpreter: ~ 20 KB
- Application code + stack: ~ 30 KB
- **Total RAM: ~ 80 KB — Comfortable fit!**

■

Dense Layers Dominate Memory

Notice that Dense(32) alone has 51,232 parameters—**97% of total parameters!** This is why modern architectures like MobileNet minimize or eliminate dense layers, using global average pooling instead.

Replacing Dense(32) with global average pooling + Dense(4):

- GlobalAveragePooling2D: 0 params, reduces $10 \times 10 \times 16 = 1,600$ to just 16 values
- Dense(4): $(16 + 1) \times 4 = 68$ params
- **New total: 1,448 parameters (36x reduction!)**

7.11.3 Efficient Architecture Patterns

```

1 # Efficient CNN for microcontrollers (~30KB quantized)
2 efficient_model = tf.keras.models.Sequential([
3     # Small input, aggressive pooling
4     tf.keras.layers.Conv2D(8, (3,3), activation='relu',
5                             input_shape=(48, 48, 1)),
6     tf.keras.layers.MaxPooling2D(2, 2),
7
8     tf.keras.layers.Conv2D(16, (3,3), activation='relu'),
9     tf.keras.layers.MaxPooling2D(2, 2),
10
11     # Minimal dense layers

```

```

12 tf.keras.layers.Flatten(),
13 tf.keras.layers.Dense(32, activation='relu'),
14 tf.keras.layers.Dense(10, activation='softmax')
15 ])
16
17 efficient_model.summary()

```

Expected Output:

```

Total params: 7,818
Trainable params: 7,818
Non-trainable params: 0

```

Code Walkthrough

This efficient architecture achieves:

- Only 7,818 parameters (vs 69,798 for our earlier Fashion MNIST CNN)
- ~8 KB in Float32, ~2 KB quantized
- Still achieves >85% accuracy on simpler tasks
- Fits easily on Cortex-M4 class microcontrollers

7.11.4 Depthwise Separable Convolutions

Depthwise separable convolutions are the foundation of efficient mobile architectures like MobileNet. They achieve dramatic parameter reduction with minimal accuracy loss.

Standard Convolution Computational Cost

For a standard convolution layer:

- Input: $D_F \times D_F \times C_{in}$ (spatial size D_F , C_{in} channels)
- Kernel: $K \times K$ with C_{out} filters
- Output: $D_F \times D_F \times C_{out}$ (assuming same padding, stride 1)

Computational cost (multiply-adds):

$$\text{Cost}_{\text{standard}} = D_F \times D_F \times C_{in} \times C_{out} \times K \times K$$

Parameter count:

$$\text{Params}_{\text{standard}} = K \times K \times C_{in} \times C_{out}$$

For example, $32 \times 32 \times 64$ input with 128 filters of size 3×3 :

- Computation: $32 \times 32 \times 64 \times 128 \times 3 \times 3 = 75,497,472$ ops
- Parameters: $3 \times 3 \times 64 \times 128 = 73,728$ params

Definition — Depthwise Separable Convolution. Depthwise separable convolution factors standard convolution into two steps:

Step 1: Depthwise Convolution

- Apply one $K \times K$ filter per input channel independently
- Output: $D_F \times D_F \times C_{in}$ (same number of channels)
- Each filter only processes one input channel (no mixing)

Computational cost:

$$\text{Cost}_{\text{depthwise}} = D_F \times D_F \times C_{in} \times K \times K$$

Parameter count:

$$\text{Params}_{\text{depthwise}} = K \times K \times C_{in}$$

Step 2: Pointwise Convolution (1×1 conv)

- Apply C_{out} filters of size $1 \times 1 \times C_{in}$
- Combines information across channels (like dense layer per spatial location)
- Output: $D_F \times D_F \times C_{out}$

Computational cost:

$$\text{Cost}_{\text{pointwise}} = D_F \times D_F \times C_{in} \times C_{out}$$

Parameter count:

$$\text{Params}_{\text{pointwise}} = C_{in} \times C_{out}$$

Efficiency Gain

Total cost of depthwise separable convolution:

$$\begin{aligned} \text{Cost}_{\text{total}} &= D_F^2 \times C_{in} \times K^2 + D_F^2 \times C_{in} \times C_{out} \\ &= D_F^2 \times C_{in} \times (K^2 + C_{out}) \end{aligned}$$

Speedup ratio compared to standard convolution:

$$\text{Speedup} = \frac{D_F^2 \times C_{in} \times C_{out} \times K^2}{D_F^2 \times C_{in} \times (K^2 + C_{out})} = \frac{C_{out} \times K^2}{K^2 + C_{out}}$$

For $K = 3$, $C_{out} = 128$:

$$\text{Speedup} = \frac{128 \times 9}{9 + 128} = \frac{1152}{137} \approx 8.4\times$$

Depthwise separable convolutions are approximately **8-9x faster and smaller** than standard convolutions with 3×3 kernels!

■ **Example — MobileNet Efficiency Calculation.** Compare standard vs depthwise separable for $32 \times 32 \times 64$ input, 128 output channels, 3×3 kernel:

Standard Convolution:

- Computation: $32^2 \times 64 \times 128 \times 9 = 75,497,472$ ops
- Parameters: $9 \times 64 \times 128 = 73,728$ params

Depthwise Separable:

- Depthwise: $32^2 \times 64 \times 9 = 589,824$ ops, $9 \times 64 = 576$ params
- Pointwise: $32^2 \times 64 \times 128 = 8,388,608$ ops, $64 \times 128 = 8,192$ params
- Total: 8,978,432 ops, 8,768 params

Reduction:

- Computation: $75,497,472 / 8,978,432 = 8.4\times$ fewer operations
- Parameters: $73,728 / 8,768 = 8.4\times$ fewer parameters
- Memory bandwidth: Proportional to parameter count, also $\sim 8\times$ reduction

This is why MobileNets can achieve 90%+ of standard CNN accuracy with a fraction of the compute! ■

When to Use Depthwise Separable Convolutions

Use depthwise separable when:

- Deploying on resource-constrained devices (MCUs, mobile phones)
- Inference speed is critical (real-time requirements)
- Power consumption matters (battery-powered devices)
- Model size must fit in limited Flash/RAM

Use standard convolutions when:

- Training from scratch with limited data (depthwise can underfit)
- First layer of network (input has only 1-3 channels, no benefit)
- Maximum accuracy is required and resources are available

Hybrid approach: Many modern architectures use standard conv in early layers (spatial feature extraction) and depthwise separable in deeper layers (efficiency).

```

1 # Implementing depthwise separable convolution in Keras
2 # (MobileNet-style block)
3 x = tf.keras.layers.DepthwiseConv2D(
4     kernel_size=(3, 3),
5     padding='same',
6     activation='relu'
7 )(input_tensor)
8
9 x = tf.keras.layers.Conv2D(
10     filters=128,
11     kernel_size=(1, 1), # Pointwise convolution
12     padding='same',
13     activation='relu'
14 )(x)
15
16 # Compare parameter count:
17 # Standard: 3x3x64x128 = 73,728 params
18 # Depthwise separable: 3x3x64 + 1x1x64x128 = 576 + 8,192 = 8,768 params

```

7.11.5 Memory Bandwidth vs Computation Trade-offs

On edge devices, **memory bandwidth** is often the bottleneck, not computation.

Memory Bandwidth Dominance

Modern edge processors (even ARM Cortex-M7) can perform multiply-accumulate (MAC) operations faster than they can fetch data from memory.

Example: STM32H7 (400 MHz Cortex-M7)

- MAC throughput: 400 million MACs/second
- SRAM bandwidth: ~ 200 MB/s
- For INT8 data: 200 million bytes/second

A standard 3×3 convolution requires reading:

- Input patch: $3 \times 3 = 9$ values
- Kernel weights: $3 \times 3 = 9$ values
- Total: 18 bytes per output for INT8

At 400M MACs/sec with 9 MACs per output: ~ 44 M outputs/sec expected. But memory provides only $200\text{M bytes/sec} \div 18 \text{ bytes/output} = \sim 11\text{M outputs/sec}$.

Conclusion: Memory bandwidth limits throughput to 25% of computational capacity!

Why Depthwise Separable Helps Memory Bandwidth

Depthwise separable convolutions reduce memory traffic:

Standard convolution per output:

- Read: $K^2 \times C_{in}$ input values + $K^2 \times C_{in}$ weights
- For 3×3 , $C_{in} = 64$: $9 \times 64 + 9 \times 64 = 1,152$ bytes (INT8)

Depthwise separable per output:

- Depthwise: K^2 input + K^2 weights = 18 bytes
- Pointwise: C_{in} input + C_{in} weights = 128 bytes
- Total: 146 bytes (7.9× less!)

This bandwidth reduction is why MobileNets achieve near-linear speedup on edge devices.

7.11.6 Method Selection Guide: CNN Architecture for Edge

Choosing the right CNN architecture is critical for edge deployment success. Different architectures make different trade-offs between accuracy, speed, and model size.

Choosing a CNN Architecture for Edge Vision

Use the following decision tree to select the best architecture:

- 1. What is your target hardware?**
 - Arduino/STM32 (<256KB RAM) → **Custom Tiny CNN** (8-16 filters, 48×48 input)
 - ESP32 (520KB RAM) → **Custom Small CNN** or **MobileNetV1-0.25** (96×96 input)
 - Raspberry Pi (1GB+ RAM) → **MobileNetV2**, **EfficientNet-Lite**, or **SqueezeNet**
- 2. What is your accuracy requirement?**
 - Simple binary classification (>90% ok) → **Custom Tiny CNN**
 - Multi-class (need >85%) → **MobileNetV1** or **custom optimized**
 - Complex scenes (need >90%) → **MobileNetV2** or **EfficientNet-Lite**
- 3. What is your latency budget?**
 - <10ms inference → **Tiny custom CNN** on MCU
 - 10-50ms acceptable → **MobileNetV1** on RPi
 - <100ms acceptable → **MobileNetV2** or **EfficientNet** on RPi
- 4. Do you need transfer learning?**
 - Yes, need ImageNet features → **MobileNetV2** or **EfficientNet** (pre-trained available)
 - No, training from scratch → **Custom architecture** (optimized for your task)

Table 7.2: CNN Architecture Comparison for Edge Deployment

Architecture	Size	Accuracy	Speed	Best For
Ultra-Low Power (MCU)				
Custom Tiny (8-16 filters)	2-10KB (int8)	75-85% (binary/3-class)	<10ms (Cortex-M4)	Arduino, simple tasks Binary classification
Low Power (ESP32 Class)				
Custom Small (16-32 filters)	10-50KB (int8)	80-88% (10 classes)	20-50ms (240 MHz)	ESP32, basic vision Simple multi-class
MobileNet-0.25 (Alpha 0.25)	50-100KB (int8)	85-90% (ImageNet)	30-80ms (dual-core)	ESP32 with transfer learning from ImageNet
Edge Compute (Raspberry Pi)				
MobileNetV1	4-16MB (float32)	89-91% (ImageNet)	50-100ms (no GPU)	RPi 3/4, real-time Good accuracy-speed
MobileNetV2	9-14MB (float32)	91-92% (ImageNet)	80-150ms (no GPU)	RPi 4, best mobile Transfer learning
EfficientNet-Lite0	12-20MB (float32)	92-93% (ImageNet)	100-200ms (no GPU)	RPi 4, high accuracy Accuracy-first apps
SqueezeNet	3-5MB (float32)	86-88% (ImageNet)	40-80ms (no GPU)	RPi with size limit AlexNet-level accuracy
For Comparison (Not Suitable for Basic Edge)				
ResNet-50	98MB (float32)	94-95% (ImageNet)	500-1000ms (RPi 4)	Cloud/GPU only Too slow for edge
VGG-16	528MB (float32)	90-92% (ImageNet)	>2000ms (RPi 4)	Cloud/GPU only Extremely large

MobileNet Design Principles

MobileNet architectures use **depthwise separable convolutions** to dramatically reduce parameters:

- **Standard conv:** $(H \times W \times C_{in}) * (K \times K \times C_{out})$ operations
- **Depthwise separable:** $(H \times W \times C_{in} \times K^2) + (H \times W \times C_{in} \times C_{out})$
- **Reduction:** 8-9x fewer parameters with minimal accuracy loss
- **Width multiplier α :** Scale channels (0.25, 0.5, 0.75, 1.0)
- **Resolution multiplier:** Scale input size (96, 128, 160, 192, 224)

MobileNetV2 adds **inverted residuals** and **linear bottlenecks** for even better efficiency.

💡 Calculating Model Feasibility

Can MobileNetV2-0.5 (224×224 input) run on ESP32 (520KB RAM)?

Memory requirements:

- Input image: $224 \times 224 \times 3 = 150\text{KB}$ (float32)
- Model weights: $\sim 1.5\text{MB}$ (need to store in Flash, not RAM)
- Tensor arena (activations): $\sim 300\text{KB}$ for intermediate layers
- Total RAM: $150\text{KB} + 300\text{KB} = 450\text{KB}$

Verdict: Just barely fits! But better approach:

- Reduce input to 96×96 : $96^2 \times 3 \times 4 = 110\text{KB}$
- Use int8 quantization: $96^2 \times 3 = 27\text{KB}$ input

- Reduce arena: $\sim 150\text{KB}$ with quantization
- Total: $27\text{KB} + 150\text{KB} = 177\text{KB} \rightarrow$ Comfortable fit!

7.12 Three-Tier Implementation Guide

Level 1: Notebook Development

Develop and train CNN architectures in Jupyter notebooks or Google Colab:

- Experiment with layer configurations and hyperparameters
- Use `model.summary()` to track parameter counts
- Visualize training with TensorBoard or matplotlib
- Save trained models with `model.save('model.h5')`

Level 2: TFLite Conversion and Testing

Convert trained models to TensorFlow Lite format:

```

1 # Convert to TFLite
2 converter = tf.lite.TFLiteConverter.from_keras_model(model)
3 converter.optimizations = [tf.lite.Optimize.DEFAULT]
4 tflite_model = converter.convert()
5
6 # Save and check size
7 with open('model.tflite', 'wb') as f:
8     f.write(tflite_model)
9 print(f"Model size: {len(tflite_model) / 1024:.1f} KB")
10
11 # Test with TFLite interpreter
12 interpreter = tf.lite.Interpreter(model_content=tflite_model)
13 interpreter.allocate_tensors()
```

Level 3: Microcontroller Deployment

For device deployment (covered in detail in later chapters):

1. Convert TFLite model to C array using `xxd`
2. Include in Arduino/PlatformIO project
3. Implement inference loop with camera input
4. Optimize tensor arena size for your specific device

7.13 Troubleshooting Guide

⚠ CNN Training Loss Not Decreasing

Symptoms: Loss stays constant around random guess level (e.g., 2.3 for 10-class) or barely decreases

Cause: Learning rate too low, wrong optimizer, data not shuffled, or labels incorrect

Solution:

1. Check data normalization: Images should be scaled to $[0, 1]$ range

2. Verify labels match classes: Print first few labels and corresponding images
3. Increase learning rate: Try 0.001 for Adam optimizer
4. Shuffle training data: Set `shuffle=True` in `model.fit()`
5. Check model architecture: Ensure final layer has correct number of neurons (`num_classes`)
6. Use appropriate loss: `categorical_crossentropy` for one-hot, `sparse_categorical_crossentropy` for integer labels

⚠ Image Augmentation Causes Accuracy Drop

Symptoms: Adding `ImageDataGenerator` with augmentation makes validation accuracy worse

Cause: Excessive augmentation, validation data also augmented, or augmentation parameters too aggressive

Solution:

1. Apply augmentation ONLY to training data, not validation: `train_gen.flow(X_train, y_train) model.fit(..., validation_data=(X_val, y_val))`
2. Reduce augmentation strength: Use smaller rotation (15° instead of 45°), zoom (0.1 instead of 0.3)
3. Check transformed images: Plot augmented samples to verify they're still recognizable
4. Avoid flipping for asymmetric classes: Don't horizontally flip if left/right matters (e.g., letters b and d)
5. Train longer: Augmentation requires more epochs to converge

⚠ Out of Memory During Training

Symptoms: `ResourceExhaustedError`: OOM when allocating tensor during CNN training

Cause: Batch size too large, input images too big, or model too deep for available GPU memory

Solution:

1. Reduce batch size: Try 32 instead of 128, or 16 for very large models
2. Resize input images: Use 128x128 or 64x64 instead of 224x224
3. Reduce model depth: Remove layers or reduce filters per layer
4. Use mixed precision training: `tf.keras.mixed_precision.set_global_policy('mixed_float16')`
5. Enable memory growth: `tf.config.experimental.set_memory_growth(gpu, True)`
6. Clear previous sessions: Add `tf.keras.backend.clear_session()` before building model

⚠ Model Overfits Despite Regularization

Symptoms: Training accuracy 99% but validation accuracy stuck at 75%, even with dropout

Cause: Dataset too small, insufficient augmentation, or model too complex

Solution:

1. Add more dropout: Increase rate to 0.5 after fully connected layers
2. Use BatchNormalization: Add after Conv2D layers
3. Implement early stopping: `EarlyStopping(patience=10,`

```
restore_best_weights=True)
```

4. Collect more training data: Aim for at least 1000 images per class
5. Use stronger augmentation: Combine rotation, zoom, shift, brightness
6. Try transfer learning: Use pre-trained MobileNet instead of training from scratch

⚠ Low Accuracy on Test Images

Symptoms: Model works well on training data but performs poorly on new real-world images

Cause: Train/test distribution mismatch, preprocessing inconsistency, or model memorizing backgrounds

Solution:

1. Apply same preprocessing: Ensure test images normalized identically to training
2. Check image dimensions: Verify test images resized to model's expected input size
3. Inspect misclassified samples: Visualize to identify patterns (lighting, angle, background)
4. Add diverse training data: Include various backgrounds, lighting, orientations
5. Use test-time augmentation: Average predictions over multiple augmented versions
6. Retrain with representative samples: Add hard examples from test set to training (carefully to avoid data leakage)

7.14 Summary

In this chapter, you learned:

- **Why CNNs work:** Local connectivity, parameter sharing, and translation invariance make CNNs ideal for image data
- **Convolution operations:** Learned filters slide across images, extracting features at each position
- **Pooling layers:** Reduce spatial dimensions while preserving important features
- **CNN architecture:** Stack convolutional blocks with increasing filter counts, followed by dense classification layers
- **ImageDataGenerator:** Efficiently handles large image datasets from disk
- **Data augmentation:** Creates training variations to prevent overfitting and improve generalization
- **Edge constraints:** Model size, inference speed, and accuracy trade-offs guide architecture decisions

Building On: LAB16 - Object Detection with YOLO

This chapter covers image classification (identifying what's in an image). In LAB16, you'll advance to object detection with YOLO:

- Detect multiple objects in a single image with bounding boxes
- Learn how YOLO adapts CNN architectures for real-time detection
- Deploy lightweight YOLO models (YOLO-tiny, YOLO-Nano) to edge devices
- Apply object detection to practical computer vision tasks

The CNN fundamentals you've mastered here are essential for understanding YOLO's architecture.

Self-Assessment Questions

Test your understanding before continuing:

1. Explain why a 3×3 convolution with 64 filters on a 28×28 image produces a $26 \times 26 \times 64$ output. What happens with padding='same'?
2. A convolutional layer with 32 filters of size $3 \times 3 \times 64$ has how many parameters? Include bias terms in your calculation.
3. Why do CNNs use far fewer parameters than fully-connected layers for image classification? Explain parameter sharing and local connectivity.
4. Identify overfitting from training vs validation accuracy curves. What data augmentation strategies would you apply to reduce it?
5. A mobilenet-style model has 200KB of parameters. After int8 quantization, will it fit on an Arduino Nano 33 BLE (256KB SRAM)? What else needs memory?

7.15 Guided Experiments

Experiment 1: Filter Count Comparison

Hypothesis: Increasing the number of convolutional filters improves accuracy but increases model size and inference time. There exists an optimal filter count for edge deployment.

Procedure:

1. Use the Fashion MNIST CNN architecture from this chapter
2. Train models with varying filter counts in first layer: 8, 16, 32, 64, 128
3. Keep all other parameters constant (same architecture depth, epochs=10)
4. Measure model size (parameters), training time, and test accuracy
5. Record inference time for each model on CPU

Data Collection:

Filter Count	Param Count	Test Acc (%)	Train Time (s)	Inference (ms)
8				
16				
32				
64				
128				

Analysis Questions:

- At what point does increasing filters stop improving accuracy significantly?
- What is the relationship between filter count and parameter count?
- For deployment on Arduino (256KB RAM), which filter count is most practical?
- What is the accuracy vs inference time trade-off?

Experiment 2: Data Augmentation Impact

Hypothesis: Data augmentation reduces overfitting and improves model generalization, especially with limited training data. Different augmentation strategies have different effectiveness.

Procedure:

1. Train the horse vs human classifier with different augmentation strategies:
 - Baseline: No augmentation

- Strategy A: Horizontal flip only
 - Strategy B: Rotation + zoom
 - Strategy C: Full augmentation (flip + rotation + zoom + shift)
2. Train each for 15 epochs with same model architecture
 3. Record training and validation accuracy at each epoch
 4. Compare final test accuracy and overfitting gap

Data Collection:

Augmentation	Train Acc (%)	Val Acc (%)	Overfitting Gap
None (baseline)			
Horizontal Flip			
Rotation + Zoom			
Full Augmentation			

Analysis Questions:

- Which augmentation strategy achieves the best validation accuracy?
- Does augmentation always reduce the training/validation gap?
- Are there diminishing returns from adding more augmentation types?
- For edge deployment, does augmentation during training affect inference speed?

Case Study: Wildlife Camera Trap for Conservation Monitoring

Context: A wildlife conservation organization deployed 200 camera traps across a remote forest preserve to monitor endangered species populations. Traditional camera traps stored all images to SD cards, requiring researchers to manually hike to each location monthly to retrieve cards and manually review 500,000+ images (99% containing no animals). They needed on-device CNN-based animal detection to transmit only relevant images via satellite uplink, reducing bandwidth costs and enabling real-time poaching alerts.

Challenge: The team trained a MobileNetV2 model achieving 94% accuracy on curated datasets, but it failed catastrophically in deployment. False positives from swaying branches, rain droplets, and shadows triggered constant satellite transmissions (\$2.50 per 10KB image). False negatives missed 40% of actual animal detections due to nighttime IR images, motion blur, and partial occlusions. The ESP32-CAM's 4MB Flash and 520KB RAM couldn't fit the 8.2MB model. Battery life was 6 days instead of the target 6 months.

Solution: The team redesigned the system using edge-optimized CV techniques:

- Retrained model on domain-specific dataset: 50,000 actual camera trap images including nighttime IR, rain, fog, and occlusions
- Applied aggressive data augmentation: rotation ($\pm 20^\circ$), brightness (0.5-1.5x), blur, and synthetic shadow overlays
- Replaced MobileNetV2 with EfficientNet-Lite0 and applied int8 quantization (8.2MB $\rightarrow$ 1.8MB)
- Implemented two-stage detection: motion-based pre-filter (passive IR sensor) + CNN inference only on motion events
- Optimized input resolution: 224x224 $\rightarrow$ 96x96 (4x fewer pixels, 3x faster inference)
- Added temporal filtering: require 3+ detections within 10 seconds to confirm animal presence
- Implemented progressive JPEG compression: 95% quality for confirmed animals, 70% for uncertain detections

Results:

- Model size: 8.2MB → 1.8MB (4.6x reduction, fits in ESP32 Flash)
- False positive rate: 850/day → 2/day (425x improvement)
- False negative rate: 40% → 8% (5x improvement)
- Inference time: 1.8s → 280ms per image (6.4x speedup at 96x96 resolution)
- Satellite bandwidth: \$4,200/month → \$85/month (98% cost reduction)
- Battery life: 6 days → 5.5 months (27x improvement via motion pre-filtering)

Lessons Learned:

- Lab accuracy is meaningless—train and validate on data from the actual deployment environment (lighting, weather, camera angles)
- Data augmentation is essential for robustness: real-world conditions (rain, fog, night, motion blur) must be represented in training
- Two-stage architectures save power: cheap motion detection (passive IR) prevents unnecessary CNN inference on static scenes
- Resolution matters more than depth: 96x96 EfficientNet often outperforms 224x224 MobileNet on edge devices with limited compute
- Temporal filtering reduces false positives: single-frame CNN confidence is unreliable, require consistent detections over time
- Edge cases dominate real deployments: partial occlusions, extreme angles, and low contrast account for 80% of false negatives
- Always include “negative” examples in training: images of common false positive triggers (branches, rain, insects) are critical

7.16 Exercises

[Intermediate] Intermediate: Exercise 1: Architecture Exploration

Modify the Fashion MNIST CNN:

1. Change the number of filters from 64 to 16. How does accuracy change? How does training speed change?
2. Remove the second convolutional block entirely. What happens?
3. Add a third convolutional block. Does accuracy improve?

Record training/validation accuracy and training time for each variant.

[Intermediate] Intermediate: Exercise 2: Visualizing Deeper Layers

Extend the visualization code to show activations from the **second** convolutional layer. Compare these features to the first layer:

- Are they more abstract?
- Can you identify what patterns they respond to?
- How does the spatial resolution differ?

[Intermediate] Intermediate: Exercise 3: Augmentation Impact

Train the horse/human classifier with different augmentation strategies:

1. No augmentation (baseline)

2. Only horizontal flipping
 3. Full augmentation as shown
 4. Extreme augmentation (rotation_range=90, zoom_range=0.5)
- Plot validation accuracy curves. Which strategy works best?

[Challenge] Challenge: Exercise 4: Edge-Optimized Architecture

Design a CNN that:

- Has fewer than 10,000 parameters
- Achieves at least 85% validation accuracy on Fashion MNIST
- Uses input images of 28×28 (no resizing needed)

Document your architecture choices and explain the trade-offs you made.

Related Labs

Prerequisites:

- **LAB02 - Neural Network Training Fundamentals:** Dense networks, activation functions, loss, and gradient descent form the foundation for understanding CNNs.

Next Steps:

- **LAB03 - Model Optimization and Quantization:** Compress your CNN models using quantization to fit on edge devices.
- **LAB16 - Advanced Computer Vision (YOLO):** Build object detection systems using YOLO for embedded vision applications.
- **LAB05 - Deployment:** Deploy your trained CNNs to Arduino or ESP32 with camera modules.

— Further Reading. Foundational Papers:

- *Gradient-based learning applied to document recognition* by LeCun, Bottou, Bengio, & Haffner (1998). Proceedings of the IEEE, 86(11):2278-2324. [*The original LeNet-5 paper establishing CNNs for computer vision*]
- *ImageNet Classification with Deep Convolutional Neural Networks (AlexNet)* by Krizhevsky, Sutskever, & Hinton (2012). NeurIPS. [*Breakthrough that reignited deep learning*] <https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf>
- *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications* by Howard et al. (2017): <https://arxiv.org/abs/1704.04861> [*Introduces depthwise separable convolutions for edge deployment*]
- *MobileNetV2: Inverted Residuals and Linear Bottlenecks* by Sandler et al. (2018): <https://arxiv.org/abs/1801.04381> [*Improved efficiency with inverted residuals*]
- *EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks* by Tan & Le (2019): <https://arxiv.org/abs/1905.11946> [*Compound scaling method for balanced architecture design*]
- *Searching for MobileNetV3* by Howard et al. (2019): <https://arxiv.org/abs/1905.02244> [*Neural architecture search for mobile applications*]
- *Visualizing and Understanding Convolutional Networks* by Zeiler & Fergus (2014). ECCV. [*Deconvolution visualization techniques for interpreting learned features*]

Textbooks and Courses:

- *Deep Learning* by Goodfellow, Bengio, & Courville - Chapter 9 (Convolutional Networks): <https://www.deeplearningbook.org/>
- **Stanford CS231n: Convolutional Neural Networks for Visual Recognition** - Comprehensive course with excellent lecture notes, assignments, and video lectures: <http://cs231n.stanford.edu/> and <http://cs231n.github.io/>
- *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers* by Warden & Situnayake (2019). O'Reilly Media. [Practical guide for deploying CNNs on microcontrollers]

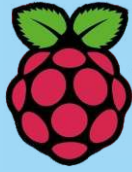
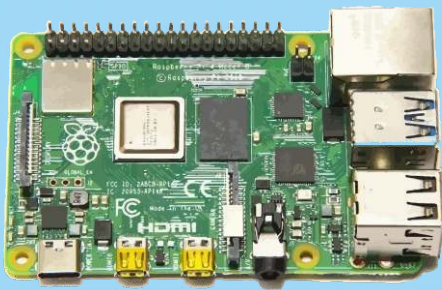
Interactive Visualizations:

- **3Blue1Brown: But what is a convolution?** - Outstanding visual explanation of convolution mathematics: <https://www.youtube.com/watch?v=KuXjwB4LzSA>
- **CNN Explainer** - Interactive browser-based visualization of CNN operations and feature maps: <https://poloclub.github.io/cnn-explainer/>
- **Distill.pub Feature Visualization** - Understanding what CNNs learn: <https://distill.pub/2017/feature-visualization/>
- **TensorFlow Playground** - Interactive neural network visualization (helpful for understanding gradient descent): <https://playground.tensorflow.org/>

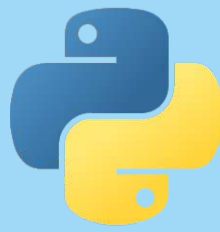
Edge Deployment Resources:

- **TensorFlow Lite for Microcontrollers** - Official documentation and examples: <https://www.tensorflow.org/lite/microcontrollers>
- **TensorFlow Lite Model Zoo** - Pre-trained models optimized for edge deployment: <https://www.tensorflow.org/lite/models>
- **Edge Impulse Computer Vision Tutorials** - Practical guides for embedded vision: <https://docs.edgeimpulse.com/docs/tutorials/image-classification>
- **Andrej Karpathy's Blog** - Deep dives into CNN internals, debugging, and best practices: <https://karpathy.medium.com/>

— **Credits.** This chapter draws on concepts from the TinyML curriculum and TensorFlow documentation. The horse/human dataset is provided by Laurence Moroney. Fashion MNIST is by Zalando Research. We have taken this code segment from the online course: *Edx Tiny Machine Learning (2021-22)*. Since we may have updated the code to reflect any changes made to the underlying frameworks where necessary. Credits should go to the original authors of this tutorial.



RaspberryPi



pythonTM

8. Arduino Sensor Programming for Edge ML

Learning Objectives

After completing this lab, you will be able to:

- Understand analog and digital sensor interfaces and their characteristics
- Set up Arduino IDE and configure boards for development
- Write Arduino sketches to read sensor data and control actuators
- Implement sensor data preprocessing for ML applications
- Design sensor circuits using breadboards and understand voltage dividers
- Collect and format sensor data for machine learning training

Interactive Notebook

Jupyter Notebook: LAB08_arduino_sensors.ipynb

[Open in Google Colab](#) | [View on GitHub](#) | [Companion Website](#)

Prerequisites

Before starting this lab, ensure you:

- Completed LAB05 (Embedded Systems Deployment): Arduino programming and TFLite deployment basics
- Basic programming concepts (variables, loops, functions)
- Familiarity with basic electronics (voltage, resistance, circuits)
- Arduino IDE 2.0+ installed and configured

Hardware Requirements

- **Level 1:** Laptop with Arduino IDE 2.0+ installed
- **Level 2:** Arduino Uno/Nano + breadboard + basic sensors (LDR, thermistor)
- **Level 3:** Arduino Nano 33 BLE Sense with integrated sensors

⚠ No Median Filtering = Noise Spikes Trigger False Readings

A single electrical spike can cause your ADC to read 1023 (max) for one sample, triggering false alarms in your ML model. Raw sensor data is rarely clean enough for direct ML input. **Always apply at least a moving average filter**, and consider median filtering for spike rejection. Test by disconnecting and reconnecting your sensor—your system should handle it gracefully.

Theory and practice. This chapter bridges the gap between software ML development and physical hardware. Understanding sensor interfaces is essential for deploying ML models that interact with the real world. We focus on practical skills needed for TinyML applications. **Execution tiers.** Level 1: Simulate sensor behavior in Python notebooks. Level 2: Program Arduino boards connected via USB. Level 3: Deploy standalone sensor nodes for continuous data collection.

Three-Tier Execution Model

Tier 1: Jupyter/Colab

Simulate sensor data patterns, design preprocessing pipelines, create synthetic datasets for testing ML models **Tier 2: Arduino Uno/Nano**

Read analog/digital sensors (temperature, light, motion), implement filtering and calibration, collect training data via serial **Tier 3: Arduino 33 BLE Sense**

Integrate IMU, microphone, and environmental sensors, run on-device ML inference, deploy autonomous sensor nodes

8.1 Introduction: Sensors as ML Data Sources

Machine learning models on edge devices need real-world data. Sensors are the bridge between the physical world and digital ML systems.

Why Sensor Knowledge Matters for Edge ML

Understanding sensors is crucial because:

- **Data quality:** Sensor noise directly affects ML accuracy
- **Sampling rates:** Must match the phenomena you're detecting
- **Preprocessing:** Many sensors need calibration and filtering
- **Power budget:** Sensors often dominate edge device power consumption

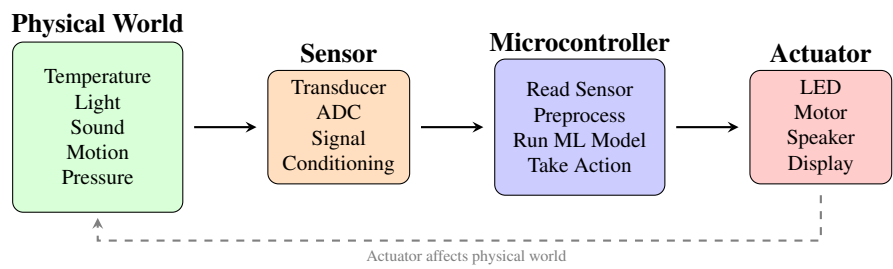


Figure 8.1: The edge ML sensing loop: Sensors convert physical quantities to digital values that MCUs can process. ML models make decisions, and actuators affect the physical world—creating a feedback loop.

8.2 Sensor Fundamentals

8.2.1 Analog vs Digital Sensors

Sensors output either analog (continuous) or digital (discrete) signals:

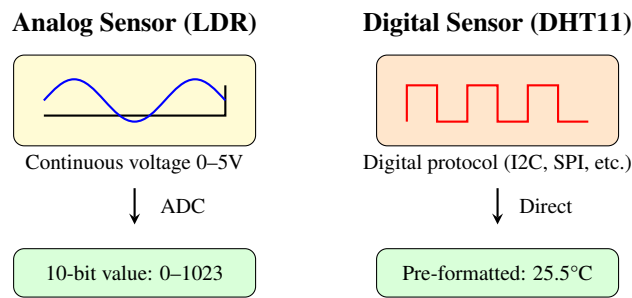


Figure 8.2: Analog sensors output continuous voltages that require ADC conversion. Digital sensors communicate via protocols and provide pre-processed values.

Table 8.1: Common sensor types and their interfaces

Sensor Type	Measures	Interface	Typical Output
LDR (Photoresistor)	Light intensity	Analog	Resistance (voltage divider)
Thermistor	Temperature	Analog	Resistance
Potentiometer	Position/angle	Analog	Voltage
DHT11/22	Temp + Humidity	Digital (1-wire)	Pre-formatted values
MPU6050	Accel + Gyro	Digital (I2C)	6-axis motion data
HC-SR04	Distance	Digital (pulse)	Timing measurement
Microphone (PDM)	Sound	Digital (PDM/I2S)	Audio samples

8.2.2 Understanding Analog-to-Digital Conversion

Arduino’s ADC converts analog voltages to digital values:

ADC Formula

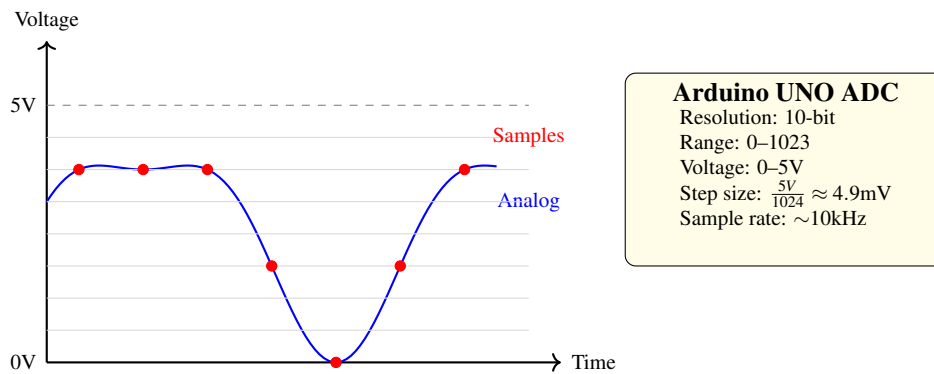


Figure 8.3: ADC conversion: The analog signal (blue) is sampled at discrete intervals. Each sample is quantized to one of 1024 levels (10-bit resolution). Higher resolution = finer voltage distinctions.

To convert an ADC reading to voltage:

$$\text{Voltage} = \frac{\text{ADC\_Value} \times V_{ref}}{2^{\text{resolution}} - 1} = \frac{\text{ADC\_Value} \times 5.0}{1023}$$

For example, an ADC reading of 512 corresponds to approximately 2.5V.

8.3 Signal Acquisition Theory

Before implementing sensor systems, understanding the mathematical foundations of signal acquisition is essential. This section covers the theory behind sampling, quantization, noise, and filtering—critical concepts that determine data quality for edge ML applications.

8.3.1 Nyquist-Shannon Sampling Theorem

The Nyquist-Shannon sampling theorem establishes the minimum sampling rate needed to accurately capture a continuous signal.

Theorem — Nyquist-Shannon Sampling Theorem. A continuous signal containing frequency components up to a maximum frequency $f_{\max}$ can be perfectly reconstructed from its samples if and only if the sampling frequency f_s satisfies:

$$f_s \geq 2f_{\max}$$

The minimum sampling rate $f_{\text{Nyquist}} = 2f_{\max}$ is called the **Nyquist rate**.

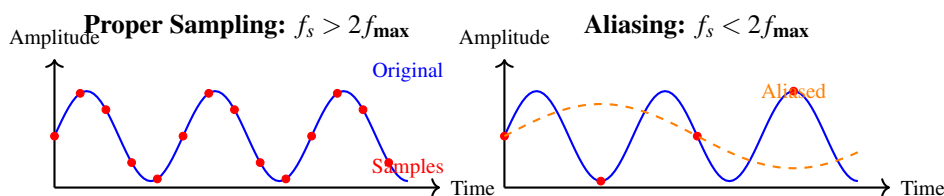


Figure 8.4: Nyquist-Shannon sampling theorem: Sampling above the Nyquist rate preserves signal information (left). Undersampling causes aliasing—the reconstructed signal has the wrong frequency (right).

Definition — Aliasing. Aliasing occurs when a signal is sampled below the Nyquist rate. High-frequency components in the original signal appear as false lower frequencies in the sampled data:

$$f_{\text{alias}} = |f_{\text{signal}} - k \cdot f_s|$$

where k is an integer chosen such that $f_{\text{alias}} < f_s/2$.

Practical Example: Accelerometer Sampling

Consider gesture recognition using an accelerometer:

- **Application:** Detecting hand waves and punches
- **Typical motion frequency:** Human hand movements are typically 1–10 Hz
- **Maximum frequency:** Fast gestures may contain components up to $f_{\text{max}} = 15$ Hz
- **Nyquist rate:** $f_{\text{Nyquist}} = 2 \times 15 = 30$ Hz
- **Recommended sampling rate:** In practice, use $f_s = 50$ – 100 Hz (3–6 times Nyquist) to ensure signal quality and account for filter roll-off

What if we sample at only 20 Hz? A 15 Hz motion component would alias to $|15 - 20| = 5$ Hz, causing the ML model to see incorrect patterns.

Practical Nyquist Considerations

The Nyquist theorem assumes:

- The signal is perfectly band-limited (no frequencies above f_{max})
- Infinite sampling precision
- Perfect reconstruction filters

In practice, use sampling rates 3–5 times higher than Nyquist to account for real-world imperfections and anti-aliasing filter roll-off.

8.3.2 ADC Quantization Theory

Analog-to-Digital Converters (ADCs) discretize continuous voltage signals into digital values. This quantization process introduces fundamental limits on measurement precision.

Definition — ADC Resolution and Quantization Step. For an ADC with n -bit resolution and reference voltage V_{ref} , the **quantization step size** (voltage resolution) is:

$$\Delta = \frac{V_{\text{ref}}}{2^n}$$

The ADC maps continuous voltages to discrete output codes:

$$\text{ADC\_Code} = \left\lfloor \frac{V_{\text{in}}}{\Delta} \right\rfloor = \left\lfloor \frac{V_{\text{in}} \cdot 2^n}{V_{\text{ref}}} \right\rfloor$$

Table 8.2: Common ADC resolutions and quantization step sizes

Resolution	Levels	Δ (for $V_{\text{ref}} = 5\text{V}$)	Example MCU
8-bit	256	19.5 mV	ESP8266
10-bit	1024	4.88 mV	Arduino UNO
12-bit	4096	1.22 mV	Arduino Due, STM32
16-bit	65536	76.3 μV	ADS1115 (external)

Worked Example: Arduino UNO ADC

Arduino UNO has a 10-bit ADC with $V_{\text{ref}} = 5\text{V}$:

$$\Delta = \frac{5\text{V}}{2^{10}} = \frac{5\text{V}}{1024} = 4.88 \text{ mV}$$

ADC range: 0 to 1023

Input voltage $V_{\text{in}} = 2.5\text{V}$:

$$\text{ADC\_Code} = \left\lfloor \frac{2.5 \times 1024}{5.0} \right\rfloor = \lfloor 512 \rfloor = 512$$

The ADC cannot distinguish voltages closer than 4.88 mV—this sets the fundamental precision limit.

Definition — Quantization Error. The **quantization error** e_q is the difference between the actual analog voltage and the quantized digital representation:

$$e_q = V_{\text{in}} - \text{ADC\_Code} \times \Delta$$

For uniform quantization with rounding, the error is bounded:

$$|e_q| \leq \frac{\Delta}{2}$$

This error is unavoidable and represents the fundamental noise floor of the ADC.

Theorem — ADC Signal-to-Noise Ratio. For an ideal n -bit ADC with a full-scale sinusoidal input, the Signal-to-Quantization-Noise Ratio (SQNR) is:

$$\text{SNR}_{\text{dB}} = 6.02n + 1.76$$

Each additional bit of resolution improves SNR by approximately 6 dB (a factor of 2 in voltage, 4 in power).

Derivation Sketch. For a full-scale sinusoidal signal with amplitude $A = V_{\text{ref}}/2$:

- Signal power: $P_{\text{signal}} = \frac{A^2}{2} = \frac{V_{\text{ref}}^2}{8}$
- Quantization noise is uniformly distributed over $[-\Delta/2, \Delta/2]$
- Quantization noise power: $P_{\text{noise}} = \frac{\Delta^2}{12} = \frac{V_{\text{ref}}^2}{12 \cdot 2^{2n}}$
- SNR ratio: $\text{SNR} = \frac{P_{\text{signal}}}{P_{\text{noise}}} = \frac{3 \cdot 2^{2n}}{2}$
- In decibels: $\text{SNR}_{\text{dB}} = 10 \log_{10}(1.5 \cdot 2^{2n}) = 10 \log_{10}(1.5) + 20n \log_{10}(2) \approx 1.76 + 6.02n$



Table 8.3: Theoretical SNR for different ADC resolutions

Resolution	SNR (dB)	Dynamic Range
8-bit	49.9 dB	~256:1
10-bit	61.9 dB	~1024:1
12-bit	74.0 dB	~4096:1
16-bit	98.1 dB	~65536:1

Choosing ADC Resolution for ML

For edge ML sensor applications:

- **10-bit** is sufficient for most environmental sensors (temperature, light, humidity)
- **12-bit** is better for audio and precision measurements
- **16-bit** is needed for high-fidelity audio or scientific instrumentation
- Higher resolution requires longer conversion time and more power

The ADC resolution should match your sensor's precision—a 10-bit ADC is wasted on a sensor with 5% accuracy.

8.3.3 Noise Analysis and Signal Conditioning

Real sensor systems contain multiple noise sources that degrade measurement quality. Understanding and mitigating noise is critical for reliable ML models.

Definition — Johnson-Nyquist Thermal Noise. Thermal noise (Johnson noise) is fundamental electrical noise caused by random thermal motion of charge carriers in resistors:

$$v_n = \sqrt{4kTRB}$$

where:

- $k = 1.38 \times 10^{-23}$ J/K (Boltzmann constant)
- T is temperature in Kelvin
- R is resistance in Ohms
- B is bandwidth in Hertz

Worked Example: LDR Thermal Noise

Consider an LDR with $R = 10 \text{ k}\Omega$ at room temperature ($T = 300 \text{ K}$), measured with bandwidth $B = 10 \text{ kHz}$:

$$\begin{aligned} v_n &= \sqrt{4 \times 1.38 \times 10^{-23} \times 300 \times 10^4 \times 10^4} \\ &= \sqrt{1.656 \times 10^{-12}} \\ &\approx 1.29 \text{ }\mu\text{V (RMS)} \end{aligned}$$

For a 5V signal, this thermal noise is negligible. However, for a 10 mV sensor signal, the SNR is:

$$\text{SNR}_{\text{thermal}} = 20 \log_{10} \left(\frac{10 \text{ mV}}{1.29 \text{ }\mu\text{V}} \right) \approx 77.8 \text{ dB}$$

Multiple Noise Sources

Real sensor systems have multiple noise contributors:

1. **Thermal noise:** From resistances in the circuit
2. **Shot noise:** From discrete electron flow (important in low-current sensors)
3. **Flicker noise (1/f):** Low-frequency noise in semiconductors
4. **Quantization noise:** From ADC digitization
5. **Environmental noise:** Power supply ripple, EMI, ground loops

Total noise power is the sum of individual noise powers:

$$v_{\text{total}}^2 = v_{\text{thermal}}^2 + v_{\text{shot}}^2 + v_{\text{quantization}}^2 + v_{\text{EMI}}^2 + \dots$$

Theorem — Noise Reduction by Averaging. Averaging N independent measurements reduces noise by a factor of $\sqrt{N}$. The SNR improvement is:

$$\text{SNR}_{\text{avg}} = \text{SNR}_{\text{single}} + 10\log_{10}(N)$$

For example, averaging 100 samples improves SNR by 20 dB.

Intuition. If noise samples are independent with variance σ^2 :

- Signal after averaging N samples: $\bar{s} = s$ (signal unchanged)
- Noise variance after averaging: $\text{Var}(\bar{n}) = \frac{\sigma^2}{N}$
- Standard deviation (RMS noise): $\sigma_{\text{avg}} = \frac{\sigma}{\sqrt{N}}$
- SNR improvement: $\frac{\text{SNR}_{\text{avg}}}{\text{SNR}_{\text{single}}} = \sqrt{N}$

■

Practical Example: Noise Reduction in Arduino

An LDR sensor has SNR = 40 dB (100:1 ratio) for single readings. By averaging different numbers of samples:

$$N = 4: \quad \text{SNR} = 40 + 10\log_{10}(4) = 40 + 6 = 46 \text{ dB}$$

$$N = 10: \quad \text{SNR} = 40 + 10\log_{10}(10) = 40 + 10 = 50 \text{ dB}$$

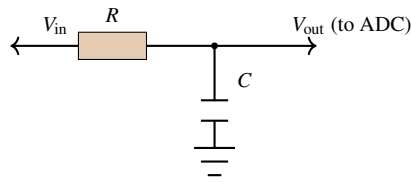
$$N = 100: \quad \text{SNR} = 40 + 10\log_{10}(100) = 40 + 20 = 60 \text{ dB}$$

However, averaging costs time and energy. For a 10 ms sampling period, averaging 100 samples takes 1 second—a significant lag for real-time applications.

8.3.4 Anti-Aliasing Filters

To prevent aliasing, signals must be band-limited before sampling. An **anti-aliasing filter** removes frequency components above $f_s/2$ before the ADC.

Definition — RC Low-Pass Filter. The simplest anti-aliasing filter is a first-order RC low-pass filter:



The **cutoff frequency** (3 dB point) is:

$$f_c = \frac{1}{2\pi RC}$$

At frequencies $f \gg f_c$, attenuation is approximately 20 dB/decade (factor of 10 reduction per 10× frequency increase).

Worked Example: Designing an Anti-Aliasing Filter

Design an anti-aliasing filter for a 100 Hz sampling rate accelerometer:

Step 1: Determine required cutoff frequency. Use $f_c = f_s/4 = 25$ Hz to provide safety margin.

Step 2: Choose component values. Select $R = 10$ kΩ (common value):

$$f_c = \frac{1}{2\pi RC}$$

$$C = \frac{1}{2\pi R f_c} = \frac{1}{2\pi \times 10^4 \times 25} \approx 0.64 \mu\text{F}$$

Use standard value: $C = 0.68 \mu\text{F}$ (actual $f_c \approx 23.4$ Hz).

Step 3: Verify attenuation at Nyquist frequency ($f_s/2 = 50$ Hz):

$$\text{Attenuation} = -20 \log_{10} \left(\sqrt{1 + \left(\frac{f}{f_c} \right)^2} \right) = -20 \log_{10} \left(\sqrt{1 + \left(\frac{50}{23.4} \right)^2} \right) \approx -6.8 \text{ dB}$$

Signals at 50 Hz are reduced by approximately 50%.

RC Filter Limitations

First-order RC filters have gentle roll-off (20 dB/decade). For better aliasing protection:

- Use multiple RC stages (second-order: 40 dB/decade)
- Consider active filters (op-amp based Butterworth or Chebyshev designs)
- Trade-off: Higher-order filters have more phase distortion

For many edge ML applications, a simple single-stage RC filter is sufficient when combined with oversampling.

Practical Anti-Aliasing Strategy for Arduino

For Arduino sensor systems:

1. **Add hardware RC filter:** Use $f_c \approx f_s/4$ to remove high-frequency noise
2. **Oversample:** Sample at 3–5× Nyquist rate
3. **Digital filtering:** Apply moving average or FIR filter in software

4. **Decimate:** Downsample to final rate for ML model

This multi-stage approach provides robust aliasing protection with minimal hardware cost.

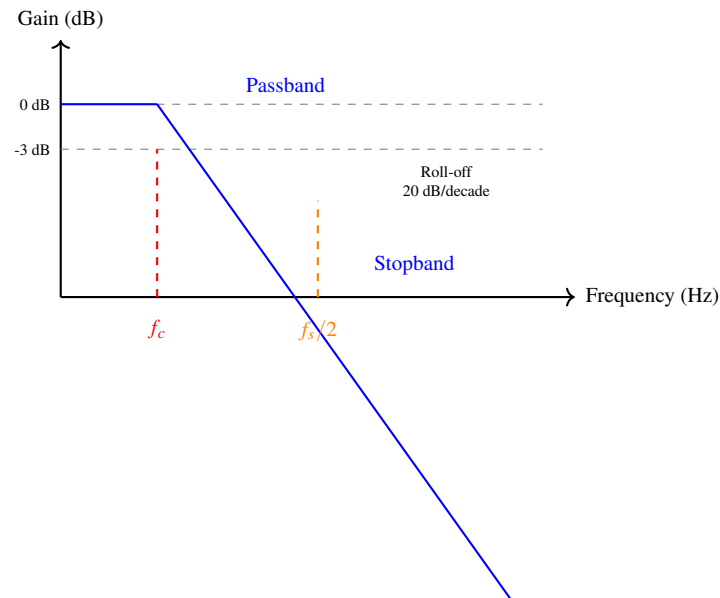


Figure 8.5: RC low-pass filter frequency response. Frequencies below f_c pass largely unaffected. Frequencies above f_c are attenuated, preventing aliasing at the Nyquist frequency $f_s/2$.

💡 Why Not Just Use Software Filtering?

Consider these questions:

1. If you can filter in software (Arduino code), why bother with hardware RC filters?
2. What happens if a 1000 Hz noise signal reaches the ADC when you're sampling at 100 Hz?
3. Can a moving average filter in software remove aliased signals that are already in your data?
4. Why is the order of operations important: filter first, then sample—not the reverse?

Hint: Once aliasing occurs, the information is corrupted. Software can smooth noise but cannot un-alias already-sampled data!

8.4 Arduino Development Environment

8.4.1 Installing Arduino IDE

1. **Download:** Visit <https://www.arduino.cc/en/software> and download the IDE for your operating system
2. **Install:** Run the installer and follow the prompts
3. **Connect:** Use a USB cable to connect your Arduino board
4. **Configure:** Select your board type and COM port from the Tools menu

Driver Installation

On Windows, you may need to install drivers for your Arduino board. The IDE usually

handles this automatically, but if your board isn't recognized:

- Open Device Manager
- Find the unknown device under "Ports"
- Right-click and select "Update Driver"
- Choose "Browse my computer" and navigate to the Arduino IDE drivers folder

8.4.2 Arduino Program Structure

Every Arduino program (called a "sketch") has two required functions:

```

1 // setup() runs once when the board powers on or resets
2 void setup() {
3     // Initialize pins, serial communication, etc.
4     Serial.begin(9600);    // Start serial at 9600 baud
5     pinMode(13, OUTPUT);   // Configure pin 13 as output
6 }
7
8 // loop() runs continuously after setup()
9 void loop() {
10    // Main program logic here
11    digitalWrite(13, HIGH); // Turn LED on
12    delay(1000);           // Wait 1 second
13    digitalWrite(13, LOW);  // Turn LED off
14    delay(1000);           // Wait 1 second
15 }
```

Code Walkthrough

setup(): Runs once at startup. Use for initialization: pin modes, serial communication, sensor calibration.

loop(): Runs continuously after setup(). This is your main program—read sensors, process data, control actuators.

Serial.begin(9600): Initializes serial communication at 9600 bits per second for debugging output.

8.5 Reading Analog Sensors

8.5.1 Light Sensor (LDR) Circuit

A Light Dependent Resistor (LDR) changes resistance based on light intensity. We use a voltage divider circuit to convert resistance to voltage:

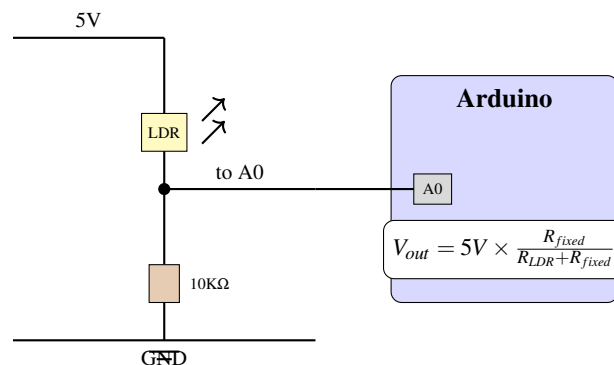


Figure 8.6: LDR voltage divider circuit. When light increases, LDR resistance decreases, causing V_{out} to increase. The Arduino ADC reads this voltage on pin A0.

Voltage Divider Intuition

In a voltage divider:

- **More light** → Lower LDR resistance → More voltage drop across fixed resistor → Higher V_{out}
- **Less light** → Higher LDR resistance → Less voltage drop across fixed resistor → Lower V_{out}

Choose the fixed resistor value to be similar to the LDR's mid-range resistance for best sensitivity.

8.5.2 Light Sensor Sketch

```
1 // Light sensor with LED indicator
2 #define LED_PIN 13
3 #define LDR_PIN A0
4 #define LIGHT_THRESHOLD 500 // Adjust based on your environment
```

Code Walkthrough

#define: Creates named constants. Using names like LDR_PIN instead of raw numbers makes code readable and easy to modify if you change pin assignments.

```
1 void setup() {
2     Serial.begin(9600);
3     pinMode(LED_PIN, OUTPUT);
4     // Analog pins don't need pinMode – they're input by default
5
6     Serial.println("Light Sensor Initialized");
7     Serial.println("Format: raw_value, voltage, status");
8 }
```

Code Walkthrough

Serial.println(): Prints text followed by a newline. Use for debugging and data output to the Serial Monitor (Tools → Serial Monitor in Arduino IDE).

Analog pins: Don't require pinMode()—they're configured for input automatically.

```
1 void loop() {
2     // Read the analog value (0–1023)
3     int lightValue = analogRead(LDR_PIN);
4
5     // Convert to voltage
6     float voltage = lightValue * (5.0 / 1023.0);
7
8     // Determine if it's light or dark
9     bool isLight = lightValue > LIGHT_THRESHOLD;
10
11    // Control LED based on light level
12    digitalWrite(LED_PIN, isLight ? LOW : HIGH);
13
14    // Output data in CSV format for easy logging
15    Serial.print(lightValue);
16    Serial.print(", ");
17    Serial.print(voltage, 2); // 2 decimal places
18    Serial.print(", ");
19    Serial.println(isLight ? "LIGHT" : "DARK");
20
21    delay(100); // Sample at 10 Hz
```

22 }

Code Walkthrough

analogRead(): Returns a value 0–1023 representing the voltage on the analog pin (0V–5V).

CSV format: Printing data as comma-separated values makes it easy to copy/paste into spreadsheets or Python for analysis.

delay(100): Pauses for 100ms, giving a 10 Hz sampling rate. Adjust based on how fast your signal changes.

Expected Serial Monitor Output:

```
Light Sensor Initialized
Format: raw_value, voltage, status
523, 2.56, LIGHT
518, 2.53, LIGHT
245, 1.20, DARK
251, 1.23, DARK
612, 2.99, LIGHT
```

8.6 Reading Digital Sensors

Digital sensors communicate via protocols and provide pre-processed data.

8.6.1 DHT11 Temperature and Humidity Sensor

The DHT11 uses a single-wire protocol to send temperature and humidity readings:

```
1 #include <DHT.h> // Install via Library Manager
2
3 #define DHT_PIN 2
4 #define DHT_TYPE DHT11
5
6 DHT dht(DHT_PIN, DHT_TYPE);
```

Code Walkthrough

#include <DHT.h>: Includes the DHT library. Install via Sketch → Include Library → Manage Libraries, then search for “DHT sensor library” by Adafruit.

DHT object: Creates a sensor object configured for pin 2 and DHT11 sensor type.

```
1 void setup() {
2     Serial.begin(9600);
3     dht.begin();
4     Serial.println("DHT11 Sensor Ready");
5     delay(2000); // DHT11 needs time to stabilize
6 }
7
8 void loop() {
9     // Read humidity and temperature
10    float humidity = dht.readHumidity();
11    float tempC = dht.readTemperature(); // Celsius
12    float tempF = dht.readTemperature(true); // Fahrenheit
13
14    // Check for read errors
15    if (isnan(humidity) || isnan(tempC)) {
16        Serial.println("ERROR: Failed to read from DHT sensor");
17        return;
18    }
```

```

19
20 // Compute heat index
21 float heatIndex = dht.computeHeatIndex(tempC, humidity, false);
22
23 // Output readings
24 Serial.print("Temp: ");
25 Serial.print(tempC);
26 Serial.print("C, Humidity: ");
27 Serial.print(humidity);
28 Serial.print("%, Heat Index: ");
29 Serial.print(heatIndex);
30 Serial.println("C");
31
32 delay(2000); // DHT11 max sample rate is 1 Hz
33 }

```

DHT11 Sampling Rate

The DHT11 can only be sampled once every 1-2 seconds. Sampling faster will return stale data or errors. The DHT22 is faster (0.5 Hz) and more accurate but more expensive.

8.7 Sensor Data for Machine Learning

Collecting high-quality sensor data is essential for training ML models.

8.7.1 Data Collection Best Practices

ML Data Collection Guidelines

When collecting sensor data for ML training:

1. **Consistent sampling:** Use fixed intervals (e.g., 100ms)
2. **Include timestamps:** Track when each sample was taken
3. **Label your data:** Record what class/state each sample represents
4. **Capture variety:** Include different conditions, orientations, users
5. **Record metadata:** Note sensor type, placement, environmental conditions

8.7.2 Accelerometer Data Collection for Gesture Recognition

This example collects accelerometer data for training a gesture classifier:

```

1 #include <Arduino_LSM9DS1.h> // For Nano 33 BLE Sense
2
3 #define SAMPLES_PER_GESTURE 119 // ~1 second at 119 Hz
4 #define NUM_GESTURES 4
5
6 const char* GESTURES[] = {"punch", "flex", "wave", "idle"};
7 int currentGesture = 0;
8
9 void setup() {
10   Serial.begin(115200);
11
12   if (!IMU.begin()) {
13     Serial.println("Failed to initialize IMU!");
14     while (1);
15   }
16
17   // Print CSV header
18   Serial.println("aX,aY,aZ,gesture");
19 }

```

```

11     Serial.println("Collecting data for: " +
12                     String(GESTURES[currentGesture]));
13 }

1 void loop() {
2     float aX, aY, aZ;
3
4     // Wait for significant motion to start recording
5     while (!motionDetected()) {
6         // Idle waiting
7     }
8
9     // Collect one gesture's worth of samples
10    for (int i = 0; i < SAMPLES_PER_GESTURE; i++) {
11        if (IMU.accelerationAvailable()) {
12            IMU.readAcceleration(aX, aY, aZ);
13
14            // Print in CSV format with gesture label
15            Serial.print(aX, 3);
16            Serial.print(",");
17            Serial.print(aY, 3);
18            Serial.print(",");
19            Serial.print(aZ, 3);
20            Serial.print(",");
21            Serial.println(GESTURES[currentGesture]);
22        }
23        delay(8); // ~125 Hz sampling
24    }
25
26    Serial.println("--- Gesture complete ---");
27    delay(1000); // Pause between gestures
28 }
29
30 bool motionDetected() {
31     float aX, aY, aZ;
32     if (IMU.accelerationAvailable()) {
33         IMU.readAcceleration(aX, aY, aZ);
34         float magnitude = sqrt(aX*aX + aY*aY + aZ*aZ);
35         return magnitude > 1.5; // Threshold for motion start
36     }
37     return false;
38 }

```

Code Walkthrough

Motion detection trigger: Waits for acceleration magnitude to exceed 1.5g before recording. This ensures we capture the actual gesture, not idle time.

Fixed sample count: Collects exactly 119 samples per gesture, creating consistent-sized input windows for ML models.

CSV output: Data is printed in comma-separated format for easy import into Python/Pandas.

8.7.3 Collecting Data via Python

Level 1: Data Collection Script

Use Python to collect and save sensor data from the serial port:

```

1 import serial
2 import csv

```

```

3 import time
4
5 # Configure serial port
6 ser = serial.Serial('/dev/ttyUSB0', 115200) # Adjust port
7 time.sleep(2) # Wait for Arduino to reset
8
9 # Open CSV file for writing
10 with open('gesture_data.csv', 'w', newline='') as csvfile:
11     writer = csv.writer(csvfile)
12     writer.writerow(['aX', 'aY', 'aZ', 'gesture', 'timestamp'])
13
14     print("Collecting data... Press Ctrl+C to stop")
15     try:
16         while True:
17             line = ser.readline().decode('utf-8').strip()
18             if ',' in line and not line.startswith('---'):
19                 # Add timestamp and write to CSV
20                 timestamp = time.time()
21                 data = line.split(',') + [timestamp]
22                 writer.writerow(data)
23                 print(f"Recorded: {line}")
24     except KeyboardInterrupt:
25         print("Data collection complete!")
26
27 ser.close()

```

8.8 Sensor Signal Preprocessing

Raw sensor data often contains noise. Preprocessing improves ML model accuracy.

8.8.1 Moving Average Filter

A simple but effective noise reduction technique:

```

1 #define WINDOW_SIZE 10
2
3 int readings[WINDOW_SIZE];
4 int readIndex = 0;
5 long total = 0;
6
7 void setup() {
8     Serial.begin(9600);
9     // Initialize readings array to 0
10    for (int i = 0; i < WINDOW_SIZE; i++) {
11        readings[i] = 0;
12    }
13 }
14
15 int movingAverage(int newReading) {
16    // Subtract oldest reading from total
17    total -= readings[readIndex];
18    // Add new reading
19    readings[readIndex] = newReading;
20    total += newReading;
21    // Advance index (circular buffer)
22    readIndex = (readIndex + 1) % WINDOW_SIZE;
23    // Return average
24    return total / WINDOW_SIZE;
25 }

```



```

26
27 void loop() {
28     int raw = analogRead(A0);
29     int filtered = movingAverage(raw);
30
31     Serial.print(raw);
32     Serial.print(",");
33     Serial.println(filtered);
34
35     delay(10);
36 }

```

Code Walkthrough

Circular buffer: Efficiently stores the last N readings without shifting array elements.

Running total: Maintains the sum incrementally, avoiding $O(n)$ summation each iteration.

Trade-off: Larger window = smoother output but more lag. Choose based on your application.

💡 How Would You Detect a Disconnected Sensor?

Sensors fail in the field. Consider these scenarios:

1. Your LDR wire comes loose. What ADC value would you read? (Hint: floating input)
2. Your I2C temperature sensor loses power. What does `readTemperature()` return?
3. How could you distinguish “sensor reads 0” from “sensor is broken”?
4. Design a simple health check that runs every 10 seconds and logs sensor status.

Robust edge systems must handle sensor failures gracefully—never assume hardware works perfectly!

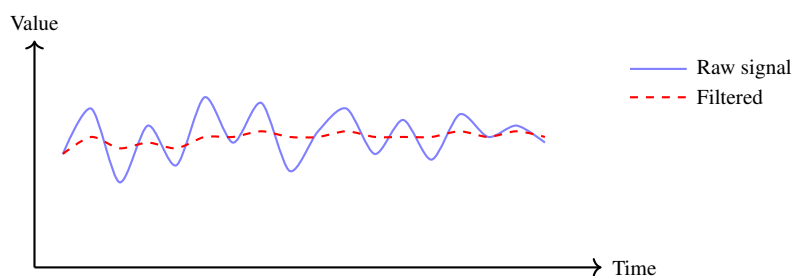


Figure 8.7: Moving average filter smooths noisy sensor readings, making patterns clearer for ML models. The trade-off is a slight delay in response to rapid changes.

8.9 Complete Light-Based ML Data Collector

Here’s a complete example that collects labeled light sensor data for training a classifier:

```

1 // Light sensor data collector for ML training
2 // Collects labeled samples of different lighting conditions
3
4 #define LDR_PIN A0
5 #define BUTTON_PIN 2
6 #define LED_PIN 13
7 #define SAMPLES_PER_CLASS 100
8 #define SAMPLE_INTERVAL 50 // ms
9

```

```

10 // Class labels
11 enum LightClass { DARK, DIM, BRIGHT, DIRECT_SUN };
12 const char* classNames[] = {"dark", "dim", "bright", "direct_sun"};
13 int currentClass = DARK;
14 int sampleCount = 0;
15
16 void setup() {
17     Serial.begin(115200);
18     pinMode(BUTTON_PIN, INPUT_PULLUP);
19     pinMode(LED_PIN, OUTPUT);
20
21     // Print CSV header
22     Serial.println("value,voltage,class");
23     Serial.println("# Press button to cycle through classes");
24     Serial.println("# Current class: dark");
25 }
26
27 void loop() {
28     // Check for button press to change class
29     if (digitalRead(BUTTON_PIN) == LOW) {
30         currentClass = (currentClass + 1) % 4;
31         Serial.print("# Switched to class: ");
32         Serial.println(classNames[currentClass]);
33         sampleCount = 0;
34
35         // Blink LED to indicate class number
36         for (int i = 0; i <= currentClass; i++) {
37             digitalWrite(LED_PIN, HIGH);
38             delay(200);
39             digitalWrite(LED_PIN, LOW);
40             delay(200);
41         }
42         delay(500); // Debounce
43     }
44
45     // Collect sample
46     if (sampleCount < SAMPLES_PER_CLASS) {
47         int raw = analogRead(LDR_PIN);
48         float voltage = raw * (5.0 / 1023.0);
49
50         Serial.print(raw);
51         Serial.print(",");
52         Serial.print(voltage, 3);
53         Serial.print(",");
54         Serial.println(classNames[currentClass]);
55
56         sampleCount++;
57
58         // Visual feedback
59         digitalWrite(LED_PIN, HIGH);
60         delay(10);
61         digitalWrite(LED_PIN, LOW);
62     }
63
64     delay(SAMPLE_INTERVAL);
65 }

```

Code Walkthrough

Button for class switching: Press to cycle through lighting condition classes.

Visual feedback: LED blinks indicate current class and sample collection.
CSV output: Ready to import into Python for model training.

8.10 Three-Tier Implementation Guide

Level 1: Simulating Sensors in Python

For development without hardware, simulate sensor data:

```

1 import numpy as np
2
3 def simulate_light_sensor(condition='indoor', noise_level=0.05):
4     """Simulate LDR readings for different conditions."""
5     base_values = {
6         'dark': 100,
7         'dim': 300,
8         'indoor': 500,
9         'bright': 700,
10        'sunlight': 900
11    }
12    base = base_values.get(condition, 500)
13    noise = np.random.normal(0, base * noise_level)
14    return int(np.clip(base + noise, 0, 1023))
15
16 # Generate training data
17 for i in range(100):
18     reading = simulate_light_sensor('indoor')
19     print(f"{reading},{reading * 5.0 / 1023:.3f},indoor")

```

Level 2: USB-Connected Development

Connect Arduino via USB for real sensor development:

1. Upload sketches using Arduino IDE
2. Monitor output via Serial Monitor
3. Collect data using Python serial scripts
4. Test preprocessing and data quality

Level 3: Standalone Sensor Nodes

Deploy autonomous sensor systems:

- Add SD card logging for offline data collection
- Use battery power with sleep modes for energy efficiency
- Implement wireless data transmission (BLE, WiFi, LoRa)
- Run ML inference on-device for real-time classification

8.11 Hardware Reference

Recommended Components

Starter Kit:

- Arduino UNO R3 or Nano 33 BLE Sense
- Breadboard and jumper wires
- Light sensor (LDR) with 10K Ω resistor
- DHT11 or DHT22 temperature/humidity sensor
- LEDs (various colors) with 220 Ω resistors
- Push buttons
- Potentiometer (10K Ω)

Advanced Sensors:

- MPU6050 (accelerometer + gyroscope)
- BMP280 (pressure + temperature)
- Microphone module (MAX4466 or PDM)
- Ultrasonic distance sensor (HC-SR04)

8.12 Troubleshooting Guide

⚠ Sensor Readings Are Unstable or Noisy

Symptoms: Sensor values fluctuate wildly, jumping between min and max, or show random spikes

Cause: Electrical noise, poor connections, insufficient power supply, or missing pull-up/pull-down resistors

Solution:

1. Add hardware filtering: Connect 0.1 μ F capacitor between sensor power and ground
2. Check connections: Ensure all wires firmly seated, no loose breadboard contacts
3. Use shorter wires: Long wires act as antennas picking up noise
4. Implement software filter: Moving average or median filter in code
5. Check power supply: Verify 3.3V or 5V stable with multimeter
6. Add pull-up resistors: For I2C, use 4.7k Ω resistors on SDA and SCL lines

⚠ I2C Device Not Found or No Acknowledgment

Symptoms: I2C device not found at address 0x... or `Wire.endTransmission()` returns error

Cause: Wrong I2C address, missing pull-up resistors, incorrect wiring, or device not powered

Solution:

1. Run I2C scanner sketch: Find actual device address
2. Check datasheet: Verify I2C address (some sensors have configurable addresses via jumpers)
3. Verify wiring: SDA to SDA, SCL to SCL, VCC and GND correct
4. Add pull-up resistors: 4.7k Ω or 10k Ω on SDA and SCL to 3.3V
5. Check voltage levels: Most sensors use 3.3V, some tolerate 5V
6. Try slower speed: Set `Wire.setClock(100000)`; for 100kHz instead of 400kHz

⚠ Analog Sensor Always Reads 0 or 1023

Symptoms: `analogRead()` returns only minimum (0) or maximum (1023/4095) values

Cause: Sensor output exceeds ADC input range, wrong reference voltage, or pin mode not set

Solution:

1. Check sensor output voltage: Should be 0-3.3V for most Arduino boards
2. Set correct reference: `analogReference(EXTERNAL)` if using external reference
3. Verify pin supports analog: Use A0, A1, etc. - not all pins have ADC
4. Check voltage divider: If sensor outputs 5V, use resistor divider to scale to 3.3V
5. Test with multimeter: Measure actual voltage at Arduino pin
6. Try different pin: ADC channel may be damaged

⚠ IMU/Accelerometer Values Don't Make Sense

Symptoms: Acceleration shows gravity in wrong direction, gyro drifts continuously, or values seem inverted

Cause: Axis orientation confusion, sensor not calibrated, or reading wrong registers

Solution:

1. Check axis orientation: Hold sensor still - Z-axis should read $\pm 1g$ (9.8 m/s^2)
2. Calibrate sensor: Many IMUs need offset calibration at startup
3. Verify data scaling: Check if values are in g, mg, or raw ADC counts
4. Read datasheet carefully: Note sensor orientation markings on PCB
5. Use sensor library: Adafruit or SparkFun libraries handle scaling automatically
6. Account for gravity: Subtract 1g from vertical axis when measuring acceleration

⚠ Servo Motor Jitters or Doesn't Move

Symptoms: Servo vibrates in place, moves erratically, or doesn't respond to commands

Cause: Insufficient power, PWM signal issues, servo damaged, or wrong pulse width range

Solution:

1. Use external power supply: Servos draw too much current for Arduino's 5V pin
2. Common ground: Connect servo GND and Arduino GND together
3. Check signal wire: Must connect to PWM-capable pin (usually marked ~)
4. Use Servo library: `#include <Servo.h>; servo.attach(pin); servo.write(90);`
5. Verify pulse width: Standard servos use $1000\text{--}2000\mu\text{s}$ (1-2ms) pulses
6. Test servo separately: Connect to separate power/microcontroller to verify it works

8.13 Summary

In this chapter, you learned:

- **Sensor fundamentals:** Analog sensors output continuous voltages; digital sensors use communication protocols
- **ADC conversion:** Arduino's 10-bit ADC maps 0–5V to values 0–1023
- **Voltage dividers:** Essential circuit for reading resistive sensors
- **Arduino programming:** `setup()` for initialization, `loop()` for main logic
- **Data collection:** CSV format, consistent sampling, proper labeling
- **Preprocessing:** Moving average filters reduce noise for better ML accuracy

Self-Assessment Questions

Test your understanding before continuing:

1. Explain the difference between analog and digital sensors. Why does Arduino use a 10-bit ADC instead of 8-bit or 16-bit?
2. An LDR voltage divider outputs 3.2V. With a 10-bit ADC and 5V reference, what ADC value (0-1023) do you read? Show your calculation.
3. Why is a voltage divider necessary when using resistive sensors like LDRs? What would happen if you connected the LDR directly between 5V and an analog pin?
4. When collecting sensor data for ML training, why is consistent sampling rate important? What problems arise from irregular timing?
5. A moving average filter with window size 10 reduces noise but introduces lag. What are the trade-offs between noise reduction and responsiveness for edge ML applications?

8.14 Guided Experiments

Experiment 1: Sensor Sampling Rate vs Accuracy

Hypothesis: Higher sampling rates capture more detail but increase power consumption and data volume. There exists an optimal sampling rate for each application.

Procedure:

1. Set up an accelerometer sensor (Arduino Nano 33 BLE Sense IMU)
2. Perform a repeatable motion (e.g., waving hand)
3. Sample the motion at different rates: 10Hz, 25Hz, 50Hz, 100Hz, 200Hz
4. Collect 20 samples of the same motion at each sampling rate
5. Measure power consumption and data size for each rate

Data Collection:

Sample Rate (Hz)	Samples per Motion	Current Draw (mA)	Data Size (bytes)
10			
25			
50			
100			
200			

Analysis Questions:

- What is the minimum sampling rate that captures the motion accurately?
- How does power consumption scale with sampling rate?
- For gesture recognition on battery power, which sampling rate offers the best trade-off?
- What happens if you sample below the Nyquist rate for your motion?

Experiment 2: Noise Filtering Comparison

Hypothesis: Different filtering techniques (moving average, median, exponential) have different effectiveness at removing noise while preserving signal features.

Procedure:

1. Read noisy sensor data from a light sensor (LDR) or temperature sensor
2. Implement three filtering approaches:
 - No filter (raw data)
 - Moving average (window size 5, 10, 20)

- Median filter (window size 5)
- Exponential moving average (alpha = 0.1, 0.3, 0.5)

3. Collect 500 samples with each filter
4. Calculate signal-to-noise ratio and response time for each

Data Collection:

Filter Type	SNR (dB)	Response Time (ms)	Spike Rejection
Raw (no filter)			
Moving Avg (5)			
Moving Avg (10)			
Median (5)			
EMA (0.3)			

Analysis Questions:

- Which filter best removes high-frequency noise?
- Which filter has the fastest response to sudden changes?
- For ML feature extraction, which filter preserves important signal characteristics?
- What is the trade-off between window size and lag for moving average?

 **Related Labs**

Prerequisites:

- **LAB05 - Embedded Systems Deployment:** Understanding of microcontroller programming and TensorFlow Lite Micro deployment is essential for integrating sensors with ML models.

Next Steps:

- **LAB09 - ESP32 Wireless Programming:** Extend sensor systems with wireless connectivity using WiFi, BLE, and MQTT for remote data collection and control.

8.15 Exercises

[Basic] Basic: Exercise 1: Multi-Sensor Data Logger

Create an Arduino sketch that:

1. Reads from both an LDR and a DHT11 sensor
2. Outputs combined readings in CSV format
3. Includes a timestamp (using `millis()`)

Collect 5 minutes of data and plot it in Python.

[Intermediate] Intermediate: Exercise 2: Calibrated Light Sensor

Improve the light sensor by:

1. Measuring ADC values at known light levels (use a light meter app)
2. Creating a calibration function that converts ADC to lux
3. Testing accuracy across different lighting conditions

[Intermediate] Intermediate: Exercise 3: Gesture Data Collection

If you have an Arduino Nano 33 BLE Sense (or similar with IMU):

1. Collect 50 samples each of 3 different hand gestures
2. Export the data to CSV
3. Visualize the accelerometer patterns in Python
4. Note which features distinguish your gestures

[Intermediate] Intermediate: Exercise 4: Adaptive Threshold

Modify the light sensor sketch to:

1. Automatically calibrate the threshold at startup
2. Adjust the threshold based on recent readings (adaptive)
3. Work correctly in different ambient lighting conditions

- **Further Reading.**
- Arduino Official Tutorials: <https://www.arduino.cc/en/Tutorial/BuiltInExamples>
 - Adafruit Sensor Guide: <https://learn.adafruit.com/category/sensors>
 - TensorFlow Lite Micro Data Collection: <https://www.tensorflow.org/lite/microcontrollers>

— **Credits.** This chapter draws on Arduino documentation and community resources. Circuit diagrams follow standard conventions for educational clarity.



Classic Bluetooth



Wireless devices streaming rich content like data, video, and audio
(device pairing required)

Bluetooth Low Energy (BLE)



Sensor devices sending small bits of data, using very little energy
(device pairing not required)



9. ESP32 Wireless Programming for Edge ML

Learning Objectives

After completing this lab, you will be able to:

- Understand ESP32 architecture and its advantages for edge ML applications
- Configure WiFi connectivity in station and access point modes
- Implement MQTT publish/subscribe patterns for IoT communication
- Use Bluetooth Low Energy (BLE) for short-range sensor data transmission
- Apply power management techniques for battery-powered edge nodes
- Build complete wireless edge ML systems with visual feedback

Interactive Notebook

Jupyter Notebook: LAB09_esp32_wireless.ipynb

Open in Google Colab | View on GitHub | Companion Website

Prerequisites

- **Knowledge:** Arduino basics (LAB08), networking fundamentals
- **Level 1:** Laptop with Arduino IDE and ESP32 board support
- **Level 2:** ESP32 DevKit + WiFi network + MQTT broker
- **Level 3:** Multiple ESP32 boards for mesh networking

Theory and practice. ESP32 combines dual-core processing, WiFi, and Bluetooth in a single chip—making it ideal for edge ML applications that need wireless connectivity. This chapter

teaches you to build complete wireless sensor nodes that run ML inference and communicate results.

Execution tiers. Level 1: Simulate ESP32 logic in Python. Level 2: Program ESP32 boards via USB with Arduino IDE. Level 3: Deploy battery-powered wireless nodes with mesh networking.

Prerequisites

Before starting this lab, ensure you:

- Completed LAB08 (Arduino Sensors): sensor data handling and Arduino programming basics
- Understanding of basic networking concepts (IP addresses, protocols)
- Familiarity with WiFi/MQTT concepts for IoT communication
- Arduino IDE 2.0+ with ESP32 board support installed

Three-Tier Execution Model

Tier 1: Jupyter/Colab

Design MQTT communication protocols, simulate wireless sensor networks, test data transmission patterns and latency

Tier 2: ESP32 DevKit

Configure WiFi connections, implement MQTT publish/subscribe, establish BLE communication, integrate sensors with wireless transmission

Tier 3: ESP32 Battery

Deploy autonomous wireless ML nodes with deep sleep, create mesh networks, implement OTA firmware updates, optimize for battery life

9.1 Introduction: ESP32 for Edge ML

The ESP32 is a powerful microcontroller designed for IoT applications. Its dual-core architecture allows running WiFi/Bluetooth on one core while ML inference runs on the other.

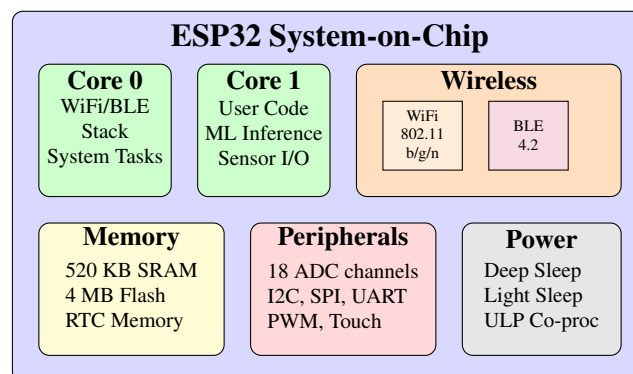


Figure 9.1: ESP32 architecture: Dual Xtensa cores, integrated WiFi/BLE, rich peripheral set, and multiple power modes make it ideal for battery-powered edge ML applications.

ESP32 vs Arduino for Edge ML

- **Processing power:** 240 MHz dual-core vs 16 MHz single-core
- **Memory:** 520 KB SRAM vs 2 KB SRAM
- **Wireless:** Built-in WiFi + BLE vs external modules
- **Power modes:** Deep sleep at $10\mu\text{A}$ vs no native sleep
- **Cost:** \$3-5 vs \$2-3

For edge ML with connectivity, ESP32 is the clear choice.

9.2 Wireless Communication Theory

Understanding the mathematical principles behind wireless communication is critical for designing reliable edge ML systems. This section provides the theoretical foundations for link budget analysis, channel capacity, and power optimization.

9.2.1 Shannon-Hartley Theorem: Channel Capacity

The Shannon-Hartley theorem establishes the fundamental limit of reliable data transmission over a noisy channel.

Theorem — Shannon-Hartley Theorem. The maximum theoretical data rate (channel capacity) C in bits per second for a communication channel with bandwidth B (Hz) and signal-to-noise ratio SNR is:

$$C = B \log_2(1 + \text{SNR}) \quad (9.1)$$

where SNR is the ratio of signal power to noise power (linear scale, not dB).

Intuition. The theorem shows that channel capacity increases logarithmically with SNR and linearly with bandwidth. This means:

- Doubling bandwidth doubles capacity (linear)
- Doubling signal power adds only 1 bit/s/Hz (logarithmic)
- Therefore, bandwidth is more valuable than power for high-rate transmission

**Practical Application: WiFi 802.11n**

For a 20 MHz channel with SNR = 30 dB:

$$\text{SNR}_{\text{linear}} = 10^{30/10} = 1000 \quad (9.2)$$

$$C = 20 \times 10^6 \times \log_2(1 + 1000) \quad (9.3)$$

$$= 20 \times 10^6 \times \log_2(1001) \quad (9.4)$$

$$= 20 \times 10^6 \times 9.97 \quad (9.5)$$

$$\approx 199.4 \text{ Mbps} \quad (9.6)$$

This is the theoretical maximum. Actual WiFi 802.11n rates (150 Mbps for single spatial stream) are lower due to:

- Protocol overhead (MAC headers, acknowledgments, beacons)
- Coding inefficiency (convolutional codes are not optimal)
- Channel estimation and synchronization losses
- Multiple access contention (CSMA/CA)

Shannon Limit in Edge ML

For edge devices:

- **BLE (1 Mbps, 2 MHz BW):** SNR $\approx$ 20 dB typical
- **WiFi 2.4 GHz (150 Mbps, 20 MHz BW):** SNR $\approx$ 25-35 dB typical
- **LoRa (0.3-50 kbps, 125 kHz BW):** Can operate at SNR $<$ 0 dB using spread spectrum

Low data rates allow robust communication even with weak signals—critical for battery-powered edge nodes.

■ **Example — Calculating Required SNR for Target Data Rate.** An edge ML node needs to transmit 100 kbps over a 1 MHz BLE channel. What minimum SNR is required?

Solution:

$$C = B \log_2(1 + \text{SNR}) \quad (9.7)$$

$$100 \times 10^3 = 1 \times 10^6 \times \log_2(1 + \text{SNR}) \quad (9.8)$$

$$0.1 = \log_2(1 + \text{SNR}) \quad (9.9)$$

$$2^{0.1} = 1 + \text{SNR} \quad (9.10)$$

$$1.0718 = 1 + \text{SNR} \quad (9.11)$$

$$\text{SNR} = 0.0718 \text{ (linear)} = -11.4 \text{ dB} \quad (9.12)$$

This shows that for low data rates, communication is possible even with very poor signal quality. In practice, BLE uses coding and modulation with additional margin, requiring SNR $>$ 0 dB for reliable 1 Mbps operation. ■

9.2.2 Path Loss Models

Path loss quantifies how signal power decreases with distance. Accurate path loss modeling is essential for predicting WiFi and BLE range in edge ML deployments.

