Embedded Machine Learning

Using the Silicon Labs EFR32XG24 BLE Microcontroller

Masudul Imtiaz, PhD
Aaron Storey, MSAI
Ajan Ahmed

Wallace H. Coulter School of Engineering and Applied Sciences
Clarkson University — Potsdam, NY, USA

EE513 Intelligent System Design — Spring 2026

Table of contents

Preface

Embedded Machine Learning grew out of a concise laboratory handout for EE513 Intelligent System Design at Clarkson University. Over several semesters that handout expanded through feedback, additional experiments, and countless office-hour conversations into the more complete booklet you now hold. The text gathers the essential signal-processing concepts, annotated code examples, and step-by-step lab instructions required to move a lightweight machine-learning model from a Jupyter notebook onto a battery-powered EFR32XG24 Bluetooth Low Energy microcontroller.

This booklet exists because many introductions to TinyML either remain anchored on the desktop side of the workflow or plunge directly into embedded C without establishing a clear rationale. Here you will find a deliberate balance. Each chapter opens with a concise technical idea supported by an illustration and the minimum mathematics needed for clarity. The discussion then shifts to a practical exercise in which the reader compiles, flashes, and benchmarks real firmware on real hardware. Throughout the text memory maps, direct memory access transfers, and quantised inference are treated not as peripheral concerns but as fundamental design constraints.

The material suits senior undergraduates and early graduate students who have completed an introductory course in either signals or embedded programming. Prior exposure to TinyML is not assumed. Python notebooks walk through dataset preparation and model training, and CMake projects demonstrate deployment on the Cortex-M33 core that powers the EFR32XG24.

Chapters One and Two introduce audio fundamentals and feature extraction techniques such as spectrograms, mel-frequency cepstral coefficients, root-mean-square energy, and several spectral statistics. Chapters Three through Five explain classical machine-learning algorithms, model compression strategies, and the mechanics of on-device inference. Chapters Six through Eight present a complete keyword-spotting case study, including latency profiling on silicon. Appendices supply build scripts, linker maps, and a curated set of open-source utilities for further exploration.

Several people shaped this project. The EE513 cohorts of 2023 through 2025 stress-tested early drafts in weekly labs and offered candid suggestions. The Electrical and Computer Engineering technical team kept a steady rotation of development boards alive for student use. Masudul Imtiaz insisted on clarity in every explanation. Aaron Storey developed the automated benchmarking harness, and Ajan Ahmed integrated the LaTeX build with the Gecko SDK toolchain. Remaining errors belong to the authors alone.

Although every code fragment compiles on our machines, the booklet has not yet undergone formal peer review. Readers are invited to report issues or contribute improvements through the GitHub repository listed in Appendix B.

Potsdam, New York
July 4, 2025

Masudul Imtiaz Aaron Storey Ajan Ahmed

Embedded Machine Learning

Chapter 1

Introduction to Embedded Machine Learning

This chapter introduces the essential concepts of embedded machine learning and highlights the growing significance of TinyML in modern embedded system designs. It emphasizes the role of microcontrollers, particularly the Silicon Labs EFR32MG24, in enabling efficient, low-power machine learning inferencing for IoT applications. Whether you are a student starting your embedded ML journey or an engineer aiming to enhance your system design skills, this textbook will serve as a valuable resource to build innovative and efficient TinyML-enabled embedded solutions.

1.1 Overview

Embedded machine learning, often referred to as TinyML, represents a paradigm shift in computational intelligence by bringing sophisticated Inferencing capabilities directly to resource-constrained embedded systems. Unlike traditional machine learning systems that rely on cloud computing or powerful edge devices, TinyML optimizes models to operate within the strict memory, processing, and power constraints of microcontrollers. This evolution enables a new class of intelligent devices that can make real-time decisions locally without requiring constant connectivity to external servers.

Why now? A confluence of breakthroughs in model compression, compiler toolchains, and ultra-low-power silicon has made it possible to achieve <1,mW/inf inference on commodity MCUs[1, 2].

At the core of TinyML systems lies the microcontroller, a compact integrated circuit that combines a processor, memory, and input/output peripherals on a single chip. Modern microcontrollers like the Silicon Labs EFR32MG24 are increasingly designed with ML workloads in mind, featuring specialized hardware accelerators and optimized instructions sets that enhance neural network performance while maintaining energy efficiency.

In recent years, the demand for local intelligence in IoT devices has surged, driven by concerns about latency, privacy, bandwidth limitations, and power consumption. TinyML addresses these challenges by enabling machine learning models to run directly on microcontrollers, processing sensor data locally and making intelligent decisions without transmitting raw data to the cloud. This approach is valuable for applications such as keyword spotting, gesture recognition, anomaly detection, and predictive maintenance in industrial settings.

The Silicon Labs EFR32MG24 series is one of the most advanced microcontrollers available for TinyML applications in 2024. Built on the ARM Cortex-M33 core operating at 78 MHz, it offers a powerful blend of performance and energy efficiency, with a memory footprint of 1536KB flash and 256KB RAM. The chip includes an AI/ML hardware accelerator that enhances neural network execution, making it ideal for deploying sophisticated TinyML models while maintaining battery life in portable devices[3].

This textbook, *Embedded Machine Learning Design with Silicon Labs EFR32MG24*, provides a comprehensive guide for students and engineers to understand and implement TinyML solutions. The book covers both theoretical foundations and practical implementations, ensuring readers gain a deep understanding of machine learning optimization for resource-constrained systems.

Throughout this book, readers will learn:

- The fundamentals of TinyML and the computational constraints paradigm
- Model compression and quantization techniques for microcontroller deployment
- Practical implementation using Google Colab for model training and Simplicity Studio for deployment
- Hands-on experience starting with the canonical 'Hello World' of TinyML: a sine wave predictor
- Advanced techniques for power optimization, performance profiling, and model efficiency
- Real-world case studies demonstrating TinyML applications across various domains

1.2 Real-World Applications of Embedded Machine Learning

Embedded machine learning is transforming countless devices and technologies by enabling local intelligence in resource-constrained environments. Examples of TinyML applications can be observed across diverse industries, showcasing the versatility and transformative potential of this technology.

In healthcare and wearables, TinyML enables continuous health monitoring without draining battery life. Smart watches and fitness trackers use embedded ML algorithms to detect irregular heartbeats, analyze sleep patterns, and recognize specific activities based on motion sensor data. These devices perform complex pattern recognition locally, only transmitting alerts or summarized insights rather than constant streams of raw data, preserving both battery life and user privacy.

Industrial IoT applications leverage TinyML for predictive maintenance and anomaly detection at the sensor level. Embedded microcontrollers equipped with ML capabilities can analyze vibration patterns from motors or machinery, detecting subtle changes that might indicate impending failure before catastrophic breakdowns occur. By processing this data directly on the device, these systems can operate in environments with limited connectivity while providing real-time insights.

Consumer electronics increasingly incorporate TinyML to enhance user experience through always-on, low-power intelligence. Voice assistants use keyword spotting models running on microcontrollers to detect wake words without sending all audio to the cloud. Smart home sensors employ ML algorithms to differentiate between routine movements and security concerns, reducing false alarms while improving response times to genuine threats.

Agricultural and environmental monitoring systems utilize TinyML to enable intelligent, autonomous operation in remote locations. Soil moisture sensors can incorporate local ML models to optimize irrigation schedules based on weather patterns, soil conditions, and crop-specific needs. Wildlife tracking devices use embedded ML to classify animal behaviors directly on the device, extending battery life from days to months by eliminating continuous data transmission.

The EFR32MG24 microcontroller is particularly well-suited for these applications due to its balance of processing power, memory resources, and energy efficiency. Its ARM Cortex-M33 core provides sufficient computational capabilities for running inference on neural networks, while its power management features enable long-term operation on battery power. The integrated ML accelerator further enhances performance for specific machine learning workloads, enabling more complex models to run efficiently.

1.3 EFR32MG24 Microcontroller

The EFR32MG24 microcontroller from Silicon Labs' Wireless Gecko family is a practical option for running TinyML workloads in resource-constrained devices. An ARM Cortex-M33 core clocked at up to 78 MHz, paired with 1.5 MB of flash and 256 kB of SRAM, provides sufficient resources to store firmware alongside compact, quantized models for entry-level classification, regression, or detection

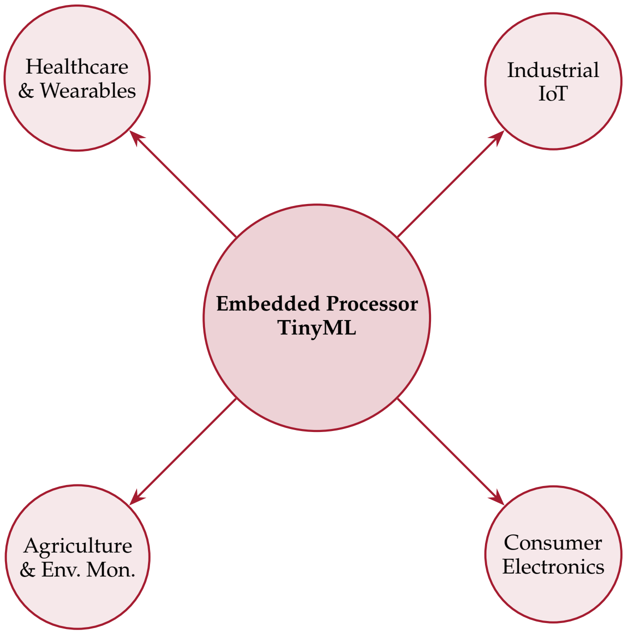


Figure 1.1: Key application domains enabled by TinyML

tasks. A built-in matrix–vector accelerator expedites the dot-product operations central to neural-network inference, while the Cortex-M33 DSP extensions assist with signal pre-processing. These features shorten latency and lower energy consumption relative to pure CPU execution.

Training Off-Chip, Inference On-Chip

TinyML models are *trained off-device*. After training, the network is pruned, quantized, and linked into the firmware as read-only weights; the microcontroller only executes the forward pass.

Several peripherals like 12-bit ADCs, DACs, UART, SPI, and I^2C support common sensor and actuator interfaces, and the integrated Bluetooth LE 5.3 radio provides a straightforward channel for firmware updates, configuration, or telemetry. Power consumption is managed by an Energy Management Unit (EMU) that can gate clocks and supplies, allowing the device to wake, perform an inference, and return to a low-power state with minimal overhead. Security features include a hardware cryptographic accelerator, secure boot, and tamper-resistant key storage. These blocks protect model weights, firmware integrity, and any sensitive application data. Development is carried out in *Simplicity Studio* with the Gecko SDK, which offers a maintained TensorFlow Lite Micro port and reference projects. All examples in this book target the low-cost `xG24-DK2601B` evaluation kit, but the source code can be adapted to other MCUs offering comparable compute, memory, and peripheral resources.

Table 1.1: EFR32MG24 specifications relevant to TinyML

Feature	**Value**	**Why it matters**
CPU core	Cortex-M33 @ 78 MHz	DSP and FPU instructions accelerate fixed- and floating-point math
Flash	1.5 MB	Holds firmware plus quantized model weights
SRAM	256 kB	Buffers activations and intermediate tensors
AI coprocessor	Matrix–vector unit	Roughly $\times 4$ faster inference than CPU alone (vendor data)
Low-power modes	EM0–EM3	Lets battery-powered sensors sleep between inferences
Wireless	Bluetooth LE 5.3	Facilitates OTA updates and data backhaul
Security	PSA Level 3 Secure Vault	Protects model IP and user data

1.4 Chapter Summary

This chapter introduced the EFR32MG24 as the reference platform for the TinyML examples that follow. Supported by the *Simplicity Studio* tool chain and Gecko SDK, the MG24 offers a balanced starting point for exploring model compression, quantization, and deployment techniques in the chapters ahead. In the next chapter we pivot from hardware to the model-side techniques like quantization, pruning, and compression that squeeze modern ML into a few hundred kilobytes.

Bibliography

[1] C. Banbury, P. Whatmough, S. Fedorov *et al.*, “Benchmarking TinyML Systems,” *Proceedings of Machine Learning and Systems (MLSys)*, pp. 98–111, 2021.

[2] R. Kumar and S. Patel, “Edge Intelligence for Privacy and Latency,” *IEEE Internet Computing*, vol. 27, no. 2, pp. 61–69, 2023.

[3] Silicon Laboratories, *EFR32MG24 Wireless Gecko SoC Family Data Sheet*, Rev. 1.2, 2024. [Online]. Available: https://www.silabs.com

Chapter 2

The Art and Science of Machine Learning

This chapter explores the fundamental principles that enable machines to learn from experience. We examine both the theoretical foundations that provide mathematical rigor for machine learning and the practical considerations that shape modern learning systems. Our journey spans from the field's historical roots to cutting-edge research and future challenges. By chapter's end, you will understand how machines acquire, represent, and apply knowledge to solve complex problems across diverse domains.

Complex concepts are presented progressively, using intuitive analogies, examples, and step-by-step explanations. Your questions and insights are welcome throughout this exploration of machine learning's art and science.

2.1 Origins and Evolution

Understanding machine learning's historical context and developmental trajectory provides essential perspective on its current state and future potential. This section traces the field's origins and highlights pivotal advances that shaped its evolution.

2.1.1 Historical Context

The vision of creating intelligent, adaptive machines has captivated humanity for centuries. Global mythology features artificially intelligent beings, from Jewish golems to ancient Chinese mechanical servants. These timeless stories reflect our enduring fascination with animating inanimate matter with cognizance.

Machine learning as a scientific discipline emerged in the mid-20th century from the convergence of four intellectual traditions:

Artificial intelligence - Creating machines capable of intelligent behavior
Statistics and probability theory - Mathematical tools for quantifying and reasoning about uncertainty
Optimization and control theory - Principles for automated decision-making and goal-directed behavior
Neuroscience and cognitive psychology - Scientific study of natural learning in biological systems

These fields contributed essential ideas and techniques that formed modern machine learning's foundation.

Early milestones include:

- *1943* - McCulloch and Pitts publish "A Logical Calculus of the Ideas Immanent in Nervous Activity", establishing artificial neural network foundations
- *1950* - Turing proposes the "Turing Test" in "Computing Machinery and Intelligence", defining machine intelligence operationally
- *1952* - Samuel creates the first computer learning program, which mastered checkers beyond its creator's ability
- *1957* - Rosenblatt invents the Perceptron, an early neural network prototype capable of learning visual pattern classification

- *1967* - Vapnik and Chervonenkis introduce covering numbers and the VC dimension, establishing statistical learning theory foundations

These pioneering efforts established the conceptual and technical framework for decades of research that transformed the field into today's thriving discipline.

2.1.2 From Rule-Based to Learning Systems

Early artificial intelligence research emphasized symbolic logic and deductive reasoning through expert systems which were programs encoding human knowledge as explicit logical rules. MYCIN, developed at Stanford University in the early 1970s for diagnosing blood infections, exemplified this approach. Its knowledge base contained hundreds of IF-THEN rules from expert physicians:

```
IF (organism-1 is gram-positive) AND
   (morphology of organism-1 is coccus) AND
   (growth-conformation of organism-1 is chains)
THEN there is suggestive evidence (0.7) that
     the identity of organism-1 is streptococcus
```

MYCIN chained these rules to produce diagnostic conclusions rivaling human specialists.

However, handcrafted knowledge engineering faced serious limitations. The knowledge acquisition bottleneck made extracting expert knowledge time-consuming and inconsistent. Systems proved brittle, struggling with noisy data and novel situations. Their complex inference chains created opaque reasoning that was difficult to understand and debug. Most critically, rule-based systems remained static, unable to learn from experience or improve automatically.

These shortcomings necessitated a new approach that could learn directly from data rather than rely on rigid symbolic rules.

2.1.3 The Statistical Revolution

The transition from rule-based to learning systems arose from two insights: real-world domains are inherently uncertain, requiring probabilistic treatment, and expertise is often implicit and intuitive, making it more suitable for statistical extraction than symbolic codification.

Medical diagnosis illustrates these complexities. Symptoms correlate imperfectly with disorders, diagnostic tests have varying accuracy levels, and diseases manifest differently across stages. Treatment effects vary by patient, and disease patterns evolve over time. In this uncertain environment, diagnosis requires probabilistic reasoning based on empirical observations and prior knowledge, not deterministic rules.

Machine learning's key innovation reframed the challenge statistically. Instead of manually encoding rules, the goal became automatically inferring probabilistic relationships from data. Rather than requiring explicit knowledge enumeration, algorithms extracted latent patterns implicitly. In place of brittle logical chains, models learned robust statistical associations that handled noise and uncertainty gracefully.

This perspective shift unlocked powerful techniques combining probability theory and optimization. Maximum likelihood estimation, developed by Fisher in the 1920s, provided a framework for inferring statistical model parameters from data. Rosenblatt's perceptron (1957) introduced neural networks capable of learning linearly separable patterns. Robbins and Monro's stochastic gradient descent (1951) offered efficient optimization for large-scale learning. Backpropagation, discovered independently in the 1970s-1980s, enabled training of multi-layer neural networks. Vapnik and Chervonenkis's VC dimension (1960s-1970s) quantified hypothesis space capacity for stable learning. Finally, Vapnik's support vector machines (1990s) delivered discriminative algorithms with strong theoretical guarantees.

These innovations established the foundation for statistical learning systems that extract knowledge from raw data, setting the stage for machine learning's transformation into one of today's most impactful technologies.

2.2 Understanding Learning Systems

Having explored machine learning's historical context and conceptual shifts, we now examine learning systems in depth. This section covers the fundamental principles defining the learning paradigm, data's central role, and the nature of patterns that learning uncovers.

2.2.1 The Learning Paradigm

Machine learning represents a radical departure from traditional programming. Consider solving object recognition using classical programming. First, we would articulate rules like "an eye has a roughly circular shape," "a nose is usually located below the eyes and above the mouth," and "a face is an arrangement of eyes, nose and mouth." Next, we would translate these into programmatic instructions: "scan the image for circular regions," "check if there are two such regions in close horizontal proximity," and "label these candidate eye regions." We would then debug and refine our program for edge cases and variability. Adding new object categories would require repeating this arduous process.

The classical approach places the entire burden on the human programmer, who must explicitly specify every decision and edge case from a blank slate.

Machine learning follows a different recipe. We collect labeled examples (images paired with object names), select a general-purpose model family capable of capturing relevant patterns (such as deep convolutional neural networks), and specify a success measure (the fraction of correctly labeled images) as our objective function. A learning algorithm automatically adjusts model parameters to optimize this objective on the provided examples. We then evaluate the trained model on a separate test set to assess generalization.

The emphasis shifts fundamentally. Rather than explaining "how" to solve tasks, we provide examples of "what" we want and let algorithms determine the "how." Instead of handcrafting solution steps, we select flexible models and delegate parameter tuning to optimization procedures. Open-ended debugging gives way to controlled experiments measuring generalization.

Consider spam email classification as a concrete example:

```python
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score

# 1. Collect labeled data
emails, labels = load_email_data()

# 2. Select a model family
vectorizer = CountVectorizer()  # convert email text to word counts
classifier = MultinomialNB()    # naive Bayes with multinomial likelihood
```

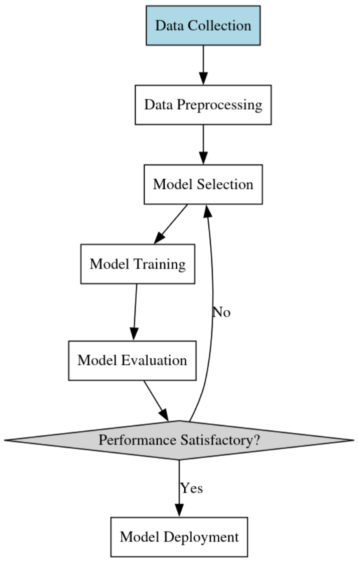


Figure 2.1: Typical machine learning workflow: data is collected, preprocessed, and used to select and train a model. The model is evaluated iteratively until performance is satisfactory, after which it is deployed.

```
# 3. Specify an objective function
def objective(model, X, y):
    return accuracy_score(y, model.predict(X))

# 4. Feed data to a learning algorithm
# learn a spam classifier from 70% of the data
train_emails, test_emails, train_labels, test_labels = train_test_split(
        emails, labels, train_size=0.7, stratify=labels)
X_train = vectorizer.fit_transform(train_emails)
classifier.fit(X_train, train_labels)

# 5. Evaluate generalization on held-out test set
X_test = vectorizer.transform(test_emails)
print("Test accuracy:", objective(classifier, X_test, test_labels))
```

This example demonstrates the key ingredients: data (emails with spam/not spam labels), a model (naive Bayes classifier relating word counts to labels), an objective (accuracy metric), a learning algorithm (the `fit` method maximizing likelihood), and generalization (evaluation on held-out test data).

Real-world applications involve larger datasets and complex models, but the paradigm remains consistent: optimizing objective functions on training data enables algorithms to extract patterns that generalize to new situations.

2.2.2 Learning from Data

Data plays a fundamental role in machine learning, distinguishing it through automatic knowledge extraction from empirical observations rather than human-encoded expertise. Understanding how learning systems leverage data is crucial.

In machine learning, a dataset contains examples, each providing a concrete task instantiation. For spam classification, each example is an email with its spam/not spam label. Formally, an example is a pair (x,y), where x represents input features (quantitative properties like word counts) and y represents the target output (binary label for spam, continuous value for regression, or complex structures).

A dataset is a collection of n examples:

$$D = (x_1, y_1), ..., (x_n, y_n)$$

Learning aims to infer a function $f : X \to Y$ such that $f(x) \approx y$ for future unseen examples.

Effective learning requires several key data properties. Representativeness ensures training examples reflect real-world distributions; systematic bias undermines generalization. Quantity matters because more data provides richer sampling and prevents overfitting, though requirements vary by task complexity. Quality suffers from noise, outliers, and missing values, necessitating careful preprocessing. Diversity ensures sufficient variability along task-relevant dimensions; homogeneous examples fail to capture necessary behavioral ranges. For supervised learning, consistent labeling proves critical, as noisy or ambiguous labels degrade model quality.

Consider this toy spam dataset:

```
train_emails = [
    "Subject: You won't believe this amazing offer!",
    "Subject: Request for project meeting",
    "Subject: URGENT: Update your information now!",
    "Hey there, just wanted to follow up on our conversation...",
    "Subject: You've been selected for a special promotion!",
]

train_labels = ["spam", "not spam", "spam", "not spam", "spam"]
```

This dataset reveals several issues. Five examples cannot produce a robust classifier covering email diversity; the model will likely overfit to specific subject lines. The narrow range (mainly short subject lines) lacks representativeness compared to real emails containing varied content. Label consistency appears questionable that is the fourth email's classification depends on unknown conversational context.

Addressing these issues requires larger, diverse labeled examples and careful preprocessing: tokenizing text into words or n-grams, removing stop words and punctuation, stemming words to collapse variants, normalizing features to avoid length bias, and resolving labeling inconsistencies.

High-quality data proves essential for successful learning. While model and algorithm choices matter, data preparation quality often determines results. As the adage states, ”garbage in, garbage out,” i.e., no algorithmic sophistication can extract meaningful patterns from noisy, biased, inconsistent data.

2.2.3 The Nature of Patterns

Having examined data's role in learning, we now explore patterns which are regularities that learning algorithms extract. What constitutes patterns in machine learning, and how do systems represent and leverage them?

Patterns are regularities or structures in data that capture task-relevant information. In spam classification, relevant patterns include words more common in spam (”special offer,” ”free trial,” ”no credit check”), unusual formatting suggesting marketing content (excessive capitalization, colorful text), and suspicious sender information (identity-address mismatches, known spam domains).

Machine learning's key insight is that patterns can be represented mathematically as operations in formal spaces. Word presence can be encoded as binary vectors, with each dimension representing a vocabulary word:

```python
vocabulary = ["credit", "offer", "special", "trial", "won't", "believe", ...]

def email_to_vector(email):
    vector = [0] * len(vocabulary)
    for word in email.split():
        if word in vocabulary:
            index = vocabulary.index(word)
            vector[index] = 1
    return vector

# Example usage
message1 = "Subject: You won't believe this amazing offer!"
message2 = "Subject: Request for project meeting"

print(email_to_vector(message1))
# Output: [0, 1, 0, 0, 1, 1, ...]

print(email_to_vector(message2))
# Output: [0, 0, 0, 0, 0, 0, ...]
```

This ”bag of words” representation transforms emails into vectors indicating vocabulary word presence. Patterns emerge and spam messages typically produce vectors with more non-zero entries, suggesting marketing language presence.

This crude representation discards potentially relevant information like word order and context. Sophisticated approaches preserve additional structure by using word counts or tf-idf weights for frequencies, extracting n-grams to preserve local word order, applying latent semantic analysis to identify themes, and learning dense embeddings mapping words to continuous semantic spaces.

These approaches share a systematic mapping from raw data to mathematically tractable representations. Learning occurs through discovering specific mapping parameters that yield effective training performance, implicitly identifying task-relevant patterns.