Free Space Path Loss (FSPL)

In ideal conditions (line-of-sight, no obstacles), electromagnetic waves spread spherically:

Definition — Free Space Path Loss. The path loss in free space at distance d (meters) and frequency f (Hz) is:

$$\text{FSPL}(d) = 20 \log_{10}(d) + 20 \log_{10}(f) + 20 \log_{10} \left(\frac{4\pi}{c} \right) \quad (9.13)$$

where $c = 3 \times 10^8$ m/s is the speed of light. Simplifying for frequency in MHz and distance in meters:

$$\text{FSPL}(d)_{\text{dB}} = 20 \log_{10}(d) + 20 \log_{10}(f_{\text{MHz}}) - 27.55 \quad (9.14)$$

Example: WiFi at 2.4 GHz

At distance $d = 10$ m and $f = 2400$ MHz:

$$\text{FSPL}(10) = 20 \log_{10}(10) + 20 \log_{10}(2400) - 27.55 \quad (9.15)$$

$$= 20(1) + 20(3.38) - 27.55 \quad (9.16)$$

$$= 20 + 67.6 - 27.55 \quad (9.17)$$

$$= 60.05 \text{ dB} \quad (9.18)$$

This means the signal loses 60 dB of power over 10 meters in free space. If the transmitter outputs +20 dBm, the received power at 10 m is: $20 - 60 = -40$ dBm (before antenna gain).

Log-Distance Path Loss Model

Real environments have walls, furniture, and multipath reflections. The log-distance model accounts for these effects:

Definition — Log-Distance Path Loss.

$$PL(d) = PL(d_0) + 10n \log_{10} \left(\frac{d}{d_0} \right) + X_\sigma \quad (9.19)$$

where:

- $PL(d_0)$ is path loss at reference distance d_0 (typically 1 m)
- n is the path loss exponent (environment-dependent)
- X_σ is a zero-mean Gaussian random variable (dB) representing shadowing

Table 9.1: Path Loss Exponents for Different Environments

Environment	Path Loss Exponent (n)	Typical Range (m)
Free space (ideal)	2.0	Unlimited
Indoor (line-of-sight)	1.6 - 1.8	50-100
Indoor (obstructed)	2.0 - 2.5	20-50
Office (multiple walls)	2.5 - 3.5	10-30
Dense urban (street level)	3.0 - 5.0	100-500
Factory floor (metal)	2.0 - 3.0	20-100
Inside building (through floors)	4.0 - 6.0	10-20

■ **Example — Indoor WiFi Range Calculation.** An ESP32 WiFi transmitter outputs +20 dBm at 2.4 GHz. The receiver requires -70 dBm for reliable reception. In an office environment with $n = 3.0$ and $PL(1 \text{ m}) = 40 \text{ dB}$, what is the maximum range?

Solution:

$$\text{Total path loss} = P_{\text{tx}} - P_{\text{rx}} = 20 - (-70) = 90 \text{ dB} \quad (9.20)$$

$$90 = 40 + 10(3.0) \log_{10} \left(\frac{d}{1} \right) \quad (9.21)$$

$$50 = 30 \log_{10}(d) \quad (9.22)$$

$$\log_{10}(d) = \frac{50}{30} = 1.667 \quad (9.23)$$

$$d = 10^{1.667} \approx 46.4 \text{ meters} \quad (9.24)$$

In practice, range is often limited to 20-30 m due to interference, multipath fading, and the stochastic shadowing term X_σ . ■

9.2.3 RSSI and Link Budget Analysis

The link budget is a complete accounting of all gains and losses in a wireless link. It determines whether a communication link will work reliably.

Definition — Link Budget Equation. The received signal power P_r (dBm) is:

$$P_r = P_t + G_t + G_r - PL \quad (9.25)$$

where:

- P_t = Transmit power (dBm)
- G_t = Transmit antenna gain (dBi)
- G_r = Receive antenna gain (dBi)
- PL = Total path loss including all losses (dB)

For reliable communication, we require:

$$P_r \geq P_{\text{sens}} + M \quad (9.26)$$

where P_{sens} is receiver sensitivity (dBm) and M is the fade margin (typically 10-20 dB).

Received Signal Strength Indicator (RSSI)

RSSI is a measurement of received signal power, typically reported by WiFi and BLE modules. In ESP32, `WiFi.RSSI()` returns the value in dBm.

Table 9.2: RSSI Thresholds and Link Quality

RSSI (dBm)	Signal Quality	Typical Behavior
> -30	Excellent	Maximum throughput, 100% packet delivery
-30 to -50	Very Good	High throughput, reliable connection
-50 to -60	Good	Moderate throughput, occasional retransmissions
-60 to -70	Fair	Reduced throughput, increased latency
-70 to -80	Weak	Minimum throughput, frequent retransmissions
-80 to -90	Very Weak	Unreliable, frequent disconnections
< -90	Unusable	Connection impossible or fails immediately

RSSI Guidelines for Edge ML

For edge ML sensor nodes:

- **Target RSSI:** -60 dBm or better for reliable operation
- **Minimum RSSI:** -75 dBm (add error handling for retransmissions)
- **BLE vs WiFi:** BLE is more tolerant of weak signals due to lower data rate
- **Adaptive transmission:** Reduce transmit power when RSSI is strong to save battery

■ **Example — Complete Link Budget for ESP32-to-Router.** Design a WiFi link budget for an ESP32 sensor node 20 meters from a router in an office ($n = 2.5$).

Given:

- ESP32 TX power: $P_t = +20$ dBm (typical)
- ESP32 antenna gain: $G_t = 2$ dBi (PCB trace antenna)
- Router antenna gain: $G_r = 3$ dBi (external antenna)
- Router sensitivity: $P_{\text{sens}} = -85$ dBm (802.11n)
- Frequency: 2.4 GHz
- Reference path loss at 1 m: 40 dB
- Fade margin required: 15 dB

Solution:

$$\text{Path loss at 20 m: } PL(20) = 40 + 10(2.5) \log_{10}(20) \quad (9.27)$$

$$= 40 + 25(1.301) = 72.5 \text{ dB} \quad (9.28)$$

$$\text{Received power: } P_r = P_t + G_t + G_r - PL \quad (9.29)$$

$$= 20 + 2 + 3 - 72.5 \quad (9.30)$$

$$= -47.5 \text{ dBm} \quad (9.31)$$

$$\text{Link margin: } M = P_r - P_{\text{sens}} \quad (9.32)$$

$$= -47.5 - (-85) = 37.5 \text{ dB} \quad (9.33)$$

Conclusion: Link margin of 37.5 dB exceeds the required 15 dB, so the link is robust. The ESP32 can reliably communicate at this distance even with obstacles and interference. ■

9.2.4 Battery Life Modeling for Wireless Edge Devices

Battery life is the most critical constraint for wireless edge ML systems. Accurate modeling allows you to predict deployment lifetime and optimize duty cycles.

Duty-Cycled Current Consumption

Most edge devices alternate between active and sleep states. The average current draw is:

Definition — Average Current with Duty Cycle. For a device with duty cycle D (fraction of time active):

$$I_{\text{avg}} = D \cdot I_{\text{active}} + (1 - D) \cdot I_{\text{sleep}} \quad (9.34)$$

where:

- $D = \frac{t_{\text{active}}}{t_{\text{active}} + t_{\text{sleep}}}$ (duty cycle, 0 to 1)
- I_{active} = current during active period (mA)
- I_{sleep} = current during sleep period (mA)

Definition — Battery Life Formula. The battery life T_{life} (hours) for a battery with capacity C (mAh) is:

$$T_{\text{life}} = \frac{C \cdot \eta}{I_{\text{avg}}} \quad (9.35)$$

where η is the battery efficiency factor (typically 0.85-0.95 for LiPo batteries, accounting for self-discharge and voltage conversion losses).

■ **Example — ESP32 Deep Sleep Battery Life.** An ESP32 wakes every 5 minutes, reads a sensor, transmits via WiFi, then sleeps. Calculate battery life with a 2000 mAh LiPo.

Power profile:

- Deep sleep: $I_{\text{sleep}} = 10 \mu\text{A} = 0.01 \text{ mA}$
- WiFi connection: 3 seconds at 200 mA
- Sensor read + inference: 2 seconds at 80 mA
- Total active time: 5 seconds every 300 seconds

Solution:

$$\text{Duty cycle: } D = \frac{5}{300} = 0.0167 \quad (9.36)$$

$$\text{Active current: } I_{\text{active}} = \frac{(3 \times 200) + (2 \times 80)}{5} = \frac{760}{5} = 152 \text{ mA} \quad (9.37)$$

$$I_{\text{avg}} = 0.0167 \times 152 + (1 - 0.0167) \times 0.01 \quad (9.38)$$

$$= 2.54 + 0.0098 = 2.55 \text{ mA} \quad (9.39)$$

$$T_{\text{life}} = \frac{2000 \times 0.9}{2.55} = 706 \text{ hours} = 29.4 \text{ days} \quad (9.40)$$

Sensitivity analysis:

- If sleep current increases to 100 μA : $T_{\text{life}} = 20.8$ days (29% reduction)
- If wake interval is 10 minutes instead of 5: $T_{\text{life}} = 56.5$ days (92% increase)

■

Table 9.3: Battery Life for Common Edge ML Scenarios (2000 mAh LiPo)

Scenario	Wake Interval	Active Time	Avg Current	Battery Life
Always-on WiFi	N/A	100%	200 mA	10 hours
Continuous BLE	N/A	100%	50 mA	40 hours
WiFi burst (1 min)	1 min	5 s	17 mA	5 days
WiFi burst (5 min)	5 min	5 s	2.5 mA	29 days
WiFi burst (15 min)	15 min	5 s	1.1 mA	68 days
BLE advertisement	10 s	0.1 s	0.5 mA	160 days
LoRa (SF12)	10 min	2 s	0.35 mA	238 days
Deep sleep only	N/A	0%	0.01 mA	20,000 hours (2.3 years)

Battery Optimization Strategies

To maximize battery life in edge ML systems:

1. **Minimize wake frequency:** Doubling the sleep interval nearly doubles battery life
2. **Reduce active time:** Optimize WiFi connection time (use static IP, disable DHCP)
3. **Use appropriate protocol:** BLE for frequent updates, WiFi for high throughput
4. **Adaptive duty cycle:** Increase wake frequency only when anomalies detected
5. **Deep sleep is critical:** Sleep current dominates for wake intervals > 1 minute
6. **Energy per transmission:**
 - WiFi: 5-20 mJ per message (including connection overhead)
 - BLE: 0.1-0.5 mJ per advertisement
 - LoRa: 1-5 mJ per transmission (depends on spreading factor)
7. **Rule of thumb:** One WiFi transmission = 100 BLE advertisements = 10-100 ML inferences

■ **Example — Event-Driven Wake vs. Periodic Wake.** Compare two battery strategies for a vibration anomaly detector:

Strategy A (Periodic): Wake every 5 minutes, run ML inference (2s at 80 mA), transmit result via WiFi (3s at 200 mA).

Strategy B (Event-Driven): Stay in light sleep with ULP coprocessor monitoring accelerometer (0.8 mA). When vibration threshold exceeded, wake and transmit. Assume 1 anomaly per hour.

Strategy A:

$$I_{\text{avg}} = \frac{5}{300} \times \frac{(2 \times 80) + (3 \times 200)}{5} + \frac{295}{300} \times 0.01 \quad (9.41)$$

$$= 2.54 + 0.0098 = 2.55 \text{ mA} \quad (9.42)$$

Strategy B:

$$I_{\text{avg}} = \frac{3595}{3600} \times 0.8 + \frac{5}{3600} \times 152 \quad (9.43)$$

$$= 0.798 + 0.211 = 1.01 \text{ mA} \quad (9.44)$$

Battery life (2000 mAh):

- Strategy A: 29.4 days
- Strategy B: 74.1 days (2.5× improvement)

Conclusion: Event-driven architectures with ULP coprocessors are vastly more efficient when events are rare. However, they require more complex firmware and may miss events if the ULP threshold is not properly calibrated. ■

9.3 Development Environment Setup

9.3.1 Using Arduino IDE with ESP32

The easiest way to program ESP32 is using the Arduino IDE with ESP32 board support:

1. Open Arduino IDE preferences (File → Preferences)
2. Add to “Additional Board Manager URLs”:

```
https://raw.githubusercontent.com/espressif/arduino-esp32/gh-pages/
package_esp32_index.json
```

3. Open Boards Manager (Tools → Board → Boards Manager)
4. Search for “esp32” and install “ESP32 by Espressif Systems”
5. Select your board (e.g., “ESP32 Dev Module”) from Tools → Board

USB Driver Issues

If your ESP32 isn’t recognized:

- **CP2102 chips:** Install Silicon Labs CP210x driver
- **CH340 chips:** Install CH340 driver
- **Upload fails:** Hold BOOT button while uploading, release when “Connecting...” appears

9.4 WiFi Connectivity

WiFi enables edge devices to send inference results to cloud services, receive model updates, and coordinate with other devices.

9.4.1 Station Mode: Connecting to Networks

Station mode allows ESP32 to join existing WiFi networks—essential for cloud connectivity and MQTT communication.

```
1 #include <WiFi.h>
2
3 const char* ssid = "YourNetworkName";
4 const char* password = "YourPassword";
```

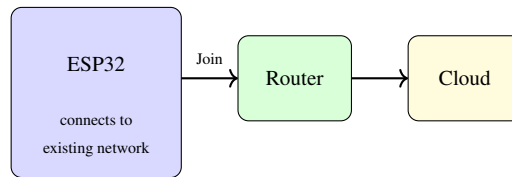
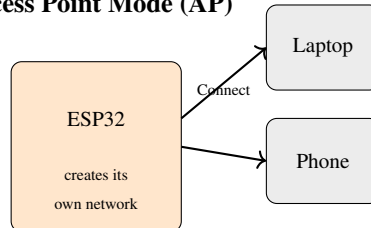
Station Mode (STA)**Access Point Mode (AP)**

Figure 9.2: ESP32 WiFi modes: Station mode joins existing networks for cloud connectivity. Access Point mode creates a local network for direct device communication.

Code Walkthrough

Include the WiFi library (built into ESP32 Arduino core). Store credentials as constants—in production, consider storing in non-volatile storage or using WiFiManager for configuration.

```

1 void setup() {
2     Serial.begin(115200);
3
4     Serial.print("Connecting to ");
5     Serial.println(ssid);
6
7     WiFi.begin(ssid, password);
8
9     // Wait for connection with timeout
10    int timeout = 20;
11    while (WiFi.status() != WL_CONNECTED && timeout > 0) {
12        delay(1000);
13        Serial.print(".");
14        timeout--;
15    }
  
```

Code Walkthrough

WiFi.begin(): Initiates connection to the specified network.

WiFi.status(): Returns connection state. WL_CONNECTED means successful.

Timeout: Prevents infinite loops if credentials are wrong or network is unavailable.

```

1     if (WiFi.status() == WL_CONNECTED) {
2         Serial.println("\nConnected!");
3         Serial.print("IP address: ");
4         Serial.println(WiFi.localIP());
5         Serial.print("Signal strength: ");
6         Serial.print(WiFi.RSSI());
7         Serial.println(" dBm");
8     } else {
  
```

```

9      Serial.println("\nConnection failed!");
10    }
11  }
12
13  void loop() {
14    // Monitor connection and reconnect if needed
15    if (WiFi.status() != WL_CONNECTED) {
16      Serial.println("Reconnecting ...");
17      WiFi.reconnect();
18      delay(5000);
19    }
20    delay(10000);
21  }

```

Expected Output:

```

Connecting to MyNetwork
.....
Connected!
IP address: 192.168.1.105
Signal strength: -45 dBm

```

9.4.2 Access Point Mode: Creating Networks

ESP32 can create its own WiFi network for direct device-to-device communication:

```

1  #include <WiFi.h>
2
3  const char* ap_ssid = "ESP32_EdgeNode";
4  const char* ap_password = "edge12345"; // Min 8 characters
5
6  void setup() {
7    Serial.begin(115200);
8
9    // Create Access Point
10   WiFi.softAP(ap_ssid, ap_password);
11
12   Serial.println("Access Point created!");
13   Serial.print("SSID: ");
14   Serial.println(ap_ssid);
15   Serial.print("IP: ");
16   Serial.println(WiFi.softAPIP()); // Usually 192.168.4.1
17 }
18
19 void loop() {
20   Serial.print("Connected clients: ");
21   Serial.println(WiFi.softAPgetStationNum());
22   delay(5000);
23 }

```

When to Use AP Mode

Access Point mode is useful for:

- Initial device configuration (captive portal)
- Direct sensor-to-phone data transfer
- Mesh networks between multiple ESP32s
- Locations without existing WiFi infrastructure

Level 1: WiFi Simulation in Python

Simulate WiFi connection logic without hardware:

```

1 import time
2 import random
3
4 class WiFiSimulator:
5     def __init__(self, ssid, password):
6         self.ssid = ssid
7         self.connected = False
8         self.ip_address = None
9
10    def connect(self, timeout=10):
11        print(f"Connecting to {self.ssid}...")
12        for i in range(timeout):
13            time.sleep(0.5)
14            print(".", end="", flush=True)
15            # Simulate connection success/failure
16            if random.random() > 0.3: # 70% success rate
17                self.connected = True
18                self.ip_address = f"192.168.1.{random.randint(100, 200)}"
19                print(f"\nConnected! IP: {self.ip_address}")
20                return True
21        print("\nConnection failed!")
22        return False
23
24    def get_signal_strength(self):
25        return random.randint(-80, -30) # dBm
26
27 # Test the simulator
28 wifi = WiFiSimulator("TestNetwork", "password123")
29 wifi.connect()
30 print(f"Signal: {wifi.get_signal_strength()} dBm")

```

Level 2: Wokwi ESP32 Simulation

Test WiFi code using Wokwi online simulator:

1. Visit <https://wokwi.com/projects/new/esp32>
2. Paste ESP32 WiFi code into editor
3. Add diagram.json for virtual components
4. Run simulation and observe Serial output
5. Note: Wokwi has limited WiFi simulation—use for logic testing only

Level 3: Physical ESP32 Deployment

Deploy on real ESP32 hardware:

- Flash code via USB using Arduino IDE or PlatformIO
- Connect to actual WiFi network
- Monitor connection with Serial Monitor at 115200 baud
- Test reconnection logic by toggling router power

- Measure actual signal strength (RSSI) in different locations

9.5 HTTP Communication for Edge ML

A common pattern: read sensors, run ML inference, send results to a server via HTTP.

```

1 #include <WiFi.h>
2 #include <HTTPClient.h>
3
4 const char* ssid = "YourNetwork";
5 const char* password = "YourPassword";
6 const char* serverUrl = "http://api.example.com/inference";
7
8 void sendInferenceResult(float sensorValue, int classId,
9                          float confidence) {
10     if (WiFi.status() != WL_CONNECTED) {
11         Serial.println("WiFi not connected!");
12         return;
13     }
14
15     HTTPClient http;
16     http.begin(serverUrl);
17     http.addHeader("Content-Type", "application/json");

```

Code Walkthrough

HTTPClient: High-level HTTP client for ESP32.

http.begin(): Initializes connection to the server URL.

Content-Type header: Tells server we're sending JSON data.

```

1 // Build JSON payload
2 String payload = "{";
3 payload += "\"device_id\": \"esp32_001\", ";
4 payload += "\"sensor_value\": " + String(sensorValue, 2) + ", ";
5 payload += "\"class_id\": " + String(classId) + ", ";
6 payload += "\"confidence\": " + String(confidence, 3) + ", ";
7 payload += "\"timestamp\": " + String(millis());
8 payload += "}";
9
10 Serial.print("Sending: ");
11 Serial.println(payload);
12
13 int responseCode = http.POST(payload);
14
15 if (responseCode > 0) {
16     Serial.print("Response: ");
17     Serial.println(http.getString());
18 } else {
19     Serial.print("Error: ");
20     Serial.println(responseCode);
21 }
22
23 http.end();
24 }

```

Code Walkthrough

JSON payload: Structure your data for easy server-side parsing. Include device ID for multi-node

deployments.

http.POST(): Sends data and returns HTTP response code (200 = success).

http.end(): Closes connection and frees resources.

9.6 Method Selection Guide: Communication Protocols

Choosing the right wireless protocol is critical for edge ML deployments. Each protocol makes different trade-offs between range, power, bandwidth, and complexity.

See Also: LAB15 - Energy-Aware Edge Computing

Power optimization is critical for wireless edge devices. In LAB15, you'll learn comprehensive energy management techniques:

- Measure and profile power consumption of WiFi, BLE, and ML inference
- Implement advanced sleep modes and wake-on-demand strategies
- Design duty-cycled edge ML systems for months-long battery life
- Balance communication frequency vs. energy consumption

This lab introduces wireless communication; LAB15 shows you how to make it energy-efficient.

Choosing a Wireless Protocol for Edge ML

Use the following decision tree to select the best communication method:

1. **What is your power budget?**
 - Ultra-low power, battery lasts months → **LoRa** or **BLE**
 - Low power, battery lasts weeks → **BLE** or **WiFi with deep sleep**
 - Wall-powered or frequent charging → **WiFi** or **Ethernet**
2. **What is your communication range?**
 - <10 meters (same room) → **BLE**
 - 10-100 meters (building) → **WiFi**
 - 100m-10km (outdoor, rural) → **LoRa** or **cellular**
3. **What is your data rate requirement?**
 - <1KB/min (sensor readings) → **LoRa** or **BLE**
 - 1KB-100KB/min (periodic inference results) → **BLE** or **WiFi**
 - >100KB/min (images, audio, video) → **WiFi** or **Ethernet**
4. **Do you need internet connectivity?**
 - Yes, need cloud access → **WiFi**, **cellular**, or **LoRaWAN**
 - No, local mesh network → **BLE Mesh** or **ESP-NOW**
 - Phone/tablet as gateway → **BLE**

Table 9.4: Wireless Protocol Comparison for Edge ML

Protocol	Power	Range	Bandwidth	Best For
Short Range Protocols				
BLE 4.2/5.0	10-50mW 0.01mW sleep	10-50m (indoor)	1 Mbps (125KB/s)	Wearables, sensors Phone connectivity
BLE Mesh	10-50mW 0.01mW sleep	10m/hop (extends)	1 Mbps (shared)	Multi-hop sensor networks, smart home
ESP-NOW	100-200mW N/A	200m (outdoor)	1 Mbps (250KB/s)	Peer-to-peer ESP32 networks, no router
Medium Range Protocols				
WiFi 802.11n (2.4 GHz)	200-400mW 1-10mW idle	50-100m (indoor)	150 Mbps (real: 5-20)	High data rate ML Images, audio
WiFi with MQTT (burst mode)	200-400mW 10μA sleep	50-100m (indoor)	150 Mbps (efficient)	IoT sensor networks with cloud integration
Long Range Protocols				
LoRa/LoRaWAN	100-300mW 1μA sleep	2-15 km (outdoor)	0.3-50 kbps (50-300B/s)	Rural sensors Smart agriculture
NB-IoT/LTE-M	200-500mW 5μA sleep	10-35 km (cellular)	20-250 kbps (50-200B/s)	Mobile edge nodes with cellular network
Wired (For Comparison)				
Ethernet	1-2W N/A	100m (cable)	100 Mbps+ (12.5 MB/s)	Fixed installations High reliability
USB Serial	<100mW N/A	5m (cable)	12 Mbps (1.5 MB/s)	Development, debugging

Power Consumption Hierarchy

For a typical ESP32 edge ML node sending inference results:

- **BLE advertisement:** 0.1-0.5 mJ per message (10ms burst)
- **WiFi transmission:** 5-20 mJ per message (connection overhead)
- **LoRa transmission:** 1-5 mJ per message (depends on spreading factor)
- **ML inference:** 1-10 mJ per inference (depends on model size)

Critical insight: A single WiFi transmission can consume as much energy as 10-100 ML inferences! Always consider transmission costs in your energy budget.

💡 Optimizing Communication Energy

Your smart doorbell runs gesture recognition (10ms inference, 5mW) every 100ms, and sends results via WiFi (200mW for 50ms) every second.

Energy per second:

- Inference: 10 inferences $\times$ 10ms $\times$ 5mW = 0.5mJ
- WiFi idle: 900ms $\times$ 10mW = 9mJ
- WiFi transmission: 50ms $\times$ 200mW = 10mJ
- WiFi connection overhead: 5mJ
- Total: 24.5mJ per second

Alternative with BLE:

- Inference: 0.5mJ (same)

- BLE sleep: $950\text{ms} \times 0.01\text{mW} = 0.0095\text{mJ}$
- BLE advertisement: $50\text{ms} \times 10\text{mW} = 0.5\text{mJ}$
- Total: 1.0mJ per second (24× reduction!)

Battery life with 2000mAh at 3.7V (26.6 Wh):

- WiFi: $26.6\text{ Wh} / 24.5\text{ mJ} = 1,087,000\text{ seconds} = 12.6\text{ days}$
 - BLE: $26.6\text{ Wh} / 1.0\text{ mJ} = 26,600,000\text{ seconds} = 308\text{ days}$
- The protocol choice matters more than the ML optimization!

WiFi Connection Overhead

Don't reconnect to WiFi for every message! Each connection takes:

- 2-5 seconds to connect
- 200-500mW during connection
- 1-2.5 joules of energy (equivalent to 100-500 ML inferences!)

Better strategies:

1. Keep WiFi connected if sending data more than once per minute
2. Use deep sleep with periodic wake + batch transmission
3. Consider BLE for frequent small messages, WiFi only for firmware updates

9.7 MQTT for IoT Communication

MQTT (Message Queuing Telemetry Transport) is a lightweight publish/subscribe protocol designed for IoT.

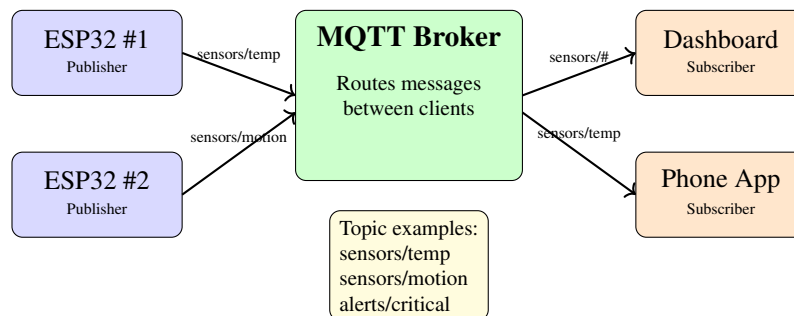


Figure 9.3: MQTT publish/subscribe model: Publishers send messages to topics. Subscribers receive messages from topics they're interested in. The broker routes messages without publishers knowing about subscribers.

MQTT vs HTTP for Edge ML

- **MQTT:** Lower overhead, persistent connection, bi-directional, ideal for frequent small messages
 - **HTTP:** Simpler, widely supported, better for large payloads, request/response pattern
- Use MQTT for real-time sensor data; HTTP for model updates or bulk data.

🔗 Choosing the Right IoT Protocol

Use this guide to select the appropriate communication protocol for your edge application:

How often does your device send data?

- Rarely (<1 message/minute) → **HTTP REST**
 - Simple, stateless, no broker needed
 - Higher latency but simpler infrastructure
- Frequently (1+ messages/second) → **MQTT**
 - Persistent connection reduces overhead
 - Ideal for sensor streaming
- Ultra-constrained devices (8KB RAM) → **CoAP**
 - UDP-based, smallest footprint
 - Similar semantics to HTTP

Do you need real-time control (server → device)?

- Yes → **MQTT** (subscribe to control topics)
- No (device just reports) → HTTP may suffice

Do you need guaranteed delivery?

- Best effort is OK → MQTT QoS 0 or HTTP
- At-least-once delivery → MQTT QoS 1
- Exactly-once delivery → MQTT QoS 2 (higher overhead)

⚠ Blocking on Network Calls

Never block the main loop waiting for network responses! If WiFi is slow or the server is unresponsive, your sensor readings will be delayed or lost. Use non-blocking patterns: check connection status, send asynchronously, and handle timeouts gracefully.

9.7.1 MQTT Client Implementation

```

1 #include <WiFi.h>
2 #include <PubSubClient.h>
3
4 const char* ssid = "YourNetwork";
5 const char* password = "YourPassword";
6 const char* mqttServer = "broker.hivemq.com";
7 const int mqttPort = 1883;
8
9 WiFiClient espClient;
10 PubSubClient mqtt(espClient);

```

Code Walkthrough

PubSubClient: Arduino MQTT client library. Install via Library Manager.

broker.hivemq.com: Free public MQTT broker for testing. Use your own broker in production.

```

1 // Callback for incoming messages
2 void mqttCallback(char* topic, byte* payload, unsigned int length) {
3     Serial.print("Message on ");
4     Serial.print(topic);
5     Serial.print(": ");
6

```

```

7   String message = "";
8   for (int i = 0; i < length; i++) {
9       message += (char)payload[i];
10  }
11  Serial.println(message);
12
13  // Handle commands
14  if (String(topic) == "edge/commands") {
15      if (message == "restart") {
16          ESP.restart();
17      } else if (message == "status") {
18          mqtt.publish("edge/status", "online");
19      }
20  }
21 }

```

Code Walkthrough

Callback function: Called automatically when messages arrive on subscribed topics.

Command handling: Use subscribed topics to receive commands from a central server or dashboard.

```

1  void reconnectMQTT() {
2      while (!mqtt.connected()) {
3          Serial.print("Connecting to MQTT...");
4          String clientId = "ESP32_" + String(random(0xffff), HEX);
5
6          if (mqtt.connect(clientId.c_str())) {
7              Serial.println("connected!");
8              mqtt.subscribe("edge/commands");
9          } else {
10             Serial.print("failed , rc=");
11             Serial.println(mqtt.state());
12             delay(5000);
13         }
14     }
15 }
16
17 void setup() {
18     Serial.begin(115200);
19
20     // Connect WiFi
21     WiFi.begin(ssid, password);
22     while (WiFi.status() != WL_CONNECTED) {
23         delay(500);
24     }
25
26     // Configure MQTT
27     mqtt.setServer(mqttServer, mqttPort);
28     mqtt.setCallback(mqttCallback);
29 }
30
31 void loop() {
32     if (!mqtt.connected()) {
33         reconnectMQTT();
34     }
35     mqtt.loop(); // Process incoming messages
36
37     // Publish sensor data every 5 seconds
38     static unsigned long lastPublish = 0;
39     if (millis() - lastPublish > 5000) {

```

```

40     float temperature = 22.5 + random(50) / 10.0;
41
42     String payload = "{"temp\":" + String(temperature, 1) +
43                     ", \"rssi\":" + String(WiFi.RSSI()) + "}";
44
45     mqtt.publish("edge/sensors/temp", payload.c_str());
46     Serial.println("Published: " + payload);
47
48     lastPublish = millis();
49 }
50 }

```

9.8 Bluetooth Low Energy (BLE)

BLE is ideal for short-range, low-power communication with smartphones and wearables.

```

1 #include <BLEDevice.h>
2 #include <BLEServer.h>
3 #include <BLEUtils.h>
4 #include <BLE2902.h>
5
6 #define SERVICE_UUID           "12345678-1234-1234-1234-123456789abc"
7 #define CHARACTERISTIC_UUID   "87654321-4321-4321-4321-cba987654321"
8
9 BLEServer* pServer = NULL;
10 BLECharacteristic* pCharacteristic = NULL;
11 bool deviceConnected = false;

```

Code Walkthrough

UUIDs: Unique identifiers for your BLE service and characteristics. Generate custom ones for your application.

Service: Groups related characteristics (e.g., all sensor readings).

Characteristic: Individual data point that can be read, written, or notified.

```

1 class ServerCallbacks: public BLEServerCallbacks {
2     void onConnect(BLEServer* pServer) {
3         deviceConnected = true;
4         Serial.println("Client connected!");
5     }
6
7     void onDisconnect(BLEServer* pServer) {
8         deviceConnected = false;
9         Serial.println("Client disconnected");
10        pServer->startAdvertising(); // Restart advertising
11    }
12 };
13
14 void setup() {
15     Serial.begin(115200);
16
17     BLEDevice::init("ESP32_Sensor");
18     pServer = BLEDevice::createServer();
19     pServer->setCallbacks(new ServerCallbacks());
20
21     BLEService *pService = pServer->createService(SERVICE_UUID);
22     pCharacteristic = pService->createCharacteristic(
23         CHARACTERISTIC_UUID,
24         BLECharacteristic::PROPERTY_READ |

```

```

25     BLECharacteristic::PROPERTY_NOTIFY
26 );
27 pCharacteristic->addDescriptor(new BLE2902());
28
29 pService->start();
30 BLEDevice::getAdvertising()->start();
31 Serial.println("BLE advertising started!");
32 }
33
34 void loop() {
35     if (deviceConnected) {
36         float temp = 22.0 + random(100) / 10.0;
37         String value = String(temp, 1) + "C";
38
39         pCharacteristic->setValue(value.c_str());
40         pCharacteristic->notify();
41
42         Serial.println("Sent: " + value);
43     }
44     delay(1000);
45 }

```

Code Walkthrough

This complete BLE example creates a peripheral device that advertises itself as “ESP32_Sensor” and sends temperature readings to connected clients. The `notify()` method pushes data to clients without them needing to poll.

Level 1: BLE Simulation in Python

Simulate BLE peripheral behavior for algorithm testing:

```

1 import time
2 import random
3 import uuid
4
5 class BLEPeripheral:
6     def __init__(self, device_name, service_uuid):
7         self.device_name = device_name
8         self.service_uuid = service_uuid
9         self.connected = False
10        self.characteristics = {}
11
12    def add_characteristic(self, char_uuid, value):
13        self.characteristics[char_uuid] = value
14
15    def notify(self, char_uuid, value):
16        if self.connected:
17            print(f"[BLE] Notifying {char_uuid}: {value}")
18            self.characteristics[char_uuid] = value
19        else:
20            print(f"[BLE] No client connected")
21
22    def start_advertising(self):
23        print(f"[BLE] Advertising as '{self.device_name}'")
24        print(f"[BLE] Service UUID: {self.service_uuid}")
25
26 # Simulate BLE sensor beacon

```

```

27 ble = BLEPeripheral("ESP32_Sensor", str(uuid.uuid4()))
28 ble.add_characteristic("temperature", 0.0)
29 ble.start_advertising()
30
31 # Simulate connection and data transmission
32 time.sleep(1)
33 ble.connected = True
34 for i in range(5):
35     temp = 20.0 + random.uniform(-2, 2)
36     ble.notify("temperature", f"{temp:.1f}C")
37     time.sleep(2)

```

Level 2: BLE Testing with Mobile Apps

Test BLE functionality using smartphone apps:

1. Install a BLE scanner app (nRF Connect, LightBlue, BLE Scanner)
2. Flash BLE code to ESP32 via USB
3. Scan for devices and connect to “ESP32_Sensor”
4. Observe characteristic values updating in real-time
5. Test notifications and write operations

Note: Wokwi does not currently support BLE simulation—use physical hardware for this tier.

Level 3: Production BLE Beacons

Deploy battery-powered BLE sensor beacons:

- Combine BLE with deep sleep for extended battery life
- Use iBeacon or Eddystone format for compatibility
- Implement secure pairing for sensitive data
- Add multiple characteristics for multi-sensor nodes
- Test range and reliability in target deployment environment

9.9 Power Management for Battery Operation

Battery-powered edge devices must minimize power consumption. ESP32 offers multiple sleep modes:

Table 9.5: ESP32 power consumption by mode

Mode	Current	Wake Sources	Use Case
Active (WiFi TX)	160-260 mA	N/A	Data transmission
Active (CPU only)	20-68 mA	N/A	ML inference
Light Sleep	0.8 mA	GPIO, Timer, WiFi	Periodic sampling
Deep Sleep	10 μ A	GPIO, Timer, Touch	Long intervals
Hibernation	5 μ A	RTC Timer only	Very long intervals

9.9.1 Deep Sleep Implementation

```

1 #include <WiFi.h>
2
3 #define WAKEUP_PIN GPIO_NUM_33
4 #define SLEEP_DURATION 60000000 // 60 seconds in microseconds
5
6 // Data preserved across deep sleep
7 RTC_DATA_ATTR int bootCount = 0;
8 RTC_DATA_ATTR float lastReading = 0;

```

Code Walkthrough

RTC_DATA_ATTR: Variables stored in RTC memory survive deep sleep. Regular variables are lost when the ESP32 resets.

SLEEP_DURATION: Specified in microseconds (1 second = 1,000,000 μ s).

```

1 void print_wakeup_reason() {
2     esp_sleep_wakeup_cause_t reason = esp_sleep_get_wakeup_cause();
3
4     switch(reason) {
5         case ESP_SLEEP_WAKEUP_TIMER:
6             Serial.println("Woke up from timer");
7             break;
8         case ESP_SLEEP_WAKEUP_EXT0:
9             Serial.println("Woke up from external GPIO");
10            break;
11            default:
12                Serial.printf("Woke up for other reason: %d\n", reason);
13        }
14    }
15
16 void setup() {
17     Serial.begin(115200);
18     delay(100);
19
20     bootCount++;
21     Serial.printf("Boot count: %d\n", bootCount);
22     print_wakeup_reason();
23
24     // Quick sensor read
25     float sensorValue = analogRead(34) * 3.3 / 4095.0;
26     Serial.printf("Sensor: %.2f V (prev: %.2f V)\n",
27                 sensorValue, lastReading);
28     lastReading = sensorValue;
29
30     // Only connect WiFi if reading changed significantly
31     if (abs(sensorValue - lastReading) > 0.5) {
32         // Connect and send data
33         // ...
34     }
35
36     // Configure wake sources
37     esp_sleep_enable_timer_wakeup(SLEEP_DURATION);
38     esp_sleep_enable_ext0_wakeup(WAKEUP_PIN, 1);
39
40     Serial.println("Entering deep sleep...");
41     Serial.flush();
42     esp_deep_sleep_start();
43     // Code never reaches here - ESP32 resets on wakeup
44 }
45

```

```

46 void loop() {
47     // Never executed in deep sleep mode
48 }

```

Battery Life Calculation

With a 2000 mAh battery:

- **Always active (WiFi):** 2000 mAh / 200 mA = 10 hours
- **Deep sleep with 1-min wakes:** Wake takes 5 seconds at 200 mA, sleep at 10 μ A
- **Average:** $(5 \times 200 + 55 \times 0.01) / 60 \approx 17$ mA $\rightarrow$ 5 days
- **Deep sleep only:** 2000 mAh / 0.01 mA = 200,000 hours (22 years theoretical)

Level 1: Power Consumption Simulation

Model battery life for different sleep strategies:

```

1 def calculate_battery_life(battery_mah, sleep_current_ma,
2                             active_current_ma, wake_duration_s,
3                             sleep_duration_s):
4     """Calculate battery life for sleep/wake cycles"""
5     cycle_time = wake_duration_s + sleep_duration_s
6     cycles_per_hour = 3600 / cycle_time
7
8     # Energy per cycle (mAh)
9     wake_energy = (active_current_ma * wake_duration_s) / 3600
10    sleep_energy = (sleep_current_ma * sleep_duration_s) / 3600
11    cycle_energy = wake_energy + sleep_energy
12
13    # Average current
14    avg_current = (cycle_energy / cycle_time) * 3600
15
16    # Battery life in hours
17    battery_life_hours = battery_mah / avg_current
18
19    return {
20        'avg_current_ma': avg_current,
21        'battery_life_hours': battery_life_hours,
22        'battery_life_days': battery_life_hours / 24
23    }
24
25 # Example: Wake for 5s every 60s
26 result = calculate_battery_life(
27     battery_mah=2000,
28     sleep_current_ma=0.01,      # Deep sleep
29     active_current_ma=200,      # WiFi active
30     wake_duration_s=5,
31     sleep_duration_s=55
32 )
33
34 print(f"Average current: {result['avg_current_ma']:.2f} mA")
35 print(f"Battery life: {result['battery_life_days']:.1f} days")

```

Level 2: Deep Sleep Testing via USB

Test and debug deep sleep on USB-connected ESP32:

1. Flash deep sleep code to ESP32
2. Monitor Serial output to see wake reasons
3. Use multimeter in series to measure current (disconnect USB power, use external supply)
4. Verify RTC memory variables persist across sleep cycles
5. Test wake-up sources (timer, GPIO, touch)

Level 3: Battery-Powered Deployment

Deploy autonomous battery-powered sensor nodes:

- Use LiPo battery with charging circuit (TP4056)
- Add voltage divider to monitor battery level
- Implement adaptive sleep duration based on battery voltage
- Use solar panel for outdoor deployments
- Log wake count and battery voltage to track performance over weeks

9.10 Over-the-Air (OTA) Firmware Updates

OTA updates allow you to upload new firmware wirelessly—critical for deployed devices that are difficult to access physically.

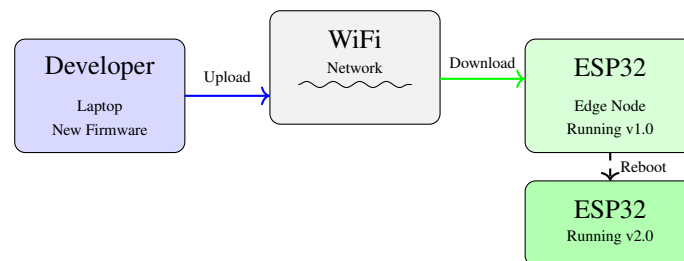


Figure 9.4: OTA update process: Developer uploads new firmware over WiFi. ESP32 downloads, validates, and installs the update, then reboots into the new version.

9.10.1 Basic OTA Implementation

```

1 #include <WiFi.h>
2 #include <ArduinoOTA.h>
3
4 const char* ssid = "YourNetwork";
5 const char* password = "YourPassword";
6
7 void setup() {
8     Serial.begin(115200);
9
10    // Connect to WiFi
11    WiFi.begin(ssid, password);
12    while (WiFi.status() != WL_CONNECTED) {
13        delay(500);

```



```

14     Serial.print(".");
15 }
16 Serial.println("\nWiFi connected");
17 Serial.print("IP: ");
18 Serial.println(WiFi.localIP());

```

Code Walkthrough

ArduinoOTA library: Built into ESP32 Arduino core, handles OTA protocol.

WiFi.localIP(): Print the IP address—you'll need this to upload firmware.

```

1  // Configure OTA
2  ArduinoOTA.setHostname("esp32-sensor-01");
3  ArduinoOTA.setPassword("update123"); // Optional but recommended
4
5  ArduinoOTA.onStart([]() {
6      String type = (ArduinoOTA.getCommand() == U_FLASH)
7                    ? "sketch" : "filesystem";
8      Serial.println("Start updating " + type);
9  });
10
11 ArduinoOTA.onEnd([]() {
12     Serial.println("\nUpdate complete!");
13 });
14
15 ArduinoOTA.onProgress([](unsigned int progress, unsigned int total) {
16     Serial.printf("Progress: %u%%\r", (progress / (total / 100)));
17 });
18
19 ArduinoOTA.onError([](ota_error_t error) {
20     Serial.printf("Error[%u]: ", error);
21     if (error == OTA_AUTH_ERROR) Serial.println("Auth Failed");
22     else if (error == OTA_BEGIN_ERROR) Serial.println("Begin Failed");
23     else if (error == OTA_CONNECT_ERROR) Serial.println("Connect Failed");
24     else if (error == OTA_RECEIVE_ERROR) Serial.println("Receive Failed");
25     else if (error == OTA_END_ERROR) Serial.println("End Failed");
26 });
27
28 ArduinoOTA.begin();
29 Serial.println("OTA ready!");
30 }
31
32 void loop() {
33     ArduinoOTA.handle(); // Must be called regularly
34
35     // Your application code here
36     delay(1000);
37 }

```

Code Walkthrough

Callbacks: Monitor update progress and handle errors.

ArduinoOTA.handle(): Checks for incoming OTA requests. Call this frequently in loop().

Password protection: Prevents unauthorized firmware uploads.

9.10.2 Uploading Firmware via OTA

Once your ESP32 is running OTA-enabled code, you can upload new firmware:

Method 1: Arduino IDE

1. Ensure ESP32 and laptop are on the same network

2. In Arduino IDE: Tools → Port → Select “esp32-sensor-01 at 192.168.x.x”
3. Click Upload normally—Arduino IDE automatically uses OTA
4. Monitor Serial output for progress

Method 2: PlatformIO

```
# platformio.ini
[env:esp32]
platform = espressif32
board = esp32dev
framework = arduino
upload_protocol = espota
upload_port = 192.168.1.105
upload_flags = --auth=update123
```

Then upload with: `pio run -t upload`

OTA Safety Considerations

- **Don’t disable serial uploads:** Always keep USB upload working as a backup
- **Include OTA in all builds:** If you upload non-OTA firmware via OTA, you lose OTA capability
- **Test updates locally first:** Bad firmware can brick a device remotely
- **Implement rollback:** Advanced OTA can revert to previous firmware if new version fails

9.10.3 Advanced OTA: Partition Scheme and Rollback

ESP32 can use dual partitions for safe updates:

```
1 #include <Update.h>
2 #include <esp_ota_ops.h>
3
4 void checkFirmwareVersion() {
5     const esp_partition_t* running = esp_ota_get_running_partition();
6     esp_ota_img_states_t ota_state;
7
8     if (esp_ota_get_state_partition(running, &ota_state) == ESP_OK) {
9         if (ota_state == ESP_OTA_IMG_PENDING_VERIFY) {
10             Serial.println("New firmware running, validating...");
11
12             // Run self-tests here
13             bool tests_passed = true; // Your validation logic
14
15             if (tests_passed) {
16                 esp_ota_mark_app_valid_cancel_rollback();
17                 Serial.println("Firmware validated!");
18             } else {
19                 Serial.println("Tests failed, rolling back...");
20                 esp_ota_mark_app_invalid_rollback_and_reboot();
21             }
22         }
23     }
24 }
```

OTA Best Practices for Edge ML

- **Version tracking:** Include firmware version in MQTT messages
- **Scheduled updates:** Update during low-traffic periods
- **Batch updates:** Update a subset of devices first, monitor for issues
- **Model updates:** OTA can update ML models stored in SPIFFS/LittleFS
- **Cellular OTA:** For remote deployments, consider OTA over cellular (ESP32 with SIM module)

Level 1: OTA Simulation

Simulate OTA update logic in Python:

```

1 import hashlib
2 import time
3
4 class FirmwareManager:
5     def __init__(self, current_version):
6         self.current_version = current_version
7         self.pending_update = None
8
9     def check_for_update(self, server_version):
10        if server_version > self.current_version:
11            print(f"Update available: v{server_version}")
12            return True
13        return False
14
15    def download_firmware(self, firmware_data):
16        print("Downloading firmware ...")
17        # Simulate download with progress
18        for i in range(0, 101, 10):
19            time.sleep(0.1)
20            print(f"Progress: {i}%")
21
22        # Verify checksum
23        checksum = hashlib.md5(firmware_data.encode()).hexdigest()
24        print(f"Checksum: {checksum}")
25        self.pending_update = firmware_data
26        return True
27
28    def apply_update(self):
29        if self.pending_update:
30            print("Applying update ...")
31            time.sleep(0.5)
32            self.current_version += 1
33            print(f"Update complete! Now running v{self.current_version}")
34            self.pending_update = None
35
36 # Test OTA simulation
37 mgr = FirmwareManager(current_version=1)
38 if mgr.check_for_update(server_version=2):
39     mgr.download_firmware("new_firmware_binary_data_here")
40     mgr.apply_update()

```

Level 2: Local Network OTA Testing

Test OTA on USB-connected ESP32:

1. Flash initial OTA-enabled firmware via USB
2. Note the ESP32's IP address from Serial Monitor
3. Make a small code change (e.g., change LED blink rate)
4. Upload via OTA from Arduino IDE
5. Observe Serial output showing update progress
6. Verify new code is running after automatic reboot

Level 3: Production OTA Infrastructure

Deploy OTA for field devices:

- Set up firmware server (HTTP or MQTT-based)
- Implement version checking on device boot
- Use signed firmware images to prevent tampering
- Configure watchdog timer to recover from bad updates
- Log update success/failure to cloud for monitoring
- Test OTA over production network (consider bandwidth, latency)

9.11 Visual Feedback with Neopixels

Neopixel (WS2812B) LEDs provide RGB feedback for ML classification results:

```

1 #include <Adafruit_NeoPixel.h>
2
3 #define LED_PIN    5
4 #define LED_COUNT  8
5
6 Adafruit_NeoPixel strip(LED_COUNT, LED_PIN, NEO_GRB + NEO_KHZ800);
7
8 void showClassification(int classId, float confidence) {
9     uint32_t colors[] = {
10         strip.Color(0, 255, 0),    // Class 0: Green (normal)
11         strip.Color(255, 255, 0),  // Class 1: Yellow (warning)
12         strip.Color(255, 0, 0)     // Class 2: Red (alert)
13     };
14
15     // Number of LEDs lit = confidence level
16     int numLit = (int)(confidence * LED_COUNT);
17
18     for(int i = 0; i < LED_COUNT; i++) {
19         if (i < numLit) {
20             strip.setPixelColor(i, colors[classId]);
21         } else {
22             strip.setPixelColor(i, 0);
23         }
24     }
25     strip.show();
26 }
27
28 void setup() {
29     strip.begin();
30     strip.setBrightness(50);
31     strip.show();
32 }
```

```

33
34 void loop() {
35     // Simulate ML inference
36     int classId = random(3);
37     float confidence = 0.5 + random(50) / 100.0;
38
39     showClassification(classId, confidence);
40     delay(2000);
41 }

```

Code Walkthrough

Color mapping: Map each ML class to a distinct color for intuitive feedback.

Confidence visualization: Number of lit LEDs indicates prediction confidence.

setBrightness(): Control power consumption and visual comfort (0-255).

9.12 Complete Edge ML Node

This example combines all concepts: WiFi, MQTT, sensors, ML inference, and visual feedback.

```

1 #include <WiFi.h>
2 #include <PubSubClient.h>
3 #include <Adafruit_NeoPixel.h>
4
5 // Configuration
6 const char* ssid = "YourNetwork";
7 const char* password = "YourPassword";
8 const char* mqttServer = "broker.hivemq.com";
9
10 #define LED_PIN 5
11 #define LED_COUNT 8
12 #define SENSOR_PIN 34
13
14 WiFiClient espClient;
15 PubSubClient mqtt(espClient);
16 Adafruit_NeoPixel strip(LED_COUNT, LED_PIN, NEO_GRB + NEO_KHZ800);
17
18 // Simple threshold classifier (replace with TFLite)
19 int classify(float voltage) {
20     if (voltage < 1.0) return 0; // Low
21     else if (voltage < 2.5) return 1; // Normal
22     else return 2; // High
23 }
24
25 void showClass(int c) {
26     uint32_t colors[] = {
27         strip.Color(0, 255, 0),
28         strip.Color(0, 0, 255),
29         strip.Color(255, 0, 0)
30     };
31     for(int i = 0; i < LED_COUNT; i++) {
32         strip.setPixelColor(i, colors[c]);
33     }
34     strip.show();
35 }
36
37 void setup() {
38     Serial.begin(115200);
39
40     strip.begin();

```

```

41     strip.setBrightness(50);
42
43     WiFi.begin(ssid, password);
44     while (WiFi.status() != WL_CONNECTED) delay(500);
45
46     mqtt.setServer(mqttServer, 1883);
47 }
48
49 void loop() {
50     if (!mqtt.connected()) {
51         mqtt.connect(("ESP32_" + String(random(0xffff), HEX)).c_str());
52     }
53     mqtt.loop();
54
55     // Read and classify
56     float voltage = analogRead(SENSOR_PIN) * 3.3 / 4095.0;
57     int classId = classify(voltage);
58
59     // Visual feedback
60     showClass(classId);
61
62     // Publish result
63     String payload = "{ \"v\": " + String(voltage, 2) +
64                     " , \"c\": " + String(classId) + " }";
65     mqtt.publish("edge/data", payload.c_str());
66
67     delay(1000);
68 }

```

Level 1: MQTT Simulation in Python

Test MQTT publish/subscribe patterns without ESP32 hardware:

```

1 import time
2 import random
3 import json
4 import paho.mqtt.client as mqtt
5
6 # Simulate sensor and classification
7 def read_sensor():
8     return random.uniform(0, 3.3)
9
10 def classify(voltage):
11     if voltage < 1.0: return 0
12     elif voltage < 2.5: return 1
13     else: return 2
14
15 # Callback for incoming messages
16 def on_message(client, userdata, msg):
17     print(f"Received on {msg.topic}: {msg.payload.decode()}")
18     if msg.topic == "edge/commands":
19         command = msg.payload.decode()
20         if command == "status":
21             client.publish("edge/status", "online")
22
23 # MQTT client setup
24 client = mqtt.Client()
25 client.on_message = on_message
26 client.connect("broker.hivemq.com", 1883)

```

```

27 client.subscribe("edge/commands")
28 client.loop_start()
29
30 # Publish sensor data
31 try:
32     while True:
33         voltage = read_sensor()
34         class_id = classify(voltage)
35         payload = json.dumps({
36             "voltage": round(voltage, 2),
37             "class": class_id,
38             "timestamp": int(time.time())
39         })
40         client.publish("edge/sim/data", payload)
41         print(f"Published: {payload}")
42         time.sleep(2)
43 except KeyboardInterrupt:
44     client.loop_stop()

```

Level 2: ESP32 via USB with MQTT Broker

Program and test ESP32 via USB with real MQTT communication:

1. Set up local MQTT broker (Mosquitto) or use public broker
2. Flash MQTT code to ESP32 using Arduino IDE
3. Monitor Serial output for connection status and published messages
4. Use MQTT Explorer or mosquitto_sub to view published data
5. Test command/control by publishing to subscribed topics
6. Debug connection issues (firewall, credentials, network)

Level 3: Production MQTT Deployment

Deploy wireless MQTT sensor nodes in production:

- Use private MQTT broker with authentication (username/password)
- Enable TLS/SSL encryption for secure communication
- Implement MQTT Last Will and Testament for offline detection
- Use QoS 1 or 2 for critical messages
- Add automatic reconnection with exponential backoff
- Monitor message latency and broker load

9.13 Troubleshooting Guide

⚠ ESP32 Won't Connect to WiFi

Symptoms: `WiFi.status()` returns `WL_DISCONNECTED` or connection times out

Cause: Wrong SSID/password, WiFi band incompatibility, weak signal, or router settings

Solution:

1. Verify credentials: Double-check SSID (case-sensitive) and password
2. Use 2.4GHz network: ESP32 doesn't support 5GHz WiFi

3. Move closer to router: Check signal strength with `WiFi.RSSI()` (>-70 dBm is good)
4. Check router: Ensure DHCP enabled, no MAC address filtering
5. Add timeout: `WiFi.begin(ssid, password); while(WiFi.status() != WL_CONNECTED && timeout<20)`
6. Print debug info: `Serial.println(WiFi.status());` to see specific error code

⚠ HTTP POST Request Fails or Times Out

Symptoms: `http.POST()` returns -1 or negative error code, server doesn't receive data

Cause: Wrong URL, server down, firewall blocking, or incorrect content type

Solution:

1. Test endpoint in browser: Verify server URL is accessible
2. Check HTTP vs HTTPS: Use `HTTPClient` for `http://`, `WiFiClientSecure` for `https://`
3. Set content type: `http.addHeader("Content-Type", "application/json");`
4. Increase timeout: `http.setTimeout(10000);` for 10-second timeout
5. Print response code: `int code = http.POST(payload); Serial.println(code);`
6. Verify JSON format: Use online validator to check payload syntax

⚠ ESP32 Keeps Crashing or Rebooting

Symptoms: Device resets every few seconds/minutes, Brownout detector or Guru Meditation Error

Cause: Insufficient power supply, memory overflow, watchdog timer, or stack overflow

Solution:

1. Use adequate power: 500mA+ supply via USB or external 5V (WiFi transmission draws 240mA)
2. Check memory usage: Print free heap with `ESP.getFreeHeap()`
3. Disable watchdog for testing: `disableCore0WDT();` (not for production!)
4. Reduce WiFi power: `WiFi.setTxPower(WIFI_POWER_11dBm);` to reduce current draw
5. Add delays: Insert `delay(10)` in tight loops to allow task switching
6. Check stack size: Increase if using deep recursion or large local arrays

⚠ MQTT Connection Refused or Disconnects

Symptoms: `client.connect()` returns false, or connection drops after few seconds

Cause: Broker unavailable, wrong credentials, client ID conflict, or keep-alive timeout

Solution:

1. Test broker: Use MQTT Explorer or `mosquitto_sub` to verify broker running
2. Use unique client ID: `String clientId = "ESP32_" + String(ESP.getEfuseMac());`
3. Check credentials: Verify username/password if broker requires authentication
4. Set keep-alive: `client.setKeepAlive(60);` for 60-second heartbeat
5. Handle reconnection: Implement `void reconnect()` function in loop
6. Check QoS level: Start with QoS 0 for testing, use QoS 1/2 for critical messages

⚠ BLE Device Not Discoverable or Won't Pair

Symptoms: Phone/computer can't find ESP32 BLE device, or pairing fails

Cause: BLE not started, wrong advertising parameters, name too long, or security mismatch

Solution:

1. Initialize BLE properly: `BLEDevice::init("ESP32_Device");` `BLEServer *pServer = BLEDevice::createServer();`
2. Start advertising: `pServer->getAdvertising()->start();`
3. Keep device name short: Max 20 characters for reliable compatibility
4. Check scanning app: Use nRF Connect (Android/iOS) for testing
5. Disable WiFi temporarily: BLE and WiFi share antenna - test one at a time
6. Set security: Match pairing requirements on both sides (No security, PIN, or bonding)

9.14 Summary

In this chapter, you learned:

- **ESP32 architecture:** Dual-core processor with integrated WiFi and BLE ideal for edge ML
- **WiFi connectivity:** Station mode for network access, AP mode for direct connections
- **HTTP communication:** Sending inference results to cloud servers via REST APIs
- **MQTT protocol:** Lightweight publish/subscribe for real-time IoT sensor data streaming
- **BLE peripheral mode:** Low-power short-range communication for sensor beacons
- **Power management:** Deep sleep modes reducing consumption to 10 μA for battery operation
- **OTA firmware updates:** Wireless firmware deployment for remote edge devices
- **Visual feedback:** Neopixel LEDs for intuitive ML classification display

Edge ML Communication Stack

This lab completes your edge ML communication toolkit:

- **Local:** BLE for phone/wearable integration (<50m range)
- **Building:** WiFi + MQTT for real-time sensor networks (<100m range)
- **Cloud:** WiFi + HTTP for analytics and model updates (unlimited range)
- **Low-power:** Deep sleep + wake-on-event for years of battery life

Combined with on-device ML inference (LAB05), you can now build complete autonomous edge intelligence systems.

Self-Assessment Questions

Test your understanding before continuing:

1. Why does WiFi consume significantly more power than BLE? When would you choose each protocol for an edge ML sensor node?
2. An ESP32 in deep sleep draws 10 μA , and wakes for 2 seconds every 5 minutes drawing 80 mA. With a 1000 mAh battery, calculate the expected battery life. Show your work.
3. Explain the difference between MQTT QoS levels 0, 1, and 2. Which QoS would you use for sending ML inference results where occasional message loss is acceptable?
4. What are the security risks of OTA firmware updates? How would you protect an ESP32 from malicious firmware being uploaded over WiFi?
5. Compare HTTP and MQTT for sending sensor data to the cloud. What are the trade-offs

in terms of latency, bandwidth, and power consumption?

9.15 Guided Experiments

Experiment 1: WiFi vs BLE Power Consumption

Hypothesis: BLE consumes significantly less power than WiFi for periodic sensor data transmission. The power difference increases with longer sleep intervals.

Procedure:

- Set up ESP32 with current measurement (multimeter in series or power monitor)
- Create two firmware versions:
 - Version A: WiFi + MQTT transmission
 - Version B: BLE advertisement
- For each version, measure current during:
 - Active transmission (10 seconds)
 - Light sleep (10 seconds)
 - Deep sleep (50 seconds)
- Calculate average power and battery life with 1000mAh battery

Data Collection:

Mode	WiFi Current (mA)	BLE Current (mA)	Duration (s)	Energy (mJ)
Active TX			10	
Light Sleep			10	
Deep Sleep			50	
Avg Power				
Battery Life (days)				

Analysis Questions:

- What is the power consumption ratio between WiFi and BLE during transmission?
- For a sensor that transmits every 5 minutes, which protocol provides longer battery life?
- At what transmission frequency does the sleep power dominate total consumption?
- How does WiFi connection overhead affect power compared to BLE?

Experiment 2: Deep Sleep Timing Optimization

Hypothesis: Longer sleep intervals reduce average power but increase data latency. There exists an optimal sleep duration for each application requirement.

Procedure:

- Program ESP32 to wake, read sensor, transmit via MQTT, then deep sleep
- Test different sleep intervals: 10s, 30s, 60s, 5min, 10min
- For each interval, measure:
 - Average current over 1 hour
 - Wake time (from trigger to sleep)
 - Data freshness (max time between readings)
- Calculate expected battery life for each configuration

Data Collection:

Sleep Interval	Avg Current (mA)	Wake Time (ms)	Data Latency (s)	Battery Life (days)
10s				
30s				
60s				
5min				
10min				

Analysis Questions:

- At what sleep interval does battery life stop improving significantly?
- What percentage of time is spent in active vs sleep modes for each interval?
- For temperature monitoring (slow-changing data), which interval is optimal?
- How would you implement event-driven wake-up for critical sensor thresholds?

 Related Labs
Prerequisites:

- **LAB08 - Arduino Sensor Programming:** Foundational understanding of sensor interfaces and data collection, which forms the basis for wireless sensor nodes.

Next Steps:

- **LAB13 - Distributed Edge ML Systems:** Build on wireless communication to create coordinated multi-node edge intelligence systems with distributed inference and data sharing.

9.16 Exercises**[Basic] Basic: Exercise 1: WiFi Signal Monitor**

Create an ESP32 sketch that:

1. Scans for available WiFi networks
2. Connects to the strongest one from a list of known networks
3. Publishes RSSI values to MQTT every 10 seconds
4. Lights Neopixels based on signal strength (green=strong, yellow=medium, red=weak)

[Intermediate] Intermediate: Exercise 2: Power-Optimized Sensor Node

Design a battery-powered sensor node that:

1. Wakes from deep sleep every 5 minutes
2. Reads a temperature sensor
3. Only connects to WiFi if temperature exceeds a threshold
4. Goes back to sleep immediately otherwise
5. Calculate expected battery life with a 1000 mAh battery

[Intermediate] Intermediate: Exercise 3: BLE Sensor with Phone App

1. Create a BLE service with temperature and humidity characteristics
2. Use a BLE scanner app on your phone to connect and read values
3. Add write capability to set sampling rate from the phone
4. Implement notifications for threshold alerts

[Intermediate] Intermediate: Exercise 4: Multi-Node MQTT System

Set up two ESP32 nodes:

1. Node 1: Publishes sensor data to “sensors/node1”
2. Node 2: Subscribes to “sensors/node1” and displays status on Neopixels
3. Add a third “command” topic for controlling both nodes
4. Test message latency and reliability

[Challenge] Challenge: Exercise 5: OTA-Enabled ML Node

Create a complete edge ML node with OTA capability:

1. Combine sensor reading, ML inference, and MQTT publishing
2. Add OTA update capability with password protection
3. Include firmware version number in MQTT messages
4. Test updating the ML classification thresholds via OTA
5. Implement a watchdog timer to recover from crashes

[Challenge] Challenge: Exercise 6: Battery-Optimized Wireless Sensor

Design an ultra-low-power wireless sensor node:

1. Use deep sleep with 5-minute wake intervals
2. Only enable WiFi if sensor reading exceeds threshold
3. Publish alerts via MQTT when anomaly detected
4. Use RTC memory to count wake cycles and track battery voltage
5. Calculate and report expected remaining battery life

Target: Achieve >30 days battery life with 2000 mAh LiPo

[Challenge] Challenge: Exercise 7: BLE-to-MQTT Gateway

Build a system that bridges BLE sensors to cloud:

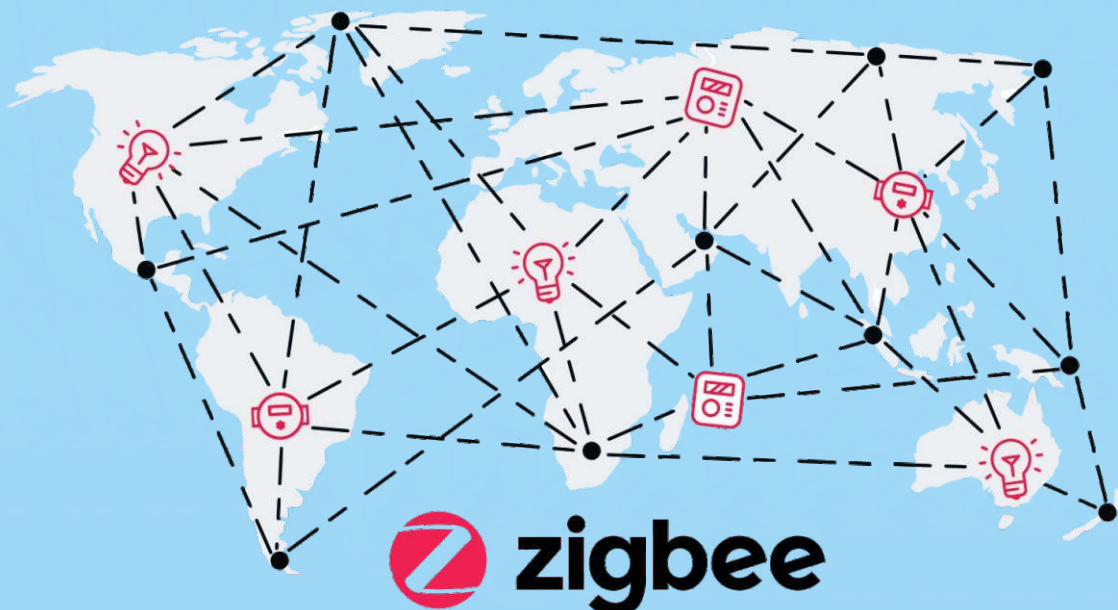
1. ESP32 #1: BLE peripheral broadcasting sensor data
2. ESP32 #2: BLE central + WiFi gateway
3. Gateway scans for BLE devices, reads data, publishes to MQTT
4. Support multiple BLE sensors simultaneously
5. Display aggregated data on web dashboard

— Further Reading. • ESP32 Documentation:

- ESP-IDF (Official Framework): <https://docs.espressif.com/projects/es>

- p-idf/
 - ESP32 Arduino Core: <https://github.com/espressif/arduino-esp32>
 - ESP32 Technical Reference Manual: https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf
- **Communication Protocols:**
 - MQTT Protocol Specification: <https://mqtt.org/>
 - BLE Getting Started: <https://learn.adafruit.com/introduction-to-bluetooth-low-energy>
 - ArduinoOTA Library: <https://github.com/esp8266/Arduino/tree/master/libraries/ArduinoOTA>
- **Tools and Simulators:**
 - Wokwi ESP32 Simulator: <https://wokwi.com>
 - MQTT Explorer (Desktop Client): <http://mqtt-explorer.com/>
 - nRF Connect (BLE Scanner App): iOS/Android app stores
- **Edge ML Integration:**
 - TensorFlow Lite Micro for ESP32: <https://github.com/tensorflow/tflite-micro-arduino-examples>
 - Edge Impulse ESP32 Support: <https://docs.edgeimpulse.com/docs/development-platforms/officially-supported-mcu-targets/espressif-esp32>

— **Credits.** This chapter uses the PubSubClient library by Nick O’Leary, Adafruit NeoPixel library, and built-in ArduinoOTA library. HiveMQ provides free public MQTT broker for testing (broker.hivemq.com). Code examples adapted from Espressif and Arduino community resources.



10. Biomedical Signal Processing: EMG-Based Edge

Learning Objectives

After completing this lab, you will be able to:

- Understand EMG (electromyography) signal generation and characteristics
- Implement signal processing techniques for noise reduction and feature extraction
- Build threshold-based and ML-based gesture classifiers
- Interface EMG sensors with Arduino for real-time processing
- Control actuators based on muscle signal classification
- Design complete bio-signal controlled systems for edge deployment under latency, memory, and energy constraints

Interactive Notebook

Jupyter Notebook: `LAB10_emg_biomedical.ipynb`

[Open in Google Colab](#) | [View on GitHub](#) | [Companion Website](#)

Prerequisites

Before starting this lab, ensure you:

- Completed LAB02 (Neural Network Training Fundamentals): ML fundamentals and model training
- Completed LAB05 (Embedded Systems Deployment): edge deployment basics
- Basic understanding of human anatomy (muscles, nerves)
- Signal processing concepts (filtering, frequency, sampling)

- Arduino programming with analog inputs

Hardware Requirements

- **Level 1:** Laptop with Python and signal processing libraries
- **Level 2:** Arduino + MyoWare EMG sensor + servo motor
- **Level 3:** Custom PCB with multiple EMG channels

⚠ Not Calibrating Per-User

EMG signal amplitude varies dramatically between individuals due to differences in muscle mass, skin impedance, electrode placement, and subcutaneous fat. A threshold calibrated on yourself will likely fail on another user. **Always implement a calibration routine** that asks the user to perform maximum voluntary contraction (MVC) and rest, then sets thresholds relative to their personal range.

■ **Theory and practice.** EMG signals measure muscle electrical activity, providing a natural interface for prosthetics and human-computer interaction. This chapter teaches signal processing fundamentals applicable to many biomedical signals (ECG, EEG, etc.) with a focus on edge deployment. You will reuse ideas from LAB02 (ML fundamentals), LAB03 (quantization and model size), and LAB08 (sensor interfacing), and the EMG classifier you design here can later be profiled in LAB11 and analysed from an energy perspective in LAB15.

Execution tiers. Level 1: Simulate EMG signals and develop processing algorithms in Python. Level 2: Connect real EMG sensors to Arduino and test classification on a simple setup. Level 3: Build complete prosthetic or gesture control systems with multi-channel EMG and actuators.

Three-Tier Execution Model

Tier 1: Jupyter/Colab

Simulate EMG signals, design bandpass filters, extract time and frequency features (RMS, MAV, ZC), train gesture classifiers on public EMG datasets

Tier 2: Arduino+MyoWare

Interface EMG sensors with analog inputs, implement real-time filtering and threshold detection, control servos based on muscle activity

Tier 3: Multi-channel EMG

Deploy pattern recognition for prosthetic control, implement user calibration routines, achieve sub-100ms gesture classification latency

10.1 Introduction: EMG for Edge ML

Electromyography (EMG) measures the electrical signals produced by muscles during contraction. These signals can be processed on edge devices to create intuitive control interfaces.

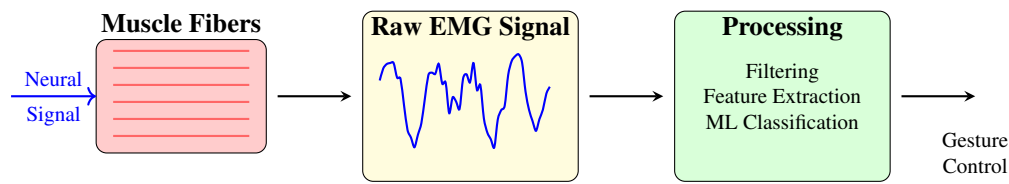


Figure 10.1: EMG signal flow: Neural signals trigger muscle fiber activation, producing electrical potentials measurable at the skin surface. Edge processing extracts features and classifies gestures for control applications.

EMG Signal Characteristics

Surface EMG signals have distinct properties:

- **Amplitude:** 0-10 mV (requires amplification)
- **Frequency range:** 20-500 Hz (most energy at 50-150 Hz)
- **Noise sources:** Power line (50/60 Hz), motion artifacts, electrode noise
- **Relationship to force:** Non-linear; increases with muscle contraction intensity

10.2 EMG Signal Acquisition

10.2.1 Hardware Setup

EMG System Components

Required:

- Arduino Uno or Nano 33 BLE Sense
- EMG sensor module (MyoWare, AD8232, or similar)
- EMG electrodes (disposable Ag/AgCl recommended)
- Servo motor for actuation demo

Optional for prosthetics:

- 3D printed prosthetic arm components
- MG996R servo motors (high torque)
- 9V batteries (2x for separate power supply)
- Fishing line and elastic cord for finger actuation

10.2.2 Electrode Placement

Proper electrode placement is critical for reliable EMG signals:

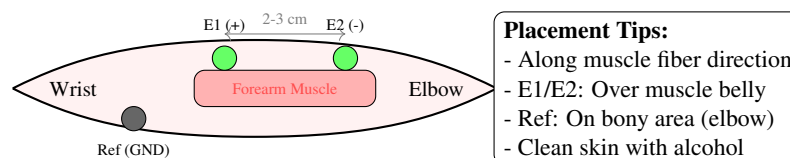


Figure 10.2: EMG electrode placement on the forearm: Two differential electrodes (E1, E2) are placed 2-3 cm apart along the muscle fiber direction. The reference electrode goes on a nearby bony area for noise cancellation.

Electrode Quality

EMG signal quality depends heavily on electrode contact:

- **Clean skin:** Remove oils and dead skin with alcohol
- **Fresh electrodes:** Replace after each session; dry gel = noisy signal
- **Secure attachment:** Movement causes artifacts
- **Consistent placement:** Mark positions for reproducible results

10.3 Signal Processing Fundamentals

10.3.1 EMG Signal Generation and Characteristics

Understanding the physiological origins of EMG signals is essential for effective processing and classification.

Definition — Motor Unit Action Potential (MUAP). When a motor neuron fires, it triggers simultaneous contraction of all muscle fibers in its motor unit. This synchronized depolarization produces a **motor unit action potential**—a brief (~ 5 -15 ms), biphasic or triphasic voltage spike measurable at the skin surface. Surface EMG is the algebraic sum of hundreds of overlapping MUAPs from different motor units.

EMG Signal Parameters

Surface EMG signals have well-characterized properties that determine processing requirements:

- **Amplitude range:** 0-10 mV at skin surface (requires 100-1000x amplification to ADC range)
- **Frequency content:** 20-500 Hz (dominant power at 50-150 Hz)
- **Bandwidth:** Approximately 300-400 Hz for surface EMG
- **Stochastic nature:** EMG is essentially Gaussian noise with time-varying amplitude
- **Force relationship:** RMS amplitude increases non-linearly with muscle force (approximately logarithmic)

The frequency content of EMG signals determines appropriate sampling rates and filter design:

Theorem — Nyquist Sampling for EMG. To avoid aliasing, the sampling rate f_s must satisfy:

$$f_s \geq 2 \cdot f_{\max}$$

where $f_{\max} = 500$ Hz is the maximum frequency component. Therefore, $f_s \geq 1000$ Hz is required. In practice, sampling at 1000-2000 Hz provides adequate signal fidelity with manageable data rates.

Signal-to-Noise Considerations

EMG signals face multiple noise sources that must be addressed through hardware and software:

- **Power line interference:** 50/60 Hz sinusoidal noise from AC mains (typically 10-100x larger than EMG)
- **Motion artifacts:** 0-20 Hz signals from electrode movement (can saturate amplifier)
- **Electrode-skin impedance:** High impedance (>10 k Ω) increases susceptibility to

external noise

- **Ambient electromagnetic interference:** Radio, Wi-Fi, fluorescent lights
- **Inherent electrode noise:** Thermal noise, shot noise ($\sim 5 \mu\text{V RMS}$)

Typical signal-to-noise ratio (SNR) for surface EMG: 20-40 dB during moderate contraction, <10 dB at rest.

10.3.2 Signal Conditioning Pipeline

A systematic filtering approach removes noise while preserving muscle activity information:

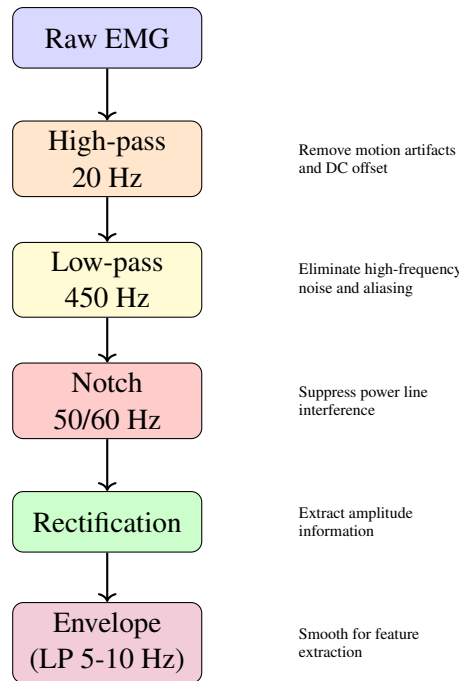


Figure 10.3: EMG signal conditioning pipeline: Sequential filtering stages remove specific noise sources while preserving muscle activity information. Each stage addresses a distinct signal quality issue.

Definition — Bandpass Filter (20-450 Hz). A bandpass filter combines high-pass and low-pass characteristics to isolate the EMG frequency band:

$$H_{BP}(f) = H_{HP}(f) \cdot H_{LP}(f)$$

High-pass cutoff (20 Hz): Removes motion artifacts and electrode drift (<20 Hz) without attenuating muscle signals (>20 Hz).

Low-pass cutoff (450 Hz): Eliminates high-frequency noise and prevents aliasing while preserving >95% of EMG signal energy.

Implementation: 2nd-order Butterworth filter provides flat passband response with minimal phase distortion:

$$H(s) = \frac{\omega_c^2}{s^2 + \sqrt{2}\omega_c s + \omega_c^2}$$

where $\omega_c = 2\pi f_c$ is the cutoff frequency.

Definition — Notch Filter (50/60 Hz). A notch filter (band-stop filter) removes power line interference at exactly 50 Hz (Europe/Asia) or 60 Hz (North America):

$$H_{\text{notch}}(f) = \frac{f^2 - f_0^2}{f^2 - f_0^2 + (f \cdot BW)}$$

where f_0 is the line frequency and $BW \approx 2$ Hz is the notch bandwidth.

Design consideration: Narrow bandwidth (2 Hz) removes interference without affecting nearby muscle frequencies. Quality factor $Q = f_0/BW \approx 30$ provides sharp notch.

Definition — Full-Wave Rectification. Rectification converts the bipolar EMG signal to a unipolar amplitude signal:

$$x_{\text{rect}}(t) = |x(t)|$$

This operation is necessary because muscle activation is encoded in signal amplitude, not polarity. The rectified signal envelope correlates with muscle force.

Definition — Envelope Extraction. After rectification, a low-pass filter extracts the signal envelope:

$$x_{\text{env}}(t) = \text{LPF}_{5-10 \text{ Hz}}(|x(t)|)$$

Cutoff selection: 5-10 Hz cutoff follows muscle force dynamics (slow compared to MUAP frequency). Lower cutoffs = smoother but slower response; higher cutoffs = faster but noisier.

Alternative: Moving average filter approximates low-pass filtering:

$$x_{\text{env}}[n] = \frac{1}{W} \sum_{k=0}^{W-1} |x[n-k]|$$

where W is the window size in samples.

10.3.3 Mathematical Foundations of EMG Signal Processing

Before implementing feature extraction algorithms, we establish the mathematical framework that underpins EMG signal analysis.

Definition — EMG as a Stochastic Process. Surface EMG can be modeled as a zero-mean, non-stationary Gaussian random process $x(t)$ whose variance $\sigma^2(t)$ encodes muscle activation:

$$x(t) \sim \mathcal{N}(0, \sigma^2(t))$$

The time-varying variance $\sigma^2(t)$ correlates with the number of active motor units and their firing rates. During isometric contraction at force F , the relationship is approximately:

$$\sigma^2(t) \propto F^\alpha, \quad \alpha \in [0.3, 0.6]$$

This sub-linear relationship arises from motor unit recruitment saturation and rate coding mechanisms.

Frequency Content and Spectral Characteristics

EMG signals exhibit well-defined spectral properties that guide filter design:

Power Spectral Density (PSD): The PSD describes how signal power is distributed

across frequencies:

$$S_{xx}(f) = \lim_{T \rightarrow \infty} \frac{1}{T} \mathbb{E} [|\mathcal{F}\{x(t)\}|^2]$$

where $\mathcal{F}\{\cdot\}$ denotes the Fourier transform.

Typical EMG spectrum:

- **Low frequency (0-20 Hz):** Motion artifacts, electrode movement, baseline drift
- **EMG band (20-150 Hz):** Primary muscle activity, peak power at 50-80 Hz
- **Mid frequency (150-300 Hz):** High-order motor unit harmonics
- **High frequency (300-500 Hz):** Minimal signal content, primarily noise

Mean frequency: First moment of the power spectrum:

$$f_{\text{mean}} = \frac{\int_0^\infty f \cdot S_{xx}(f) df}{\int_0^\infty S_{xx}(f) df}$$

Median frequency: Frequency dividing the spectrum into equal power halves:

$$\int_0^{f_{\text{med}}} S_{xx}(f) df = \int_{f_{\text{med}}}^\infty S_{xx}(f) df = \frac{1}{2} \int_0^\infty S_{xx}(f) df$$

Both metrics decrease with muscle fatigue as motor unit conduction velocity slows (typical shift: 80 Hz $\rightarrow$ 60 Hz after 5 minutes of sustained MVC).

Theorem — Digital Filter Implementation: Butterworth Bandpass. A k -th order Butterworth filter has transfer function:

$$|H(j\omega)|^2 = \frac{1}{1 + \left(\frac{\omega}{\omega_c}\right)^{2k}}$$

where ω_c is the cutoff frequency.

For bandpass filtering (20-450 Hz), we cascade high-pass and low-pass filters:

$$H_{\text{BP}}(j\omega) = H_{\text{HP}}(j\omega) \cdot H_{\text{LP}}(j\omega)$$

Digital implementation via bilinear transform: The continuous-time filter is mapped to discrete-time using:

$$s = \frac{2}{T} \cdot \frac{1 - z^{-1}}{1 + z^{-1}}$$

where $T = 1/f_s$ is the sampling period.

This yields the difference equation:

$$y[n] = \sum_{i=0}^k b_i x[n-i] - \sum_{j=1}^k a_j y[n-j]$$

Filter coefficients: For 2nd-order Butterworth at $f_c = 20$ Hz with $f_s = 1000$ Hz:

$$\omega_c = 2\pi f_c, \quad \lambda = \omega_c / f_s$$

$$b_0 = \frac{\lambda^2}{1 + \sqrt{2}\lambda + \lambda^2}, \quad a_1 = \frac{2(\lambda^2 - 1)}{1 + \sqrt{2}\lambda + \lambda^2}$$

Computational complexity: $O(k)$ multiplications per sample, where k is filter order.

Windowing and Segmentation for Feature Extraction

EMG classification requires partitioning continuous signals into analysis windows.

Window size selection: The window length W (in samples) determines temporal resolution vs statistical stability:

$$W = f_s \cdot T_w$$

where T_w is the window duration in seconds.

Trade-offs:

- **Small windows (50-100 ms):** Fast response, high temporal resolution, but noisy features
- **Medium windows (200-300 ms):** Balance between latency and stability (recommended for prosthetics)
- **Large windows (500-1000 ms):** Smooth features, low variance, but slow to detect changes

Overlap processing: To maintain temporal continuity, adjacent windows often overlap by 50-75%:

$$\text{Window}_i: \quad t_i = t_0 + i \cdot (1 - \alpha) \cdot T_w$$

where $\alpha \in [0.5, 0.75]$ is the overlap fraction.

Latency analysis: Total system latency from muscle activation to classification:

$$\text{Latency} = T_{\text{sensor}} + T_w + T_{\text{compute}} + T_{\text{actuation}}$$

For real-time prosthetic control, target total latency <200 ms to avoid perceptual lag.

Definition — Slope Sign Changes (SSC). SSC counts the number of times the signal slope changes sign, indicating waveform complexity:

$$\text{SSC} = \sum_{i=2}^{N-1} \mathbf{1}[(x_i - x_{i-1})(x_i - x_{i+1}) \geq \varepsilon]$$

where ε is a threshold to suppress noise-induced changes (typically $\varepsilon = 0.01 \cdot \max(|x|)$).

Interpretation: High SSC indicates rapid signal fluctuations, correlating with motor unit firing rates. SSC increases with muscle force as more motor units are recruited.

Computational advantage: Like ZC, SSC provides frequency information without FFT computation ($O(N)$ vs $O(N \log N)$).

Definition — Willison Amplitude (WAMP). WAMP measures the number of times the signal amplitude difference exceeds a threshold:

$$\text{WAMP} = \sum_{i=1}^{N-1} \mathbf{1}[|x_i - x_{i+1}| \geq \theta]$$

where θ is an application-specific threshold (e.g., $\theta = 0.05 \cdot \max(|x|)$).

Interpretation: WAMP captures rapid amplitude changes that characterize muscle activation bursts. Higher WAMP correlates with increased motor unit recruitment and synchronization.

Noise robustness: The threshold θ makes WAMP more robust to baseline noise than raw amplitude variance.

Theorem — Spectral Moments for Frequency Analysis. The m -th spectral moment characterizes frequency distribution:

$$M_m = \int_0^\infty f^m \cdot S_{xx}(f) df$$

Key moments:

- $M_0 = \int S_{xx}(f) df$: Total signal power
- M_1/M_0 : Mean frequency
- M_2 : Variance of frequency distribution

Spectral width (normalized 2nd moment):

$$\text{Spectral Width} = \sqrt{\frac{M_2}{M_0} - \left(\frac{M_1}{M_0}\right)^2}$$

This quantifies frequency spread and increases during high-intensity contractions due to broadband motor unit recruitment.

Computational approach: For discrete signals, approximate using FFT:

$$M_m \approx \sum_{k=0}^{N/2} f_k^m \cdot |X[k]|^2$$

where $X[k]$ is the discrete Fourier transform and $f_k = k \cdot f_s/N$.

Feature Normalization and Scaling

Raw features span vastly different numerical ranges, complicating ML model training.

Feature scale comparison:

- RMS: typically 0.01-1.0 (depends on amplitude units)
- ZC rate: 0-0.5 (normalized crossings per sample)
- WL: 10-1000 (cumulative sample differences)
- SSC: 10-200 (count of slope changes)

Min-Max Normalization:

$$x'_i = \frac{x_i - \min(x)}{\max(x) - \min(x)} \in [0, 1]$$

Z-score Standardization:

$$x'_i = \frac{x_i - \mu}{\sigma}, \quad \mu = \mathbb{E}[x], \quad \sigma = \sqrt{\text{Var}(x)}$$

MVC Normalization (for amplitude features):

$$x'_i = \frac{x_i}{x_{\text{MVC}}} \times 100\%$$

where x_{MVC} is the feature value during maximum voluntary contraction.

MVC normalization is physiologically interpretable and reduces inter-subject variability by 30-50%.

Theorem — Signal-to-Noise Ratio (SNR) Optimization. The SNR quantifies signal quality:

$$\text{SNR} = 10 \log_{10} \frac{P_{\text{signal}}}{P_{\text{noise}}} \text{ (dB)}$$

For EMG systems:

$$P_{\text{signal}} = \mathbb{E}[x^2(t)]_{\text{contraction}}, \quad P_{\text{noise}} = \mathbb{E}[x^2(t)]_{\text{rest}}$$

SNR improvement via filtering: Bandpass filtering increases SNR by rejecting out-of-band noise:

$$\text{SNR}_{\text{filtered}} = \text{SNR}_{\text{raw}} + 10 \log_{10} \frac{B_{\text{raw}}}{B_{\text{filter}}}$$

where B_{raw} is the raw signal bandwidth and $B_{\text{filter}} = 450 - 20 = 430$ Hz.

For typical EMG: $B_{\text{raw}} \approx 5000$ Hz (Nyquist at 10 kHz sampling), yielding:

$$\text{SNR improvement} \approx 10 \log_{10}(5000/430) \approx 10.7 \text{ dB}$$

Averaging improves SNR: For N independent samples, SNR improves by:

$$\text{SNR}_{\text{averaged}} = \text{SNR}_{\text{single}} + 10 \log_{10}(N)$$

Example: 100-sample moving average provides 20 dB SNR improvement.

10.3.4 Time-Domain Feature Extraction

Building on the mathematical foundations, we now implement efficient feature extraction algorithms that compress raw signal windows into compact numerical descriptors for classification.

Definition — Mean Absolute Value (MAV). The MAV measures the average rectified EMG amplitude over a time window:

$$\text{MAV} = \frac{1}{N} \sum_{i=1}^N |x_i|$$

where x_i is the i -th sample and N is the window size (typically 100-500 samples).

Interpretation: MAV correlates with muscle contraction intensity. Higher MAV indicates stronger activation.

Computational cost: $O(N)$ with only addition and absolute value operations—highly efficient for edge devices.

Definition — Root Mean Square (RMS). RMS provides a measure of signal power and correlates more strongly with muscle force than MAV:

$$\text{RMS} = \sqrt{\frac{1}{N} \sum_{i=1}^N x_i^2}$$

Interpretation: RMS weights large amplitude spikes more heavily than MAV due to squaring. This better captures motor unit recruitment patterns.

Relationship to force: For isometric contractions, $\text{RMS} \propto F^{0.5}$ approximately, where F is muscle force.

Computational cost: $O(N)$ but requires multiplication and square root—more expensive than MAV but still microcontroller-friendly.

Definition — Zero Crossing Rate (ZC). ZC counts the number of times the signal crosses zero (changes sign):

$$ZC = \sum_{i=1}^{N-1} \mathbf{1}[x_i \cdot x_{i+1} < 0]$$

where $\mathbf{1}[\cdot]$ is the indicator function (1 if condition true, 0 otherwise).

Interpretation: ZC provides frequency information. Higher muscle force recruits faster motor units, increasing dominant frequency and zero crossings.

Noise sensitivity: Raw ZC is sensitive to high-frequency noise. Apply threshold to ignore small crossings:

$$ZC_{\text{thresh}} = \sum_{i=1}^{N-1} \mathbf{1}[|x_i - x_{i+1}| \geq \varepsilon]$$

where ε is a small threshold (e.g., 5% of signal range).

Definition — Waveform Length (WL). WL measures the cumulative length of the waveform, capturing signal complexity:

$$WL = \sum_{i=1}^{N-1} |x_{i+1} - x_i|$$

Interpretation: WL increases with both amplitude and frequency content. Combines information from MAV and ZC.

Relationship to other features: WL is related to variance but emphasizes high-frequency changes:

- High WL, high MAV: Strong, rapid contraction
- High WL, low MAV: Tremor or noise
- Low WL, high MAV: Smooth, sustained contraction

Table 10.1: Comparison of EMG features for gesture recognition

Feature	Domain	Strengths	Complexity	Noise Robustness
MAV	Amplitude	Simple, correlates with force	Low	Medium
RMS	Power	Best force correlation, physiologically meaningful	Medium	High
Variance	Spread	Captures signal variability	Medium	Medium
ZC	Frequency	Frequency information without FFT	Low	Low (without threshold)
WL	Complexity	Combines amplitude and frequency	Low	Medium

Recommendation for gesture classification:

Use combination of RMS + MAV + WL for best accuracy/complexity trade-off.

Add ZC if frequency information improves class separability.

Why EMG for Gesture Recognition?

EMG offers unique advantages for human-computer interaction and prosthetic control:

- **Natural interface:** Detects muscle intention before visible movement occurs (~50-100

ms lead time)

- **Residual muscle control:** Amputees retain muscle tissue that can be voluntarily activated
- **Intuitive mapping:** Direct correspondence between muscle activation and desired action
- **Multiple channels:** Different muscles produce distinct signals, enabling multi-gesture recognition
- **High bandwidth:** Can detect rapid gesture transitions (<200 ms)
- **Proportional control:** Signal amplitude correlates with force, enabling variable-strength control

Limitations: Requires consistent electrode placement, sensitive to electrode drift, needs per-user calibration, and can fatigue with extended use.

Electrode Placement Criticality

EMG signal quality depends critically on electrode positioning:

- **Anatomical precision:** Electrodes must be over target muscle belly, not tendon or bone
- **Fiber direction:** Differential electrodes aligned parallel to muscle fibers maximize signal
- **Inter-electrode distance:** 2-3 cm spacing balances selectivity (smaller = more local) vs amplitude (larger = more motor units)
- **Cross-talk:** Poor placement picks up signals from adjacent muscles, reducing classification accuracy
- **Reproducibility:** Marking electrode positions or using 3D-printed guides improves session-to-session consistency

Practical impact: Studies show 5mm electrode displacement can change signal amplitude by 20-40%, necessitating recalibration.

Per-User Calibration Necessity

EMG signals exhibit high inter-subject variability due to:

- **Anatomical differences:** Muscle size, subcutaneous fat thickness (fat attenuates signal by 10-50%)
- **Skin properties:** Impedance varies 1-100 k Ω between individuals
- **Motor unit recruitment:** Different neural strategies produce same force with different EMG patterns
- **Electrode coupling:** Contact quality, gel conductivity, skin preparation

Calibration approaches:

- **MVC normalization:** Scale signals to percentage of maximum voluntary contraction
- **Per-user thresholds:** Determine decision boundaries from user's rest and activation samples
- **Transfer learning:** Pre-train on population, fine-tune with 50-100 user samples

Calibration frequency: Daily for same-day electrode placement, every session if electrodes are removed/reapplied.

Feature Selection Trade-offs

Choosing the right features involves balancing accuracy, computational cost, and robustness:

- **Single feature (e.g., RMS only):**
 - Pros: Minimal computation, fast inference (<1 ms)
 - Cons: Limited discrimination between similar gestures, sensitive to noise
 - Use case: Binary classification (rest vs active), very resource-constrained devices
- **2-3 features (RMS + MAV + WL):**
 - Pros: Good accuracy (85-90%), still efficient (<5 ms)
 - Cons: May miss frequency-based distinctions
 - Use case: Multi-gesture prosthetic control (3-5 gestures)
- **5+ features (add variance, ZC, frequency-domain):**
 - Pros: Highest accuracy (90-95%), captures multiple signal aspects
 - Cons: Increased computation, potential overfitting with small training sets
 - Use case: Research applications, high-precision control (>6 gestures)

Feature redundancy: MAV and RMS are correlated ($r > 0.9$); using both provides diminishing returns. Combine features from different domains (amplitude + frequency + complexity) for best discrimination.

10.3.5 Classification Algorithms for EMG Pattern Recognition

Once features are extracted, classification algorithms map feature vectors to gesture labels.

Definition — Linear Discriminant Analysis (LDA). LDA finds the linear combination of features that best separates classes. For binary classification, the decision boundary is:

$$w^T x + b = 0$$

where w is the weight vector and b is the bias.

Optimal weights: LDA maximizes the ratio of between-class to within-class variance:

$$w = S_W^{-1}(\mu_1 - \mu_2)$$

where S_W is the within-class scatter matrix and μ_i are class means.

Computational advantages:

- Training: $O(d^3)$ for d features (matrix inversion)
- Inference: $O(d)$ (dot product + comparison)
- Memory: Store only w and b ($d + 1$ parameters)

Limitations: Assumes Gaussian distribution and linear separability. Works well for 2-4 gesture classes but struggles with complex, overlapping patterns.

Definition — Support Vector Machine (SVM). SVM finds the maximum-margin hyperplane separating classes:

$$\min_{w,b} \frac{1}{2} \|w\|^2 + C \sum_{i=1}^N \xi_i$$

subject to $y_i(w^T x_i + b) \geq 1 - \xi_i$, $\xi_i \geq 0$

where C is the regularization parameter and ξ_i are slack variables allowing misclassification.

Kernel trick: Non-linear boundaries via kernel functions:

- Linear: $K(x_i, x_j) = x_i^T x_j$
- RBF (Gaussian): $K(x_i, x_j) = \exp(-\gamma \|x_i - x_j\|^2)$

- Polynomial: $K(x_i, x_j) = (x_i^T x_j + c)^p$

For edge deployment:

- Training: Computationally expensive, done offline
- Inference: $O(n_{SV} \cdot d)$ where n_{SV} is number of support vectors
- Memory: Store support vectors ($n_{SV} \times d$) and weights

EMG performance: RBF-SVM achieves 85-92% accuracy for 5-gesture classification with careful hyperparameter tuning (C, γ).

Definition — Random Forest Classifier. Random Forest ensembles multiple decision trees trained on bootstrap samples:

$$\hat{y} = \text{mode}\{h_1(x), h_2(x), \dots, h_T(x)\}$$

where h_t is the t -th decision tree and T is the number of trees.

Decision tree structure: Each tree recursively splits data based on feature thresholds:

$$\text{Split at node } i: \quad x_j < \theta \Rightarrow \text{left child, else right child}$$

Split criterion (Gini impurity):

$$G = \sum_{k=1}^K p_k(1 - p_k)$$

where p_k is the proportion of samples in class k .

Computational characteristics:

- Training: $O(T \cdot N \log N \cdot d)$ for N samples, T trees, d features
- Inference: $O(T \cdot \log_2(n_{\text{leaves}}))$ (traverse T trees)
- Memory: Store tree structures (~ 10 -50 KB for 10-100 trees)

Advantages for EMG:

- Handles non-linear relationships without kernel tricks
- Robust to outliers and noisy features
- Provides feature importance rankings
- Achieves 88-93% accuracy with 10-50 trees

Definition — Convolutional Neural Network (CNN) for Raw EMG. CNNs learn hierarchical features directly from raw or minimally processed EMG:

$$\text{Input: } x \in \mathbb{R}^{T \times C} \rightarrow \text{Conv layers} \rightarrow \text{Pooling} \rightarrow \text{Dense} \rightarrow \text{Softmax}$$

where T is the time window length and C is the number of channels.

1D convolutional layer:

$$y[n] = \sigma \left(\sum_{k=0}^{K-1} w[k] \cdot x[n-k] + b \right)$$

where w is the learned filter, K is kernel size, and σ is activation (ReLU, tanh).

Typical architecture for EMG:

- Input: 200 samples $\times$ 4 channels (200 ms @ 1000 Hz, 4 EMG electrodes)
- Conv1D: 32 filters, kernel size 5, ReLU
- MaxPool1D: pool size 2

- Conv1D: 64 filters, kernel size 3, ReLU
 - GlobalAveragePooling1D
 - Dense: 32 units, ReLU
 - Dense: 5 units (gestures), Softmax
- Parameter count:** ~50,000-200,000 parameters

Quantization for edge deployment:

- Float32: ~200-800 KB
- Int8 quantization: ~50-200 KB (4x reduction)
- Accuracy trade-off: <1-2% accuracy loss

Advantages: Learns optimal features automatically, achieves 92-96% accuracy, robust to electrode placement variations.

Disadvantages: Requires large training datasets (>1000 samples per gesture), higher computational cost (10-50 ms inference on microcontrollers).

Definition — Recurrent Neural Network (RNN/LSTM) for Temporal Dynamics. RNNs capture temporal dependencies in EMG sequences:

$$h_t = \sigma(W_{hh}h_{t-1} + W_{xh}x_t + b_h)$$

$$y_t = W_{hy}h_t + b_y$$

where h_t is the hidden state encoding temporal context.

LSTM (Long Short-Term Memory): Addresses vanishing gradients with gating mechanisms:

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f) \quad (\text{forget gate})$$

$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i) \quad (\text{input gate})$$

$$\tilde{C}_t = \tanh(W_C[h_{t-1}, x_t] + b_C) \quad (\text{candidate cell state})$$

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \quad (\text{cell state update})$$

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o) \quad (\text{output gate})$$

$$h_t = o_t \odot \tanh(C_t) \quad (\text{hidden state})$$

EMG application: Useful for detecting gesture transitions and temporal patterns (e.g., sustained vs brief contractions).

Computational cost: Higher than CNNs due to sequential processing (cannot parallelize across time). Inference: 20-100 ms on microcontrollers.

Typical performance: 90-95% accuracy for gesture sequences, excels at detecting dynamic gestures (waves, rotations) vs static (grips, poses).

Model Complexity vs Edge Constraints

Choosing a classification algorithm for edge deployment requires balancing accuracy, latency, memory, and energy:

Classification algorithm comparison for edge EMG systems

Algorithm	Accuracy	Latency	Memory	Energy	Training Data
Threshold	75-82%	<1 ms	<1 KB	Very Low	None
LDA	80-86%	1-2 ms	1-5 KB	Very Low	50-200
SVM (RBF)	85-92%	2-5 ms	5-50 KB	Low	200-500
Random Forest	88-93%	3-10 ms	10-50 KB	Low	200-500
CNN (quantized)	92-96%	10-50 ms	50-200 KB	Medium	1000-5000
LSTM (quantized)	90-95%	20-100 ms	100-500 KB	High	1000-5000

Decision framework:

- **MCU with <32 KB RAM:** Threshold or LDA
- **Arduino-class (32-256 KB):** SVM or Random Forest
- **ARM Cortex-M4 (256 KB+):** Quantized CNN
- **High-performance edge (ESP32):** Full CNN or LSTM

Theorem — Real-Time Inference Constraints. For prosthetic control, the total system latency must satisfy:

$$T_{\text{total}} = T_{\text{acq}} + T_{\text{feature}} + T_{\text{classify}} + T_{\text{actuate}} < 200 \text{ ms}$$

Latency budget breakdown:

- T_{acq} : Window accumulation = 100-200 ms (depends on window size)
- T_{feature} : Feature extraction = 1-5 ms (time-domain) or 10-50 ms (FFT)
- T_{classify} : Model inference = 1-50 ms (depends on algorithm)
- T_{actuate} : Servo/motor response = 10-50 ms

Optimization strategies:

1. **Reduce window size:** 100 ms windows instead of 200 ms (saves 100 ms but increases noise)
2. **Use overlap:** Process 50% overlapping windows for continuous classification
3. **Parallel processing:** Compute features for window n while classifying window $n - 1$
4. **Early termination:** Stop classification when confidence >95% (saves 20-40% compute)
5. **Cascaded classifiers:** Fast threshold pre-filter, expensive model only for uncertain cases

Perceptual threshold: Users perceive latency >200 ms as "sluggish" and <100 ms as "natural."

Training Data Collection Best Practices

High-quality EMG datasets are essential for robust classification:

Data collection protocol:

1. **Calibration session:** 5-10 minutes per user
2. **Gesture repetitions:** 20-50 trials per gesture (randomized order)
3. **Hold duration:** 3-5 seconds per gesture with 2-second rest between
4. **Multiple sessions:** Collect data across 3+ sessions (different days) to capture variability
5. **Electrode replacement:** Remove and reapply electrodes between sessions to test robustness

Data augmentation:

- **Time shifting:** Shift windows by ± 10 -50 ms to increase samples
- **Amplitude scaling:** Scale by 0.8-1.2x to simulate force variation
- **Noise injection:** Add Gaussian noise (SNR 20-30 dB) to improve robustness

- **Window jittering:** Random start positions within gesture segments

Labeling strategy:

- Exclude first/last 500 ms of each gesture (transient periods)
- Label only stable contraction phases
- Use majority voting for overlapping windows

Cross-validation:

- **Within-session:** 80/20 train/test split (optimistic estimate)
- **Cross-session:** Train on sessions 1-2, test on session 3 (realistic)
- **Leave-one-user-out:** Test generalization to new users (pessimistic)

Raw EMG signals require processing before classification.

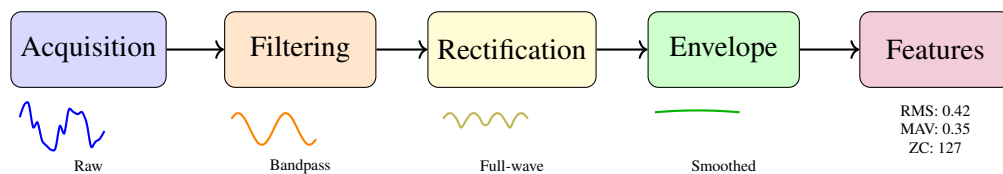


Figure 10.4: EMG signal processing pipeline: Raw signals are filtered to remove noise, rectified to capture magnitude, smoothed to extract the envelope, and then features are computed for classification.

10.3.6 Python Implementation: Filtering and Feature Extraction

Before deploying to microcontrollers, develop and test filtering algorithms in Python:

```

1 import numpy as np
2 from scipy import signal
3 import matplotlib.pyplot as plt
4
5 def bandpass_filter(data, lowcut=20, highcut=450, fs=1000, order=2):
6     """
7     Apply Butterworth bandpass filter to EMG signal.
8
9     Args:
10         data: Raw EMG signal
11         lowcut: High-pass cutoff (Hz) - removes motion artifacts
12         highcut: Low-pass cutoff (Hz) - removes high-frequency noise
13         fs: Sampling frequency (Hz)
14         order: Filter order (2 = gentle rolloff, 4 = steeper)
15
16     Returns:
17         Filtered EMG signal
18     """
19     nyquist = 0.5 * fs
20     low = lowcut / nyquist
21     high = highcut / nyquist
22
23     # Design Butterworth bandpass filter
24     b, a = signal.butter(order, [low, high], btype='band')
25
26     # Apply filter (use filtfilt for zero-phase filtering)
27     filtered_data = signal.filtfilt(b, a, data)
28
29     return filtered_data
30
31 def notch_filter(data, f0=60, Q=30, fs=1000):

```

```

32 """
33 Apply notch filter to remove power line interference.
34
35 Args:
36     data: Input signal
37     f0: Notch frequency (50 Hz Europe/Asia, 60 Hz Americas)
38     Q: Quality factor (higher = narrower notch)
39     fs: Sampling frequency
40
41 Returns:
42     Notch-filtered signal
43 """
44 # Design notch filter
45 b, a = signal.iirnotch(f0, Q, fs)
46
47 # Apply filter
48 filtered_data = signal.filtfilt(b, a, data)
49
50 return filtered_data
51
52 # Example usage
53 fs = 1000 # 1000 Hz sampling rate
54 t = np.linspace(0, 2, 2*fs) # 2 seconds of data
55
56 # Simulate EMG with noise
57 emg_signal = 0.5 * np.random.randn(len(t)) # Muscle activity
58 powerline_noise = 0.3 * np.sin(2*np.pi*60*t) # 60 Hz interference
59 motion_artifact = 0.2 * np.sin(2*np.pi*3*t) # 3 Hz drift
60 raw_emg = emg_signal + powerline_noise + motion_artifact
61
62 # Apply filtering pipeline
63 filtered_emg = bandpass_filter(raw_emg, fs=fs)
64 clean_emg = notch_filter(filtered_emg, f0=60, fs=fs)
65
66 # Visualize
67 plt.figure(figsize=(12, 6))
68 plt.subplot(3, 1, 1)
69 plt.plot(t[:500], raw_emg[:500])
70 plt.title('Raw EMG (with 60 Hz noise and motion artifacts)')
71 plt.ylabel('Amplitude (mV)')
72
73 plt.subplot(3, 1, 2)
74 plt.plot(t[:500], filtered_emg[:500])
75 plt.title('After Bandpass Filter (20–450 Hz)')
76 plt.ylabel('Amplitude (mV)')
77
78 plt.subplot(3, 1, 3)
79 plt.plot(t[:500], clean_emg[:500])
80 plt.title('After Notch Filter (60 Hz removed)')
81 plt.xlabel('Time (s)')
82 plt.ylabel('Amplitude (mV)')
83 plt.tight_layout()
84 plt.show()

```

Code Walkthrough

scipy.signal.butter: Designs Butterworth filter coefficients. Butterworth filters have maximally flat passband response, ideal for preserving EMG signal shape.

signal.filtfilt: Applies filter forward and backward to eliminate phase shift. Standard filtering introduces delay; filtfilt maintains temporal alignment.

Quality factor Q: Controls notch width. $Q=30$ gives ≈ 2 Hz notch bandwidth, removing 60 Hz without affecting 58 Hz or 62 Hz components.

```

1 def extract_emg_features(signal_window):
2     """
3     Extract time-domain features from EMG signal window.
4
5     Args:
6         signal_window: 1D array of EMG samples (e.g., 200–500 samples)
7
8     Returns:
9         Dictionary of feature values
10    """
11    N = len(signal_window)
12
13    # Mean Absolute Value (MAV)
14    mav = np.mean(np.abs(signal_window))
15
16    # Root Mean Square (RMS)
17    rms = np.sqrt(np.mean(signal_window**2))
18
19    # Variance
20    variance = np.var(signal_window)
21
22    # Zero Crossing Rate (with threshold to reduce noise sensitivity)
23    threshold = 0.05 * np.max(np.abs(signal_window))
24    zc = 0
25    for i in range(N-1):
26        if (signal_window[i] * signal_window[i+1] < 0 and
27            np.abs(signal_window[i] - signal_window[i+1]) > threshold):
28            zc += 1
29    zc_rate = zc / N
30
31    # Waveform Length (WL)
32    wl = np.sum(np.abs(np.diff(signal_window)))
33
34    features = {
35        'MAV': mav,
36        'RMS': rms,
37        'Variance': variance,
38        'ZC_Rate': zc_rate,
39        'WL': wl
40    }
41
42    return features
43
44 # Example: Extract features from filtered EMG
45 window_size = 200 # 200 samples = 200ms at 1000 Hz
46 window = clean_emg[500:700] # Example window
47
48 features = extract_emg_features(window)
49 print("Extracted Features:")
50 for name, value in features.items():
51     print(f" {name}: {value:.4f}")

```

Code Walkthrough

Threshold in ZC: The condition `np.abs(signal_window[i] - signal_window[i+1]) > threshold` filters out noise-induced zero crossings. Without this, high-frequency noise dominates ZC count.

Feature normalization: For machine learning, normalize features to [0,1] or standardize to zero mean, unit variance. Different features have vastly different scales (RMS ~ 0.1 , WL ~ 100).

```

1 def simple_threshold_classifier(rms_value, threshold_rest=0.1,
2                               threshold_active=0.3):
3     """
4     Simple threshold-based gesture classifier.
5
6     Args:
7         rms_value: RMS feature value
8         threshold_rest: Below this = rest
9         threshold_active: Above this = strong contraction
10
11     Returns:
12         0 = rest, 1 = light contraction, 2 = strong contraction
13     """
14     if rms_value < threshold_rest:
15         return 0 # Rest
16     elif rms_value < threshold_active:
17         return 1 # Light contraction
18     else:
19         return 2 # Strong contraction
20
21 # Example: Classify windows
22 for i in range(0, len(clean_emg)-window_size, window_size):
23     window = clean_emg[i:i+window_size]
24     features = extract_emg_features(window)
25     gesture = simple_threshold_classifier(features['RMS'])
26
27     gesture_names = ['Rest', 'Light', 'Strong']
28     print(f"Window {i//window_size}: RMS={features['RMS']:.3f}, "
29           f"Gesture={gesture_names[gesture]}")

```

Code Walkthrough

Threshold selection: Determine thresholds empirically by collecting calibration data. Compute 90th percentile of rest RMS for lower threshold, 50th percentile of active RMS for upper threshold.

Hysteresis: Production code should add hysteresis bands to prevent oscillation near boundaries.

10.3.7 Arduino Implementation: Moving Average Filter

Translate Python algorithms to efficient C code for real-time microcontroller execution:

```

1 // EMG signal processing for Arduino
2 const int EMG_PIN = A0;
3 const int WINDOW_SIZE = 50; // 50 samples = 100ms at 500Hz
4
5 int signalBuffer[WINDOW_SIZE];
6 int bufferIndex = 0;
7 long runningSum = 0;

```

Code Walkthrough

WINDOW_SIZE = 50: At 500 Hz sampling, this gives a 100ms averaging window. Larger windows = smoother output but slower response.

runningSum: Maintains cumulative sum for O(1) average calculation.

```

1 void setup() {
2     Serial.begin(115200);
3     for (int i = 0; i < WINDOW_SIZE; i++) {

```

```

4     signalBuffer[i] = 0;
5 }
6 Serial.println("EMG Processing Ready");
7 }
8
9 int getFilteredEMG() {
10     int rawValue = analogRead(EMG_PIN);
11
12     // Update circular buffer
13     runningSum -= signalBuffer[bufferIndex];
14     signalBuffer[bufferIndex] = rawValue;
15     runningSum += rawValue;
16     bufferIndex = (bufferIndex + 1) % WINDOW_SIZE;
17
18     // Return moving average
19     return runningSum / WINDOW_SIZE;
20 }

```

Code Walkthrough

Circular buffer: Efficiently stores last N samples without shifting.

Running sum update: Subtract old value, add new value—O(1) complexity regardless of window size.

10.3.8 Feature Extraction

For ML-based classification, we extract features from signal windows:

Table 10.2: Common EMG features for gesture classification

Feature	Formula	Interpretation
Mean Absolute Value (MAV)	$\frac{1}{N} \sum x_i $	Average signal intensity
Root Mean Square (RMS)	$\sqrt{\frac{1}{N} \sum x_i^2}$	Signal power
Zero Crossings (ZC)	Count of sign changes	Frequency information
Waveform Length (WL)	$\sum x_{i+1} - x_i $	Signal complexity
Variance	$\frac{1}{N} \sum (x_i - \bar{x})^2$	Signal variability

```

1 struct EMGFeatures {
2     float mean;
3     float rms;
4     float variance;
5     float zeroCrossings;
6     float meanAbsValue;
7     float waveformLength;
8 };
9
10 EMGFeatures extractFeatures(float* window, int windowSize) {
11     EMGFeatures features;
12     float sum = 0, sumSquares = 0, sumAbs = 0;
13     int zeroCrossings = 0;
14     float waveformLength = 0;
15
16     for (int i = 0; i < windowSize; i++) {
17         float val = window[i];
18         sum += val;
19         sumSquares += val * val;
20         sumAbs += abs(val);

```

```

21
22     if (i > 0) {
23         // Zero crossing detection
24         if ((window[i-1] > 0 && val < 0) ||
25             (window[i-1] < 0 && val > 0)) {
26             zeroCrossings++;
27         }
28         // Waveform length
29         waveformLength += abs(val - window[i-1]);
30     }
31 }
32
33 features.mean = sum / windowSize;
34 features.rms = sqrt(sumSquares / windowSize);
35 features.variance = (sumSquares / windowSize) -
36                     (features.mean * features.mean);
37 features.zeroCrossings = (float)zeroCrossings / windowSize;
38 features.meanAbsValue = sumAbs / windowSize;
39 features.waveformLength = waveformLength;
40
41 return features;
42 }

```

Code Walkthrough

Single-pass computation: All features computed in one loop through the data—efficient for micro-controllers.

Normalization: ZC is normalized by window size for consistent scale across different window sizes.

Feature struct: Groups related data for easy passing to classifier.

10.4 Gesture Classification

10.4.1 Threshold-Based Classification

For simple applications, threshold-based classification works well:

```

1  const int THRESHOLD_REST = 100;
2  const int THRESHOLD_LIGHT = 300;
3  const int THRESHOLD_MEDIUM = 500;
4
5  int classifyGesture(int filteredEMG) {
6      if (filteredEMG < THRESHOLD_REST) {
7          return 0; // Rest
8      } else if (filteredEMG < THRESHOLD_LIGHT) {
9          return 1; // Light contraction
10     } else if (filteredEMG < THRESHOLD_MEDIUM) {
11         return 2; // Medium contraction
12     } else {
13         return 3; // Strong contraction
14     }
15 }

```

Finding Optimal Thresholds

To determine thresholds for your setup:

1. Log raw/filtered values to Serial Monitor
2. Ask user to perform each gesture 10 times
3. Find the range of values for each gesture

4. Set thresholds at midpoints between ranges
5. Add hysteresis to prevent oscillation

💡 Designing for Accessibility

Consider building an EMG-controlled device for someone with a limb difference:

1. Where might residual muscles be located that could generate useful EMG signals?
2. How would your calibration procedure change if the user has limited range of motion?
3. What happens if the user's muscle fatigues during extended use? How would you adapt?
4. What feedback mechanisms (visual, auditory, haptic) would help the user learn to control the device?

Inclusive design requires thinking beyond “average” users from the start.

10.4.2 Adding Hysteresis

Hysteresis prevents rapid switching between states near threshold boundaries:

```

1  const int HYSTERESIS = 20;
2  int currentGesture = 0;
3
4  int classifyWithHysteresis(int filteredEMG) {
5      int newGesture = currentGesture;
6
7      // Only change state if signal moves significantly
8      if (currentGesture == 0 && filteredEMG > THRESHOLD_REST + HYSTERESIS) {
9          newGesture = 1;
10     } else if (currentGesture == 1) {
11         if (filteredEMG < THRESHOLD_REST - HYSTERESIS) {
12             newGesture = 0;
13         } else if (filteredEMG > THRESHOLD_LIGHT + HYSTERESIS) {
14             newGesture = 2;
15         }
16     }
17     // ... continue for other states
18
19     currentGesture = newGesture;
20     return currentGesture;
21 }
```

Code Walkthrough

Hysteresis band: The signal must move beyond the threshold by the hysteresis amount to trigger a state change. This prevents rapid toggling when the signal hovers near a threshold.

10.4.3 ML-Based Classification

For more robust classification, train an ML model on extracted features:

Level 1: Training in Python

```

1  import numpy as np
2  from sklearn.ensemble import RandomForestClassifier
3  from sklearn.model_selection import train_test_split
```

```

4 import tensorflow as tf
5
6 # Load collected EMG feature data
7 # Format: [MAV, RMS, Variance, ZC, WL, label]
8 data = np.loadtxt('emg_features.csv', delimiter=',')
9 X = data[:, :-1] # Features
10 y = data[:, -1] # Labels
11
12 # Split data
13 X_train, X_test, y_train, y_test = train_test_split(
14     X, y, test_size=0.2, random_state=42
15 )
16
17 # Train a simple classifier
18 clf = RandomForestClassifier(n_estimators=10, max_depth=5)
19 clf.fit(X_train, y_train)
20 print(f"Accuracy: {clf.score(X_test, y_test):.2%}")

```

```

1 # For edge deployment, train a tiny neural network
2 model = tf.keras.Sequential([
3     tf.keras.layers.Dense(8, activation='relu', input_shape=(5,)),
4     tf.keras.layers.Dense(4, activation='softmax')
5 ])
6
7 model.compile(optimizer='adam',
8               loss='sparse_categorical_crossentropy',
9               metrics=['accuracy'])
10
11 model.fit(X_train, y_train, epochs=50, validation_split=0.2)
12
13 # Convert for TFLite Micro
14 converter = tf.lite.TFLiteConverter.from_keras_model(model)
15 converter.optimizations = [tf.lite.Optimize.DEFAULT]
16 tflite_model = converter.convert()
17 print(f"Model size: {len(tflite_model)} bytes")

```

Model Size for EMG Classification

A typical EMG gesture classifier with 5 input features and 4 output classes:

- **Dense(5) → Dense(8) → Dense(4):** ~200 parameters
- **Float32 size:** ~800 bytes
- **Int8 quantized:** ~300 bytes

This easily fits on any Arduino-class microcontroller!

10.5 Actuator Control

Control servo motors based on gesture classification:

```

1 #include <Servo.h>
2
3 Servo gripServo;
4 const int SERVO_PIN = 9;
5
6 void setup() {
7     gripServo.attach(SERVO_PIN);
8     gripServo.write(0); // Start open
9 }

```

```

10
11 void controlGrip(int gestureClass) {
12     switch (gestureClass) {
13         case 0: // Rest – fully open
14             gripServo.write(0);
15             break;
16         case 1: // Light – partially closed
17             gripServo.write(45);
18             break;
19         case 2: // Medium – mostly closed
20             gripServo.write(120);
21             break;
22         case 3: // Strong – fully closed (grip)
23             gripServo.write(180);
24             break;
25     }
26 }

```

Power Supply for Servos

Servos draw significant current (500mA - 2A depending on load). Do NOT power directly from Arduino's 5V pin:

- Use external 5-6V power supply for servos
- Connect servo ground to Arduino ground
- Only connect signal wire to Arduino PWM pin

Insufficient power causes erratic behavior and can damage the Arduino.

10.6 Complete EMG Control System

Here's a complete system combining all components:

```

1 #include <Servo.h>
2
3 // Hardware pins
4 const int EMG_PIN = A0;
5 const int SERVO_PIN = 9;
6 const int LED_PIN = LED_BUILTIN;
7
8 // Signal processing
9 const int WINDOW_SIZE = 50;
10 const int SAMPLE_RATE = 500;
11 int signalBuffer[WINDOW_SIZE];
12 int bufferIndex = 0;
13 long runningSum = 0;
14
15 // Classification
16 const int THRESHOLD_REST = 100;
17 const int THRESHOLD_GRIP = 400;
18 const int HYSTERESIS = 30;
19 int currentGesture = 0;
20
21 Servo gripServo;
22 unsigned long lastSampleTime = 0;
23
24 void setup() {
25     Serial.begin(115200);
26     pinMode(LED_PIN, OUTPUT);
27     gripServo.attach(SERVO_PIN);

```

```

28   gripServo.write(0);
29
30   // Initialize buffer
31   for (int i = 0; i < WINDOW_SIZE; i++) {
32       signalBuffer[i] = 0;
33   }
34
35   Serial.println("EMG Control System Ready");
36 }
37
38 void loop() {
39     unsigned long currentTime = millis();
40
41     if (currentTime - lastSampleTime >= (1000 / SAMPLE_RATE)) {
42         lastSampleTime = currentTime;
43
44         // Read and filter
45         int rawEMG = analogRead(EMG_PIN);
46         runningSum -= signalBuffer[bufferIndex];
47         signalBuffer[bufferIndex] = rawEMG;
48         runningSum += rawEMG;
49         bufferIndex = (bufferIndex + 1) % WINDOW_SIZE;
50         int filteredEMG = runningSum / WINDOW_SIZE;
51
52         // Classify with hysteresis
53         if (currentGesture == 0 &&
54             filteredEMG > THRESHOLD_GRIP + HYSTERESIS) {
55             currentGesture = 1;
56         } else if (currentGesture == 1 &&
57             filteredEMG < THRESHOLD_REST - HYSTERESIS) {
58             currentGesture = 0;
59         }
60
61         // Control actuator
62         if (currentGesture == 1) {
63             gripServo.write(180); // Close grip
64             digitalWrite(LED_PIN, HIGH);
65         } else {
66             gripServo.write(0); // Open grip
67             digitalWrite(LED_PIN, LOW);
68         }
69
70         // Debug output
71         Serial.print(rawEMG);
72         Serial.print(",");
73         Serial.print(filteredEMG);
74         Serial.print(",");
75         Serial.println(currentGesture);
76     }
77 }

```

10.7 Three-Tier Implementation Guide

Level 1: Simulation and Algorithm Development

Develop and test algorithms with simulated EMG signals:

```

1 import numpy as np
2 import matplotlib.pyplot as plt

```



```

3
4 def simulate_emg(duration_s, sampling_rate, is_contracted):
5     """Generate synthetic EMG signal."""
6     n_samples = int(duration_s * sampling_rate)
7     t = np.linspace(0, duration_s, n_samples)
8
9     if is_contracted:
10        # Higher amplitude during contraction
11        amplitude = 0.5 + 0.3 * np.random.rand()
12        freq = 80 + 40 * np.random.rand()
13    else:
14        # Low amplitude at rest
15        amplitude = 0.1 * np.random.rand()
16        freq = 30
17
18    # Simulate motor unit action potentials
19    signal = amplitude * np.sin(2 * np.pi * freq * t)
20    signal += 0.1 * np.random.randn(n_samples) # Noise
21
22    return signal
23
24 # Generate training data
25 rest_signal = simulate_emg(2, 1000, False)
26 contract_signal = simulate_emg(2, 1000, True)
27
28 plt.figure(figsize=(12, 4))
29 plt.subplot(1, 2, 1)
30 plt.plot(rest_signal[:500])
31 plt.title('Rest')
32 plt.subplot(1, 2, 2)
33 plt.plot(contract_signal[:500])
34 plt.title('Contracted')
35 plt.show()

```

Level 2: Hardware Testing

Connect real EMG sensor and test processing:

1. Wire EMG sensor to Arduino (signal to A0, power to 5V/GND)
2. Upload processing sketch and open Serial Plotter
3. Attach electrodes and observe signal during rest/contraction
4. Determine appropriate thresholds for your setup
5. Add servo and test closed-loop control

Level 3: Prosthetic System

Build complete prosthetic arm:

1. 3D print arm components (fingers, palm, wrist)
2. Assemble with servo motors and fishing line tendons
3. Integrate EMG sensor with robust electrode attachment
4. Implement multi-gesture recognition (open, close, pinch)
5. Add battery power for portable operation

10.8 Common Challenges and Solutions

Table 10.3: Troubleshooting EMG systems

Problem	Solution
Noisy signal	Check electrode contact; clean skin; use fresh electrodes; add 50/60 Hz notch filter
Signal drift	Allow warmup time; use AC-coupled amplifier; high-pass filter at 20 Hz
False triggers	Increase hysteresis; add temporal filtering; require sustained activation
Weak signal	Reposition electrodes over muscle belly; reduce reference electrode distance
Servo jitter	Separate servo power supply; add capacitors; implement dead zone
Muscle fatigue	Add rest requirements; reduce activation threshold over time

10.9 Troubleshooting Guide

⚠ EMG Signal Too Noisy or Contains 50/60Hz Hum

Symptoms: Signal dominated by power line frequency, making muscle activity hard to detect

Cause: AC power line interference, poor electrode contact, inadequate shielding

Solution:

1. Apply notch filter: Remove 50Hz or 60Hz component digitally
2. Improve electrode placement: Clean skin with alcohol, apply conductive gel
3. Use shielded cables: Prevents electromagnetic interference pickup
4. Check ground electrode: Ensure proper connection to reference point
5. Use differential amplifier: MyoWare or AD8232 has built-in CMRR
6. Move away from power sources: Keep setup distant from power adapters, fluorescent lights

⚠ Sensor Readings Saturate at Max Value

Symptoms: Output stays at maximum (1023, 4095) regardless of muscle activity

Cause: Amplifier gain too high, electrodes placed on nerve instead of muscle

Solution:

1. Reduce gain: Adjust potentiometer on MyoWare sensor (clockwise to decrease)
2. Reposition electrodes: Place on muscle belly, not near bones or nerves
3. Check electrode spacing: 2-3cm apart for surface EMG
4. Verify reference electrode: Should be on electrically neutral location (elbow, wrist bone)
5. Test with relaxed muscle first: Should show baseline before flexing

⚠️ Gesture Recognition Accuracy Low

Symptoms: Model confuses similar gestures, inconsistent predictions

Cause: Insufficient training data, feature extraction issues, electrode drift

Solution:

1. Collect more training samples: At least 50 examples per gesture
2. Normalize signal amplitude: Different users have different muscle strength
3. Extract frequency domain features: Use FFT to capture muscle activation patterns
4. Implement sliding window: Continuous classification instead of fixed segments
5. Recalibrate per session: EMG signal varies with electrode placement
6. Add motion artifact rejection: Filter out movements unrelated to gestures

10.10 Summary

In this chapter, you learned:

- **EMG fundamentals:** Muscle electrical signals range 0-10mV at 20-500 Hz
- **Electrode placement:** Differential electrodes along muscle fibers with distant reference
- **Signal processing:** Filtering, rectification, envelope extraction
- **Feature extraction:** MAV, RMS, zero crossings, waveform length
- **Classification:** Threshold-based for simple cases, ML for robust recognition
- **Actuator control:** Servo motors with proper power supply and hysteresis

Self-Assessment Questions

Test your understanding before continuing:

1. Describe how EMG sensors detect muscle electrical activity. Why is signal amplitude in the range of 0-5mV and what amplification is typically needed?
2. Explain why a moving average filter with window size 50 reduces noise but introduces lag. What is the trade-off for real-time prosthetic control?
3. Calculate RMS, MAV, and ZC features from a 200ms EMG signal window sampled at 500Hz. Why does RMS correlate better with muscle force than raw amplitude?
4. Why is hysteresis necessary in threshold-based gesture classification? What problems occur without it?
5. A servo motor draws 500mA at stall current but Arduino can only safely provide 40mA per pin. Explain why a separate power supply is required and how to wire it safely.

10.11 Guided Experiments

EMG Feature Comparison: MAV vs RMS vs WL

Hypothesis: Different EMG features (Mean Absolute Value, Root Mean Square, Waveform Length) will show different sensitivities to muscle contraction intensity, with RMS correlating most strongly with force due to its power-based calculation.

Procedure:

1. Collect 50 EMG samples each for three contraction levels: rest (0%), light (30%), strong (70% MVC)
2. For each window of 50 samples, compute:
 - $MAV = \frac{1}{N} \sum |x_i|$

- $\text{RMS} = \sqrt{\frac{1}{N} \sum x_i^2}$
- $\text{WL} = \sum |x_{i+1} - x_i|$

3. Record results in table below

4. Calculate coefficient of variation (CV = std/mean) for each feature at each level

Data Collection:

Feature	Rest	Light (30%)	Strong (70%)
MAV			
RMS			
WL			
MAV CV			
RMS CV			
WL CV			

Analysis Questions:

- Which feature shows the greatest dynamic range between rest and strong contraction?
- Which feature has the lowest coefficient of variation (most consistent)?
- Does RMS increase more than MAV as hypothesized? Why might this be?
- Which feature would you choose for a simple threshold-based classifier? Why?

Window Size Effect on Signal Smoothing

Hypothesis: Larger window sizes will produce smoother EMG envelopes but introduce greater delay in detecting changes in muscle activation state.

Procedure:

1. Configure moving average filter with window size = 10 samples (20ms at 500Hz)
2. Collect EMG during rapid flex-relax cycle: rest (2s) → contract (2s) → rest (2s)
3. Measure time from start of contraction to filtered signal crossing 50% of max
4. Repeat experiment with window sizes: 25, 50, 100, 200 samples
5. Record detection delay and signal noise (std during rest)

Data Collection:

Window Size	Time (ms)	Detection Delay (ms)	Noise (std)
10	20		
25	50		
50	100		
100	200		
200	400		

Analysis Questions:

- Plot detection delay vs window size. Is the relationship linear?
- Plot noise vs window size. What is the noise reduction factor for doubling window size?
- For a prosthetic hand requiring <100ms response, what is the maximum acceptable window size?
- How would you choose window size for a fatigue-monitoring system vs a prosthetic control?

Related Labs

Prerequisites:

- **LAB02 - Neural Network Training Fundamentals:** Understanding of neural network architectures and training is essential for building EMG classification models.
- **LAB05 - Embedded Systems Deployment:** Knowledge of TensorFlow Lite deployment is needed to run inference on edge devices for real-time gesture recognition.

Next Steps:

- **LAB11 - Profiling and Optimization:** Apply profiling techniques to optimize EMG models for real-time inference on resource-constrained medical devices.

10.12 Exercises

[Intermediate] Intermediate: Exercise 1: Signal Analysis

Collect 30 seconds of EMG data for each state (rest, light flex, strong flex):

1. Log raw and filtered values to Serial
2. Import to Python and plot histograms
3. Calculate optimal thresholds from data distributions
4. Verify classification accuracy on held-out data

[Intermediate] Intermediate: Exercise 2: Feature Comparison

Compare classification accuracy using different feature sets:

1. RMS only (single feature)
2. RMS + MAV (two features)
3. All five features from Table 10.2

Which combination gives best accuracy with lowest complexity?

[Challenge] Challenge: Exercise 3: Fatigue Compensation

During extended use, muscle fatigue reduces EMG amplitude:

1. Collect data at start and after 5 minutes of repeated gestures
2. Measure how thresholds should change
3. Implement adaptive threshold adjustment
4. Test robustness over 10-minute sessions

[Challenge] Challenge: Exercise 4: Multi-Gesture System

Extend the system to recognize 4 gestures:

1. Rest (no activation)
2. Wrist flexion
3. Wrist extension
4. Grip (fingers closed)

Hint: May require multiple electrode channels or time-domain patterns.

- **Further Reading.**
- MyoWare Muscle Sensor Guide: <https://learn.sparkfun.com/tutorials/myoware-muscle-sensor-kit>
 - OpenBCI EMG Processing: <https://docs.openbci.com/>
 - InMoov Open-Source Prosthetic: <https://inmoov.fr/>
 - EMG Signal Processing Review: De Luca, C.J. “The Use of Surface Electromyography in Biomechanics”

— **Credits.** This chapter draws on biomedical signal processing principles and open-source prosthetics projects. The InMoov project provides excellent resources for 3D-printed prosthetics.

Case Study: Open-Source Prosthetic Hand with EMG Control

Context: A university research lab partnered with a nonprofit to develop low-cost prosthetic hands for amputees in developing countries. Traditional myoelectric prosthetics cost \$20,000-\$80,000, making them inaccessible to 95% of global amputees. The goal was to create a 3D-printable prosthetic hand controlled by EMG sensors that cost under \$500 in materials and could be fabricated locally. The hand needed to recognize 5 gestures (rest, grip, pinch, point, thumbs-up) with >90% accuracy and <200ms response latency.

Challenge: Initial prototypes using commercial EMG sensors and Raspberry Pi achieved only 73% gesture accuracy in controlled testing and degraded to 45% accuracy during real-world use. Major issues included: (1) electrode placement inconsistency between users and across sessions, (2) signal drift from electrode impedance changes due to sweat and movement, (3) inter-user variability—models trained on one person failed on others, (4) 350ms latency from sensor to actuation felt unnatural. The Raspberry Pi solution was also too bulky (85mm x 56mm) and power-hungry (2.5W) to integrate into a prosthetic socket.

Solution: The team redesigned the system using edge ML techniques from this lab:

- Switched to Arduino Nano 33 BLE Sense with custom 4-channel EMG frontend (35mm x 18mm, 150mW)
- Implemented robust feature extraction: RMS, MAV, zero-crossing rate, and waveform length computed in 200ms windows with 50ms overlap
- Trained personalized models: each user performed 10-minute calibration generating 300 training samples per gesture
- Applied transfer learning: base model pre-trained on 50 users, fine-tuned per-user with 50 samples in <2 minutes
- Implemented adaptive thresholding: automatically adjust to account for electrode impedance drift every 5 minutes
- Deployed TFLite Micro int8 quantized CNN (12KB model size, 18ms inference)
- Added gesture smoothing: require 3 consecutive predictions to change state, reducing false activations

Results:

- Gesture accuracy: 73% → 92.3% in controlled testing, 45% → 86.7% in real-world use
- Response latency: 350ms → 165ms (sensor sampling + feature extraction + inference + actuation)
- Cross-user generalization: Transfer learning reduced per-user calibration from 30

minutes to 2 minutes

- System cost: \$2,400 (Raspberry Pi) → \$380 (Arduino + custom EMG board + servo motors + 3D printed parts)
- Power consumption: 2.5W → 0.15W (enables 12-hour battery life from 2000mAh pack)
- Form factor: Fully integrated into prosthetic socket (35mm x 18mm board fits in forearm cuff)

Lessons Learned:

- Personalized calibration is essential for EMG—biological variability between users is too high for universal models
- Transfer learning dramatically reduces calibration burden: pre-train on diverse users, fine-tune per-user with minimal data
- Feature engineering matters more than model complexity: well-designed time-domain features (RMS, MAV, ZCR) outperform raw signal CNNs for EMG
- Electrode placement training is critical—provide 3D-printed electrode holders and visual guides to ensure consistency
- Temporal smoothing prevents false activations: single misclassifications are common, require consecutive predictions for state changes
- Real-world testing reveals issues invisible in lab: sweat, temperature, muscle fatigue, and motion artifacts dominate failure modes
- Latency perception threshold is 200ms—users report unnaturalness above this, even with 95%+ accuracy

10.13 Exercises

1. **[Challenge] Challenge: Multi-Channel EMG Classification:** Extend the EMG system to use two sensor channels (bicep and tricep). Extract features from both channels and train an ML classifier to recognize 5 distinct gestures: rest, bicep contraction, tricep contraction, both contracted, and finger grip. Compare accuracy against single-channel detection and analyze which gesture pairs benefit most from dual-channel sensing.
2. **[Challenge] Challenge: Adaptive Threshold Calibration:** Implement an automatic calibration routine that adjusts detection thresholds based on user-specific signal characteristics. Collect 30 seconds of maximum voluntary contraction (MVC) and rest data, compute percentile-based thresholds (e.g., 20th percentile for rest, 80th percentile for activation), and test robustness across multiple sessions. Add a recalibration prompt when signal drift is detected.
3. **[Challenge] Challenge: Real-Time Frequency Domain Analysis:** Implement FFT-based frequency analysis to detect muscle fatigue during extended use. Extract the median frequency from 1-second windows of EMG data and track how it decreases over time during sustained contractions. Design a fatigue indicator that warns users when signal quality degrades below reliable detection levels, and test how accuracy changes after 1, 5, and 10 minutes of continuous use.



11. Edge Device Profiling and Optimization

Learning Objectives

After completing this lab, you will be able to:

- Measure execution time using millisecond and microsecond timers
- Profile memory usage including SRAM, Flash, and stack consumption
- Measure power consumption using current sensors and multimeters
- Profile TensorFlow Lite Micro inference performance
- Compare Float32 vs Int8 quantized model efficiency
- Apply targeted optimizations based on profiling results

Interactive Notebook

Jupyter Notebook: LAB11_profiling.ipynb

[Open in Google Colab](#) | [View on GitHub](#) | [Companion Website](#)

Prerequisites

Before starting this lab, ensure you:

- Completed LAB05 (Embedded Systems Deployment): TensorFlow Lite Micro basics
- Completed LAB08 (Arduino Sensors): microcontroller programming and sensor interfacing
- Understanding of basic statistics (mean, standard deviation)
- Familiarity with memory concepts (RAM vs Flash)

Hardware Requirements

- **Level 1:** Laptop with TensorFlow for model analysis
- **Level 2:** Arduino Nano 33 BLE or ESP32 for timing profiles
- **Level 3:** INA219 current sensor or USB power meter

⚠ Measuring First Run (Includes Cache Warmup)

The first inference run is often 2-10x slower than subsequent runs due to cache misses, lazy initialization, and JIT compilation effects. If you report only the first measurement, you'll vastly overestimate latency. **Always discard the first 1-5 runs** as warmup before collecting timing statistics. Better yet, report both cold-start and warm latency.

■ **Theory and practice.** Profiling transforms guesswork into science. Without measurements, optimization is speculation. This chapter teaches systematic measurement of the three critical edge constraints: memory, latency, and power. You are encouraged to reuse models and code from LAB03 (quantization), LAB04 (keyword spotting), LAB10 (EMG control), and other application labs so that the numbers you collect directly inform real deployments. The insights you gain here will feed into energy-focused design decisions in LAB15.

Execution tiers. Level 1: Simulate timing and resource estimates in notebooks. Level 2: Profile real code via serial output on development boards. Level 3: Connect hardware power monitors for comprehensive profiling of deployed systems.

Three-Tier Execution Model

Tier 1: Jupyter/Colab

Estimate model complexity with FLOPs and parameter counts, calculate theoretical memory requirements, simulate inference timing based on architecture

Tier 2: Arduino/ESP32
Measure real inference latency with micros(), profile SRAM and Flash usage, benchmark float32 vs int8 model performance

Tier 3: INA219/Power Meter
Measure actual power consumption during inference, calculate energy per inference, estimate battery life for deployment scenarios

11.1 Why Profiling Matters

Edge devices have severe constraints compared to desktop machines:

Profiling Answers Critical Questions

1. **Will my model fit?** Memory profiling reveals SRAM and Flash requirements
2. **Is inference fast enough?** Timing profiling measures latency
3. **How long will the battery last?** Power profiling calculates energy budget
4. **Where are the bottlenecks?** Detailed profiling identifies optimization targets

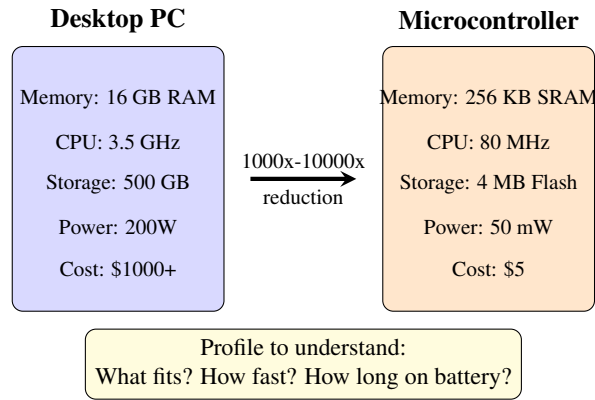


Figure 11.1: The edge computing constraint gap: Edge devices have 1000-10000x less memory, speed, and power than desktop machines. Profiling determines if your application fits these constraints.

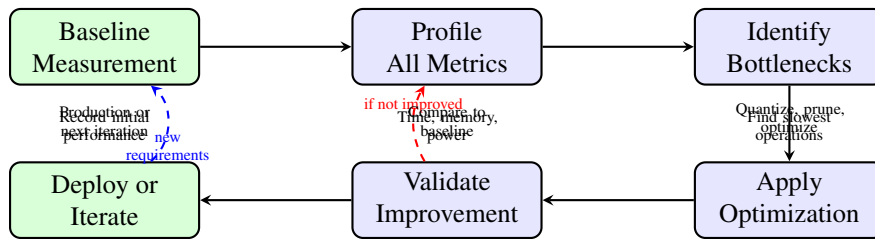


Figure 11.2: Profiling workflow: systematic measurement drives optimization decisions. Always establish a baseline before optimizing, then iterate until requirements are met.

11.2 Computational Complexity Foundations

Before diving into practical profiling, we need mathematical tools to predict performance and understand trade-offs. This section provides the analytical framework for estimating computational cost, memory requirements, and energy consumption.

11.2.1 FLOPs: Counting Floating-Point Operations

FLOPs (Floating-Point Operations) quantify computational cost. Each neural network layer has a predictable FLOPs formula based on its dimensions.

Definition — FLOPs for Convolutional Layer. For a 2D convolution with:

- Input: C_{in} channels, $H_{in} \times W_{in}$ spatial dimensions
- Kernel: $K \times K$ filter size
- Output: C_{out} channels, $H_{out} \times W_{out}$ spatial dimensions

The FLOPs are:

$$\text{FLOPs}_{\text{conv}} = 2 \cdot K^2 \cdot C_{in} \cdot C_{out} \cdot H_{out} \cdot W_{out}$$

The factor of 2 accounts for multiply-add operations (MAC): each position requires $K^2 \cdot C_{in}$ multiplications and additions.

Definition — FLOPs for Dense (Fully Connected) Layer. For a dense layer with N_{in} input

neurons and N_{out} output neurons:

$$\text{FLOPs}_{\text{dense}} = 2 \cdot N_{in} \cdot N_{out}$$

Each output neuron computes a dot product with all N_{in} inputs (multiply-add operations).

■ **Example — FLOPs Calculation for a Small CNN.** Consider a CNN for MNIST digit recognition:

Architecture:

1. Input: $28 \times 28 \times 1$ grayscale image
2. Conv1: 16 filters, 3×3 kernel, output: $26 \times 26 \times 16$
3. MaxPool: 2×2 , output: $13 \times 13 \times 16$
4. Conv2: 32 filters, 3×3 kernel, output: $11 \times 11 \times 32$
5. MaxPool: 2×2 , output: $5 \times 5 \times 32$
6. Flatten: $5 \times 5 \times 32 = 800$ neurons
7. Dense1: 128 neurons
8. Dense2 (output): 10 neurons

FLOPs per layer:

$$\text{Conv1: } 2 \cdot 3^2 \cdot 1 \cdot 16 \cdot 26 \cdot 26 = 194,688$$

$$\text{Conv2: } 2 \cdot 3^2 \cdot 16 \cdot 32 \cdot 11 \cdot 11 = 1,397,952$$

$$\text{Dense1: } 2 \cdot 800 \cdot 128 = 204,800$$

$$\text{Dense2: } 2 \cdot 128 \cdot 10 = 2,560$$

$$\text{Total: } 1,800,000 \approx 1.8 \text{ MFLOPs per inference}$$

Estimating inference time: If an Arduino Nano 33 BLE performs ~ 10 MFLOPs/s for Int8 operations:

$$\text{Latency} \approx \frac{1.8 \text{ MFLOPs}}{10 \text{ MFLOPs/s}} = 0.18 \text{ seconds} = 180 \text{ ms}$$

This is a rough estimate; memory bandwidth and overhead add additional time. ■

Why FLOPs Matter

- **Model comparison:** Compare architectures before training
- **Hardware selection:** Choose devices with sufficient compute capability
- **Real-time feasibility:** Estimate if latency meets requirements
- **Optimization target:** Identify compute-heavy layers for pruning/quantization

⚠ FLOPs Don't Tell the Whole Story

FLOPs measure *compute* but ignore:

- **Memory bandwidth:** Loading weights and activations from memory
- **Cache effects:** Small models run faster due to better locality
- **Instruction overhead:** Loop control, indexing, branching
- **Quantization:** Int8 operations are faster than Float32, but FLOPs count is the same

Always validate FLOPs estimates with actual profiling!

11.2.2 Amdahl's Law: Understanding Speedup Limits

Amdahl's Law quantifies the maximum speedup achievable when optimizing only part of a system.

Theorem — Amdahl's Law. Let p be the fraction of execution time that can be parallelized/optimized, and s be the speedup factor for that portion. The overall speedup S is:

$$S = \frac{1}{(1-p) + \frac{p}{s}}$$

Key insight: Even if you make the optimizable portion infinitely fast ($s \rightarrow \infty$), the maximum speedup is:

$$S_{\max} = \frac{1}{1-p}$$

■ **Example — Parallelizing Inference Pipeline.** Your edge device spends:

- 60% of time in matrix multiplication (parallelizable)
- 30% in memory transfers (not parallelizable)
- 10% in activation functions (parallelizable)

If you parallelize matrix multiplication and activations with 4 cores ($s = 4$):

$$p = 0.60 + 0.10 = 0.70 \quad (70\% \text{ parallelizable})$$

$$S = \frac{1}{(1-0.70) + \frac{0.70}{4}} = \frac{1}{0.30 + 0.175} = \frac{1}{0.475} \approx 2.1\times$$

Even with perfect $4\times$ speedup on 70% of the code, overall speedup is only $2.1\times$. The 30% memory transfer bottleneck limits gains.

Maximum possible speedup (if parallelization were perfect, $s \rightarrow \infty$):

$$S_{\max} = \frac{1}{1-0.70} = \frac{1}{0.30} = 3.33\times$$

■

Implications for Edge Optimization

- **Bottleneck identification is critical:** Optimizing non-bottleneck code yields minimal gains
- **Sequential portions dominate:** Memory I/O and initialization limit speedup
- **Focus on the 80/20 rule:** Profile first, then optimize the slowest 20%
- **Quantization effectiveness:** If only 50% of time is in compute, $4\times$ faster compute yields only $1.6\times$ total speedup

11.2.3 Memory Bandwidth and Compute-Memory Balance

Modern processors are often *memory-bound* rather than compute-bound. Understanding memory bandwidth is crucial for edge optimization.

Definition — Memory Bandwidth. Memory bandwidth BW is the rate at which data can be transferred between memory and processor:

$$BW = \text{frequency} \times \text{bus width} \times \text{efficiency}$$

Typical values:

- **DDR4 RAM:** 25-40 GB/s
- **Microcontroller SRAM:** 0.5-2 GB/s
- **L1 Cache:** 50-200 GB/s

Definition — Arithmetic Intensity. Arithmetic intensity measures compute vs memory traffic:

$$I = \frac{\text{FLOPs}}{\text{bytes transferred}}$$

Classification:

- $I < 1$: **Memory-bound** (more time moving data than computing)
- $I > 10$: **Compute-bound** (computation dominates)

■ **Example — Compute vs Memory Bound Analysis.** Scenario: Dense layer with $N_{in} = 1024$, $N_{out} = 256$.

Compute cost:

$$\text{FLOPs} = 2 \cdot 1024 \cdot 256 = 524,288$$

Memory cost (Float32):

- Weights: $1024 \times 256 \times 4 = 1,048,576$ bytes
- Input: $1024 \times 4 = 4,096$ bytes
- Output: $256 \times 4 = 1,024$ bytes
- Total: $\approx 1,053,696$ bytes

Arithmetic intensity:

$$I = \frac{524,288 \text{ FLOPs}}{1,053,696 \text{ bytes}} \approx 0.5 \text{ FLOPs/byte}$$

This is **memory-bound**! With SRAM bandwidth of 1 GB/s:

$$\text{Memory time} = \frac{1.05 \text{ MB}}{1 \text{ GB/s}} = 1.05 \text{ ms}$$

With compute rate of 10 MFLOPs/s:

$$\text{Compute time} = \frac{0.524 \text{ MFLOPs}}{10 \text{ MFLOPs/s}} = 0.052 \text{ ms}$$

Conclusion: Memory transfer takes 20× longer than computation! Optimizing compute (e.g., faster processor) won't help. Instead:

- Use Int8 quantization: Reduces memory traffic 4×
- Store weights in cache-friendly layout
- Batch multiple inferences to amortize weight loading

■

Roofline Model Insight

The *roofline model* visualizes performance limits:

- **Flat roof:** Compute-bound operations limited by peak FLOPs/s
- **Sloped roof:** Memory-bound operations limited by bandwidth
- **Edge devices:** Most ML operations hit the memory bandwidth roof

Optimization strategies depend on the bottleneck:

- If compute-bound: Quantization, pruning, smaller models
- If memory-bound: Data layout optimization, caching, tiling

11.2.4 Energy Modeling for Battery-Powered Devices

Energy consumption determines battery life—often the hardest constraint for edge devices.

Definition — Dynamic Power Consumption. Dynamic power (power consumed during active computation):

$$P_{\text{dyn}} = \alpha \cdot C \cdot V^2 \cdot f$$

Where:

- α : Activity factor (fraction of circuits switching, typically 0.1-0.3)
- C : Capacitance (hardware-dependent)
- V : Supply voltage (e.g., 3.3V for microcontrollers)
- f : Clock frequency (e.g., 64 MHz)

Key insight: Power scales *quadratically* with voltage and linearly with frequency. Reducing clock speed saves energy!

Definition — Energy per Inference. Total energy for one inference:

$$E = P_{\text{avg}} \cdot t_{\text{inference}}$$

Where P_{avg} includes dynamic power, static power (leakage), and peripheral power (sensors, WiFi).

Table 11.1: Energy per operation for different hardware platforms (approximate values)

Operation	MCU (Int8)	Desktop CPU (Float32)	GPU (Float32)
MAC (multiply-add)	0.2 nJ	2 nJ	0.05 nJ
Memory read (1 KB)	5 nJ	20 nJ	1 nJ
Cache access (1 KB)	0.5 nJ	2 nJ	0.2 nJ
WiFi transmission (1 packet)	1000 nJ	—	—
Deep sleep (1 second)	0.01 mJ	—	—

■ **Example — Battery Life Calculation with Duty Cycling.** Device: Arduino Nano 33 BLE running keyword spotting

Measured power consumption:

- Deep sleep: 0.01 mA @ 3.3V = 0.033 mW
- Inference: 80 mA @ 3.3V = 264 mW
- Inference time: 12 ms

Duty cycle: Inference every 1 second

Energy per cycle:

$$E_{\text{inference}} = 264 \text{ mW} \times 0.012 \text{ s} = 3.168 \text{ mJ}$$

$$E_{\text{sleep}} = 0.033 \text{ mW} \times 0.988 \text{ s} = 0.033 \text{ mJ}$$

$$E_{\text{total}} = 3.168 + 0.033 = 3.20 \text{ mJ per cycle}$$

Average power:

$$P_{\text{avg}} = \frac{3.20 \text{ mJ}}{1 \text{ s}} = 3.20 \text{ mW} = 0.97 \text{ mA @ 3.3V}$$

Battery life with 1000 mAh battery:

$$\text{Life} = \frac{1000 \text{ mAh}}{0.97 \text{ mA}} \approx 1030 \text{ hours} = 43 \text{ days}$$

Without duty cycling (continuous inference):

$$\text{Life} = \frac{1000 \text{ mAh}}{80 \text{ mA}} = 12.5 \text{ hours}$$

Conclusion: Duty cycling provides 82× longer battery life! ■

Energy Optimization Strategies

Ordered by impact:

1. **Duty cycling:** Sleep between inferences (10-100× energy reduction)
2. **Quantization:** Int8 vs Float32 (2-4× energy reduction)
3. **Model compression:** Smaller models = less memory traffic
4. **Voltage/frequency scaling:** Lower clock speed when possible
5. **Peripheral management:** Turn off WiFi, BLE when not needed

11.2.5 Latency Breakdown and System-Level Analysis

End-to-end latency includes more than just computation. Understanding the full pipeline helps identify hidden bottlenecks.

Definition — Total Latency Decomposition.

$$T_{\text{total}} = T_{\text{data}} + T_{\text{compute}} + T_{\text{memory}} + T_{\text{overhead}}$$

Where:

- T_{data} : Data acquisition (sensor read, I/O)
- T_{compute} : Actual computation (FLOPs)
- T_{memory} : Memory transfers (weights, activations)
- T_{overhead} : Framework overhead, function calls, scheduling

■ **Example — Latency Breakdown for Image Classification.** System: ESP32-CAM running person detection

Measured latency components:

Component	Time (ms)	% Total
Camera capture (96×96 RGB)	45	5.0%
Preprocessing (resize, normalize)	32	3.6%
Weight loading from Flash	120	13.3%
Inference (compute)	650	72.2%
Postprocessing (NMS, thresholding)	28	3.1%
Result transmission (WiFi)	25	2.8%
Total	900	100%

Analysis:

- Compute dominates (72%), making this *compute-bound*
- Weight loading (13%) is significant due to Flash memory speed
- Camera capture overhead (5%) limits frame rate even if inference speeds up

Optimization priorities:

1. Quantize model to reduce compute time (primary bottleneck)
2. Cache weights in SRAM if space permits (eliminate Flash reads)
3. Camera timing cannot be improved without hardware change

Expected improvement with Int8 quantization:

$$T_{\text{compute, new}} = \frac{650}{4} = 162.5 \text{ ms}$$

$$T_{\text{total, new}} = 45 + 32 + 120 + 162.5 + 28 + 25 = 412.5 \text{ ms}$$

$$\text{Speedup} = \frac{900}{412.5} \approx 2.2\times$$

Note: Speedup is less than 4× because only 72% of time was in compute (Amdahl's Law!). ■

⚠ Hidden Overheads in ML Inference

Profiling often reveals unexpected bottlenecks:

- **Framework initialization:** TFLite Micro setup can take 50-200 ms
- **Memory allocation:** Dynamic allocation on embedded systems is slow
- **Data marshaling:** Converting between formats (uint8 ↔ float)
- **Cache misses:** First inference slower than subsequent runs
- **Interrupt handling:** Unpredictable delays from sensors, timers

Always profile the *full pipeline*, not just model inference!

System-Level Performance Modeling

To predict real-world performance:

1. **Measure each component** separately (data, compute, memory, overhead)
2. **Identify the bottleneck** using profiling data
3. **Apply Amdahl's Law** to predict optimization impact
4. **Validate with end-to-end testing** on target hardware

Rule of thumb: If profiling shows no single component >50% of time, system-level optimizations (pipelining, caching, batching) may help more than algorithmic changes.

11.3 Measuring Execution Time**11.3.1 Basic Timing with millis() and micros()**

Arduino provides two timing functions:

```

1 // timing_demo.ino - Measure code execution time
2 void setup() {
3     Serial.begin(115200);
4 }
5
6 void loop() {
7     // millis() for longer durations (millisecond resolution)
8     unsigned long start_ms = millis();
9     delay(100); // Simulated workload
10    unsigned long elapsed_ms = millis() - start_ms;
11
12    Serial.print("Duration (ms): ");
13    Serial.println(elapsed_ms);

```

Code Walkthrough

millis(): Returns milliseconds since boot. Resolution of 1ms. Use for operations taking 10ms or longer.

Overflow: millis() overflows after ~49 days. Subtraction handles this correctly due to unsigned arithmetic.

```

1 // micros() for short durations (microsecond resolution)
2 unsigned long start_us = micros();
3
4 // Fast computation to measure
5 volatile float x = 0;
6 for (int i = 0; i < 1000; i++) {
7     x += sin(i * 0.01);
8 }
9
10 unsigned long elapsed_us = micros() - start_us;
11
12 Serial.print("Duration (us): ");
13 Serial.println(elapsed_us);
14 Serial.print("Result: ");
15 Serial.println(x); // Prevent optimizer removal
16
17 delay(1000);
18 }

```

Code Walkthrough

micros(): Returns microseconds since boot. Resolution of 4 μ s on most boards. Use for fast operations.

volatile keyword: Prevents the compiler from optimizing away the computation, ensuring accurate measurement.

Expected Output:

```

Duration (ms): 100
Duration (us): 4523
Result: 9.31

```

11.3.2 Statistical Benchmarking**⚙️ Expected Timing Resolution**

Different boards have different timing capabilities:

- **Arduino Uno/Nano:** micros() resolution: 4 microseconds
- **ESP32:** micros() resolution: 1 microsecond
- **Raspberry Pi Pico:** Sub-microsecond resolution available
- **Desktop:** Nanosecond resolution (clock_gettime)

For operations taking less than 100 μ s, use higher-resolution boards or average over many iterations.

Single measurements are noisy. Run multiple trials and compute statistics:

```

1 #define NUM_TRIALS 100
2 unsigned long measurements[NUM_TRIALS];
3
4 void runBenchmark(const char* name, void (*func)()) {
5     Serial.print("\n=== Benchmarking: ");
6     Serial.println(name);
7

```

```

8 // Warm-up run (eliminates cache effects)
9 func();
10
11 // Collect measurements
12 for (int i = 0; i < NUM_TRIALS; i++) {
13     unsigned long start = micros();
14     func();
15     measurements[i] = micros() - start;
16 }

```

Code Walkthrough

Function pointer: Allows passing any workload function to the benchmarking code.

Warm-up run: The first execution may be slower due to cache misses. Warm-up ensures consistent measurements.

```

1 // Calculate statistics
2 unsigned long sum = 0, minVal = ULONG_MAX, maxVal = 0;
3 for (int i = 0; i < NUM_TRIALS; i++) {
4     sum += measurements[i];
5     if (measurements[i] < minVal) minVal = measurements[i];
6     if (measurements[i] > maxVal) maxVal = measurements[i];
7 }
8 float mean = (float)sum / NUM_TRIALS;
9
10 // Standard deviation
11 float variance = 0;
12 for (int i = 0; i < NUM_TRIALS; i++) {
13     float diff = measurements[i] - mean;
14     variance += diff * diff;
15 }
16 float stddev = sqrt(variance / NUM_TRIALS);
17
18 // Report results
19 Serial.print(" Mean: "); Serial.print(mean, 1); Serial.println(" us");
20 Serial.print(" Std Dev: "); Serial.print(stddev, 1); Serial.println(" us");
21 ;
22 Serial.print(" Min/Max: "); Serial.print(minVal);
23 Serial.print("/"); Serial.print(maxVal); Serial.println(" us");
24 Serial.print(" Throughput: ");
25 Serial.print(1000000.0 / mean, 1); Serial.println(" ops/sec");

```

Reducing Measurement Variance

High standard deviation indicates inconsistent timing. To reduce variance:

- **Disable interrupts:** `noInterrupts()` during measurement
- **Avoid dynamic allocation:** Use static buffers
- **Stabilize power:** Consistent voltage reduces clock jitter
- **Run more trials:** 100+ trials for statistical significance

11.4 Measuring Memory Usage

11.4.1 Compile-Time Memory Report

Arduino IDE reports memory usage after compilation:

```

Sketch uses 24580 bytes (7%) of program storage space.
Maximum is 327680 bytes.

```

Global variables use 1284 bytes (15%) of dynamic memory, leaving 6764 bytes for local variables. Maximum is 8192 bytes.

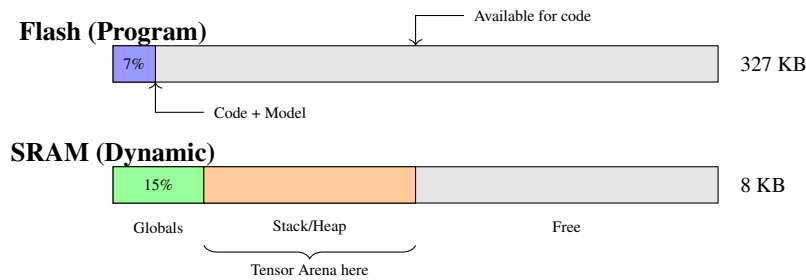


Figure 11.3: Memory layout: Flash stores code and model weights (persistent). SRAM stores variables, stack, heap, and tensor arena (volatile). The tensor arena for ML must fit in free SRAM.

ML Model Memory Components

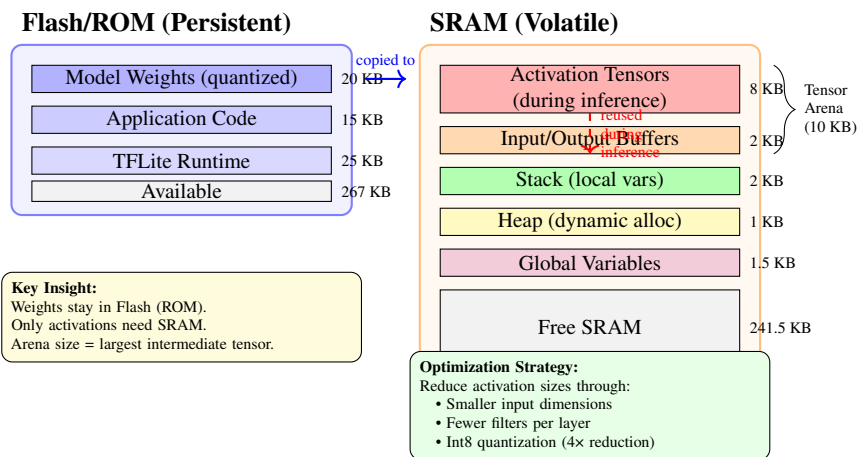


Figure 11.4: Detailed ML model memory layout showing how different components use Flash vs SRAM. Model weights are stored in Flash (persistent), while activations and intermediate tensors require SRAM (volatile) during inference. The tensor arena size determines the minimum SRAM requirement.

11.4.2 Runtime Memory Measurement

```

1 // For ESP32 boards
2 #ifdef ESP32
3 void printMemoryReport() {
4     Serial.println("\n=== Memory Report ===");
5     Serial.print("Free Heap: ");
6     Serial.print(ESP.getFreeHeap() / 1024.0, 1);
7     Serial.println(" KB");
8
9     Serial.print("Minimum Free Heap: ");
10    Serial.print(ESP.getMinFreeHeap() / 1024.0, 1);
11    Serial.println(" KB");
12
13    Serial.print("Total Heap Size: ");
14    Serial.print(ESP.getHeapSize() / 1024.0, 1);

```

```

15     Serial.println(" KB");
16
17     // PSRAM if available
18     if (ESP.getPsramSize() > 0) {
19         Serial.print("PSRAM: ");
20         Serial.print(ESP.getPsramSize() / 1024 / 1024.0, 1);
21         Serial.println(" MB");
22     }
23 }
24 #endif

```

Code Walkthrough

getFreeHeap(): Current free memory—changes during execution.

getMinFreeHeap(): Minimum free memory since boot—shows peak usage.

PSRAM: External RAM on some ESP32 boards (4-8 MB)—useful for large models.

```

1 // For AVR boards (Uno, Nano, Mega)
2 #ifdef __AVR__
3 extern int __heap_start, *__brkval;
4
5 int freeMemory() {
6     int v;
7     return (int)&v - (__brkval == 0 ?
8         (int)&__heap_start : (int)__brkval);
9 }
10 #endif
11
12 // For ARM Cortex-M boards (Arduino Nano 33 BLE, etc.)
13 #ifdef ARDUINO_ARCH_MBED
14 extern "C" char* sbrk(int incr);
15 int freeMemory() {
16     char top;
17     return &top - reinterpret_cast<char*>(sbrk(0));
18 }
19 #endif
20
21 void setup() {
22     Serial.begin(115200);
23     while (!Serial) delay(10); // Wait for serial connection
24
25     Serial.println("=== Memory Profiling Demo ===");
26
27     #if defined(ESP32)
28     printMemoryReport();
29     #else
30     Serial.print("Initial Free Memory: ");
31     Serial.print(freeMemory());
32     Serial.println(" bytes");
33     #endif
34
35     // Simulate tensor arena allocation
36     const int ARENA_SIZE = 4096;
37     static uint8_t tensor_arena[ARENA_SIZE];
38     memset(tensor_arena, 0, sizeof(tensor_arena));
39
40     Serial.print("\nAfter allocating ");
41     Serial.print(ARENA_SIZE);
42     Serial.println(" byte tensor arena:");
43

```

```

44  #if defined(ESP32)
45  printMemoryReport();
46  #else
47  Serial.print("Free Memory: ");
48  Serial.print(freeMemory());
49  Serial.println(" bytes");
50  #endif
51 }

```

Typical Memory Usage for TFLite Micro

Expected memory consumption for common models:

- **Tiny MLP (784-32-10):** ~25 KB total (2 KB arena)
- **Small CNN:** ~40 KB total (8-12 KB arena)
- **Keyword Spotting:** ~20 KB model + 10 KB arena
- **Image Classification:** ~80 KB model + 15-30 KB arena

If your Arduino Nano 33 BLE (256 KB SRAM) runs out of memory, try:

1. Reduce tensor arena size in 1 KB increments
2. Use Int8 quantization (4× smaller)
3. Remove unused TensorFlow operations from compilation

11.5 Power Consumption Measurement

Power profiling is essential for battery-powered devices.

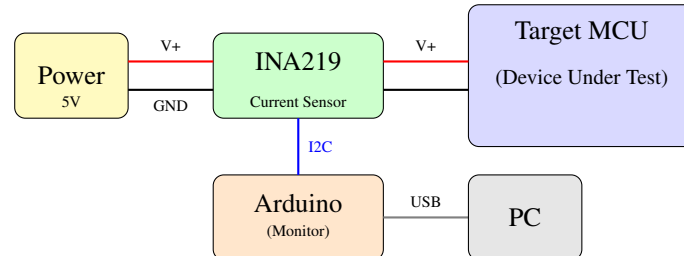


Figure 11.5: Power measurement setup: The INA219 current sensor is placed in series between the power supply and target device. A separate Arduino reads the sensor via I2C and reports to PC.

11.5.1 Using INA219 Current Sensor

```

1  #include <Wire.h>
2  #include <Adafruit_INA219.h>
3
4  Adafruit_INA219 ina219;
5
6  void setup() {
7      Serial.begin(115200);
8
9      if (!ina219.begin()) {
10         Serial.println("Failed to find INA219!");
11         while (1);
12     }
13
14     Serial.println("Time(ms) , Voltage(V) , Current(mA) , Power(mW)");

```

```

15 }
16
17 void loop() {
18     float voltage = ina219.getBusVoltage_V();
19     float current = ina219.getCurrent_mA();
20     float power = ina219.getPower_mW();
21
22     Serial.print(millis());
23     Serial.print(",");
24     Serial.print(voltage, 3);
25     Serial.print(",");
26     Serial.print(current, 2);
27     Serial.print(",");
28     Serial.println(power, 2);
29
30     delay(100); // 10 Hz sampling
31 }

```

Code Walkthrough

INA219: High-side current sensor with I2C interface. Measures voltage (0-26V) and current (up to 3.2A).

CSV output: Format for easy import into Python/Excel for analysis and plotting.

Wiring: Connect VIN+ to power source, VIN- to device, SDA/SCL to monitor Arduino.

Expected Output:

```

Time(ms),Voltage(V),Current(mA),Power(mW)
0,3.300,45.23,149.26
100,3.298,46.10,152.03
200,3.301,44.87,148.10
300,3.299,180.45,595.28  <- Inference spike
400,3.300,182.12,600.99
500,3.298,47.20,155.69  <- Back to idle

```

Typical Power Consumption

Expected current draw for common edge devices:

- **Arduino Nano 33 BLE (idle):** 5-8 mA @ 3.3V (16-26 mW)
- **Arduino Nano 33 BLE (active):** 15-25 mA @ 3.3V (50-82 mW)
- **Arduino Nano 33 BLE (inference):** 40-80 mA @ 3.3V (132-264 mW)
- **ESP32 (WiFi active):** 160-260 mA @ 3.3V (528-858 mW)
- **ESP32 (deep sleep):** 0.01-0.15 mA @ 3.3V (0.03-0.5 mW)
- **Raspberry Pi Zero W:** 100-200 mA @ 5V (500-1000 mW)

Inference typically causes 2-5× current spike for duration of forward pass.

11.5.2 Battery Life Estimation

```

1 void estimateBatteryLife(float avg_current_mA, int battery_mAh) {
2     // Simple estimation
3     float hours = battery_mAh / avg_current_mA;
4
5     Serial.println("\n=== Battery Life Estimate ===");
6     Serial.print("Average Current: ");
7     Serial.print(avg_current_mA, 2); Serial.println(" mA");
8     Serial.print("Battery Capacity: ");
9     Serial.print(battery_mAh); Serial.println(" mAh");

```

```

10 Serial.print("Estimated Life: ");
11 Serial.print(hours, 1); Serial.print(" hours (");
12 Serial.print(hours / 24, 1); Serial.println(" days)");

1 // With duty cycling
2 float active_current = 80.0; // mA when running
3 float sleep_current = 0.01; // mA in deep sleep
4 float active_time = 0.1; // seconds
5 float sleep_time = 60.0; // seconds
6
7 float effective_current =
8     (active_current * active_time + sleep_current * sleep_time) /
9     (active_time + sleep_time);
10
11 float duty_hours = battery_mAh / effective_current;
12
13 Serial.print("\nWith duty cycling: ");
14 Serial.print(effective_current, 3); Serial.println(" mA average");
15 Serial.print("Estimated Life: ");
16 Serial.print(duty_hours / 24, 0); Serial.println(" days");
17 }

```

Duty Cycling Impact

For a 2000 mAh battery:

- **Always active at 80 mA:** $2000/80 = 25$ hours
- **0.1s active + 60s sleep:** Average ≈ 0.13 mA $\rightarrow 15,000$ hours (1.7 years!)

Power optimization often matters more than speed optimization for edge devices.

11.6 Python Benchmarking for Model Comparison

Before deploying to hardware, benchmark models in Python to identify the best candidates:

Choosing the Right Profiling Tool

- **Quick function-level overview:** Use cProfile
- **Line-by-line bottleneck hunting:** Use line_profiler with @profile decorator
- **Memory issues:** Start with memory_profiler, use Valgrind for deep debugging
- **ML model optimization:** Use TensorFlow Profiler or PyTorch Profiler
- **Production systems:** Use py-spy (no code modification needed)
- **Hardware-level analysis:** Use perf on Linux systems

11.6.1 Latency Benchmarking Script

```

1 import time
2 import numpy as np
3 import tensorflow as tf
4
5 def benchmark_model(model, input_shape, num_warmup=10, num_runs=100):
6     """Benchmark TensorFlow model latency with proper warmup."""
7
8     # Create random input
9     test_input = np.random.randn(1, *input_shape).astype(np.float32)
10
11     # Warmup runs (JIT compilation, cache warming)

```


Table 11.2: Profiling tools comparison: choose the right tool for your measurement needs

Tool	Type	Metrics	Overhead		Best For
cProfile	CPU	Function call time, count	Medium	(5-10%)	Python function profiling, identifying hot functions
line_profiler	CPU	Line-by-line execution time	High	(20-50%)	Finding exact bottleneck lines within functions
memory_profiler	Memory	RAM usage per line	High	(10-100x slower)	Detecting memory leaks, tracking allocations
TF Profiler	ML Framework	Op execution, memory, device utilization	Low	(2-5%)	TensorFlow model optimization, GPU profiling
perf	System-level	CPU cycles, cache misses, instructions	Very Low	(<1%)	System-level analysis, hardware counters
py-spy	CPU	Sampling profiler	Very Low	(<1%)	Production profiling without code changes
Valgrind	Memory	Detailed allocation tracking	Very High	(10-50x slower)	Memory leak detection, undefined behavior

```

12     print(f"Warming up ({num_warmup} runs)...")
13     for _ in range(num_warmup):
14         _ = model.predict(test_input, verbose=0)
15
16     # Benchmark runs
17     print(f"Benchmarking ({num_runs} runs)...")
18     latencies = []
19     for i in range(num_runs):
20         start = time.perf_counter()
21         _ = model.predict(test_input, verbose=0)
22         end = time.perf_counter()
23         latencies.append((end - start) * 1000) # Convert to ms
24
25     # Calculate statistics
26     latencies = np.array(latencies)
27     return {
28         'mean': np.mean(latencies),
29         'std': np.std(latencies),
30         'min': np.min(latencies),
31         'max': np.max(latencies),
32         'p50': np.percentile(latencies, 50),
33         'p95': np.percentile(latencies, 95),
34         'p99': np.percentile(latencies, 99),
35         'throughput': 1000 / np.mean(latencies) # inferences/sec
36     }
37
38 def print_benchmark_results(name, results):
39     """Pretty-print benchmark results."""
40     print(f"\n=== {name} ===")
41     print(f"    Mean:      {results['mean']:>8.3f} ms")
42     print(f"    Std Dev:   {results['std']:>8.3f} ms")
43     print(f"    Min/Max:   {results['min']:>8.3f} / {results['max']:>8.3f} ms")
44     print(f"    Percentiles:")
45     print(f"        P50:   {results['p50']:>8.3f} ms")
46     print(f"        P95:   {results['p95']:>8.3f} ms")
47     print(f"        P99:   {results['p99']:>8.3f} ms")
48     print(f"    Throughput: {results['throughput']:>8.1f} fps")

```

```

1 # Example usage
2 model = tf.keras.models.load_model('my_model.h5')
3 results = benchmark_model(model, input_shape=(28, 28), num_runs=100)
4 print_benchmark_results("MNIST Classifier", results)

```

Expected Output:

```

=== MNIST Classifier ===
Mean:      1.234 ms
Std Dev:   0.089 ms
Min/Max:   1.102 / 1.567 ms
Percentiles:
  P50:     1.221 ms
  P95:     1.398 ms
  P99:     1.489 ms
Throughput: 810.4 fps

```

Estimating MCU Performance from Desktop Benchmarks

Desktop benchmarks run on CPU with optimizations. To estimate MCU performance:

- **Arduino Nano 33 BLE:** ~100-200× slower than desktop

- **ESP32:** $\sim 50\text{--}100\times$ slower than desktop
- **Raspberry Pi 4:** $\sim 5\text{--}10\times$ slower than desktop

Example: 1.2 ms on desktop $\rightarrow$ 120-240 ms on Arduino Nano 33 BLE.
Always verify with on-device testing!