Supervised learning algorithms extract patterns by learning a function $f : X \to Y$ mapping inputs to outputs. Most algorithms define a parametrized function family F_θ and search for parameters minimizing empirical risk on training examples:

$$\theta* = argmin_\theta 1/n \sum_i L(F_\theta(x_i), y_i)$$

Here L quantifies discrepancy between predictions $F_\theta(x_i)$ and true labels y_i, summing over n training examples.

Different algorithms employ specific function families and loss functions, but all seek patterns which are captured by parameters θ enabling predictions to match actual labels.

Consider logistic regression for spam classification, which learns a linear function:

$$F_\theta(x) = \sigma(\theta^T x)$$

Here x represents word-presence features, θ contains real-valued weights, and σ is the sigmoid function squashing the linear combination to a spam probability between 0 and 1.

With binary cross-entropy loss, the objective becomes:

$$\theta * = argmin_\theta 1/n \sum_i [-y_i log(F_\theta(x_i)) - (1-y_i) log(1-F_\theta(x_i))]$$

where $y_i \in 0, 1$ indicates true labels.

Gradient descent optimization yields weights where words common in spam (”offer,” ”free”) receive positive weights, increasing spam probability, while words common in legitimate emails (”meeting,” ”project”) receive negative weights. Weight magnitudes indicate predictive strength that is larger positive weights signal spam, larger negative weights signal legitimate mail.

Learning automatically discovers word usage patterns distinguishing spam from non-spam, summarized in weights $\theta *$. These weights define a decision boundary in feature space i.e. emails on one side classify as spam, those on the other as legitimate.

This example illustrates key properties common to learning algorithms. Parameters θ compactly summarize task-relevant patterns, capturing word-label correlations. The process is data-driven, with weights determined by empirical word frequency distributions rather than hand-coded rules. Patterns are task-specific, tuned for spam classification performance. Pattern expressiveness depends on model family and linear models capture simple relationships, while deep networks capture richer patterns.

Modern systems employ higher-dimensional features, elaborate models, and sophisticated optimization, but the principle remains: adjusting flexible model parameters to minimize training risk enables algorithms to discover generalizable patterns for novel examples.

2.3 The Nature of Machine Learning

Having examined learning systems’ fundamental components which are the data they process and patterns they extract, we now address higher-level questions about learning itself. What does machine ”learning” mean? How does it differ from other artificial intelligence approaches? What challenges and opportunities does this paradigm present?

2.3.1 Learning as Induction

Machine learning fundamentally represents inductive inference that is drawing general conclusions from specific examples. This contrasts with deductive inference, which derives specific conclusions from general premises.

Consider deductive reasoning: All men are mortal (premise), Socrates is a man (premise), therefore Socrates is mortal (conclusion). The conclusion follows necessarily from the premises.

Inductive reasoning reverses this direction: Socrates is a man and is mortal, Plato is a man and is mortal, Aristotle is a man and is mortal, therefore all men are mortal. Here, the conclusion remains uncertain despite true premises which is that we cannot guarantee the next man encountered will be mortal. Conclusions are probable, with confidence depending on example quantity and diversity.

Machine learning performs algorithmic induction i.e. learning algorithms discover data patterns and make predictions about novel cases. Like human induction, machine learning conclusions are never guaranteed but can be highly probable with representative training data and appropriate models.

In spam classification, we learn a function f mapping email features x to labels y, where $f(x) \approx y$ for new examples. Using logistic regression:

$$f(x) = \sigma(\theta^T x)$$

where θ represents learned weights and σ is the sigmoid function.

After training on 1000 labeled emails, finding weights $\theta*$ that minimize cross-entropy loss, we classify new emails:

```
def predict_spam(email, weights):
    features = email_to_vector(email)
    score = weights.dot(features)
    probability = sigmoid(score)
    return probability > 0.5

# Example usage
weights = train_logistic_regression(train_emails, train_labels)

new_email = "Subject: Amazing opportunity to work from home!"
prediction = predict_spam(new_email, weights)

print(prediction)  # Output: True
```

This process embodies induction: starting with specific examples (training emails), learning general rules (weight vector $\theta*$), and applying these rules to unseen examples. Predictions lack guarantees like the learned rule generalizes from limited training samples. However, sound inductive reasoning capturing meaningful regularities yields mostly correct predictions, improving with larger, diverse datasets.

Complex domains like computer vision or natural language processing involve higher-dimensional features, elaborate models, and intensive optimization, but the inductive principle remains: collect representative training examples, define expressive models and objectives, optimize parameters on training data, apply learned models to new inputs, and iterate based on evaluation results.

This paradigm's power lies in its generality which is framing pattern discovery as optimization enables application across domains without detailed knowledge engineering. The same approach learns patterns in images, text, speech, and countless data types.

However, this generality introduces challenges. Performance depends fundamentally on data quality and representativeness. Learned representations can be complex and opaque, especially in deep neural networks. Statistical pattern discovery struggles with explicit logical reasoning requiring deliberation or symbolic manipulation. Training data biases can perpetuate or amplify societal inequities, necessitating careful auditing.

Despite challenges, inductive learning proves remarkably effective across applications from medical diagnosis to autonomous vehicles, discovering patterns too subtle for manual specification. Growing data and computing resources will likely expand machine learning's scope and impact.

2.3.2 The Role of Uncertainty

Machine learning is fundamentally probabilistic. Algorithm conclusions represent probability statements based on training patterns, never absolute certainty.

In spam classification, even extensive training cannot guarantee pattern persistence for all future emails. New spam types or unusual legitimate emails may contradict learned patterns. Good models provide probability estimates, not definite classifications. Logistic regression outputs numbers between 0 and 1 representing spam probability given input features.

This probabilistic nature strengthens the approach by quantifying prediction confidence for downstream decision-making. An email client can set confidence thresholds like moving only messages with 95% spam probability to spam folders trading false positives against false negatives principled.

Well-calibrated probabilities enable uncertainty-aware decisions: deferring borderline cases to human judgment, gathering additional information to resolve uncertainty, hedging decisions balancing risk and reward, and combining multiple model predictions for improved confidence.

Probability calibration accuracy is essential. Models predicting 95% confidence for 60% occurrence events provide unreliable uncertainty estimates.

Techniques for quantifying and calibrating uncertainty include explicit probability models (Bayesian networks, Gaussian processes) incorporating prior knowledge, ensemble methods (bagging, boosting) where prediction variation measures uncertainty, calibration methods (Platt scaling, isotonic regression) adjusting raw confidence scores, and conformal prediction providing guaranteed coverage rates.

Quantifying uncertainty builds trust and promotes responsible system use. Meaningful confidence estimates enable informed reliance decisions, especially crucial in high-stakes domains like healthcare or criminal justice.

Uncertainty reasoning is central to advanced paradigms. Reinforcement learning agents make decisions under stochastic environments with uncertain action consequences. Active learning models query for labels reducing uncertainty most effectively based on current knowledge.

As machine learning boundaries expand, proper uncertainty quantification becomes increasingly essential for robust models and effective human-AI collaboration.

2.3.3 Model Complexity and Regularization

Machine learning must balance model complexity against generalization performance. Models need sufficient expressiveness to capture meaningful patterns without overfitting to training set noise or idiosyncrasies.

This bias-variance dilemma involves two error sources. Bias represents error from modeling assumptions and simplifications that is high-bias models make strong assumptions potentially causing underfitting. Variance represents error from sensitivity to training data fluctuations and thus high-variance models fit training sets well but may overfit and fail to generalize.

Consider fitting curves to scattered points. Linear models have high bias, low variance assuming linear relationships limits complex pattern fitting but provides stability across data subsets. High-degree polynomials have low bias, high variance fitting training points extremely well while oscillating wildly between them.

Increasing model complexity (adding features, deepening networks, reducing regularization) decreases bias but increases variance. The goal is finding optimal complexity capturing relevant patterns without overfitting noise.

Hypothesis space choice controls complexity. Simple spaces (linear functions) have low variance, potentially high bias. Complex spaces (deep networks) have low bias, potentially high variance.

Regularization constrains parameters limiting overfitting. Common approaches include parameter norm penalties (L2 weight decay penalizing squared norms, L1 penalizing absolute values), dropout (randomly zeroing network activations during training), and early stopping (halting optimization when validation error increases despite decreasing training error).

Regularization amount and type require tuning based on data and model characteristics. Excessive regularization causes underfitting; insufficient regularization causes overfitting. Cross-validation and information criteria guide appropriate settings.

The bias-variance tradeoff varies with training data quantity. Classical small-data regimes require essential regularization preventing overfitting. Modern large-dataset, overparameterized regimes show reduced overfitting risk with subtler regularization roles.

Recent research reveals ”double descent” behavior in overparameterized models i.e. complexity increases beyond training data interpolation can improve generalization, challenging classical bias-variance understanding.

Despite evolving understanding, managing complexity and using regularization for generalization remain central to machine learning practice. Training increasingly powerful models on larger datasets requires balancing expressiveness against constraints for robust, reliable performance.

2.4 Building Learning Systems

Having explored machine learning’s theoretical principles, we now address practical considerations for building effective learning systems. What components comprise successful machine learning pipelines, and how do they integrate?

2.4.1 Data Preparation

Data preparation constitutes the first and most crucial step in machine learning projects. As discussed in Section 6.2.2, training data quality and representativeness are essential for learning patterns that generalize to new examples. No algorithmic sophistication compensates for fundamentally flawed or insufficient data.

Key data preparation aspects include data cleaning (identifying and correcting errors, inconsistencies, and missing values through duplicate removal, format standardization, and statistical imputation), feature engineering (transforming raw data into learning-amenable representations through normalization, categorical encoding, domain-specific extraction, and dimensionality reduction), data augmentation (expanding training sets through transformations, particularly in computer vision using random cropping, flipping, and color jittering), and data splitting (dividing data into training, validation, and testing sets for parameter fitting, hyperparameter tuning, and performance evaluation respectively).

Data preparation specifics vary by domain and characteristics, but ensuring quality, representativeness, and learning suitability remains universal. While the claim that data preparation comprises 80% of machine learning work may exaggerate, it underscores the critical importance of proper data handling.

Consider housing price prediction based on square footage, bedrooms, and location. Raw data might appear as:

```
Address,Sq.Ft.,Beds,Baths,Price
123 Main St,2000,3,2.5,$500,000
456 Oak Ave,1500,2,1,"$350,000"
789 Elm Rd,1800,3,2,425000
```

Preparation steps include standardizing the `Price` column (by removing “$” and “,” characters and converting to numeric), imputing missing values for bedrooms and bathrooms using the median or mode, normalizing square footage by subtracting the mean and dividing by the standard deviation,

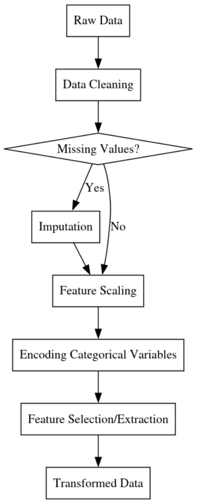


Figure 2.2: Data preprocessing pipeline: raw data undergoes cleaning, handling of missing values through imputation if needed, feature scaling, encoding of categorical variables, and feature selection or extraction, resulting in transformed data ready for modeling.

applying one-hot encoding to the `Address` field to create binary location features, and performing stratified splitting to ensure representative price distributions across training and test sets.

The prepared result:

```
Sq.Ft._Norm,Beds,Baths,123_Main_St,456_Oak_Ave,789_Elm_Rd,...,Price
,3,2.5,1,0,0,...,500000
-0.58,2,1,0,1,0,...,350000
,3,2,0,0,1,...,425000
...
```

This simplified example illustrates that careful data preparation investment is essential for successful learning systems.

2.4.2 Model Selection and Training

After data preparation, model family and training procedure selection follows. As discussed in Section 6.3.3, this requires balancing model complexity against generalization ability through cross-validation and regularization.

Model selection considerations include inductive biases (built-in assumptions like convolutional networks' translation invariance for image recognition), parameter complexity (learnable parameter count affecting pattern-fitting capacity and overfitting potential), computational complexity (training and inference time/memory requirements potentially necessitating specialized hardware), and interpretability (human inspection and understanding capability, crucial for healthcare or finance applications).

Model family choice depends on problem nature and data characteristics. Structured data with clear semantics suits ”shallow” models like linear regression, decision trees, or support vector machines.

Unstructured data (images, audio, text) typically requires "deep" models like convolutional or recurrent networks.

Training involves instantiating models with initial parameters, defining loss functions measuring prediction-label discrepancies, using optimization algorithms (stochastic gradient descent) for iterative parameter updates, monitoring validation performance for overfitting detection and hyperparameter tuning, and stopping when validation performance plateaus or degrades.

For housing price prediction using regularized linear regression:

$$price = w_0 + w_1 * sqft + w_2 * beds + w_3 * baths + ...$$

Training uses gradient descent on mean squared error:

```python
def mse_loss(y_true, y_pred):
    return np.mean((y_true - y_pred) ** 2)

def gradient_descent(X, y, w, lr=0.01, num_iters=100):
    for i in range(num_iters):
        y_pred = np.dot(X, w)
        error = y_pred - y
        gradient = 2 * np.dot(X.T, error) / len(y)
        w -= lr * gradient
    return w

# Add a bias term to the feature matrix
X = np.c_[np.ones(len(X)), X]

# Initialize weights to zero
w = np.zeros(X.shape[1])

# Train the model
w = gradient_descent(X, y, w)
```

Adding L2 regularization prevents overfitting:

```python
def mse_loss_regularized(y_true, y_pred, w, alpha=0.01):
    return mse_loss(y_true, y_pred) + alpha * np.sum(w**2)

def gradient_descent_regularized(X, y, w, lr=0.01, alpha=0.01, num_iters=100):
    for i in range(num_iters):
        y_pred = np.dot(X, w)
        error = y_pred - y
        gradient = 2 * np.dot(X.T, error) / len(y) + 2 * alpha * w
        w -= lr * gradient
    return w

# Train the regularized model
w = gradient_descent_regularized(X, y, w)
```

The `alpha` parameter controls regularization strength thus larger values constrain weights more strongly. Tuning learning rate, regularization strength, and iterations finds optimal balance between training fit and generalization.

Alternative models like decision trees, random forests, or neural networks offer different hyperparameter sets. Success requires combining domain knowledge, empirical validation, and iterative refinement.

2.4.3 Model Evaluation and Deployment

After achieving strong validation performance, final evaluation on held-out test sets provides unbiased generalization estimates.

Classification metrics include accuracy (correctly classified fraction), precision (true positive fraction among positive predictions), recall (predicted positive fraction among actual positives), F1 score (precision-recall harmonic mean), and ROC AUC (receiver operating characteristic curve area).

Regression metrics include mean squared error (average squared prediction-actual differences), mean absolute error (average absolute differences), and R-squared (predictable target variance proportion).

Metric selection must align with business goals. Fraud detection prioritizes recall (catching fraudulent transactions) over precision (avoiding false alarms), while medical diagnosis prioritizes precision (avoiding false positives causing unnecessary treatments).

Satisfactory test performance enables production deployment, integrating trained models into larger systems applying predictions to new data for end users.

Deployment considerations include scalability (handling production data volume/velocity through batch processing, streaming, or distributed computation), latency (meeting business requirements through model compression, quantization, or hardware acceleration), monitoring (tracking metrics, detecting drift, periodic retraining), and security (protecting models and predictions through input validation, output filtering, access controls).

Production deployment and maintenance requires collaboration between data scientists, engineers, and domain experts, with tools and practices evolving continuously.

For housing price prediction, test evaluation proceeds:

```python
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score

# Generate predictions on the test set
y_pred = np.dot(X_test, w)

# Calculate evaluation metrics
mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print(f"Test MSE: {mse:.2f}")
print(f"Test MAE: {mae:.2f}")
print(f"Test R^2: {r2:.2f}")
```

Satisfactory performance enables deployment as a housing price estimation service: wrapping models in web service APIs accepting features and returning predictions, integrating APIs with user applications for property information input and estimate receipt, establishing data pipelines for continuous collection and periodic retraining capturing market changes, defining monitoring dashboards and alerts tracking performance and detecting anomalies, and establishing governance policies managing model lifecycles from development through retirement.

This simplified example illustrates the complete machine learning system process from data preparation through development, deployment, and maintenance.

2.5 The Ethics and Governance of Machine Learning

As machine learning systems become increasingly prevalent and powerful, addressing the ethical implications of their development and deployment becomes crucial. This final section explores key ethical considerations and governance principles for responsible machine learning.

2.5.1 Fairness and Bias

Ensuring model fairness and eliminating bias represents one of machine learning's most pressing ethical challenges. Training data reflecting historical biases or discrimination can produce models that perpetuate or amplify these biases in predictions.

Consider a hiring model trained on past decisions to predict candidate success. If training data comes from companies with discriminatory practices, the model may penalize underrepresented groups regardless of actual job performance predictability.

Active research on detecting and mitigating bias includes demographically balancing datasets for equal group representation, adversarial debiasing to remove sensitive information from representations, regularization techniques penalizing disparate impact, and post-processing methods adjusting outputs to satisfy fairness constraints.

These techniques have limitations, often revealing tradeoffs between fairness and accuracy. Fairness extends beyond technical solutions, requiring ongoing collaboration among machine learning practitioners, domain experts, policymakers, and affected communities.

2.5.2 Transparency and Accountability

Transparency and accountability form essential ethical principles for machine learning. Increasing model complexity and consequence makes understanding prediction mechanisms and tracing training data provenance more difficult.

This opacity complicates auditing for bias, safety, or regulatory compliance. It also hinders challenging or appealing machine learning decisions, reducing human agency and recourse.

Techniques promoting transparency include model interpretability methods providing human-understandable explanations, provenance tracking documenting data and model lineage, audit trails enabling reproducibility and historical analysis, and participatory design involving stakeholders in development and governance.

Transparency requires institutional structures beyond technical solutions. This involves designating responsible individuals for ethical development and deployment, establishing review boards assessing social impact and governance, creating feedback channels for affected communities, and developing legal frameworks enforcing transparency standards.

2.5.3 Safety and Robustness

Machine learning deployment in high-stakes domains like healthcare, transportation, criminal justice makes safety and robustness paramount. Brittle, unreliable, or easily fooled models can cause serious harm without careful design and testing.

Key safety and robustness challenges include distributional shift (models failing on different data distributions), adversarial attacks (malicious inputs causing egregious errors), reward hacking (wrong objectives producing harmful behaviors), and safe exploration (learning without catastrophic actions).

Improvement techniques include anomaly detection flagging out-of-distribution inputs, adversarial training increasing perturbation resilience, constrained optimization respecting safety boundaries, and formal verification providing behavioral guarantees.

Building truly safe systems requires rigorous safety culture throughout development, collaboration between practitioners and safety professionals, proactive engagement aligning development with societal values, and ongoing monitoring catching unintended consequences.

2.5.4 Ethical Principles and Governance Frameworks

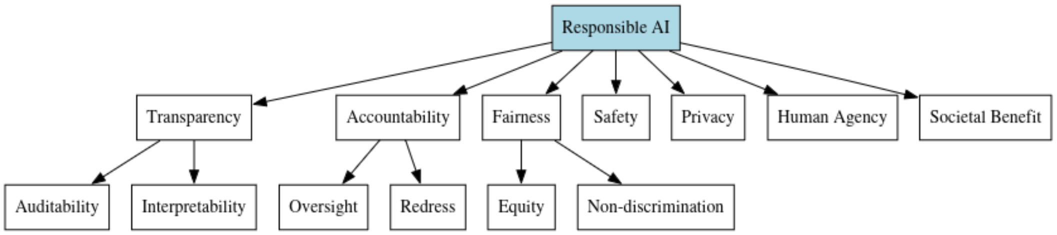


Figure 2.3: Key pillars of Responsible AI: including transparency, accountability, fairness, safety, privacy, human agency, and societal benefit. Each pillar is further supported by sub-principles such as interpretability, oversight, equity, and non-discrimination.

Navigating machine learning's ethical landscape requires clear principles and governance frameworks guiding responsible development. Key principles include transparency (auditable, understandable systems), accountability (clear oversight and redress mechanisms), fairness (equitable treatment avoiding discrimination), safety (reliable, robust lifecycle performance), privacy (respecting rights with appropriate protections), human agency (preserving autonomy and control), and societal benefit (prioritizing social good and collective wellbeing).

Translating principles into practical governance involves ethical codes and professional standards, impact assessment and risk management processes, stakeholder engagement ensuring community voice, regulatory sandboxes testing governance approaches, and international standards addressing global development nature.

Machine learning governance ultimately aims to align technology development with human values, enhancing rather than undermining human flourishing. This complex, ongoing process demands sustained collaboration across disciplines, sectors, and geographies.

2.6 Conclusion

This chapter has comprehensively explored machine learning foundations, from historical roots through modern techniques to future challenges. We have witnessed the field's evolution from rule-based expert systems to data-driven statistical learning, powered by expanding data availability and computing power.

We examined fundamental machine learning components i.e. the data systems process, patterns they extract, and algorithms powering learning. Key concepts including inductive bias, generalization, overfitting, and regularization revealed the art of building effective models. We detailed practical pipeline construction from data preparation through model selection to deployment and monitoring. Finally, we addressed ethical challenges and governance principles arising when building systems significantly impacting human lives.

Machine learning continues evolving rapidly, with new breakthroughs and challenges emerging annually. Key frontiers and open questions include continual and lifelong learning (building models that learn continuously and adapt without forgetting previous knowledge), causality and interpretability (moving beyond associations to uncover causal relationships in human-interpretable models), robustness and safety (guaranteeing reliable behavior despite distributional shift, adversarial attacks, or unexpected situations), human-AI collaboration (designing systems augmenting rather than replacing human intelligence), and ethical alignment (ensuring development promotes beneficial societal outcomes aligned with human values).

Advancing machine learning requires interdisciplinary collaboration spanning computer science, statistics, psychology, social science, philosophy, and ethics. It demands engagement with policymakers, industry leaders, and the public to ensure responsible, inclusive development. The goal is building systems that learn from experience to make decisions benefiting humanity across healthcare, scientific discovery, and daily life improvement.

Realizing this potential extends beyond technical progress to addressing fairness, accountability, transparency, and safety while navigating ethical and governance challenges. Machine learning practitioners must push technological boundaries while considering their work's broader impact. Embracing diverse perspectives and collaborating beyond our field shapes AI's future. Maintaining curiosity, critical thinking, and commitment to responsible development ensures machine learning serves society for generations to come.

Bibliography

[1] D. Amodei, C. Olah, J. Steinhardt *et al.*, "Concrete Problems in AI Safety," *arXiv preprint arXiv:1606.06565*, 2016.

[2] S. Barocas, M. Hardt, and A. Narayanan, *Fairness and Machine Learning: Limitations and Opportunities*. fairmlbook.org, 2019.

[3] C. M. Bishop, *Pattern Recognition and Machine Learning*. New York, NY: Springer, 2006.

[4] L. Breiman, "Statistical Modeling: The Two Cultures," *Statistical Science*, vol. 16, no. 3, pp. 199–231, 2001.

[5] P. Domingos, "A Few Useful Things to Know About Machine Learning," *Communications of the ACM*, vol. 55, no. 10, pp. 78–87, 2012.

[6] C. Dwork, M. Hardt, T. Pitassi *et al.*, "Fairness Through Awareness," *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, pp. 214–226, 2012.

[7] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA: MIT Press, 2016.

[8] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, 2nd ed. New York, NY: Springer, 2009.

[9] M. I. Jordan and T. M. Mitchell, "Machine Learning: Trends, Perspectives, and Prospects," *Science*, vol. 349, no. 6245, pp. 255–260, 2015.

[10] Y. LeCun, Y. Bengio, and G. Hinton, "Deep Learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[11] W. S. McCulloch and W. Pitts, "A Logical Calculus of the Ideas Immanent in Nervous Activity," *The Bulletin of Mathematical Biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[12] T. M. Mitchell, *Machine Learning*. New York, NY: McGraw-Hill, 1997.

[13] K. P. Murphy, *Machine Learning: A Probabilistic Perspective*. Cambridge, MA: MIT Press, 2012.

[14] J. Pearl, *Causality: Models, Reasoning, and Inference*, 2nd ed. Cambridge, UK: Cambridge University Press, 2009.

[15] H. Robbins and S. Monro, "A Stochastic Approximation Method," *The Annals of Mathematical Statistics*, vol. 22, no. 3, pp. 400–407, 1951.

[16] F. Rosenblatt, "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain," *Psychological Review*, vol. 65, no. 6, pp. 386–408, 1958.

[17] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed. Upper Saddle River, NJ: Pearson, 2020.

[18] A. L. Samuel, "Some Studies in Machine Learning Using the Game of Checkers," *IBM Journal of Research and Development*, vol. 3, no. 3, pp. 210–229, 1959.

[19] S. Shalev-Shwartz and S. Ben-David, *Understanding Machine Learning: From Theory to Algorithms*. Cambridge, UK: Cambridge University Press, 2014.

[20] E. H. Shortliffe, *Computer-Based Medical Consultations: MYCIN*. New York, NY: Elsevier, 1976.