11.6.2 TFLite Conversion and Size Comparison

```

1 def convert_and_measure(model, optimization=None):
2     """Convert to TFLite and measure size."""
3     converter = tf.lite.TFLiteConverter.from_keras_model(model)
4
5     if optimization == 'dynamic':
6         converter.optimizations = [tf.lite.Optimize.DEFAULT]
7     elif optimization == 'int8':
8         # Int8 quantization requires representative dataset
9         converter.optimizations = [tf.lite.Optimize.DEFAULT]
10        converter.representative_dataset = representative_data_gen
11        converter.target_spec.supported_ops = [
12            tf.lite.OpsSet.TFLITE_BUILTINS_INT8
13        ]
14        converter.inference_input_type = tf.int8
15        converter.inference_output_type = tf.int8
16
17        tflite_model = converter.convert()
18        return len(tflite_model) # Size in bytes
19
20 # Compare quantization options
21 original_size = model.count_params() * 4 # float32 = 4 bytes
22 dynamic_size = convert_and_measure(model, 'dynamic')
23 int8_size = convert_and_measure(model, 'int8')
24
25 print(f"Original (float32): {original_size/1024:.1f} KB")
26 print(f"Dynamic quant:      {dynamic_size/1024:.1f} KB")
27 print(f"Int8 quant:          {int8_size/1024:.1f} KB")
28 print(f"Reduction:           {original_size/int8_size:.1f}x")

```

11.7 ML Inference Profiling

11.7.1 TensorFlow Lite Micro Profiling

```

1 void profileInference(int num_runs) {
2     Serial.println("\n=== TFLite Inference Profile ===");
3
4     unsigned long times[100];
5     int actual_runs = min(num_runs, 100);
6
7     // Warm-up
8     interpreter->Invoke();
9
10    // Profile inference
11    for (int i = 0; i < actual_runs; i++) {
12        unsigned long start = micros();
13        interpreter->Invoke();
14        times[i] = micros() - start;
15    }
16
17    // Calculate statistics
18    unsigned long total = 0, minT = times[0], maxT = times[0];
19    for (int i = 0; i < actual_runs; i++) {

```

```

20     total += times[i];
21     if (times[i] < minT) minT = times[i];
22     if (times[i] > maxT) maxT = times[i];
23 }
24 float mean = (float)total / actual_runs;
25
26 Serial.print("Latency: "); Serial.print(mean/1000, 2);
27 Serial.println(" ms (mean)");
28 Serial.print("Throughput: "); Serial.print(1000000.0/mean, 1);
29 Serial.println(" inferences/sec");
30
31 // Arena usage
32 Serial.print("Arena: "); Serial.print(interpreter->arena_used_bytes());
33 Serial.print(" / "); Serial.print(kTensorArenaSize);
34 Serial.println(" bytes");
35 }

```

11.7.2 Float32 vs Int8 Comparison

Table 11.3: Comprehensive Float32 vs Int8 model comparison across different architectures

Metric	Model	Float32	Int8	Improvement
Model Size	MNIST MLP	84 KB	21 KB	4.0×
	Keyword Spot	18 KB	5 KB	3.6×
	MobileNetV1	16.9 MB	4.3 MB	3.9×
Tensor Arena	MNIST MLP	24 KB	8 KB	3.0×
	Keyword Spot	10 KB	4 KB	2.5×
	MobileNetV1	800 KB	300 KB	2.7×
Inference Time (ms)	MNIST MLP	5.2 ms	1.1 ms	4.7×
	Keyword Spot	45 ms	12 ms	3.8×
	Person Detect	N/A	850 ms	—
Energy/Inf.	MNIST MLP	1.04 mJ	0.22 mJ	4.7×
	Keyword Spot	9.0 mJ	2.4 mJ	3.8×
	Person Detect	N/A	170 mJ	—
Accuracy	MNIST MLP	98.7%	98.1%	-0.6%
	Keyword Spot	90.5%	89.8%	-0.7%
	MobileNetV1	70.9%	70.4%	-0.5%

Expected Inference Latency on Arduino Nano 33 BLE

Typical inference times for Int8 quantized models at 64 MHz:

- **Tiny MLP (784-32-10):** 1-2 ms
- **Small CNN (2 conv, 1 dense):** 8-15 ms
- **Keyword Spotting (DS-CNN):** 10-20 ms
- **Gesture Recognition:** 25-40 ms
- **Person Detection (96×96):** 800-1000 ms
- **Wake Word Detection:** 5-8 ms

Real-time requirements:

- Audio (16 kHz): Need <62.5 ms per frame (1000 samples)
- Gesture (50 Hz): Need <20 ms per inference
- Image (30 fps): Need <33 ms per frame

Profiling Before Optimizing

Always profile **before** applying optimizations:

1. Profile baseline model
2. Identify the actual bottleneck (memory? latency? power?)
3. Apply targeted optimization
4. Profile again to verify improvement

Don't assume—measure!

💡 Is $\pm 10\text{ms}$ Variance Acceptable?

Your gesture recognition system shows inference times of $45\text{ms} \pm 10\text{ms}$ (mean $\pm$ std). Consider:

1. If the application requires 30 FPS video processing (33ms per frame), can you use this model?
2. What if the worst-case timing (max, not mean) matters for safety-critical applications?
3. How would you design a system to handle variable inference times? (Hint: buffering, deadline scheduling)
4. At what point does variance become unacceptable vs. mean latency being too high?

Real-time systems care about worst-case, not average-case performance.

11.8 Optimization Guidelines

Based on profiling results, apply appropriate optimizations:

Table 11.4: Optimization strategies based on profiling results

Bottleneck	Symptom	Solutions
Memory (SRAM)	Arena allocation fails	Quantize model to Int8, reduce model size, use smaller tensors
Memory (Flash)	Sketch too large	Remove unused code, compress model, use PROGMEM
Latency	Inference > 100ms	Quantize, prune, use smaller input, simpler architecture
Power	Battery drains fast	Use deep sleep, reduce frequency, batch processing
Consistency	High timing variance	Disable interrupts, use static buffers, stable power

11.9 Complete Profiling Workflow Example

This section shows a complete end-to-end profiling workflow:

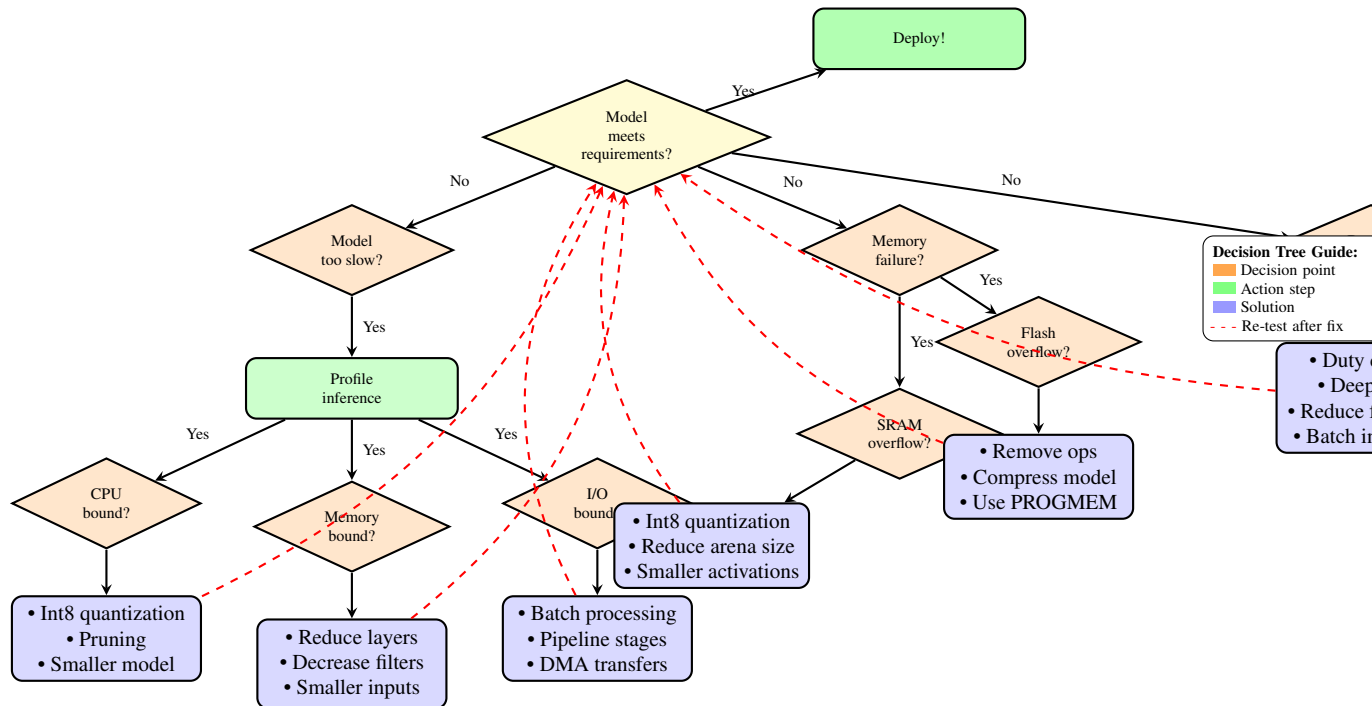


Figure 11.6: Optimization decision tree: systematic approach to identifying and fixing performance bottlenecks based on profiling results. Always re-test after applying optimizations to verify improvement.

```

1 #include <TensorFlowLite.h>
2 #include "model_data.h" // Your TFLite model
3
4 // TFLite globals
5 namespace {
6   const int kTensorArenaSize = 10 * 1024;
7   uint8_t tensor_arena[kTensorArenaSize];
8   tflite::MicroInterpreter* interpreter = nullptr;
9 }
10
11 // Memory tracking
12 #ifdef ARDUINO_ARCH_MBED
13 extern "C" char* sbrk(int incr);
14 int freeMemory() {
15   char top;
16   return &top - reinterpret_cast<char*>(sbrk(0));
17 }
18 #endif
19
20 void setup() {
21   Serial.begin(115200);
22   while (!Serial) delay(10);
23
24   Serial.println("\n=== Complete Profiling Report ===");
25
26   // 1. Memory profiling
27   Serial.println("\n--- Memory Profile ---");
28   int mem_before = freeMemory();
29   Serial.print("Free memory before TFLite: ");
30   Serial.print(mem_before);

```

```

31 Serial.println(" bytes");
32
33 // Initialize TFLite (omitted for brevity – see LAB05)
34 // ... model setup code ...
35
36 int mem_after = freeMemory();
37 Serial.print("Free memory after TFLite: ");
38 Serial.print(mem_after);
39 Serial.println(" bytes");
40 Serial.print("TFLite overhead: ");
41 Serial.print(mem_before - mem_after);
42 Serial.println(" bytes");
43
44 // 2. Inference timing profile
45 Serial.println("\n--- Inference Timing Profile ---");
46 const int NUM_RUNS = 50;
47 unsigned long times[NUM_RUNS];
48
49 // Warmup
50 interpreter->Invoke();
51
52 // Benchmark
53 for (int i = 0; i < NUM_RUNS; i++) {
54     unsigned long start = micros();
55     interpreter->Invoke();
56     times[i] = micros() - start;
57 }
58
59 // Statistics
60 unsigned long sum = 0, minT = times[0], maxT = times[0];
61 for (int i = 0; i < NUM_RUNS; i++) {
62     sum += times[i];
63     if (times[i] < minT) minT = times[i];
64     if (times[i] > maxT) maxT = times[i];
65 }
66 float mean = (float)sum / NUM_RUNS;
67
68 // Std deviation
69 float variance = 0;
70 for (int i = 0; i < NUM_RUNS; i++) {
71     float diff = times[i] - mean;
72     variance += diff * diff;
73 }
74 float stddev = sqrt(variance / NUM_RUNS);
75
76 Serial.print("Mean latency: ");
77 Serial.print(mean / 1000, 2);
78 Serial.println(" ms");
79 Serial.print("Std deviation: ");
80 Serial.print(stddev / 1000, 2);
81 Serial.println(" ms");
82 Serial.print("Min/Max: ");
83 Serial.print(minT / 1000, 2);
84 Serial.print(" / ");
85 Serial.print(maxT / 1000, 2);
86 Serial.println(" ms");
87 Serial.print("Throughput: ");
88 Serial.print(1000000.0 / mean, 1);
89 Serial.println(" inferences/sec");
90
91 // 3. Arena usage

```

```

92 Serial.println("\n--- Memory Arena ---");
93 Serial.print("Arena used: ");
94 Serial.print(interpreter->arena_used_bytes());
95 Serial.print(" / ");
96 Serial.print(kTensorArenaSize);
97 Serial.print(" bytes ");
98 Serial.print(100.0 * interpreter->arena_used_bytes() / kTensorArenaSize, 1);
99 Serial.println("%");
100
101 // 4. Power estimation
102 Serial.println("\n--- Power Estimation ---");
103 float avg_current_mA = 60.0; // Measured average during inference
104 float inference_time_s = mean / 1000000.0;
105 float energy_mJ = 3.3 * avg_current_mA * inference_time_s;
106
107 Serial.print("Estimated current: ");
108 Serial.print(avg_current_mA, 1);
109 Serial.println(" mA");
110 Serial.print("Energy per inference: ");
111 Serial.print(energy_mJ, 3);
112 Serial.println(" mJ");
113
114 // Battery life (1000 mAh, 1 inference/sec)
115 float battery_hours = 1000.0 / avg_current_mA;
116 Serial.print("Battery life (1 inf/s): ");
117 Serial.print(battery_hours, 1);
118 Serial.print(" hours ");
119 Serial.print(battery_hours / 24, 1);
120 Serial.println(" days");
121
122 Serial.println("\n=== Profiling Complete ===");
123 }
124
125 void loop() {
126 // Empty - profiling done in setup
127 }

```

Listing 11.1: Complete profiling sketch for Arduino

Expected Output:

```

=== Complete Profiling Report ===

--- Memory Profile ---
Free memory before TFLite: 245760 bytes
Free memory after TFLite: 230144 bytes
TFLite overhead:          15616 bytes

--- Inference Timing Profile ---
Mean latency:    12.34 ms
Std deviation:   0.87 ms
Min/Max:         11.21 / 14.56 ms
Throughput:      81.0 inferences/sec

--- Memory Arena ---
Arena used:      8192 / 10240 bytes (80.0%)

--- Power Estimation ---
Estimated current: 60.0 mA
Energy per inference: 2.445 mJ
Battery life (1 inf/s): 16.7 hours (0.7 days)

```



```
=== Profiling Complete ===
```

11.10 Three-Tier Implementation Guide

Level 1: Estimation in Notebooks

Estimate resource requirements before hardware testing:

```
1 import tensorflow as tf
2 import numpy as np
3
4 # Load and analyze model
5 model = tf.keras.models.load_model('gesture_model.h5')
6 model.summary() # Parameter count
7
8 # Estimate memory
9 total_params = model.count_params()
10 float32_bytes = total_params * 4
11 int8_bytes = total_params * 1
12 print(f"\n=== Memory Estimation ===")
13 print(f"Float32 model: {float32_bytes/1024:.1f} KB")
14 print(f"Int8 model: {int8_bytes/1024:.1f} KB")
15
16 # Estimate tensor arena (rough heuristic: 2x model size)
17 arena_estimate = int8_bytes * 2
18 print(f"Estimated arena: {arena_estimate/1024:.1f} KB")
19 print(f"Total SRAM needed: {(int8_bytes + arena_estimate)/1024:.1f} KB")
20
21 # Check device compatibility
22 devices = {
23     'Arduino Nano 33 BLE': 256,
24     'ESP32': 520,
25     'Raspberry Pi Pico': 264,
26     'STM32 Nucleo': 128
27 }
28
29 total_kb = (int8_bytes + arena_estimate) / 1024
30 print(f"\n=== Device Compatibility ===")
31 for name, sram_kb in devices.items():
32     fits = "YES" if total_kb < sram_kb * 0.7 else "NO"
33     print(f"{name:20s}: {fits:3s} ({sram_kb} KB SRAM)")
34
35 # Benchmark and estimate MCU time
36 from benchmark_model import benchmark_model
37 results = benchmark_model(model, (28, 28), num_runs=50)
38 mcu_estimate = results['mean'] * 150 # Nano is ~150x slower
39
40 print(f"\n=== Timing Estimation ===")
41 print(f"Desktop latency: {results['mean']:.2f} ms")
42 print(f"Estimated MCU: {mcu_estimate:.1f} ms")
43 print(f"Real-time (30fps): {'OK' if mcu_estimate < 33 else 'TOO SLOW'}")
```

Level 2: Serial-Based Profiling

Use Serial output for detailed profiling on actual hardware:

1. Upload profiling sketch to device
2. Capture serial output using Arduino IDE or screen
3. Parse CSV data for timing analysis:

```

1 import pandas as pd
2 import matplotlib.pyplot as plt
3
4 # Read profiling data
5 data = pd.read_csv('profile_output.csv')
6
7 # Plot latency distribution
8 plt.figure(figsize=(10, 4))
9 plt.subplot(1, 2, 1)
10 plt.hist(data['latency_ms'], bins=30, edgecolor='black')
11 plt.xlabel('Latency (ms)')
12 plt.ylabel('Frequency')
13 plt.title('Inference Latency Distribution')
14
15 plt.subplot(1, 2, 2)
16 plt.plot(data['time_s'], data['current_mA'])
17 plt.xlabel('Time (s)')
18 plt.ylabel('Current (mA)')
19 plt.title('Current Draw Over Time')
20 plt.tight_layout()
21 plt.savefig('profiling_results.png')

```

4. Compare results across models and optimizations

Level 3: Hardware Power Monitoring

For production-level profiling with real power measurements:

- **Hardware setup:**
 - Connect INA219 sensor in series with device power
 - Use dedicated monitoring Arduino to log INA219 data
 - Alternatively, use USB power meter (e.g., Nordic PPK2)
- **Measurement protocol:**
 1. Measure idle current (no inference)
 2. Measure active current during continuous inference
 3. Measure peak current during single inference
 4. Log full duty cycle (sleep + wake + inference)
- **Analysis:**
 - Calculate energy per inference: $E = V \times I \times t$
 - Estimate battery life with realistic duty cycle
 - Identify power optimization opportunities
- **Validation:**
 - Compare measured vs estimated values
 - Test on actual battery to verify calculations
 - Document worst-case power consumption

11.11 Profiling Case Studies

Real-world examples showing how profiling drives optimization decisions:

11.11.1 Case Study 1: Keyword Spotting Model

Profiling Revealed: Memory, Not Speed, Was the Bottleneck

Initial Problem: Model crashes on Arduino Nano 33 BLE during deployment.

Profiling Results:

- Model size: 18 KB (fits in Flash)
- Tensor arena required: 28 KB
- Available SRAM: 256 KB total, but only 12 KB free after system overhead
- **Root cause:** Arena allocation exceeding available memory

Optimization Applied:

1. Applied Int8 quantization: Arena $\rightarrow$ 10 KB
2. Reduced input window: $16 \text{ kHz} \times 1 \text{ s} \rightarrow 16 \text{ kHz} \times 0.5 \text{ s}$
3. Final arena: 6 KB (fits comfortably)

Result: Model deploys successfully with 85% accuracy (vs 87% for original).

11.11.2 Case Study 2: Gesture Recognition Latency

Profiling Revealed: Dense Layers Dominated Compute Time

Initial Problem: 50 Hz gesture recognition (20 ms deadline) achieves only 15 Hz.

Profiling Results:

- Total inference: 67 ms average
- Conv layers: 15 ms (22%)
- Dense layers: 48 ms (72%)
- Other: 4 ms (6%)

Layer-by-layer breakdown:

Layer	Type	Time (ms)	% Total
conv2d_1	Conv2D	8.2	12%
conv2d_2	Conv2D	6.8	10%
dense_1	Dense	31.5	47%
dense_2	Dense	16.3	24%
output	Dense	4.2	6%

Optimization Applied:

1. Reduced Dense(512) $\rightarrow$ Dense(128): -25 ms
2. Reduced Dense(256) $\rightarrow$ Dense(64): -12 ms
3. Final inference: 30 ms (well under 20 ms deadline with margin)

Result: Achieved 30 fps with 2% accuracy drop (93% $\rightarrow$ 91%).

11.11.3 Case Study 3: Battery-Powered Person Detection

Profiling Revealed: Continuous Inference Killed Battery

Initial Problem: Battery lasts only 8 hours (target: 7 days).

Profiling Results:

- Inference time: 850 ms @ 80 mA
- Running continuously: 80 mA average
- Battery: 2500 mAh $\rightarrow 2500/80 = 31$ hours (not 8!)
- **Additional issue:** WiFi reporting uses 180 mA for 2 seconds per detection

Power breakdown:

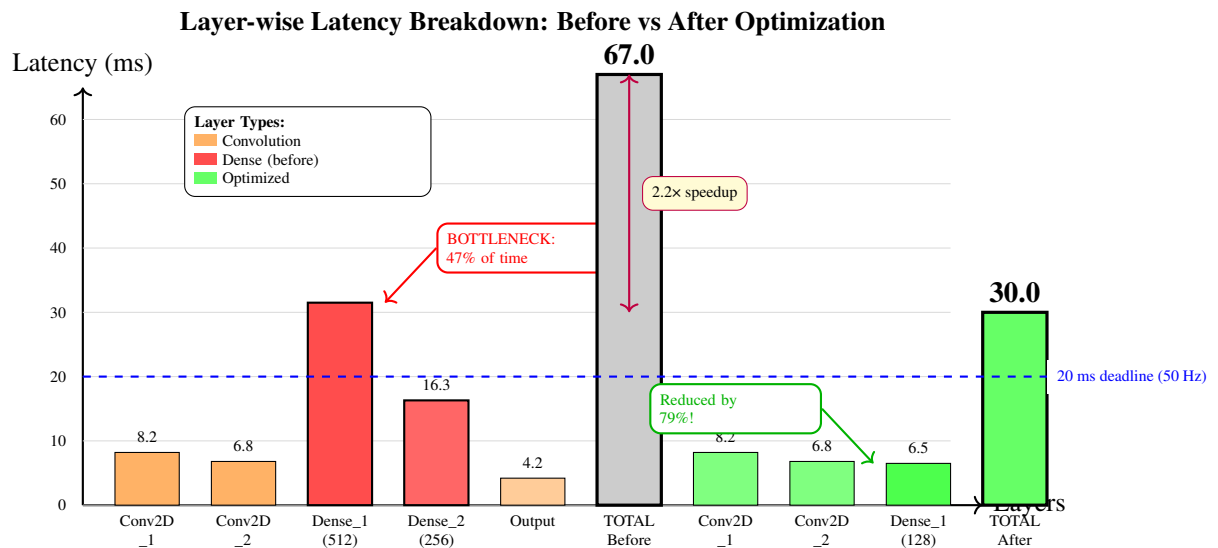


Figure 11.7: Layer-wise latency breakdown for gesture recognition model. Profiling revealed that dense layers consumed 72% of inference time. Reducing dense layer sizes achieved 2.2× speedup, bringing latency under the 20 ms real-time deadline.

Component	Current (mA)	Duty Cycle	Effective (mA)
Deep Sleep	0.01	85%	0.008
MCU Active	25	14%	3.5
Inference	80	1%	0.8
WiFi TX	180	0.02%	0.036
TOTAL:			4.3 mA

Optimization Applied:

1. Duty cycle: Inference every 10 seconds (10% active) instead of continuous
2. Deep sleep between inferences: 0.01 mA
3. Batch WiFi reports: Send every 10 detections instead of each
4. Result: 4.3 mA average → 580 hours (24 days)

Result: Exceeded 7-day target with 3.4× margin. Key insight: Duty cycling matters more than inference speed!

11.12 Troubleshooting Guide

⚠ Profiling Overhead Distorts Measurements

Symptoms: Inference time much higher with profiling enabled, measurements inconsistent

Cause: Instrumentation adds overhead, Serial.println delays, timestamping takes time

Solution:

1. Minimize prints: Only log start/end times, not every operation
2. Use hardware timers: DWT cycle counter on ARM Cortex-M for sub-microsecond precision
3. Disable unnecessary features: Turn off Serial during critical measurements
4. Run multiple iterations: Average over 100+ runs to reduce variance
5. Use release build: Disable debug symbols and enable optimizations (-O2 or -O3)

⚠ Memory Profiler Shows Incorrect Values

Symptoms: Free heap negative, memory usage doesn't match expectations

Cause: Stack overflow, fragmentation, or memory corruption

Solution:

1. Check for stack/heap collision: Ensure stack doesn't grow into heap
2. Use linker memory map: Verify .bss, .data, .rodata sections
3. Print memory at startup: Establish baseline before any allocations
4. Look for leaks: Every malloc/new must have corresponding free/delete
5. Use static analysis: Tools like Valgrind (for host) or AddressSanitizer

⚠ Power Measurements Noisy or Unstable

Symptoms: Power consumption varies wildly, can't get consistent readings

Cause: Measurement setup issues, WiFi activity, or shared power rail

Solution:

1. Use dedicated power supply: Isolated from computer USB noise
2. Add smoothing capacitor: 100 μ F on power rails
3. Disable WiFi/BLE during measurement: `WiFi.mode(WIFI_OFF)`
4. Measure at shunt resistor: Use INA219 or dedicated power monitor
5. Average over time: Sample at least 1 second for stable reading

11.13 Looking Ahead

Looking Ahead

Profiling reveals *where* resources are consumed; the next step is optimizing based on those measurements. LAB12 applies these profiling techniques to streaming data pipelines, measuring throughput and latency under continuous inference. LAB15 extends profiling to energy optimization, using the power measurement skills you've learned to design battery-powered systems with duty cycling and adaptive inference strategies.

11.14 Summary

In this chapter, you learned:

- **Timing measurement:** Use `millis()`/`micros()` with statistical analysis
- **Memory profiling:** Measure SRAM and Flash usage at compile and runtime
- **Power measurement:** Use INA219 or multimeter for current monitoring
- **ML profiling:** Profile inference latency and arena usage
- **Comparison:** Quantify improvements from optimizations
- **Optimization:** Apply targeted fixes based on actual bottlenecks

Self-Assessment Questions

Test your understanding before continuing:

1. Using `micros()`, you measure a function takes 250-280 microseconds across multiple runs. Why does the time vary? What causes timing measurement overhead?
2. An Arduino sketch reports 45% SRAM usage after compilation. At runtime, allocating a 20KB tensor arena causes a crash. Explain why compile-time estimates can be

misleading.

3. A device draws 50mA during inference (100ms) and 5mA idle. For a 1000 mAh battery with inferences every 10 seconds, calculate expected battery life in hours.
4. Compare int8 vs float32 model inference: If float32 takes 450ms, estimate int8 latency assuming 3-4x speedup. What factors affect this ratio on different hardware?
5. Your ML application runs slowly. Profiling shows 60% time in memory allocation, 30% in computation, 10% in I/O. Which optimization will have the biggest impact?

11.15 Guided Experiments

Profiling Overhead Measurement

Hypothesis: The act of profiling (calling timing functions and storing results) introduces measurable overhead that varies with measurement frequency, with overhead becoming negligible for operations taking $>1\text{ms}$.

Procedure:

1. Baseline: Execute 1000 iterations of `sin(x)` without any timing
2. Measure total time with `millis()` at start and end
3. Repeat with `micros()` timing for every iteration (store in array)
4. Repeat with `micros()` timing but immediate printing (no array)
5. Calculate overhead = (measured time - baseline) / 1000 iterations

Data Collection:

Method	Total Time (ms)	Per-iter (us)	Overhead (us)
Baseline (no profiling)			0
<code>millis()</code> at start/end			
<code>micros()</code> per iteration + store			
<code>micros()</code> per iteration + print			

Analysis Questions:

- What is the overhead of a single `micros()` call?
- Why does printing introduce more overhead than array storage?
- For operations taking 10ms, what percentage overhead does per-iteration profiling add?
- When would you choose `millis()` vs `micros()` for profiling?

Optimization Technique Comparison

Hypothesis: Different optimization techniques (loop unrolling, lookup tables, compiler flags) will show varying effectiveness depending on the operation being optimized, with memory-bound operations benefiting less than compute-bound ones.

Procedure:

1. Implement baseline: 1000 iterations of `sqrt(i)` in a loop
2. Optimization 1: Manual loop unrolling (4x unroll)
3. Optimization 2: Lookup table for integer sqrt values (0-1023)
4. Optimization 3: Compile with `-O3` flag instead of default `-Os`
5. Profile each version with 100 runs, record mean and std deviation

Data Collection:

Version	Mean (us)	Std Dev (us)	Speedup	Code Size (B)
Baseline (-Os)			1.0x	
Loop unroll 4x				
Lookup table				
Compiler -O3				

Analysis Questions:

- Which optimization gave the best speedup? Why might this be?
- Did any optimization increase code size significantly? Is the trade-off worth it?
- Why might loop unrolling help or hurt depending on MCU cache size?
- For a memory-constrained device, which optimization would you choose?

11.16 Exercises**[Basic] Basic: Exercise 1: Benchmark Comparison**

Create a benchmark comparing these operations on your board:

1. Integer multiplication (1000 iterations)
2. Floating-point multiplication (1000 iterations)
3. `sin()` function calls (1000 iterations)
4. `analogRead()` calls (1000 samples)

Report mean time and throughput for each.

[Intermediate] Intermediate: Exercise 2: Memory Stress Test

Determine the maximum tensor arena size your board supports:

1. Start with a small arena (1 KB)
2. Increase in 1 KB increments until allocation fails
3. Record free memory at each step
4. Plot arena size vs free memory

[Intermediate] Intermediate: Exercise 3: Power Profile

If you have an INA219 (or multimeter), profile power during:

1. Idle (empty loop)
2. CPU computation (`sin()` loop)
3. ML inference (if you have a model)
4. WiFi transmission (if using ESP32)

Calculate energy per operation.

[Intermediate] Intermediate: Exercise 4: Optimization Verification

Take an existing model and apply one optimization:

1. Profile baseline (memory + latency)

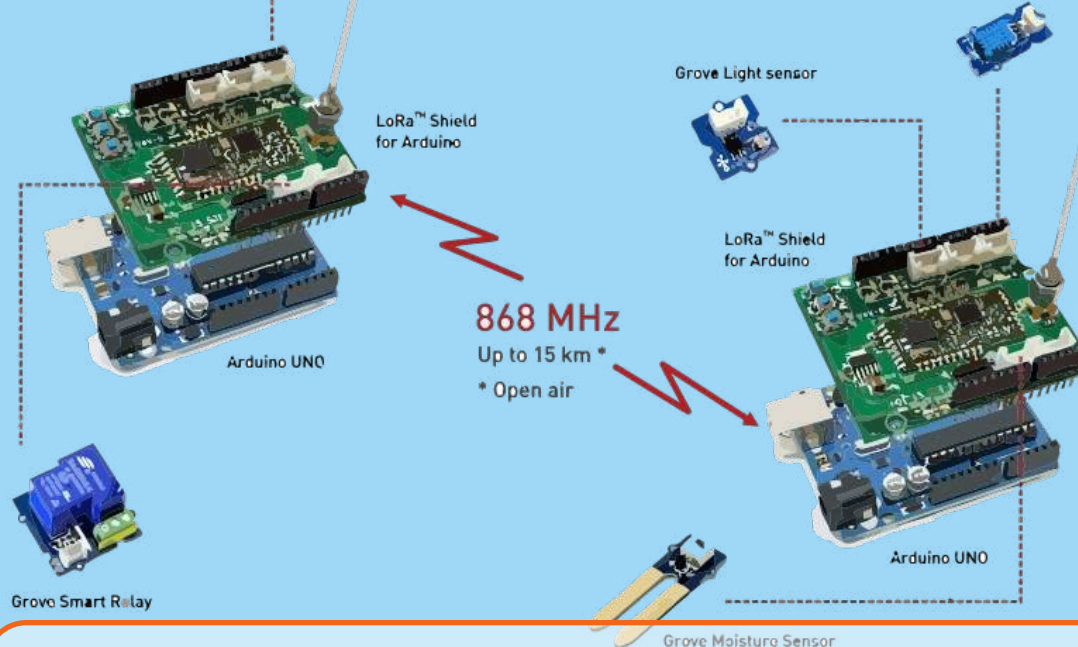
2. Apply Int8 quantization
3. Profile optimized model
4. Calculate actual improvement percentage
5. Compare to theoretical expectations

- **Further Reading.**
- Arduino Memory Guide: <https://www.arduino.cc/en/Tutorial/Foundations/Memory>
 - Adafruit INA219 Tutorial: <https://learn.adafruit.com/adafruit-ina219-current-sensor-breakout>
 - TFLite Micro Profiling: <https://www.tensorflow.org/lite/microcontrollers>
 - Embedded Systems Optimization: “Making Embedded Systems” by Elecia White

— **Credits.** This chapter draws on embedded systems profiling best practices and Arduino documentation. Power measurement techniques adapted from Adafruit tutorials.

11.17 Exercises

1. **[Challenge] Challenge: Bottleneck Identification and Optimization:** Profile a MobileNetV2 model running inference on Raspberry Pi. Use cProfile and TensorFlow Lite benchmarking to identify the three slowest operations. Experiment with at least three optimization strategies: reducing input resolution, quantization to INT8, and operator fusion. Create a performance comparison table showing inference time, memory usage, and accuracy trade-offs for each configuration.
2. **[Challenge] Challenge: Memory Profiling Under Constraints:** Implement memory profiling for a model that must run on a device with only 512 MB RAM. Use `memory_profiler` to track allocation patterns during inference, identify peak memory usage, and implement at least two memory reduction techniques: layer-wise inference (processing one layer at a time), gradient checkpointing for training scenarios, or model pruning. Verify the model fits within the constraint while maintaining acceptable accuracy.
3. **[Challenge] Challenge: Multi-Model Benchmark Suite:** Create a benchmarking framework that compares inference performance of 4 different model architectures (e.g., MobileNet, EfficientNet-Lite, SqueezeNet, MobileViT) across 3 hardware platforms (laptop CPU, Raspberry Pi, and Coral TPU or similar accelerator). Measure FPS, latency percentiles (p50, p95, p99), energy per inference, and accuracy. Generate visualization comparing the Pareto frontier of accuracy vs. speed vs. energy.
4. **[Challenge] Challenge: Real-Time Performance Analysis with Variability:** Design an experiment to measure inference time variability under realistic edge conditions. Run 1000 inferences while simulating system load (background processes, thermal throttling, network activity) and record timing distributions. Identify sources of variance, calculate coefficient of variation, and propose strategies to ensure consistent real-time performance (e.g., CPU affinity, priority scheduling, thermal monitoring). Demonstrate improved p99 latency through your optimizations.



12. Real-Time Stream Processing and IoT Pipelines

Learning Objectives

After completing this lab, you will be able to:

- Understand stream processing fundamentals: buffering, windowing, and backpressure
- Design streaming data pipelines from sensors to visualization
- Read serial data from Arduino using Node.js and Python
- Process and transform real-time sensor measurements with rolling statistics
- Implement time-windowed data aggregation for edge analytics
- Send streaming data to local or remote visualization services
- Handle producer-consumer rate mismatches gracefully

Prerequisites

Before starting this lab, ensure you:

- Completed LAB08 (Arduino Sensors): sensor programming and serial communication
- Completed LAB09 (ESP32 Wireless): understanding of wireless sensor communication
- Familiarity with Python data structures (lists, dictionaries)
- Node.js basics for Level 2+ (can install packages with npm)
- Understanding of producer-consumer patterns helpful but not required

Interactive Notebook

Jupyter Notebook: LAB12_streaming.ipynb

Open in Google Colab | View on GitHub | Companion Website

Prerequisites

- **Knowledge:** Python basics, Arduino serial communication (LAB08)
- **Level 1:** Laptop with Python and matplotlib
- **Level 2:** Arduino + sensor + Raspberry Pi for edge processing
- **Level 3:** Dashboard target such as Dash or InfluxDB+Grafana

■ **Theory and Practice.** Streaming pipelines require fundamentally different thinking than batch processing. You must consider latency budgets, memory constraints, and what happens when data arrives faster than you can process it. This lab builds both conceptual understanding and practical implementation skills, using the sensor knowledge from LAB08 and preparing data flows that will later be stored and queried in LAB13 and profiled for energy in LAB15.

Execution Tiers. Level 1: Simulate sensor streams entirely in notebook/host using synthetic data. Level 2: Run Python/Node.js on Raspberry Pi (or your laptop) reading actual Arduino serial data and applying windowed analytics. Level 3: Deploy a complete pipeline with Arduino sensors, Pi edge processor, and a managed or self-hosted visualization target (for example Dash or InfluxDB+Grafana).

Troubleshooting. Serial flaky? Check baud rate match (both sides must agree), verify port with `ls /dev/tty*` or `dmesg` on Pi. Cloud slow? Buffer locally and send in batches; implement offline plotting fallbacks for network outages.

Three-Tier Execution Model

Tier 1: Jupyter/Colab

Simulate streaming data with generators, implement windowing and buffering algorithms, visualize real-time plots with matplotlib animation

Tier 2: Raspberry Pi

Read Arduino serial data with Python/Node.js, apply rolling statistics and anomaly detection, handle backpressure with bounded buffers

Tier 3: Edge+Dashboard Pipeline

Deploy Arduino sensors streaming to Pi edge processor, implement MQTT for pub/sub, send aggregated data to a managed or self-hosted dashboard, handle network failures gracefully

12.1 Introduction: From Batch to Stream Processing

⚠ Ignoring Backpressure Until It's Too Late

The most insidious streaming bug is ignoring what happens when data arrives faster than you can process it. Your system works perfectly in testing with 1 sample/second, then crashes in production at 100 samples/second. The symptom: memory usage grows unbounded until the process is killed, or old data silently disappears.

Common causes:

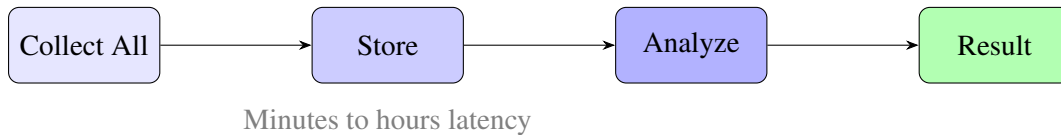
- Using unbounded lists instead of fixed-size ring buffers
- Blocking network calls in the data processing path
- Cloud service rate limits that aren't handled gracefully
- No monitoring of buffer fill levels

Prevention: Always use `deque(maxlen=N)` or similar bounded buffers. Monitor and

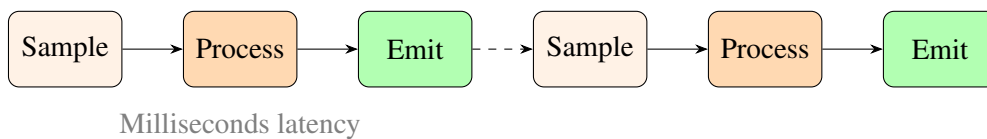
alert on buffer utilization. Test with 10× expected data rates.

Traditional data processing collects all data first, then analyzes. Stream processing analyzes data *as it arrives*—essential for real-time IoT applications where waiting is not an option.

Batch Processing



Stream Processing



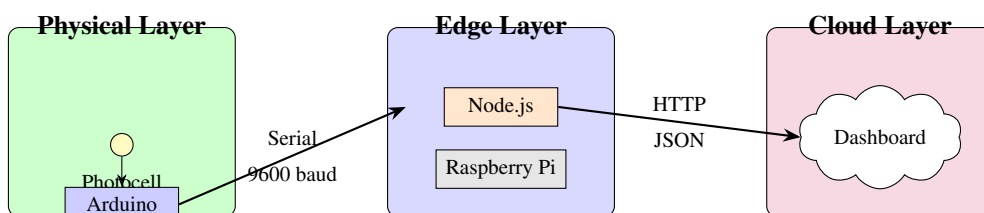
Why Stream Processing for Edge AI?

Edge devices often have limited memory and cannot store hours of sensor data. Stream processing:

- **Reduces memory:** Process and discard, don't accumulate
- **Enables real-time:** React immediately to detected events
- **Scales naturally:** Same code handles 1 sample/sec or 1000
- **Handles infinity:** Sensor streams have no natural “end”

12.2 Streaming Pipeline Architecture

Our IoT streaming pipeline connects three layers: physical sensors, edge coordination, and a visualization target.



Light → ADC → Serial → Parse → Transform → Visualize

Figure 12.1: Three-layer IoT streaming architecture

12.2.1 Components and Supplies

- Arduino Uno or Nano microcontroller
- Photocell (light-dependent resistor / LDR)

- 1K Ω resistor (for voltage divider)
- Breadboard and jumper wires
- USB cable (Arduino to computer/Pi)
- Computer with Node.js installed (or Raspberry Pi)

12.3 The Physical Layer: Arduino Light Sensing

The Arduino reads analog light values from a photocell and streams them over serial.

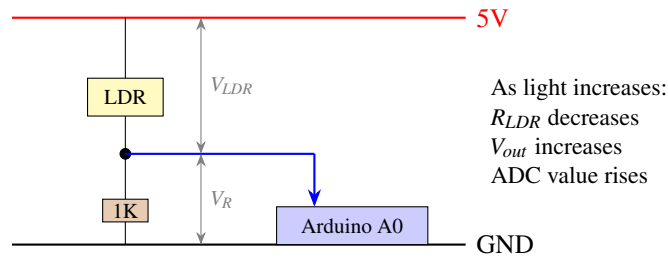


Figure 12.2: LDR voltage divider circuit

12.3.1 Arduino Streaming Sketch

```

1 #define CDS_INPUT 0    // Analog pin for photocell
2
3 void setup() {
4     Serial.begin(9600);
5 }
6
7 void loop() {
8     int value = analogRead(CDS_INPUT);
9     Serial.println(value);
10    delay(1000); // 1 sample per second
11 }

```

Listing 12.1: Light sensor streaming to serial

Code Aliases

The variable name CDS_INPUT refers to a Cadmium Sulfide (CdS) photocell—a common type of LDR.

Code Walkthrough

`Serial.begin(9600)` Initialize serial at 9600 baud—must match the receiver
`analogRead(CDS_INPUT)` Read 10-bit ADC value (0-1023) from voltage divider
`Serial.println(value)` Send value followed by newline (important for parsing!)
`delay(1000)` Wait 1 second—this sets our *sample rate*

Understanding the Output

Higher values = more light (higher voltage at ADC input). Typical indoor ranges:

- Dark room: 50–200
- Normal indoor: 300–600

- Bright light/sunlight: 800–1000+

12.4 Stream Processing Concepts

Before building the coordination layer, we need to understand key streaming concepts.

12.4.1 Buffering and Windows

Stream processing often needs to look at multiple samples together. A **window** is a fixed-size view of recent data.

Buffering Strategies

Different buffering strategies offer distinct trade-offs between memory usage, latency, and data integrity:

Strategy	Memory	Latency	Data Loss	Use Case
Fixed Buffer	Predictable	Variable	Possible	Simple sensors
Ring Buffer	Fixed	Low	Overwrites old	Audio streaming
Double Buffer	2x size	Consistent	None	Video frames
Adaptive	Variable	Optimized	Controlled	Network streams

Table 12.1: Comparison of buffering strategies for stream processing

Fixed buffers allocate a predetermined amount of memory and may drop data when full. **Ring buffers** automatically overwrite the oldest data, making them ideal for real-time applications where recent data is more valuable. **Double buffering** uses two alternating buffers to eliminate data loss during processing—one buffer collects new data while the other is being processed. **Adaptive buffers** dynamically resize based on load, trading memory for reliability.

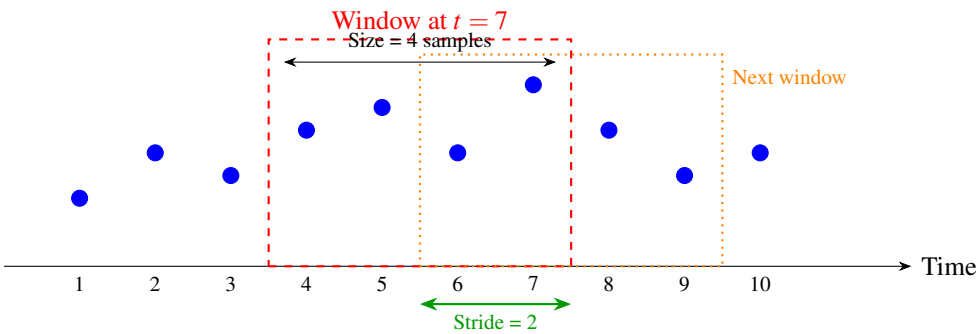


Figure 12.3: Sliding window with stride

Window Parameters

Window Size: How many samples to consider at once (determines smoothing)

Stride: How far to advance between windows (stride < size = overlap)

Overlap: $(size - stride) / size \times 100\%$ of data shared between windows

Windowing Techniques

Different windowing techniques are suited for different streaming analytics use cases:

Window Type	Overlap	Memory	Latency	Best For
Tumbling	None	Low	Fixed	Batch aggregation
Sliding	Yes	Higher	Variable	Trend detection
Session	Variable	Medium	Event-based	User sessions
Hopping	Partial	Medium	Fixed	Time series

Table 12.2: Comparison of windowing techniques for stream analytics

Tumbling windows partition data into non-overlapping, fixed-duration segments—each event belongs to exactly one window. **Sliding windows** continuously move forward with overlap, providing smooth updates ideal for trend detection. **Session windows** dynamically group events based on activity gaps, perfect for user interaction analysis. **Hopping windows** (also called jumping windows) combine fixed size with fixed advance, offering controlled overlap for time-series aggregation.

12.4.2 Backpressure: When Data Arrives Too Fast

What happens when sensors produce data faster than we can process it?

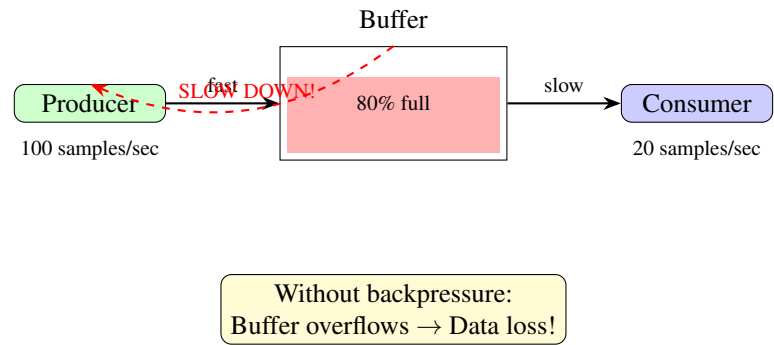


Figure 12.4: Backpressure handling in streaming systems

Handling Rate Mismatches

Three strategies for handling backpressure:

- 1. **Drop oldest:** Ring buffer overwrites old data (good for real-time)
- 2. **Drop newest:** Reject new data when full (preserves history)
- 3. **Block producer:** Slow down data source (may not be possible with sensors!)

For sensor data, dropping oldest is usually correct—stale data has less value.

Backpressure Strategy Comparison

Choosing the right backpressure handling strategy depends on your application’s tolerance for data loss and latency:

Strategy	Data Loss	Latency Impact	Complexity	When to Use
Drop oldest	Yes	None	Low	Real-time priority
Drop newest	Yes	None	Low	Historical priority
Block producer	No	High	Medium	No-loss required
Throttle	No	Medium	High	Rate limiting

Table 12.3: Comparison of backpressure handling strategies

Drop oldest (ring buffer) is ideal for real-time monitoring where the latest data is most critical—think heart rate monitors or motion sensors. **Drop newest** preserves historical continuity, useful when maintaining a complete record matters more than timeliness. **Block producer** applies backpressure by slowing the data source, guaranteeing no loss but potentially causing timeout issues if the producer can't be throttled. **Throttle** uses rate limiting and flow control to match producer speed to consumer capacity without dropping data.

💡 Choosing Window Size and Stride

Consider a temperature monitoring system sampling at 10 Hz (10 samples/second):

- Window size = 100 samples (10 seconds of history)
- Stride = 10 samples (emit one result per second)

What changes if you:

1. Increase window size to 200 samples? (More smoothing, more latency, more memory)
2. Decrease stride to 1 sample? (Output at input rate, 90% overlap, more CPU)
3. Set stride = window size? (Non-overlapping windows, may miss events at boundaries)

There's no "correct" answer—it depends on your latency budget, the frequency content of your signal, and your compute resources. Run experiments!

12.5 Stream Processing Theory

Understanding the mathematical foundations of stream processing helps us design robust, efficient streaming systems. This section presents the key theoretical results from queueing theory, buffer analysis, and windowing algebra.

12.5.1 Little's Law: The Fundamental Queueing Result

Little's Law is one of the most important results in queueing theory, relating the average number of items in a system to the arrival rate and average time spent in the system.

Definition — Little's Law. For a stable queueing system in steady state, the average number of items in the system L is related to the average arrival rate λ and the average time spent in the system W by:

$$L = \lambda \cdot W$$

Where:

- L = average number of items in system (includes queue + service)
- λ = average arrival rate (items/second)
- W = average time an item spends in system (seconds)

Key insight: This relationship holds regardless of the arrival distribution, service time distribution, or queueing discipline (FIFO, LIFO, priority, etc.).

Little's Law for Sensor Streams

Consider a sensor streaming pipeline:

- Sensor produces data at $\lambda = 10$ samples/second
- Average processing time per sample: $W = 50$ ms = 0.05 seconds
- By Little's Law: $L = 10 \times 0.05 = 0.5$ samples

This means on average, only half a sample is in the system at any time. If we observe $L = 5$ samples in our buffer, something is wrong—either the arrival rate increased or processing slowed down!

Practical application: Monitor L (queue length) to detect system degradation. If L grows unbounded, your system is unstable: $\lambda > \mu$ (arrival rate exceeds service rate).

Little's Law Example: IoT Gateway

■ **Example — Edge Gateway Buffer Sizing.** An IoT edge gateway receives sensor data from multiple devices:

- 20 sensors, each sending 1 packet/second $\rightarrow \lambda = 20$ packets/sec
- Average processing time per packet: $W = 100$ ms = 0.1 seconds
- Expected queue length: $L = 20 \times 0.1 = 2$ packets

Buffer sizing decision: Allocate buffer for $L + \text{safety margin}$. With 2 packets expected, a buffer of 10 packets provides $5 \times$ headroom for bursts.

What if we observe $L = 15$ packets?

- Either λ increased (more sensors?) or W increased (slower processing?)
- Using $L = \lambda W$: If λ is still 20, then $W = 15/20 = 0.75$ seconds
- Processing time increased by $7.5 \times$ —investigate immediately!

12.5.2 M/M/1 Queue Model

The M/M/1 queue (Markovian arrivals, Markovian service, 1 server) provides analytical formulas for performance metrics. While real systems rarely have perfectly exponential distributions, M/M/1 gives useful approximations and design intuitions.

Definition — M/M/1 Queue Model. A single-server queue with:

- **Poisson arrivals:** Rate λ (inter-arrival times exponentially distributed)
- **Exponential service:** Rate μ (service times exponentially distributed)
- **One server:** FIFO discipline

Key performance metrics:

$$\text{Utilization: } \rho = \frac{\lambda}{\mu} \quad (\text{fraction of time server is busy})$$

$$\text{Avg items in system: } L = \frac{\rho}{1 - \rho} = \frac{\lambda}{\mu - \lambda}$$

$$\text{Avg items in queue: } L_q = \frac{\rho^2}{1 - \rho} = \frac{\lambda^2}{\mu(\mu - \lambda)}$$

$$\text{Avg time in system: } W = \frac{1}{\mu - \lambda}$$

$$\text{Avg time in queue: } W_q = \frac{\lambda}{\mu(\mu - \lambda)}$$

Stability condition: $\lambda < \mu$ (arrival rate must be less than service rate). When $\lambda \geq \mu$, the queue grows unbounded.

The $\rho \rightarrow 1$ Danger Zone

Notice what happens as utilization ρ approaches 1 (arrival rate approaches service rate):

$$L = \frac{\rho}{1 - \rho} \rightarrow \infty \text{ as } \rho \rightarrow 1$$

Example: $\mu = 100$ samples/sec

λ	ρ	L (avg items)
50	0.50	1.0
80	0.80	4.0
90	0.90	9.0
95	0.95	19.0
99	0.99	99.0

At 99% utilization, the average queue has 99 items! **Rule of thumb:** Keep $\rho < 0.7$ for acceptable latency.

Applying M/M/1 to Stream Processing

■ **Example — Designing a Real-Time Anomaly Detector.** We’re building a streaming anomaly detection system:

- Sensor data arrives at $\lambda = 50$ samples/second
- Processing time: 15 ms/sample $\rightarrow \mu = 1/0.015 = 66.67$ samples/sec
- Utilization: $\rho = 50/66.67 = 0.75$ (75%)

Performance predictions:

$$L = \frac{0.75}{1 - 0.75} = 3.0 \text{ samples (avg in system)}$$

$$W = \frac{1}{66.67 - 50} = 0.06 \text{ seconds} = 60 \text{ ms (avg latency)}$$

$$L_q = \frac{0.75^2}{1 - 0.75} = 2.25 \text{ samples (avg in queue)}$$

$$W_q = \frac{50}{66.67 \times 16.67} = 0.045 \text{ seconds} = 45 \text{ ms (avg wait time)}$$

Interpretation:

- Each sample spends 60 ms in the system on average (45 ms waiting, 15 ms processing)
- We need a buffer of at least $L + \text{margin} \approx 10$ samples
- If requirements demand latency < 30 ms, we must reduce λ or increase μ

12.5.3 Buffer Sizing for Streaming Systems

Proper buffer sizing is critical for reliable streaming systems. Too small, and you drop data; too large, and you waste memory and increase latency.

Definition — Bandwidth-Delay Product (BDP). For network streaming, the **Bandwidth-Delay Product** determines the optimal buffer size:

$$\text{BDP} = \text{RTT} \times C$$

Where:

- RTT = Round-Trip Time (seconds)
- C = Channel capacity (bits/second or samples/second)

Interpretation: BDP is the amount of data “in flight” on the network. The sender must buffer at least BDP bytes to keep the pipe full.

Edge streaming example:

- Sensor to edge gateway: RTT = 10 ms, data rate = 1000 samples/sec
- $\text{BDP} = 0.01 \times 1000 = 10$ samples
- Buffer at least 10 samples to prevent pipeline stalls

Definition — Buffer Formula for Burst Tolerance. To handle bursty arrivals, size your buffer to accommodate the maximum burst duration:

$$B \geq \lambda \cdot T_{\max}$$

Where:

- B = buffer capacity (samples)
- λ = average arrival rate (samples/second)
- $T_{\max}$ = maximum burst duration you want to tolerate (seconds)

Example: If sensor bursts at $\lambda = 100$ samples/sec for up to $T_{\max} = 2$ seconds, allocate $B \geq 100 \times 2 = 200$ samples.

Combined with M/M/1 insight: Use $\max(L + k\sigma_L, \lambda \cdot T_{\max})$ where $k = 3$ for 99.7% coverage.

Practical Buffer Sizing Strategy

For edge streaming systems, combine multiple approaches:

1. **Steady-state:** Use Little's Law to estimate average $L = \lambda W$
2. **Burst handling:** Add capacity for worst-case burst: $B_{\text{burst}} = \lambda T_{\max}$
3. **Safety margin:** Multiply by 1.5–2× for unexpected conditions
4. **Memory constraint:** Cap at maximum available memory

Formula:

$$B = \min(2 \times \max(L, \lambda T_{\max}), B_{\max})$$

Monitor actual queue depth in production—if consistently near capacity, either increase buffer or reduce λ/μ ratio.

12.5.4 Backpressure Mathematics

Backpressure mechanisms prevent buffer overflow by regulating the flow of data. Two common mathematical models are token buckets and rate limiting.

Definition — Token Bucket Algorithm. The token bucket algorithm limits the average rate and allows controlled bursts:

Parameters:

- r = token generation rate (tokens/second)
- b = bucket capacity (tokens)
- Each packet consumes 1 token

Dynamics:

$$\text{Tokens}(t) = \min(b, \text{Tokens}(t-1) + r \cdot \Delta t - \text{packets\_sent})$$

$$\text{Send packet} \Leftrightarrow \text{Tokens}(t) \geq 1$$

Key properties:

- Long-term average rate $\leq r$
- Short-term burst up to b packets
- When bucket empty, enforces minimum inter-packet time of $1/r$ seconds

■ **Example — Token Bucket for Sensor Data.** Sensor sends data to cloud with rate limiting:

- Token rate: $r = 10$ tokens/second
- Bucket capacity: $b = 50$ tokens
- Initially: bucket full (50 tokens)

Scenario 1: Burst then steady

- Sensor bursts 50 packets instantly → depletes bucket, all sent (burst allowed!)
- Bucket empty, now tokens arrive at $r = 10/\text{sec}$
- Sustained rate limited to 10 packets/sec

Scenario 2: Steady below rate

- Sensor sends 5 packets/sec (below r)
- Bucket fills: $50 + (10 - 5) \times t$ up to capacity $b = 50$
- System ready for next burst

Definition — Leaky Bucket Algorithm. Alternative to token bucket, enforces strict constant output rate:

Concept: Queue with constant service rate μ , regardless of arrival pattern.

Difference from token bucket:

- **Token bucket:** Allows bursts up to b (output bursty if input bursty)
- **Leaky bucket:** Smooths traffic to exactly μ (output never bursty)

Formula: If queue length $Q(t) > 0$, output rate is exactly μ . Otherwise, output rate matches input rate (if $\lambda < \mu$).

Edge use case: Leaky bucket for cloud uploads ensures predictable network usage; token bucket for local processing allows burst handling.

12.5.5 Windowing Algebra

Windowing operations partition infinite streams into finite chunks for computation. Let's formalize the mathematics.

Definition — Tumbling Window. A **tumbling window** partitions the stream into non-overlapping, contiguous windows of fixed size.

Formal definition: For a stream $S = \{s_1, s_2, s_3, \dots\}$ and window size w , tumbling windows are:

$$W_i = \{s_{(i-1)w+1}, s_{(i-1)w+2}, \dots, s_{iw}\} \quad \text{for } i = 1, 2, 3, \dots$$

Properties:

- Each element appears in exactly one window: $W_i \cap W_j = \emptyset$ for $i \neq j$
- Windows cover entire stream: $\bigcup_{i=1}^{\infty} W_i = S$
- Output frequency: 1 result per w samples

Memory requirement: $O(w)$ (store current window only)

Computational complexity: $O(w)$ per window if aggregation requires full scan

Definition — Sliding Window. A **sliding window** moves forward by a **stride** $s < w$, creating overlapping windows.

Formal definition: For stream S with window size w and stride s :

$$W_i = \{s_{(i-1)s+1}, s_{(i-1)s+2}, \dots, s_{(i-1)s+w}\} \quad \text{for } i = 1, 2, 3, \dots$$

Overlap factor:

$$\text{Overlap} = \frac{w-s}{w} \times 100\%$$

Properties:

- Elements appear in multiple windows: each element in $\lceil w/s \rceil$ windows
- Output frequency: 1 result per s samples
- Special case: $s = w$ reduces to tumbling window
- Special case: $s = 1$ gives maximal overlap

Memory requirement: $O(w)$

Computational complexity:

- Naive: $O(w)$ per window
- Incremental update: $O(1)$ amortized (for mean, sum, etc.)

Incremental Window Updates

For sliding windows, recomputing the entire window statistic from scratch is wasteful. Use incremental updates:

Rolling sum:

$$\text{Sum}_{i+1} = \text{Sum}_i - s_{i-w+1} + s_{i+1}$$

(Remove oldest element, add newest element)

Rolling mean:

$$\mu_{i+1} = \mu_i + \frac{s_{i+1} - s_{i-w+1}}{w}$$

Rolling variance (Welford's algorithm): Maintain running mean μ and sum of squared differences M_2 :

$$\begin{aligned}\mu_{\text{new}} &= \mu_{\text{old}} + \frac{x_{\text{new}} - x_{\text{old}}}{w} \\ M_{2,\text{new}} &= M_{2,\text{old}} + (x_{\text{new}} - \mu_{\text{old}})(x_{\text{new}} - \mu_{\text{new}}) \\ &\quad - (x_{\text{old}} - \mu_{\text{old}})(x_{\text{old}} - \mu_{\text{new}}) \\ \sigma^2 &= \frac{M_2}{w}\end{aligned}$$

Complexity: $O(1)$ per update instead of $O(w)$ full recomputation—critical for high-frequency streams!

■ **Example — Windowing Complexity Analysis.** Consider a stream processing system computing rolling averages:

- Input rate: $\lambda = 1000$ samples/second
- Window size: $w = 100$ samples
- Stride: $s = 10$ samples (90% overlap)

Naive approach (recompute each window):

- Output frequency: $1000/10 = 100$ results/second
- Each result requires summing 100 values: 100 operations
- Total: $100 \times 100 = 10,000$ operations/second

Incremental approach:

- Each update: subtract old, add new, divide by $w = 3$ operations
- Total: $100 \times 3 = 300$ operations/second

Speedup: $10,000/300 = 33\times$ faster! Essential for resource-constrained edge devices.

Definition — Session Window. A **session window** groups events separated by at most a gap g (inactivity timeout).

Formal definition: A session window W is a maximal set of events where consecutive events are separated by at most g time units:

$$W = \{e_i, e_{i+1}, \dots, e_j\} \text{ where } t_{k+1} - t_k \leq g \text{ for all } i \leq k < j$$

and $t_{j+1} - t_j > g$ (session ends after gap).

Properties:

- Variable-length windows (data-dependent)
- Non-overlapping: sessions separated by gaps $> g$
- Requires event timestamps

Edge IoT example: Motion sensor triggers—session window groups all motion events within 30-second gaps, treating them as a single “activity session.”

💡 Choosing Window Parameters for Edge Systems

Windowing parameters critically affect system behavior:

Trade-off 1: Window size w

- **Large w :** More smoothing, more memory, higher latency, slower adaptation
- **Small w :** Less smoothing, less memory, lower latency, faster adaptation

Trade-off 2: Stride s (for sliding windows)

- **Small s :** More output updates, more CPU, smoother transitions
- **Large s :** Fewer updates, less CPU, may miss transient events

Design questions:

1. What is the timescale of events you want to detect? Set w to cover this timescale.
2. What is your maximum tolerable latency? Ensure $w \times t_{\text{sample}} < \text{latency budget}$.
3. How much overlap do you need for smooth detection? Set s accordingly.
4. What are your memory constraints? Ensure $w \times \text{sample\_size} < \text{RAM}$.

Example: Heart rate monitor (sampling at 100 Hz):

- Detect events lasting ≥ 1 second $\rightarrow w \geq 100$ samples
- Real-time requirement: latency < 200 ms $\rightarrow w \leq 20$ samples
- **Conflict!** Resolution: Use $w = 20$ with multiple scale analysis or increase latency tolerance.

12.6 The Coordination Layer: Node.js Serial Reader

Node.js on a Raspberry Pi (or your computer) reads serial data from Arduino and forwards it to the cloud.

12.6.1 Step 1: Install Dependencies

```
1 npm install serialport
```

Listing 12.2: Installing Node.js serial library

12.6.2 Step 2: Basic Serial Reader

```
1 const { SerialPort } = require('serialport');
2 const { ReadlineParser } = require('@serialport/parser-readline');
3
4 // Configure serial port - CHANGE THIS to match your system!
5 const portName = '/dev/ttyACM0'; // Linux/Pi
6 // const portName = '/dev/tty.usbmodem1411'; // Mac
7 // const portName = 'COM3'; // Windows
8
9 const port = new SerialPort({
10   path: portName,
11   baudRate: 9600
12 });
```

```

13
14 const parser = port.pipe(new ReadlineParser({ delimiter: '\r\n' }));

```

Listing 12.3: server1.js - Basic serial reading

Code Walkthrough

SerialPort Main class for serial communication
ReadlineParser Splits incoming bytes at newline characters
baudRate: 9600 Must match Arduino's `Serial.begin(9600)`
delimiter: `'\r\n'` Arduino's `println()` sends carriage return + newline

```

1 // Handle incoming data
2 parser.on('data', (line) => {
3   const value = parseInt(line.trim(), 10);
4
5   if (!isNaN(value) && value >= 0 && value <= 1023) {
6     console.log(`Light: ${value}`);
7   }
8 });
9
10 // Handle errors
11 port.on('error', (err) => {
12   console.error('Serial error:', err.message);
13 });
14
15 console.log(`Listening on ${portName}...`);

```

Listing 12.4: Handling incoming data

Code Walkthrough

`parser.on('data', ...)` Callback fired for each complete line
`parseInt(line.trim(), 10)` Parse string to integer, base 10
`value >= 0 && value <= 1023` Validate ADC range (10-bit)
`port.on('error', ...)` Handle connection issues gracefully

Finding Your Serial Port

Linux/Pi: `ls /dev/tty*` or check `dmesg | tail`

Mac: `ls /dev/tty.usb*`

Windows: Check Device Manager → Ports (COM & LPT)

Arduino IDE: Tools → Port shows the correct port

12.7 Real-Time Processing Pipeline

Let's add rolling statistics to our stream processor.

12.7.1 Python Stream Processor

```

1 from collections import deque
2 import numpy as np

```

```

3
4 class StreamProcessor:
5     """Processes streaming sensor data with windowed statistics."""
6
7     def __init__(self, window_size=20):
8         self.window_size = window_size
9         self.buffer = deque(maxlen=window_size)

```

Listing 12.5: stream_processor.py - Rolling statistics

Code Walkthrough

`deque(maxlen=window_size)` Automatically drops oldest when full—perfect for streaming!
`window_size=20` At 1 sample/sec, this is a 20-second rolling window

```

1     def process(self, sample):
2         """Process sample and compute rolling statistics."""
3         self.buffer.append(sample)
4
5         if len(self.buffer) < 2:
6             return {'value': sample, 'ready': False}
7
8         values = np.array(self.buffer)
9
10        return {
11            'value': sample,
12            'ready': True,
13            'mean': np.mean(values),
14            'std': np.std(values),
15            'min': np.min(values),
16            'max': np.max(values),
17            'trend': values[-1] - values[0] # Change over window
18        }

```

Listing 12.6: Processing each sample

Code Walkthrough

`self.buffer.append(sample)` Add new sample (oldest auto-removed if full)
`ready: False` Signal that we need more samples for valid stats
`trend` Positive = increasing, negative = decreasing over window

12.7.2 Anomaly Detection

```

1     def detect_anomaly(self, sample, threshold=2.0):
2         """Detect anomalies using z-score."""
3         if len(self.buffer) < self.window_size:
4             return False
5
6         values = np.array(self.buffer)
7         mean = np.mean(values)
8         std = np.std(values)
9
10        if std < 0.01: # Avoid division by near-zero
11            return False
12

```

```

13     z_score = abs(sample - mean) / std
14     return z_score > threshold

```

Listing 12.7: Z-score based anomaly detection

Code Walkthrough

`z_score = abs(sample - mean) / std` How many standard deviations from mean
`threshold=2.0` Values beyond 2 std are flagged (catches 5% of normal data)
`std < 0.01` Guard against division by zero in constant signals

Streaming Anomaly Detection

For real-time anomaly detection, we compare each new sample against recent history. The z-score method is simple but effective:

- $|z| > 2$: Unusual (95% confidence)
- $|z| > 3$: Very unusual (99.7% confidence)

Adjust threshold based on your false positive tolerance.

12.8 Time-Windowed Aggregation

For cloud transmission, we often aggregate multiple samples into time windows to reduce bandwidth.

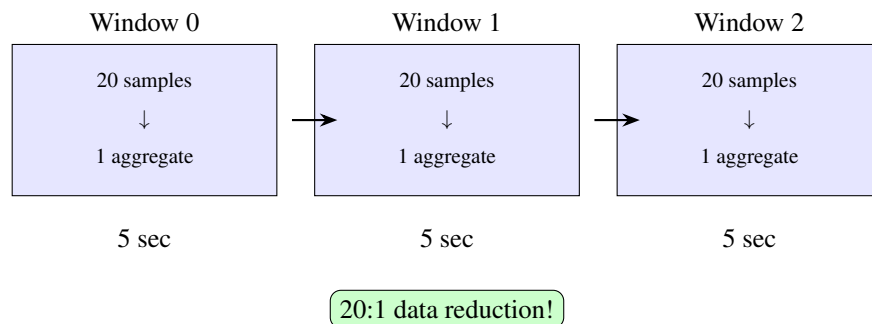


Figure 12.5: Time-windowed aggregation reduces data volume

```

1 import time
2 from datetime import datetime
3
4 class WindowedAggregator:
5     """Aggregate streaming data into time windows."""
6
7     def __init__(self, window_seconds=5):
8         self.window_seconds = window_seconds
9         self.current_window = []
10        self.window_start = None

```

Listing 12.8: Windowed aggregator class

```

1     def add(self, sample):
2         """Add sample, return aggregate if window closed."""
3         now = time.time()

```



```

4
5     if self.window_start is None:
6         self.window_start = now
7
8     # Check if window should close
9     if now - self.window_start >= self.window_seconds:
10         aggregate = self._close_window()
11         self.window_start = now
12         self.current_window = [sample]
13         return aggregate
14
15     self.current_window.append(sample)
16     return None # Window still open

```

Listing 12.9: Adding samples and closing windows

```

1     def _close_window(self):
2         """Compute aggregate statistics for closed window."""
3         if not self.current_window:
4             return None
5
6         values = np.array(self.current_window)
7
8         return {
9             'timestamp': datetime.now().isoformat(),
10            'count': len(values),
11            'min': float(np.min(values)),
12            'max': float(np.max(values)),
13            'mean': float(np.mean(values)),
14            'std': float(np.std(values))
15        }

```

Listing 12.10: Computing window aggregate

Code Walkthrough

window_seconds=5 Aggregate every 5 seconds of data
 self._close_window() Called when window expires, returns aggregate
 return None Signals “still collecting”—no data to send yet
 float(...) Convert numpy types to JSON-serializable Python floats

12.9 Visualization Targets

A common deployment pattern is to forward aggregated sensor data from the edge node to a dashboard. Historically, Plotly Chart Studio exposed a streaming-token workflow, but that cloud feature is no longer generally available. The example below is retained as a legacy integration pattern; for current deployments, prefer a local Dash app, an HTTP/WebSocket dashboard, or InfluxDB + Grafana.

12.9.1 Choosing a Streaming Protocol

Before implementing cloud connectivity, it’s important to select the right protocol for your edge streaming application:

Protocol	Latency	Reliability	Overhead	Edge Suitability
MQTT	Low	QoS levels	Very Low	Excellent
WebSocket	Very Low	TCP-based	Low	Good
gRPC	Low	HTTP/2	Medium	Good
Kafka	Medium	High	High	For aggregation

Table 12.4: Comparison of edge streaming protocols

MQTT is the gold standard for IoT edge devices, offering lightweight publish-subscribe messaging with three Quality of Service (QoS) levels and minimal bandwidth requirements—ideal for constrained devices. **WebSocket** provides full-duplex communication over TCP with very low latency, suitable for real-time dashboards and browser-based interfaces. **gRPC** uses HTTP/2 for efficient binary serialization with built-in flow control, excellent for microservices but more resource-intensive. **Kafka** excels at high-throughput data aggregation and stream processing but requires substantial infrastructure, making it better suited for edge gateways aggregating multiple sensor streams rather than individual sensors.

12.9.2 Legacy Plotly Setup

If you are maintaining an older or private deployment that still exposes Plotly-style streaming credentials:

1. Confirm your private or enterprise Plotly deployment still supports streaming tokens
2. Export: `PLOTLY_USERNAME`, `PLOTLY_API_KEY`, and `PLOTLY_STREAM_TOKEN`
3. Install library: `npm install plotly`

12.9.3 Legacy Plotly Streaming Example

```

1 const { SerialPort } = require('serialport');
2 const { ReadlineParser } = require('@serialport/parser-readline');
3 const plotlyUsername = process.env.PLOTLY_USERNAME;
4 const plotlyApiKey = process.env.PLOTLY_API_KEY;
5 const token = process.env.PLOTLY_STREAM_TOKEN;
6
7 if (!plotlyUsername || !plotlyApiKey || !token) {
8   throw new Error('Set PLOTLY_USERNAME, PLOTLY_API_KEY, and
9     PLOTLY_STREAM_TOKEN for your legacy deployment. ');
10 }
11
12 const plotly = require('plotly')(plotlyUsername, plotlyApiKey);
13 const portName = '/dev/ttyACM0';
14
15 // Serial port setup
16 const port = new SerialPort({ path: portName, baudRate: 9600 });
17 const parser = port.pipe(new ReadlineParser({ delimiter: '\r\n' }));

```

Listing 12.11: server2.js - Legacy Plotly streaming pipeline

```

1 // Helper: format timestamp
2 function getTimestamp() {
3   return new Date().toISOString();
4 }
5
6 // Initialize Plotly plot
7 const initData = [{
8   x: [],
9   y: [],
10  stream: { token: token, maxpoints: 500 }

```

```

11 }];
12
13 const layout = {
14   filename: 'light-sensor-stream',
15   fileopt: 'extend'
16 };

```

Listing 12.12: Initialize legacy Plotly stream

```

1 plotly.plot(initData, layout, (err, msg) => {
2   if (err) return console.error('Plotly error:', err);
3
4   console.log('Plot URL:', msg.url);
5
6   // Open streaming connection
7   const stream = plotly.stream(token, (err, res) => {
8     if (err) console.error('Stream error:', err);
9   });
10
11   // Forward serial data to Plotly
12   parser.on('data', (line) => {
13     const value = parseInt(line.trim(), 10);
14
15     if (isNaN(value) || value < 0 || value > 1023) return;
16
17     const dataPoint = JSON.stringify({
18       x: getTimestamp(),
19       y: value
20     });
21
22     stream.write(dataPoint + '\n');
23     console.log(`Sent: ${value}`);
24   });
25 });

```

Listing 12.13: Stream data to legacy Plotly endpoint

Code Walkthrough

maxpoints: 500 Keep last 500 points on the plot (prevents infinite growth)
 fileopt: 'extend' Append to existing file rather than overwrite
 plotly.stream(token, ...) Opens the legacy Plotly streaming connection
 stream.write(dataPoint + '\n') Send JSON-formatted point to the remote dashboard endpoint

Visualization Service Considerations

- **Service quotas:** Managed dashboards often enforce ingestion limits, so aggregate locally before upload
- **Network failures:** Buffer locally and retry on reconnection
- **Cost:** Hosted offerings may have usage tiers; consider self-hosted alternatives
- **Privacy:** Sensor data goes to third-party servers

For production, consider a local Dash app or self-hosted alternatives like InfluxDB + Grafana.

12.10 Handling Backpressure

Implementing a buffer with backpressure handling:

```

1 from collections import deque
2
3 class BufferedStream:
4     """Stream buffer with backpressure handling."""
5
6     def __init__(self, capacity=100):
7         self.buffer = deque(maxlen=capacity)
8         self.capacity = capacity
9         self.dropped = 0
10        self.total = 0

```

Listing 12.14: Buffered stream with backpressure

```

1     def put(self, sample):
2         """Add sample. Returns False if dropped due to backpressure."""
3         self.total += 1
4
5         if len(self.buffer) >= self.capacity:
6             self.dropped += 1
7             # maxlen will auto-drop oldest
8
9         self.buffer.append(sample)
10        return len(self.buffer) < self.capacity
11
12    def get(self):
13        """Get oldest sample, or None if empty."""
14        return self.buffer.popleft() if self.buffer else None
15
16    def stats(self):
17        """Get buffer statistics."""
18        return {
19            'size': len(self.buffer),
20            'capacity': self.capacity,
21            'utilization': len(self.buffer) / self.capacity * 100,
22            'dropped': self.dropped,
23            'drop_rate': self.dropped / self.total * 100 if self.total else
24            0
25        }

```

Listing 12.15: Put and get with statistics

Code Walkthrough

`deque(maxlen=capacity)` Automatically discards oldest when capacity exceeded
`return len(self.buffer) < self.capacity` Returns False as backpressure signal
`drop_rate` Percentage of samples lost—monitor this!

Monitoring Buffer Health

In production streaming systems, monitor:

- **Buffer utilization:** Consistently >80% = consumer too slow
- **Drop rate:** Any drops indicate capacity issues
- **Latency:** Time from sample arrival to processing

Alert when these metrics exceed thresholds.

12.11 Complete Python Pipeline Example

Here's a complete example combining serial reading, processing, and local visualization:

```

1 import serial
2 import time
3 import numpy as np
4 from collections import deque
5 import matplotlib.pyplot as plt
6 from matplotlib.animation import FuncAnimation
7
8 # Configuration
9 SERIAL_PORT = '/dev/ttyACM0'
10 BAUD_RATE = 9600
11 WINDOW_SIZE = 50
12 PLOT_POINTS = 200
13
14 # Data buffers
15 raw_data = deque(maxlen=PLOT_POINTS)
16 smoothed_data = deque(maxlen=PLOT_POINTS)
17 timestamps = deque(maxlen=PLOT_POINTS)
18 window_buffer = deque(maxlen=WINDOW_SIZE)

```

Listing 12.16: complete_pipeline.py - Full streaming system

```

1 def read_serial(port, callback):
2     """Read serial data in background thread."""
3     ser = serial.Serial(port, BAUD_RATE, timeout=1)
4     time.sleep(2) # Wait for Arduino reset
5
6     while True:
7         if ser.in_waiting:
8             try:
9                 line = ser.readline().decode('utf-8').strip()
10                 value = int(line)
11                 if 0 <= value <= 1023:
12                     callback(value)
13             except (ValueError, UnicodeDecodeError):
14                 pass

```

Listing 12.17: Serial reading thread

```

1 def process_sample(value):
2     """Process each incoming sample."""
3     global window_buffer
4
5     # Add to window
6     window_buffer.append(value)
7
8     # Compute smoothed value
9     smoothed = np.mean(window_buffer)
10
11     # Store for plotting
12     timestamps.append(time.time())
13     raw_data.append(value)
14     smoothed_data.append(smoothed)

```

Listing 12.18: Processing callback

```

1 def animate(frame):
2     """Update plot animation."""
3     if len(timestamps) < 2:
4         return
5
6     t = np.array(timestamps)
7     t = t - t[0] # Relative time
8
9     ax.clear()
10    ax.plot(t, raw_data, 'b-', alpha=0.3, label='Raw')
11    ax.plot(t, smoothed_data, 'r-', linewidth=2, label='Smoothed')
12    ax.set_xlabel('Time (s)')
13    ax.set_ylabel('Light (ADC)')
14    ax.legend(loc='upper right')
15    ax.set_ylim(0, 1024)
16
17 # Start serial reader in background
18 import threading
19 thread = threading.Thread(target=read_serial,
20                           args=(SERIAL_PORT, process_sample))
21 thread.daemon = True
22 thread.start()
23
24 # Start animation
25 fig, ax = plt.subplots(figsize=(12, 6))
26 ani = FuncAnimation(fig, animate, interval=100)
27 plt.show()

```

Listing 12.19: Real-time visualization

Code Walkthrough

thread.daemon = True Thread dies when main program exits
 FuncAnimation(fig, animate, interval=100) Update plot every 100ms
 t = t - t[0] Convert absolute timestamps to relative (starts at 0)
 alpha=0.3 Transparency for raw data, making smoothed line stand out

12.12 Three-Tier Implementation Guide**Level 1: Simulated Streaming**

Run the notebook LAB12_streaming.ipynb to practice stream concepts without hardware:

- DataStream class generating synthetic sensor data
- StreamProcessor for rolling statistics
- WindowedAggregator for time-based batching
- BufferedStream for backpressure simulation

Practice windowing, aggregation, and anomaly detection on simulated data before connecting real hardware.

Level 2: Real Hardware with Local Processing

Connect real Arduino and process data locally on your laptop or Raspberry Pi:

1. Build LDR circuit (Figure 12.2)
2. Upload Arduino sketch
3. Run `python complete_pipeline.py`
4. Observe real-time light measurements in matplotlib

This validates your serial communication and processing logic before adding network complexity.

Level 3: Full Production Pipeline

Add cloud integration and production monitoring:

1. Choose a visualization target: managed dashboard/API, local Dash app, or self-hosted InfluxDB + Grafana
2. If maintaining a legacy Plotly deployment, adapt `server2.js`; otherwise send aggregated windows to your chosen endpoint
3. For self-hosted: Install InfluxDB + Grafana on Pi
4. Add alerting for anomaly detection
5. Implement reconnection logic for network failures

Production Considerations

Cloud services have rate limits and privacy implications. Consider self-hosted solutions (InfluxDB + Grafana) for sensitive data or high-frequency streams.

12.13 Troubleshooting Guide

⚠ Buffer Overflow / Data Loss

Symptoms: Sensor readings dropped, buffer full warnings, data gaps in output

Cause: Processing too slow, buffer too small, or sensor sampling faster than consumption

Solution:

1. Increase buffer size: Allocate larger circular buffer (trade memory for reliability)
2. Optimize processing: Reduce computation per sample
3. Downsample input: Reduce sensor sample rate to match processing capacity
4. Use interrupts: Priority-based ISR to capture samples even during processing
5. Implement backpressure: Signal sensor to slow down when buffer nearly full

⚠ High Latency in Stream Processing

Symptoms: Output lags behind input by seconds, real-time requirements not met

Cause: Large window sizes, blocking operations, or inefficient algorithms

Solution:

1. Reduce window size: Smaller windows = faster updates but less smoothing
2. Use incremental updates: Update statistics on each sample, don't recompute all
3. Avoid blocking I/O: Use non-blocking serial or async operations
4. Profile critical path: Identify and optimize slowest operations
5. Consider hardware acceleration: Use DMA for data movement

⚠ Circular Buffer Implementation Bugs

Symptoms: Corrupted data, array out of bounds, or infinite loops

Cause: Incorrect wraparound logic, off-by-one errors, race conditions

Solution:

1. Use modulo operator: `index = (index + 1) % BUFFER_SIZE`
2. Separate read/write pointers: Track head and tail independently
3. Check empty/full conditions: `if ((head + 1) % size == tail)` for full
4. Add bounds checking: Assert indices always `< BUFFER_SIZE`
5. Test edge cases: Empty buffer, full buffer, single element

12.14 Summary

- **Stream vs Batch:** Stream processing handles infinite data with bounded memory by processing samples as they arrive rather than collecting first.
- **Windowing:** Sliding windows let us compute statistics over recent history. Window size controls smoothing; stride controls overlap.
- **Backpressure:** When producers outpace consumers, use ring buffers with explicit drop handling. Monitor drop rates in production.
- **Serial Communication:** Arduino and edge processor must agree on baud rate. Parse lines with `ReadlineParser` (Node.js) or `readline()` (Python).
- **Aggregation:** Time-windowed aggregation reduces bandwidth to cloud by summarizing many samples into statistics.
- **Anomaly Detection:** Z-score method compares each sample to recent statistics. Threshold of 2-3 catches unusual events.
- **Visualization Integration:** Managed or self-hosted dashboards enable real-time monitoring, but quotas, privacy, and failure handling must be designed explicitly.

12.15 Guided Experiments

Buffer Size vs Latency Trade-off

Hypothesis: Increasing buffer size reduces data loss under bursty traffic but increases end-to-end latency linearly, with optimal buffer size dependent on the ratio of producer to consumer rates.

Procedure:

1. Simulate streaming data with producer rate = 50 samples/sec
2. Implement consumer that processes at 40 samples/sec (20% slower)
3. Test buffer sizes: 10, 50, 100, 500 samples
4. For each configuration, run 60 seconds and measure:
 - Number of dropped samples
 - Average latency (time from production to consumption)
 - Peak buffer utilization

Data Collection:

Buffer Size	Dropped Samples	Avg Latency (ms)	Peak Utilization (%)
10			
50			
100			
500			

Analysis Questions:

- At what buffer size do drops stop occurring? Why?
- Plot latency vs buffer size. What is the relationship?
- Calculate theoretical minimum buffer size to avoid drops: $(producer\_rate - consumer\_rate) \times duration$
- For a real-time system requiring $<200\text{ms}$ latency, what constraints does this place on buffer size?

Batch Size Effect on Streaming Throughput

Hypothesis: Batching samples before processing reduces per-sample overhead and increases throughput, but introduces latency equal to batch accumulation time, with diminishing returns beyond a certain batch size.

Procedure:

1. Create streaming pipeline that computes rolling mean over window of 20 samples
2. Vary batch size: 1 (no batching), 5, 10, 20, 50 samples
3. For each batch size, process 1000 samples and measure:
 - Total processing time
 - Throughput (samples/second)
 - Average latency from sample arrival to result emission

Data Collection:

Batch Size	Total Time (s)	Throughput (samples/s)	Avg Latency (ms)
1			
5			
10			
20			
50			

Analysis Questions:

- Plot throughput vs batch size. At what point do gains plateau?
- What is the theoretical best-case latency for batch size N at input rate R ?
- Why does batching improve throughput? (Hint: consider Python/NumPy overhead)
- For a system requiring both high throughput and $<100\text{ms}$ latency at 100 samples/sec, what is the optimal batch size?

Related Labs**Prerequisites:**

- **LAB04 - Keyword Spotting:** Understanding of audio stream processing and windowing techniques that apply to general streaming data pipelines.
- **LAB11 - Profiling and Optimization:** Knowledge of performance measurement and bottleneck analysis is essential for optimizing streaming systems.