[21] D. Silver, A. Huang, C. J. Maddison *et al.*, "Mastering the Game of Go with Deep Neural Networks and Tree Search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[22] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. Cambridge, MA: MIT Press, 2018.

[23] A. M. Turing, "Computing Machinery and Intelligence," *Mind*, vol. 59, no. 236, pp. 433–460, 1950.

[24] V. Vapnik, *The Nature of Statistical Learning Theory*. New York, NY: Springer, 1995.

[25] V. Vapnik and A. Chervonenkis, "On the Uniform Convergence of Relative Frequencies of Events to Their Probabilities," *Theory of Probability and Its Applications*, vol. 16, no. 2, pp. 264–280, 1971.

Chapter 3

The "Hello World" of TinyML

Machine learning at the edge represents a major change in computational intelligence, allowing advanced inference capabilities on resource-limited embedded systems such as the EFR32MG24 Wireless Gecko microcontroller. This chapter explores the theoretical foundations and practical implementations of TinyML specifically designed for microcontroller deployment, with particular focus on the sine wave prediction model as the standard "Hello World" example of TinyML.

The concept of a "Hello World" example has long been a tradition in programming, where new technologies are introduced with simple code that shows basic functionality. In the domain of TinyML, our sine wave prediction serves as a clear introduction to the complete process of building, training, and deploying models to microcontrollers.

3.1 Theoretical Foundations of TinyML for Microcontrollers

3.1.1 The Computational Constraints Paradigm

Traditional machine learning systems operate under the assumption of plentiful computational resources, where model complexity and size are less important than performance metrics. TinyML, however, reverses this paradigm, placing primary emphasis on resource efficiency while maintaining acceptable inference quality.

For the EFR32MG24 platform, with its ARM Cortex-M33 core, limited memory footprint (1536KB flash and 256KB RAM), and power-sensitive applications, we must consider several key factors. First, memory-constrained learning requires operating within a 256KB RAM budget, which means models must have minimal memory footprints. Second, computation-constrained inference demands algorithmic optimizations to achieve real-time performance on the 78MHz Cortex-M33 processor. Third, energy-constrained execution is essential since battery-powered applications require power-aware ML implementations.

These constraints fundamentally change our approach to machine learning model design, training methodologies, and deployment strategies.

3.1.2 Model Compression and Quantization

Central to TinyML is the concept of model compression, which can be expressed as an optimization problem:

$$\min_{\theta'} \mathcal{L}(f_{\theta'}(X), Y) \quad \text{s.t.} \quad |\theta'| \ll |\theta|$$

Where θ represents the parameters of the original model, θ' the compressed model parameters, $\mathcal{L}$ the loss function, and $f_{\theta'}(X)$ the model predictions on input X compared against ground truth Y.

Quantization—a key technique in this domain—transforms floating-point weights and activations to reduced-precision integers:

$$Q(w) = \text{round}\left(\frac{w}{\Delta}\right) \cdot \Delta$$

Where Δ represents the quantization step size. This transformation reduces both memory requirements and computational complexity at the cost of some precision.

3.2 Building Our Sine Wave Model in Google Colab

3.2.1 Generating and Processing the Dataset

For our introductory TinyML example, we will create a sine wave predictor that learns to approximate the sine function. This represents an ideal starting point for several reasons. The sine function is mathematically well-defined and bounded, making it predictable and stable for learning. The input-output relationship shows nonlinearity that requires proper model architecture, providing a meaningful challenge for the neural network. Additionally, the implementation can produce visually verifiable results on a microcontroller, allowing easy validation of the deployed model.

Let’s begin by creating a Google Colab notebook to build and train our model. Open a new notebook and start with the following code to generate our training data:

```python
import numpy as np
import math
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras import layers
import os

# Generate a uniformly distributed set of random numbers in the range from
# 0 to 2 , which covers a complete sine wave oscillation
SAMPLES = 1000
np.random.seed(1337)
x_values = np.random.uniform(low=0, high=2*math.pi, size=SAMPLES)
# Shuffle the values to guarantee they're not in order
np.random.shuffle(x_values)
# Calculate the corresponding sine values
y_values = np.sin(x_values)

# Add a small random number to each y value to simulate noise
y_values += 0.1 * np.random.randn(*y_values.shape)

# Split into train/validation/test sets
TRAIN_SPLIT = int(0.6 * SAMPLES)
TEST_SPLIT = int(0.2 * SAMPLES + TRAIN_SPLIT)
x_train, x_validate, x_test = np.split(x_values, [TRAIN_SPLIT, TEST_SPLIT])
y_train, y_validate, y_test = np.split(y_values, [TRAIN_SPLIT, TEST_SPLIT])

# Plot our data points
plt.figure(figsize=(10, 6))
plt.scatter(x_train, y_train, label='Training data')
plt.scatter(x_validate, y_validate, label='Validation data')
plt.scatter(x_test, y_test, label='Test data')
plt.legend()
plt.title('Sine Wave with Random Noise')
plt.xlabel('x values')
plt.ylabel('y values (sine of x + noise)')
plt.show()
```

3.2.2 Constructing and Training the Neural Network Model

Now we will construct a simple neural network to learn the sine function:

```
# Create a model with 2 layers of 16 neurons each
model = tf.keras.Sequential()
# First layer takes a scalar input and feeds it through 16 "neurons"
model.add(layers.Dense(16, activation='relu', input_shape=(1,)))
# Second layer with 16 neurons to capture non-linear relationships
model.add(layers.Dense(16, activation='relu'))
# Final layer is a single neuron for our output value
model.add(layers.Dense(1))
# Compile the model using a standard optimizer and loss function for regression
model.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])

# Display model summary to understand its structure
model.summary()

# Train the model on our data
history = model.fit(x_train, y_train,
                    epochs=500,
                    batch_size=16,
                    validation_data=(x_validate, y_validate),
                    verbose=1)

# Plot the training and validation loss
plt.figure(figsize=(10, 6))
plt.plot(history.history['loss'], label='Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()

# Plot the training and validation mean absolute error
plt.figure(figsize=(10, 6))
plt.plot(history.history['mae'], label='MAE')
plt.plot(history.history['val_mae'], label='Validation MAE')
plt.title('Training and Validation Mean Absolute Error')
plt.xlabel('Epoch')
plt.ylabel('MAE')
plt.legend()
plt.show()

# Evaluate the model on our test data
test_loss, test_mae = model.evaluate(x_test, y_test)
print(f'Test Loss: {test_loss:.4f}')
print(f'Test MAE: {test_mae:.4f}')

# Generate predictions across the full range for visualization
x_dense = np.linspace(0, 2*math.pi, 200)
y_dense_true = np.sin(x_dense)
y_dense_pred = model.predict(x_dense)

# Plot the true sine curve against our model's predictions
plt.figure(figsize=(10, 6))
plt.plot(x_dense, y_dense_true, 'b-', label='True Sine')
plt.plot(x_dense, y_dense_pred, 'r-', label='Model Prediction')
plt.scatter(x_test, y_test, alpha=0.3, label='Test Data')
plt.title('Sine Wave Prediction')
plt.xlabel('x')
plt.ylabel('sin(x)')
plt.legend()
plt.show()
```

This architecture, though simple, is carefully designed to capture the nonlinear relationship of the sine function. The ReLU (Rectified Linear Unit) activation function is particularly important as it introduces nonlinearity:

$$\text{ReLU}(x) = \max(0, x)$$

We train the model using the mean squared error loss function, which for a regression problem is defined as:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Where y_i represents the actual sine value and $\hat{y}_i$ represents our model's prediction.

3.3 Optimizing for Microcontroller Deployment

3.3.1 Model Conversion and Quantization for TensorFlow Lite

To deploy our trained model to the EFR32MG24 microcontroller, we must convert it into a format suitable for resource-constrained devices:

```
# Convert the model to the TensorFlow Lite format without quantization
converter = tf.lite.TFLiteConverter.from_keras_model(model)
tflite_model = converter.convert()

# Save the model to disk
```

```
with open("sine_model.tflite", "wb") as f:
    f.write(tflite_model)

# Convert with quantization for further optimization
converter = tf.lite.TFLiteConverter.from_keras_model(model)
converter.optimizations = [tf.lite.Optimize.DEFAULT]

# Define a generator function that provides our test data's x values
# as a representative dataset
def representative_dataset_generator():
  for value in x_test:
    yield [np.array(value, dtype=np.float32, ndmin=2)]

converter.representative_dataset = representative_dataset_generator
tflite_model_quantized = converter.convert()

# Save the quantized model to disk
with open("sine_model_quantized.tflite", "wb") as f:
    f.write(tflite_model_quantized)

# Print the size reduction achieved through quantization
print(f"Original model size: {len(tflite_model)} bytes")
print(f"Quantized model size: {len(tflite_model_quantized)} bytes")
print(f"Size reduction: {(1 - len(tflite_model_quantized) / len(tflite_model)) * 100:.2f}%")
```

3.3.2 Converting to C Code for Embedded Systems

For deployment on microcontrollers like the EFR32MG24, we need to convert our quantized model into a C header file that can be directly included in our firmware:

```
# Function to convert the model to a C array
def convert_tflite_to_c_array(tflite_model, array_name):
    hex_data = ["0x{:02x}".format(byte) for byte in tflite_model]
    c_array = f"const unsigned char {array_name}[] = {{\n"

    # Format the hex data into rows
    chunk_size = 12
    for i in range(0, len(hex_data), chunk_size):
        c_array += "  " + ", ".join(hex_data[i:i+chunk_size]) + ",\n"

    c_array = c_array[:-2] + "\n};\n"
    c_array += f"const unsigned int {array_name}_len = {len(tflite_model)};\n"

    return c_array

# Generate the C array for our model
c_array = convert_tflite_to_c_array(tflite_model_quantized, "g_sine_model_data")
```

```
# Save to a header file
with open("sine_model_data.h", "w") as f:
    f.write("#ifndef SINE_MODEL_DATA_H_\n")
    f.write("#define SINE_MODEL_DATA_H_\n\n")
    f.write("#include <stdint.h>\n\n")
    f.write(c_array)
    f.write("\n#endif  // SINE_MODEL_DATA_H_\n")

print("C header file generated: sine_model_data.h")

# Download the files
from google.colab import files
files.download("sine_model.tflite")
files.download("sine_model_quantized.tflite")
files.download("sine_model_data.h")
```

3.4 Deploying with Simplicity Studio and Gecko SDK

Now that we have our trained model in a format suitable for microcontrollers, we will implement the TinyML application using Simplicity Studio and the Gecko SDK. This approach simplifies development by providing a structured framework for EFR32 devices.

3.4.1 Creating a New Project in Simplicity Studio

To create a new project in Simplicity Studio, start by launching the application and connecting your EFR32MG24 development board to your computer. Once connected, select your device in the "Debug Adapters" view. Next, click on "Create New Project" in the Launcher perspective to begin the project creation process. When the project wizard opens, select "Silicon Labs Project Wizard" and click "Next" to proceed.

In the following screen, choose "Gecko SDK" as the project type. To find the appropriate template, filter for "example" and select the "TensorFlow Lite Micro Example" template, which provides the necessary framework for our TinyML application. For the project configuration, name your project `sine_wave_predictor` and ensure you are using the latest SDK version. After reviewing your settings, click "Next" and then "Finish" to create the project.

3.4.2 Project Structure and Important Files

Simplicity Studio creates a project with several important files that form the foundation of our TinyML application. The file **app.c** serves as the main application entry point where our program execution begins. The files **sl_tflite_micro_model.{h,c}** contain the TensorFlow Lite Micro integration code that enables machine learning inference on the microcontroller. For controlling the LED output based on our predictions, the files **sl_pwm.{h,c}** provide PWM control functionality. Finally, **sine_model_data.h** contains the model data, which we will replace with our trained model from the previous steps.

3.4.3 Adding Our Trained Model

To incorporate our trained model into the project, we need to import the model data file we generated earlier. In Simplicity Studio, locate the project's `inc` folder in the project explorer. Right-click on this folder and select "Import" from the context menu, then navigate to "General" and choose "File System". In the import dialog, browse to the location where you saved the `sine_model_data.h` file that contains your trained sine wave model. Select this file and click "Finish" to complete the import process, which will add your trained model to the project structure.

3.4.4 Implementing the Application Logic

Now we'll modify the application code to use our sine wave model. Open `app.c` and replace its contents with the following:

```
/***************************************************************************//**
 * @file app.c
 * @brief TinyML Sine Wave Predictor application
 *******************************************************************************
 * # License
 * <b>Copyright 2023 Silicon Laboratories Inc. www.silabs.com</b>
 *******************************************************************************
 *
 * SPDX-License-Identifier: Zlib
 *
 * The licensor of this software is Silicon Laboratories Inc.
 *
 * This software is provided 'as-is', without any express or implied
 * warranty. In no event will the authors be held liable for any damages
 * arising from the use of this software.
 *
 * Permission is granted to anyone to use this software for any purpose,
 * including commercial applications, and to alter it and redistribute it
 * freely, subject to the following restrictions:
 *
 * 1. The origin of this software must not be misrepresented; you must not
 *    claim that you wrote the original software. If you use this software
 *    in a product, an acknowledgment in the product documentation would be
 *    appreciated but is not required.
 * 2. Altered source versions must be plainly marked as such, and must not be
 *    misrepresented as being the original software.
 * 3. This notice may not be removed or altered from any source distribution.
 *
 ******************************************************************************/
#include "sl_component_catalog.h"
#include "sl_system_init.h"
#include "app.h"
#if defined(SL_CATALOG_POWER_MANAGER_PRESENT)
#include "sl_power_manager.h"
#endif
#include "sl_system_process_action.h"

#include <stdio.h>
```

```c
#include <math.h>

#include "sl_tflite_micro_model.h"
#include "sl_led.h"
#include "sl_pwm.h"
#include "sl_sleeptimer.h"

// Constants for sine wave demonstration
#define INFERENCES_PER_CYCLE  32
#define X_RANGE               (2.0f * 3.14159265359f)  // 2  radians
#define PWM_FREQUENCY_HZ      10000
#define INFERENCE_INTERVAL_MS 50

// Global variables
static int inference_count = 0;

void app_init(void)
{
  // Initialize TFLite model
  sl_status_t status = sl_tflite_micro_init();
  if (status != SL_STATUS_OK) {
    printf("Failed to initialize TensorFlow Lite Micro\n");
    return;
  }

  // Initialize PWM for LED control
  sl_pwm_config_t pwm_config = {
    .frequency = PWM_FREQUENCY_HZ,
    .polarity = SL_PWM_ACTIVE_HIGH
  };

  sl_pwm_init(SL_PWM_LED0, &pwm_config);

  printf("Sine Wave Predictor initialized\n");
}

void app_process_action(void)
{
  // Calculate x value based on our position in the cycle
  float position = (float)inference_count / (float)INFERENCES_PER_CYCLE;
  float x_val = position * X_RANGE;

  // Prepare the input tensor with our x value
  float input_data[1] = { x_val };
  sl_tflite_micro_tensor_t input_tensor;
  sl_status_t status = sl_tflite_micro_get_input_tensor(0, &input_tensor);
  if (status != SL_STATUS_OK) {
    printf("Failed to get input tensor\n");
    return;
  }

  // Copy our input data to the input tensor
  status = sl_tflite_micro_set_tensor_data(
    &input_tensor, input_data, sizeof(input_data));

  if (status != SL_STATUS_OK) {
    printf("Failed to set input tensor data\n");
    return;
  }

  // Run inference
  status = sl_tflite_micro_invoke();
  if (status != SL_STATUS_OK) {
    printf("Inference failed\n");
    return;
  }

  // Get the output tensor
  sl_tflite_micro_tensor_t output_tensor;
  status = sl_tflite_micro_get_output_tensor(0, &output_tensor);
  if (status != SL_STATUS_OK) {
    printf("Failed to get output tensor\n");
    return;
  }

// Get the predicted sine value
float predicted_sine = 0.0f;
status = sl_tflite_micro_get_tensor_data(
  &output_tensor, &predicted_sine, sizeof(predicted_sine));

if (status != SL_STATUS_OK) {
  printf("Failed to get output tensor data\n");
  return;
}



  // Map the sine value (-1 to 1) to PWM duty cycle (0 to 100%)
  uint8_t duty_cycle = (uint8_t)((predicted_sine + 1.0f) * 50.0f);

  // Set LED brightness using PWM
  sl_pwm_set_duty_cycle(SL_PWM_LED0, duty_cycle);

  // Log the values (only every 8th inference to reduce console traffic)
  if (inference_count % 8 == 0) {
    printf("x: %.3f, predicted sine: %.3f, duty cycle: %d%%\n",
           x_val, predicted_sine, duty_cycle);
  }

  // Increment the inference counter
  inference_count++;
  if (inference_count >= INFERENCES_PER_CYCLE) {
    inference_count = 0;
  }

  // Add a delay before the next inference
  sl_sleeptimer_delay_millisecond(INFERENCE_INTERVAL_MS);
}
```

3.4.5 Creating the Model Integration File

Create a new file called `sl_tflite_micro_model.c` in the `src` folder with the following content:

```
#include "sl_tflite_micro_model.h"
#include "sine_model_data.h"
#include <stdio.h>

// TensorFlow Lite for Microcontrollers components
#include "tensorflow/lite/micro/all_ops_resolver.h"
#include "tensorflow/lite/micro/micro_error_reporter.h"
#include "tensorflow/lite/micro/micro_interpreter.h"
#include "tensorflow/lite/schema/schema_generated.h"
#include "tensorflow/lite/version.h"

// Global variables
static tflite::MicroErrorReporter micro_error_reporter;
static tflite::ErrorReporter* error_reporter = &micro_error_reporter;
static const tflite::Model* model = nullptr;
static tflite::MicroInterpreter* interpreter = nullptr;
static TfLiteTensor* input_tensor = nullptr;
static TfLiteTensor* output_tensor = nullptr;

// Create an area of memory for input, output, and intermediate arrays
constexpr int kTensorArenaSize = 8 * 1024;
static uint8_t tensor_arena[kTensorArenaSize];

sl_status_t sl_tflite_micro_init(void)
{
  // Map the model into a usable data structure
  model = tflite::GetModel(g_sine_model_data);
  if (model->version() != TFLITE_SCHEMA_VERSION) {
    printf(
"Model version mismatch: %d vs %d\n",
           model->version(), TFLITE_SCHEMA_VERSION);

    return SL_STATUS_FAIL;
  }

  // Create an all operations resolver
  static tflite::AllOpsResolver resolver;

  // Build an interpreter to run the model
  static tflite::MicroInterpreter static_interpreter(
    model, resolver, tensor_arena, kTensorArenaSize, error_reporter);
  interpreter = &static_interpreter;

  // Allocate memory for all tensors
  TfLiteStatus allocate_status = interpreter->AllocateTensors();
  if (allocate_status != kTfLiteOk) {
    printf("AllocateTensors() failed\n");
    return SL_STATUS_ALLOCATION_FAILED;
  }

  // Get pointers to the model's input and output tensors
  input_tensor = interpreter->input(0);
  output_tensor = interpreter->output(0);

  // Check that input and output tensors are the expected size and type
  if (input_tensor->dims->size != 2 || input_tensor->dims->data[0] != 1 ||
      input_tensor->dims->data[1] != 1 || input_tensor->type != kTfLiteFloat32) {
    printf("Unexpected input tensor format\n");
    return SL_STATUS_INVALID_PARAMETER;
  }

  if (output_tensor->dims->size != 2 || output_tensor->dims->data[0] != 1 ||
      output_tensor->dims->data[1] != 1 || output_tensor->type != kTfLiteFloat32) {
    printf("Unexpected output tensor format\n");
    return SL_STATUS_INVALID_PARAMETER;
  }

  printf("TensorFlow Lite Micro initialized successfully\n");
  return SL_STATUS_OK;
}

sl_status_t sl_tflite_micro_get_input_tensor(
    uint8_t index,
    sl_tflite_micro_tensor_t* tensor)
{
  if (
      interpreter == nullptr ||
      index >= interpreter->inputs_size()) {
    return SL_STATUS_INVALID_PARAMETER;
  }


  tensor->tensor = interpreter->input(index);
  return SL_STATUS_OK;
}

sl_status_t sl_tflite_micro_get_output_tensor(
    uint8_t index,
    sl_tflite_micro_tensor_t* tensor)
{
  if (
      interpreter == nullptr ||
      index >= interpreter->outputs_size()) {
    return SL_STATUS_INVALID_PARAMETER;
  }


  tensor->tensor = interpreter->output(index);
  return SL_STATUS_OK;
}

sl_status_t sl_tflite_micro_set_tensor_data(sl_tflite_micro_tensor_t* tensor,
                                            const void* data,
                                            size_t size)
{
  if (tensor == nullptr || tensor->tensor == nullptr || data == nullptr) {
    return SL_STATUS_NULL_POINTER;
  }

  // Size check based on tensor type and dims
  size_t tensor_size = 1;
  for (int i = 0; i < tensor->tensor->dims->size; i++) {
    tensor_size *= tensor->tensor->dims->data[i];
  }

  if (tensor->tensor->type == kTfLiteFloat32) {
    tensor_size *= sizeof(float);
  } else if (tensor->tensor->type == kTfLiteInt8) {
    tensor_size *= sizeof(int8_t);
  } else if (tensor->tensor->type == kTfLiteUInt8) {
    tensor_size *= sizeof(uint8_t);
  } else {
    return SL_STATUS_NOT_SUPPORTED;
  }

  if (size > tensor_size) {
    return SL_STATUS_WOULD_OVERFLOW;
  }

  // Copy the data to the tensor
  memcpy(tensor->tensor->data.raw, data, size);
  return SL_STATUS_OK;
}

sl_status_t sl_tflite_micro_get_tensor_data(sl_tflite_micro_tensor_t* tensor,
                                            void* data,
                                            size_t size)
{
  if (tensor == nullptr || tensor->tensor == nullptr || data == nullptr) {
    return SL_STATUS_NULL_POINTER;
  }

  // Size check based on tensor type and dims
  size_t tensor_size = 1;
  for (int i = 0; i < tensor->tensor->dims->size; i++) {
    tensor_size *= tensor->tensor->dims->data[i];
  }

  if (tensor->tensor->type == kTfLiteFloat32) {
    tensor_size *= sizeof(float);
  } else if (tensor->tensor->type == kTfLiteInt8) {
    tensor_size *= sizeof(int8_t);
  } else if (tensor->tensor->type == kTfLiteUInt8) {
    tensor_size *= sizeof(uint8_t);
  } else {
    return SL_STATUS_NOT_SUPPORTED;
  }

  if (size > tensor_size) {
    return SL_STATUS_WOULD_OVERFLOW;
  }

  // Copy the data from the tensor
  memcpy(data, tensor->tensor->data.raw, size);
  return SL_STATUS_OK;
}

sl_status_t sl_tflite_micro_invoke(void)
{
  if (interpreter == nullptr) {
    return SL_STATUS_NOT_INITIALIZED;
  }

  TfLiteStatus status = interpreter->Invoke();
  if (status != kTfLiteOk) {
    return SL_STATUS_FAIL;
  }

  return SL_STATUS_OK;
}
```

Now, create the header file `sl_tflite_micro_model.h` in the `inc` folder:

```
#ifndef SL_TFLITE_MICRO_MODEL_H
#define SL_TFLITE_MICRO_MODEL_H

#include "sl_status.h"
#include <stdint.h>
#include <stddef.h>

#ifdef __cplusplus
extern "C" {
#endif

// Forward declarations from TensorFlow Lite
#ifdef __cplusplus
namespace tflite {
struct TfLiteTensor;
}  // namespace tflite
typedef struct tflite::TfLiteTensor TfLiteTensor;
#else
typedef struct TfLiteTensor TfLiteTensor;
#endif

// Tensor structure
typedef struct {
  TfLiteTensor* tensor;
} sl_tflite_micro_tensor_t;

/**
 * @brief Initialize TensorFlow Lite Micro with the sine model
 *
 * @return sl_status_t SL_STATUS_OK if successful
 */
sl_status_t sl_tflite_micro_init(void);

/**
 * @brief Get an input tensor by index
 *
 * @param index  Index of the input tensor
 * @param tensor Pointer to the tensor structure to fill
 * @return       sl_status_t SL_STATUS_OK if successful
 */
sl_status_t sl_tflite_micro_get_input_tensor(
    uint8_t index,
    sl_tflite_micro_tensor_t* tensor);

/**
 * @brief Get an output tensor by index
 *
 * @param index  Index of the output tensor
 * @param tensor Pointer to the tensor structure to fill
 * @return       sl_status_t SL_STATUS_OK if successful
 */
sl_status_t sl_tflite_micro_get_output_tensor(
    uint8_t index,
    sl_tflite_micro_tensor_t* tensor);


/**
 * @brief Set data to a tensor
 *
 * @param tensor Pointer to the tensor
 * @param data Pointer to the data to copy
 * @param size Size of the data in bytes
 * @return sl_status_t SL_STATUS_OK if successful
 */
sl_status_t sl_tflite_micro_set_tensor_data(sl_tflite_micro_tensor_t* tensor,
                                          const void* data,
                                          size_t size);

/**
 * @brief Get data from a tensor
 *
 * @param tensor Pointer to the tensor
 * @param data Pointer to the buffer to receive the data
 * @param size Size of the buffer in bytes
 * @return sl_status_t SL_STATUS_OK if successful
 */
sl_status_t sl_tflite_micro_get_tensor_data(sl_tflite_micro_tensor_t* tensor,
                                          void* data,
                                          size_t size);

/**
 * @brief Run inference using the TensorFlow Lite model
 *
 * @return sl_status_t SL_STATUS_OK if successful
 */
sl_status_t sl_tflite_micro_invoke(void);

#ifdef __cplusplus
}
#endif

#endif // SL_TFLITE_MICRO_MODEL_H
```

3.4.6 Building and Flashing the Application

To build and deploy your TinyML application to the EFR32MG24 device, begin by right-clicking on the project in Simplicity Studio and selecting “Build Project” from the context menu. This action compiles your code and prepares it for deployment. After the compilation completes successfully, right-click on the project again and navigate to “Run As”, then select “Silicon Labs ARM Program”. This process will flash the compiled application to your EFR32MG24 device and automatically start its execution.