Next Steps:

- **LAB13 - Distributed Edge ML Systems:** Extend streaming concepts to distributed systems where multiple edge nodes coordinate and share inference results in real-time.

12.16 Exercises**Self-Assessment Questions**

Test your understanding before continuing:

1. What is the fundamental difference between batch and stream processing? Why can't you use batch processing for real-time anomaly detection?
2. Explain why `deque(maxlen=N)` helps with memory in streaming. What happens when the $N+1$ -th element is added? Why is this better than a regular list?
3. A sliding window of size 100 with stride 50 processes data with 50% overlap. Compare this to stride 100 (no overlap) in terms of detection latency and smoothness. Which would you use for critical safety applications?
4. When buffer capacity is exceeded, you can drop old data, drop new data, or block the producer. Explain the trade-offs of each strategy for a medical monitoring system vs. a weather station.
5. Arduino sends at 115200 baud but Node.js reads at 9600 baud. What symptoms would you observe? How would you debug this issue?

Adaptive Sampling Rate

Modify the Arduino sketch to increase sampling rate when rapid changes are detected:

- Track the difference between consecutive readings
- If $\text{delta} > \text{threshold}$, reduce delay to 100ms
- Otherwise, use 1000ms delay to save power
- Print the current sample rate for debugging

This is called *event-triggered sampling*.

Multi-Sensor Stream

Extend the pipeline to read from multiple sensors:

- Add a temperature sensor (LM35 or TMP36)
- Send both values as CSV: `light,temperature`
- Parse and process both streams independently
- Detect correlations between light and temperature

Offline Buffering

Add network failure resilience:

- Buffer aggregated windows locally when cloud unavailable
- Detect cloud reconnection

- Replay buffered data with original timestamps
- Limit local buffer size (consider disk persistence)

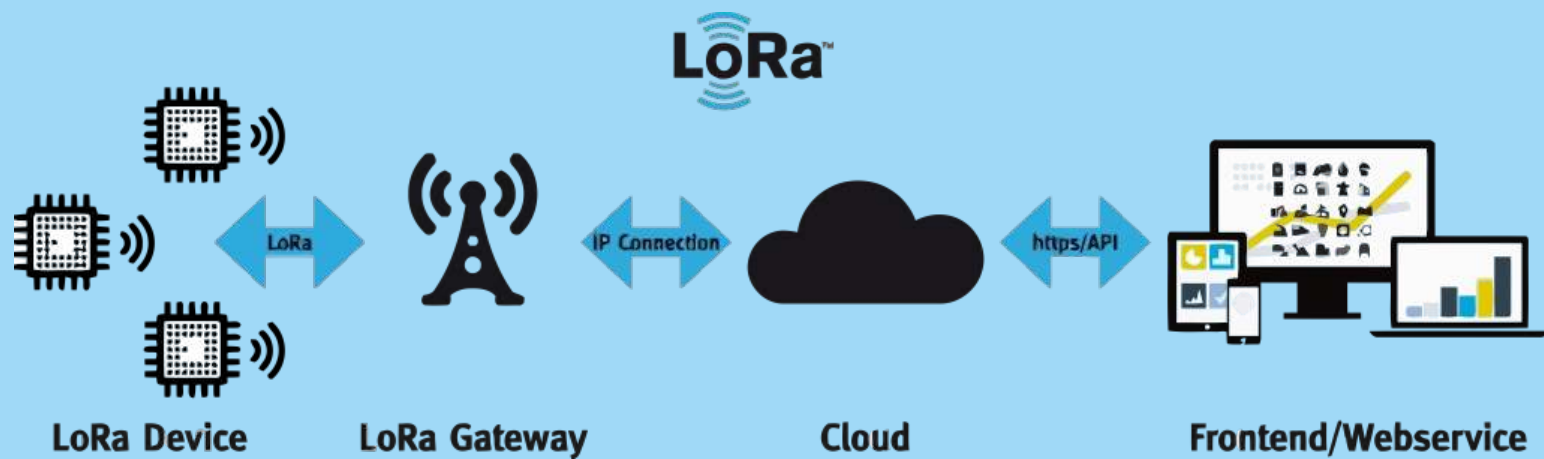
— **Further Reading.**

- Node.js SerialPort: <https://serialport.io/>
- Apache Kafka for IoT Streaming: <https://kafka.apache.org/>
- InfluxDB Time-Series Database: <https://www.influxdata.com/>
- Grafana Real-Time Dashboards: <https://grafana.com/>
- MQTT Protocol for IoT: <https://mqtt.org/>

— **Credits.** Original materials from Coursera IoT Specialization, adapted and expanded for edge analytics education.

12.17 Exercises

1. **[Challenge] Challenge: Adaptive Window Sizing for Stream Processing:** Implement an adaptive windowing system that automatically adjusts window size based on data characteristics. Start with a 50-sample sliding window and monitor variance—increase window size when variance is high (noisy signal) and decrease when variance is low (stable signal). Test on both synthetic data and real sensor streams. Compare detection latency and smoothing effectiveness against fixed-size windows.
2. **[Challenge] Challenge: Multi-Sensor Correlation Detection:** Extend the streaming pipeline to ingest data from three different sensor types (e.g., temperature, humidity, light). Implement cross-correlation analysis to detect when sensor readings move together or diverge. Calculate Pearson correlation coefficients over rolling 2-minute windows and trigger alerts when correlations change significantly (e.g., temperature and humidity normally correlate 0.7+ but drop below 0.3). Visualize correlation matrices in real-time.
3. **[Challenge] Challenge: Backpressure Handling and Buffer Tuning:** Design experiments to stress-test your streaming pipeline under varying producer/consumer speeds. Create a fast producer (100 samples/sec) and slow consumer (10 samples/sec), implement three backpressure strategies: drop oldest, drop newest, and block producer. Measure and compare drop rates, latency distribution, and memory usage for each strategy. Determine optimal buffer size that balances memory footprint against acceptable packet loss (<1%).
4. **[Challenge] Challenge: Fault-Tolerant Cloud Streaming with Replay:** Build a production-ready streaming system with network resilience. Implement local buffering that persists aggregated data to disk when cloud connectivity fails, detects reconnection, and replays buffered data with original timestamps. Test by deliberately disconnecting network for 5 minutes during active streaming. Verify data integrity and temporal ordering after reconnection. Add monitoring for buffer disk usage and automatic old data purging.



13. Edge-to-Cloud Data Ingestion and Storage

Learning Objectives

After completing this lab, you will be able to:

- Design database schemas optimized for IoT time-series data
- Implement HTTP-based data ingestion pipelines from edge devices
- Connect Arduino to network infrastructure via Ethernet
- Store and query sensor data using SQL databases
- Understand trade-offs between edge storage, cloud databases, and hybrid approaches
- Build RESTful APIs for sensor data collection and retrieval

Prerequisites

Before starting this lab, ensure you:

- Completed LAB05 (Embedded Systems Deployment): edge device programming and deployment
- Basic SQL knowledge (SELECT, INSERT, CREATE TABLE)
- Understanding of HTTP request/response cycle
- Python basics including working with dictionaries and JSON
- Familiarity with REST API concepts helpful but not required

Interactive Notebook

Jupyter Notebook: LAB13_distributed.ipynb

[Open in Google Colab](#) | [View on GitHub](#) | [Companion Website](#)

Prerequisites

- **Knowledge:** HTTP basics, SQL fundamentals, Arduino networking
- **Level 1:** Laptop with Python, Flask, and SQLite
- **Level 2:** Raspberry Pi running Flask API with MySQL
- **Level 3:** Arduino Ethernet Shield or ESP32 for device connectivity

Theory and Practice. This lab bridges edge sensing with data persistence. While we demonstrate HTTP+MySQL for simplicity, production IoT often uses lighter protocols (MQTT, CoAP) and specialized time-series databases. The concepts—schema design, ingestion patterns, query optimization—transfer to any stack. The data you stream in LAB12 can be ingested and stored using the patterns in this chapter, and the resulting database can later be profiled (LAB11) and analysed for energy and cost trade-offs (LAB15).

Execution Tiers. Level 1: Simulate sensor data and practice SQL in notebook with SQLite. Level 2: Run Python Flask API on Pi or laptop with a local database; test with simulated POST requests (or a Docker container). Level 3: Deploy Arduino with Ethernet shield or ESP32 sending real sensor data to the API.

Troubleshooting. Ethernet fails? Verify MAC uniqueness, check DHCP or static IP configuration, test connectivity with ping. Database errors? Verify credentials, check table exists, ensure foreign keys match. Arduino memory tight? Minimize string operations, use `F()` macro for literals.

Three-Tier Execution Model

Tier 1: Jupyter/Colab

Design database schemas with SQLite, practice SQL queries with pandas, simulate sensor data ingestion and aggregation

Tier 2: Raspberry Pi

Deploy Flask API server with MySQL or PostgreSQL, test HTTP endpoints with curl and Python requests

Tier 3: Arduino/ESP32

Connect Arduino with Ethernet shield or ESP32 with WiFi, send real sensor readings to database via HTTP POST requests

13.1 Introduction: From Ephemeral to Persistent Data

⚠ Forgetting Database Indexes = Queries That Never Finish

A table with 1 million sensor readings takes milliseconds to query with proper indexes, but *minutes* without them. The classic mistake: your system works fine in development with 1,000 rows, then grinds to a halt in production with 1,000,000.

What happens without indexes:

- `SELECT * FROM readings WHERE device_id=1` scans *every row*
- Query time grows linearly with table size ($O(n)$)
- Multiple concurrent queries compound the problem

The fix: Always index columns used in WHERE, JOIN, and ORDER BY. For IoT data, `CREATE INDEX ON readings(device_id, timestamp)` covers 90% of queries. But re-

member: indexes slow down INSERT—don't over-index high-frequency data.

Edge devices generate continuous sensor streams, but RAM is limited. We need persistent storage for:

- **Historical analysis:** Trends over hours, days, months
- **Offline operation:** Buffer data when network unavailable
- **Cross-device queries:** Compare data from multiple sensors
- **Compliance:** Audit trails for industrial/medical applications

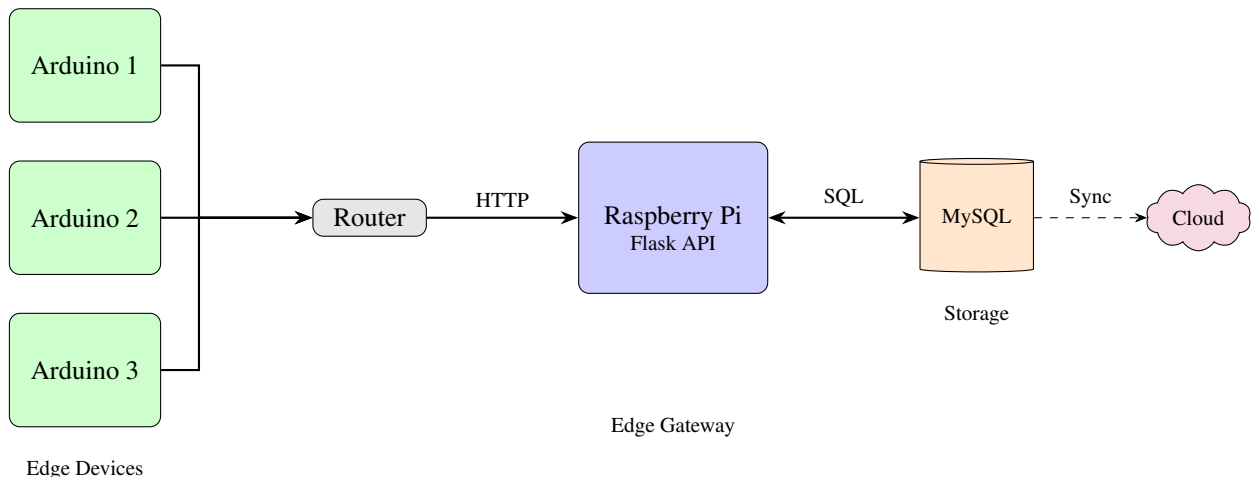


Figure 13.1: Edge-to-database data ingestion architecture

Why Not Store Directly to Cloud?

- **Latency:** Local storage responds in milliseconds; cloud in hundreds of ms
- **Reliability:** Network outages don't lose data
- **Cost:** Cloud storage charges per GB; local storage is one-time cost
- **Privacy:** Sensitive data stays on-premises

Edge-first storage with optional cloud sync offers the best of both worlds.

13.2 Distributed Systems Fundamentals

Edge computing is fundamentally a distributed system—multiple independent nodes (sensors, gateways, cloud servers) coordinating over unreliable networks. Understanding distributed systems theory helps you design robust, scalable IoT architectures.

13.2.1 The CAP Theorem

Definition — CAP Theorem. A distributed system can provide at most two of three guarantees:

- **Consistency (C):** Every read receives the most recent write or an error
- **Availability (A):** Every request receives a (non-error) response, without guarantee of most recent write
- **Partition Tolerance (P):** System continues operating despite arbitrary message loss or network failures

Since network partitions are inevitable in real-world systems (WiFi drops, cellular dead

zones, router failures), you *must* choose partition tolerance. The practical choice is between **CP** (sacrifice availability during partitions) or **AP** (sacrifice consistency).

Edge systems typically choose AP: Sensors continue logging data locally even when disconnected from the cloud, accepting that readings may be temporarily inconsistent across nodes.

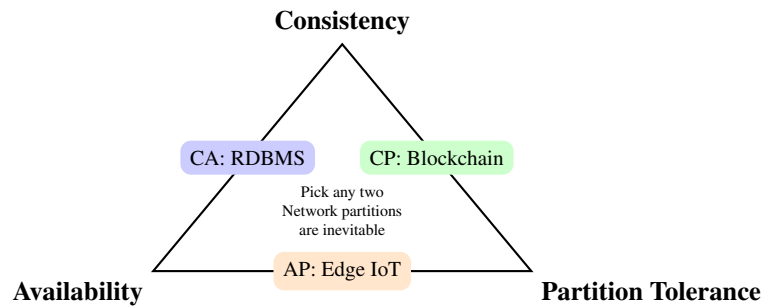


Figure 13.2: CAP theorem trade-off triangle

13.2.2 Data Consistency Models

Different consistency guarantees have different costs in latency and complexity.

Definition — Consistency Levels. Strong Consistency: All nodes see the same data at the same time. Requires coordination—reads may block until all replicas agree.

- Example: Banking transactions, inventory management
- Cost: High latency, reduced availability during network issues
- Implementation: Two-phase commit, Paxos/Raft consensus

Eventual Consistency: Given no new updates, all nodes eventually converge to the same state. Reads may return stale data temporarily.

- Example: Social media feeds, DNS, sensor aggregation
- Cost: Application must tolerate temporary inconsistencies
- Implementation: Gossip protocols, conflict-free replicated data types (CRDTs)

Causal Consistency: Preserves cause-effect relationships. If write A happened before write B, all nodes see A before B.

- Example: Chat messages, collaborative editing
- Cost: Moderate overhead to track causality
- Implementation: Vector clocks, Lamport timestamps

Consistency for Edge-Cloud Systems

Most edge IoT deployments use **eventual consistency**:

- Edge nodes write to local storage immediately (low latency)
- Background sync pushes data to cloud when connected
- Cloud aggregations may lag real-time by minutes/hours
- Critical control decisions use local data only

This model tolerates network failures gracefully—devices continue operating independently, syncing when connectivity returns.

13.2.3 Network Latency Mathematics

Understanding latency components helps you architect responsive systems.

End-to-End Latency Breakdown

Total latency for edge-to-cloud communication:

$$T_{\text{total}} = T_{\text{edge}} + T_{\text{network}} + T_{\text{cloud}} + T_{\text{response}}$$

where typical values are:

- T_{edge} : 10–50ms (local sensor reading + preprocessing)
- T_{network} : 50–200ms (round-trip to cloud, depends on distance)
- T_{cloud} : 100–500ms (cloud processing, varies by workload)
- T_{response} : 50–200ms (network return path)

Total: 210–950ms for cloud-based decision.

For real-time control (e.g., emergency stop on industrial machinery), this is unacceptable. Edge processing reduces latency to T_{edge} only: 10–50ms.

💡 Latency Budget Analysis

Calculate bandwidth and latency requirements for a sensor network:

Scenario: Factory with 100 temperature sensors sampling at 1 Hz. Each reading is 16 bytes (device ID, timestamp, temperature, checksum).

Bandwidth calculation:

$$\text{Bandwidth} = \text{Sample Rate} \times \text{Sample Size} \times \text{Num Sensors}$$

$$= 1 \text{ Hz} \times 16 \text{ bytes} \times 100 = 1600 \text{ bytes/sec} = 138 \text{ MB/day}$$

Over cellular (\$10/GB), that's \$1.38/day = \$42/month just for data transfer.

Edge aggregation alternative: Compute hourly averages locally (24 values/day/sensor). Transmit aggregates only.

$$\text{Bandwidth}_{\text{aggregated}} = 24 \times 16 \times 100 / 86400 = 0.44 \text{ bytes/sec} = 38 \text{ KB/day}$$

Cost reduction: 99.97% (\$0.01/month vs \$42/month)

Questions to consider:

1. What if sensors report only on significant changes (delta encoding)?
2. How much latency does hourly aggregation add to cloud visibility?
3. What happens when you need the raw data for debugging?

13.2.4 Additional Consistency Models

Beyond strong and eventual consistency, distributed systems employ intermediate models that balance performance and guarantees.

Read-Your-Writes Consistency

Guarantee: A process always sees its own writes reflected in subsequent reads, even if other nodes have not yet received the update.

Implementation:

- Client reads from same node it wrote to (session affinity)
- Track per-client version numbers/timestamps
- Local cache maintains writes until confirmed replicated

Use Case: User profile updates—user should immediately see their own edits, even if other users see stale data temporarily.

Formal Definition:

$$\text{If } \text{write}(x, v_1) \prec_{\text{po}} \text{read}(x) \implies \text{read}(x) \text{ returns } v_1 \text{ or later}$$

where $\prec_{\text{po}}$ denotes program order within a single client session.

Monotonic Reads

Guarantee: Once a process reads a value, subsequent reads never return older values (no "going back in time").

Mathematical Formulation:

$$\text{read}(x) \rightarrow v_i \wedge \text{read}(x) \rightarrow v_j \implies v_i \leq v_j$$

where reads occur in causal order.

Edge Example: Sensor dashboard showing temperature readings must not display 25°C at 10:00, then 23°C at 10:05 when 23°C was actually measured at 9:55.

13.2.5 Distributed Consensus Algorithms

Distributed systems need consensus algorithms to agree on shared state despite failures and network delays.

Paxos: The Classic Consensus Protocol

Purpose: Achieve agreement among distributed nodes on a single value, even when some nodes fail or messages are lost.

Key Roles:

- **Proposers:** Propose values
- **Acceptors:** Vote on proposals
- **Learners:** Learn the chosen value

Two-Phase Protocol:

1. **Prepare Phase:** Proposer sends proposal number n to acceptors
2. **Accept Phase:** If majority responds, proposer sends value

Safety Guarantee:

If value v is chosen, no other value $v' \neq v$ can ever be chosen

Limitation: Complex to implement correctly; infamous for being difficult to understand and deploy.

Raft: Consensus Made Understandable

Design Goal: Provide same guarantees as Paxos with simpler, more understandable algorithm.

Core Concepts:

- **Leader Election:** One node elected as leader; handles all client requests
- **Log Replication:** Leader replicates log entries to followers
- **Safety:** Committed entries survive leader failures

States: Each node is either Leader, Follower, or Candidate

Leader Election via Heartbeats:

- Leader sends periodic heartbeats to followers
- If follower timeout expires without heartbeat, becomes candidate
- Candidate requests votes from majority
- First candidate to get majority becomes leader

Log Consistency:

$$\forall i, j : \log[i].term = \log[j].term \implies \log[i].value = \log[j].value$$

Edge Application: Multi-gateway IoT deployments use Raft to elect primary gateway for cloud communication.

Byzantine Fault Tolerance (BFT)

Problem: Traditional consensus assumes fail-stop failures (nodes crash). BFT handles *arbitrary* failures—nodes may lie, send contradictory messages, or be malicious.

Byzantine Generals Problem: Generals must agree on attack plan, but some may be traitors sending false messages. How many traitors can be tolerated?

Fundamental Result:

BFT requires $n \geq 3f + 1$ nodes to tolerate f Byzantine failures

Why $3f+1$?

- Need majority to agree: $f + 1$ honest nodes
- Must overcome f Byzantine nodes voting against
- Need f more nodes to detect disagreement
- Total: $(f + 1) + f + f = 3f + 1$

Practical Byzantine Fault Tolerance (PBFT):

- Three-phase protocol: Pre-Prepare, Prepare, Commit
- Client waits for $f + 1$ matching replies before accepting
- Used in blockchain systems (proof-of-stake consensus)

Edge IoT Relevance: Critical infrastructure (power grid sensors, medical devices) may use BFT to prevent compromised devices from corrupting system state.

13.2.6 Quorum Systems

Quorum-based replication achieves fault tolerance without requiring all nodes to agree.

Definition — Read-Write Quorums. Replicate data across N nodes. Define:

- W = number of nodes that must acknowledge a write
- R = number of nodes that must respond to a read

Consistency Condition:

$$W + R > N$$

This ensures read and write quorums *overlap*, guaranteeing reads see most recent write.

Proof Intuition: If $W + R > N$, at least one node is in both quorums, so read finds latest value.

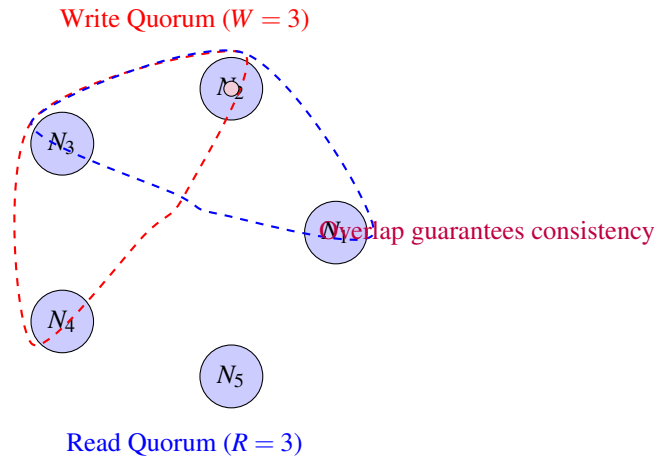


Figure 13.3: Quorum overlap with $N = 5, W = 3, R = 3$ (satisfies $W + R > N$)

Quorum Configuration Trade-offs

High Write Availability: $W = 1, R = N$

- Write succeeds if any one node is available
- Read must contact all nodes (expensive, less fault-tolerant)

High Read Availability: $W = N, R = 1$

- Write requires all nodes (low availability)
- Read from any single node (fast)
- Suitable for read-heavy workloads

Balanced: $W = R = \lceil (N + 1) / 2 \rceil$ (majority quorum)

- Symmetric read/write performance
- Tolerates $\lfloor N/2 \rfloor$ node failures
- Common configuration: $N = 5, W = 3, R = 3$

Edge Example: Sensor network with 5 gateways uses $W = 3, R = 3$ quorum. Any 3 gateways can process writes; any 3 can serve reads. System survives 2 gateway failures.

13.2.7 Data Partitioning Strategies

Large distributed datasets must be partitioned across nodes for scalability.

Definition — Partitioning Approaches. Horizontal Partitioning (Sharding): Split rows across nodes. Each node stores subset of records.

Example: Sensor readings partitioned by device ID:

- Node 1: device_id 1-100
- Node 2: device_id 101-200
- Node 3: device_id 201-300

Vertical Partitioning: Split columns across nodes. Each node stores subset of attributes.

Example:

- Node 1: (device_id, timestamp, temperature)
- Node 2: (device_id, timestamp, humidity, pressure)

Trade-off: Horizontal partitioning scales better; vertical partitioning reduces query load if columns accessed separately.

Consistent Hashing: The Key to Scalable Partitioning

Problem: Simple hash partitioning $\text{node} = h(\text{key}) \bmod N$ breaks when N changes (adding/removing nodes).

Solution: Map both nodes and keys to points on a circular hash ring.

Algorithm:

1. Hash ring: $[0, 2^{32} - 1]$ wraps around
2. Place nodes at hashed positions: $h(\text{node\_id})$
3. Place data at hashed positions: $h(\text{key})$
4. Key stored on first node clockwise from its position

Hash Function:

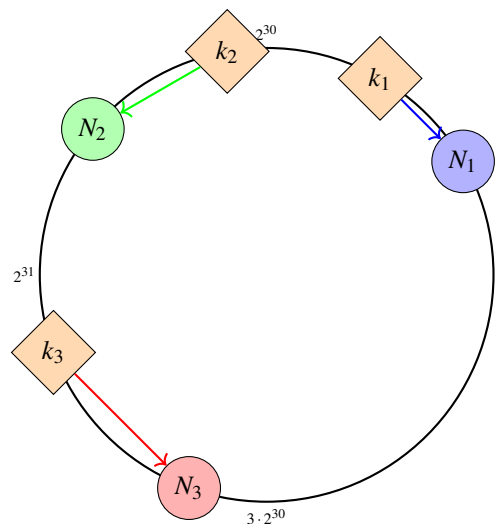
$$\text{node}(\text{key}) = \min\{h(n) \mid h(n) \geq h(\text{key}) \bmod 2^{32}\}$$

Advantage: Adding/removing node only affects neighboring keys, not entire dataset.

Load Balancing Issue: Simple consistent hashing may distribute keys unevenly.

Solution: Virtual Nodes:

- Each physical node maps to multiple virtual nodes
- Hash each virtual node separately: $h(\text{node\_id} \parallel i)$ for $i = 1 \dots V$
- Spreads load more evenly



Keys map to next clockwise node

Figure 13.4: Consistent hashing ring: keys stored on next node clockwise

13.2.8 Replication Strategies

Replication provides fault tolerance and improves read performance.

Definition — Replication Models. Primary-Backup (Master-Slave):

- One primary node handles all writes
- Writes propagate asynchronously to backups
- Reads can go to backups (eventual consistency)
- Simple but primary is single point of failure

Multi-Master (Active-Active):

- All nodes accept writes
- Conflict resolution required (Last-Write-Wins, CRDTs)
- Higher availability but more complex

Chain Replication:

- Nodes form chain: $N_1 \rightarrow N_2 \rightarrow N_3$
- Writes go to head, propagate down chain
- Reads from tail (guaranteed to see committed writes)
- Good balance of consistency and performance

13.2.9 Data Synchronization Patterns

How do edge and cloud databases stay synchronized?

Definition — Synchronization Strategies. Store-and-Forward:

- Edge buffers data locally in queue or database
- Background process sends batches when network available
- Resilient to intermittent connectivity
- Trade-off: Cloud data lags by sync interval

Conflict Resolution: When the same data is modified on edge and cloud, conflicts arise.

- **Last-Write-Wins (LWW):** Timestamp-based, simplest but loses data
- **Vector Clocks:** Track causality, detect concurrent modifications
- **Application-Specific:** Domain logic decides (e.g., max value wins for temperature)

Batch vs Streaming Sync:

- **Batch:** Accumulate N readings, send in one HTTP request. Efficient but adds latency.
- **Streaming:** Send each reading immediately via WebSocket/MQTT. Real-time but higher overhead.
- **Hybrid:** Stream critical events, batch routine telemetry

Conflict-Free Replicated Data Types (CRDTs)

Problem: Multi-master replication creates conflicts when same data modified concurrently.

CRDT Solution: Design data structures where concurrent updates *provably* converge without coordination.

Key Properties:

- **Commutativity:** $A \circ B = B \circ A$ (order doesn't matter)
- **Associativity:** $(A \circ B) \circ C = A \circ (B \circ C)$
- **Idempotence:** $A \circ A = A$ (duplicate delivery safe)

CRDT Examples:

Grow-Only Counter (G-Counter):

- Each node maintains local counter: count_i
- Increment only, never decrement
- Merge: $\text{count}_{\text{total}} = \sum_i \text{count}_i$
- Convergence guaranteed

Last-Write-Wins Register (LWW-Register):

- Each write tagged with timestamp
- Merge: $\text{value} = \arg \max_v \text{timestamp}(v)$
- Simple but loses concurrent writes

Observed-Remove Set (OR-Set):

- Set operations (add, remove)
- Each element tagged with unique ID
- Add wins over concurrent remove
- Complex but preserves all information

Edge IoT Application: Distributed sensor dashboard uses G-Counter CRDT to aggregate total event count across gateways. Each gateway increments locally; UI merges counts without coordination.

Offline-First Design Principles

Robust edge systems assume network is *unavailable by default*:

1. All critical logic runs locally on edge device
 2. Local storage buffers data during outages
 3. Sync process uses retry with exponential backoff
 4. Application UI shows sync status clearly
 5. Cloud enhances capabilities (analytics, dashboards) but isn't required for core operation
- This inverts the traditional cloud-centric model where devices are thin clients.

13.2.10 Communication Patterns in Distributed Systems

Different communication patterns suit different IoT requirements.

Definition — Publish-Subscribe Pattern. Concept: Publishers send messages to topics; subscribers receive messages from topics they've subscribed to. Publishers and subscribers are decoupled—they don't know about each other.

Components:

- **Publisher:** Produces messages tagged with topic
- **Broker:** Routes messages from publishers to subscribers
- **Subscriber:** Receives messages for subscribed topics

Advantages:

- Loose coupling: Publishers/subscribers can change independently
- Scalability: Add subscribers without modifying publishers
- Filtering: Subscribers only receive relevant messages

Protocols: MQTT, AMQP, Kafka

Edge IoT Example: Temperature sensors publish to `sensors/temperature/{device_id}`.

Multiple consumers subscribe: database logger, alerting system, dashboard.

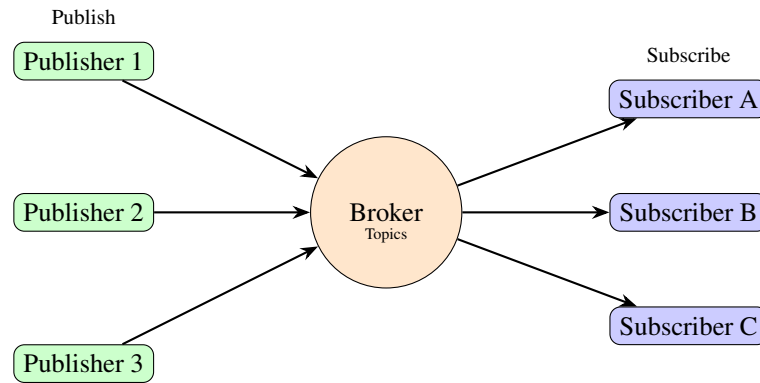


Figure 13.5: Publish-Subscribe decouples message producers from consumers

Definition — Request-Response Pattern. **Concept:** Client sends request to server; waits for response. Synchronous, tightly coupled communication.

Characteristics:

- Client blocks until response received (or timeout)
- Server must be available when request sent
- Direct one-to-one communication

Protocols: HTTP/REST, gRPC, RPC

Advantages:

- Simple: Easy to reason about
- Immediate feedback: Client knows if request succeeded
- Transaction semantics: Natural for CRUD operations

Disadvantages:

- Tight coupling: Client must know server address
- Blocking: Client waits for response
- No offline capability: Requires network connectivity

Edge IoT Example: Dashboard requests latest sensor readings via `GET /api/readings/{device_id}`.

Definition — Streaming Pattern. **Concept:** Long-lived connection continuously sends data from producer to consumer.

Variants:

- **Server-Sent Events (SSE):** Server pushes updates to client over HTTP
- **WebSockets:** Bidirectional, full-duplex communication
- **gRPC Streaming:** Unidirectional or bidirectional RPC streams

Advantages:

- Low latency: No connection overhead per message
- Real-time: Updates pushed immediately
- Efficient: Single connection for many messages

Disadvantages:

- **Stateful:** Must maintain connection state
- **Scalability:** One connection per client
- **Reconnection logic:** Handle network interruptions

Edge IoT Example: Real-time dashboard uses WebSocket to stream sensor readings as they arrive.

Definition — Message Queue Pattern. Concept: Asynchronous messaging via persistent queue. Producers enqueue messages; consumers dequeue.

Key Properties:

- **Persistence:** Messages stored until consumed
- **Decoupling:** Producer and consumer don't interact directly
- **Buffering:** Queue absorbs rate mismatches
- **Ordering:** FIFO or priority-based

Delivery Semantics:

- **At-most-once:** May lose messages (fast, no guarantees)
- **At-least-once:** May duplicate messages (requires idempotent consumers)
- **Exactly-once:** Each message delivered once (expensive, complex)

Protocols: RabbitMQ, Amazon SQS, Apache Kafka

Edge IoT Example: Edge gateway enqueues sensor batches; cloud worker consumes and processes asynchronously. Queue buffers data during cloud outages.

Choosing Communication Patterns for Edge IoT

Use Publish-Subscribe when:

- Multiple consumers need same data (fan-out)
- Consumers unknown at design time
- Need topic-based filtering

Use Request-Response when:

- Need immediate confirmation
- Simple CRUD operations
- Transactional semantics required

Use Streaming when:

- Real-time updates critical
- High-frequency data (video, audio, rapid sensor readings)
- Bidirectional communication needed

Use Message Queues when:

- Producers and consumers run at different rates
- Need persistence across failures
- Workflow/pipeline processing

Hybrid Approach: Many systems combine patterns. Example: MQTT pub-sub for sensor data ingestion, HTTP request-response for configuration queries, message queue for batch analytics processing.

13.2.11 Failure Handling in Distributed Systems

Failures are inevitable in distributed edge systems. Robust designs detect and handle them gracefully.

Heartbeats and Failure Detection

Problem: How does system detect when a node has failed?

Heartbeat Protocol:

1. Each node periodically sends HEARTBEAT message to monitor
2. Monitor maintains timestamp of last heartbeat per node
3. If time since last heartbeat > timeout, declare node failed

Timeout Calculation:

$$T_{\text{timeout}} = T_{\text{heartbeat}} \times k + T_{\text{network\_max}}$$

where $k \geq 3$ to avoid false positives from delayed messages.

Trade-offs:

- **Short timeout:** Faster failure detection, more false positives (network delays)
- **Long timeout:** Fewer false positives, slower recovery

Adaptive Timeouts:

$$T_{\text{timeout}} = \text{mean}(T_{\text{RTT}}) + 4 \times \text{stddev}(T_{\text{RTT}})$$

Adjusts to network conditions automatically.

Edge Example: Edge gateways send heartbeat to cloud every 30s. Cloud declares gateway offline after 120s without heartbeat (allows 3 missed heartbeats + network jitter).

Circuit Breaker Pattern

Problem: Cascading failures when calling a failing service repeatedly.

Circuit Breaker States:

- **Closed:** Normal operation, requests passed through
- **Open:** Failure threshold exceeded, reject requests immediately (fail fast)
- **Half-Open:** After timeout, allow limited requests to test recovery

State Transitions:

- Closed → Open: When failure rate > θ (e.g., 50% failures in last N requests)
- Open → Half-Open: After cooldown period T_{reset} (e.g., 60 seconds)
- Half-Open → Closed: If test requests succeed
- Half-Open → Open: If test requests fail

Benefits:

- Prevents resource exhaustion (don't waste threads waiting for timeout)
- Faster failure detection and response
- Gives failing service time to recover

Edge Example: Edge device calls cloud API. After 5 consecutive failures, circuit opens. Device uses cached configuration locally for 60s before retrying cloud.

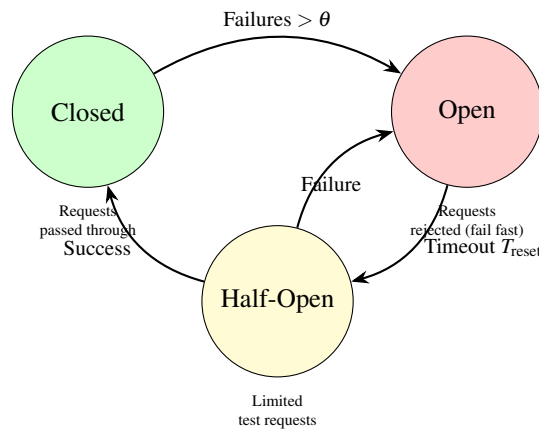


Figure 13.6: Circuit breaker state machine

Retry Strategies with Exponential Backoff

Problem: Transient failures (network glitches) should trigger retry; permanent failures shouldn't waste resources.

Exponential Backoff Algorithm:

1. Attempt operation
2. If fails, wait $T_{\text{base}} \times 2^{\text{attempt}}$ seconds
3. Retry up to `max_attempts`
4. Add jitter to prevent thundering herd

Backoff Formula:

$$T_{\text{wait}} = \min(T_{\text{max}}, T_{\text{base}} \times 2^n) + \text{random}(0, \text{jitter})$$

where:

- T_{base} : Initial retry delay (e.g., 1 second)
- n : Retry attempt number (0, 1, 2, ...)
- T_{max} : Cap on wait time (e.g., 60 seconds)
- jitter: Random offset (e.g., ± 0.5 seconds)

Example Sequence:

- Attempt 1 fails → wait 1s
- Attempt 2 fails → wait 2s
- Attempt 3 fails → wait 4s
- Attempt 4 fails → wait 8s
- Attempt 5 fails → wait 16s
- Attempt 6 fails → give up

Why Exponential?

- Transient failures resolve quickly (short initial wait)
- Prolonged outages need longer intervals (avoid hammering server)
- Exponential growth caps retry rate

Why Jitter? Multiple clients retrying simultaneously create thundering herd. Random jitter staggers retries.

Edge Example: Edge device uploads data to cloud. Network failure triggers retry with backoff: 1s, 2s, 4s, 8s, 16s, then store locally.

Idempotency for Reliable Retries

Problem: Retry after network timeout may duplicate operation (e.g., insert same sensor reading twice).

Solution: Idempotent Operations

An operation is **idempotent** if applying it multiple times has the same effect as applying it once:

$$f(f(x)) = f(x)$$

Making HTTP POST Idempotent:

1. Client generates unique request ID (UUID)
2. Include ID in POST request: {"request_id": "abc-123", "data": ...}
3. Server checks if request_id already processed
4. If yes: Return cached result (don't reprocess)
5. If no: Process request, cache result with request_id

Idempotency Examples:

- **Idempotent:** SET temperature=25 (repeated execution safe)
- **Idempotent:** DELETE device WHERE id=5 (deleting twice same as once)
- **Not idempotent:** INSERT INTO readings VALUES (...) (creates duplicate)
- **Not idempotent:** counter = counter + 1 (increments each time)

Edge IoT Application: Arduino sends sensor reading with UUID. Network fails after server receives but before Arduino gets response. Arduino retries with same UUID. Server detects duplicate UUID, returns success without re-inserting.

Definition — Timeout Hierarchy. Different operation types need different timeouts:

- **Connection timeout:** Time to establish TCP connection (e.g., 5s)
- **Request timeout:** Time for server to respond (e.g., 30s)
- **Idle timeout:** Time connection can be inactive (e.g., 60s)

Formula for Total Timeout:

$$T_{\text{total}} = T_{\text{connect}} + T_{\text{request}} + n \times T_{\text{retry}}$$

Edge Recommendation:

- Lightweight sensor POST: 5s total (2s connect + 3s request)
- Bulk upload: 60s total (handles network variability)
- Configuration fetch: 10s (fails fast, uses cached config)

Set timeouts *shorter* than worst-case latency to fail fast and retry or use local fallback.

13.2.12 Edge-Specific Distributed Patterns

Edge computing introduces unique distributed systems challenges due to geographic distribution and heterogeneous resources.

Latency-Aware Routing

Problem: Requests should be routed to nearest/fastest node to minimize latency.

Routing Strategies:

Geographic Routing:

- Map client location to nearest edge node
- Use GeoDNS or anycast routing
- Minimizes network distance

Latency-Based Routing:

- Measure actual RTT to candidate nodes
- Route to node with lowest latency
- Adapts to network conditions dynamically

Load-Aware Routing:

- Consider both latency and node load
- Weighted scoring: $\text{score} = \alpha \cdot \text{latency} + \beta \cdot \text{load}$
- Prevents hot spots

Example Routing Algorithm:

1. Client pings k nearest edge nodes (e.g., $k = 3$)
2. Measure RTT: $\{RTT_1, RTT_2, RTT_3\}$
3. Query load: $\{CPU_1, CPU_2, CPU_3\}$
4. Compute scores: $s_i = w_1 \cdot RTT_i + w_2 \cdot CPU_i$
5. Route to $\arg \min_i s_i$

Edge IoT Example: Mobile sensor device connects to nearest edge gateway. As device moves, routing updates to maintain low latency connection.

Edge-Cloud Coordination Patterns

Hierarchical Processing Pipeline:

Distribute computation across edge-fog-cloud tiers based on latency/compute requirements.

- **Tier 1 (Edge Device):** Immediate response (<10ms)
 - Sensor fusion, filtering
 - Safety-critical control
 - Simple anomaly detection
- **Tier 2 (Edge Gateway):** Local aggregation (10-100ms)
 - Multi-sensor correlation
 - Local ML inference
 - Time-series downsampling
- **Tier 3 (Fog Node):** Regional processing (100ms-1s)
 - Cross-gateway analytics
 - Model training/updates
 - Regional dashboards
- **Tier 4 (Cloud):** Global analytics (1s+)
 - Long-term storage
 - Deep learning training
 - Multi-site comparison

Data Flow:

Device $\xrightarrow{\text{streaming}}$ Gateway $\xrightarrow{\text{batch}}$ Fog $\xrightarrow{\text{async}}$ Cloud

Decision Placement: Process data at lowest tier that meets requirements—don't send to cloud what edge can handle.

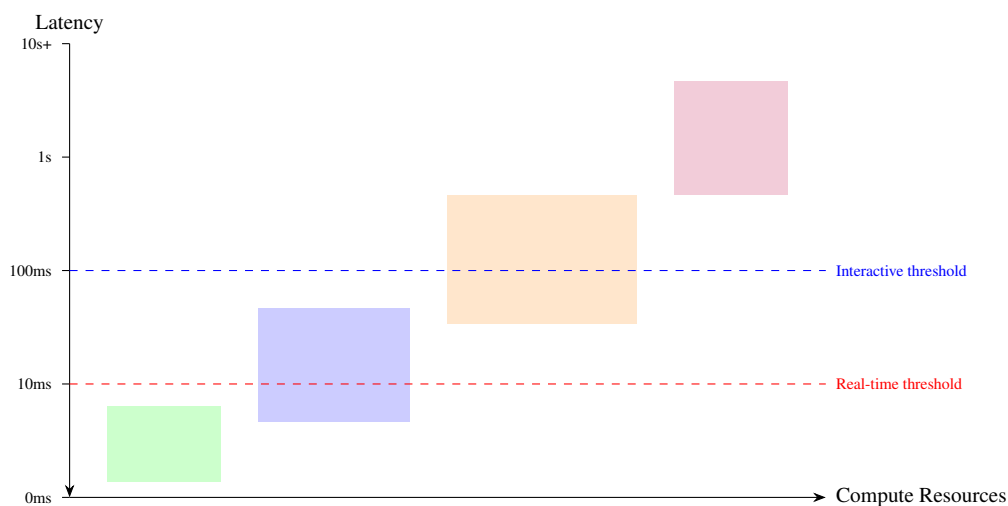


Figure 13.7: Computation placement: latency vs resources trade-off

Definition — Data Synchronization Strategies for Edge. Delta Synchronization:

- Send only changes since last sync, not full state
- Reduces bandwidth by 90-99% for slow-changing data
- Implementation: Track last sync timestamp, send records where `updated_at > last_sync`

Merkle Trees for Efficient Sync:

- Hash-based tree structure for detecting differences
- Compare root hashes to detect any differences ($O(1)$)
- If different, recursively compare subtrees to find changed data
- Used by databases (Cassandra), version control (Git)

Eventual Consistency with Versioning:

- Each update increments version number
- Sync sends: (key, value, version)
- Recipient keeps highest version number
- Conflicts detected when versions diverge

Edge Example: Gateway syncs to cloud hourly. Instead of uploading all 3600 readings, sends only 50 that exceeded alert thresholds. Cloud has eventual view of critical events.

CAP Theorem in Edge IoT Practice

Edge-Local Systems (AP):

- Each edge node operates independently
- Partitions tolerated (network outages common)
- Always available for local writes
- Eventually consistent across nodes
- Example: Edge devices continue logging during cloud outage

Critical Cloud Systems (CP):

- Device registration, user authentication
- Requires strong consistency
- Blocks during partitions (sacrifice availability)
- Example: Can't register new device without cloud confirmation

Design Pattern: Hybrid approach

- Core telemetry: AP (eventually consistent, always available)
- Configuration/state: CP (strongly consistent, may block)
- Separate data flows based on consistency requirements

13.3 Database Theory for Edge Systems

13.3.1 ACID vs BASE Properties

Definition — ACID (Traditional Databases). Strong consistency model for relational databases:

- **Atomicity:** Transactions complete fully or not at all
- **Consistency:** Database always in valid state
- **Isolation:** Concurrent transactions don't interfere
- **Durability:** Committed data survives crashes

Suitable for: Financial data, inventory, device registration

Definition — BASE (NoSQL/Distributed Systems). Eventual consistency model for scalable systems:

- **Basically Available:** System responds even during failures
- **Soft state:** State may change without input (background sync)
- **Eventual consistency:** Data converges over time

Suitable for: Sensor readings, logs, analytics

Choosing Database Type for Edge

Use SQL (SQLite, PostgreSQL, MySQL) when:

- Data has relational structure (devices, users, readings)
- Complex queries needed (JOINS, aggregations)
- Transactions required (device registration, configuration)
- Data size moderate (<100 GB on edge)

Use NoSQL (MongoDB, Cassandra) when:

- Schema changes frequently
- Horizontal scaling needed (multi-node clusters)
- Write throughput critical (millions of writes/sec)
- Data naturally document-oriented (JSON blobs)

Use Time-Series DB (InfluxDB, TimescaleDB) when:

- Data is timestamped sensor readings
- Downsampling/aggregation is primary query pattern
- Retention policies needed (auto-delete old data)
- Storage efficiency critical (compression for time-series)

13.3.2 Index Performance Analysis

Query Complexity with Indexes

Without index: Linear scan, $O(n)$ complexity

$$T_{\text{query}} = k \cdot n$$

where k is time to check one row, n is table size.

With B-tree index: Logarithmic lookup, $O(\log n)$ complexity

$$T_{\text{query}} = k \cdot \log_2 n$$

Example: Table with 1,000,000 rows, $k = 1\mu s$ per row check.

- Without index: $1,000,000 \times 1\mu s = 1$ second
- With index: $\log_2(1,000,000) \approx 20$ lookups = $20\mu s$
- **Speedup: 50,000×**

But indexes aren't free:

- Storage overhead: $\sim 10\text{--}20\%$ per index
- Insert penalty: Each insert updates all indexes
- Maintenance: Indexes need periodic optimization

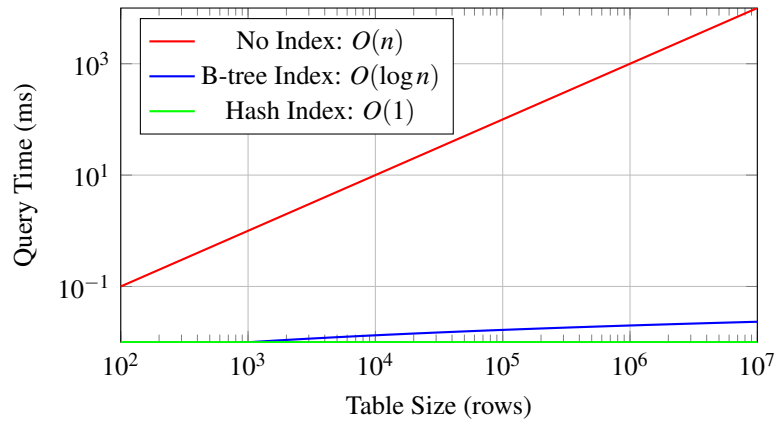
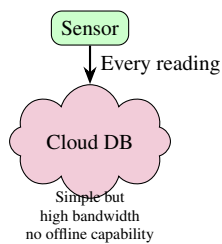


Figure 13.8: Query time complexity: indexed vs non-indexed tables

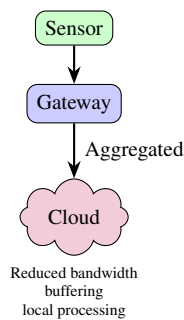
13.4 Edge-Cloud Architecture Patterns

13.4.1 Common Deployment Patterns

Pattern 1: Direct to Cloud



Pattern 2: Edge Gateway



Pattern 3: Hierarchical

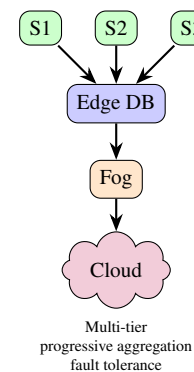


Figure 13.9: Three common edge-cloud deployment patterns

Data Aggregation Strategies

Reduce bandwidth and storage by aggregating data before cloud upload:

Temporal Aggregation:

- Downsample: Average every 10 readings into 1 (10× reduction)
- Time windows: Hourly min/max/avg (86400× reduction for 1 Hz data)
- Adaptive: Full resolution for anomalies, downsampled for normal operation

Spatial Aggregation:

- Average across multiple sensors in same room
- Compute statistics: mean, median, std dev across sensor array
- Delta encoding: Send only values that changed > threshold

Semantic Aggregation:

- Extract features: “temperature rising rapidly” instead of raw values
- Event detection: Send alert instead of continuous stream
- Compress: Run-length encoding for stable readings

Trade-off: Aggregation saves bandwidth but loses granularity. Design aggregation strategy based on query patterns.

13.4.2 Security Considerations for Distributed Systems

Security Threat Model

Distributed edge systems face unique security challenges:

Network Layer:

- Eavesdropping: Unencrypted sensor data on WiFi
- Man-in-the-middle: Attacker intercepts cloud API calls
- DDoS: Flooding edge gateway with requests
- Mitigation: TLS/HTTPS, mutual authentication, rate limiting

Device Layer:

- Physical tampering: Attacker gains access to edge device
- Firmware modification: Malware injected into sensor node
- Credential theft: API keys stored in plain text on device
- Mitigation: Secure boot, encrypted storage, hardware security modules

Application Layer:

- SQL injection: Malicious input in API parameters
- Unauthorized access: Missing authentication on API endpoints
- Data exfiltration: Compromised cloud account leaks sensor data
- Mitigation: Input validation, API keys, least-privilege access control

REST API Best Practices for Edge

Design robust APIs for unreliable networks:

Authentication:

- Use API keys in headers, not URL parameters (prevents logging)
- Rotate keys periodically
- Per-device keys enable revocation without affecting other devices

Rate Limiting:

- Limit requests per device: 100/minute prevents accidental loops
- Token bucket algorithm: Allow bursts but limit sustained rate
- Return HTTP 429 (Too Many Requests) when limit exceeded

Retry with Exponential Backoff:

- First retry: Wait 1 second
- Second retry: Wait 2 seconds
- Third retry: Wait 4 seconds
- Max: Wait 60 seconds, then give up or store locally

Idempotency for Reliability:

- POST requests include unique ID (UUID)
- Server deduplicates: ignore duplicate IDs
- Retries don't create duplicate database entries
- Critical for unreliable networks where response may be lost

13.5 Database Schema Design for IoT

A well-designed schema makes queries fast and storage efficient.

13.5.1 Core Tables

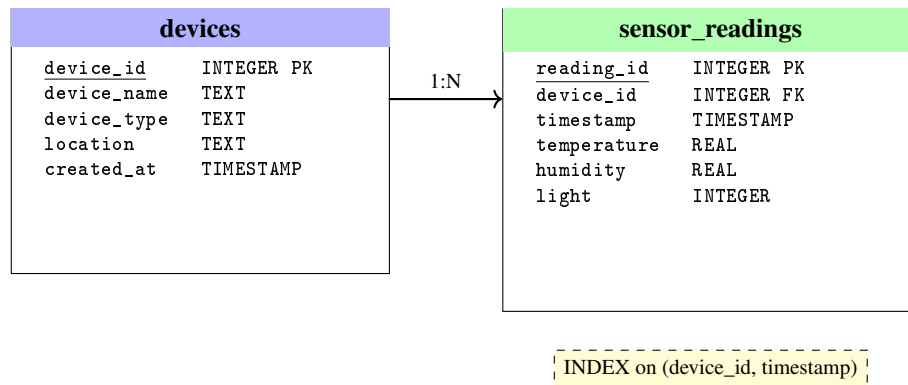


Figure 13.10: Entity-relationship diagram for sensor data

```

1 import sqlite3
2
3 # Create connection (use ':memory:' for testing)
4 conn = sqlite3.connect('sensors.db')
5 cursor = conn.cursor()
6
7 # Create devices table
8 cursor.execute('''
9 CREATE TABLE IF NOT EXISTS devices (
10     device_id INTEGER PRIMARY KEY AUTOINCREMENT,
11     device_name TEXT NOT NULL UNIQUE,
12     device_type TEXT,
13     location TEXT,
14     created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
15 )
16 ''')
```

Listing 13.1: Creating the database schema

Code Walkthrough

PRIMARY KEY AUTOINCREMENT Auto-generates unique IDs

NOT NULL UNIQUE Device names must exist and be unique

DEFAULT CURRENT_TIMESTAMP Auto-fills creation time

```

1 # Create sensor readings table
2 cursor.execute('''
3 CREATE TABLE IF NOT EXISTS sensor_readings (
4     reading_id INTEGER PRIMARY KEY AUTOINCREMENT,
5     device_id INTEGER NOT NULL,
6     timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
7     temperature REAL,
8     humidity REAL,
9     light INTEGER,
10     FOREIGN KEY (device_id) REFERENCES devices(device_id)
11 )
12 ''')
13
14 # Create indexes for fast queries
```

```

15 cursor.execute('''
16 CREATE INDEX IF NOT EXISTS idx_readings_device_time
17 ON sensor_readings(device_id, timestamp)
18 ''')
19
20 conn.commit()
21 print("Schema created successfully!")

```

Listing 13.2: Creating sensor readings table with indexes

Code Walkthrough

FOREIGN KEY Ensures referential integrity—can't insert reading for non-existent device

CREATE INDEX Dramatically speeds up queries filtering by device and time range

REAL SQLite's floating-point type for temperature/humidity

Index Trade-offs

Indexes speed up **SELECT** queries but slow down **INSERT** operations. For high-frequency sensor data:

- Index columns used in **WHERE** and **ORDER BY**
- Don't index columns rarely queried
- Consider composite indexes for multi-column filters

The `(device_id, timestamp)` index covers most IoT query patterns.

13.6 Data Ingestion Patterns

How data flows from sensor to database.

13.6.1 Single Insert vs Batch Insert

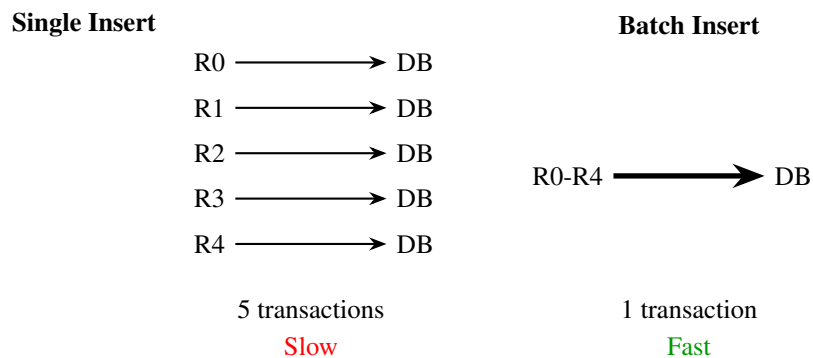


Figure 13.11: Single vs batch insert performance

```

1 class SensorDataManager:
2     """Manages sensor data storage and retrieval."""
3
4     def __init__(self, connection):
5         self.conn = connection
6         self.cursor = connection.cursor()
7

```

```

8     def register_device(self, name, device_type, location):
9         """Register a new IoT device."""
10        self.cursor.execute(
11            '''INSERT INTO devices (device_name, device_type, location)
12               VALUES (?, ?, ?)''',
13            (name, device_type, location)
14        )
15        self.conn.commit()
16        return self.cursor.lastrowid

```

Listing 13.3: Data manager class with batch insert

```

1     def insert_reading(self, device_id, temp, humidity, light):
2         """Insert a single sensor reading."""
3         self.cursor.execute(
4             '''INSERT INTO sensor_readings
5                (device_id, temperature, humidity, light)
6                VALUES (?, ?, ?, ?)''',
7             (device_id, temp, humidity, light)
8         )
9         self.conn.commit()
10        return self.cursor.lastrowid
11
12    def insert_batch(self, readings):
13        """Insert multiple readings efficiently."""
14        self.cursor.executemany(
15            '''INSERT INTO sensor_readings
16               (device_id, timestamp, temperature, humidity, light)
17               VALUES (?, ?, ?, ?, ?)''',
18            readings
19        )
20        self.conn.commit()
21        return len(readings)

```

Listing 13.4: Single and batch insert methods

Code Walkthrough

`execute(..., (params,))` Single insert with parameter binding
`executemany(..., list)` Batch insert—one transaction for all rows
`self.cursor.lastrowid` Returns the auto-generated ID
`conn.commit()` Persist changes to disk

When to Use Batch Insert

- Edge gateway buffering data from multiple sensors
 - Replaying buffered data after network recovery
 - Initial data import or migration
 - Any time you have multiple readings ready
- Batch inserts can be 10-100× faster than individual inserts.

13.7 Querying Sensor Data

SQL enables powerful analysis of historical sensor data.

13.7.1 Basic Queries

```
1 import pandas as pd
2
3 # Latest 10 readings across all devices
4 query = '''
5 SELECT d.device_name, d.location,
6         r.temperature, r.humidity, r.light,
7         r.timestamp
8 FROM sensor_readings r
9 JOIN devices d ON r.device_id = d.device_id
10 ORDER BY r.timestamp DESC
11 LIMIT 10
12 '''
13
14 df = pd.read_sql_query(query, conn)
15 print(df)
```

Listing 13.5: Retrieving latest readings

Code Walkthrough

`JOIN ... ON` Combines readings with device metadata
`ORDER BY ... DESC` Newest first
`LIMIT 10` Restrict result size—important for large tables!
`pd.read_sql_query` Returns results as DataFrame for analysis

13.7.2 Aggregation Queries

```
1 query = '''
2 SELECT
3     d.device_name,
4     strftime('%H', r.timestamp) AS hour,
5     COUNT(*) AS num_readings,
6     ROUND(AVG(r.temperature), 2) AS avg_temp,
7     ROUND(MIN(r.temperature), 2) AS min_temp,
8     ROUND(MAX(r.temperature), 2) AS max_temp,
9     ROUND(AVG(r.humidity), 2) AS avg_humidity
10 FROM sensor_readings r
11 JOIN devices d ON r.device_id = d.device_id
12 WHERE r.timestamp >= datetime('now', '-24 hours')
13 GROUP BY d.device_name, hour
14 ORDER BY d.device_name, hour
15 '''
16
17 df_hourly = pd.read_sql_query(query, conn)
```

Listing 13.6: Hourly statistics per device

Code Walkthrough

`strftime('%H', timestamp)` Extract hour from timestamp
`COUNT(*)` Number of readings in each group
`AVG, MIN, MAX` Aggregate functions compute statistics
`GROUP BY` Create groups for aggregation

```
datetime('now', '-24 hours') Filter to last 24 hours
```

Common IoT Query Patterns

- **Latest reading:** ORDER BY timestamp DESC LIMIT 1
- **Time range:** WHERE timestamp BETWEEN ? AND ?
- **Anomalies:** WHERE temperature > (SELECT AVG(...) + 2*STDDEV(...))
- **Downsampling:** GROUP BY strftime('%Y-%m-%d %H', timestamp)

13.8 HTTP API for Data Ingestion

A REST API enables any HTTP-capable device to submit data.

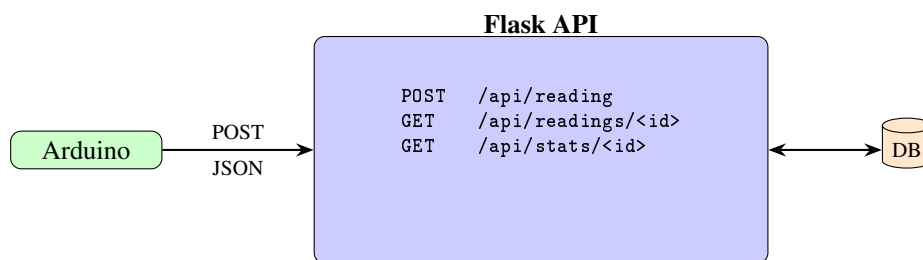


Figure 13.12: RESTful API architecture

```

1 from flask import Flask, request, jsonify
2 import sqlite3
3
4 app = Flask(__name__)
5 DB_PATH = 'sensors.db'
6
7 def get_db():
8     """Get database connection with row factory."""
9     conn = sqlite3.connect(DB_PATH)
10    conn.row_factory = sqlite3.Row # Access by column name
11    return conn
  
```

Listing 13.7: Flask API setup - sensor_api.py

```

1 @app.route('/api/reading', methods=['POST'])
2 def add_reading():
3     """Endpoint for Arduino to POST sensor data."""
4
5     # Accept both JSON body and query parameters
6     data = request.json or request.args
7
8     # Validate required fields
9     required = ['device_id', 'temperature', 'humidity']
10    if not all(key in data for key in required):
11        return jsonify({'error': 'Missing required fields'}), 400
12
13    # Insert into database
14    conn = get_db()
15    cursor = conn.cursor()
16    cursor.execute(
17        '''INSERT INTO sensor_readings
  
```

```

18         (device_id, temperature, humidity, light)
19         VALUES (?, ?, ?, ?)'''
20     (data['device_id'], data['temperature'],
21      data['humidity'], data.get('light', 0))
22 )
23 conn.commit()
24
25 return jsonify({
26     'status': 'ok',
27     'reading_id': cursor.lastrowid
28 })

```

Listing 13.8: POST endpoint for receiving sensor data

Code Walkthrough

`@app.route('/api/reading', methods=['POST'])` Decorator defines URL and allowed HTTP method

`request.json` or `request.args` Accept data in body or URL parameters

`return jsonify(...), 400` Return error with HTTP status code 400 (Bad Request)

`data.get('light', 0)` Optional field with default value

```

1 @app.route('/api/readings/<int:device_id>', methods=['GET'])
2 def get_readings(device_id):
3     """Get recent readings for a specific device."""
4
5     limit = request.args.get('limit', 100, type=int)
6
7     conn = get_db()
8     cursor = conn.cursor()
9     cursor.execute(
10         '''SELECT * FROM sensor_readings
11            WHERE device_id = ?
12            ORDER BY timestamp DESC
13            LIMIT ?'''
14         (device_id, limit)
15     )
16
17     readings = [dict(row) for row in cursor.fetchall()]
18     return jsonify(readings)

```

Listing 13.9: GET endpoint for retrieving readings

```

1 if __name__ == '__main__':
2     app.run(host='0.0.0.0', port=5000, debug=True)

```

Listing 13.10: Running the Flask server

Code Walkthrough

`<int:device_id>` URL parameter converted to integer

`request.args.get('limit', 100, type=int)` Optional query parameter with default

`host='0.0.0.0'` Listen on all network interfaces (required for external access)

`debug=True` Auto-reload on code changes (disable in production!)

API Security

This example lacks authentication! For production:

- Add API key validation in headers
- Use HTTPS to encrypt data in transit
- Rate-limit requests to prevent DoS
- Validate and sanitize all input data

💡 When Does Edge-First Make Sense?

Consider these three IoT deployments:

Scenario A: Smart building with 100 temperature sensors, reliable WiFi, facility management dashboard.

Scenario B: Agricultural monitoring across 50 fields, cellular connectivity (expensive, unreliable), weekly analysis.

Scenario C: Industrial press machine, safety-critical anomaly detection, must respond in <10ms.

For each scenario, consider:

1. What happens during a 4-hour network outage?
2. What's the cost per MB of data transmission?
3. What's the acceptable latency for decisions?
4. Who needs access to the data, and from where?

Scenario A might work cloud-first. Scenario B benefits from edge aggregation to reduce cellular costs. Scenario C *requires* edge processing—you can't afford 100ms round-trip latency for safety decisions.

13.9 Arduino HTTP Client

The Arduino sends sensor data to our API via HTTP.

13.9.1 Hardware Setup

- Arduino Uno with Ethernet Shield 2 stacked on top
- DHT11 temperature/humidity sensor on pin 2
- Ethernet cable to router/switch

13.9.2 Arduino Sketch

```

1 #include <SPI.h>
2 #include <Ethernet.h>
3 #include "DHT.h"
4
5 // Network configuration
6 byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
7 IPAddress server(192, 168, 1, 100); // Pi IP address
8 const int port = 5000;
9
10 // Sensor configuration
11 #define DHTPIN 2
12 #define DHTTYPE DHT11
13 #define DEVICE_ID 1
14
15 DHT dht(DHTPIN, DHTTYPE);

```

```
16 EthernetClient client;
```

Listing 13.11: Includes and configuration

Code Walkthrough

`byte mac[]` Unique MAC address for Ethernet shield (change if multiple Arduinos!)

`IPAddress server` IP address of Raspberry Pi running Flask

`DEVICE_ID` Must match a registered device in database

```
1 void setup() {
2     Serial.begin(9600);
3     dht.begin();
4
5     Serial.println(F("Initializing Ethernet..."));
6
7     if (Ethernet.begin(mac) == 0) {
8         Serial.println(F("DHCP failed, using static IP"));
9         IPAddress ip(192, 168, 1, 177);
10        Ethernet.begin(mac, ip);
11    }
12
13    Serial.print(F("IP address: "));
14    Serial.println(Ethernet.localIP());
15
16    delay(1000); // Give Ethernet time to initialize
17 }
```

Listing 13.12: Setup function

Code Walkthrough

`F("...")` Store string in flash memory—saves precious RAM!

`Ethernet.begin(mac)` Attempt DHCP; returns 0 on failure

`Ethernet.begin(mac, ip)` Fall back to static IP

`Ethernet.localIP()` Print assigned IP for debugging

```
1 void sendData(float temperature, float humidity) {
2     if (client.connect(server, port)) {
3         Serial.println(F("Connected to server"));
4
5         // Build HTTP GET request with query parameters
6         client.print(F("GET /api/reading?device_id="));
7         client.print(DEVICE_ID);
8         client.print(F("&temperature="));
9         client.print(temperature, 1); // 1 decimal place
10        client.print(F("&humidity="));
11        client.print(humidity, 1);
12        client.println(F(" HTTP/1.1"));
13
14        client.print(F("Host: "));
15        client.println(server);
16        client.println(F("Connection: close"));
17        client.println();
18    }
```

Listing 13.13: Function to send data to API

```

1      // Wait for and print response
2      unsigned long timeout = millis() + 5000;
3      while (client.connected() && millis() < timeout) {
4          if (client.available()) {
5              char c = client.read();
6              Serial.print(c);
7          }
8      }
9
10     client.stop();
11     Serial.println(F("\nData sent successfully"));
12 } else {
13     Serial.println(F("Connection failed"));
14 }
15 }

```

Listing 13.14: Reading response and cleanup

Code Walkthrough

`client.connect(server, port)` Establish TCP connection

`client.print(...)` Build HTTP request piece by piece

HTTP/1.1 Protocol version (space before it is required!)

Connection: close Close connection after response

`millis() + 5000` 5-second timeout prevents infinite wait

```

1 void loop() {
2     float humidity = dht.readHumidity();
3     float temperature = dht.readTemperature();
4
5     if (!isnan(humidity) && !isnan(temperature)) {
6         Serial.print(F("Temp: "));
7         Serial.print(temperature);
8         Serial.print(F("C, Humidity: "));
9         Serial.print(humidity);
10        Serial.println(F("%"));
11
12        sendData(temperature, humidity);
13    } else {
14        Serial.println(F("Failed to read DHT sensor"));
15    }
16
17    delay(30000); // Send every 30 seconds
18 }

```

Listing 13.15: Main loop

Code Walkthrough

`isnan()` Check for sensor read failure (returns NaN)

`delay(30000)` 30-second interval between readings

Optimizing Arduino Memory

Use these techniques to reduce memory usage:

- F("string") for string literals (saves RAM)
- Avoid String class—use char[] arrays
- Minimize local variables in functions
- Use PROGMEM for large constant data

Check memory with: `Serial.println(freeMemory());`

13.10 Edge vs Cloud Storage Trade-offs

Where should you store IoT data? It depends on your requirements.

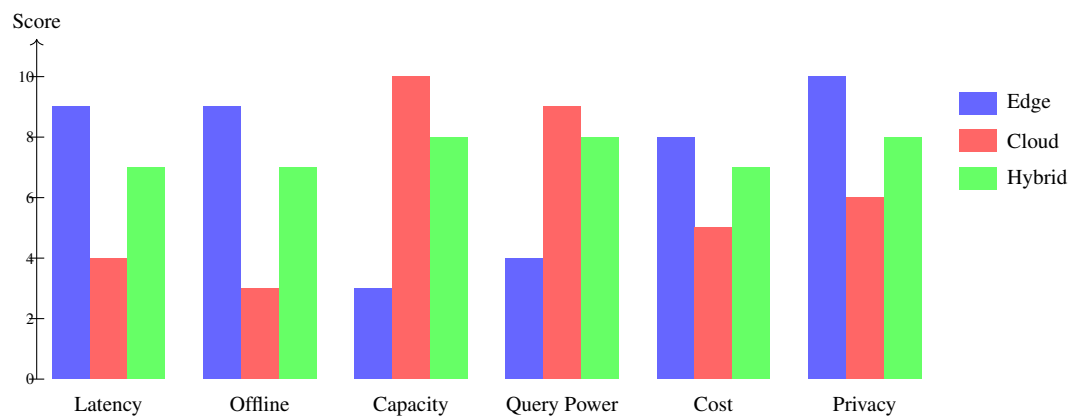


Figure 13.13: Trade-off comparison: Edge vs Cloud vs Hybrid storage

When to Use Each Approach

Edge-Only (SQLite on Pi):

- Real-time control systems (low latency critical)
- Privacy-sensitive applications (medical, personal)
- Remote sites with unreliable connectivity

Cloud-Only (MySQL/PostgreSQL on server):

- Multi-site aggregation and comparison
- Long-term storage (years of data)
- Complex analytics and machine learning

Hybrid (Edge + Cloud sync):

- Best overall balance
- Edge for real-time, cloud for analytics
- Graceful degradation during network outages

13.11 Three-Tier Implementation Guide

Use SQLite in-memory. Practice schema design, queries, aggregations.

Tier 1: Notebook Simulation

Run Flask API on Pi/laptop. Test with curl or Python requests library.

Tier 2: Python API + Database

Deploy Arduino + Ethernet + DHT11. Real sensor data to real database.

Tier 3: Full Hardware Stack

13.11.1 Tier 1: Notebook Practice

Run LAB13_distributed.ipynb to:

- Create schema and insert synthetic data
- Practice SQL queries with pandas
- Visualize hourly patterns

13.11.2 Tier 2: API Development

1. Create `sensors.db` with schema
2. Run `python sensor_api.py`
3. Test with curl:
`curl "http://localhost:5000/api/reading?device_id=1&temperature=22.5&humidity=65"`
4. Query: `curl "http://localhost:5000/api/readings/1?limit=5"`

13.11.3 Tier 3: Full Deployment

1. Configure Arduino with your Pi's IP address
2. Ensure Ethernet shield has unique MAC
3. Upload sketch and monitor Serial output
4. Verify data appears in database

13.12 Troubleshooting Guide

⚠ Database Connection Failed

Symptoms: `OperationalError: unable to connect` or `Access denied for user`

Cause: Wrong credentials, firewall blocking, or database server not running

Solution:

1. Verify server running: Check PostgreSQL/MySQL service status
2. Test connection string: Use `psql` or `mysql` command-line tool first
3. Check firewall: Ensure port 5432 (PostgreSQL) or 3306 (MySQL) open
4. Verify credentials: Username, password, database name all correct

5. For InfluxDB: Ensure token/API key has write permissions
6. Use localhost vs 127.0.0.1: Some systems treat these differently

⚠ Slow Query Performance / Timeouts

Symptoms: Queries take >10 seconds, database locks up, timeout errors

Cause: Missing indexes, full table scans, or inefficient query structure

Solution:

1. Add indexes: `CREATE INDEX idx_timestamp ON readings(timestamp);`
2. Use EXPLAIN: Analyze query execution plan
3. Limit result set: Add `LIMIT 1000` to large queries
4. Partition tables: Split by time range (daily/monthly tables)
5. Use prepared statements: Avoid SQL injection and improve performance
6. Enable query caching: For repeated queries

⚠ Data Insertion Too Slow

Symptoms: Sensor data accumulates faster than it can be written, backlog grows

Cause: Individual inserts, network latency, or synchronous writes

Solution:

1. Use batch inserts: `executemany()` for 10-100x speedup
2. Disable auto-commit: Wrap batches in transactions
3. Use asynchronous writes: Queue data, write in background thread
4. Reduce logging: Disable verbose database logs during bulk operations
5. Consider time-series DB: InfluxDB optimized for sensor data
6. Use connection pooling: Reuse connections instead of creating new ones

13.13 Summary

- **Schema Design:** Separate devices and readings tables with foreign key relationship. Index timestamp and device_id for fast queries.
- **Batch Inserts:** Use `executemany()` for 10-100× better performance than individual inserts.
- **SQL Aggregations:** `GROUP BY` with `AVG`, `MIN`, `MAX` enable hourly/daily statistics without loading all data.
- **REST API:** Flask provides simple HTTP endpoints. Accept both JSON and query parameters for flexibility.
- **Arduino HTTP:** Build GET requests with `client.print()`. Use `F()` macro to save RAM.
- **Storage Trade-offs:** Edge for low latency and privacy; cloud for capacity and analytics; hybrid for best overall balance.

13.14 Guided Experiments

Database Index Performance

Hypothesis: Adding a composite index on (device_id, timestamp) will dramatically reduce query time for time-range queries, with speedup increasing proportionally to table size.

Procedure:

1. Create sensor_readings table without indexes

2. Insert 10,000 readings from 5 devices
3. Measure query time: `SELECT * FROM sensor_readings WHERE device_id=1 AND timestamp > datetime('now', '-1 hour')`
4. Add index: `CREATE INDEX idx_device_time ON sensor_readings(device_id, timestamp)`
5. Repeat query and measure time
6. Repeat with 50,000 and 100,000 readings

Data Collection:

Table Size	Without Index (ms)	With Index (ms)	Speedup
10,000 rows			
50,000 rows			
100,000 rows			

Analysis Questions:

- Does the speedup increase with table size? Why or why not?
- What is the time complexity of full table scan vs indexed lookup?
- Measure index creation time. Is there an overhead to maintaining indexes?
- What would happen with a query on `device_id` only vs `timestamp` only?

Batch Insert Performance

Hypothesis: Batch inserting N rows is faster than N individual inserts by a factor proportional to database transaction overhead, with performance gains saturating beyond a certain batch size.

Procedure:

1. Prepare 1000 sensor readings in memory
2. Method 1: Insert one-by-one with individual `execute()` and `commit()` calls
3. Method 2: Insert one-by-one but single `commit()` at end
4. Method 3: Use `executemany()` with batch sizes: 10, 50, 100, 500, 1000
5. Measure total time for each method

Data Collection:

Method	Total Time (s)	Inserts/sec
Individual inserts + commits		
Individual inserts + single commit		
Batch size 10		
Batch size 50		
Batch size 100		
Batch size 500		
Batch size 1000		

Analysis Questions:

- What is the primary bottleneck: commits or SQL parsing?
- Plot throughput vs batch size. Where do gains diminish?
- For real-time IoT with readings arriving at 10/sec, what batch size balances throughput and latency?
- Would results differ on MySQL vs SQLite? Why?

13.15 Exercises

Self-Assessment Questions

Test your understanding before continuing:

1. Why do we create an index on (device_id, timestamp) in the database? How does this improve query performance for time-series sensor data?
2. Explain the difference between `execute()` and `executemany()` in Python SQLite. For 1000 inserts, estimate the performance difference.
3. Why does the Arduino sketch use `F("string")` instead of `"string"` for constant strings? Where are these stored differently and why does it matter?
4. What HTTP status code should the API return for missing required fields? What about server errors? Explain the difference between 4xx and 5xx codes.
5. Compare edge-only, cloud-only, and hybrid storage architectures. For a smart building with 100 sensors, which would you choose and why? Consider latency, cost, and privacy.

Add Data Retention

Implement automatic deletion of old data:

- Add API endpoint: `DELETE /api/cleanup`
- Delete readings older than 30 days
- Return count of deleted rows
- Bonus: Make retention period configurable

Implement POST with JSON Body

Modify the Arduino sketch to send JSON instead of URL parameters:

- Use POST instead of GET
- Set Content-Type: `application/json`
- Build JSON string: `{"device_id":1,"temperature":22.5}`
- Update API to prefer JSON body

Add Alerting

Create an alert system for anomalies:

- API endpoint checks if latest reading is anomalous
- Compare against rolling average (last 100 readings)
- Return alert if temperature > 2 standard deviations from mean
- Bonus: Send email or push notification

- **Further Reading.**
- Arduino Ethernet Library: <https://www.arduino.cc/en/Reference/Ethernet>
 - Flask Quickstart: <https://flask.palletsprojects.com/quickstart/>

- SQLite Documentation: <https://www.sqlite.org/docs.html>
- InfluxDB for Time-Series: <https://www.influxdata.com/>
- TimescaleDB: <https://www.timescale.com/>

— **Credits.** Adapted from IoT coursework materials with expanded pedagogical content.

13.16 Exercises

1. **[Challenge] Challenge: Multi-Device Sensor Network with Aggregation:** Deploy a distributed sensor network with at least 3 edge nodes (Arduino or ESP32) sending data to a centralized database via REST API. Implement device registration, health monitoring (last-seen timestamps), and aggregate queries that compute hourly statistics across all devices. Add a web dashboard that displays per-device metrics and network-wide trends. Test fault tolerance by disconnecting one device and verifying the system continues operating.
2. **[Challenge] Challenge: Edge-Cloud Hybrid Architecture with Local Buffering:** Design a hybrid storage system where edge devices store data locally in SQLite during normal operation and sync to cloud PostgreSQL every hour. Implement bidirectional sync: edge pushes new readings, cloud returns configuration updates. Handle network failures gracefully with exponential backoff retry logic. Measure and compare query latency for recent data (edge-local) vs. historical data (cloud). Analyze bandwidth usage and identify optimal sync intervals.
3. **[Challenge] Challenge: Time-Series Database Migration and Performance Comparison:** Migrate your sensor data from SQLite to a specialized time-series database (InfluxDB or TimescaleDB). Implement identical queries in both systems: last 24 hours average, hourly rollups for past month, anomaly detection using Z-scores. Benchmark query performance, storage efficiency, and ingestion rates for both databases at 100 readings/sec. Document when the complexity of a specialized time-series DB is justified vs. using standard SQL.
4. **[Challenge] Challenge: Secure API with Authentication and Rate Limiting:** Enhance your Flask API with production-grade security features. Implement API key authentication using Flask-HTTPAuth, per-device rate limiting (max 60 requests/minute), input validation preventing SQL injection, and HTTPS using self-signed certificates. Add comprehensive logging of all API requests with timestamps, device IDs, and response codes. Test security by attempting unauthorized access, rate limit violations, and malformed requests. Verify logs capture all violations.



14. Anomaly Detection on Edge Devices

Learning Objectives

After completing this lab, you will be able to:

- Understand anomaly detection approaches and their suitability for edge deployment
- Implement lightweight statistical methods (Z-score, moving average) for microcontrollers
- Apply K-means clustering for unsupervised anomaly detection
- Build and deploy autoencoder-based anomaly detectors
- Evaluate trade-offs between detection accuracy, memory usage, and latency
- Choose appropriate methods based on resource constraints

Interactive Notebook

Jupyter Notebook: LAB14_anomaly_detection.ipynb

[Open in Google Colab](#) | [View on GitHub](#) | [Companion Website](#)

Prerequisites

- **Knowledge:** ML basics (LAB02), TFLite conversion (LAB03)
- **Level 1:** Laptop with TensorFlow and scikit-learn
- **Level 2:** Raspberry Pi for autoencoder benchmarking
- **Level 3:** Arduino for Z-score detector, ESP32 for autoencoder

Theory and Practice. Anomaly detection identifies data points that deviate from expected patterns—critical for predictive maintenance, security, and quality control. This lab progresses

from simple statistical methods suitable for tiny microcontrollers to neural network approaches for more capable edge devices, and connects directly to the streaming pipelines of LAB12 and the storage/query patterns of LAB13 (where anomalies may trigger alerts or retention policies).

Execution Tiers. Level 1: Explore all methods in notebooks with synthetic or recorded sensor data. Level 2: Convert an autoencoder (or other model) to TFLite and benchmark on a Raspberry Pi or similar edge node. Level 3: Deploy a Z-score detector on Arduino-class MCUs and an autoencoder-based detector on ESP32/Raspberry Pi, measuring memory, latency, and (optionally, via LAB11/LAB15) power.

Troubleshooting. High false positives? Lower sensitivity threshold or increase training data. Missed anomalies? Check if training data truly represents “normal.” Autoencoder not converging? Normalize inputs and reduce model size.

Prerequisites

Before starting this lab, ensure you:

- Completed LAB02 (Neural Network Training Fundamentals): ML basics and model training
- Completed LAB13 (Edge-to-Cloud Data Ingestion): understanding of distributed data systems
- Python data processing basics: familiarity with NumPy, pandas, and scikit-learn
- Understanding of training, evaluation, and model performance metrics

Three-Tier Execution Model

Tier 1: Jupyter/Colab

Implement Z-score, moving average, K-means clustering, and autoencoder detectors on simulated sensor data streams

Tier 2: Raspberry Pi

Convert autoencoder to TFLite, benchmark inference latency, compare detection accuracy across methods

Tier 3: Arduino/ESP32

Deploy lightweight Z-score detector on Arduino with real sensors, run TFLite autoencoder on ESP32 for complex anomaly patterns

14.1 Introduction: What is Anomaly Detection?

Anomaly detection identifies observations that deviate significantly from the expected pattern. Unlike classification, we typically have abundant “normal” data but few or no examples of anomalies.

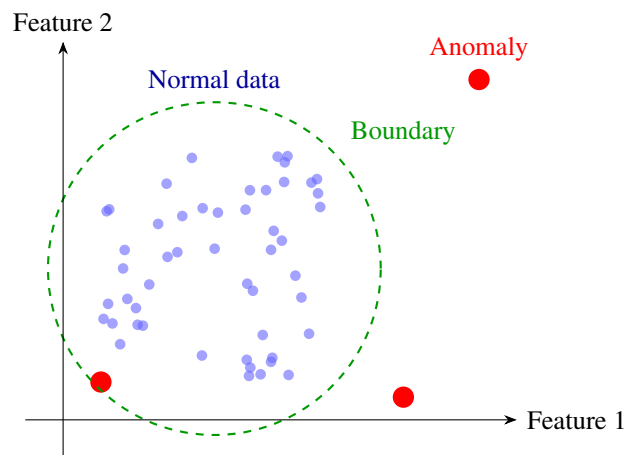


Figure 14.1: Anomaly detection: finding points outside the normal boundary

14.1.1 Edge Use Cases

- **Predictive maintenance:** Detect unusual vibration patterns before equipment failure
- **Security:** Identify abnormal sensor readings indicating intrusion
- **Quality control:** Flag defective products on manufacturing lines
- **Health monitoring:** Detect abnormal heart rhythms or movement patterns
- **Smart buildings:** Identify HVAC faults from temperature/humidity anomalies

Why Anomaly Detection on Edge?

- **Low latency:** React to anomalies immediately, not after cloud round-trip
- **Bandwidth efficiency:** Send alerts, not raw data streams
- **Privacy:** Keep sensitive sensor data local
- **Reliability:** Work during network outages

Building On: LAB13 - Edge-to-Cloud Data Ingestion

Anomaly detection often triggers data retention and alerting decisions. In LAB13, you'll learn how to:

- Store anomalous data samples for later analysis
- Implement selective data transmission (only send anomalies to cloud)
- Build edge-to-cloud pipelines with time-series databases
- Query and visualize anomaly trends over time

This lab detects anomalies locally; LAB13 shows you how to manage and analyze them in distributed systems.

14.2 Method Comparison for Edge Deployment

Different methods suit different resource constraints:

Method	Memory	Latency	Complexity	Best For
Z-Score	~100 B	<1 ms	Very Low	Arduino
Moving Avg	~1 KB	<1 ms	Low	Time series
K-Means	~10 KB	~5 ms	Medium	Multi-dim
Autoencoder	~50 KB	~10 ms	Higher	Complex

Figure 14.2: Anomaly detection methods compared for edge deployment

Choosing an Anomaly Detection Method

Step 1: What is your hardware platform?

- Arduino (32KB Flash, 2KB RAM) → **Z-score** or **Moving Average**
- ESP32 (520KB SRAM) → **K-means** or small **Autoencoder**
- Raspberry Pi (1-8GB RAM) → Any method, including **Isolation Forest**

Step 2: What is your data distribution?

- Single Gaussian (bell curve) → **Z-score** (simplest, effective)
- Multiple clusters (e.g., day/night modes) → **K-means** per cluster
- Complex/non-linear patterns → **Autoencoder**
- Unknown distribution → Plot histogram first, then decide

Step 3: Do you need adaptivity?

- Patterns change over time → **Exponential Moving Average** or retrain periodically
- Patterns are stable → Pre-compute mean/std, use fixed thresholds

Step 4: When to use statistical vs ML approaches?

- Statistical (Z-score, Moving Average): Use when data is well-understood, need explainability, or have limited computational resources
- ML (K-means, Autoencoder): Use when patterns are complex, multivariate data, or need to discover unknown structure

Tip: Start with Z-score. If false positive rate is too high, try K-means or autoencoder.