3.4.7 Observing the Results

Once your application is running on the EFR32MG24 device, you can observe the TinyML model in action through several indicators. The LED on your development board will pulse with brightness that follows the sine wave pattern predicted by your neural network model. To view detailed information about the model’s operation, open the Serial Console in Simplicity Studio, which provides real-time debug output. In the console, you will see logs displaying the input value being fed to the model, the predicted sine value output, and the corresponding LED duty cycle that controls the brightness.

3.5 How it Works: Understanding the Implementation

Our TinyML sine wave predictor consists of several key components that work together to demonstrate machine learning on a microcontroller. The process begins with model training and conversion, where we used Google Colab to train a small neural network to approximate the sine function. We then converted this trained model to TensorFlow Lite format and finally transformed it into a C array suitable for embedded deployment.

The TensorFlow Lite Micro integration provides the inference engine for our application. We have implemented a simple wrapper around TensorFlow Lite Micro’s C++ API, which provides a clean C interface that our application can use easily. This wrapper handles the complexity of model initialization and inference operations.

The application logic operates in a continuous loop that performs several steps. First, it calculates an x value based on the current position in the sine wave cycle. This value is then fed into the trained model as input. The model processes this input and returns a predicted sine value. Finally, the application maps this prediction to LED brightness using PWM control, creating a visual representation of the sine wave.

The visual output serves as immediate feedback for our TinyML implementation. The LED brightness follows a sine wave pattern, providing visual confirmation that our model is working correctly and demonstrating the successful deployment of machine learning on the microcontroller.

3.6 Extending the TinyML Application

Now that we have our basic "Hello World" TinyML application running, there are several ways we can extend and enhance it:

3.6.1 1. Adding Multiple LED Support

For devices with multiple LEDs, we can create more interesting visual patterns by controlling multiple LEDs based on different phases of the sine wave:

```
// In app.c, add phase offsets for each LED
#define LED_COUNT 4  // Assuming 4 available LEDs
const float phase_offsets[LED_COUNT] = {
  0.0f,                  // LED0: No phase offset
  0.5f * X_RANGE / 4.0f, // LED1: 45 degrees offset
  X_RANGE / 4.0f,        // LED2: 90 degrees offset
  1.5f * X_RANGE / 4.0f  // LED3: 135 degrees offset
};

// Then in app_process_action(), add a loop to control all LEDs
for (int i = 0; i < LED_COUNT; i++) {
  // Calculate offset x value for this LED
  float led_x_val = x_val + phase_offsets[i];
  if (led_x_val >= X_RANGE) {
    led_x_val -= X_RANGE;  // Wrap around to stay in range
  }

  // Prepare input tensor with our x value
  float input_data[1] = { led_x_val };
  sl_tflite_micro_tensor_t input_tensor;
  status = sl_tflite_micro_get_input_tensor(0, &input_tensor);
  if (status != SL_STATUS_OK) continue;

  // Set tensor data, invoke model, and get output as before...

  // Set corresponding LED brightness
  uint8_t duty_cycle = (uint8_t)((predicted_sine + 1.0f) * 50.0f);
  sl_pwm_set_duty_cycle(i, duty_cycle);  // Assuming LED PWM instances are indexed
}
```

3.6.2 2. Adding LCD Display Support

If your EFR32MG24 development board includes an LCD display, you can visualize the sine wave more directly:

```
// In app.c, add LCD-related includes
#include "sl_glib.h"
#include "sl_simple_lcd.h"
```

```
// Add LCD dimensions and buffers
#define LCD_WIDTH  128
#define LCD_HEIGHT 64
#define HISTORY_LENGTH LCD_WIDTH
static float sine_history[HISTORY_LENGTH];
static int history_index = 0;

// Initialize the LCD in app_init()
sl_simple_lcd_init();
sl_glib_init();
// Clear history buffer
for (int i = 0; i < HISTORY_LENGTH; i++) {
  sine_history[i] = 0.0f;
}

// In app_process_action(), after getting the predicted sine:
// Store in history buffer
sine_history[history_index] = predicted_sine;
history_index = (history_index + 1) % HISTORY_LENGTH;

// Every few inferences, update the display
if (inference_count % 4 == 0) {
  GLIB_Context_t context;
  sl_glib_get_context(&context);

  // Clear display
  GLIB_clear(&context);

  // Draw x and y axes
  GLIB_drawLineH(&context, 0, LCD_WIDTH-1, LCD_HEIGHT/2);
  GLIB_drawLineV(&context, 0, 0, LCD_HEIGHT-1);

 // Draw the sine wave
for (int i = 0; i < HISTORY_LENGTH - 1; i++) {
  int x1 = i;
  int y1 = (int)(
      LCD_HEIGHT / 2 -
      (sine_history[(history_index + i) % HISTORY_LENGTH] * LCD_HEIGHT / 4));
  int x2 = i + 1;
  int y2 = (int)(
      LCD_HEIGHT / 2 -
      (sine_history[(history_index + i + 1) % HISTORY_LENGTH] * LCD_HEIGHT / 4));
  GLIB_drawLine(&context, x1, y1, x2, y2);
}


  // Update display
  sl_glib_update_display();
}
```

3.6.3 3. Implementing Power Optimization

To make our TinyML application more power-efficient for battery-powered operation, we can add sleep modes between inferences:

```
// Replace the fixed delay with sleep mode
// Instead of: sl_sleeptimer_delay_millisecond(INFERENCE_INTERVAL_MS);

#if defined(SL_CATALOG_POWER_MANAGER_PRESENT)
  // Schedule next wakeup
  sl_sleeptimer_tick_t ticks = sl_sleeptimer_ms_to_tick(INFERENCE_INTERVAL_MS);
  sl_power_manager_schedule_wakeup(ticks, NULL, NULL);

  // Enter sleep mode
  sl_power_manager_sleep();
#else
  // Fall back to delay if power manager isn't available
  sl_sleeptimer_delay_millisecond(INFERENCE_INTERVAL_MS);
#endif
```

3.6.4 4. Enhanced User Interface with Buttons

We can use the buttons on the development board to control aspects of the application:

```
// Include button support
#include "sl_button.h"
#include "sl_simple_button.h"
#include "sl_simple_button_btn0_config.h"

// Add state variables
static bool paused = false;
static float speed_factor = 1.0f;

// In app_init, initialize buttons
sl_button_init(&sl_button_btn0);

// Check button state in app_process_action
if (sl_button_get_state(&sl_button_btn0) == SL_SIMPLE_BUTTON_PRESSED) {
  // Toggle pause state
  paused = !paused;
  printf("Application %s\n", paused ? "paused" : "resumed");
}

// Only update inference_count if not paused
if (!paused) {
  inference_count += 1;
  if (inference_count >= INFERENCES_PER_CYCLE) inference_count = 0;
}
```

3.6.5 5. Performance Profiling and Optimization

To understand and optimize the performance of our TinyML application, we can add timing measurements:

```
// Add profiling includes
#include "em_cmu.h"
#include "em_timer.h"

// Setup timer for profiling in app_init()
CMU_ClockEnable(cmuClock_TIMER1, true);
TIMER_Init_TypeDef timerInit = TIMER_INIT_DEFAULT;
TIMER_Init(TIMER1, &timerInit);

// In app_process_action(), measure inference time
// Reset timer
TIMER_CounterSet(TIMER1, 0);

// Start timer
TIMER_Enable(TIMER1, true);

// Run inference
status = sl_tflite_micro_invoke();

// Stop timer and read value
TIMER_Enable(TIMER1, false);
uint32_t ticks = TIMER_CounterGet(TIMER1);

// Convert ticks to microseconds (depends on timer clock)
uint32_t us = ticks / (CMU_ClockFreqGet(cmuClock_TIMER1) / 1000000);

// Log every Nth inference
if (inference_count % 10 == 0) {
  printf("Inference time: %lu microseconds\n", us);
}
```

3.7 Building TinyML Applications with the Gecko SDK

Compared to the traditional TinyML approach with direct TensorFlow Lite for Microcontrollers integration, the Gecko SDK approach offers several advantages for EFR32 developers:

3.7.1 Simplified Project Setup

The Gecko SDK provides a structured approach to project creation with built-in TinyML support that greatly simplifies the development process. Project templates in Simplicity Studio's project wizard include TinyML templates that automatically set up the necessary directory structure, build configuration, and initial code. This eliminates the need for manual configuration and reduces setup time significantly.

The integrated build system handles all aspects of compilation, including compiler flags, library dependencies, and linking. This removes the complexity of creating and maintaining custom Makefiles, allowing developers to focus on their application logic rather than build configuration. Additionally, the Hardware Abstraction Layer (HAL) provides hardware-specific drivers and APIs for peripherals such as PWM, GPIOs, and timers. This abstraction makes it much easier to integrate TinyML functionality with device hardware features.

3.7.2 Streamlined Development Workflow

The development workflow with Simplicity Studio and Gecko SDK follows a clear and efficient process. Begin by using Google Colab or TensorFlow on your computer to train and convert your machine learning models. Next, launch Simplicity Studio and select the TensorFlow Lite Micro example template to create a new project with all necessary configurations.

After project creation, import your model header file containing the trained neural network weights and architecture. Then write your application logic in C to initialize the model, prepare input data, run inference operations, and process the model outputs. The integrated tools in Simplicity Studio allow you to compile the code and flash it directly to your EFR32 device with a simple click. Finally, use the Serial Console and Energy Profiler tools to monitor application behavior, debug issues, and optimize performance.

3.7.3 Hardware-Specific Optimizations

The Gecko SDK includes optimizations specifically designed for the EFR32 platform that enhance TinyML performance. Memory optimization features are carefully tuned for the EFR32's memory architecture, ensuring efficient use of the limited RAM and flash resources. The SDK integrates with the Energy Management Unit (EMU), providing fine-grained control over active, sleep, and deep sleep states. This integration is crucial for battery-powered applications that require long operational life.

Furthermore, the SDK provides direct access to hardware accelerators and peripherals that can significantly enhance TinyML performance. These hardware-specific features allow developers to leverage the full capabilities of the EFR32 platform while maintaining code portability and ease of use.

3.8 Conclusion

The sine wave predictor represents an elegant "Hello World" example of TinyML deployment on the EFR32MG24 platform. While seemingly simple, this implementation encompasses all the key elements of machine learning on microcontrollers. These elements include model design with careful consideration for resource constraints, training and evaluation on standard datasets, quantization and optimization for embedded deployment, seamless integration with Simplicity Studio and Gecko SDK, and hardware output integration through GPIO and PWM capabilities.

Through this foundation, developers can extend to more complex TinyML applications on the EFR32MG24. These applications might include sensor fusion for combining multiple data sources, predictive maintenance for anticipating equipment failures, anomaly detection for identifying unusual patterns, and keyword spotting for voice-activated features. All these applications can operate within a power envelope suitable for long-term battery-powered operation.

The techniques demonstrated here provide a template that can be adapted for more sophisticated machine learning tasks. Model quantization reduces memory requirements while maintaining accuracy.

C code generation enables direct integration with embedded systems. Deployment through Simplicity Studio streamlines the development process. Together, these techniques enable a new class of intelligent edge devices based on the EFR32MG24 platform, bringing machine learning capabilities to resource-constrained environments where traditional approaches would not be feasible.

Bibliography

[1] P. Warden, D. Situnayake, “TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers,” *O’Reilly Media*, 2019.

[2] R. David, J. Duke, A. Jain *et al.*, “TensorFlow Lite Micro: Embedded Machine Learning for TinyML Systems,” *Proceedings of Machine Learning and Systems*, 2021.

[3] C. Banbury, V. Reddi, P. Torelli *et al.*, “MLPerf Tiny Benchmark,” *arXiv preprint arXiv:2106.07597*, 2021.