14.2.1 Comprehensive Method Comparison

Table 14.1: Anomaly Detection Methods: Detailed Comparison

Method	Memory	Computation	Accuracy	Training Data Needed	Best For
Z-Score	~100 B	Very Low (<1 ms)	Good for Gaussian	100+ samples	Single sensor, stable distribution
Moving Average	~1 KB	Low (<1 ms)	Good for trends	50+ samples	Time series, smooth patterns
K-Means	~10 KB	Medium (~5 ms)	Good for clusters	500+ samples	Multi-modal data, known clusters
Autoencoder	~50 KB	High (~10 ms)	Excellent	1000+ samples	Complex patterns, high-dimensional
Isolation Forest	~100 KB	Medium (~8 ms)	Excellent	500+ samples	Mixed distributions, outliers

⚠ Using Z-Score on Non-Gaussian Data

Z-score assumes your data follows a normal (Gaussian) distribution. If your data is multimodal (e.g., sensor has different “normal” modes for day/night), Z-score will produce many false positives. **Always plot a histogram** of your training data before choosing a method!

💡 Threshold Selection

You set threshold = 2.5 standard deviations. Consider:

1. What percentage of normal data will be (incorrectly) flagged as anomalies? (Answer: ~1.2%)
2. If you have 1000 readings/day, how many false alarms per day? (Answer: ~12)
3. Is this acceptable for your application? What’s the cost of a false alarm vs. missed anomaly?

Trade-off: Lower threshold = more sensitive (catches more anomalies) but more false positives.

14.3 Statistical Methods: Lightweight Detection

Statistical methods are ideal for resource-constrained microcontrollers.

14.3.1 Z-Score Detection

The Z-score measures how many standard deviations a value is from the mean:

$$z = \frac{x - \mu}{\sigma} \quad (14.1)$$

Values with $|z| > threshold$ (typically 2-3) are flagged as anomalies.

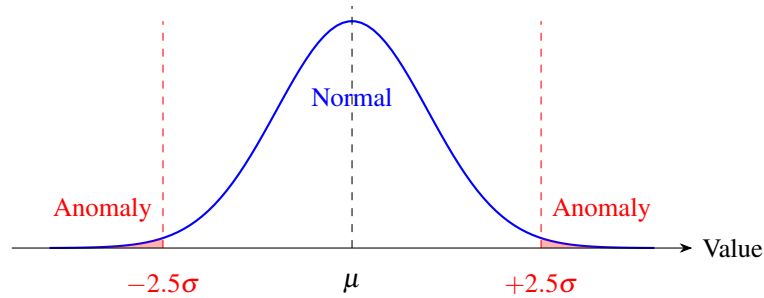


Figure 14.3: Z-score thresholding for anomaly detection

```

1 class OnlineZScoreDetector:
2     """Lightweight Z-score anomaly detector for edge devices."""
3
4     def __init__(self, window_size=100, threshold=3.0):
5         self.window_size = window_size
6         self.threshold = threshold
7         self.buffer = []
8         self.mean = 0.0
9         self.std = 1.0

```

Listing 14.1: Online Z-score detector

```

1     def update_stats(self, value):
2         """Update running mean and std with new normal value."""
3         self.buffer.append(value)
4
5         # Keep buffer size bounded
6         if len(self.buffer) > self.window_size:
7             self.buffer.pop(0)
8
9         # Need minimum samples for reliable statistics
10        if len(self.buffer) >= 10:
11            self.mean = sum(self.buffer) / len(self.buffer)
12            variance = sum((x - self.mean)**2 for x in self.buffer)
13            self.std = (variance / len(self.buffer)) ** 0.5 + 1e-6

```

Listing 14.2: Updating running statistics

Code Walkthrough

window_size=100 Rolling window of recent values—larger = more stable
 threshold=3.0 Values beyond 3σ flagged (catches $\sim 0.3\%$ of normal data)
 + 1e-6 Prevent division by zero if all values are identical

```

1     def detect(self, value):
2         """Check if value is anomalous."""
3         z_score = abs(value - self.mean) / self.std
4         is_anomaly = z_score > self.threshold
5
6         # Only update statistics with normal values!
7         if not is_anomaly:
8             self.update_stats(value)
9
10        return is_anomaly, z_score

```



```

11
12     def memory_usage(self):
13         """Estimate memory in bytes."""
14         return self.window_size * 4 + 24 # floats + overhead

```

Listing 14.3: Detection function

Code Walkthrough

if not is_anomaly Critical: don't let anomalies corrupt the statistics!
 window_size * 4 Each float is 4 bytes

Don't Poison Your Statistics

Never update mean/std with anomalous values. If an anomaly sneaks into the training window, future anomalies may appear “normal” and be missed. Always filter before updating!

14.3.2 Arduino Implementation

```

1  const int WINDOW_SIZE = 50;
2  const float THRESHOLD = 2.5;
3
4  float buffer[WINDOW_SIZE];
5  int bufferIndex = 0;
6  int bufferCount = 0;
7  float runningSum = 0;
8  float runningSumSq = 0;
9
10 bool detectAnomaly(float value) {
11     // Calculate current mean and std
12     float mean = runningSum / max(bufferCount, 1);
13     float variance = (runningSumSq / max(bufferCount, 1)) - (mean * mean);
14     float std = sqrt(max(variance, 0.0001));
15
16     // Calculate Z-score
17     float z = abs(value - mean) / std;
18     bool isAnomaly = (bufferCount >= 10) && (z > THRESHOLD);

```

Listing 14.4: Z-score detector for Arduino

```

1  // Only update with normal values
2  if (!isAnomaly) {
3      // Remove old value from sums
4      if (bufferCount == WINDOW_SIZE) {
5          runningSum -= buffer[bufferIndex];
6          runningSumSq -= buffer[bufferIndex] * buffer[bufferIndex];
7      } else {
8          bufferCount++;
9      }
10
11     // Add new value
12     runningSum += value;
13     runningSumSq += value * value;
14     buffer[bufferIndex] = value;
15     bufferIndex = (bufferIndex + 1) % WINDOW_SIZE;
16 }
17

```

```

18     return isAnomaly;
19 }

```

Listing 14.5: Updating statistics (only for normal values)

Code Walkthrough

runningSum, runningSumSq Maintain sums for $O(1)$ mean/variance calculation
 max(bufferCount, 1) Prevent division by zero at startup
 bufferCount >= 10 Require minimum data before flagging anomalies

Memory Efficiency

This Arduino implementation uses:

- $50 \times 4 = 200$ bytes for the buffer
- 20 bytes for counters and sums
- Total: ~ 220 bytes—fits on any Arduino!

14.4 K-Means Clustering for Anomaly Detection

K-means finds clusters in the training data. Points far from all cluster centers are anomalies.

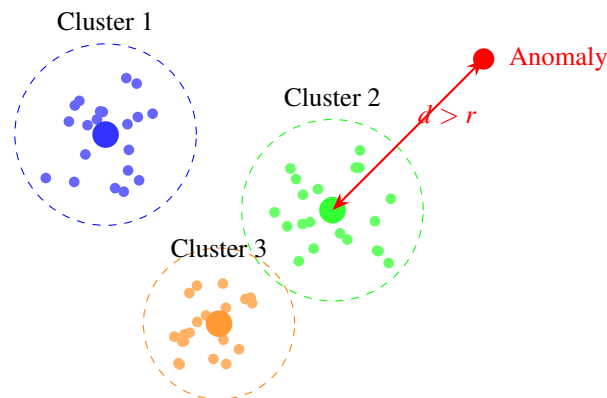


Figure 14.4: K-means anomaly detection: points beyond cluster radius are anomalies

14.4.1 Training Phase

```

1  from sklearn.cluster import KMeans
2  import numpy as np
3
4  # Generate training data (normal operation)
5  np.random.seed(42)
6  n_samples = 300
7  n_clusters = 3
8
9  # Simulate 3 normal operating modes
10 X_train = np.vstack([
11     np.random.randn(100, 2) * 0.4 + [1, 3], # Mode 1
12     np.random.randn(100, 2) * 0.4 + [4, 2], # Mode 2
13     np.random.randn(100, 2) * 0.3 + [2.5, 0.5] # Mode 3
14 ])

```

```

15
16 # Fit K-means
17 kmeans = KMeans(n_clusters=n_clusters, random_state=0)
18 kmeans.fit(X_train)
19
20 print(f"Cluster centers:\n{kmeans.cluster_centers_}")

```

Listing 14.6: Training K-means for anomaly detection

Expected Output

Cluster centers:

```

[[0.98 3.02]
 [3.98 1.99]
 [2.51 0.48]]

```

The three centroids correspond to our simulated operating modes.

```

1 # Calculate distance to nearest centroid for each training point
2 train_distances = kmeans.transform(X_train) # Distance to each center
3 train_labels = kmeans.predict(X_train)
4
5 # Get distance to assigned cluster for each point
6 min_distances = [train_distances[i, train_labels[i]]
7                  for i in range(len(X_train))]
8
9 # Calculate 99th percentile threshold per cluster
10 thresholds = {}
11 for k in range(n_clusters):
12     cluster_distances = [min_distances[i]
13                          for i in range(len(X_train))
14                          if train_labels[i] == k]
15     thresholds[k] = np.percentile(cluster_distances, 99)
16
17 print(f"Thresholds: {thresholds}")

```

Listing 14.7: Calculating anomaly threshold per cluster

Expected Output

Thresholds: {0: 0.82, 1: 0.85, 2: 0.64}

Different clusters have different radius thresholds based on their spread.

Code Walkthrough

`kmeans.transform(X)` Returns distance from each point to each centroid
 99th percentile Threshold that captures 99% of normal data
 per cluster Different clusters may have different spread

14.4.2 Detection Phase

```

1 def detect_kmeans_anomaly(point, kmeans, thresholds):
2     """Detect if point is anomalous."""
3     distances = kmeans.transform([point])[0]
4     nearest_cluster = np.argmin(distances)
5     min_distance = distances[nearest_cluster]
6
7     threshold = thresholds[nearest_cluster]
8     is_anomaly = min_distance > threshold
9
10    return is_anomaly, nearest_cluster, min_distance
11
12 # Test with anomalous point
13 anomaly_point = [6, 4]
14 is_anom, cluster, dist = detect_kmeans_anomaly(
15     anomaly_point, kmeans, thresholds
16 )
17
18 print(f"Point {anomaly_point}: Anomaly={is_anom}, "
19       f"Nearest cluster={cluster}, Distance={dist:.2f}")

```

Listing 14.8: Detecting anomalies with trained K-means

Expected Output

Point [6, 4]: Anomaly=True, Nearest cluster=1, Distance=2.83

The point is 2.83 units from its nearest cluster, well beyond the threshold of 0.85.

Deploying K-Means on Edge

For edge deployment, you only need:

- Cluster centroids ($K \times D$ floats)
- Thresholds (K floats)
- Distance calculation (just Euclidean distance!)

No need for the full sklearn library—implement distance calculation directly.

14.5 Autoencoder-Based Anomaly Detection

Autoencoders learn to compress and reconstruct normal data. Anomalies produce high reconstruction error.

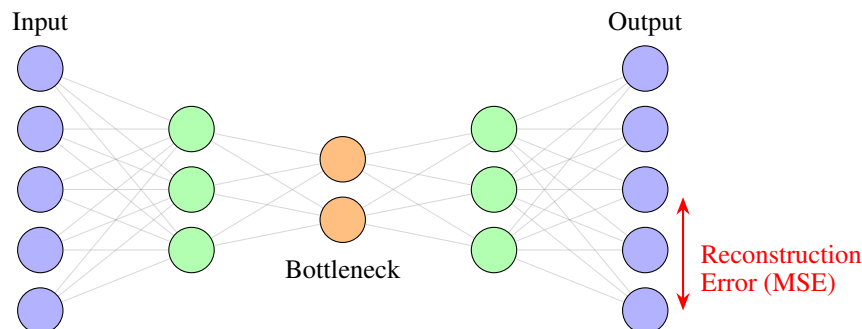


Figure 14.5: Autoencoder architecture for anomaly detection

14.5.1 Building a Tiny Autoencoder

```

1 import tensorflow as tf
2 import numpy as np
3 from sklearn.preprocessing import StandardScaler
4
5 def create_sequences(data, window_size=20):
6     """Create sliding window sequences."""
7     sequences = []
8     for i in range(len(data) - window_size + 1):
9         sequences.append(data[i:i + window_size])
10    return np.array(sequences)
11
12 # Assuming 'normal_data' contains training sensor values
13 window_size = 20
14 sequences = create_sequences(normal_data, window_size)
15
16 # Normalize for stable training
17 scaler = StandardScaler()
18 sequences_norm = scaler.fit_transform(sequences)
19
20 print(f"Training sequences: {sequences_norm.shape}")

```

Listing 14.9: Preparing windowed data

Code Walkthrough

window_size=20 Look at 20 consecutive readings together

StandardScaler Normalize to mean=0, std=1 for stable training

create_sequences Convert time series to fixed-size windows

```

1 def create_tiny_autoencoder(input_dim, encoding_dim=4):
2     """Create minimal autoencoder for edge deployment."""
3
4     # Encoder
5     inputs = tf.keras.layers.Input(shape=(input_dim,))
6     x = tf.keras.layers.Dense(8, activation='relu')(inputs)
7     encoded = tf.keras.layers.Dense(encoding_dim, activation='relu')(x)
8
9     # Decoder
10    x = tf.keras.layers.Dense(8, activation='relu')(encoded)
11    decoded = tf.keras.layers.Dense(input_dim, activation='linear')(x)
12
13    model = tf.keras.Model(inputs, decoded)
14    model.compile(optimizer='adam', loss='mse')
15
16    return model
17
18 # Create and train
19 autoencoder = create_tiny_autoencoder(window_size, encoding_dim=4)
20 autoencoder.summary()

```

Listing 14.10: Creating a tiny autoencoder for edge

Code Walkthrough

encoding_dim=4 Compress 20 values to 4—forces learning of patterns

```
activation='relu' Efficient for edge deployment
loss='mse' Mean squared error between input and output
```

Expected output:

Model: "model"

```
-----
Layer (type)                Output Shape          Param #
=====
input_1 (InputLayer)        [(None, 20)]          0
dense (Dense)                (None, 8)              168
dense_1 (Dense)              (None, 4)              36
dense_2 (Dense)              (None, 8)              40
dense_3 (Dense)              (None, 20)            180
=====
Total params: 424
Trainable params: 424
```

Model Size

424 parameters $\times$ 4 bytes = 1.7 KB—easily fits on ESP32 or Raspberry Pi!

14.5.2 Training on Normal Data Only

```
1 # Train ONLY on normal data
2 history = autoencoder.fit(
3     sequences_norm, sequences_norm, # Input = Target
4     epochs=50,
5     batch_size=32,
6     validation_split=0.1,
7     verbose=1
8 )
9
10 print(f"Final loss: {history.history['loss'][-1]:.6f}")
```

Listing 14.11: Training the autoencoder

Expected Output

```
Epoch 1/50 - loss: 0.8534 - val_loss: 0.7821
Epoch 10/50 - loss: 0.1245 - val_loss: 0.1189
...
Epoch 50/50 - loss: 0.0023 - val_loss: 0.0028
Final loss: 0.002341
```

Loss decreases as the model learns to reconstruct normal patterns.

Train on Normal Only!

The autoencoder must be trained exclusively on normal data. If anomalies are included:

- Model learns to reconstruct anomalies too

- Reconstruction error for anomalies becomes low
- Detection fails!

14.5.3 Setting the Threshold

```

1 # Calculate reconstruction error on training data
2 train_reconstructions = autoencoder.predict(sequences_norm)
3 train_mse = np.mean((sequences_norm - train_reconstructions)**2, axis=1)
4
5 # Set threshold at 95th percentile of training error
6 threshold = np.percentile(train_mse, 95)
7
8 print(f"Training MSE: mean={train_mse.mean():.6f}, "
9       f"std={train_mse.std():.6f}")
10 print(f"Threshold (95th percentile): {threshold:.6f}")

```

Listing 14.12: Determining reconstruction error threshold

```

1 def detect_autoencoder_anomaly(sequence, autoencoder, scaler, threshold):
2     """Detect anomaly using autoencoder reconstruction error."""
3
4     # Normalize
5     seq_norm = scaler.transform([sequence])
6
7     # Reconstruct
8     reconstruction = autoencoder.predict(seq_norm, verbose=0)
9
10    # Calculate MSE
11    mse = np.mean((seq_norm - reconstruction)**2)
12
13    return mse > threshold, mse

```

Listing 14.13: Anomaly detection function

Code Walkthrough

95th percentile Accept 95% of training data as normal; 5% false positive baseline

`mse > threshold` High reconstruction error = model can't explain this input = anomaly

14.6 Evaluation Metrics

	Predicted N	Predicted A
Actual N	TN	FP
Actual A	FN	TP

$$\text{Precision} = \frac{TP}{TP+FP}$$

“Of detected, how many real?”

$$\text{Recall} = \frac{TP}{TP+FN}$$

“Of real, how many detected?”

$$\text{F1} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

Figure 14.6: Confusion matrix and evaluation metrics

```

1 from sklearn.metrics import precision_score, recall_score, f1_score
2
3 # predictions: 0=normal, 1=anomaly
4 # labels: ground truth
5
6 precision = precision_score(labels, predictions)
7 recall = recall_score(labels, predictions)
8 f1 = f1_score(labels, predictions)
9
10 print(f"Precision: {precision:.2%}")
11 print(f"Recall: {recall:.2%}")
12 print(f"F1 Score: {f1:.2%}")

```

Listing 14.14: Evaluating detector performance

Precision vs Recall Trade-off

- **High precision priority:** Minimize false alarms (manufacturing QC)
- **High recall priority:** Don't miss any anomalies (security, safety)
- Adjust threshold to balance based on application requirements

Unsupervised vs Supervised Anomaly Detection

Understanding the fundamental difference between approaches:

- **Unsupervised (most common):** Train only on normal data, detect deviations from learned patterns
- Examples: Z-score, K-means, Autoencoder—no labeled anomalies required
- **Supervised (rare):** Requires labeled examples of both normal and anomaly classes
- Edge advantage: Unsupervised methods need less data and adapt to new anomaly types
- Key challenge: Defining “normal” requires clean training data without anomalies

Threshold Selection Strategies

Choosing the right threshold determines detection sensitivity:

- **Statistical approach:** Use percentile of training distribution (95th, 99th)
- **Fixed sigma:** Z-score > 2.5-3.0 standard deviations (catches 0.3-1.2% of normal data)
- **Cost-based:** Balance cost of false positives vs missed detections
- **Adaptive:** Recalculate threshold periodically to handle concept drift
- Start conservative (high threshold), then tighten based on false positive rate

Edge-Specific Anomaly Detection Challenges

Edge deployment introduces unique constraints and opportunities:

- **Memory limits:** Arduino (2KB RAM) requires Z-score; ESP32 (520KB) can run autoencoders
- **Concept drift:** Sensor characteristics change over time (aging, temperature, calibration)
- **No ground truth:** Can't validate anomalies in real-time—must use threshold tuning
- **Energy constraints:** Always-on anomaly detection drains batteries—use duty cycling
- **Latency requirements:** Must detect and respond in <100ms for safety-critical applica-

tions

14.7 Three-Tier Implementation Guide

Run all methods on synthetic data. Compare F1 scores and memory usage.

Tier 1: Notebook Exploration

Convert autoencoder to TFLite. Benchmark inference time on desktop.

Tier 2: TFLite Conversion

Deploy Z-score on Arduino (real sensor). Autoencoder on ESP32/Pi.

Tier 3: Device Deployment

14.7.1 Tier 1: Compare Methods

Run notebook to compare all methods on the same dataset. Focus on:

- F1 score for each method
- Memory requirements
- Processing latency

14.7.2 Tier 2: TFLite Conversion

```

1 # Convert to TFLite
2 converter = tf.lite.TFLiteConverter.from_keras_model(autoencoder)
3 converter.optimizations = [tf.lite.Optimize.DEFAULT]
4 tflite_model = converter.convert()
5
6 # Save
7 with open('anomaly_detector.tflite', 'wb') as f:
8     f.write(tflite_model)
9
10 print(f"TFLite model size: {len(tflite_model) / 1024:.1f} KB")

```

Listing 14.15: Converting autoencoder to TFLite

14.7.3 Tier 3: Arduino Z-Score Deployment

Use the Arduino implementation from Section 3 with real sensor data.

14.8 Troubleshooting Guide

⚠ Too Many False Positives

Symptoms: Anomaly detector triggers constantly on normal data, alert fatigue

Cause: Threshold too tight, insufficient training data, or concept drift

Solution:

1. Relax threshold: Increase Z-score threshold from 3 to 4-5 sigma

2. Collect more training data: Capture full range of normal variations
3. Use adaptive threshold: Adjust based on recent false positive rate
4. Add hysteresis: Require multiple consecutive anomalies before alerting
5. Feature engineering: Remove noisy features that cause false alarms
6. Implement feedback loop: Let user mark false positives to retrain

⚠️ Anomalies Not Detected / False Negatives

Symptoms: Clear outliers pass through undetected, model too permissive

Cause: Threshold too loose, wrong features, or training data contaminated with anomalies

Solution:

1. Tighten threshold: Reduce to 2-2.5 sigma for more sensitive detection
2. Clean training data: Remove known anomalies from baseline
3. Add domain-specific features: Derivative, rate of change, frequency components
4. Use multiple methods: Combine Z-score with Isolation Forest or Autoencoder
5. Check feature scaling: Normalize all features to same range
6. Validate with synthetic anomalies: Inject known outliers to test detection

⚠️ Autoencoder Reconstruction Error Always High/Low

Symptoms: All samples have similar reconstruction error, no discrimination

Cause: Model underfit/overfit, wrong architecture, or improper training

Solution:

1. Adjust bottleneck size: Too large = no compression, too small = loss of detail
2. Train longer: Ensure model converged (loss plateaued)
3. Use appropriate activation: ReLU for hidden layers, sigmoid/linear for output
4. Normalize inputs: Scale to [0, 1] or standardize to mean=0, std=1
5. Check loss function: Use MSE for continuous data, binary crossentropy for binary
6. Tune threshold empirically: Plot reconstruction error distribution, pick 95th percentile

14.9 Looking Ahead

Looking Ahead

Anomaly detection on edge devices often runs continuously, making energy consumption critical. LAB15 teaches you to optimize these always-on systems through duty cycling, adaptive inference, and power profiling—essential skills for deploying anomaly detectors in battery-powered applications like predictive maintenance sensors and environmental monitoring nodes that must operate for months without recharging.

14.10 Summary

- **Z-Score:** Simplest method, ideal for Arduino. Uses ~200 bytes, <1ms latency. Good for single-sensor monitoring.
- **K-Means:** Unsupervised clustering with distance-based detection. Store only centroids and thresholds for deployment. Good for multi-modal normal behavior.
- **Autoencoder:** Neural network that learns to reconstruct normal patterns. Higher accuracy for complex data, requires ~2KB model. Suitable for ESP32/Pi.

- **Training:** Always train on normal data only. Anomalies in training data will corrupt the model.
- **Threshold Selection:** Use percentile of training error (95-99%). Lower = more false positives; higher = more missed anomalies.
- **Evaluation:** F1 score balances precision (false alarm rate) and recall (miss rate). Choose based on application cost of each error type.

14.11 Guided Experiments

Anomaly Threshold Tuning

Hypothesis: Increasing the anomaly detection threshold reduces false positive rate but increases false negative rate, with optimal threshold dependent on the cost ratio of false alarms to missed detections.

Procedure:

1. Generate 1000 normal samples from $\mathcal{N}(50, 5)$ and 50 anomalies from $\mathcal{N}(80, 10)$
2. Train Z-score detector on normal samples
3. Test thresholds: 1.5, 2.0, 2.5, 3.0, 3.5 standard deviations
4. For each threshold, calculate:
 - True Positives (anomalies detected)
 - False Positives (normal flagged as anomaly)
 - Precision, Recall, F1-score

Data Collection:

Threshold	TP / FP	Precision	Recall	F1-score
1.5				
2.0				
2.5				
3.0				
3.5				

Analysis Questions:

- Plot ROC curve (TPR vs FPR). What is the optimal operating point?
- If false alarms cost \$10 and missed anomalies cost \$1000, which threshold minimizes expected cost?
- For a safety-critical system, would you prefer high precision or high recall? Why?
- How would you automatically tune threshold from validation data?

Detection Method Comparison

Hypothesis: Simple statistical methods (Z-score) will match or outperform complex methods (autoencoder) on unimodal Gaussian data, but underperform on complex multimodal or non-linear patterns.

Procedure:

1. Create two datasets:
 - Dataset A: Simple univariate Gaussian (normal = $\mathcal{N}(50, 5)$, anomaly = $\mathcal{N}(80, 10)$)
 - Dataset B: Complex 2D spiral (normal = spiral points, anomaly = random outliers)
2. Implement three detectors: Z-score, K-means (K=3), Autoencoder (32-16-8-16-32)

3. Train on 1000 normal samples, test on 200 normal + 50 anomalies
4. Measure F1-score, latency, and memory usage for each

Data Collection:

Method	F1 (Dataset A)	F1 (Dataset B)	Latency (ms)	Memory (KB)
Z-score				
K-means				
Autoencoder				

Analysis Questions:

- Which method performed best on simple vs complex data? Why?
- Calculate accuracy/memory and accuracy/latency ratios. Which is most efficient?
- For Arduino (2KB RAM, 10ms latency budget), which method is feasible?
- When would the complexity of autoencoder be justified?

14.12 Exercises**Self-Assessment Questions**

Test your understanding before continuing:

1. Why should you never train anomaly detection models on data containing anomalies? What would happen to the learned normal distribution?
2. What does a Z-score threshold of 3.0 mean statistically? Approximately what percentage of normal data points would be flagged as anomalous?
3. Explain how autoencoder bottleneck size affects anomaly detection. What happens if the bottleneck is too large? Too small?
4. Compare Z-score and K-means for anomaly detection. When would K-means outperform Z-score for multivariate sensor data?
5. For a medical monitoring system vs. a weather station, how would you balance precision and recall? Explain the cost of false positives vs. false negatives in each case.

Adaptive Threshold

Implement a threshold that adapts over time:

- Track recent reconstruction errors
- Adjust threshold based on recent distribution
- Handle gradual drift in sensor behavior
- Test on data with slowly changing baseline

Multi-Sensor Fusion

Combine detections from multiple sensors:

- Run Z-score on temperature, humidity, and vibration
- Flag anomaly if any sensor is anomalous (OR rule)

- Flag only if multiple sensors agree (AND rule)
- Compare false positive rates

Anomaly Classification

Extend detection to classify anomaly types:

- Spike (sudden high value)
- Drop (sudden low value)
- Stuck (constant value)
- Drift (gradual change)

This enables targeted maintenance responses.

- **Further Reading.**
- Isolation Forest: <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.IsolationForest.html>
 - Autoencoders for Anomaly Detection: <https://arxiv.org/abs/1906.02994>
 - One-Class SVM: <https://scikit-learn.org/stable/modules/generated/sklearn.svm.OneClassSVM.html>
 - TinyML Anomaly Detection: <https://www.edgeimpulse.com/blog/anomaly-detection>

— **Credits.** Adapted from TinyMLx course materials and scikit-learn documentation.

Case Study: Wind Turbine Predictive Maintenance System

Context: A wind farm operator managing 85 offshore turbines needed to reduce catastrophic gearbox failures that cost \$250,000 per incident in repairs and lost revenue. Traditional scheduled maintenance was inefficient—inspections occurred every 6 months regardless of actual component health, costing \$12,000 per turbine per visit. The company wanted to implement edge-based anomaly detection using vibration, temperature, and acoustic sensors to predict failures 2-4 weeks in advance, enabling just-in-time maintenance during favorable weather windows.

Challenge: The initial cloud-based system using deep autoencoder models achieved 89% failure prediction accuracy in testing but failed in production. Issues included: (1) 15-30 minute latency from sensor to cloud due to limited satellite connectivity, missing rapid failure onset, (2) \$8,500/month in cloud compute and data transfer costs, (3) false positive rate of 18% triggered unnecessary maintenance visits costing \$102,000/year, (4) model trained on manufacturer test data didn't generalize to real offshore conditions (salt spray, temperature extremes, variable loads). The deep learning model also required GPU inference—impossible to deploy on edge devices in the harsh turbine environment.

Solution: The team deployed edge-based anomaly detection using techniques from this lab:

- Installed ruggedized ESP32 nodes in each turbine nacelle with vibration (3-axis accelerometer), temperature (6 sensors), and current sensors
- Implemented hybrid anomaly detection: Z-score for simple thresholds (temperature >85°C), Isolation Forest for multivariate patterns
- Trained Isolation Forest on 6 months of normal operation data (8.2M samples) from 10 reference turbines, achieving 45KB model size

- Deployed quantized model via TFLite Micro with 12ms inference on 1-minute aggregated windows
- Implemented adaptive thresholding: recalculate baseline statistics every 7 days to account for seasonal and wear-based drift
- Added rule-based override: immediate alerts for critical thresholds (vibration >5g, oil temp >95°C) bypass ML model
- Deployed local edge gateway: aggregates anomaly scores from 85 turbines, sends alerts via satellite only when anomaly score >0.8 for 3+ consecutive readings

Results:

- Failure prediction lead time: 0-3 hours (cloud) → 18-32 days (edge), enabling planned maintenance
- False positive rate: 18% → 3.2% (reduced unnecessary maintenance by \$87,000/year)
- True positive rate: 89% → 94% (caught 2 failures that cloud system missed due to rapid onset)
- Detection latency: 15-30 minutes → <60 seconds (local inference + aggregation)
- Cloud costs: \$8,500/month → \$400/month (96% reduction, only alerts sent to cloud)
- Prevented failures: 7 gearbox failures in first year (\$1.75M in avoided costs vs \$68K edge deployment)

Lessons Learned:

- Hybrid approaches work best: simple rule-based thresholds for critical safety, ML for subtle multivariate patterns
- Isolation Forest is ideal for edge anomaly detection: small model size, no GPU needed, naturally handles multivariate sensor data
- Adaptive baselines are essential for long-lived deployments: turbines age, weather changes, and operating profiles evolve over months/years
- False positives cost real money—optimize for precision even at expense of some recall to avoid unnecessary maintenance
- Train on normal operation from multiple reference units, not just one turbine—captures natural variation and improves generalization
- Temporal aggregation (1-minute windows) reduces false positives from transient spikes while maintaining adequate response time
- Edge gateways for alert aggregation: avoid satellite transmission costs for every anomaly, aggregate scores across fleet first

14.13 Exercises

1. **[Challenge] Challenge: Comparative Anomaly Detection Benchmark:** Implement and compare three anomaly detection methods (Z-score, Isolation Forest, Autoencoder) on the same multi-sensor dataset. Create synthetic normal data with added anomalies: spikes (sudden changes), drifts (gradual shifts), and stuck sensors (constant values). For each method, measure precision, recall, F1 score, memory footprint, and inference latency. Generate ROC curves showing true positive vs. false positive rates at different thresholds. Recommend which method suits Arduino, ESP32, and Raspberry Pi deployments based on resource constraints.
2. **[Challenge] Challenge: Adaptive Threshold Tuning for Concept Drift:** Implement an adaptive anomaly detector that automatically adjusts detection thresholds over time to handle concept drift. Start with Z-score detection using training data statistics, then implement a sliding window that recalculates mean/std every 1000 samples. Test on sensor data with

gradual baseline shifts (e.g., temperature increasing 5°C over 24 hours due to seasonal change). Compare static vs. adaptive thresholds: measure false positive rates during drift and verify the adaptive system maintains consistent sensitivity.

3. **[Challenge] Challenge: Multi-Sensor Fusion Anomaly Detection:** Build an anomaly detector that combines evidence from multiple sensors using voting and Bayesian fusion. Deploy Z-score detectors on 4 correlated sensors (e.g., temperature, humidity, pressure, light). Implement majority voting (flag anomaly if 2+ sensors agree) and Bayesian combination (weight sensor reliability). Inject single-sensor faults and cross-sensor anomalies (e.g., temperature spike without humidity change). Measure how fusion reduces false alarms compared to single-sensor detection while maintaining high recall on true anomalies.
4. **[Challenge] Challenge: Real-Time Autoencoder Deployment with TFLite Micro:** Convert a trained autoencoder to TFLite Micro format and deploy on ESP32 or Arduino Nano 33 BLE. Optimize the model to fit within 256KB Flash and 100KB RAM constraints through pruning, quantization, and architecture simplification. Implement streaming reconstruction error calculation and threshold-based flagging. Measure end-to-end latency from sensor reading to anomaly classification. Compare on-device autoencoder performance against cloud-based processing for latency, privacy, and energy efficiency.



15. Energy-Aware Edge Computing and Optimization

Learning Objectives

After completing this lab, you will be able to:

- Understand power consumption patterns in edge ML systems
- Measure power consumption of edge devices and ML inference
- Calculate energy budgets for battery-powered deployments
- Apply model optimization techniques (quantization, pruning) for energy efficiency
- Implement duty cycling and sleep modes for battery longevity
- Design early-exit networks for adaptive energy-accuracy trade-offs

Prerequisites

Before starting this lab, ensure you:

- Completed LAB05 (Embedded Systems Deployment): edge device programming
- Completed LAB09 (ESP32 Wireless): wireless communication and power management
- Basic understanding of electrical concepts (voltage, current, power)
- Familiarity with ESP32 or similar microcontroller programming
- Understanding of deep sleep modes helpful but not required

Interactive Notebook

Jupyter Notebook: LAB15_energy.ipynb

Open in Google Colab | View on GitHub | Companion Website

Prerequisites

- **Knowledge:** Profiling basics (LAB11), model optimization (LAB03)
- **Level 1:** Laptop with TensorFlow for energy estimation
- **Level 2:** USB power meter or INA219 current sensor
- **Level 3:** Battery-powered MCU setup for real-world testing

Theory and Practice. A model achieving 99% accuracy but draining the battery in 2 hours is useless for wearables that need to last a week. This lab teaches you to think in milliwatts and milliamp-hours, not just accuracy percentages, and to reuse profiling techniques from LAB11 together with models from earlier labs (e.g., LAB03, LAB04, LAB10) to see how design and deployment decisions change the energy budget.

Execution Tiers. Level 1: Estimate energy from model complexity in notebooks. Level 2: Measure power with USB meters or INA219 on Pi/ESP32. Level 3: Implement adaptive inference with power budgets on MCU.

Troubleshooting. Power measurements noisy? Average over multiple runs and control temperature. Battery life shorter than expected? Check sleep mode configuration and radio duty cycles.

Three-Tier Execution Model

Tier 1: Jupyter/Colab

Calculate energy budgets from model MACs and precision, simulate duty cycling scenarios, design early-exit networks for energy-accuracy trade-offs

Measure real power consumption with INA219 sensor or USB power meter, profile inference and transmission energy costs

Implement deep sleep with wake intervals, deploy quantized models with adaptive inference, measure battery life under realistic workloads

15.1 Why Energy Matters for Edge ML

⚠ Optimizing the Wrong Component

The most common energy optimization mistake is obsessing over inference efficiency while ignoring the radio. On a typical IoT device:

- A single WiFi transmission burst uses more energy than *1,000 ML inferences*
- BLE advertising uses 10-50× less power than WiFi
- Keeping WiFi connected (even idle) drains batteries faster than periodic inference

The fix: Profile your *entire* power budget—radio, CPU, sensors, display—before optimizing. Often, reducing transmission frequency by 2× saves more energy than halving inference time.

Edge devices often run on batteries with severe energy constraints:

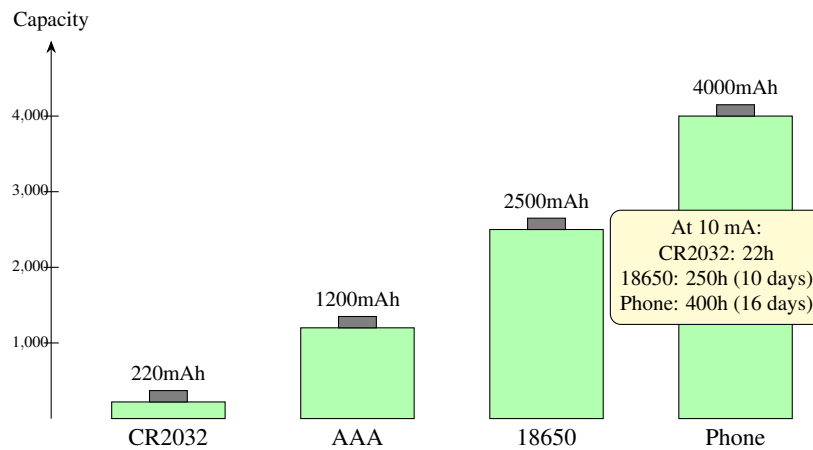


Figure 15.1: Common battery capacities for edge deployments

The Energy Equation

- **Power (mW)** = Voltage (V) × Current (mA)
- **Energy (mWh)** = Power (mW) × Time (hours)
- **Battery Life (hours)** = Capacity (mAh) / Average Current (mA)

15.2 Power Consumption by Component

Understanding where power goes helps optimize it:

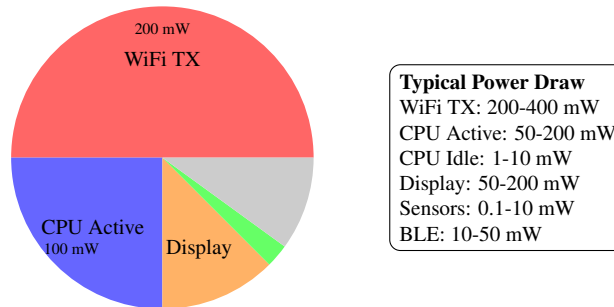


Figure 15.2: Power consumption breakdown for typical edge device

The Radio is Expensive!

WiFi transmission often consumes more power than everything else combined. Strategies:

- Use BLE instead of WiFi when possible (10× less power)
- Batch data and send in bursts rather than continuous streaming
- Process locally to reduce data that needs transmission

15.3 Energy Cost of ML Operations

Not all operations cost the same energy:

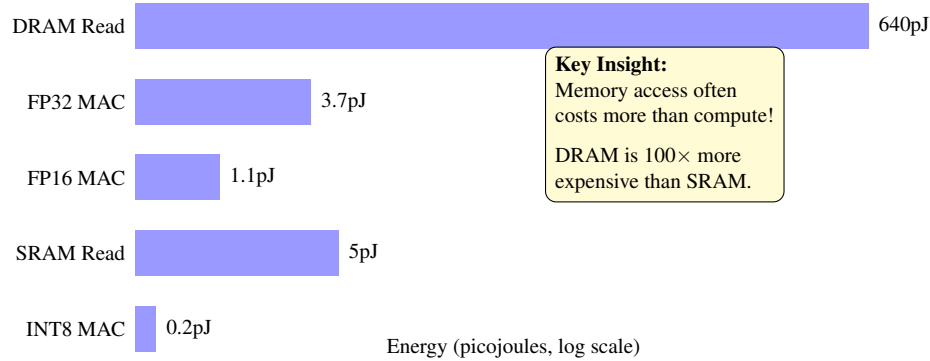


Figure 15.3: Energy cost per operation (approximate)

Why Quantization Saves Energy

INT8 inference is $\sim 10\text{--}18\times$ more energy-efficient than FP32:

- INT8 MAC operations use less energy than FP32
- $4\times$ smaller model = $4\times$ less memory traffic
- Smaller data = fits in faster, cheaper SRAM instead of DRAM

15.4 Energy Budget Calculator

Plan your deployment before building it:

```
1 class EnergyBudgetCalculator:
2     """Calculate if your deployment meets battery life targets."""
3
4     def __init__(self, battery_capacity_mAh, voltage=3.7):
5         self.battery_mAh = battery_capacity_mAh
6         self.battery_mWh = battery_capacity_mAh * voltage
7         self.voltage = voltage
8         self.components = []
```

Listing 15.1: Energy budget calculator class

```
1 def add_component(self, name, power_mW, duty_cycle=1.0):
2     """Add a power-consuming component.
3
4     Args:
5         name: Component identifier
6         power_mW: Power when active
7         duty_cycle: Fraction of time active (0.0 to 1.0)
8     """
9     self.components.append({
10         'name': name,
11         'power_mW': power_mW,
12         'duty_cycle': duty_cycle,
13         'effective_power_mW': power_mW * duty_cycle
14     })
```

Listing 15.2: Adding components with duty cycles

Code Walkthrough

duty_cycle Fraction of time component is active (0.01 = 1%)
 effective_power_mW Average power accounting for duty cycle
 battery_mWh Energy capacity = mAh × Voltage

```

1  def calculate(self, target_days):
2      """Check if deployment meets target battery life."""
3
4      total_power_mW = sum(c['effective_power_mW']
5                           for c in self.components)
6      total_current_mA = total_power_mW / self.voltage
7
8      life_hours = self.battery_mAh / total_current_mA
9      life_days = life_hours / 24
10
11     print(f"Total Power: {total_power_mW:.2f} mW")
12     print(f"Battery Life: {life_days:.1f} days")
13     print(f"Target: {target_days} days")
14
15     if life_days >= target_days:
16         margin = (life_days / target_days - 1) * 100
17         print(f"[OK] Feasible with {margin:.0f}% margin")
18     else:
19         deficit = (target_days / life_days - 1) * 100
20         print(f"[FAIL] Need {deficit:.0f}% power reduction")
21
22     return life_days >= target_days

```

Listing 15.3: Calculating battery life

15.4.1 Example: Smart Agriculture Sensor

```

1  # Target: 30-day battery life with 2500mAh battery
2  calc = EnergyBudgetCalculator(battery_capacity_mAh=2500)
3
4  # MCU mostly sleeping, wakes every 10 minutes
5  calc.add_component("MCU Sleep", power_mW=0.02, duty_cycle=0.98)
6  calc.add_component("MCU Active", power_mW=55, duty_cycle=0.02)
7
8  # ML inference takes 100ms every 10 minutes
9  calc.add_component("ML Inference", power_mW=74, duty_cycle=0.0002)
10
11 # WiFi transmit for 500ms every hour
12 calc.add_component("WiFi TX", power_mW=200, duty_cycle=0.00014)
13
14 # Sensor reads
15 calc.add_component("Sensors", power_mW=5, duty_cycle=0.001)
16
17 # Check feasibility
18 calc.calculate(target_days=30)

```

Listing 15.4: Planning a 30-day deployment

Expected output:

Total Power: 1.14 mW
 Battery Life: 338.6 days
 Target: 30 days
 [OK] Feasible with 1029% margin

Design Margin

Always include at least 50% margin in energy budgets. Real-world factors like temperature, battery aging, and unexpected events will increase power consumption.

15.5 Duty Cycling for Battery Life

The most effective energy optimization: spend most time asleep!

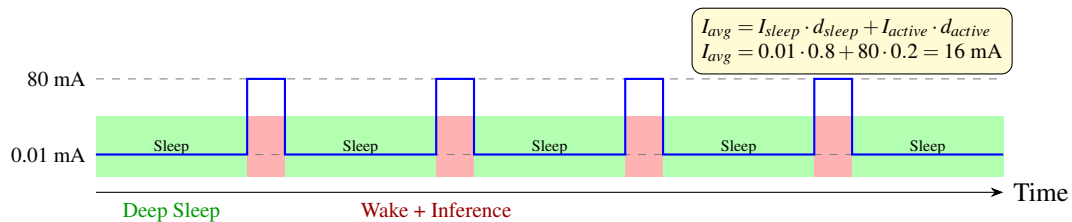


Figure 15.4: Duty cycling: alternate between sleep and active states

15.5.1 ESP32 Deep Sleep Implementation

```

1 #include <esp_sleep.h>
2
3 // Configuration
4 const int WAKE_INTERVAL_SEC = 600;    // Wake every 10 minutes
5 const int ACTIVE_TIME_MS = 2000;      // Stay awake for 2 seconds
6
7 // Power estimates (mA)
8 const float DEEP_SLEEP_CURRENT = 0.01;
9 const float ACTIVE_CURRENT = 80.0;
10
11 void calculateBatteryLife(int battery_mAh) {
12     float duty_cycle = (float)ACTIVE_TIME_MS / (WAKE_INTERVAL_SEC * 1000);
13     float avg_current = (DEEP_SLEEP_CURRENT * (1 - duty_cycle)) +
14                         (ACTIVE_CURRENT * duty_cycle);
15
16     float hours = battery_mAh / avg_current;
17     float days = hours / 24;
18
19     Serial.printf("Duty cycle: %.3f%%\n", duty_cycle * 100);
20     Serial.printf("Avg current: %.3f mA\n", avg_current);
21     Serial.printf("Battery life: %.1f days\n", days);
22 }

```

Listing 15.5: ESP32 duty cycling with deep sleep

```

1 void setup() {
2     Serial.begin(115200);
3
4     // Print battery life estimate
5     calculateBatteryLife(2500); // 18650 battery
6
7     // Do useful work
8     float sensor_value = readSensor();
9     int prediction = runInference(sensor_value);
10
11     // Only transmit if anomaly detected (saves energy!)
12     if (prediction == ANOMALY) {

```

```

13     sendAlert();
14 }
15
16 // Configure wake timer
17 esp_sleep_enable_timer_wakeup(WAKE_INTERVAL_SEC * 1000000ULL);
18
19 // Enter deep sleep (execution ends here)
20 esp_deep_sleep_start();
21 }
22
23 void loop() {
24     // Never reached - device resets after deep sleep
25 }

```

Listing 15.6: Main loop with sleep

Code Walkthrough

esp_sleep_enable_timer_wakeup Set wake-up timer in microseconds
esp_deep_sleep_start Enter deep sleep; CPU stops, RTC runs
void loop() {} Never executed—device resets on wake

Deep Sleep Limitations

Deep sleep resets the device! You lose:

- All RAM contents (except RTC memory)
- WiFi/BLE connections
- Any running timers or state

Use `RTC_DATA_ATTR` to preserve variables across sleep cycles.

15.5.2 Light Sleep vs Deep Sleep

Choose the right sleep mode for your use case:

Table 15.1: ESP32 Sleep Modes Comparison

Mode	Power (μ A)	Wake Time (μ s)	State Preserved?	Use Case
Active	80,000	—	Yes	Inference, radio
Light Sleep	800	200	Yes	Frequent wakes
Deep Sleep	10	400,000	No	Long sleep periods
Hibernation	2.5	600,000	No	Ultra-low power

```

1 #include <esp_sleep.h>
2
3 void setup() {
4     Serial.begin(115200);
5
6     // Configure wake sources
7     esp_sleep_enable_timer_wakeup(10 * 1000); // Wake every 10ms
8
9     // Run frequent inference loop
10    for (int i = 0; i < 100; i++) {

```

```

11     // Quick sensor read
12     float value = analogRead(A0);
13
14     // Check if inference needed
15     if (value > threshold) {
16         runInference(); // Takes ~50ms
17     }
18
19     // Light sleep preserves RAM, fast wake
20     esp_light_sleep_start(); // Only 200us overhead
21 }
22
23 // After burst, enter deep sleep
24 esp_sleep_enable_timer_wakeup(60 * 1000000ULL); // 1 minute
25 esp_deep_sleep_start();
26 }
27
28 void loop() {
29     // Not reached
30 }

```

Listing 15.7: Light sleep example

Choosing Sleep Modes

- **Light Sleep:** Wake intervals < 1 second, need to preserve state
- **Deep Sleep:** Wake intervals > 10 seconds, can afford reset overhead
- **Hybrid:** Use light sleep for short periods, then deep sleep for long periods

15.5.3 Preserving State Across Deep Sleep

```

1 #include <esp_sleep.h>
2
3 // RTC memory survives deep sleep (8KB available)
4 RTC_DATA_ATTR int boot_count = 0;
5 RTC_DATA_ATTR float sensor_history[100];
6 RTC_DATA_ATTR int history_index = 0;
7
8 void setup() {
9     Serial.begin(115200);
10
11     boot_count++;
12     Serial.printf("Boot number: %d\n", boot_count);
13
14     // Read sensor
15     float sensor_value = readSensor();
16
17     // Store in RTC memory
18     sensor_history[history_index] = sensor_value;
19     history_index = (history_index + 1) % 100;
20
21     // Run inference using history
22     if (boot_count > 10) { // Need some history
23         float prediction = runInference(sensor_history, history_index);
24         Serial.printf("Prediction: %.2f\n", prediction);
25     }
26
27     // Sleep for 10 minutes

```



```

28     esp_sleep_enable_timer_wakeup(600 * 1000000ULL);
29     esp_deep_sleep_start();
30 }
31
32 void loop() {
33     // Not reached
34 }

```

Listing 15.8: Using RTC memory for persistent state

Code Walkthrough

RTC_DATA_ATTR Marks variable to be stored in RTC memory

boot_count Increments across deep sleep cycles

sensor_history Ring buffer survives resets

💡 Designing for a 1-Year Battery Life

Your client wants a soil moisture sensor that runs for 1 year on a CR2032 coin cell (220 mAh at 3V = 660 mWh total).

Calculate the average power budget:

- 1 year = 8,760 hours
- 660 mWh / 8,760 h = 0.075 mW average power
- At 3V, that's only 25 μ A average current!

Consider these design questions:

1. How often can you afford to wake up and sample? (Hint: if active at 50 mA, duty cycle < 0.05%)
2. Can you do any ML on-device, or must you transmit raw data?
3. What communication protocol can achieve sub-100 μ A average?
4. How will temperature extremes affect your battery capacity?

The answer often requires aggressive duty cycling (<0.1%), BLE instead of WiFi, and possibly no cloud connectivity at all.

15.6 Early Exit Networks

Stop inference early for “easy” samples to save energy:

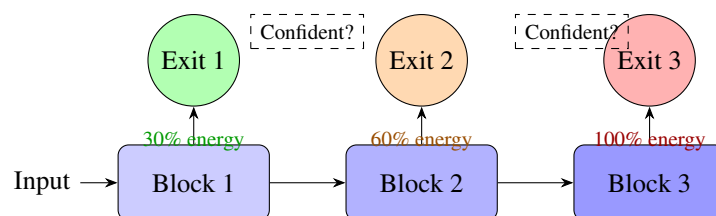


Figure 15.5: Early exit network: stop when confident enough

```

1 import tensorflow as tf
2
3 def create_early_exit_model(num_classes):
4     """Create CNN with multiple exit points."""
5

```

```

6 inputs = tf.keras.layers.Input(shape=(28, 28, 1))
7
8 # Block 1 (lightweight)
9 x = tf.keras.layers.Conv2D(8, 3, activation='relu')(inputs)
10 x = tf.keras.layers.MaxPooling2D()(x)
11
12 # Exit 1: Quick decision for easy samples
13 exit1 = tf.keras.layers.GlobalAveragePooling2D()(x)
14 exit1 = tf.keras.layers.Dense(num_classes,
15                               activation='softmax',
16                               name='exit1')(exit1)

```

Listing 15.9: Early exit network in Keras

```

1 # Block 2 (more features)
2 x = tf.keras.layers.Conv2D(16, 3, activation='relu')(x)
3 x = tf.keras.layers.MaxPooling2D()(x)
4
5 # Exit 2: Medium accuracy
6 exit2 = tf.keras.layers.GlobalAveragePooling2D()(x)
7 exit2 = tf.keras.layers.Dense(num_classes,
8                               activation='softmax',
9                               name='exit2')(exit2)
10
11 # Block 3 (full model)
12 x = tf.keras.layers.Conv2D(32, 3, activation='relu')(x)
13 x = tf.keras.layers.Flatten()(x)
14 x = tf.keras.layers.Dense(64, activation='relu')(x)
15 exit3 = tf.keras.layers.Dense(num_classes,
16                               activation='softmax',
17                               name='exit3')(x)
18
19 return tf.keras.Model(inputs=inputs,
20                       outputs=[exit1, exit2, exit3])

```

Listing 15.10: Adding more exits

```

1 def predict_with_early_exit(model, x, threshold=0.9):
2     """Stop at first exit with confidence above threshold."""
3
4     # Get all three outputs
5     predictions = model.predict(x)
6
7     for i, pred in enumerate(predictions):
8         confidence = pred.max()
9
10        if confidence >= threshold:
11            return pred, i + 1 # Return prediction and exit used
12
13    return predictions[-1], 3 # Use final exit

```

Listing 15.11: Inference with early exit

Code Walkthrough

`GlobalAveragePooling2D` Cheap way to get fixed-size features

`threshold=0.9` Exit if 90%+ confident

`exit used` Track which exit was used for energy accounting

Early Exit Energy Savings

If 60% of samples exit at Exit 1 (30% energy) and 30% at Exit 2 (60% energy):

$$\begin{aligned}\text{Average Energy} &= 0.6 \times 0.3 + 0.3 \times 0.6 + 0.1 \times 1.0 \\ &= 0.18 + 0.18 + 0.10 = 0.46 \text{ (46\% of full model)}\end{aligned}$$

54% energy savings with minimal accuracy loss!

15.6.1 Training Early Exit Networks

Early exit models require special training with multiple loss functions:

```

1 # Compile with multiple outputs
2 model.compile(
3     optimizer='adam',
4     loss={
5         'exit1': 'sparse_categorical_crossentropy',
6         'exit2': 'sparse_categorical_crossentropy',
7         'exit3': 'sparse_categorical_crossentropy'
8     },
9     loss_weights={
10         'exit1': 0.3, # Earlier exits get lower weight
11         'exit2': 0.4,
12         'exit3': 1.0 # Final exit gets full weight
13     },
14     metrics=['accuracy']
15 )
16
17 # Train with all three exits
18 history = model.fit(
19     train_images,
20     {'exit1': train_labels, 'exit2': train_labels, 'exit3': train_labels},
21     epochs=10,
22     validation_data=(test_images,
23                     {'exit1': test_labels, 'exit2': test_labels,
24                      'exit3': test_labels})
25 )

```

Listing 15.12: Training with weighted exit losses

```

1 def adaptive_inference(model, x, confidence_threshold=0.9,
2                         energy_budget_mJ=1.0):
3     """
4     Dynamically choose exit based on confidence and energy budget.
5     """
6     # Get all predictions
7     exit1_pred, exit2_pred, exit3_pred = model.predict(x)
8
9     # Exit 1: Fastest (30% energy)
10    exit1_conf = exit1_pred.max()
11    if exit1_conf >= confidence_threshold:
12        return exit1_pred.argmax(), 1, 0.3 * energy_budget_mJ
13
14    # Exit 2: Medium (60% energy)
15    exit2_conf = exit2_pred.max()
16    if exit2_conf >= confidence_threshold:
17        return exit2_pred.argmax(), 2, 0.6 * energy_budget_mJ
18
19    # Exit 3: Full model (100% energy)

```

```

20     return exit3_pred.argmax(), 3, 1.0 * energy_budget_mJ
21
22 # Example usage
23 predictions = []
24 total_energy = 0
25
26 for image in test_images:
27     pred, exit_used, energy = adaptive_inference(
28         model, image.reshape(1, 28, 28, 1)
29     )
30     predictions.append(pred)
31     total_energy += energy
32
33 print(f"Average energy per inference: {total_energy / len(test_images):.2f}
      mJ")

```

Listing 15.13: Adaptive inference with confidence threshold

Code Walkthrough

loss_weights Balance between early and late exits during training
confidence_threshold Higher = use deeper exits more often
energy_budget_mJ Track actual energy cost per sample

15.7 Adaptive Inference Strategies

Beyond early exit, several strategies adapt inference to available energy:

15.7.1 Battery-Aware Model Selection

```

1 class BatteryAwareInference:
2     def __init__(self, tiny_model, small_model, large_model):
3         self.models = {
4             'tiny': tiny_model,      # 0.5 mJ/inference, 95% accuracy
5             'small': small_model,    # 1.2 mJ/inference, 97% accuracy
6             'large': large_model     # 3.5 mJ/inference, 99% accuracy
7         }
8         self.energy_costs = {'tiny': 0.5, 'small': 1.2, 'large': 3.5}
9
10    def predict(self, x, battery_percent):
11        """Choose model based on remaining battery."""
12
13        if battery_percent > 60:
14            model_name = 'large'      # High accuracy when battery OK
15        elif battery_percent > 30:
16            model_name = 'small'      # Balance accuracy and energy
17        else:
18            model_name = 'tiny'       # Conserve battery at all costs
19
20        model = self.models[model_name]
21        prediction = model.predict(x)
22        energy_used = self.energy_costs[model_name]
23
24        return prediction, energy_used, model_name

```

Listing 15.14: Choose model size based on battery level

15.7.2 Event-Driven Inference

Only run inference when something changes:

```

1 #include <esp_sleep.h>
2
3 const int PIR_PIN = 4;           // Motion sensor
4 const int ACCEL_INT_PIN = 5;     // Accelerometer interrupt
5
6 void setup() {
7     pinMode(PIR_PIN, INPUT);
8
9     // Configure wake on motion
10    esp_sleep_enable_ext0_wakeup(PIR_PIN, HIGH);
11
12    // Only run inference if motion detected
13    if (digitalRead(PIR_PIN) == HIGH) {
14        // Motion! Run expensive inference
15        runFullInference();
16        sendAlert();
17    }
18
19    // Back to sleep (no periodic wake-ups)
20    esp_deep_sleep_start();
21 }

```

Listing 15.15: Motion-triggered inference

15.7.3 Cascaded Inference

Use cheap classifiers to filter out easy cases:

```

1 def cascaded_inference(image, cheap_model, expensive_model):
2     """
3     Stage 1: Fast binary classifier (is this interesting?)
4     Stage 2: Accurate multi-class classifier (what is it?)
5     """
6
7     # Stage 1: Cheap anomaly detector (0.2 mJ)
8     is_anomaly = cheap_model.predict(image)[0] > 0.5
9
10    if not is_anomaly:
11        return 'normal', 0.2 # 90% of samples exit here
12
13    # Stage 2: Expensive classifier (3.5 mJ)
14    class_id = expensive_model.predict(image).argmax()
15
16    return class_id, 0.2 + 3.5 # Total energy
17
18 # Example: 90% normal, 10% anomaly
19 # Average energy = 0.9 * 0.2 + 0.1 * 3.7 = 0.55 mJ
20 # Compared to always running expensive model: 3.5 mJ
21 # Savings: 84%!

```

Listing 15.16: Two-stage cascaded classifier

Cascaded Inference Savings

When most inputs are “uninteresting”, a cheap pre-filter can save 80-90% of energy by avoiding expensive inference on common cases.

15.8 Measuring Power Consumption

15.8.1 Using INA219 Current Sensor

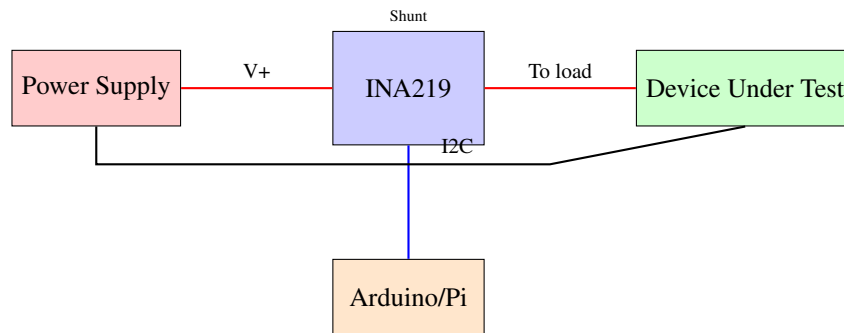


Figure 15.6: INA219 power measurement setup

```

1 #include <Wire.h>
2 #include <Adafruit_INA219.h>
3
4 Adafruit_INA219 ina219;
5
6 float total_energy_mWh = 0;
7 unsigned long last_sample = 0;
8
9 void setup() {
10     Serial.begin(115200);
11     ina219.begin();
12     last_sample = millis();
13 }
14
15 void loop() {
16     // Measure power
17     float voltage = ina219.getBusVoltage_V();
18     float current_mA = ina219.getCurrent_mA();
19     float power_mW = ina219.getPower_mW();
20
21     // Accumulate energy
22     unsigned long now = millis();
23     float dt_hours = (now - last_sample) / 3600000.0;
24     total_energy_mWh += power_mW * dt_hours;
25     last_sample = now;
26
27     // Print
28     Serial.printf("%.2fV, %.1fmA, %.1fmW, Total: %.3fmWh\n",
29                 voltage, current_mA, power_mW, total_energy_mWh);
30
31     delay(100);
32 }
  
```

Listing 15.17: Power measurement with INA219

15.8.2 Alternative: USB Power Meter

For quick testing without soldering, use a USB power meter:

```

1 import serial
2 import time
3
  
```

```

4 # Connect to USB power meter (adjust port as needed)
5 ser = serial.Serial('/dev/ttyUSB0', 9600)
6
7 measurements = []
8
9 for _ in range(100): # Collect 100 samples
10     line = ser.readline().decode('utf-8').strip()
11     # Parse format: "V:5.12 A:0.234 W:1.20"
12     parts = line.split()
13     voltage = float(parts[0].split(':')[1])
14     current = float(parts[1].split(':')[1])
15     power = float(parts[2].split(':')[1])
16
17     measurements.append({
18         'voltage': voltage,
19         'current': current,
20         'power': power,
21         'timestamp': time.time()
22     })
23
24     time.sleep(0.1)
25
26 # Calculate statistics
27 avg_power = sum(m['power'] for m in measurements) / len(measurements)
28 peak_power = max(m['power'] for m in measurements)
29
30 print(f"Average Power: {avg_power:.2f} W")
31 print(f"Peak Power: {peak_power:.2f} W")

```

Listing 15.18: Reading USB power meter data

Power Meter Recommendations

Popular USB power meters for edge ML testing:

- YZXstudio ZY1276 (high accuracy, data logging)
- UM34C/UM25C (Bluetooth, real-time graphs)
- AVHzY CT-3 (affordable, good for quick checks)

Code Walkthrough

getBusVoltage_V() Voltage at the load
 getCurrent_mA() Current through shunt resistor
 getPower_mW() Calculated power ($V \times I$)
 dt_hours Time delta in hours for energy integration

15.9 Model Architecture Energy Comparison

Different model architectures have vastly different energy profiles:

Table 15.2: Energy vs Accuracy Trade-off for Image Classification (MNIST)

Model	Params	Energy/Inf (mJ)	Accuracy (%)	Efficiency (Acc/mJ)
Tiny CNN (8-16-32)	8.2K	0.45	96.8	215
Small CNN (16-32-64)	32K	1.2	98.1	82
MobileNet-like	125K	3.5	98.9	28
Full ResNet-18	11M	45	99.2	2.2

The Efficiency Metric

For battery-powered devices, optimize for **Accuracy per millijoule**, not just accuracy. A 96.8% model that uses 100× less energy may be better than a 99.2% model.

15.9.1 Profiling Different Model Sizes

```

1 import tensorflow as tf
2 import numpy as np
3 import time
4
5 def profile_model_energy(model, test_images, num_runs=100):
6     """Estimate energy consumption for a model."""
7
8     # Warm-up
9     _ = model.predict(test_images[:1])
10
11     # Measure inference time as proxy for energy
12     start = time.perf_counter()
13     for _ in range(num_runs):
14         _ = model.predict(test_images[:1], verbose=0)
15     end = time.perf_counter()
16
17     avg_time_ms = ((end - start) / num_runs) * 1000
18
19     # Estimate energy (assuming 50 mW average power for CPU)
20     # This is a rough estimate - real measurements vary by device
21     energy_per_inference_mJ = avg_time_ms * 0.050
22
23     return {
24         'time_ms': avg_time_ms,
25         'energy_mJ': energy_per_inference_mJ,
26         'params': model.count_params()
27     }

```

Listing 15.19: Energy profiling script for model comparison

```

1 def create_tiny_cnn():
2     """Ultra-lightweight model for maximum battery life."""
3     model = tf.keras.Sequential([
4         tf.keras.layers.Conv2D(8, 3, activation='relu',
5                                 input_shape=(28, 28, 1)),
6         tf.keras.layers.MaxPooling2D(),
7         tf.keras.layers.Conv2D(16, 3, activation='relu'),
8         tf.keras.layers.MaxPooling2D(),
9         tf.keras.layers.Flatten(),
10        tf.keras.layers.Dense(10, activation='softmax')
11    ])

```



```

12     return model
13
14 def create_small_cnn():
15     """Balanced model for moderate accuracy and energy."""
16     model = tf.keras.Sequential([
17         tf.keras.layers.Conv2D(16, 3, activation='relu',
18                                input_shape=(28, 28, 1)),
19         tf.keras.layers.MaxPooling2D(),
20         tf.keras.layers.Conv2D(32, 3, activation='relu'),
21         tf.keras.layers.MaxPooling2D(),
22         tf.keras.layers.Conv2D(64, 3, activation='relu'),
23         tf.keras.layers.Flatten(),
24         tf.keras.layers.Dense(128, activation='relu'),
25         tf.keras.layers.Dense(10, activation='softmax')
26     ])
27     return model
28
29 # Train and compare
30 models = {
31     'Tiny': create_tiny_cnn(),
32     'Small': create_small_cnn()
33 }
34
35 for name, model in models.items():
36     model.compile(optimizer='adam',
37                  loss='sparse_categorical_crossentropy',
38                  metrics=['accuracy'])
39     model.fit(train_images, train_labels, epochs=5, verbose=0)
40
41     # Evaluate
42     test_acc = model.evaluate(test_images, test_labels, verbose=0)[1]
43     profile = profile_model_energy(model, test_images)
44
45     efficiency = test_acc / profile['energy_mJ']
46
47     print(f"\n{name} Model:")
48     print(f"    Accuracy: {test_acc*100:.1f}%")
49     print(f"    Params: {profile['params']:,}")
50     print(f"    Energy: {profile['energy_mJ']:.2f} mJ/inference")
51     print(f"    Efficiency: {efficiency:.1f} (accuracy/mJ)")

```

Listing 15.20: Compare multiple model architectures

Code Walkthrough

`perf_counter()` High-resolution timer for accurate timing
`energy_mJ` Estimated energy assuming 50 mW CPU power
`efficiency` Higher is better: accuracy divided by energy cost

💡 When to Use Which Model?

Consider these deployment scenarios:

1. **Always-on wearable** (CR2032 battery, 1-year target): Use Tiny CNN despite lower accuracy
2. **Solar-powered sensor** (intermittent power): Use Small CNN, run only when charged
3. **Plugged-in security camera** (unlimited power): Use largest model for best accuracy

4. **Hybrid approach:** Use Tiny CNN most of the time, wake a larger model only for uncertain cases

15.10 Energy Optimization Strategies Comparison

 **Choosing Energy Optimization Techniques**

Decision Flow:

- Is your workload continuous or sporadic?**
 - Continuous (always processing) → Use DVFS + Model Pruning
 - Sporadic (periodic wake-ups) → Use Duty Cycling + Deep Sleep
- Can you sacrifice some accuracy for energy?**
 - Yes, flexible accuracy → Early Exit Networks + Cascaded Inference
 - No, accuracy critical → Duty Cycling + Quantization only
- What is your target battery life?**
 - Days to weeks → Moderate duty cycling (1-10% active)
 - Months to years → Aggressive duty cycling (<0.1% active) + event-driven wake
- Is real-time response required?**
 - Yes (<100ms latency) → Light sleep + DVFS, avoid deep sleep
 - No (seconds acceptable) → Deep sleep with wake intervals

Recommended Combinations:

- Always-on wearable:** DVFS + Early Exit + INT8 quantization
- Security camera:** Event-driven (motion) + Cascaded inference + Light sleep
- Environmental sensor:** Deep sleep (99% duty cycle) + minimal inference

15.10.1 Energy Optimization Techniques: Comprehensive Comparison

Table 15.3: Energy Optimization Techniques Comparison					
Technique	Power Savings	Accuracy Impact	Implementation Best For Complexity		
DVFS	40-60%	None	Low (hardware support)	Continuous workloads, CPU-bound tasks	
Duty Cycling	90-99%	None	Low (sleep APIs)	Sporadic sensing, periodic inference	
Early Exit	40-70%	-0.5 to -2%	Medium (multi-head training)	Variable complexity inputs	
Model Pruning	30-50%	-1 to -5%	High (retrain required)	Static deployment, one-time optimization	
INT8 Quantization	60-75%	-0.5 to -2%	Medium (TFLite conversion)	All edge ML deployments	
Cascaded Inference	80-90%	-0 to -3%	Medium (two-stage design)	Skewed class distributions	

15.10.2 Battery Chemistry Comparison

Table 15.4: Battery Technologies for Edge Devices

Chemistry	Voltage	Energy Density	Cycle Life	Best For
Alkaline	1.5V	150 Wh/kg	1 cycle	Disposable, low-drain devices
LiPo	3.7V	150-250 Wh/kg	300-500	Rechargeable, high-drain, compact
LiFePO4	3.2V	90-120 Wh/kg	2000+	Long life, safety-critical, stable
Li-ion 18650	3.7V	200-260 Wh/kg	500-1000	High capacity, cost-effective
Coin Cell (CR2032)	3V	200 Wh/kg	1 cycle	Ultra-compact, low-power

15.11 Dynamic Voltage and Frequency Scaling (DVFS)

Many edge devices support adjusting CPU frequency to trade performance for energy:

```

1 #include <Arduino.h>
2
3 void setup() {
4     Serial.begin(115200);
5
6     // ESP32 supports: 240, 160, 80, 40, 20, 10 MHz
7     // Lower frequency = less power but slower inference
8
9     // Measure at different frequencies
10    for (int freq : {240, 160, 80}) {
11        setCpuFrequencyMhz(freq);
12        Serial.printf("\n--- CPU @ %d MHz ---\n", freq);
13
14        // Benchmark inference
15        unsigned long start = micros();
16        float result = runInference(); // Your ML inference
17        unsigned long duration = micros() - start;
18
19        Serial.printf("Inference time: %lu us\n", duration);
20        Serial.printf("Result: %.2f\n", result);
21
22        delay(1000);
23    }
24 }
25
26 void loop() {
27     // Production code: choose frequency based on battery level
28     int battery_percent = readBatteryLevel();
29
30     if (battery_percent > 50) {
31         setCpuFrequencyMhz(160); // Fast, moderate power
32     } else if (battery_percent > 20) {
33         setCpuFrequencyMhz(80); // Slower, save battery
34     } else {
35         setCpuFrequencyMhz(40); // Emergency mode
36     }
37
38     // Do inference at selected frequency
39     runInference();

```

```

40     esp_light_sleep_start();
41 }

```

Listing 15.21: ESP32 CPU frequency scaling

DVFS Energy Savings

CPU power consumption scales roughly as $P \propto f \cdot V^2$, where f is frequency and V is voltage. Halving the frequency can reduce power by 60-75%, at the cost of $2\times$ longer inference time.

Table 15.5: ESP32 DVFS Power Consumption (Typical)

Frequency (MHz)	Power (mW)	Inference Time (ms)	Energy/Inf (mJ)
240	95	12	1.14
160	68	18	1.22
80	38	36	1.37
40	22	72	1.58

DVFS Trade-offs

For sporadic inference with sleep modes, running at higher frequency and returning to sleep faster often saves more energy than running slower at lower frequency. DVFS works best for continuous workloads.

15.12 Three-Tier Implementation Guide

Use EnergyBudgetCalculator in notebook. Estimate from model MACs and precision.

Tier 1: Energy Estimation

Measure real power with INA219 or USB meter. Compare to estimates.

Tier 2: Power Measurement

Implement duty cycling + early exit. Verify battery life targets.

Tier 3: Optimized Deployment

15.12.1 Tier 1: Notebook Estimation

Use the EnergyBudgetCalculator to plan your deployment before building hardware.

15.12.2 Tier 2: Real Measurements

Connect INA219 inline with your device's power supply and measure:

- Idle/sleep current
- Active current during inference
- Peak current during WiFi transmission

15.12.3 Tier 3: Optimized System

Combine multiple optimization techniques:

- INT8 quantized model
- Early exit for easy samples
- Aggressive duty cycling (1% duty cycle)
- Edge processing to minimize radio usage

15.13 Troubleshooting Guide

⚠ Power Consumption Higher Than Expected

Symptoms: Device drains battery faster than calculated, measured current much higher than datasheet

Cause: Peripherals not disabled in sleep, WiFi left on, LED indicators drawing power

Solution:

1. Disable all peripherals before sleep: `WiFi.disconnect(); BLE.end();`
2. Remove/disable power LEDs: Desolder or cut trace to LED
3. Use lowest power sleep mode: Deep sleep, not light sleep
4. Measure actual current: Use multimeter in series to verify
5. Check for pull-up resistors: Floating pins draw extra current
6. Verify regulator efficiency: Low-dropout (LDO) regulators waste less

⚠ Device Won't Wake from Deep Sleep

Symptoms: ESP32/Arduino enters deep sleep but never wakes up, unresponsive

Cause: Wrong wake source, GPIO not configured, or corrupted RTC memory

Solution:

1. Configure wake source properly: `esp_sleep_enable_timer_wakeup(10e6);` for 10s timer
2. For GPIO wake: Use RTC pins only (GPIO 0, 2, 4, 12-15, 25-27, 32-39 on ESP32)
3. Check pull configuration: `pinMode(WAKE_PIN, INPUT_PULLUP);`
4. Add LED blink before sleep: Verify code reaches sleep command
5. Use external reset: Manually reset to recover, then debug wake logic
6. For Arduino: Some boards don't support deep sleep - use `delay()` instead

⚠ Inaccurate Battery Life Estimates

Symptoms: Calculated battery life doesn't match real-world performance

Cause: Duty cycle assumptions wrong, temperature effects, or battery capacity rating misleading

Solution:

1. Measure actual duty cycle: Log timestamps to verify sleep/wake pattern
2. Account for startup transients: Power spikes during boot
3. Use realistic battery capacity: Usable capacity often 70-80% of rated
4. Temperature derating: Cold temperatures reduce capacity by 20-50%

5. Add safety margin: Design for 1.5-2x calculated consumption
6. Field test prototype: Deploy for 24-48 hours to validate estimates

15.14 Looking Ahead

Looking Ahead

Energy optimization enables long-running edge deployments, but what if multiple devices need to learn collaboratively without sharing raw data? LAB17 introduces federated learning with Flower, where energy-efficient edge devices train models locally and share only model updates—combining the privacy benefits of local processing with the power of distributed learning. This approach is essential for privacy-sensitive applications across healthcare, finance, and IoT.

15.15 Summary

- **Power Basics:** Power = Voltage $\times$ Current. Energy = Power $\times$ Time. Battery life = Capacity / Average current.
- **Where Power Goes:** Radio transmission often dominates. Memory access can cost more than computation.
- **Quantization:** INT8 uses 10-18 $\times$ less energy than FP32 due to simpler operations and smaller memory footprint.
- **Duty Cycling:** Spending 99% of time in deep sleep can extend battery life from hours to months. Choose light sleep for frequent wakes, deep sleep for long periods.
- **Sleep Modes:** ESP32 offers light sleep (800 μ A, preserves state), deep sleep (10 μ A, resets device), and hibernation (2.5 μ A). Use RTC memory to preserve critical state across deep sleep.
- **Power Measurement:** Use INA219 sensors for accurate inline measurement or USB power meters for quick testing. Always measure real devices—estimates can be off by 2-3 $\times$.
- **Model Architecture Trade-offs:** Optimize for accuracy-per-millijoule, not just accuracy. A 96% model using 100 $\times$ less energy beats a 99% model for battery-powered devices.
- **Early Exit Networks:** Stop inference early for confident predictions. Can save 50-80% energy with minimal accuracy loss. Train with weighted multi-output losses.
- **Adaptive Strategies:** Battery-aware model selection, event-driven inference, and cascaded classifiers can reduce average energy by 80-90% in real deployments.
- **DVFS:** Dynamic voltage/frequency scaling trades speed for power. Best for continuous workloads; for sporadic inference, running fast and returning to sleep is often more efficient.
- **Energy Budgets:** Calculate before building. Include 50%+ margin for real-world variability. Use the EnergyBudgetCalculator to validate feasibility.

15.16 Guided Experiments

Duty Cycle vs Battery Life

Hypothesis: Battery life increases linearly with sleep duty cycle percentage, with 90% sleep providing approximately 10x longer life than continuous operation, assuming negligible sleep current.

Procedure:

1. Measure ESP32 current: active = 80mA, deep sleep = 0.01mA
2. Calculate battery life for 2500mAh battery with duty cycles:
 - 100% active (no sleep)
 - Wake every 1 second for 100ms inference (90% sleep)
 - Wake every 10 seconds for 100ms (99% sleep)
 - Wake every 60 seconds for 100ms (99.83% sleep)
 - Wake every 600 seconds for 100ms (99.98% sleep)
3. Verify one configuration experimentally with INA219

Data Collection:

Wake Interval (s)	Avg Current (mA)	Battery Life (days)	Improvement
Always active	80.00		1.0x
1			
10			
60			
600			

Analysis Questions:

- Plot battery life vs wake interval on log-log scale. Is the relationship linear?
- At what wake interval do you achieve 30-day battery life?
- If sleep current increases to 1mA (poor design), how does this change results?
- For a motion-activated system, how would you implement adaptive wake intervals?

DVFS Energy Comparison

Hypothesis: Running ML inference at lower CPU frequency reduces power consumption but increases latency, with energy per inference (power $\times$ time) minimized at moderate frequencies due to non-linear power-frequency relationship.

Procedure:

1. Deploy a simple CNN model on Raspberry Pi
2. Set CPU frequencies: 600MHz, 800MHz, 1000MHz, 1200MHz, 1500MHz
3. For each frequency:
 - Measure inference latency (average of 100 runs)
 - Measure power consumption with USB meter
 - Calculate energy per inference = power $\times$ latency

Data Collection:

CPU Freq (MHz)	Latency (ms)	Power (mW)	Energy/Inf (mJ)
600			
800			
1000			
1200			
1500			

Analysis Questions:

- Plot energy per inference vs frequency. What frequency is most energy-efficient?
- Does “race to sleep” (run fast, finish quickly, sleep) save energy?
- How does memory-bound vs compute-bound code affect optimal frequency?
- For real-time system with 100ms deadline, which frequency would you choose?

15.17 Exercises**Self-Assessment Questions**

Test your understanding before continuing:

1. Calculate battery life for a device drawing 50mA during inference (100ms) and 5mA idle, running every 10 seconds, with a 2000 mAh battery. Show your work.
2. Why does INT8 quantization reduce energy consumption by more than 4x compared to float32? Consider both memory bandwidth and computation.
3. Explain duty cycling with a concrete example. If a device sleeps 99% of the time at 10 microamps and wakes 1% at 50mA, what is the average current draw?
4. How do early-exit networks save energy? Why is the energy saving non-linear with the number of layers processed?
5. An ESP32 at 240 MHz draws 80mA; at 80 MHz it draws 40mA. If inference takes 100ms at 240 MHz and 250ms at 80 MHz, which frequency is more energy-efficient? Justify your answer.

Optimize for 1 Year Battery Life

Design a system to last 1 year on a CR2032 coin cell (220 mAh):

- Choose wake interval and active duration
- Decide when to use WiFi vs local-only processing
- Calculate duty cycle needed
- Verify with EnergyBudgetCalculator

Adaptive Duty Cycling

Implement a system that adjusts its wake interval based on recent activity:

- Wake every 1 minute if anomalies detected recently
- Wake every 10 minutes during normal operation
- Wake every hour during quiet periods
- Calculate energy savings compared to fixed interval

Compare Quantization Energy

Measure actual power consumption for the same model in:

- Float32 (no quantization)
- Float16 (weight quantization)
- INT8 (full quantization)

Compare to theoretical estimates.

Build an Early-Exit Network

Create and train an early-exit CNN:

1. Design a 3-exit network for MNIST
2. Train with appropriate loss weights
3. Measure exit distribution on test set
4. Calculate average energy savings
5. Plot accuracy vs energy trade-off for different confidence thresholds

Power Measurement Lab

If you have access to INA219 or USB power meter:

1. Measure idle current of your device
2. Measure current during inference
3. Measure current during WiFi transmission
4. Calculate duty cycle needed for 30-day battery life
5. Verify with actual battery test (run for 1 hour, extrapolate)

Cascaded Classifier Challenge

Implement a two-stage cascaded system:

1. Train a lightweight binary classifier (normal vs anomaly)
2. Train a detailed multi-class classifier
3. Combine them: run stage 2 only if stage 1 detects anomaly
4. Measure energy savings compared to always running stage 2
5. Experiment with different stage 1 thresholds

DVFS Optimization

On ESP32, compare different CPU frequencies:

1. Measure inference time at 240, 160, 80 MHz
2. Estimate power consumption for each
3. Calculate energy per inference (power $\times$ time)
4. Determine optimal frequency for your workload
5. Consider: Does faster finish = less total energy?

- **Further Reading.**
- TinyML Energy Efficiency: <https://www.tinyml.org/>
 - TensorFlow Model Optimization: https://www.tensorflow.org/model_optimization
 - ESP32 Deep Sleep: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/sleep_modes.html
 - Energy-Efficient Neural Networks: <https://arxiv.org/abs/1906.02243>

— **Credits.** Original materials from TinyML course materials, adapted with additional practical exercises.

Case Study: Solar-Powered Wildlife GPS Tracker

Context: A marine biology research team needed to track migration patterns of endangered sea turtles using GPS + accelerometer loggers. The devices had to operate for 18+ months while the turtles migrate 10,000+ km across oceans. A 25mm x 40mm solar panel and 500mAh LiPo battery provided limited energy harvesting (avg 15mAh/day in ocean conditions). The team needed on-device ML to classify behaviors (swimming, diving, resting, feeding) and optimize GPS sampling to maximize battery life while capturing critical migration data.

Challenge: The initial prototype using an off-the-shelf GPS module and continuous accelerometer sampling lasted only 12 days on battery power. Power breakdown: GPS module consumed 120mW during acquisition (2-3 minutes to get fix), accelerometer at 100Hz drew 18mW, and the microcontroller in active mode consumed 45mW. With solar harvesting providing only 50mWh/day average (cloudy ocean conditions, surface time limited), the system drained batteries in under 2 weeks. The team needed 50x energy reduction to achieve 18-month operation.

Solution: The team applied aggressive energy optimization using techniques from this lab:

- Replaced continuous GPS with adaptive sampling: GPS fix every 6 hours during migration (classified via accelerometer), hourly during feeding/resting
- Implemented behavior classification with TFLite Micro: 1D CNN (8KB model) running on 10-second accelerometer windows
- Used event-driven architecture: accelerometer interrupt wakes MCU only when movement detected ($>0.2g$ change)
- Applied dynamic voltage and frequency scaling (DVFS): run MCU at 8MHz during inference, 80MHz only for GPS communication
- Implemented deep sleep everywhere: MCU in $12\mu A$ deep sleep between accelerometer interrupts (avg 90% duty cycle)
- Optimized GPS acquisition: use almanac prediction to reduce time-to-first-fix from 180s to 35s (6x power reduction)
- Added solar MPPT charging controller and voltage monitoring to adapt sampling based on battery state
- Implemented data compression: store only behavior transitions and GPS waypoints, not raw accelerometer (50x storage reduction)

Results:

- Battery life: 12 days $\rightarrow$ 22 months (67x improvement, exceeds 18-month target)
- Average power consumption: 180mW $\rightarrow$ 2.8mW (64x reduction)
- GPS power: 120mW for 3 min/fix $\rightarrow$ 120mW for 35s/fix (5x reduction per fix)
- MCU active time: 100% $\rightarrow$ 8% (deep sleep 92% of time)
- Data storage: 2GB raw accelerometer $\rightarrow$ 40MB behavior summaries + GPS (50x compression)
- Behavior classification accuracy: 91% (swimming, diving, resting, feeding vs ground truth video)
- Solar efficiency: Captures 45mWh/day, consumes avg 28mWh/day (net positive with margin)

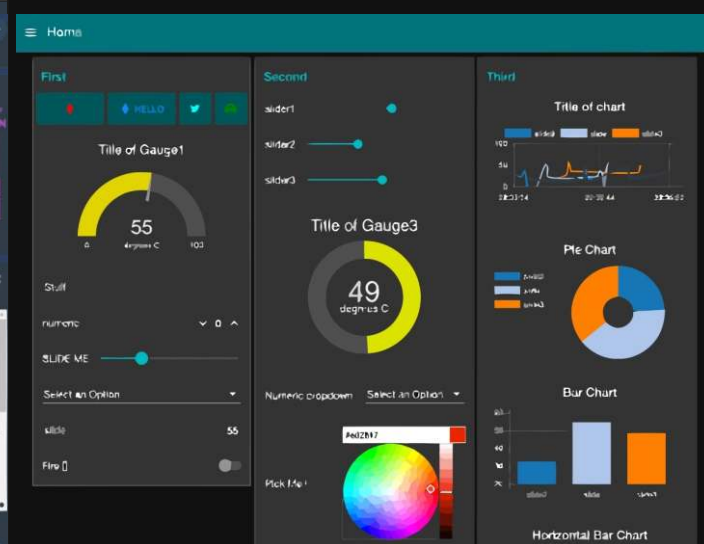
Lessons Learned:

- GPS is the energy killer—reduce acquisition frequency and optimize time-to-first-fix (warm start, almanac prediction)
- Deep sleep is essential: 90%+ duty cycle in $<100\mu A$ mode makes multi-month operation possible

- Event-driven beats polling: accelerometer interrupt-driven wake-up saves 10x power vs continuous sampling
- Adaptive sampling wins: use cheap sensors (accelerometer) to infer context and control expensive sensors (GPS)
- Solar harvesting is unreliable—design for worst-case conditions (cloudy, underwater, winter), not average
- DVFS matters: running MCU at 8MHz vs 80MHz reduces active power by 6x with minimal inference latency impact (<100ms)
- Store insights, not raw data: behavior classifications + GPS waypoints provide same scientific value as raw 100Hz accelerometer at 1/50th storage
- Always measure real power: simulated/datasheet values are optimistic, actual deployment sees 20-40% higher consumption

15.18 Exercises

1. **[Challenge] Challenge: Energy Budget Design for Year-Long Deployment:** Design a battery-powered edge ML system that operates for 12 months on a 2500 mAh LiPo battery. Choose a target application (person detection, keyword spotting, or anomaly detection), select appropriate hardware (ESP32, STM32, etc.), and calculate required duty cycle, inference frequency, and communication intervals. Use the EnergyBudgetCalculator to validate feasibility. Implement a prototype with aggressive power management (deep sleep, event-driven wake-up, adaptive sampling) and measure actual current draw. Compare measured battery life projection against design target and iterate to achieve 365+ day operation.
2. **[Challenge] Challenge: Early-Exit Network Training and Deployment:** Design and train a multi-exit neural network for MNIST or CIFAR-10 with 3 exit points at different depths. Implement confidence-based early exiting where inference stops when prediction confidence exceeds threshold (e.g., 0.95). Train the network with weighted losses to balance accuracy across all exits. Measure the distribution of samples exiting at each layer and calculate average energy savings. Experiment with different confidence thresholds (0.8, 0.9, 0.95, 0.99) and plot the Pareto curve of accuracy vs. energy consumption. Deploy on ESP32 and verify real-world energy reduction using INA219 measurements.
3. **[Challenge] Challenge: Quantization Energy Analysis Across Precisions:** Train identical CNN architectures (MobileNetV1 or similar) with different numeric precisions: Float32, Float16, INT8, and INT4 (if supported). For each variant, measure training accuracy, model size, inference latency, and energy per inference on target hardware (Raspberry Pi or ESP32). Use TensorFlow Lite Micro benchmarking and INA219 current monitoring for accurate energy measurements. Compare theoretical MACs reduction (Float32=1.0, INT8=0.25) against measured energy savings. Document the accuracy-energy trade-off and identify the sweet spot for battery-powered deployment.
4. **[Challenge] Challenge: Dynamic Power Management with Adaptive Strategies:** Implement a smart duty-cycling system that adapts behavior based on battery level, detected events, and time of day. Start with baseline 1-minute wake intervals, but implement adaptive rules: increase sampling to 10-second intervals when anomalies detected, reduce to hourly during nighttime or when battery drops below 20%, and enter emergency mode (one sample per 6 hours) below 5% battery. Deploy on ESP32 with real sensor and battery monitoring. Run for 48+ hours logging all wake events, inference results, and battery levels. Analyze energy savings from adaptation vs. fixed-interval baseline and verify system maintains responsiveness during critical events despite aggressive power management.



16. Real-Time Computer Vision and Object Detect

Learning Objectives

After completing this lab, you will be able to:

- Understand object detection architectures: one-stage vs two-stage detectors
- Implement YOLO object detection using OpenCV DNN module
- Apply face detection and recognition using deep learning embeddings
- Optimize vision models for edge deployment (Tiny-YOLO, MobileNet-SSD)
- Deploy real-time object detection on Raspberry Pi
- Balance accuracy vs speed trade-offs for embedded applications

Prerequisites

Before starting this lab, ensure you:

- Completed LAB07 (CNNs) or understanding of convolutional neural networks
- Basic OpenCV experience (reading images, drawing on frames)
- Understanding of image preprocessing (resizing, normalization)
- Python programming with NumPy array manipulation
- For Level 3: Raspberry Pi setup with camera module experience

Interactive Notebook

Jupyter Notebook: LAB16_cv_yolo.ipynb

[Open in Google Colab](#) | [View on GitHub](#) | [Companion Website](#)

Prerequisites

- **Knowledge:** CNNs (LAB07), OpenCV basics
- **Level 1:** Laptop with OpenCV and webcam
- **Level 2:** Desktop or Raspberry Pi for video pipeline prototyping
- **Level 3:** Raspberry Pi 4 + Pi Camera Module (optionally with accelerator such as Coral or NCS2)

Theory and Practice. One-stage detectors (YOLO, SSD) trade accuracy for speed; two-stage (Faster R-CNN) are more accurate but slower. Input resolution and anchor sizes control the accuracy-latency trade-off. For edge devices, Tiny-YOLO or MobileNet-SSD are preferred, and the models and profiling ideas from LAB03 and LAB11 should be reused here to make informed speed/memory/energy trade-offs.

Execution Tiers. Level 1: Run YOLO/Tiny-YOLO on single images in a notebook to understand architecture, outputs, and thresholds. Level 2: Process webcam or video streams on a laptop or Raspberry Pi, measure FPS and resource usage, and experiment with threaded capture and resolution changes. Level 3: Deploy Tiny-YOLO (or similar) on Raspberry Pi with camera (optionally with accelerator), and profile performance and energy in realistic scenarios.

Troubleshooting. Low FPS? Reduce input resolution and use fewer classes. Camera not found on Pi? Check `/dev/video*` permissions. Memory errors? Use Tiny-YOLO instead of full YOLO, or move to an accelerator.

Three-Tier Execution Model

Tier 1: Jupyter/Colab

Run YOLO and Tiny-YOLO on test images, visualize bounding boxes, experiment with confidence and NMS thresholds

Tier 2: Raspberry Pi

Process webcam video streams with OpenCV DNN module, measure frames per second, implement threaded capture for improved performance

Tier 3: Raspberry Pi 4

Deploy Tiny-YOLO with Pi Camera module, optimize input resolution for real-time detection, optionally add Coral TPU or Neural Compute Stick acceleration

16.1 Object Detection Fundamentals

⚠ Using Full YOLO on Raspberry Pi

The most common mistake when deploying object detection on edge devices is using the wrong model variant. Full YOLOv3 (237 MB, 65 million parameters) achieves <1 FPS on Raspberry Pi 4—essentially unusable for real-time applications.

The symptoms:

- Video feels like a slideshow (0.5-1 FPS)
- System becomes unresponsive during inference
- Memory warnings or crashes on 2GB Pi models

The fix: Start with Tiny-YOLO (34 MB, 2-4 FPS on Pi 4) or MobileNet-SSD (23 MB,

4-6 FPS). Only use full YOLO if you have GPU acceleration (Coral TPU, Jetson Nano).

Object detection simultaneously answers *where* are objects and *what* are they. Unlike classification (one label per image), detection outputs multiple bounding boxes with class labels.

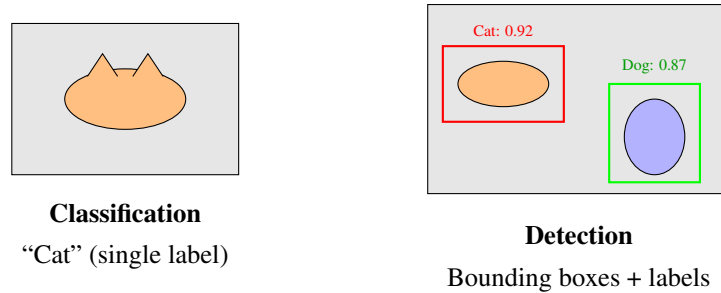


Figure 16.1: Classification vs Object Detection

16.1.1 Two-Stage vs One-Stage Detectors

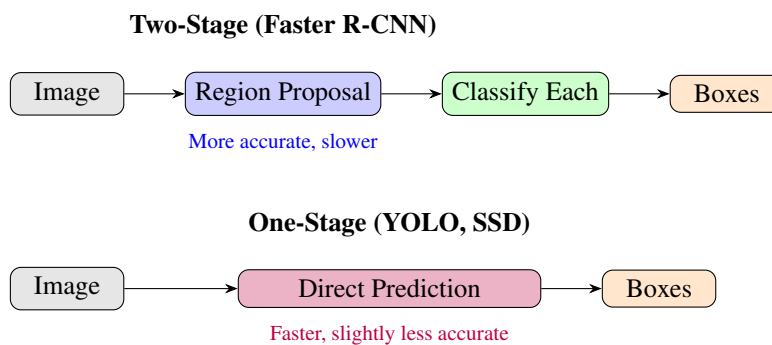


Figure 16.2: Two-stage vs one-stage detector architectures

YOLO Philosophy

You Only Look Once: The image is divided into a grid. Each cell predicts bounding boxes and class probabilities simultaneously in a single forward pass.

- YOLOv3: 51-57% mAP on COCO, 30+ FPS on GPU
- Tiny-YOLO: 23.7% mAP, 200+ FPS on GPU, suitable for edge

16.1.2 How YOLO Predictions Work

Understanding YOLO’s prediction mechanism is essential for interpreting outputs and tuning detection.

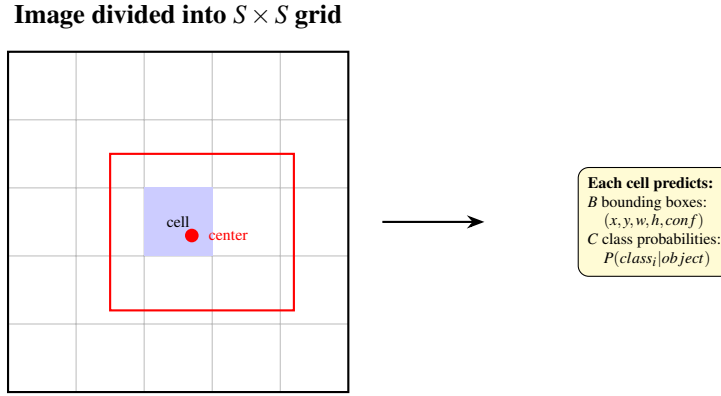


Figure 16.3: YOLO divides the image into a grid. The cell containing an object's center is responsible for detecting it.

For each grid cell, YOLO predicts:

1. B **bounding boxes**, each with 5 values:
 - (x, y) : Center offset relative to grid cell (0-1)
 - (w, h) : Width and height relative to image (0-1)
 - *confidence*: $P(Object) \times IoU_{pred}^{truth}$
2. C **class probabilities**: $P(Class_i | Object)$ for each class

Output tensor shape: $S \times S \times (B \times 5 + C)$

For YOLOv3 with COCO (80 classes), using $S = 13$ and $B = 3$:

$$13 \times 13 \times (3 \times 5 + 80) = 13 \times 13 \times 255$$

Anchor Boxes

YOLO uses *anchor boxes*—predefined box shapes that help detect objects of different aspect ratios. Instead of predicting absolute dimensions, YOLO predicts *offsets* from these anchors:

$$w = anchor_w \times e^{t_w}, \quad h = anchor_h \times e^{t_h}$$

This makes training more stable and helps detect both tall/thin objects (people) and wide/short objects (cars).

16.1.3 Intersection over Union (IoU)

IoU measures how well a predicted box overlaps with the ground truth:

$$IoU = \frac{\text{Area of Intersection}}{\text{Area of Union}} = \frac{|A \cap B|}{|A \cup B|} \quad (16.1)$$

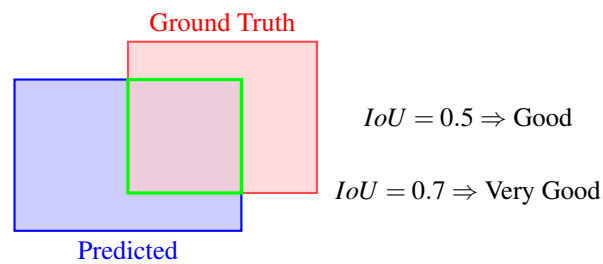


Figure 16.4: IoU calculation: higher values indicate better overlap between predicted and actual boxes.

- $IoU = 1.0$: Perfect overlap
- $IoU > 0.5$: Typically considered a correct detection
- $IoU = 0$: No overlap at all

16.2 YOLO Detection with OpenCV

OpenCV's DNN module can load and run YOLO models directly.

16.2.1 Loading YOLO Model

```

1 import cv2
2 import numpy as np
3
4 # Download from https://pjreddie.com/darknet/yolo/
5 # yolov3.weights (~237 MB)
6 # yolov3.cfg (configuration file)
7
8 net = cv2.dnn.readNetFromDarknet('yolov3.cfg', 'yolov3.weights')
9 net.setPreferableBackend(cv2.dnn.DNN_BACKEND_OPENCV)
10 net.setPreferableTarget(cv2.dnn.DNN_TARGET_CPU) # Or DNN_TARGET_CUDA
11
12 # Get output layer names
13 layer_names = net.getLayerNames()
14 output_layers = [layer_names[i - 1]
15                  for i in net.getUnconnectedOutLayers()]
16
17 print(f"Output layers: {output_layers}")

```

Listing 16.1: Loading YOLO network in OpenCV

Expected Output

```
Output layers: ['yolo_82', 'yolo_94', 'yolo_106']
```

These three output layers correspond to detection at different scales.

Code Walkthrough

`readNetFromDarknet` Loads Darknet-format YOLO model

`setPreferableBackend` Use OpenCV (default) or CUDA backend

`getUnconnectedOutLayers` Gets indices of output layers for multi-scale detection

16.2.2 YOLO's Multi-Scale Detection

YOLO detects objects at three scales for different object sizes:

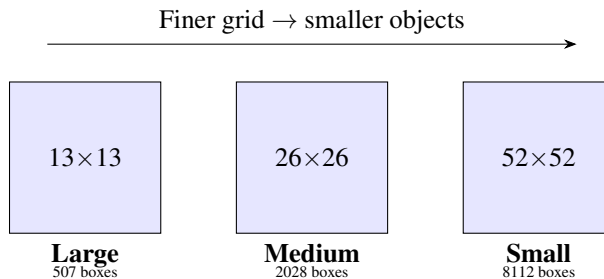


Figure 16.5: YOLO's three detection scales

16.2.3 Running Detection

```

1 # Load image
2 img = cv2.imread('image.jpg')
3 height, width = img.shape[:2]
4
5 # Create blob (preprocessed input)
6 blob = cv2.dnn.blobFromImage(
7     img,
8     scalefactor=1/255.0,      # Normalize to [0,1]
9     size=(416, 416),         # YOLO input size
10    swapRB=True,              # OpenCV uses BGR, YOLO expects RGB
11    crop=False
12 )
13
14 # Run forward pass
15 net.setInput(blob)
16 outputs = net.forward(output_layers)
17
18 print(f"Number of detections: {sum(len(o) for o in outputs)}")

```

Listing 16.2: Creating blob and running forward pass

Expected Output

Number of detections: 10647

YOLO produces many candidate detections. Most will be filtered out by confidence thresholding and NMS.

Code Walkthrough

`blobFromImage` Converts image to 4D tensor (batch, channels, height, width)
`scalefactor=1/255.0` Normalize pixel values to [0, 1]

```
size=(416, 416) Standard YOLO input size (can also use 320, 608)
swapRB=True Convert BGR (OpenCV) to RGB (YOLO)
```

16.2.4 Processing Detections

```
1 # Load COCO class names
2 with open('coco.names', 'r') as f:
3     classes = [line.strip() for line in f.readlines()]
4
5 # Detection parameters
6 conf_threshold = 0.5 # Confidence threshold
7 nms_threshold = 0.4 # Non-maximum suppression
8
9 boxes = []
10 confidences = []
11 class_ids = []
12
13 for output in outputs:
14     for detection in output:
15         scores = detection[5:] # Class probabilities
16         class_id = np.argmax(scores) # Best class
17         confidence = scores[class_id] # Best confidence
18
19         if confidence > conf_threshold:
20             # Scale box coordinates to image size
21             center_x = int(detection[0] * width)
22             center_y = int(detection[1] * height)
23             w = int(detection[2] * width)
24             h = int(detection[3] * height)
25
26             # Convert to corner coordinates
27             x = int(center_x - w / 2)
28             y = int(center_y - h / 2)
29
30             boxes.append([x, y, w, h])
31             confidences.append(float(confidence))
32             class_ids.append(class_id)
```

Listing 16.3: Extracting bounding boxes and classes

Code Walkthrough

```
detection[5:] First 5 values are box coordinates + objectness; rest are class probabilities
detection[0:4] Box coordinates are normalized (0-1), need to scale to image size
center_x, center_y, w, h YOLO uses center-based coordinates
```

16.2.5 Non-Maximum Suppression

Multiple overlapping detections of the same object are common. NMS keeps only the best one.

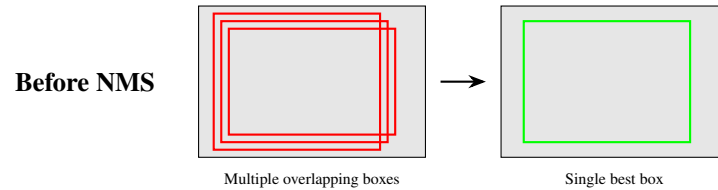


Figure 16.6: Non-Maximum Suppression removes duplicate detections

```

1 # Apply Non-Maximum Suppression
2 indices = cv2.dnn.NMSBoxes(boxes, confidences,
3                             conf_threshold, nms_threshold)
4
5 # Draw results
6 colors = np.random.uniform(0, 255, size=(len(classes), 3))
7
8 for i in indices.flatten():
9     x, y, w, h = boxes[i]
10    label = f"{classes[class_ids[i]]}: {confidences[i]:.2f}"
11    color = colors[class_ids[i]]
12
13    cv2.rectangle(img, (x, y), (x + w, y + h), color, 2)
14    cv2.putText(img, label, (x, y - 10),
15                cv2.FONT_HERSHEY_SIMPLEX, 0.5, color, 2)
16
17 cv2.imshow("YOLO Detection", img)
18 cv2.waitKey(0)

```

Listing 16.4: Applying NMS and drawing results

Expected Output (Console)

```

After NMS: 3 objects detected
- person: 0.95 at (125, 43, 287, 398)
- dog: 0.87 at (402, 156, 198, 231)
- car: 0.82 at (12, 289, 156, 97)

```

The displayed image will show bounding boxes and labels overlaid on the detected objects.

Code Walkthrough

`NMSBoxes` Returns indices of boxes to keep after suppression
`nms_threshold` IoU threshold; boxes with IoU > threshold are suppressed
`indices.flatten()` `NMSBoxes` returns 2D array, flatten for iteration

16.3 Face Detection and Recognition

Face recognition involves two steps: detection (where is the face?) and recognition (whose face is it?).

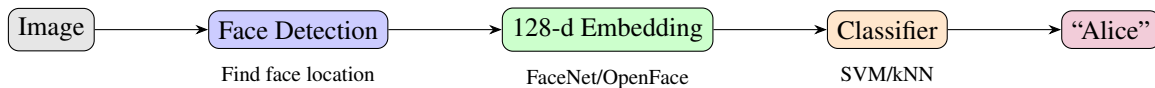


Figure 16.7: Face recognition pipeline

16.3.1 Face Detection with DNN

```

1 # Load Caffe face detector model
2 # Download from OpenCV's GitHub
3 model_file = "res10_300x300_ssd_iter_140000.caffemodel"
4 config_file = "deploy.prototxt"
5
6 face_net = cv2.dnn.readNetFromCaffe(config_file, model_file)

```

Listing 16.5: Loading face detector model

```

1 def detect_faces(image, confidence_threshold=0.5):
2     """Detect faces using deep learning model."""
3     h, w = image.shape[:2]
4
5     # Create blob
6     blob = cv2.dnn.blobFromImage(
7         cv2.resize(image, (300, 300)),
8         1.0,
9         (300, 300),
10        (104.0, 177.0, 123.0) # Mean subtraction values
11    )
12
13    face_net.setInput(blob)
14    detections = face_net.forward()
15
16    faces = []
17    for i in range(detections.shape[2]):
18        confidence = detections[0, 0, i, 2]
19
20        if confidence > confidence_threshold:
21            box = detections[0, 0, i, 3:7] * np.array([w, h, w, h])
22            (x1, y1, x2, y2) = box.astype("int")
23            faces.append((x1, y1, x2, y2, confidence))
24
25    return faces

```

Listing 16.6: Detecting faces in an image

Code Walkthrough

(104.0, 177.0, 123.0) ImageNet mean values for BGR channels
 detections[0, 0, i, 2] Confidence score for detection *i*
 detections[0, 0, i, 3:7] Normalized bounding box coordinates

16.3.2 Face Embeddings

```

1 # Load OpenFace embedding model
2 embedder = cv2.dnn.readNetFromTorch(
3     "openface_nn4_small12.v1.t7"
4 )
5

```

```

6 def get_face_embedding(face_image):
7     """Compute 128-d embedding for a face."""
8     blob = cv2.dnn.blobFromImage(
9         face_image, 1.0 / 255,
10        (96, 96),                                # OpenFace input size
11        (0, 0, 0),
12        swapRB=True,
13        crop=False
14    )
15
16    embedder.setInput(blob)
17    embedding = embedder.forward()
18
19    return embedding.flatten() # 128-d vector

```

Listing 16.7: Computing face embeddings

Face Embedding Comparison

To compare faces, compute the Euclidean distance between embeddings:

- Distance < 0.6: Likely same person
- Distance > 1.0: Likely different people

Use SVM or k-NN classifier trained on known embeddings for recognition.

YOLO Single-Pass Architecture

YOLO's fundamental innovation is treating object detection as a single regression problem:

- One forward pass through the network predicts all bounding boxes and classes
- Grid-based prediction: divide image into $S \times S$ cells (e.g., 13×13)
- Each cell predicts B bounding boxes with confidence scores
- This eliminates the two-stage pipeline of region proposals + classification
- Result: 10-100× faster than R-CNN approaches

Anchor Boxes and Grid Cells

YOLO uses predefined anchor boxes to improve detection of objects with different aspect ratios:

- Anchor boxes are predefined shapes (e.g., tall/thin for person, wide/short for car)
- Each grid cell has multiple anchors to detect different object shapes
- Network predicts *offsets* from anchor dimensions: $w = anchor_w \times e^{t_w}$
- Multi-scale detection: YOLO uses 3 scales (13×13 , 26×26 , 52×52) to detect small and large objects
- The cell containing an object's center is responsible for detecting it

Non-Maximum Suppression (NMS)

NMS is critical for eliminating duplicate detections:

- YOLO generates thousands of candidate boxes (10,000+ for a single image)
- Many boxes overlap for the same object—NMS keeps only the best one
- Algorithm: Sort boxes by confidence, suppress boxes with IoU > threshold (typically

0.4-0.5)

- Without NMS, a single object would have 5-10 bounding boxes
- NMS threshold trade-off: lower = fewer duplicates, higher = preserves overlapping objects

Model Size vs Accuracy for Edge

Edge deployment requires careful model selection based on platform constraints:

- YOLOv3 (237 MB): 57% mAP, 0.5 FPS on Pi 4—unusable for real-time
- Tiny-YOLO (34 MB): 24% mAP, 2-4 FPS on Pi 4—acceptable for many applications
- MobileNet-SSD (23 MB): 21% mAP, 4-6 FPS on Pi 4—best speed/accuracy balance
- Key trade-offs: accuracy vs speed, model size vs memory, power consumption
- Rule: Start with smallest model that meets accuracy requirements

Real-Time Inference Considerations

Achieving real-time performance on edge requires holistic optimization:

- Reduce input resolution: 320×320 vs 608×608 gives $4 \times$ speedup with acceptable accuracy loss
- Threaded capture: Separate threads for camera I/O and inference prevent blocking
- Skip frames: Process every 2nd or 3rd frame while displaying all frames
- Hardware acceleration: Coral TPU or Intel NCS2 provide $5-10 \times$ speedup
- Optimize post-processing: NMS and drawing can be bottleneck—vectorize operations

16.4 Edge Deployment: Tiny-YOLO on Raspberry Pi

Full YOLO is too heavy for Raspberry Pi. Tiny-YOLO achieves usable speeds.

🔗 Choosing an Object Detection Model

Decision Tree:

1. **What is your target platform?**
 - GPU available (Jetson, desktop) → YOLOv3 or EfficientDet
 - Raspberry Pi 4 → Tiny-YOLO or MobileNet-SSD
 - Raspberry Pi 3 or older → MobileNet-SSD only
 - Edge TPU accelerator → EfficientDet-Lite or MobileNet-SSD (EdgeTPU)
2. **What is your accuracy requirement?**
 - Highest accuracy ($>50\%$ mAP) → YOLOv3 (GPU required)
 - Moderate accuracy (20-30% mAP) → Tiny-YOLO or MobileNet-SSD
 - Acceptable accuracy ($<20\%$ mAP) → Ultra-light models with quantization
3. **What is your latency requirement?**
 - Real-time (>15 FPS) → MobileNet-SSD on Pi 4 or use accelerator
 - Near-real-time (5-15 FPS) → Tiny-YOLO on Pi 4
 - Batch processing (<5 FPS) → Any model, optimize for accuracy
4. **How many object classes?**
 - Single class (person only) → Simplified MobileNet-SSD
 - Few classes (<10) → Fine-tune any model, prune unused classes
 - Many classes (80 COCO) → Full pre-trained models

Recommended Combinations:

- **People counting (retail):** MobileNet-SSD person-only + Pi 4 (6 FPS)
- **Security camera:** Tiny-YOLO + motion trigger (2-4 FPS when active)
- **Autonomous robot:** EfficientDet + Jetson Nano (15-30 FPS)

16.4.1 Object Detection Models: Comprehensive Comparison

Table 16.1: Object Detection Models for Edge Deployment

Model	Size (MB)	mAP (COCO)	FPS (Pi 4)	FPS (Jetson Nano)	Best For
YOLOv3	237	57%	0.5	15-20	High accuracy, GPU
YOLOv3-Tiny	34	24%	2-4	30-40	Balanced, Pi 4
MobileNet-SSD	23	21%	4-6	40-60	Speed, Pi 4
EfficientDet-D0	15	34%	1-2	20-25	Balanced, modern
EfficientDet-Lite	5	27%	8-12	50+	EdgeTPU optimized
SSD-MobileNetV2	67	22%	3-5	25-35	Good accuracy/speed

16.4.2 Input Resolution Trade-offs

Table 16.2: Impact of Input Resolution on Performance

Resolution	FPS (Pi 4)	Small Object mAP	Memory (MB)	Recommendation
320×320	8-10	15%	45	Close objects, speed priority
416×416	4-6	24%	65	Balanced (default for Tiny-YOLO)
608×608	1-2	31%	120	Distant objects, accuracy priority
1024×1024	0.3-0.5	38%	280	GPU only, highest accuracy

Model	Size	mAP	FPS (Pi 4)
YOLOv3	237 MB	57%	0.5
Tiny-YOLO	34 MB	24%	2-4
MobileNet-SSD	23 MB	21%	4-6

Figure 16.8: Model comparison for Raspberry Pi deployment

16.4.3 Optimizing for Raspberry Pi

```
1 import cv2
2 from picamera2 import Picamera2
3 import time
4
5 # Load Tiny-YOLO (much smaller and faster)
6 net = cv2.dnn.readNetFromDarknet(
7     'yolov3-tiny.cfg',
```



```

8     'yolov3-tiny.weights'
9 )
10
11 # Use OpenCV optimized for ARM
12 net.setPreferableBackend(cv2.dnn.DNN_BACKEND_OPENCV)
13 net.setPreferableTarget(cv2.dnn.DNN_TARGET_CPU)
14
15 # Reduce input size for faster processing
16 INPUT_SIZE = 320 # Smaller than 416 default
17
18 # Initialize Pi Camera
19 picam2 = Picamera2()
20 config = picam2.create_preview_configuration(
21     main={"size": (640, 480), "format": "RGB888"}
22 )
23 picam2.configure(config)
24 picam2.start()

```

Listing 16.8: Optimized detection for Raspberry Pi

```

1 fps_counter = []
2
3 while True:
4     start_time = time.time()
5
6     # Capture frame
7     frame = picam2.capture_array()
8
9     # Create blob (smaller size for speed)
10    blob = cv2.dnn.blobFromImage(
11        frame, 1/255.0, (INPUT_SIZE, INPUT_SIZE),
12        swapRB=True, crop=False
13    )
14
15    # Run detection
16    net.setInput(blob)
17    outputs = net.forward(output_layers)
18
19    # Process detections (same as before)
20    # ... process boxes, apply NMS, draw results ...
21
22    # Calculate FPS
23    elapsed = time.time() - start_time
24    fps_counter.append(1 / elapsed)
25    if len(fps_counter) > 30:
26        fps_counter.pop(0)
27    fps = sum(fps_counter) / len(fps_counter)
28
29    cv2.putText(frame, f"FPS: {fps:.1f}", (10, 30),
30                cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 255, 0), 2)
31
32    cv2.imshow("Detection", frame)
33    if cv2.waitKey(1) & 0xFF == ord('q'):
34        break
35
36 picam2.stop()
37 cv2.destroyAllWindows()

```

Listing 16.9: Main detection loop

Raspberry Pi Performance Tips

1. Use Tiny-YOLO or MobileNet-SSD, not full YOLO
2. Reduce input size (320×320 instead of 416×416)
3. Use Pi Camera module (faster than USB cameras)
4. Consider Intel Neural Compute Stick for $3\text{--}5\times$ speedup
5. Run headless (no display) for better performance

💡 Accuracy vs Speed: Making the Trade-off

You're building a warehouse robot that needs to detect packages on conveyor belts moving at 1 meter/second. The camera sees packages for about 2 seconds before they exit the frame.

Consider these scenarios:

Scenario A: Full YOLOv3 at 0.5 FPS (57% mAP)

- 1 detection attempt per package
- High accuracy per frame, but only one chance

Scenario B: Tiny-YOLO at 4 FPS (24% mAP)

- 8 detection attempts per package
- Lower per-frame accuracy, but multiple chances

Questions to consider:

1. Which approach has higher *system-level* accuracy? (Hint: multiple low-confidence detections can be combined)
2. What's the latency requirement if the robot must react before the package leaves frame?
3. Could you use temporal filtering (tracking) to boost Tiny-YOLO's effective accuracy?

Often, “worse” models running faster achieve better *real-world* performance than “better” models running slowly.

16.5 Video Stream Processing

For real-time video, optimize the pipeline:

```

1 from threading import Thread
2 import cv2
3
4 class VideoStream:
5     """Threaded video capture for improved FPS."""
6
7     def __init__(self, src=0):
8         self.stream = cv2.VideoCapture(src)
9         self.grabbed, self.frame = self.stream.read()
10        self.stopped = False
11
12    def start(self):
13        Thread(target=self.update, daemon=True).start()
14        return self
15
16    def update(self):
17        while not self.stopped:
18            self.grabbed, self.frame = self.stream.read()
19
20    def read(self):
21        return self.frame
22
```

```

23     def stop(self):
24         self.stopped = True
25         self.stream.release()
26
27     # Usage
28     vs = VideoStream(src=0).start()
29     time.sleep(2.0) # Warm up camera
30
31     while True:
32         frame = vs.read()
33         # Process frame...
34
35         if cv2.waitKey(1) & 0xFF == ord('q'):
36             break
37
38     vs.stop()

```

Listing 16.10: Threaded video capture for better FPS

Code Walkthrough

Thread(daemon=True) Thread dies when main program exits

time.sleep(2.0) Allow camera to warm up before reading

Benefit Main thread processes frames while capture thread reads—reduces latency

16.6 Three-Tier Implementation Guide

Run YOLO on test images in notebook. Visualize detections, tune thresholds.

Tier 1: Single Image Detection

Process webcam stream on laptop. Measure FPS, implement threaded capture.

Tier 2: Video Stream Processing

Deploy Tiny-YOLO on Pi with camera. Optimize for real-time performance.

Tier 3: Raspberry Pi Deployment**16.6.1 Tier 1: Notebook Experiments**

1. Download YOLO weights and config files
2. Run detection on sample images
3. Visualize bounding boxes and class labels
4. Experiment with confidence thresholds

16.6.2 Tier 2: Video Processing

1. Connect webcam or use video file
2. Implement threaded capture

3. Measure and optimize FPS
4. Add face detection alongside object detection

16.6.3 Tier 3: Edge Deployment

1. Install OpenCV on Raspberry Pi
2. Use Tiny-YOLO for speed
3. Configure Pi Camera
4. Optional: Add Neural Compute Stick

16.7 Troubleshooting Guide

⚠️ YOLO Model Loading Failed

Symptoms: `cv2.dnn.readNet()` fails or model file not found error

Cause: Incorrect file path, model format mismatch, or corrupted download

Solution:

1. Verify files exist: Check `yolov3.weights` and `yolov3.cfg` in correct directory
2. Use absolute paths: `net = cv2.dnn.readNetFromDarknet('/full/path/to/yolov3.cfg', '/full/path/to/yolov3.weights')`
3. Re-download weights: File may be corrupted (should be 240MB for YOLOv3)
4. Check OpenCV version: Needs OpenCV 3.4.2+ with DNN module
5. For YOLOv5: Use PyTorch loading instead: `model = torch.hub.load('ultralytics/yolov5', 'yolov5s')`

⚠️ Object Detection Accuracy Low / Missing Objects

Symptoms: YOLO misses obvious objects or detects wrong classes

Cause: Wrong confidence threshold, input image not preprocessed correctly, or model not suitable for domain

Solution:

1. Lower confidence threshold: Try 0.3 instead of 0.5 for more detections
2. Adjust NMS threshold: Increase to 0.5-0.6 to keep more overlapping boxes
3. Check input size: YOLO expects 416x416 or 608x608, resize properly
4. Verify color space: OpenCV uses BGR, ensure correct channel order
5. Use domain-specific model: Retrain on custom dataset if detecting specialized objects
6. Improve lighting: Poor lighting dramatically reduces accuracy

⚠️ YOLO Inference Too Slow for Real-Time

Symptoms: Processing <5 FPS, can't achieve 15-30 FPS target

Cause: No GPU acceleration, model too large, or inefficient post-processing

Solution:

1. Enable GPU: `net.setPreferableBackend(cv2.dnn.DNN_BACKEND_CUDA); net.setPreferableTarget(cv2.dnn.DNN_TARGET_CUDA)`
2. Use smaller model: YOLOv3-tiny instead of YOLOv3 (10x faster)
3. Reduce input resolution: 320x320 instead of 608x608 (4x faster)
4. Skip frames: Process every 2nd or 3rd frame
5. Optimize NMS: Use `cv2.dnn.NMSBoxes` instead of custom implementation
6. For Raspberry Pi: Use Coral TPU accelerator or switch to lighter model (MobileNet-)

SSD)

16.8 Summary

- **Detector Types:** One-stage (YOLO, SSD) are faster; two-stage (Faster R-CNN) are more accurate. Choose based on speed vs accuracy needs.
- **YOLO Pipeline:** Create blob → Forward pass → Filter by confidence → Apply NMS → Draw boxes.
- **Multi-Scale Detection:** YOLO uses 13×13 , 26×26 , and 52×52 grids to detect objects of different sizes.
- **Face Recognition:** Detect faces → Extract 128-d embeddings → Compare with SVM/kNN classifier.
- **Edge Optimization:** Use Tiny-YOLO (34 MB vs 237 MB), reduce input size, use threaded capture, consider hardware accelerators.
- **Performance:** Expect 2-4 FPS with Tiny-YOLO on Pi 4; 4-6 FPS with Neural Compute Stick.

16.9 Guided Experiments

Confidence Threshold Tuning

Hypothesis: Increasing the confidence threshold reduces false positive detections but increases false negatives, with optimal threshold balancing precision and recall based on application requirements.

Procedure:

1. Run YOLO on a test set of 50 images with known ground truth
2. Test confidence thresholds: 0.1, 0.25, 0.5, 0.75, 0.9
3. For each threshold, count:
 - True Positives (correct detections)
 - False Positives (incorrect detections)
 - False Negatives (missed objects)
4. Calculate Precision, Recall, F1-score, mAP

Data Collection:

Threshold	TP	FP	FN	Precision	Recall
0.1					
0.25					
0.5					
0.75					
0.9					

Analysis Questions:

- Plot precision-recall curve. What is the optimal threshold for F1-score?
- For security camera (miss nothing), would you use high or low threshold?
- For production line (flag only defects), would you prefer high precision or recall?
- How does threshold affect inference FPS (if at all)?

NMS IoU Threshold Experiment

Hypothesis: Lower NMS IoU thresholds (more aggressive suppression) reduce duplicate detections but may suppress legitimate overlapping objects, with optimal threshold dependent on object density in typical scenes.

Procedure:

1. Select test images with: (A) isolated objects, (B) overlapping objects, (C) dense crowds
2. Run YOLO with confidence=0.5 and NMS IoU thresholds: 0.2, 0.3, 0.4, 0.5, 0.6
3. For each configuration:
 - Count total detections per image
 - Count duplicate detections (same object multiple boxes)
 - Count missed objects

Data Collection:

NMS IoU	Isolated (avg boxes)	Overlapping	Dense Crowd
0.2			
0.3			
0.4			
0.5			
0.6			

Analysis Questions:

- For isolated objects, does NMS threshold matter? Why or why not?
- At what IoU do you start suppressing legitimate overlapping detections?
- How would you automatically tune NMS threshold based on scene density?
- What is the computational cost of NMS? Does threshold affect it?

Related Labs

Prerequisites:

- **LAB07 - Convolutional Neural Networks:** Understanding of CNN architectures and layers is essential for comprehending how YOLO processes images.
- **LAB03 - Model Optimization and Quantization:** Knowledge of model optimization techniques is critical for deploying efficient object detection on edge devices.

Next Steps:

- **LAB15 - Energy-Aware Edge Computing:** Apply energy profiling to computer vision workloads to optimize battery life for mobile object detection systems.

16.10 Exercises

Self-Assessment Questions

Test your understanding before continuing:

1. Explain the difference between one-stage (YOLO, SSD) and two-stage (Faster R-CNN) object detectors. What are the speed and accuracy trade-offs?
2. How does Non-Maximum Suppression (NMS) work? Why is it necessary when YOLO predicts multiple bounding boxes for the same object?

3. Why does YOLO use three different grid scales (13x13, 26x26, 52x52)? What size objects does each scale detect best?
4. Describe how face embeddings are used for recognition. Why is a 128-d embedding more efficient than comparing full images?
5. Compare Tiny-YOLO (34 MB, 5-10 FPS) vs. YOLOv3 (237 MB, 1-2 FPS) on Raspberry Pi 4. For a battery-powered person detector, which would you choose and why?

Custom Object Detector

Train YOLO to detect custom objects:

- Collect 100+ images of your target object
- Annotate bounding boxes using LabelImg
- Fine-tune Tiny-YOLO on your dataset
- Deploy on Raspberry Pi

See Darknet documentation for training instructions.

Face Attendance System

Build an automated attendance system:

- Collect face images of participants
- Train SVM classifier on embeddings
- Run real-time recognition on Pi with camera
- Log recognized faces with timestamps

Speed Optimization Challenge

Maximize FPS on Raspberry Pi:

- Compare input sizes: 320, 416, 608
- Compare models: YOLOv3-tiny vs MobileNet-SSD
- Test with and without display output
- Measure impact of threaded vs sequential capture

Document which combinations achieve the best performance.

- **Further Reading.**
- YOLO Original Paper: <https://arxiv.org/abs/1506.02640>
 - YOLOv3 Paper: <https://arxiv.org/abs/1804.02767>
 - OpenCV DNN Tutorial: https://docs.opencv.org/master/d2/d58/tutorial_table_of_content_dnn.html
 - FaceNet Paper: <https://arxiv.org/abs/1503.03832>
 - Edge TPU Documentation: <https://coral.ai/docs/>

— **Credits.** YOLO concepts from Redmon et al. OpenCV face recognition pipeline adapted from PyImageSearch tutorials. Raspberry Pi optimization techniques from community best practices.

Case Study: Retail Store People Counter for Occupancy Monitoring

Context: A retail chain with 250+ stores needed automated occupancy counting to comply with COVID-19 capacity restrictions (max 40% occupancy) and gather foot traffic analytics. Traditional solutions required \$3,000-\$8,000 per store for commercial people counting systems with cloud subscriptions. The company wanted a low-cost edge solution using Raspberry Pi + camera that could detect and track people entering/exiting in real-time, display current occupancy on a screen, and log analytics locally for \$200 per store.

Challenge: Initial prototypes using YOLOv3 on Raspberry Pi 4 achieved only 2.1 FPS at 416x416 resolution—far too slow for real-time counting at store entrances where 5-10 people/minute pass through. This caused missed detections and inaccurate counts. Smaller YOLOv3-Tiny improved speed to 8 FPS but suffered 15-25% counting errors due to: (1) false positives from shopping carts, mannequins, and posters, (2) missed detections when people walked in groups or carried large bags, (3) double-counting when people lingered in the detection zone, (4) lighting variations between morning/afternoon/evening. Privacy concerns also required on-device processing—no cloud transmission of video.

Solution: The team implemented an optimized edge detection pipeline:

- Deployed MobileNet-SSD v2 (person class only) quantized to int8 via TFLite (6.2MB → 1.8MB)
- Reduced input resolution to 320x240 and cropped to entrance region-of-interest (50% fewer pixels to process)
- Implemented multi-threaded pipeline: dedicated threads for camera capture, inference, and tracking/counting
- Added centroid tracking algorithm: assign unique IDs to detected persons, count only when crossing virtual entrance line
- Applied temporal filtering: require 3 consecutive detections before counting to eliminate false positives
- Implemented calibration mode: store background during setup, suppress detections on static objects (mannequins, displays)
- Optimized Raspberry Pi: disabled desktop GUI, overclocked to 1.8GHz, enabled hardware H.264 encoding for optional video logs
- Added automatic brightness adjustment for camera to handle changing daylight conditions

Results:

- Inference speed: 2.1 FPS (YOLOv3) → 18 FPS (MobileNet-SSD + optimization)
- Counting accuracy: 76% (YOLOv3-Tiny baseline) → 94.3% vs manual ground truth counts
- False positives: 25/hour → 2/hour (mannequins, carts filtered via background calibration)
- Missed detections: 18% → 4.8% (temporal filtering + higher FPS catches brief appearances)
- System cost: \$200 per store (Raspberry Pi 4 + camera + display) vs \$3,000-\$8,000 commercial solutions
- Privacy compliance: 100% on-device processing, zero cloud transmission, optional video retention of 24 hours locally

Lessons Learned:

- MobileNet-SSD outperforms YOLO-Tiny for single-class edge detection: smaller model, faster inference, adequate accuracy
- Region-of-interest cropping is free performance: if only monitoring entrance, don't

process full frame

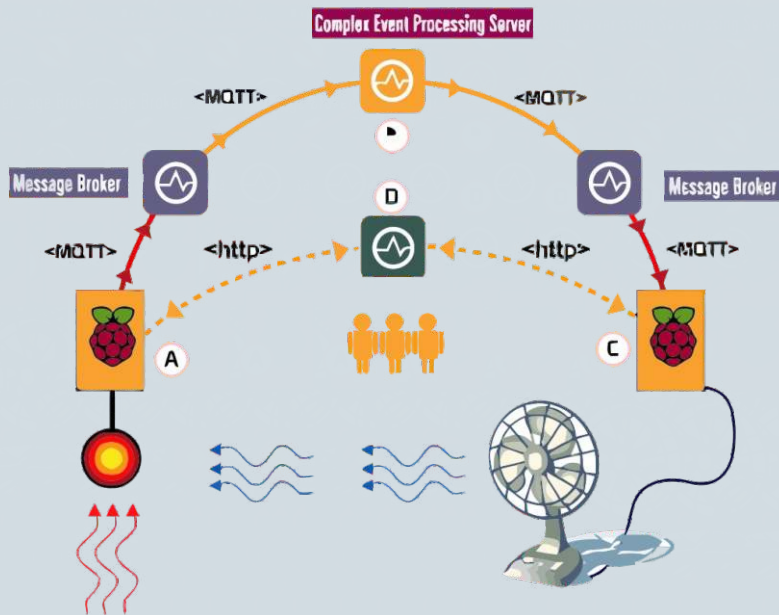
- Tracking algorithm is essential for counting: raw per-frame detection counts are wildly inaccurate due to flickering detections
- Temporal filtering eliminates most false positives: require detections in N consecutive frames before triggering actions
- Background calibration matters: static false positive sources (mannequins, posters) can be learned and suppressed
- Multi-threading unlocks edge performance: camera capture, inference, and display can run concurrently on quad-core Pi
- Lighting adaptation is critical: auto-adjust camera exposure/gain for morning sun vs evening fluorescent lighting
- Ground truth validation is essential: test against manual counts over full day to catch morning/afternoon/evening failure modes

16.11 Exercises

1. **[Challenge] Challenge: Model Comparison and Performance Profiling:** Systematically compare object detection models for edge deployment: YOLOv3, YOLOv3-Tiny, MobileNet-SSD, and MobileNet-SSD-Lite. For each model, measure FPS, mAP (mean Average Precision), memory usage, and model size on both laptop and Raspberry Pi 4. Test on a custom validation set of 100 images covering various object scales and occlusions. Create visualization showing the accuracy-speed Pareto frontier and recommend optimal models for three scenarios: maximum accuracy (plugged-in device), balanced (solar-powered), and maximum speed (battery-powered real-time tracking).
2. **[Challenge] Challenge: Real-Time Video Pipeline Optimization with Threading:** Implement and optimize a multi-threaded video processing pipeline on Raspberry Pi for achieving maximum FPS. Create three separate threads: frame capture (VideoStream class), inference (YOLO detection), and display/logging. Use queues for inter-thread communication with bounded capacity to handle backpressure. Measure FPS with and without threading, experiment with different input resolutions (320x320, 416x416, 608x608), and test impact of reducing detection to every Nth frame while displaying all frames. Document how to achieve 10+ FPS on Raspberry Pi 4 with Tiny-YOLO through pipeline optimization.
3. **[Challenge] Challenge: Custom Object Detection Training and Transfer Learning:** Collect and annotate a custom dataset of 200+ images for a specific detection task (e.g., detecting packages, pets, or vehicles). Use LabelImg or similar tools for bounding box annotation in YOLO format. Fine-tune YOLOv3-Tiny on your custom dataset using Darknet or TensorFlow, starting from pretrained COCO weights. Implement data augmentation (flipping, rotation, brightness adjustment) to improve generalization. Evaluate on a held-out test set and deploy the optimized model on Raspberry Pi. Compare custom model performance against generic COCO-pretrained model on your specific use case.
4. **[Challenge] Challenge: Edge Accelerator Integration and Benchmarking:** Deploy object detection on Raspberry Pi with hardware acceleration using Google Coral USB Accelerator, Intel Neural Compute Stick 2, or Raspberry Pi AI Kit. Convert your YOLO or MobileNet-SSD model to the appropriate format (EdgeTPU TFLite, OpenVINO IR). Implement the accelerated inference pipeline and measure detailed performance metrics: inference latency distribution, frame preprocessing overhead, postprocessing time, and end-to-end FPS. Compare accelerated performance against CPU-only baseline. Calculate energy consumption and cost-effectiveness (FPS per dollar, FPS per watt) to determine when

hardware acceleration is justified for edge computer vision deployments.

5. **[Challenge] Challenge: NMS Threshold Tuning and Precision-Recall Analysis:** Investigate the impact of Non-Maximum Suppression (NMS) IoU threshold on detection performance across different scene types. Create a comprehensive evaluation dataset with three categories: sparse scenes (isolated objects), moderate overlap (2-3 objects partially overlapping), and dense scenes (crowds, shelves with many products). For each scene type, run YOLO detection with fixed confidence threshold (0.5) while varying NMS IoU threshold from 0.1 to 0.9 in steps of 0.1. For each configuration, calculate precision (what fraction of detections are correct), recall (what fraction of ground truth objects are detected), F1-score, and average number of duplicate boxes per object. Plot precision-recall curves for each NMS threshold and identify the optimal operating point. Analyze trade-offs: when does aggressive NMS (low IoU) start suppressing legitimate detections in crowded scenes? When does lenient NMS (high IoU) create excessive duplicates? Implement an adaptive NMS strategy that automatically adjusts threshold based on detection density in local image regions.
6. **[Challenge] Challenge: YOLO Model Size Comparison for Edge Deployment:** Compare YOLO model variants (YOLOv8-nano, YOLOv8-small, YOLOv8-medium, YOLOv5-nano, YOLOv3-tiny) for edge deployment across multiple hardware platforms: laptop/desktop CPU, Raspberry Pi 4, and optionally Jetson Nano or edge TPU. For each model-hardware combination, measure: (1) model size and memory footprint, (2) inference latency (mean, p50, p95, p99), (3) throughput (FPS) with and without preprocessing/postprocessing, (4) detection accuracy (mAP@0.5 and mAP@0.5:0.95) on COCO validation or custom test set, (5) energy consumption per inference using LAB15 techniques. Create comprehensive comparison visualizations: accuracy vs FPS scatter plot, model size vs mAP, latency distribution box plots, and energy efficiency (mAP per joule). Determine deployment recommendations: which model is best for battery-powered applications, solar-powered remote cameras, and plugged-in real-time systems? Document the Pareto frontier of models (non-dominated solutions) and identify use cases where smaller models outperform larger ones due to system-level constraints.
7. **[Challenge] Challenge: Hardware-Specific YOLO Optimization Pipeline:** Select a target edge device (Raspberry Pi 4, Jetson Nano, or Coral Dev Board) and systematically optimize YOLO inference for maximum performance. Start with baseline YOLOv8-small or Tiny-YOLO and apply a progression of optimizations: (1) quantization (FP32 $\rightarrow$ FP16 $\rightarrow$ INT8) using TensorFlow Lite or PyTorch quantization, (2) input resolution reduction ($640 \times 640 \rightarrow 416 \times 416 \rightarrow 320 \times 320$), (3) model pruning to remove low-importance weights, (4) layer fusion and operator optimization for target hardware, (5) batch processing if applicable, (6) mixed precision inference combining FP16 and INT8 layers. For each optimization step, measure the impact on: inference latency, accuracy (mAP degradation), memory usage, and power consumption. Establish the complete performance-accuracy curve and identify the sweet spot that achieves target FPS (e.g., 10 FPS for real-time) with minimal accuracy loss. Document the complete optimization recipe including conversion commands, calibration procedures for quantization, and deployment code. Compare your optimized pipeline against baseline and commercial solutions to demonstrate achievable performance gains on resource-constrained edge hardware.



17. Federated Learning with the Flower Framework

Learning Objectives

After completing this lab, you will be able to:

- Understand federated learning principles: decentralized training with local data
- Implement Federated Averaging (FedAvg) for distributed model aggregation
- Set up Flower server and client architecture
- Design client implementations for TensorFlow/Keras models
- Handle data partitioning and non-IID distributions
- Deploy federated learning on Raspberry Pi edge nodes

Prerequisites

Before starting this lab, ensure you:

- Completed LAB02 (Neural Network Training Fundamentals): solid TensorFlow/Keras experience
- Completed LAB13 (Edge-to-Cloud Data Ingestion): understanding of distributed systems
- Understanding of model training loops (epochs, batches, loss)
- Basic networking concepts (IP addresses, ports, client-server)
- Ability to run multiple terminal sessions simultaneously

Interactive Notebook

Jupyter Notebook: LAB17_flower_federated.ipynb

Open in Google Colab | View on GitHub | Companion Website

Prerequisites

- **Knowledge:** TensorFlow/Keras (LAB02), networking basics
- **Level 1:** Laptop with Python and Flower framework
- **Level 2:** Multiple machines on same network (VMs or physical) or multiple processes on one host
- **Level 3:** 2–3 Raspberry Pi devices for distributed training

Theory and Practice. Federated learning trains models across decentralized devices while keeping data local—critical for privacy-sensitive edge applications such as those in LAB10 (biomedical), LAB12 (streaming), and LAB16 (vision). Flower provides a production-ready framework that handles the complexity of distributed coordination, and this lab connects conceptually to LAB14 (anomaly detection) and LAB18 (on-device adaptation) by showing an alternative way to adapt models using many clients rather than a single device.

Execution Tiers. Level 1: Simulate FL with multiple clients on a single machine (separate processes or notebooks). Level 2: Run server and clients across multiple machines or VMs on a LAN, experimenting with IID vs non-IID partitions and network conditions. Level 3: Deploy clients on Raspberry Pi devices that train on real local sensor data, observing bandwidth, latency, and energy implications.

Troubleshooting. Rounds stalling? Lower client fraction or increase timeout. Slow convergence? Increase local epochs or use learning rate warmup. Memory issues on Pi? Reduce batch size and model size.

Three-Tier Execution Model

Tier 1: Jupyter/Colab

Simulate Flower server and multiple clients on single machine, implement FedAvg with partitioned MNIST data, observe convergence over rounds

Tier 2: Multi-Machine

Run Flower server on laptop and clients on separate machines or VMs, experiment with IID vs non-IID data distributions, measure communication overhead

Tier 3: Raspberry Pi

Deploy Flower clients on Raspberry Pi devices with real sensor data, measure training time and bandwidth usage, implement privacy-preserving aggregation

17.1 Introduction: Why Federated Learning?

⚠ Model Architecture Mismatch Between Clients

The most cryptic FL error occurs when clients have subtly different model architectures. Flower aggregates weights by position—if Client 1 has a 128-neuron layer where Client 2 has 256 neurons, aggregation silently produces garbage.

Common causes:

- Copy-pasting model code and accidentally modifying one client
- Different TensorFlow/Keras versions creating different default initializations
- Conditional model branches that resolve differently on different machines

Symptoms:

- Accuracy drops to random chance after first aggregation
- `ValueError: Shapes don't match` (if you're lucky)
- Server runs but model never converges

Prevention: Define the model in *one shared file* that all clients import. Print `model.summary()` on each client and verify they match exactly.

Traditional ML requires centralizing all data. Federated Learning (FL) trains models *where the data lives*.

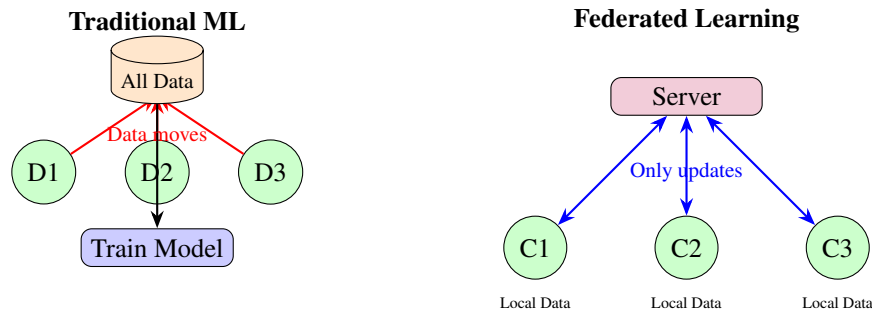


Figure 17.1: Traditional ML vs Federated Learning

Federated Learning Benefits

- **Privacy:** Raw data never leaves the device
- **Bandwidth:** Send model updates (KB-MB), not raw data (GB)
- **Latency:** Local inference with periodically improved models
- **Compliance:** Easier GDPR/HIPAA compliance

17.2 Federated Averaging (FedAvg)

The core FL algorithm: clients train locally, server averages the updates.

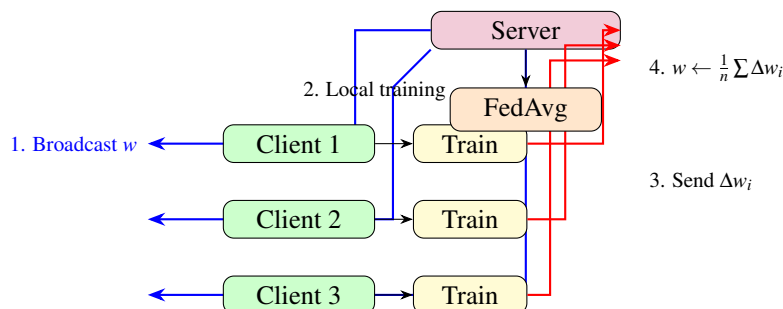


Figure 17.2: FedAvg algorithm: one round of federated training

FedAvg Algorithm

For each round:

1. Server broadcasts current global model weights w to selected clients

2. Each client i trains on local data for E epochs
3. Clients send updated weights w_i back to server
4. Server computes weighted average: $w_{new} = \sum_i \frac{n_i}{n} w_i$
where n_i is the number of samples on client i and $n = \sum n_i$.

17.2.1 Mathematical Formulation

The FedAvg update rule can be expressed as:

$$w_{t+1} = \sum_{k=1}^K \frac{n_k}{n} w_k^{t+1} \quad (17.1)$$

where:

- w_t : Global model at round t
- K : Number of participating clients this round
- n_k : Number of training samples on client k
- $n = \sum_{k=1}^K n_k$: Total samples across all clients
- w_k^{t+1} : Client k 's model after local training

Local training on client k :

$$w_k^{t+1} = w_t - \eta \sum_{e=1}^E \nabla L_k(w; \mathcal{D}_k) \quad (17.2)$$

where E is the number of local epochs and $\mathcal{D}_k$ is client k 's local dataset.

Client Sampling

In practice, not all clients participate every round:

- **Client fraction C :** Percentage of clients selected per round
- Typical values: $C = 0.1$ (10%) for large deployments
- Random selection prevents bias toward faster clients

🔗 Choosing a Federated Learning Algorithm

Decision Tree:

1. **Is your data IID (identically distributed) across clients?**
 - Yes, IID data → FedAvg (simplest, most efficient)
 - No, non-IID data → FedProx or FedOpt
2. **How reliable is client connectivity?**
 - All clients always available → FedAvg with high client fraction
 - Intermittent connectivity → FedProx (tolerates stragglers)
 - Highly unreliable → Asynchronous FL or Scaffold
3. **What is your convergence speed priority?**
 - Fast convergence critical → FedOpt (server-side momentum)
 - Moderate speed acceptable → FedAvg or FedProx
 - Slow convergence OK → Minimize communication with fewer rounds
4. **Do you need privacy guarantees?**
 - Differential privacy required → Add DP noise to FedAvg/FedProx
 - Secure aggregation needed → Use cryptographic protocols
 - Basic privacy sufficient → Standard FL (data never leaves device)

When to use FL vs alternatives:

- **Use Federated Learning:** Privacy/compliance required, data legally/ethically cannot be centralized, or naturally distributed data
- **Use Centralized Training:** No privacy concerns, all data accessible, easier debugging needed
- **Use On-Device Learning (LAB18):** Single device adaptation, personalization, or no connectivity to server

17.2.2 Federated Learning Algorithms: Comprehensive Comparison

Table 17.1: FL Algorithm Comparison

Algorithm	Communication Cost	Non-IID Handling	Handling	Convergence Speed	Best For
FedAvg	Low (weight averaging)	Poor (drift)		Moderate	IID data, reliable clients
FedProx	Low (proximal term)	Good (μ regularization)		Moderate-Fast	Non-IID, stragglers
FedOpt (FedAdam)	Medium (server optimizer)	Good (momentum)		Fast	Need fast convergence
Scaffold	High (control variates)	Excellent		Fast	Severe non-IID
FedBN	Low (local BatchNorm)	Good		Moderate	Domain shift between clients

17.2.3 Convergence Properties

FedAvg converges to the same solution as centralized training under certain conditions:

Table 17.2: Factors affecting FL convergence

Factor	Effect	Mitigation
IID data	Fast convergence	Ideal case
Non-IID data	Slower, may oscillate	FedProx, data sharing
Client dropout	Delays, bias	Increase participation
Few local epochs (E)	More communication	Balance with bandwidth
Many local epochs (E)	Client drift	Use smaller learning rate

Tuning FedAvg

Start with these hyperparameters:

- Local epochs $E = 1 - 5$ (more = less communication)
- Client fraction $C = 0.1 - 1.0$ (depends on total clients)
- Learning rate: Start $10\times$ lower than centralized
- Rounds: 50-500 depending on dataset size

17.3 The Flower Framework

Flower is a unified FL framework that handles distributed coordination.

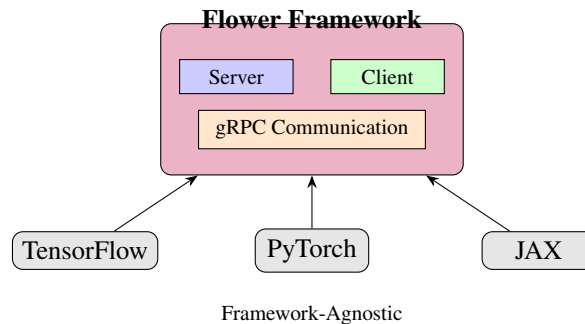


Figure 17.3: Flower integrates with any ML framework

17.3.1 Installation

```
1 pip install flwr tensorflow numpy
```

Listing 17.1: Installing Flower

Version Compatibility

Flower works with TensorFlow 2.x, PyTorch 1.x+, and most ML frameworks. Check the Flower documentation for specific version requirements.

17.4 Implementing the Flower Server

The server coordinates training rounds and aggregates updates.

```

1 import flwr as fl
2
3 # Define the aggregation strategy
4 strategy = fl.server.strategy.FedAvg(
5     fraction_fit=1.0,           # Use all available clients
6     fraction_evaluate=1.0,      # Evaluate on all clients
7     min_fit_clients=3,          # Minimum clients for training
8     min_evaluate_clients=3,     # Minimum clients for evaluation
9     min_available_clients=3,    # Wait for this many clients
10 )

```

Listing 17.2: server.py - Flower server setup

Code Walkthrough

`fraction_fit` Fraction of clients to use per round (1.0 = all)
`min_fit_clients` Minimum clients needed to start a round
`min_available_clients` Wait until this many clients connect

```

1 if __name__ == "__main__":
2     fl.server.start_server(
3         server_address="0.0.0.0:8080", # Listen on all interfaces
4         config=fl.server.ServerConfig(num_rounds=5),

```



```

5     strategy=strategy,
6 )
7 print("Federated training complete!")

```

Listing 17.3: Starting the server

Code Walkthrough

server_address IP:port for client connections
 num_rounds=5 Number of FL rounds to run
 strategy Aggregation strategy (FedAvg)

Network Configuration

Use 0.0.0.0:8080 to listen on all network interfaces. For production, use firewall rules and consider TLS encryption for gRPC.

17.5 Implementing the Flower Client

Clients train on local data and communicate with the server.

17.5.1 Data Loading and Partitioning

```

1 import tensorflow as tf
2 from tensorflow import keras
3 import numpy as np
4
5 def load_data(client_id, num_clients=3):
6     """Load MNIST and partition for this client."""
7
8     # Load full MNIST dataset
9     (x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
10
11     # Normalize pixel values to [0, 1]
12     x_train = x_train.astype("float32") / 255.0
13     x_test = x_test.astype("float32") / 255.0
14
15     # Partition training data among clients
16     partition_size = len(x_train) // num_clients
17     start_idx = (client_id - 1) * partition_size
18     end_idx = start_idx + partition_size
19
20     x_train_client = x_train[start_idx:end_idx]
21     y_train_client = y_train[start_idx:end_idx]
22
23     print(f"Client {client_id}: {len(x_train_client)} training samples")
24
25     return (x_train_client, y_train_client), (x_test, y_test)

```

Listing 17.4: Loading and partitioning MNIST data

Expected Output

Client 1: 20000 training samples
 Client 2: 20000 training samples
 Client 3: 20000 training samples

Each client receives an equal partition of the 60,000 MNIST training samples.

Code Walkthrough

`partition_size` Each client gets $1/N$ of the data
`client_id - 1` Zero-indexed partitioning
`x_test, y_test` All clients share the same test set for fair comparison

17.5.2 Model Definition

```

1 def create_model():
2     """Create a simple neural network for MNIST."""
3     model = keras.Sequential([
4         keras.layers.Flatten(input_shape=(28, 28)),
5         keras.layers.Dense(128, activation="relu"),
6         keras.layers.Dropout(0.2),
7         keras.layers.Dense(10, activation="softmax")
8     ])
9
10    model.compile(
11        optimizer="adam",
12        loss="sparse_categorical_crossentropy",
13        metrics=["accuracy"]
14    )
15
16    return model

```

Listing 17.5: Creating the neural network model

Model Consistency

All clients must use the **exact same model architecture**. The server aggregates weights by position—different architectures will cause errors.

17.5.3 Flower Client Class

```

1 import flwr as fl
2
3 class MNISTClient(fl.client.NumPyClient):
4     """Flower client for MNIST federated learning."""
5
6     def __init__(self, model, x_train, y_train, x_test, y_test):
7         self.model = model
8         self.x_train = x_train
9         self.y_train = y_train
10        self.x_test = x_test
11        self.y_test = y_test
12
13    def get_parameters(self, config):
14        """Return current model weights."""

```

```
15     return self.model.get_weights()
```

Listing 17.6: Implementing the Flower NumPyClient

```
1  def fit(self, parameters, config):
2      """Train model on local data."""
3      # Set weights from server
4      self.model.set_weights(parameters)
5
6      # Train locally
7      self.model.fit(
8          self.x_train, self.y_train,
9          epochs=1,
10         batch_size=32,
11         verbose=1
12     )
13
14     # Return updated weights and sample count
15     return self.model.get_weights(), len(self.x_train), {}
16
17     def evaluate(self, parameters, config):
18         """Evaluate model on local test data."""
19         self.model.set_weights(parameters)
20         loss, accuracy = self.model.evaluate(
21             self.x_test, self.y_test, verbose=0
22         )
23         return loss, len(self.x_test), {"accuracy": accuracy}
```

Listing 17.7: Training and evaluation methods

Code Walkthrough

get_parameters Server calls this to get initial weights
 fit Server calls this for local training; returns weights + sample count
 evaluate Server calls this for distributed evaluation
 len(self.x_train) Sample count used for weighted averaging

17.5.4 Starting the Client

```
1  CLIENT_ID = 1 # Change this for each client
2
3  if __name__ == "__main__":
4      print(f"Starting Flower Client {CLIENT_ID}")
5
6      # Load local data partition
7      (x_train, y_train), (x_test, y_test) = load_data(CLIENT_ID)
8
9      # Create model
10     model = create_model()
11
12     # Create Flower client
13     client = MNISTClient(model, x_train, y_train, x_test, y_test)
14
15     # Connect to server
16     fl.client.start_numpy_client(
17         server_address="127.0.0.1:8080",
18         client=client
```

19)

Listing 17.8: client.py - Complete client script

17.6 Running Federated Training

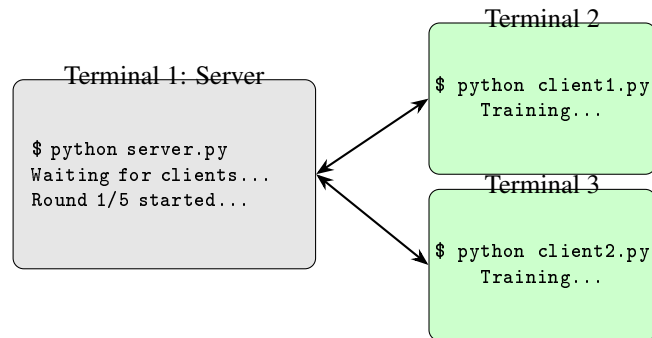


Figure 17.4: Running FL: one server + multiple clients in separate terminals

17.6.1 Step-by-Step Execution

1. Start the server:

```
1 python server.py
2
```

Server waits for `min_available_clients` to connect.

2. Start clients (in separate terminals):

```
1 # Terminal 2
2 python client.py # with CLIENT_ID = 1
3
4 # Terminal 3
5 python client.py # with CLIENT_ID = 2
6
7 # Terminal 4
8 python client.py # with CLIENT_ID = 3
9
```

3. Observe training: Once all clients connect, training begins. Each round:

- Server sends global model to clients
- Clients train locally and return updates
- Server aggregates updates

Expected Server Output

```
FL round 1/5
Clients: 3
Training: loss=0.4521, accuracy=0.8734
Evaluation: loss=0.2189, accuracy=0.9356
-----
FL round 2/5
Training: loss=0.1876, accuracy=0.9421
...
```

```

FL round 5/5
Training: loss=0.0823, accuracy=0.9745
Evaluation: loss=0.0892, accuracy=0.9721
Federated training complete!

```

Accuracy improves with each round as the global model aggregates knowledge from all clients.

17.7 Handling Non-IID Data

Real-world FL often has non-IID (non-identically distributed) data across clients.

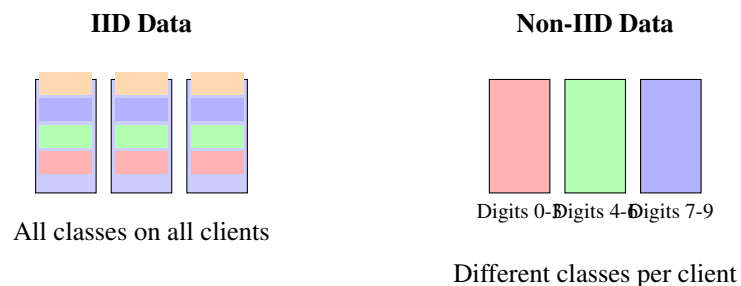


Figure 17.5: IID vs Non-IID data distribution

Non-IID Challenges

Non-IID data causes:

- Slower convergence (more rounds needed)
- Client drift (local optima diverge)
- Potential accuracy degradation

Mitigations: FedProx, client sampling strategies, or local data augmentation.

💡 When Is Federated Learning Worth the Complexity?

Federated learning adds significant complexity: client coordination, communication overhead, convergence challenges, and debugging difficulty. Before choosing FL, consider these scenarios:

Scenario A: Hospital network with 50 sites, patient health records, strict HIPAA compliance.

- Data cannot leave hospital premises by law
- Each site has thousands of records
- → FL is *required*—there's no legal alternative

Scenario B: Smart home company with 10,000 devices, non-sensitive temperature/humidity data.

- Data could be centralized (no privacy concerns)
- Devices have limited compute and unreliable connectivity
- → Consider: Is FL worth the complexity? Centralized might be simpler

Scenario C: Mobile keyboard with 1 million users, sensitive typing patterns.

- Users won't consent to uploading keystrokes

- Each user has unique patterns (highly non-IID)
- → FL enables training impossible otherwise, *and* improves personalization

Key questions: Is centralization legally/ethically impossible? Is the data naturally distributed? Do you have enough clients to make aggregation meaningful?

17.8 Three-Tier Implementation Guide

Run server + clients in separate terminals on same machine.

Tier 1: Single-Machine Simulation

Server on laptop/desktop, clients on other machines in same network.

Tier 2: Multi-Machine LAN

Deploy clients on Raspberry Pi devices with real sensor data.

Tier 3: Edge Deployment

17.8.1 Tier 3: Raspberry Pi Deployment

```

1 # client_pi.py - Raspberry Pi FL client
2
3 import flwr as fl
4 from sensor_data import collect_sensor_data # Your data collection
5
6 # Configuration
7 SERVER_IP = "192.168.1.100" # Your server's IP
8 CLIENT_ID = "pi_living_room"
9
10 def load_local_sensor_data():
11     """Load sensor data collected on this Pi."""
12     # Read from local database/files
13     x_train, y_train = collect_sensor_data()
14     return x_train, y_train
15
16 # Start client with reduced resources
17 if __name__ == "__main__":
18     # Reduce TensorFlow memory usage
19     import tensorflow as tf
20     tf.config.set_soft_device_placement(True)
21
22     (x_train, y_train), (x_test, y_test) = load_local_sensor_data()
23     model = create_model()
24     client = SensorClient(model, x_train, y_train, x_test, y_test)
25
26     fl.client.start_numpy_client(
27         server_address=f"{SERVER_IP}:8080",
28         client=client
29     )

```

Listing 17.9: Modifications for Pi deployment

17.9 Troubleshooting Guide

⚠ Flower Client Can't Connect to Server

Symptoms: `fl.client.start_numpy_client()` hangs or fails with connection error

Cause: Wrong server address, firewall blocking, or server not running

Solution:

1. Verify server running: Check server console shows "FL starting"
2. Use correct address: `fl.client.start_numpy_client(server_address="localhost:8080")`
3. Check firewall: Allow TCP port 8080 (or configured port)
4. For remote server: Use public IP, not localhost
5. Test connectivity: `telnet server_ip 8080` to verify port open
6. Increase timeout: Add `grpc_max_message_length` parameter

⚠ Federated Model Doesn't Converge

Symptoms: Accuracy stays low or oscillates wildly across rounds

Cause: Non-IID data distribution, too few clients, or learning rate too high

Solution:

1. Increase number of rounds: Try 20-50 rounds instead of 10
2. Reduce client learning rate: Use 0.001 instead of 0.01
3. Ensure balanced data: Each client should have similar number of samples
4. Use more clients per round: `min_fit_clients=5` instead of 2
5. Add server-side optimizer: Use FedAdam instead of FedAvg
6. Check individual client accuracy: Ensure clients can train locally before federating

⚠ FL Round Takes Too Long / Stragglers

Symptoms: Some clients much slower, round waits for slowest client

Cause: Heterogeneous hardware, network latency, or client doing too much computation

Solution:

1. Set client timeout: `config="timeout": 60` to drop slow clients
2. Use asynchronous FL: Don't wait for all clients (advanced)
3. Reduce local epochs: Train for 1-2 epochs per round instead of 5
4. Sample faster clients: Preferentially select clients with recent fast responses
5. Optimize client code: Profile to find bottlenecks
6. Use smaller batch size on slow clients: Adaptive batch sizing

17.10 Looking Ahead

Looking Ahead

Federated learning trains a shared global model across many devices, but individual users often need personalization. LAB18 explores on-device learning and continual adaptation, where edge devices fine-tune models on local data without server coordination. You'll learn transfer learning, few-shot adaptation, and catastrophic forgetting mitigation—enabling models that combine the collaborative benefits of federated learning with personalization to each user's unique patterns.

Communication Efficiency in Federated Learning

Bandwidth is the primary bottleneck in federated learning at scale:

- **Model size matters:** Full ResNet-50 (98MB) vs MobileNetV2 (14MB)—50x reduction via architecture choice
- **Gradient compression:** Sparsification (send only top-k gradients) reduces traffic by 75-90%
- **Quantization:** 8-bit weights vs 32-bit reduces communication by 4x with minimal accuracy loss
- **Reduce communication rounds:** Increase local epochs ($E=5$ vs $E=1$) to train more between updates
- Typical FL deployment: 1-10MB per client per round after optimization

Non-IID Data Challenges

Real federated learning deployments face heterogeneous data distributions:

- **Label skew:** Client 1 has digits 0-3, Client 2 has digits 7-9—different class distributions
- **Quantity skew:** Some clients have 100 samples, others 10,000—imbalanced contributions
- **Feature skew:** Hospital A has pediatric X-rays, Hospital B has geriatric—different domains
- **Impact:** Slower convergence (2-5x more rounds), lower final accuracy (5-15% drop)
- **Mitigations:** FedProx (proximal term), client sampling, data augmentation, weighted averaging

Privacy Guarantees of Federated Learning

Federated learning provides multiple layers of privacy protection:

- **Basic FL:** Raw data never leaves device—only model updates transmitted
- **Differential Privacy:** Add calibrated noise to gradients for formal privacy bounds (epsilon, delta)
- **Secure Aggregation:** Cryptographic protocols ensure server sees only aggregated updates, not individual contributions
- **Limitations:** Model updates can still leak information (membership inference, gradient inversion attacks)
- Best practice: Combine FL + differential privacy + secure aggregation for strong guarantees

17.11 Summary

- **Federated Learning:** Train models across devices without sharing raw data. Clients send model updates, not data.
- **FedAvg:** Core algorithm—server broadcasts model, clients train locally, server averages the updates weighted by sample count.
- **Flower Framework:** Production-ready FL framework. Provides NumPyClient interface and handles gRPC communication.
- **Server Role:** Coordinates rounds, selects clients, aggregates updates. Configure with FedAvg strategy.

- **Client Role:** Implements `get_parameters`, `fit`, and `evaluate`. Returns weights + sample count for weighted averaging.
- **Non-IID Data:** Real FL often has skewed data distributions. May require more rounds or advanced strategies.

17.12 Guided Experiments

Number of Clients Impact

Hypothesis: Increasing the number of federated learning clients improves final model accuracy due to more diverse data, but also increases communication overhead and convergence time, with diminishing returns beyond a certain client count.

Procedure:

1. Split MNIST into equal partitions for N clients
2. Test configurations: 2, 5, 10, 20, 50 clients
3. For each configuration, run 20 training rounds
4. Measure:
 - Final test accuracy
 - Total communication time (weight transfer)
 - Rounds to reach 95% accuracy
5. Keep samples per client constant (e.g., 1000 samples each)

Data Collection:

Num Clients	Final Accuracy (%)	Rounds to 95%	Comm Time (s)
2			
5			
10			
20			
50			

Analysis Questions:

- Plot accuracy vs number of clients. Where do gains plateau?
- How does communication time scale with client count?
- For a production system with 1000 devices, how would you sample clients per round?
- What happens if some clients have significantly more data than others?

Non-IID Data Impact

Hypothesis: Non-IID (non-independent and identically distributed) data across clients degrades federated learning convergence and final accuracy compared to IID data, with severity proportional to data heterogeneity.

Procedure:

1. Create data distributions for 10 clients:
 - IID: Each client has uniform random sample of all 10 digits
 - Mild non-IID: Each client biased toward 2-3 digits (70% of data)
 - Severe non-IID: Each client has only 1-2 digit classes
2. Train federated model for 30 rounds on each distribution
3. Measure per-round test accuracy and convergence stability

Data Collection:

Distribution	Final Accuracy (%)	Rounds to Converge	Accuracy Std Dev
IID			
Mild non-IID			
Severe non-IID			

Analysis Questions:

- Plot learning curves for all three scenarios. How does non-IID affect convergence?
- Why does severe non-IID hurt performance? (Hint: think about gradient directions)
- Propose mitigation strategies for non-IID data (e.g., client sampling, local epochs)
- In a real deployment, what causes data to be non-IID?

17.13 Exercises**See Also: LAB18 - On-Device Learning and Personalization**

Federated learning trains a shared global model across many devices. In LAB18, you'll learn how individual devices can personalize models:

- Fine-tune the global FL model on local user data
- Implement transfer learning for device-specific adaptation
- Balance global knowledge with local personalization
- Update models on-device without server coordination

FL builds collaborative models; on-device learning makes them personal to each user.

Self-Assessment Questions

Test your understanding before continuing:

1. Compare federated learning and traditional centralized training. What privacy advantages does FL provide? What are the communication and convergence trade-offs?
2. Explain how FedAvg aggregates client updates. Why do we weight by sample count? What happens if one client has 10x more data than others?
3. Why must all clients use the same model architecture in FedAvg? What would happen if Client 1 has 3 layers and Client 2 has 5 layers?
4. If `min_available_clients=3` but only 2 clients connect, what happens to the federated round? How would you handle intermittent client connectivity?
5. Explain how non-IID (non-independent and identically distributed) data affects FL convergence. If Client 1 only has digits 0-3 and Client 2 only has 7-9, why is training harder?

Increase Training Rounds

Modify the server to run 20 rounds instead of 5:

- Track accuracy after each round
- Plot accuracy vs round number
- Identify when accuracy saturates
- Compare final accuracy to centralized training

Non-IID Data Simulation

Modify data loading to create non-IID partitions:

- Client 1: Only digits 0-3
- Client 2: Only digits 4-6
- Client 3: Only digits 7-9

Compare convergence speed to IID partitioning.

Deploy on Raspberry Pi

Set up a real FL system:

- Run server on laptop
- Deploy clients on 2-3 Raspberry Pis
- Use real sensor data (temperature, motion)
- Measure training time per round

- **Further Reading.**
- Flower Documentation: <https://flower.dev/docs/>
 - Flower GitHub: <https://github.com/adap/flower>
 - FedAvg Paper: <https://arxiv.org/abs/1602.05629>
 - Federated Learning at Scale: <https://arxiv.org/abs/1902.01046>
 - FedProx for Non-IID: <https://arxiv.org/abs/1812.06127>

— **Credits.** Flower framework examples adapted from official documentation. FedAvg algorithm from McMahan et al.

Case Study: Multi-Hospital Pneumonia Detection via Federated Learning

Context: A consortium of 8 regional hospitals wanted to collaboratively train a pneumonia detection model from chest X-rays to improve diagnostic accuracy. Each hospital had 2,000-12,000 annotated X-ray images, but strict HIPAA regulations prohibited sharing patient data across institutions. Centralized training was legally impossible, and individual hospital datasets were too small—single-hospital models achieved only 78-82% accuracy. The consortium needed a privacy-preserving approach to train a shared model without exposing patient data.

Challenge: Initial attempts using a naive federated learning approach with FedAvg failed due to extreme data heterogeneity: (1) Hospital A specialized in pediatric care (mostly children's X-rays), Hospital B in geriatric patients, (2) X-ray equipment varied widely—4 different manufacturers with different image quality and contrast, (3) annotation quality differed—some hospitals had radiologist labels, others used resident annotations, (4) class imbalance varied—Hospital C had 40% positive pneumonia cases (ICU-focused), Hospital D only 8% (general screening). The federated model achieved only 81% accuracy after 100 rounds—no better than the best single-hospital model. Communication overhead was also extreme: transmitting full ResNet-50 models (98MB) every round over hospital VPNs took 25+ minutes.

Solution: The consortium implemented an optimized federated learning pipeline:

- Replaced ResNet-50 with MobileNetV2 and applied quantization-aware training

(98MB $\rightarrow$ 8.2MB)

- Implemented gradient compression: transmit only updated layers and apply 75% sparsification (8.2MB $\rightarrow$ 1.8MB/round)
- Used FedProx algorithm with proximal term ($\mu = 0.01$) to handle heterogeneous data and stragglers
- Applied differential privacy: add Gaussian noise (epsilon=8.0, delta= 10^{-5}) to gradients before aggregation
- Implemented weighted FedAvg: weight contributions by dataset size and validation performance
- Added client selection: randomly sample 5 of 8 hospitals per round to reduce communication and increase robustness
- Deployed secure aggregation protocol: server sees only aggregated gradients, never individual hospital updates
- Implemented data augmentation standardization: all hospitals apply same preprocessing (normalize, CLAHE, resize to 224x224)

Results:

- Model accuracy: 82% (best single hospital) $\rightarrow$ 91.3% (federated, 50 rounds)
- Convergence speed: 100+ rounds (FedAvg) $\rightarrow$ 50 rounds (FedProx + weighted averaging)
- Communication overhead: 98MB/round $\rightarrow$ 1.8MB/round (54x reduction via MobileNet + compression)
- Training time: 45 min/round $\rightarrow$ 8 min/round (faster model + gradient compression)
- Privacy guarantee: Differential privacy epsilon=8.0 with formal privacy bounds
- Fairness: Per-hospital accuracy improved for all 8 hospitals (weakest: 78% $\rightarrow$ 89%)
- Deployment: Model deployed to all 8 hospitals, processing 2,500+ X-rays/day with 91% accuracy

Lessons Learned:

- Data heterogeneity is the norm in federated learning—FedProx outperforms FedAvg for non-IID medical data
- Model size dominates communication cost—use lightweight architectures (MobileNet) + quantization from day one
- Gradient compression (sparsification, quantization) provides massive bandwidth savings with minimal accuracy loss
- Differential privacy is essential for medical federated learning—but high noise (epsilon < 3) severely degrades accuracy
- Data augmentation standardization is critical—hospitals must apply identical preprocessing for model to generalize
- Client selection (sampling K of N participants) improves convergence and reduces stragglers without hurting accuracy
- Secure aggregation adds 15-20% overhead but is mandatory for true privacy guarantees
- Weighted averaging by dataset quality (validation accuracy) improves fairness and overall model performance

17.14 Exercises

1. **[Challenge] Challenge: Non-IID Data Partitioning and Convergence Analysis:** Create and compare three different data partitioning strategies for federated MNIST: (1) IID random

partitioning, (2) label-skewed where each client has only 2-3 digit classes, and (3) quantity-skewed where clients have vastly different dataset sizes (e.g., 100, 500, 2000 samples). Run FedAvg for 50 rounds on each partitioning scheme, tracking global model accuracy and per-client accuracy after each round. Analyze convergence speed, final accuracy, and fairness across clients. Implement FedProx algorithm and compare its robustness to non-IID data against vanilla FedAvg.

2. **[Challenge] Challenge: Federated Learning with Raspberry Pi Edge Nodes:** Deploy a production federated learning system with Flower server on laptop/cloud and 3+ Raspberry Pi devices as clients. Each Pi collects real sensor data (temperature, motion, light) or processes local image datasets. Implement client selection based on battery level and network quality. Measure communication overhead (bytes sent/received per round), training time per round on constrained hardware, and impact of stragglers (slow clients). Add monitoring dashboard showing real-time training progress, client participation, and model performance. Test resilience by simulating client failures and network interruptions.
3. **[Challenge] Challenge: Privacy-Preserving Federated Learning with Differential Privacy:** Enhance your Flower implementation with differential privacy to protect individual client data. Implement gradient clipping and Gaussian noise addition to client updates before sending to server (use TensorFlow Privacy or similar). Experiment with different privacy budgets ($\epsilon = 0.1, 1.0, 10.0$) and measure the privacy-accuracy trade-off. Implement secure aggregation where the server cannot see individual client updates, only aggregated results. Verify privacy guarantees through membership inference attacks—train a model with and without specific training samples and measure if an attacker can distinguish which data was used.
4. **[Challenge] Challenge: Personalized Federated Learning with Local Adaptation:** Implement personalized FL where each client maintains both a global shared model and a personalized local model. After each federated round, clients fine-tune the global model on their local data for 2-3 epochs. Compare three strategies: (1) full personalization (completely local model), (2) no personalization (pure global model), and (3) hybrid (global base + personalized head layers). Evaluate on heterogeneous client data distributions (e.g., digit 0-3 for client 1, digits 4-6 for client 2, etc.). Measure per-client accuracy for each strategy and analyze when personalization provides significant benefits over the global model.

DESCRIPTIVE

What
happened



DIAGNOSTIC

Why it
happened



PREDICTIVE

What will
happen



PRESCRIPTIVE

What action
to take



18. On-Device Learning and Model Adaptation •

Learning Objectives

After completing this lab, you will be able to:

- Understand why deployed models need continuous adaptation
- Implement transfer learning for edge devices with frozen base layers
- Design incremental learning systems that prevent catastrophic forgetting
- Apply personalization techniques to create user-specific models
- Build on-device training pipelines with version control and rollback

Prerequisites

Before starting this lab, ensure you:

- Completed LAB02 (Neural Network Training Fundamentals): strong TensorFlow/Keras foundation
- Completed LAB17 (Federated Learning): understanding of distributed training and model adaptation
- Understanding of transfer learning concepts (pre-training, fine-tuning)
- Familiarity with gradient descent and backpropagation basics
- Understanding of overfitting and regularization concepts

Interactive Notebook

Jupyter Notebook: LAB18_onddevice_learning.ipynb

Open in Google Colab | View on GitHub | Companion Website

Prerequisites

- **Knowledge:** Transfer learning concepts, TFLite (LAB03, LAB05)
- **Level 1:** Laptop with TensorFlow 2.x
- **Level 2:** TensorFlow Lite with training support
- **Level 3:** Raspberry Pi or ESP32 for on-device adaptation

■ **Theory meets practice.** On-device learning represents the frontier of edge AI—models that improve after deployment without cloud connectivity. This lab moves from concept to implementation by building on deployment work from LAB05, profiling insights from LAB11, federated learning ideas from LAB17, and energy considerations from LAB15.

Execution tiers:

- **Level 1 (Notebook):** Simulate on-device learning and drift scenarios in Python using standard TensorFlow
- **Level 2 (TFLite):** Use TensorFlow Lite’s on-device training or fine-tuning capabilities on a Raspberry Pi or similar edge node
- **Level 3 (Device):** Deploy adaptive models on Raspberry Pi or MCU-class boards, focusing on which parts of the model can realistically be updated on-device

Three-Tier Execution Model

Tier 1: Jupyter/Colab

Implement transfer learning with frozen base layers, simulate user-specific data distributions, build incremental learning with replay buffers

Tier 2: Raspberry Pi

Deploy TFLite models with on-device training support, fine-tune classifier heads on local data, measure adaptation time and memory usage

Tier 3: Raspberry Pi/ESP32

Implement personalized models that adapt to user patterns, use version control with rollback, optimize for energy-constrained continuous learning

18.1 Why Models Need to Adapt

⚠ Catastrophic Forgetting Without Replay

The most insidious on-device learning bug: your model improves on new data but silently forgets everything it knew before. You deploy an update, users report that “the old stuff doesn’t work anymore,” and you have no idea why.

What happens:

- User provides 50 examples of a new gesture
- Model trains on those 50 examples
- Accuracy on the new gesture: 95%!
- Accuracy on original gestures: drops from 90% to 40%

Why it happens: Neural networks overwrite old knowledge when learning new patterns. Gradient updates push weights toward the new task, away from the old.

Prevention:

1. Always use a replay buffer mixing old and new data
2. Test on *all* classes after adaptation, not just new ones
3. Keep a frozen baseline model for comparison
4. Implement automatic rollback if performance regresses

When you deploy an ML model, the real world immediately starts to differ from your training data. On-device learning enables models to evolve continuously.

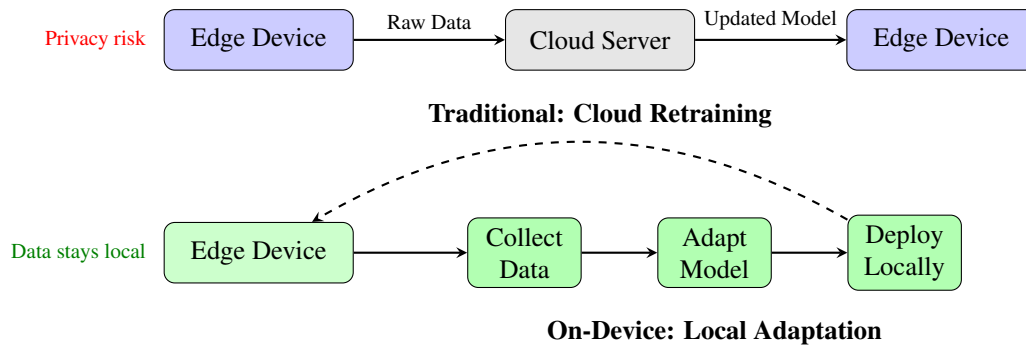


Figure 18.1: Cloud retraining vs. on-device learning: keeping data private while adapting to change

18.1.1 Types of Distribution Shift

Four Reasons Models Drift

1. **Data drift:** Input distribution changes (seasons, trends, behavior)
2. **Concept drift:** Relationship between inputs and outputs changes
3. **Personalization:** Each user has unique patterns the general model misses
4. **Environmental shift:** Sensors age, lighting changes, noise varies

```

1 # Detecting distribution shift with simple statistics
2 import numpy as np
3
4 def detect_drift(baseline_mean, baseline_std, new_samples, threshold=2.0):
5     """Detect if new data differs significantly from training data"""
6     new_mean = np.mean(new_samples)
7     z_score = abs(new_mean - baseline_mean) / baseline_std
8
9     if z_score > threshold:
10         print(f"DRIFT DETECTED: z-score = {z_score:.2f}")
11         return True
12     return False

```

Code Walkthrough

We compute how many standard deviations the new data mean is from the training mean. A z-score above 2.0 indicates significant drift warranting model adaptation.

18.2 Mathematical Foundations of Continual Learning

On-device learning requires rigorous mathematical foundations to understand why models forget, how to prevent it, and the trade-offs involved. This section formalizes the key concepts.

18.2.1 The Catastrophic Forgetting Problem

Definition — Catastrophic Forgetting. When training on task B after task A , the loss on task A increases dramatically:

$$\mathcal{L}_A(\theta_B^*) \gg \mathcal{L}_A(\theta_A^*)$$

where θ_A^* are optimal weights for task A and θ_B^* are weights after training on B .

More precisely, let $\mathcal{D}_A = \{(x_i^A, y_i^A)\}_{i=1}^{n_A}$ and $\mathcal{D}_B = \{(x_j^B, y_j^B)\}_{j=1}^{n_B}$ be datasets for tasks A and B . Standard sequential training optimizes:

$$\theta_A^* = \arg \min_{\theta} \mathcal{L}_A(\theta) = \arg \min_{\theta} \frac{1}{n_A} \sum_{i=1}^{n_A} \ell(f_{\theta}(x_i^A), y_i^A)$$

$$\theta_B^* = \arg \min_{\theta} \mathcal{L}_B(\theta) = \arg \min_{\theta} \frac{1}{n_B} \sum_{j=1}^{n_B} \ell(f_{\theta}(x_j^B), y_j^B)$$

When initializing task B training with θ_A^* , gradient descent moves weights to minimize $\mathcal{L}_B(\theta)$, which typically increases $\mathcal{L}_A(\theta)$ since the loss landscapes are misaligned.

Why Catastrophic Forgetting Happens

Neural networks store knowledge distributedly across weights. Training on new data overwrites weights important for previous tasks because standard gradient descent has no mechanism to protect important parameters. Mathematically, gradient updates:

$$\theta_{t+1} = \theta_t - \eta \nabla_{\theta} \mathcal{L}_B(\theta_t)$$

push weights toward regions optimal for task B , with no constraint to preserve performance on task A . The gradient $\nabla_{\theta} \mathcal{L}_B(\theta_t)$ is computed solely from data in $\mathcal{D}_B$, making it “blind” to task A requirements.

Empirical evidence: When training MNIST digit classifier on digits 0-4, then on 5-9 without replay, accuracy on 0-4 drops from 95% to <10%. The weight changes $\Delta\theta = \theta_B^* - \theta_A^*$ are large enough to completely erase the decision boundaries learned for the first task.

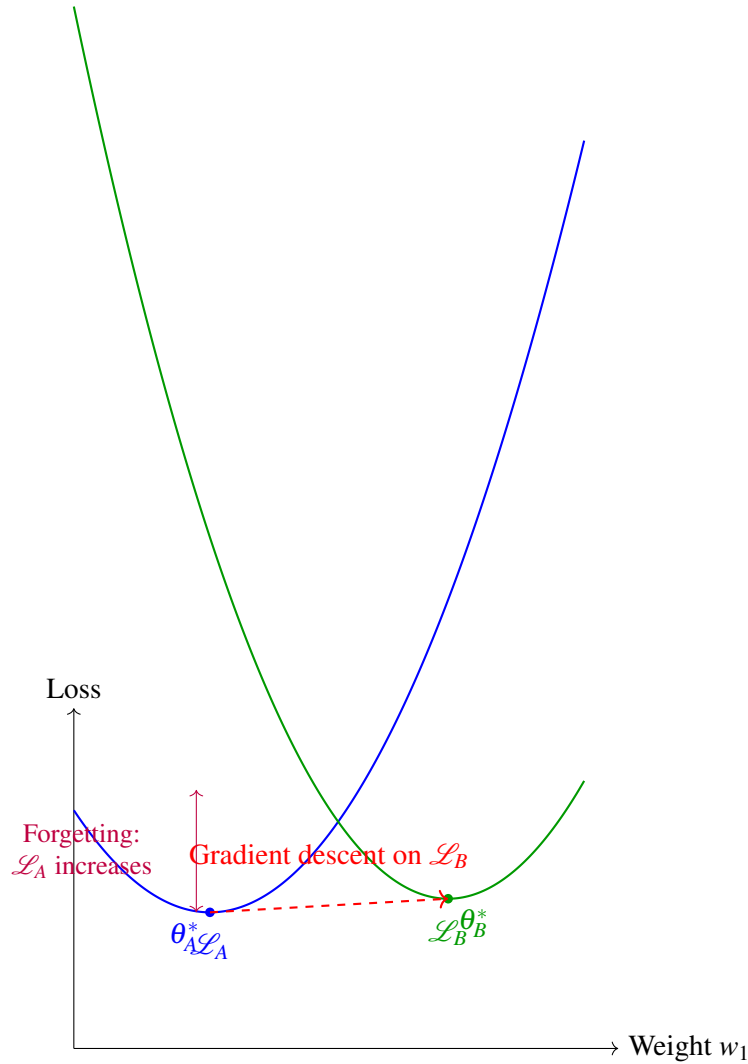


Figure 18.2: Catastrophic forgetting: moving weights to minimize $\mathcal{L}_B$ increases $\mathcal{L}_A$

18.2.2 The Stability-Plasticity Dilemma

Stability-Plasticity Dilemma

Continual learning systems face a fundamental trade-off:

- **Plasticity:** Ability to learn new patterns from new data (adapt to task B)
- **Stability:** Ability to retain previously learned knowledge (preserve task A)

Mathematically, we seek weights θ^* that simultaneously minimize:

$$\theta^* = \arg \min_{\theta} [\alpha \mathcal{L}_A(\theta) + (1 - \alpha) \mathcal{L}_B(\theta)]$$

where $\alpha \in [0, 1]$ balances old vs. new knowledge. The challenge: we no longer have access to $\mathcal{D}_A$ during task B training, so we cannot directly compute $\mathcal{L}_A(\theta)$ without storing all historical data. Continual learning methods provide different approximations to this ideal joint optimization.

Why neural networks forget: Gradient descent on task B has no inherent mechanism to preserve task A performance. Updates $\theta_{t+1} = \theta_t - \eta \nabla_{\theta} \mathcal{L}_B(\theta_t)$ move weights toward

minimizing $\mathcal{L}_B$ regardless of impact on $\mathcal{L}_A$. The parameter space where both tasks perform well (if it exists) is typically a small intersection, difficult to find via gradient descent starting from θ_A^* .

18.2.3 Elastic Weight Consolidation (EWC)

EWC addresses catastrophic forgetting by adding a quadratic penalty that protects important weights from changing.

Definition — EWC Loss Function. EWC augments the task B loss with a regularization term that penalizes deviations from task A optimal weights:

$$\mathcal{L}_{EWC}(\theta) = \mathcal{L}_B(\theta) + \frac{\lambda}{2} \sum_{i=1}^{|\theta|} F_i (\theta_i - \theta_{A,i}^*)^2$$

where:

- $\mathcal{L}_B(\theta)$ is the loss on the new task (task B)
- F_i is the Fisher information for parameter i , measuring its importance for task A
- θ_A^* are the optimal weights after learning task A
- $\lambda > 0$ controls the importance of old knowledge (typical values: 100–10,000)

The EWC objective encourages finding weights that perform well on task B while staying close to θ_A^* on dimensions where F_i is large (important parameters).

Intuition: Imagine weights as springs attached to their task A values θ_A^* . The Fisher information F_i determines spring stiffness—important weights have stiff springs (hard to move), unimportant weights have loose springs (free to adapt). Learning task B pulls weights toward new optima, but springs resist proportionally to importance.

Definition — Fisher Information Matrix. The Fisher information measures parameter importance for the current task. For parameter θ_i :

$$F_i = \mathbb{E}_{x \sim \mathcal{D}_A} \left[\left(\frac{\partial \log p(y|x, \theta_A^*)}{\partial \theta_i} \right)^2 \right]$$

In practice, we approximate this expectation using empirical samples:

$$F_i \approx \frac{1}{n_A} \sum_{k=1}^{n_A} \left(\frac{\partial \log p(y_k|x_k, \theta_A^*)}{\partial \theta_i} \right)^2$$

Interpretation:

- High F_i means parameter θ_i is highly sensitive—small changes cause large loss increases
- Low F_i means parameter θ_i can be changed freely without hurting task A performance
- The Fisher information is the expected squared gradient of the log-likelihood

For classification with softmax outputs, the gradient of the log-likelihood is:

$$\frac{\partial \log p(y|x, \theta)}{\partial \theta_i} = \frac{\partial}{\partial \theta_i} \left[\log \frac{e^{z_y}}{\sum_c e^{z_c}} \right]$$

where $z_c = f_\theta(x)_c$ is the logit for class c .

EWC Gradient Update

The gradient of the EWC loss combines task-specific and regularization terms:

$$\nabla_{\theta} \mathcal{L}_{EWC}(\theta) = \nabla_{\theta} \mathcal{L}_B(\theta) + \lambda \sum_{i=1}^{|\theta|} F_i(\theta_i - \theta_{A,i}^*) \mathbf{e}_i$$

where $\mathbf{e}_i$ is the i -th unit vector.

The regularization term acts as a “spring” pulling weights back toward θ_A^* , with stiffness proportional to F_i . Parameters with high Fisher information resist change, while unimportant parameters can adapt freely to task B .

Gradient descent update:

$$\theta_{t+1} = \theta_t - \eta [\nabla_{\theta} \mathcal{L}_B(\theta_t) + \lambda F \odot (\theta_t - \theta_A^*)]$$

where $\odot$ denotes element-wise multiplication.

Table 18.1: EWC hyperparameter λ impact

λ	Task A Retention	Task B Learning	Use Case
$\lambda = 0$	Poor (forgetting)	Excellent	Naive sequential training
$\lambda = 100$	Fair (some forgetting)	Good	Fast adaptation, acceptable forgetting
$\lambda = 1000$	Good	Fair	Balanced continual learning
$\lambda = 10000$	Excellent	Poor (slow learning)	Critical knowledge preservation
$\lambda \rightarrow \infty$	Perfect (frozen)	None	Equivalent to freezing weights

18.2.4 Experience Replay

Replay buffers prevent forgetting by interleaving old and new data during training.

Definition — Replay Buffer Training. Maintain a buffer $\mathcal{B} = \{(x_k, y_k)\}_{k=1}^K$ storing representative samples from task A . When training on task B , sample mini-batches from the combined dataset $\mathcal{D}_B \cup \mathcal{B}$:

$$\theta_{t+1} = \theta_t - \eta \nabla_{\theta} \left[\frac{1}{|\mathcal{M}_B|} \sum_{(x,y) \in \mathcal{M}_B} \ell(f_{\theta}(x), y) + \frac{1}{|\mathcal{M}_A|} \sum_{(x,y) \in \mathcal{M}_A} \ell(f_{\theta}(x), y) \right]$$

where $\mathcal{M}_B \subset \mathcal{D}_B$ is a mini-batch from new data and $\mathcal{M}_A \subset \mathcal{B}$ is a mini-batch from the buffer.

Mixing ratio: Typically use equal proportions ($|\mathcal{M}_A| = |\mathcal{M}_B|$) or weight by dataset size.

Definition — Reservoir Sampling. To maintain a fixed-size buffer $\mathcal{B}$ of K samples from a stream of n items (where $n \gg K$ and grows over time):

Algorithm:

1. Store first K items directly into $\mathcal{B}$
2. For each new item x_i where $i > K$:
 - With probability $p = K/i$, include x_i in buffer
 - If included, replace uniformly random item in $\mathcal{B}$ with x_i

Proof of uniform sampling: This algorithm ensures each item from the stream of n items

has equal probability K/n of being in the buffer. By induction:

- *Base case* ($i = K$): All K items are in buffer with probability 1
- *Inductive step*: If each of i items has probability K/i of being in buffer, then for item $i + 1$:
 - Item $i + 1$ enters with probability $K/(i + 1)$
 - Each existing item stays with probability $1 - \frac{K/(i+1) \cdot 1/K}{1} = 1 - 1/(i + 1) = i/(i + 1)$
 - Combined: each existing item has probability $\frac{K}{i} \cdot \frac{i}{i+1} = \frac{K}{i+1}$

Buffer Size Trade-offs

Larger buffers prevent forgetting better but consume more memory:

$$\text{Memory}_{\text{buffer}} = K \times (\text{input size} + \text{label size})$$

For images of size $28 \times 28 \times 1$ (MNIST) with 1-byte labels:

$$\text{Memory} = K \times (784 + 1) \approx 785K \text{ bytes}$$

On edge devices:

- **ESP32 (520KB RAM)**: $K \approx 50$ – 100 images (40–80 KB)
- **Raspberry Pi 4 (4GB RAM)**: $K \approx 10,000$ images (7.5 MB), no practical limit
- **Arduino Nano 33 BLE (256KB RAM)**: $K \approx 20$ – 50 images (15–40 KB)

Rule of thumb: Allocate 10–20 samples per class. For 10-class classification, use $K = 100$ – 200 .

Table 18.2: Continual learning strategy comparison with complexity analysis

Strategy	Memory	Compute Overhead	Forgetting Prevention	Best For
Naive Sequential	$O(1)$	None	Poor	Experimental only
Replay Buffer	$O(K)$	None (mixing only)	Excellent	Devices with >100 MB RAM
EWC	$O(\theta)$	$O(\theta)$ per update	Good	Memory-constrained (<100 MB)
Progressive Networks	$O(T \cdot \theta)$	$O(T)$ inference	Perfect (no interference)	Multi-task, growing capacity
Knowledge Distillation	$O(1)$	$O(\theta)$ (soft targets)	Good	Model compression

18.2.5 Other Continual Learning Methods

Beyond EWC and replay buffers, several advanced techniques address catastrophic forgetting.

Definition — Progressive Neural Networks. Progressive networks grow capacity by adding new neural network columns for each task while freezing previous columns.

Architecture: After learning task A with network f_{θ_A} , introduce new network f_{θ_B} for task B

with lateral connections:

$$h_l^{(B)} = \sigma \left(W_l^{(B)} h_{l-1}^{(B)} + \sum_{i=1}^{l-1} U_{l,i}^{(B \leftarrow A)} h_i^{(A)} \right)$$

where $h_l^{(B)}$ is the activation at layer l of task B network, $W_l^{(B)}$ are standard weights, and $U_{l,i}^{(B \leftarrow A)}$ are lateral connections from task A network layer i to task B layer l .

Properties:

- **Zero forgetting:** Previous task columns are frozen, guaranteeing no interference
- **Transfer learning:** Lateral connections enable feature reuse from earlier tasks
- **Memory cost:** Grows linearly with number of tasks: $O(T \cdot |\theta|)$ for T tasks
- **Inference cost:** Must evaluate all task columns: $O(T)$ computational overhead
- **On-device feasibility:** Not practical for resource-constrained edge devices due to unbounded growth. Suitable for edge servers or when number of tasks is small and known ($T \leq 5$).

Definition — Learning without Forgetting (LwF). LwF preserves old task knowledge using knowledge distillation without storing old data.

Training objective: When learning task B , minimize:

$$\mathcal{L}_{LwF}(\theta) = \mathcal{L}_B(\theta) + \lambda \sum_{x \in \mathcal{D}_B} \text{KL} \left(p^{(A)}(y|x, \theta_A^*) \| p^{(A)}(y|x, \theta) \right)$$

where the first term is standard task B loss and the second term ensures predictions on task A outputs don't change. The key insight: use task B data to preserve task A outputs via distillation.

Algorithm:

1. Before learning task B , record predictions $\hat{y}_i^{(A)} = f_{\theta_A^*}(x_i)$ for all $x_i \in \mathcal{D}_B$
2. While training on task B , minimize combined loss:
 - Task loss: $\mathcal{L}_B = \text{CE}(y_i^{(B)}, f_{\theta}(x_i))$ using true task B labels
 - Distillation loss: $\mathcal{L}_{dist} = \text{KL}(\hat{y}_i^{(A)}, f_{\theta}(x_i))$ preserving task A outputs

Memory efficiency: Requires storing only soft predictions $\hat{y}_i^{(A)}$ (not raw data), reducing memory by factor of input dimensionality. For MNIST: store 10 logits vs. 784 pixels per sample ($78\times$ reduction).

Limitations: Assumes task B data distribution overlaps with task A . If disjoint distributions, distillation loss provides no gradient signal for preserving task A .

Definition — Gradient Episodic Memory (GEM). GEM uses a small memory buffer to constrain gradient updates, ensuring new learning doesn't increase loss on old tasks.

Constraint optimization: When training on task B , solve:

$$\min_{\theta} \mathcal{L}_B(\theta) \quad \text{subject to} \quad \langle \nabla_{\theta} \mathcal{L}_A(\theta), \theta - \theta_t \rangle \geq 0$$

This ensures the gradient update $\Delta\theta = \theta_{t+1} - \theta_t$ doesn't increase loss on task A (positive dot product means loss decreases or stays constant).

Practical implementation: Store episodic memory $\mathcal{M}_A \subset \mathcal{D}_A$ of K samples. At each update:

1. Compute task B gradient: $g_B = \nabla_{\theta} \mathcal{L}_B(\theta_t)$
2. Compute task A gradient on memory: $g_A = \nabla_{\theta} \mathcal{L}_A(\theta_t; \mathcal{M}_A)$

3. If $\langle g_B, g_A \rangle < 0$ (conflicting gradients), project:

$$g_{projected} = g_B - \frac{\langle g_B, g_A \rangle}{\|g_A\|^2} g_A$$

4. Update with projected gradient: $\theta_{t+1} = \theta_t - \eta g_{projected}$

Geometric interpretation: GEM projects the task B gradient onto the space tangent to the task A loss surface, ensuring updates improve task B without harming task A .

Complexity: Memory $O(K)$ for buffer, computation $O(|\theta|)$ per update for gradient projection. More computationally expensive than replay but with stronger forgetting guarantees.

18.2.6 Online Learning Mathematics

On-device models must adapt to streaming data in real-time, requiring specialized optimization strategies.

Definition — Stochastic Gradient Descent for Streaming Data. In online learning, data arrives sequentially $(x_1, y_1), (x_2, y_2), \dots$ and the model updates after each sample (or mini-batch):

$$\theta_{t+1} = \theta_t - \eta_t \nabla_{\theta} \ell(f_{\theta_t}(x_t), y_t)$$

Key differences from batch training:

- **Single-pass learning:** Each sample seen once (no epochs)
- **Non-stationary distribution:** $p(x_t, y_t)$ may change over time
- **Adaptive learning rate:** η_t must decrease to ensure convergence

Convergence condition (Robbins-Monro): Learning rate schedule $\{\eta_t\}$ must satisfy:

$$\sum_{t=1}^{\infty} \eta_t = \infty \quad \text{and} \quad \sum_{t=1}^{\infty} \eta_t^2 < \infty$$

Example schedules:

- Polynomial decay: $\eta_t = \frac{\eta_0}{(1+t)^\alpha}$ with $\alpha \in [0.5, 1]$
- Exponential decay: $\eta_t = \eta_0 \gamma^t$ with $\gamma \in [0.95, 0.99]$
- Step decay: $\eta_t = \eta_0 \times 0.1^{\lfloor t/T \rfloor}$ at fixed intervals T

Learning Rate Schedules for Edge Streaming

On edge devices processing streaming data, learning rate must balance three objectives:

Fast adaptation: Large η enables rapid response to distribution shifts

$$\text{Convergence time} \propto \frac{1}{\eta}$$

Stability: Small η prevents oscillation from noisy gradients

$$\text{Variance}[\theta_t] \propto \eta^2 \sigma_g^2$$

where σ_g^2 is gradient variance.

Forgetting prevention: Decreasing η over time consolidates learning

$$\mathbb{E}[\|\theta_T - \theta^*\|^2] \leq \frac{C_1}{\sum_{t=1}^T \eta_t} + C_2 \sum_{t=1}^T \eta_t^2$$

Practical edge strategy:

1. Start with moderate rate: $\eta_0 = 0.001$
2. Decay every N samples: $\eta_t = \eta_0 \times 0.95^{\lfloor t/N \rfloor}$ where $N = 100$
3. Reset learning rate on detected distribution shift: if $D_{KL}(p_{\text{recent}} \| p_{\text{baseline}}) > \tau$, set $\eta = \eta_0$
4. Use smaller final rate: $\eta_{\min} = 0.0001$ to prevent drift

Definition — Mini-Batch Considerations on Edge. Edge devices face memory constraints limiting batch size. For device with RAM M and model size $|\theta|$:

Memory budget:

$$M_{\text{available}} = M - M_{\text{OS}} - 2|\theta| \times 4 \text{ bytes} - M_{\text{other}}$$

Maximum batch size:

$$B_{\max} = \left\lfloor \frac{M_{\text{available}}}{d_{\text{input}} \times 4 + d_{\text{output}} \times 4 + M_{\text{activations}}} \right\rfloor$$

where d_{input} , d_{output} are input/output dimensions and $M_{\text{activations}}$ is intermediate activation memory.

Example (ESP32 with 520KB RAM):

- Available RAM: $520 - 100 \text{ (OS)} - 200 \text{ (model+gradients)} = 220 \text{ KB}$
- MNIST input: $28 \times 28 = 784 \text{ floats} \times 4 \text{ bytes} = 3.1 \text{ KB}$
- With activations: $\approx 5 \text{ KB}$ per sample
- Maximum batch: $B_{\max} \approx 40$ samples
- Practical batch: $B = 8\text{--}16$ (leave margin for safety)

Batch size vs. convergence trade-off:

$$\text{Update variance} \propto \frac{1}{B}, \quad \text{Updates per second} \propto \frac{1}{B \times T_{\text{backprop}}}$$

Smaller batches provide noisier gradients but more frequent updates, often beneficial for non-stationary distributions.

18.2.7 Knowledge Distillation for Continual Learning

Knowledge distillation transfers knowledge from one model to another using soft probability distributions.

Definition — Temperature-Scaled Softmax. Standard softmax with temperature parameter T :

$$q_i = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)}$$

where z_i are logits and $T > 0$ is temperature.

Temperature effects:

- $T = 1$: Standard softmax (sharp distribution)
- $T > 1$: Softened probabilities reveal model's uncertainty and similarities between classes
- $T \rightarrow \infty$: Uniform distribution $q_i \rightarrow 1/C$ for C classes
- $T \rightarrow 0$: One-hot distribution $q_i \rightarrow \mathbb{I}[i = \arg \max_j z_j]$

Example: For logits $[2.0, 1.0, 0.1]$:

- $T = 1$: $[0.659, 0.242, 0.099]$ (standard)

- $T = 3$: [0.422, 0.321, 0.257] (softer, reveals second-best class)
- $T = 10$: [0.354, 0.333, 0.313] (nearly uniform)

Soft targets with $T > 1$ contain more information: relative probabilities reveal learned structure beyond winner-take-all classification.

Definition — Distillation Loss. Train student model to mimic teacher outputs using combination of hard and soft targets:

$$\mathcal{L}_{\text{distill}} = \alpha \mathcal{L}_{\text{hard}}(y, f_{\theta_S}(x)) + (1 - \alpha) T^2 \mathcal{L}_{\text{soft}}(f_{\theta_T}(x)/T, f_{\theta_S}(x)/T)$$

where:

- $\mathcal{L}_{\text{hard}} = \text{CE}(y, f_{\theta_S}(x))$ is cross-entropy with true labels y
- $\mathcal{L}_{\text{soft}} = \text{KL}(f_{\theta_T}(x)/T \| f_{\theta_S}(x)/T)$ is KL divergence between teacher and student
- $\alpha \in [0, 1]$ balances true labels vs. teacher guidance (typically $\alpha = 0.1$)
- $T > 1$ is temperature for soft targets (typically $T = 3\text{--}5$)
- T^2 scaling factor accounts for temperature's effect on gradient magnitude

Gradient analysis: The soft loss gradient is:

$$\nabla_{\theta_S} \mathcal{L}_{\text{soft}} = \frac{1}{T} \mathbb{E}_x [(q_T - q_S) \nabla_{\theta_S} f_{\theta_S}(x)]$$

where q_T, q_S are teacher/student probabilities. Higher temperature T amplifies small probability differences, transferring finer-grained knowledge.

Self-Distillation for Continual Learning

Use the model's own previous version as teacher to preserve old knowledge while adapting:

Algorithm:

1. Train model on task A to convergence: θ_A^*
2. When task B arrives, use θ_A^* as frozen teacher
3. Train new model with distillation loss:

$$\mathcal{L} = \mathcal{L}_B(\theta) + \lambda T^2 \text{KL}(f_{\theta_A^*}(x)/T \| f_{\theta}(x)/T)$$

where $x \sim \mathcal{D}_B$ (task B data used for both objectives)

Advantages over LwF:

- No need to store previous predictions (teacher computed on-the-fly)
- Works with shared task data when distributions overlap
- Softer constraint than EWC (transfers output behavior, not weights)

Memory cost: Must store teacher model θ_A^* in addition to training model θ . For edge deployment:

- Quantize teacher to int8: reduces size by $4\times$
- Use same architecture: share feature extraction, only duplicate classifier head
- Prune teacher: keep only essential weights for distillation

18.2.8 Evaluation Metrics for Continual Learning

Proper evaluation requires measuring both learning ability and forgetting across multiple tasks.

Definition — Average Accuracy After Task T . After learning sequence of tasks $1, 2, \dots, T$,

the average accuracy is:

$$A_T = \frac{1}{T} \sum_{i=1}^T a_{T,i}$$

where $a_{T,i}$ is the accuracy on task i after training on task T . This measures overall performance retention.

Upper bound: Joint training on all tasks gives A_T^{joint} . Continual learning aims to approach this:

$$\text{Performance gap} = A_T^{\text{joint}} - A_T$$

Definition — Backward Transfer (Forgetting Measure). Backward transfer quantifies how much learning new tasks causes forgetting of old tasks:

$$\text{BWT} = \frac{1}{T-1} \sum_{i=1}^{T-1} (a_{T,i} - a_{i,i})$$

where $a_{i,i}$ is accuracy on task i immediately after training on task i , and $a_{T,i}$ is accuracy on task i after training on all tasks $1, \dots, T$.

Interpretation:

- $\text{BWT} < 0$: Forgetting occurred (later tasks hurt earlier task performance)
- $\text{BWT} = 0$: No forgetting (ideal stability)
- $\text{BWT} > 0$: Positive backward transfer (later tasks improved earlier task performance via shared representations)

Example: Three tasks with accuracies:

After task	Task 1	Task 2	Task 3
1	90%	-	-
2	85%	88%	-
3	80%	82%	91%

$$\text{BWT} = \frac{1}{2} [(80 - 90) + (82 - 88)] = \frac{-10 - 6}{2} = -8\%$$

The model forgot 8% on average from earlier tasks.

Definition — Forward Transfer. Forward transfer measures whether learning earlier tasks helps learning later tasks:

$$\text{FWT} = \frac{1}{T-1} \sum_{i=2}^T (a_{i-1,i} - b_i)$$

where $a_{i-1,i}$ is accuracy on task i after training on tasks $1, \dots, i-1$ (before training on i), and b_i is accuracy on task i with random initialization (no prior training).

Interpretation:

- $\text{FWT} > 0$: Positive transfer (earlier tasks help later tasks)
- $\text{FWT} = 0$: No transfer (later tasks learn independently)
- $\text{FWT} < 0$: Negative transfer (earlier tasks hurt later task learning)

Why measure FWT: Helps distinguish methods that prevent forgetting via isolation (no transfer) vs. methods that enable knowledge sharing across tasks.

Complete Evaluation Protocol

For rigorous continual learning evaluation on sequence of T tasks:

Metrics to report:

1. **Final average accuracy:** $A_T = \frac{1}{T} \sum_{i=1}^T a_{T,i}$
2. **Forgetting:** $BWT = \frac{1}{T-1} \sum_{i=1}^{T-1} (a_{T,i} - a_{i,i})$
3. **Forward transfer:** $FWT = \frac{1}{T-1} \sum_{i=2}^T (a_{i-1,i} - b_i)$
4. **Learning curve:** Plot $a_{t,t}$ (accuracy on current task during training)
5. **Forgetting curve:** Plot $a_{t,i}$ for fixed early task i as t increases

Baselines to compare:

- **Upper bound:** Joint training on all tasks (multitask learning)
- **Lower bound:** Sequential training without forgetting prevention
- **Fine-tuning:** Train on each task independently (reset weights)

Statistical significance: Run experiments with 3–5 random seeds, report mean $\pm$ standard deviation. Use paired t-tests for comparing methods.

Table 18.3: Example continual learning evaluation results

Method	Avg Acc (%)	BWT (%)	FWT (%)	Memory	Time
Joint Training	94.5 ± 0.3	-	-	High	High
Sequential (Naive)	45.2 ± 2.1	-42.3	0.0	Low	Low
Replay (K=200)	89.7 ± 0.8	-3.2	+1.5	Medium	Medium
EWC ($\lambda=5000$)	82.1 ± 1.2	-8.9	+0.3	Low	Medium
Progressive Nets	93.8 ± 0.5	0.0	+4.2	Very High	High
GEM (K=100)	87.3 ± 1.0	-5.1	+2.1	Medium	High

18.2.9 On-Device Learning Constraints

Edge devices impose strict constraints that affect which continual learning techniques are feasible.

Memory Constraints for Fisher Information

EWC requires storing the Fisher information matrix, which has the same dimensionality as the model parameters.

Fisher matrix memory: For model with $|\theta|$ parameters:

$$M_{\text{Fisher}} = |\theta| \times 4 \text{ bytes (float32)} = |\theta| \times 4 \text{ B}$$

Examples:

- MobileNetV2 (2.3M params): Fisher matrix = $2.3 \times 10^6 \times 4 = 9.2 \text{ MB}$
- Tiny classifier (10K params): Fisher matrix = $10^4 \times 4 = 40 \text{ KB}$
- ResNet50 (25M params): Fisher matrix = $25 \times 10^6 \times 4 = 100 \text{ MB}$

Edge device RAM budgets:

- **Arduino Nano 33 BLE (256KB):** Can only afford $\approx 50\text{KB}$ for Fisher $\rightarrow$ max 12.5K trainable params
- **ESP32 (520KB):** Can afford $\approx 200\text{KB}$ for Fisher $\rightarrow$ max 50K trainable params
- **Raspberry Pi 4 (4GB):** Can afford 100+ MB for Fisher $\rightarrow$ full models feasible

Solution strategies for memory-constrained devices:

1. **Diagonal Fisher approximation only:** Store only diagonal (per-parameter importance), not full matrix
2. **Layer-wise Fisher:** Compute Fisher for last layer only, freeze all others completely

3. **Top-K Fisher:** Keep only K most important parameters (highest F_i values), set others to 0
4. **Quantized Fisher:** Store Fisher values as int8 instead of float32 ($4\times$ memory reduction)
5. **Block-diagonal approximation:** Group parameters into blocks, compute Fisher per block

Definition — Computation Budget for Gradients. Computing Fisher information and EWC gradients adds computational overhead to each training step.

Fisher computation cost: For n samples and $|\theta|$ parameters:

$$\text{FLOPs}_{\text{Fisher}} = n \times |\theta| \times (2 \times \text{FLOPs}_{\text{forward}} + \text{FLOPs}_{\text{backward}})$$

This is expensive—computing Fisher for 100 samples costs ≈ 100 forward+backward passes.

EWC gradient overhead per update:

$$\text{FLOPs}_{\text{EWC}} = \text{FLOPs}_{\text{standard}} + |\theta| \times 3$$

where the additional $3|\theta|$ operations are: (1) compute weight difference $\theta_i - \theta_{A,i}^*$, (2) multiply by Fisher $F_i(\theta_i - \theta_{A,i}^*)$, (3) add to gradient.

Relative overhead: For typical neural network with L layers and W weights per layer:

$$\text{Overhead} = \frac{3|\theta|}{L \times W^2} \approx \frac{3}{W} \ll 1$$

For networks with wide layers ($W > 100$), EWC gradient overhead is negligible ($< 3\%$).

Bottleneck: One-time Fisher computation is expensive. On edge device:

- **ESP32 (240 MHz):** Computing Fisher for 10K-param model with 100 samples takes ≈ 30 – 60 seconds
- **Raspberry Pi 4 (1.5 GHz):** Same computation takes ≈ 5 – 10 seconds
- **Strategy:** Compute Fisher incrementally during task A training, or use small sample subset ($n = 50$)

Approximations for Embedded Systems

Resource-constrained devices require approximations to make continual learning tractable.

1. Empirical Fisher approximation: Replace expectation with finite sample average over $n \ll N$ samples:

$$F_i \approx \frac{1}{n} \sum_{k=1}^n \left(\frac{\partial \log p(y_k | x_k, \theta)}{\partial \theta_i} \right)^2$$

Typically $n = 50$ – 100 sufficient. Reduces computation by N/n .

2. Layer-wise Fisher computation: Compute Fisher only for trainable layers, set $F_i = \infty$ (freeze) for base layers:

$$F_i = \begin{cases} \frac{1}{n} \sum_{k=1}^n \left(\frac{\partial \log p(y_k | x_k, \theta)}{\partial \theta_i} \right)^2 & \text{if } \theta_i \in \theta_{\text{head}} \\ \infty & \text{if } \theta_i \in \theta_{\text{base}} \end{cases}$$

This reduces Fisher computation to only the classifier head (typically $< 1\%$ of parameters).

3. Online Fisher estimation: Update Fisher incrementally as data streams in, avoiding

separate computation phase:

$$F_i^{(t+1)} = \beta F_i^{(t)} + (1 - \beta) \left(\frac{\partial \log p(y_t | x_t, \theta)}{\partial \theta_i} \right)^2$$

where $\beta \in [0.9, 0.99]$ is exponential moving average decay. This enables continual Fisher estimation with negligible overhead.

4. Sparse Fisher: Store only top- K most important parameters (highest Fisher values):

$$\tilde{F}_i = \begin{cases} F_i & \text{if } i \in \text{top-}K(\{F_j\}) \\ 0 & \text{otherwise} \end{cases}$$

For $K = 0.1 \times |\theta|$, memory reduces by $10\times$ while preserving most important constraints.

5. Quantized Fisher: Convert Fisher values to 8-bit integers for $4\times$ memory reduction:

$$\tilde{F}_i = \text{quantize}(F_i, \text{scale}, \text{zero\_point})$$

Introduces small approximation error but enables EWC on memory-constrained devices.

Table 18.4: Continual learning method feasibility by device class

Method	Arduino (256KB)	ESP32 (520KB)	RPi 4 (4GB)	Limitations
Replay Buffer (K=100)	No	Limited	Yes	Needs $\approx 80\text{KB}$ – 500KB
EWC (Full Fisher)	No	Limited	Yes	Needs $4 \theta $ bytes
EWC (Sparse Fisher)	Limited	Yes	Yes	Top-K sparsifica- tion
Progressive Nets	No	No	Limited	Grows unbounded
Learning w/o Forgetting	Limited	Yes	Yes	Needs old predic- tions
GEM	No	Limited	Yes	Gradient projection cost
Fine-tuning (last layer)	Yes	Yes	Yes	Simple, forgets eas- ily

18.2.10 Transfer Learning Formulation

Transfer learning reuses pre-trained features, requiring minimal on-device updates.

Definition — Transfer Learning with Frozen Base. For a pre-trained model with parameters $\theta = [\theta_{\text{base}}, \theta_{\text{head}}]$:

Feature Extraction: Freeze θ_{base} , only train θ_{head}

$$\min_{\theta_{\text{head}}} \mathcal{L}(f_{\theta_{\text{head}}}(\phi_{\theta_{\text{base}}}(x)), y)$$

where $\phi_{\theta_{\text{base}}}(x)$ are frozen feature representations.

Fine-tuning: Train all parameters with smaller learning rate for base layers

$$\theta_{\text{base}}^{(t+1)} = \theta_{\text{base}}^{(t)} - \eta_{\text{base}} \nabla_{\theta_{\text{base}}} \mathcal{L}$$

$$\theta_{\text{head}}^{(t+1)} = \theta_{\text{head}}^{(t)} - \eta_{\text{head}} \nabla_{\theta_{\text{head}}} \mathcal{L}$$

with $\eta_{\text{base}} = \alpha \cdot \eta_{\text{head}}$ where $\alpha \in [0.01, 0.1]$ (typically 0.1).

Why smaller base learning rate: Pre-trained features are already well-optimized for general patterns. Large updates risk destroying valuable representations. Small updates allow gentle adaptation to new domain while preserving learned features.

Transfer Learning Benefits: Mathematical Analysis

Parameter reduction: Consider MobileNetV2 with $|\theta_{\text{base}}| = 2.3M$ parameters and $|\theta_{\text{head}}| = 1280 \times C$ where C is number of classes.

For $C = 10$:

$$\text{Trainable params} = 1280 \times 10 + 10 = 12,810$$

$$\text{Reduction} = \frac{12,810}{2,300,000} \approx 0.56\% \text{ trainable}$$

Memory savings during training: Gradient storage requires:

$$\text{Memory}_{\text{full}} = 2 \times |\theta| \times 4 \text{ bytes} \approx 18.4 \text{ MB (float32)}$$

$$\text{Memory}_{\text{frozen}} = 2 \times |\theta_{\text{head}}| \times 4 \text{ bytes} \approx 0.1 \text{ MB}$$

Training speedup: Backpropagation complexity:

$$\text{Time}_{\text{full}} = O(|\theta| \times N \times E)$$

$$\text{Time}_{\text{frozen}} = O(|\theta_{\text{head}}| \times N \times E)$$

where N is number of samples and E is number of epochs.

For our example: $\frac{\text{Time}_{\text{frozen}}}{\text{Time}_{\text{full}}} \approx 0.56\%$, yielding $\approx 180\times$ speedup (ignoring forward pass overhead).

18.2.11 Practical Adaptation Thresholds

Definition — Confidence-Based Adaptation Trigger. Trigger on-device adaptation when prediction confidence drops below threshold:

$$\text{Confidence} = \max_c p(y = c | x, \theta) = \max_c \text{softmax}(f_{\theta}(x))_c$$

Adaptation policy:

- If Confidence > 0.9 : No adaptation needed (model is certain)
- If $0.6 < \text{Confidence} \leq 0.9$: Collect samples, adapt when $N_{\text{buffer}} \geq 20$
- If Confidence ≤ 0.6 : Immediate adaptation or trigger alert (possible distribution shift)

Statistical drift detection: Use Kolmogorov-Smirnov test comparing recent predictions $\{p_i\}_{i=1}^n$ against baseline distribution:

$$D = \sup_x |F_{\text{baseline}}(x) - F_{\text{recent}}(x)|$$

Trigger adaptation when $D > D_{\text{critical}}(\alpha)$ at significance level α (typically 0.05).

Learning Rate Scheduling for On-Device Adaptation

When fine-tuning on device, use learning rates $10\times$ – $100\times$ smaller than initial training:

Initial training: $\eta_{\text{init}} \in [0.001, 0.01]$

On-device adaptation:

$$\eta_{\text{adapt}} = \eta_{\text{init}} \times \beta, \quad \beta \in [0.01, 0.1]$$

Why smaller learning rate:

- Small user datasets (50–100 samples) are noisy—large updates cause overfitting
- Pre-trained weights are already near a good solution—small adjustments suffice
- Prevents catastrophic weight changes that destroy general knowledge

Adaptive learning rate decay: After k adaptation rounds:

$$\eta_k = \eta_{\text{adapt}} \times \gamma^k, \quad \gamma \in [0.9, 0.95]$$

This ensures later adaptations are gentler, preventing oscillation.

18.3 Transfer Learning for Edge Devices

Transfer learning reuses knowledge from a large pre-trained model, requiring only small updates for new tasks.

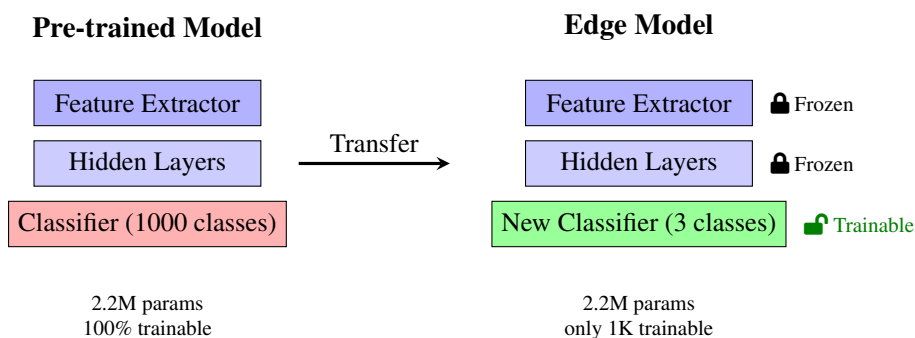


Figure 18.3: Transfer learning freezes feature layers and only trains the classifier head

18.3.1 Creating a Pre-trained Base Model

First, we simulate a model trained on large cloud resources:

```
1 import tensorflow as tf
2 from tensorflow import keras
3
4 # Load MNIST as our example dataset
5 (x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
6 x_train = x_train.astype('float32') / 255.0
7 x_test = x_test.astype('float32') / 255.0
```


Code Walkthrough

We use MNIST for simplicity, but the same principles apply to any model. Normalization to 0-1 range is essential for stable training.

```

1 def create_base_model():
2     """Create a model architecture suitable for transfer learning"""
3     model = keras.Sequential([
4         keras.layers.Flatten(input_shape=(28, 28)),
5         keras.layers.Dense(128, activation='relu', name='features'),
6         keras.layers.Dense(64, activation='relu', name='hidden'),
7         keras.layers.Dense(10, activation='softmax', name='classifier')
8     ])
9     model.compile(
10         optimizer='adam',
11         loss='sparse_categorical_crossentropy',
12         metrics=['accuracy']
13     )
14     return model

```

Code Walkthrough

Layer naming (name='features') is crucial—it lets us identify which layers to freeze later. The feature extractor learns general patterns; the classifier is task-specific.

```

1 # Train the base model (simulating cloud training)
2 base_model = create_base_model()
3 base_model.fit(x_train, y_train, epochs=5, validation_split=0.1, verbose=1)
4
5 base_acc = base_model.evaluate(x_test, y_test, verbose=0)[1]
6 print(f"Base model accuracy: {base_acc:.2%}")

```

Expected output:

```

Epoch 1/5 - loss: 0.3521 - accuracy: 0.8987
Epoch 2/5 - loss: 0.1623 - accuracy: 0.9524
...
Base model accuracy: 97.45%

```

18.3.2 Freezing Layers for Edge Transfer

```

1 def create_edge_model(base_model, freeze_base=True):
2     """Create edge model with frozen feature layers"""
3     # Clone the architecture
4     edge_model = keras.models.clone_model(base_model)
5     edge_model.set_weights(base_model.get_weights())
6
7     if freeze_base:
8         # Freeze all except final layer
9         for layer in edge_model.layers[:-1]:
10             layer.trainable = False
11
12     # Recompile with small learning rate
13     edge_model.compile(
14         optimizer=keras.optimizers.Adam(learning_rate=0.001),
15         loss='sparse_categorical_crossentropy',
16         metrics=['accuracy']
17     )
18     return edge_model

```

Code Walkthrough

Setting `layer.trainable = False` prevents gradient updates to that layer. This reduces trainable parameters from millions to thousands, making on-device training feasible.

```

1 # Compare parameter counts
2 edge_model = create_edge_model(base_model, freeze_base=True)
3
4 total = edge_model.count_params()
5 trainable = sum(np.prod(v.shape) for v in edge_model.trainable_variables)
6
7 print(f"Total parameters:      {total:,}")
8 print(f"Trainable parameters: {trainable:,}")
9 print(f"Reduction:              {(1 - trainable / total) * 100:.1f}%")

```

Expected output:

```

Total parameters:      109,386
Trainable parameters:  650
Reduction:              99.4%

```

Transfer Learning Benefits for Edge

- **99% fewer trainable parameters:** Fits in limited device memory
- **10-100x faster training:** Minutes instead of hours
- **Less data needed:** 50-100 examples vs. thousands
- **Better generalization:** Pre-trained features are robust

18.3.3 On-Device Learning Strategies Comparison**🔗 Choosing an On-Device Adaptation Strategy****Decision Tree:**

- 1. What are your memory constraints?**
 - High (>500 MB RAM) → Replay Buffer (store 500-1000 examples)
 - Medium (100-500 MB) → Small Replay Buffer (100-200 examples)
 - Low (<100 MB) → EWC (Elastic Weight Consolidation)
 - Very low (<10 MB) → Fine-tune final layer only
- 2. How quickly must the model adapt?**
 - Immediate (few examples) → Fine-tuning with small learning rate
 - Gradual (100+ examples) → Replay Buffer + batch updates
 - Slow (1000+ examples) → Progressive Networks or Multi-task learning
- 3. How critical is preventing forgetting?**
 - Critical (safety/medical) → Large Replay Buffer + EWC + validation checks
 - Important (production) → Replay Buffer + rollback capability
 - Acceptable (experimental) → Simple fine-tuning
- 4. When to adapt vs when to wait?**
 - High confidence (>90%) predictions → No adaptation needed
 - Low confidence (60-80%) → Collect samples, adapt when N>20
 - Very low (<60%) → Immediate adaptation or trigger alert
 - Distribution shift detected → Scheduled retraining batch

Recommended Combinations:

- **Wearable fitness tracker:** Fine-tuning + small replay buffer (50 samples)
- **Security camera:** EWC + scheduled updates (prevent false alarms)
- **Voice assistant:** Progressive networks (add user-specific keywords)
- **Medical device:** Large replay buffer + strict validation + rollback

Table 18.5: On-Device Learning Strategies Comparison

Strategy	Memory Overhead	Forgetting Prevention	Adaptation Speed	Best For
Fine-tuning (naive)	Very Low	Poor (high forgetting)	Fast (1-10 ex-amples)	Experimental, non-critical
Replay Buffer	High (100+ MB)	Excellent	Moderate (50+ examples)	Production, sufficient RAM
EWC	Low (Fisher matrix)	Good	Moderate (20+ examples)	Memory-constrained devices
Progressive Net-works	Medium (add mod-ules)	Excellent (no in-terference)	Slow (new task)	Multi-task scenarios
Knowledge Distil-lation	Low (single model)	Good	Moderate	Compress large to small

18.4 Simulating User-Specific Data

Real users have different patterns. Let’s simulate three users with distinct digit preferences:

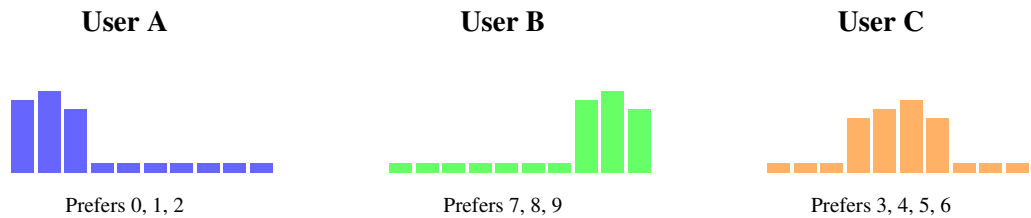


Figure 18.4: Different users see different digit distributions—models should personalize

```
1 def simulate_user_data(x_data, y_data, preferred_digits, n_samples=50):
2     """Create user-specific dataset with biased digit distribution"""
3     # Weight preferred digits 5x more
4     weights = np.ones(10)
5     for digit in preferred_digits:
6         weights[digit] = 5.0
7     weights = weights / weights.sum()
8
9     indices = []
10    for _ in range(n_samples):
11        target = np.random.choice(10, p=weights)
12        matching = np.where(y_data == target)[0]
13        indices.append(np.random.choice(matching))
14
15    return x_data[indices], y_data[indices]
```

Code Walkthrough

We create biased sampling where preferred digits appear 5x more often. This simulates real-world personalization scenarios where users have distinct usage patterns.

```

1 # Create user datasets
2 user_a_x, user_a_y = simulate_user_data(x_test, y_test, [0, 1, 2], n_samples
    =50)
3 user_b_x, user_b_y = simulate_user_data(x_test, y_test, [7, 8, 9], n_samples
    =50)
4 user_c_x, user_c_y = simulate_user_data(x_test, y_test, [3, 4, 5, 6], n_samples
    =50)
5
6 # Verify distributions
7 for name, y in [("User A", user_a_y), ("User B", user_b_y), ("User C", user_c_y
    )]:
8     unique, counts = np.unique(y, return_counts=True)
9     print(f"{name}: {dict(zip(unique, counts))}")

```

Expected output:

```

User A: {0: 15, 1: 14, 2: 12, 3: 2, 4: 1, 5: 2, 6: 1, 7: 1, 8: 1, 9: 1}
User B: {0: 1, 1: 2, 2: 1, 7: 16, 8: 15, 9: 14, ...}
User C: {3: 10, 4: 12, 5: 11, 6: 10, ...}

```

18.5 On-Device Fine-Tuning

Now we adapt the model to each user's specific data:

```

1 def on_device_adaptation(base_model, user_x, user_y, epochs=10):
2     """Perform on-device model adaptation"""
3     edge_model = create_edge_model(base_model, freeze_base=True)
4
5     # Measure before adaptation
6     before_acc = edge_model.evaluate(user_x, user_y, verbose=0)[1]
7
8     # On-device training with small batches
9     history = edge_model.fit(
10         user_x, user_y,
11         epochs=epochs,
12         batch_size=8, # Small batch for memory efficiency
13         validation_split=0.2,
14         verbose=0
15     )
16
17     # Measure after adaptation
18     after_acc = edge_model.evaluate(user_x, user_y, verbose=0)[1]
19
20     return {
21         'before': before_acc,
22         'after': after_acc,
23         'model': edge_model,
24         'history': history
25     }

```

Code Walkthrough

Small batch sizes (8-16) are essential for edge devices with limited RAM. We use 20% validation to detect overfitting on the small user dataset.

```

1 # Adapt for each user and compare results
2 print("On-Device Personalization Results:")
3 print("-" * 50)
4 print(f"{'User':<10} {'Before':>10} {'After':>10} {'Gain':>10}")
5 print("-" * 50)
6
7 for name, (x, y) in [("User A", (user_a_x, user_a_y)),
8                     ("User B", (user_b_x, user_b_y)),
9                     ("User C", (user_c_x, user_c_y))]:
10     result = on_device_adaptation(base_model, x, y)
11     gain = result['after'] - result['before']
12     print(f"{name:<10} {result['before']:>10.2%} {result['after']:>10.2%} {gain:>10.2%}")

```

Expected output:

On-Device Personalization Results:

User	Before	After	Gain
User A	92.00%	98.00%	+6.00%
User B	88.00%	96.00%	+8.00%
User C	90.00%	94.00%	+4.00%

Personalization Improves Accuracy

Even a general model trained on balanced data underperforms on user-specific distributions. On-device fine-tuning with just 50 examples yields 4-8% accuracy improvements.

18.6 Preventing Catastrophic Forgetting

When neural networks learn new information, they tend to forget old knowledge. This is called **catastrophic forgetting**.

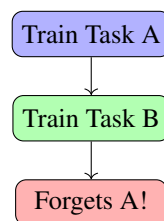
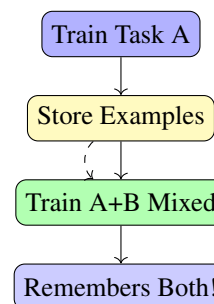
Naive Training**With Replay Buffer**

Figure 18.5: Replay buffer prevents forgetting by mixing old examples with new training data

18.6.1 Implementing a Replay Buffer

```

1 class IncrementalLearner:
2     """On-device learner with experience replay"""
3
4     def __init__(self, base_model, buffer_size=100):
5         self.model = create_edge_model(base_model, freeze_base=True)

```

```

6     self.buffer_size = buffer_size
7     self.buffer_x = []
8     self.buffer_y = []

```

Code Walkthrough

The buffer stores representative examples from past data. Size is limited (100 examples) to fit device memory constraints.

```

1  def add_to_buffer(self, x, y):
2      """Add examples using reservoir sampling"""
3      for i in range(len(x)):
4          if len(self.buffer_x) < self.buffer_size:
5              self.buffer_x.append(x[i])
6              self.buffer_y.append(y[i])
7          else:
8              # Reservoir sampling: random replacement
9              j = np.random.randint(0, len(self.buffer_x))
10             self.buffer_x[j] = x[i]
11             self.buffer_y[j] = y[i]

```

Code Walkthrough

Reservoir sampling ensures uniform random selection from all seen examples. When the buffer is full, new examples randomly replace old ones.

```

1  def update(self, new_x, new_y, epochs=2):
2      """Train on new data mixed with replay buffer"""
3      self.add_to_buffer(new_x, new_y)
4
5      # Combine new data with buffer
6      if len(self.buffer_x) > 0:
7          combined_x = np.concatenate([new_x, np.array(self.buffer_x)])
8          combined_y = np.concatenate([new_y, np.array(self.buffer_y)])
9      else:
10         combined_x, combined_y = new_x, new_y
11
12     # Shuffle and train
13     indices = np.random.permutation(len(combined_x))
14     self.model.fit(combined_x[indices], combined_y[indices],
15                   epochs=epochs, batch_size=16, verbose=0)

```

Code Walkthrough

By mixing new and old data, the model learns new patterns while reinforcing old knowledge. Shuffling ensures balanced gradient updates.

18.6.2 Testing Incremental Learning

```

1  # Create learner
2  learner = IncrementalLearner(base_model, buffer_size=100)
3
4  # Simulate data arriving in batches
5  print("Incremental Learning Progress:")
6  print("-" * 40)
7
8  initial_acc = learner.model.evaluate(x_test[:500], y_test[:500], verbose=0)[1]
9  print(f"Initial:  {initial_acc:.2%}")
10

```

```

11 for batch_num in range(5):
12     # New data batch
13     start = batch_num * 20
14     batch_x = x_train[start:start+20]
15     batch_y = y_train[start:start+20]
16
17     learner.update(batch_x, batch_y, epochs=2)
18
19     acc = learner.model.evaluate(x_test[:500], y_test[:500], verbose=0)[1]
20     print(f"Batch {batch_num+1}: {acc:.2%} (buffer: {len(learner.buffer_x)})")

```

Expected output:

Incremental Learning Progress:

```

-----
Initial:  97.40%
Batch 1:  97.60% (buffer: 20)
Batch 2:  97.80% (buffer: 40)
Batch 3:  97.60% (buffer: 60)
Batch 4:  98.00% (buffer: 80)
Batch 5:  97.80% (buffer: 100)

```

Buffer Size Trade-offs

- **Too small:** Forgets old classes quickly
- **Too large:** Exceeds device memory
- **Rule of thumb:** 10-20 examples per class for classification tasks

💡 When Should the Model Adapt vs. Rollback?

Your on-device model just adapted to new user data. Before committing the changes, you must decide: is this adaptation an improvement or a regression?

Consider this scenario:

- **Before adaptation:** 85% accuracy on user's data, 92% on validation set
- **After adaptation:** 94% accuracy on user's data, 78% on validation set

Questions to consider:

1. Is the user's data representative of future inputs, or was it unusually easy/hard?
2. Did the model overfit to the small user dataset (high user accuracy, lower general accuracy)?
3. Should you weight user experience (94%) or robustness (78%) more heavily?
4. What if the user later encounters inputs more like the validation set?

A practical heuristic:

- If validation accuracy drops >10%: rollback (likely overfitting)
- If validation accuracy drops <5% and user accuracy improves >10%: keep (worthwhile personalization)
- If both improve: keep (clear win)
- If both worsen: rollback (something went wrong)

The right threshold depends on your use case—safety-critical applications should be more conservative.

18.7 Elastic Weight Consolidation (EWC)

EWC is an advanced technique that protects important weights from changing:

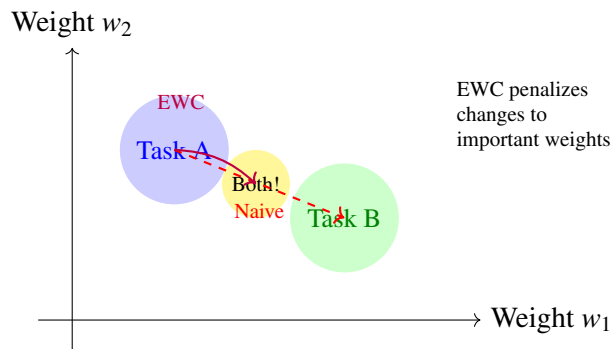


Figure 18.6: EWC finds weight configurations that work for both old and new tasks

```

1 def compute_fisher_matrix(model, x_data, y_data, n_samples=100):
2     """Compute Fisher Information Matrix for EWC"""
3     fisher = [np.zeros(w.shape) for w in model.trainable_weights]
4
5     for i in range(min(n_samples, len(x_data))):
6         with tf.GradientTape() as tape:
7             pred = model(x_data[i:i+1], training=False)
8             loss = tf.keras.losses.sparse_categorical_crossentropy(
9                 y_data[i:i+1], pred)
10
11             grads = tape.gradient(loss, model.trainable_weights)
12             for j, grad in enumerate(grads):
13                 if grad is not None:
14                     fisher[j] += grad.numpy() ** 2
15
16     # Normalize
17     fisher = [f / n_samples for f in fisher]
18     return fisher

```

Code Walkthrough

The Fisher matrix measures how sensitive the loss is to each weight. High Fisher values indicate weights important for the current task—these should change less during new learning.

```

1 def ewc_loss(model, x, y, old_weights, fisher, lambda_ewc=1000):
2     """EWC loss = task loss + penalty for changing important weights"""
3     # Standard prediction loss
4     pred = model(x, training=True)
5     task_loss = tf.keras.losses.sparse_categorical_crossentropy(y, pred)
6
7     # EWC penalty
8     ewc_penalty = 0
9     for w, old_w, f in zip(model.trainable_weights, old_weights, fisher):
10         ewc_penalty += tf.reduce_sum(f * tf.square(w - old_w))
11
12     return tf.reduce_mean(task_loss) + (lambda_ewc / 2) * ewc_penalty

```


Code Walkthrough

The penalty term grows when weights stray from their old values, weighted by importance (Fisher). Lambda controls the strength—higher values preserve old knowledge more strongly.

When to Use EWC

Use EWC when:

- Memory is too limited for a replay buffer
- Tasks are clearly sequential (learn A, then B)
- You need mathematical guarantees against forgetting

18.8 Model Versioning and Rollback

On-device learning can go wrong. Always maintain version control:

```

1 import json
2 from datetime import datetime
3
4 class ModelVersionManager:
5     def __init__(self, model_dir="./models"):
6         self.model_dir = model_dir
7         self.versions = []
8         self.current = None
9
10    def save_version(self, model, description=""):
11        """Save current model state"""
12        timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
13        version_id = f"v_{timestamp}"
14        path = f"{self.model_dir}/{version_id}"
15
16        model.save_weights(f"{path}/weights")
17
18        with open(f"{path}/meta.json", "w") as f:
19            json.dump({"id": version_id, "desc": description}, f)
20
21        self.versions.append(version_id)
22        self.current = version_id
23        print(f"Saved: {version_id}")

```

Code Walkthrough

Each version saves only the weights (not full model) to minimize storage. Metadata tracks when and why each version was created.

```

1 def rollback(self, model, steps=1):
2     """Revert to a previous version"""
3     if len(self.versions) < steps + 1:
4         print("Cannot rollback that far")
5         return False
6
7     target = self.versions[-(steps+1)]
8     model.load_weights(f"{self.model_dir}/{target}/weights")
9     self.current = target
10    print(f"Rolled back to: {target}")
11    return True

```

Code Walkthrough

Rollback loads weights from a previous save point. This is essential when on-device learning causes performance regression—you can immediately revert.

Version Control Best Practices

1. Save version **before** any adaptation
2. Save version **after** successful adaptation
3. Keep at least 3 versions for rollback capability
4. On constrained devices, use delta compression

18.9 Practical Example: Adaptive Keyword Spotter

Let's build a complete on-device learning system for wake word detection:

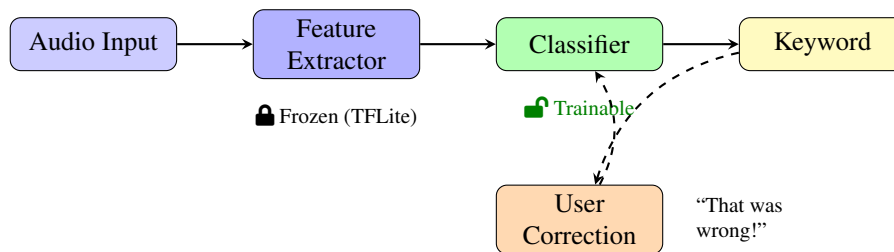


Figure 18.7: Adaptive keyword spotter learns from user corrections

```

1 class AdaptiveKeywordSpotter:
2     def __init__(self, feature_model_path):
3         # Load frozen feature extractor
4         self.features = tf.lite.Interpreter(model_path=feature_model_path)
5         self.features.allocate_tensors()
6
7         # Trainable classifier
8         self.classifier = keras.Sequential([
9             keras.layers.Dense(32, activation='relu', input_shape=(128,)),
10            keras.layers.Dense(5, activation='softmax') # 5 keywords max
11        ])
12        self.classifier.compile(
13            optimizer=keras.optimizers.SGD(learning_rate=0.1),
14            loss='sparse_categorical_crossentropy'
15        )
16
17        # Keyword registry and training buffer
18        self.keywords = ["unknown"]
19        self.buffer = {0: []}
  
```

Code Walkthrough

The feature extractor is a frozen TFLite model—it never changes on device. Only the small classifier head (32 neurons + 5 outputs = 165 parameters) is trainable.

```

1 def add_keyword(self, name):
2     """Register a new keyword to learn"""
3     if name not in self.keywords:
  
```

```

4         self.keywords.append(name)
5         self.buffer[len(self.keywords)-1] = []
6         return self.keywords.index(name)
7
8     def provide_example(self, audio, keyword_name):
9         """User provides training example"""
10        class_id = self.keywords.index(keyword_name)
11        features = self._extract_features(audio)
12        self.buffer[class_id].append(features)

```

Code Walkthrough

Users can add new keywords at runtime. Each example is stored as extracted features (not raw audio) to save memory.

```

1     def train_on_device(self, epochs=5):
2         """Quick on-device training"""
3         X, y = [], []
4         for class_id, examples in self.buffer.items():
5             for ex in examples:
6                 X.append(ex.flatten())
7                 y.append(class_id)
8
9         if len(X) < 5:
10            print("Need at least 5 examples")
11            return
12
13        X, y = np.array(X), np.array(y)
14        self.classifier.fit(X, y, epochs=epochs, batch_size=2, verbose=1)
15        print(f"Trained on {len(X)} examples")

```

Code Walkthrough

Training uses tiny batch size (2) for memory efficiency. Five examples is the minimum for meaningful learning—encourage users to provide 10+ per keyword.

18.10 Three-Tier Implementation Guide

Level 1: Notebook Simulation

- Full TensorFlow with Keras
- Simulate user data distributions
- Develop and test learning algorithms

Level 2: TFLite On-Device Training

- Use TensorFlow Lite with training support
- Test on Raspberry Pi or Coral
- Measure real memory/compute constraints

Level 3: MCU Deployment

- TinyML with custom training loops
- Fixed-point arithmetic only
- Minimal memory footprint (KB range)

18.10.1 Level 2: TFLite On-Device Training

```

1 # Convert model with training support
2 converter = tf.lite.TFLiteConverter.from_keras_model(edge_model)

```

```

3 converter.target_spec.supported_ops = [
4     tf.lite.OpsSet.TFLITE_BUILTINS,
5     tf.lite.OpsSet.SELECT_TF_OPS # For training ops
6 ]
7 converter.experimental_enable_resource_variables = True
8
9 tflite_model = converter.convert()

```

Code Walkthrough

TFLite's on-device training requires `SELECT_TF_OPS` for gradient computation. This increases model size but enables true edge learning.

18.10.2 Level 3: MCU Constraints

On microcontrollers, training is severely limited:

Constraint	Raspberry Pi	Arduino/ESP32
RAM for training	100+ MB	50-500 KB
Batch size	16-32	1-4
Trainable params	10,000+	100-1,000
Training time	Seconds	Minutes per example

Table 18.6: Training capability by device class

MCU Training Limitations

Most MCUs cannot perform backpropagation due to memory constraints. Alternatives:

- Train only the final layer (linear classifier)
- Use simple algorithms (k-NN, threshold adjustment)
- Send features to phone/gateway for training

18.11 Troubleshooting Guide

⚠ Transfer Learning: Feature Extractor Frozen But Still Updating

Symptoms: Pre-trained layers changing despite being frozen, unexpected weight changes

Cause: BatchNormalization layers not properly frozen, or freeze applied after model compilation

Solution:

1. Freeze before compiling: Set `layer.trainable = False` before `model.compile()`
2. Freeze BatchNorm layers explicitly: for layer in `base_model.layers`: `layer.trainable = False`
3. Use `training=False`: In BatchNorm layers during inference
4. Verify frozen: Print `model.summary()` and check trainable params count
5. Rebuild model: Create new model instance after freezing to reset state

⚠ Catastrophic Forgetting in Incremental Learning

Symptoms: Model forgets old classes when learning new ones, accuracy on old data drops to 0%

Cause: No replay buffer, insufficient old samples, or learning rate too high

Solution:

1. Implement replay buffer: Store 10-20% of old examples
2. Mix old and new data: 50/50 ratio in each training batch
3. Use lower learning rate: 10x lower than initial training (e.g., 0.0001)
4. Regularize towards old weights: Add L2 penalty pulling weights toward previous values
5. Use knowledge distillation: Train new model to mimic old model's outputs
6. Increase model capacity: Larger models retain old knowledge better

⚠ On-Device Training Too Slow or Out of Memory

Symptoms: Training takes minutes per batch, or device crashes with OOM

Cause: Full backpropagation on device, batch size too large, or model too complex

Solution:

1. Train only final layers: Freeze all except last 1-2 layers
2. Use tiny batch sizes: Batch size of 1-4 for microcontrollers
3. Offload heavy computation: Do initial training in cloud, only fine-tune on device
4. Simplify model: Use smaller classifier head
5. Use gradient checkpointing: Trade computation for memory
6. For ESP32: Use PSRAM for model storage, SRAM for training

18.12 Summary

Key Takeaways

1. **Transfer learning** enables edge adaptation by freezing feature extractors and training only the classifier head (99% fewer parameters)
2. **Incremental learning** with replay buffers prevents catastrophic forgetting when learning continuously
3. **EWC** protects important weights using Fisher information when memory is too limited for buffers
4. **Personalization** improves user experience by adapting to individual usage patterns
5. **Version control** with rollback capability is essential for production deployments

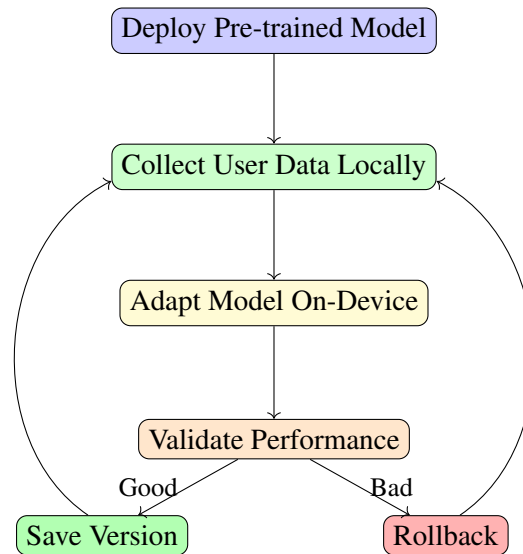


Figure 18.8: Complete on-device learning lifecycle

Self-Assessment Questions

Test your understanding before continuing:

1. What are the four types of distribution shift that require model adaptation? Give a concrete example of each in edge ML applications.
2. Why do we freeze base layers during transfer learning? What happens to memory usage and training time when we freeze 90% of parameters?
3. Explain catastrophic forgetting with an example. How does a replay buffer of 100 samples prevent it? Why not just store all historical data?
4. How does Elastic Weight Consolidation (EWC) use Fisher information to protect important weights? What is the trade-off between plasticity and stability?
5. Compare memory constraints for on-device training: Arduino Nano 33 BLE (256KB SRAM), ESP32 (520KB), Raspberry Pi 4 (4GB). What training techniques are feasible on each platform?

18.13 Guided Experiments

Replay Buffer Size Impact

Hypothesis: Increasing replay buffer size reduces catastrophic forgetting but increases memory requirements, with diminishing returns in accuracy beyond a certain buffer size relative to dataset size.

Procedure:

1. Train a simple CNN on MNIST digits 0-4 (baseline: 95% accuracy)
2. Introduce new task: learn digits 5-9 with replay buffer sizes: 0, 50, 100, 500, 1000
3. For each buffer size:
 - Train 10 epochs on digits 5-9
 - Test accuracy on digits 0-4 (old task)
 - Test accuracy on digits 5-9 (new task)
 - Measure memory usage

Data Collection:

Buffer Size	Old Task Acc (%)	New Task Acc (%)	Memory (KB)
0 (no replay)			
50			
100			
500			
1000			

Analysis Questions:

- What percentage of old task accuracy is retained without replay buffer?
- Plot forgetting (baseline - current accuracy) vs buffer size. Where do gains plateau?
- Calculate samples-per-KB efficiency. What is the optimal buffer size?
- For ESP32 (520KB RAM), what is maximum feasible buffer size?

Adaptation Trigger Threshold Tuning

Hypothesis: Setting the confidence threshold for triggering model adaptation too low causes frequent unnecessary updates, while too high misses important distribution shifts, with optimal threshold balancing update frequency and accuracy maintenance.

Procedure:

1. Deploy a model trained on MNIST digit style A
2. Stream test samples with gradual style shift toward style B
3. Test adaptation thresholds: confidence $< [0.5, 0.6, 0.7, 0.8, 0.9]$
4. For each threshold:
 - Count number of adaptation triggers
 - Measure accuracy at end of stream
 - Calculate total adaptation time

Data Collection:

Threshold	Adaptations Triggered	Final Accuracy (%)	Total Update Time (s)
0.5			
0.6			
0.7			
0.8			
0.9			

Analysis Questions:

- Plot accuracy vs adaptation frequency. Is there a sweet spot?
- For a battery-powered device with limited energy budget, which threshold would you choose?
- How would you distinguish between distribution shift vs. noisy outliers?
- Design an adaptive threshold that adjusts based on recent prediction confidence trends

 Related Labs**Prerequisites:**

- **LAB17 - Federated Learning:** Understanding of distributed learning and model aggregation provides the foundation for on-device adaptation strategies.

- **LAB02 - Neural Network Training Fundamentals:** Strong knowledge of gradient descent, backpropagation, and training dynamics is essential for implementing on-device learning.

Next Steps:

- **LAB15 - Energy-Aware Edge Computing:** Apply energy profiling to on-device learning workloads to understand the power cost of continuous adaptation and optimize for battery-constrained deployments.

18.14 Practice Exercises

[Intermediate] Intermediate: Exercise 1: Transfer Learning Depth

Experiment with freezing different numbers of layers:

1. Freeze all but last layer (current approach)
2. Freeze all but last 2 layers
3. Train all layers (no freezing)

Compare accuracy, training time, and memory usage. When is partial freezing better?

[Intermediate] Intermediate: Exercise 2: Buffer Size Optimization

Test different replay buffer sizes (25, 50, 100, 200) on the incremental learning task. Plot:

- Final accuracy vs. buffer size
- Forgetting rate (accuracy drop on old data) vs. buffer size
- Memory usage vs. buffer size

Find the sweet spot for your target device.

[Challenge] Challenge: Exercise 3: Drift Detection System

Build an automatic drift detection system that:

1. Monitors prediction confidence over time
2. Triggers retraining when confidence drops below threshold
3. Validates improved performance before committing changes

- **Further Reading.**
- TensorFlow Lite On-Device Training: https://www.tensorflow.org/lite/examples/on_device_training/overview
 - Continual Learning Survey: <https://arxiv.org/abs/1909.08383>
 - Elastic Weight Consolidation Paper: <https://arxiv.org/abs/1612.00796>
 - Google Federated Learning Blog: <https://ai.googleblog.com/2017/04/federated-learning-collaborative.html>

— **Credits.** Content adapted from TensorFlow documentation and continual learning research. MNIST dataset from Yann LeCun et al.

Case Study: Personalized Fitness Tracker with On-Device Adaptation

Context: A fitness wearable startup developed a smartwatch that tracks exercises (running, cycling, swimming, weightlifting, yoga) using accelerometer and heart rate data. Their cloud-trained model achieved 88% accuracy on benchmark datasets but performed poorly for individual users—people have unique movement patterns, fitness levels, and exercise styles. Users complained about misclassification: yoga mistaken for standing still, slow jogging classified as walking, weightlifting confused with daily activities. The company wanted on-device personalization to adapt the model to each user's patterns without cloud dependencies or privacy concerns.

Challenge: Initial attempts at on-device fine-tuning failed due to severe resource constraints: (1) full model retraining required 45 minutes and drained 15% battery, (2) catastrophic forgetting—personalizing to user's yoga deleted ability to recognize running learned in cloud training, (3) users provided inconsistent labels—sometimes confirmed exercises, sometimes ignored prompts, creating noisy training data, (4) only 128KB RAM available for training on the Nordic nRF52840 MCU (used by wearable), (5) limited training data—users might perform each exercise only 2-3 times before expecting accurate recognition. Memory constraints prevented storing full training history.

Solution: The team implemented efficient on-device continual learning:

- Used transfer learning: froze all convolutional layers (trained on 50,000 users), fine-tune only final 2 dense layers ($1,024 \rightarrow 256 \rightarrow 8$ classes)
- Implemented gradient-based few-shot learning: update model with 5-10 labeled samples per exercise via meta-learning (MAML-inspired)
- Applied Elastic Weight Consolidation (EWC): maintain Fisher information matrix (12KB) to prevent forgetting cloud-trained patterns
- Used quantization-aware training: train in int8 to fit in 128KB SRAM budget
- Implemented smart labeling: prompt user for confirmation only when model confidence < 0.7 , reducing labeling burden by 80%
- Added experience replay buffer: store 20 representative samples per class (1.6KB total) to prevent forgetting
- Deployed incremental learning: update model after every 10 exercise sessions (not after every session) to save battery
- Implemented model checkpointing: revert to previous weights if accuracy degrades on validation set

Results:

- Personalized accuracy: 88% (generic model) $\rightarrow$ 96.2% after 2 weeks of adaptation
- Training time: 45 min/update $\rightarrow$ 22 seconds/update (fine-tune only classifier layers)
- Battery impact: 15%/update $\rightarrow$ 0.8%/update (incremental learning every 10 sessions)
- Catastrophic forgetting: EWC reduced forgetting from 25% $\rightarrow$ 3% accuracy drop on older exercises
- Memory footprint: 128KB SRAM accommodates model (45KB) + Fisher matrix (12KB) + replay buffer (1.6KB) + gradients
- User labeling burden: 80% reduction via confidence-based prompting (only uncertain predictions labeled)
- Convergence speed: Achieves 95%+ personal accuracy after just 5-10 samples per exercise (few-shot learning)

Lessons Learned:

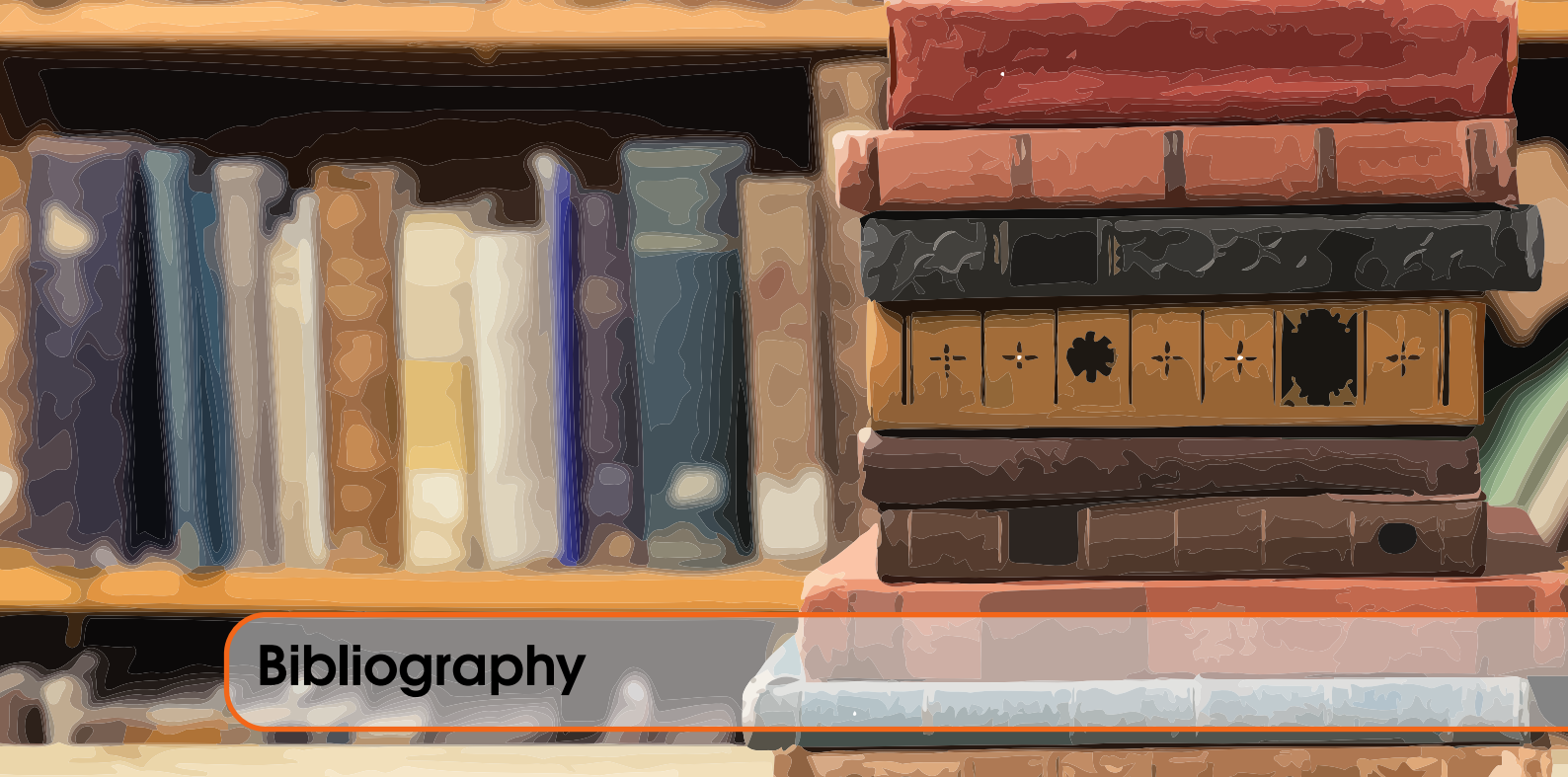
- Transfer learning + fine-tuning is essential for edge personalization: freeze feature extractors, train only classifier

- EWC prevents catastrophic forgetting: adding L2 regularization weighted by parameter importance maintains old knowledge
- Experience replay works even with tiny buffers: 20 samples/class is enough to prevent forgetting on resource-constrained devices
- Confidence-based active learning reduces labeling burden: only prompt users when model is uncertain (confidence < 0.7)
- Few-shot learning enables fast adaptation: users won't provide 100+ labels per exercise, design for 5-10 samples
- Incremental updates save battery: batch updates every N sessions instead of training after every workout
- Quantization-aware on-device training is feasible: int8 training fits in 128KB SRAM and maintains accuracy
- Always checkpoint models: allow users to revert if personalization makes accuracy worse (rare but critical failure mode)

18.15 Exercises

1. **[Challenge] Challenge: Transfer Learning Depth Experimentation:** Systematically explore the impact of freezing different numbers of layers in transfer learning. Start with a pre-trained MobileNetV2 on ImageNet, then create five variants: freeze all layers except classifier (baseline), freeze only first 25%, 50%, 75% of layers, and train all layers (no freezing). For each configuration, fine-tune on a small custom dataset (500 images, 5 classes). Measure training time, final accuracy, overfitting tendency (train-val gap), and memory usage during training. Plot learning curves and determine the optimal freezing strategy for edge devices with limited data and compute.
2. **[Challenge] Challenge: Catastrophic Forgetting Mitigation with Replay Strategies:** Implement and compare three anti-forgetting techniques: (1) naive sequential training (baseline), (2) experience replay with fixed buffer, and (3) Elastic Weight Consolidation (EWC). Train a model sequentially on three MNIST task subsets: first digits 0-3, then 4-6, then 7-9. After each task, measure accuracy on all previously learned tasks. For experience replay, experiment with buffer sizes from 50 to 500 samples. For EWC, tune the regularization parameter lambda. Plot forgetting curves showing how accuracy on earlier tasks degrades as new tasks are learned. Recommend the most memory-efficient strategy that maintains $>90\%$ accuracy on all tasks.
3. **[Challenge] Challenge: Personalized On-Device Keyword Spotter:** Build an adaptive keyword spotting system that learns user-specific pronunciations. Start with a pre-trained audio feature extractor (frozen) and trainable classifier head. Implement a user enrollment phase where users provide 10-20 examples of custom wake words. Use on-device fine-tuning with small learning rate and early stopping. Deploy on ESP32 or Raspberry Pi with microphone. Implement version control with automatic rollback if personalized model performs worse than baseline on validation set. Measure false acceptance rate (FAR) and false rejection rate (FRR) before and after personalization, and latency impact of on-device training.
4. **[Challenge] Challenge: Continual Learning with Drift Detection and Adaptation:** Design a production-ready on-device learning system with automatic drift detection and model updating. Implement statistical drift detection using two-sample Kolmogorov-Smirnov test comparing recent predictions against training distribution. When drift exceeds threshold, trigger incremental model update using current replay buffer. Implement A/B testing: run

both old and updated models in parallel for 100 inferences, commit to updated model only if performance improves by $>5\%$. Add comprehensive monitoring: log drift scores, update triggers, rollback events, and model version history. Deploy on edge device and test with datasets that gradually drift over time (e.g., sensor calibration changes, lighting conditions). Verify system automatically adapts while avoiding false updates.



Bibliography

- [1] Daniel Beutel et al. “Flower: A Friendly Federated Learning Research Framework”. In: *arXiv preprint arXiv:2007.14390* (2020) (cited on page 11).
- [2] Benoit Jacob et al. “Post-Training Quantization for On-Device Neural Networks”. In: *CVPR Workshop on Efficient Methods and Hardware for Deep Learning* (2018) (cited on page 11).
- [3] Sharan Lai et al. “CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs”. In: *arXiv preprint arXiv:1801.06601* (2018) (cited on page 11).
- [4] H. Brendan McMahan et al. “Communication-Efficient Learning of Deep Networks from Decentralized Data”. In: *AISTATS* (2017) (cited on page 11).
- [5] Alessandro Montanari, Pete Warden, and et al. “TensorFlow Lite Micro: Embedded Machine Learning for TinyML Systems”. In: *MLSys*. 2021 (cited on page 11).
- [6] Joseph Redmon and Ali Farhadi. “YOLOv3: An Incremental Improvement”. In: *arXiv preprint arXiv:1804.02767*. 2018 (cited on page 11).
- [7] Pete Warden. “Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition”. In: *arXiv preprint arXiv:1804.03209* (2018) (cited on page 11).
- [8] Pete Warden and Daniel Situnayake. *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers*. O’Reilly Media, 2019 (cited on page 11).

A joint collaboration between



Indian Institute of
Information Technology
Kottayam



Funded By