[4] L. Lai, N. Suda, V. Chandra, “CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs,” *arXiv preprint arXiv:1801.06601*, 2018.

[5] Silicon Labs, “EFR32MG24 Wireless Gecko SoC Family Data Sheet,” *Silicon Labs Technical Documentation*, 2024.

[6] TensorFlow Team, “TensorFlow Lite for Microcontrollers,” *TensorFlow Documentation*, 2024.

[7] B. Jacob, S. Kligys, B. Chen *et al.*, “Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference,” *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018.

[8] J. Lin, W. Chen, Y. Lin *et al.*, “MCUNet: Tiny Deep Learning on IoT Devices,” *Advances in Neural Information Processing Systems*, 2020.

[9] I. Fedorov, R. Adams, M. Mattina *et al.*, “SpArSe: Sparse Architecture Search for CNNs on Resource-Constrained Microcontrollers,” *Advances in Neural Information Processing Systems*, 2019.

[10] ARM Limited, “Arm Cortex-M33 Processor Technical Reference Manual,” *ARM Documentation*, 2023.

[11] A. Gholami, S. Kim, Z. Dong *et al.*, “A Survey of Quantization Methods for Efficient Neural Network Inference,” *arXiv preprint arXiv:2103.13630*, 2022.

[12] R. Krishnamoorthi, “Quantizing Deep Convolutional Networks for Efficient Inference: A Whitepaper,” *arXiv preprint arXiv:1806.08342*, 2018.

[13] Silicon Labs, “Simplicity Studio 5 User’s Guide,” *Silicon Labs Technical Documentation*, 2024.

[14] Silicon Labs, “Gecko SDK Documentation,” *Silicon Labs Developer Documentation*, 2024.

[15] M. Abadi, P. Barham, J. Chen *et al.*, “TensorFlow: A System for Large-Scale Machine Learning,” *12th USENIX Symposium on Operating Systems Design and Implementation*, 2016.

Chapter 4

Building a TinyML Application

In the previous chapter, we trained a neural network model to predict sine wave values and prepared it for deployment on an EFR32MG24 microcontroller. Now we will build a complete application around this model and deploy it to the hardware. This chapter focuses on the practical aspects of implementing TinyML using Silicon Labs' Gecko SDK and Simplicity Studio rather than the traditional TensorFlow Lite for Microcontrollers approach.

4.1 Understanding the Gecko SDK Approach to TinyML

The Gecko SDK provides a structured approach to embedded development specifically optimized for Silicon Labs' microcontrollers. This approach offers several advantages over the more generic TinyML implementations. The SDK includes optimized TensorFlow Lite Micro components already configured for EFR32 devices, providing direct integration with EFR32 peripherals through a consistent API. Furthermore, Simplicity Studio provides starting points for TinyML applications through project templates, while advanced tooling such as debugging, energy profiling, and configuration tools are built into the development environment.

4.2 Setting Up Your Development Environment

Before we begin building our application, ensure you have the following tools installed. First, download and install **Simplicity Studio 5** from Silicon Labs' website. The latest version of the **Gecko SDK** will be installed through Simplicity Studio. **J-Link Drivers** should be installed with Simplicity Studio. Additionally, connect your **EFR32MG24 Development Board** to your computer via USB.

4.3 Creating a New Project in Simplicity Studio

To create a TinyML project in Simplicity Studio, begin by launching Simplicity Studio 5. In the Launcher perspective, click on your connected EFR32MG24 device, then click "Create New Project" in the "Overview" tab. Select "Silicon Labs Project Wizard" and click "NEXT". In the SDK Selection dialog, ensure the latest Gecko SDK is selected and click "NEXT". In the Project Generation dialog, filter for "example" in the search box and select "TensorFlow Lite Micro Example", then click "NEXT". Configure your project by naming it `sine_wave_predictor`, keep the default location, and click "FINISH".

Simplicity Studio will generate a project with the necessary components for a TinyML application. We will explore the project structure before making our modifications.

4.4 Exploring the Project Structure

The generated project includes several important directories and files. The **config/** directory contains hardware configuration files for your specific board, while the **gecko_sdk/** directory contains the Gecko SDK source code, including TensorFlow Lite Micro. The **autogen/** directory contains auto-generated initialization code for the device. The **app.c** file serves as your application's main source file, and the **app.h** file functions as the header file for your application.

The TensorFlow Lite Micro example comes with a sample model that classifies motion patterns. We will replace this with our sine wave model.

4.5 Importing the Sine Wave Model

To import the sine model data that we generated in the previous chapter, right-click on the project in the "Project Explorer" view and select "Import" → "General" → "File System". Browse to the location where you saved `sine_model_data.h`, select the file, and click "Finish".

The model data will be added to your project. Now we will modify the application code to use our sine wave model.

4.6 Implementing the Application

Let's replace the content of `app.c` with our sine wave prediction application code. Open `app.c` and replace its contents with the following:

```
/***************************************************************************//**
 * @file app.c
 * @brief TinyML Sine Wave Predictor application
 *******************************************************************************
 * # License
 * <b>Copyright 2023 Silicon Laboratories Inc. www.silabs.com</b>
 *******************************************************************************
 *
 * SPDX-License-Identifier: Zlib
 *
 * The licensor of this software is Silicon Laboratories Inc.
 *
 * This software is provided 'as-is', without any express or implied
 * warranty. In no event will the authors be held liable for any damages
 * arising from the use of this software.
 *
 * Permission is granted to anyone to use this software for any purpose,
 * including commercial applications, and to alter it and redistribute it
 * freely, subject to the following restrictions:
 *
 * 1. The origin of this software must not be misrepresented; you must not
 *    claim that you wrote the original software. If you use this software
 *    in a product, an acknowledgment in the product documentation would be
 *    appreciated but is not required.
 * 2. Altered source versions must be plainly marked as such, and must not be
 *    misrepresented as being the original software.
 * 3. This notice may not be removed or altered from any source distribution.
 *
 ******************************************************************************/
#include "sl_component_catalog.h"
#include "sl_system_init.h"
#include "app.h"
```

```
#if defined(SL_CATALOG_POWER_MANAGER_PRESENT)
#include "sl_power_manager.h"
#endif
#include "sl_system_process_action.h"

/* Additional includes for our application */
#include <stdio.h>
#include <math.h>

/* Include TensorFlow Lite components */
#include "tensorflow/lite/micro/all_ops_resolver.h"
#include "tensorflow/lite/micro/micro_error_reporter.h"
#include "tensorflow/lite/micro/micro_interpreter.h"
#include "tensorflow/lite/schema/schema_generated.h"
#include "tensorflow/lite/version.h"

/* Include our model data */
#include "sine_model_data.h"

/* Include hardware control components */
#include "sl_led.h"
#include "sl_simple_led_instances.h"
#include "sl_sleeptimer.h"

/* Constants for sine wave demonstration */
#define INFERENCES_PER_CYCLE  32
#define X_RANGE               (2.0f * 3.14159265359f)  /* 2  radians */
#define INFERENCE_INTERVAL_MS 50

/* Global variables for TensorFlow Lite model */
static tflite::MicroErrorReporter micro_error_reporter;
static tflite::ErrorReporter* error_reporter = &micro_error_reporter;
/* We'll use the C version of TensorFlow Lite Micro API */
static tflite::MicroInterpreter* interpreter = nullptr;
static TfLiteTensor* input = nullptr;
static TfLiteTensor* output = nullptr;

/* Create an area of memory for input, output, and intermediate arrays */
#define TENSOR_ARENA_SIZE (10 * 1024)
static uint8_t tensor_arena[TENSOR_ARENA_SIZE];

/* Application state variables */
static int inference_count = 0;

/***************************************************************************//**
 * Initialize application.
 ******************************************************************************/
```

```
void app_init(void)
{
  /* Map the model into a usable data structure */
  model = tflite::GetModel(g_sine_model_data);
  if (model->version() != TFLITE_SCHEMA_VERSION) {
    error_reporter->Report(
        "Model provided is schema version %d not equal "
        "to supported version %d.\n",
        model->version(), TFLITE_SCHEMA_VERSION);
    return;
  }

  /* This pulls in all the operation implementations we need */
  static tflite::AllOpsResolver resolver;

  /* Build an interpreter to run the model with */
  static tflite::MicroInterpreter static_interpreter(
      model, resolver, tensor_arena, TENSOR_ARENA_SIZE, error_reporter);
  interpreter = &static_interpreter;

  /* Allocate memory from the tensor_arena for the model's tensors */
  TfLiteStatus allocate_status = interpreter->AllocateTensors();
  if (allocate_status != kTfLiteOk) {
    error_reporter->Report("AllocateTensors() failed");
    return;
  }

  /* Obtain pointers to the model's input and output tensors */
  input = interpreter->input(0);
  output = interpreter->output(0);

  /* Check that input tensor dimensions are as expected */
  if (input->dims->size != 2 || input->dims->data[0] != 1 ||
      input->dims->data[1] != 1 || input->type != kTfLiteFloat32) {
    error_reporter->Report("Unexpected input tensor dimensions or type");
    return;
  }

  /* Initialize LED */
  sl_led_init(SL_SIMPLE_LED_INSTANCE(0));

  /* Print initialization message */
  printf("Sine Wave Predictor initialized successfully!\n");
  printf("Model input dims: %d x %d, type: %d\n",
         input->dims->data[0], input->dims->data[1], input->type);
}
```

```
/*****************************************************************************//**
 * App ticking function.
 ******************************************************************************/
void app_process_action(void)
{
  /* Calculate an x value to feed into the model based on current inference count */
  float position = (float)(inference_count) / (float)(INFERENCES_PER_CYCLE);
  float x_val = position * X_RANGE;

  /* Set the input tensor with our calculated x value */
  input->data.f[0] = x_val;

  /* Run inference, and report any error */
  TfLiteStatus invoke_status = TF_MicroInterpreter_Invoke(interpreter);
  if (invoke_status != kTfLiteOk) {
    printf("Invoke failed on x_val: %f\n", (double)x_val);
    return;
  }

  /* Read the predicted y value from the model's output tensor */
  float y_val = output->data.f[0];

  /* Map the sine output (-1 to 1) to LED brightness
   * For simplicity, we'll just turn the LED on when the value is positive
   * and off when it's negative. For PWM control, you would need to
   * configure a PWM peripheral. */
  if (y_val > 0) {
    sl_led_turn_on(SL_SIMPLE_LED_INSTANCE(0));
  } else {
    sl_led_turn_off(SL_SIMPLE_LED_INSTANCE(0));
  }

  /* Log every 4th inference to avoid flooding the console */
  if (inference_count % 4 == 0) {
    printf("x_value: %f, predicted_sine: %f\n", (double)x_val, (double)y_val);
  }

  /* Increment the inference_count, and reset it if we have reached
   * the total number per cycle */
  inference_count += 1;
  if (inference_count >= INFERENCES_PER_CYCLE) inference_count = 0;

  /* Add a delay between inferences */
  sl_sleeptimer_delay_millisecond(INFERENCE_INTERVAL_MS);
}
```

This application will: 1. Initialize the TensorFlow Lite Micro interpreter with our sine model 2. Set up an LED for output 3. In each loop iteration: - Calculate an x value within our 0 to 2π range - Run inference to get the predicted sine value - Toggle the LED based on whether the sine value is positive or negative - Log the values to the console - Increment the inference counter

4.7 Enhancing Output with PWM Control

The basic application merely toggles an LED, but we can create a more interesting visualization by controlling LED brightness with PWM. To create a PWM component for our project, right-click on your project in the Project Explorer and select "Configure Project". Click on "SOFTWARE COMPONENTS" and in the search box, type "PWM". Find "PWM Driver" → "Simple PWM" and click "Install". Click "Force Install" if prompted, then click "DONE" to save the configuration.

Now, modify the application code to use PWM for LED brightness control. Replace the LED control section in `app_process_action()` with:

```
/* Map the sine output (-1 to 1) to PWM duty cycle (0 to 100%) */
uint8_t duty_cycle = (uint8_t)((y_val + 1.0f) * 50.0f);

/* Set LED brightness using PWM */
sl_pwm_set_duty_cycle(SL_PWM_INSTANCE(0), duty_cycle);
```

Also, add the PWM initialization in the `app_init()` function after the LED initialization:

```
/* Initialize PWM for LED brightness control */
sl_pwm_config_t pwm_config = {
  .frequency = 10000,  /* 10 kHz PWM frequency */
  .polarity = SL_PWM_ACTIVE_HIGH
};
sl_pwm_init(SL_PWM_INSTANCE(0), &pwm_config);
```

Don't forget to include the PWM header at the top of the file:

```
#include "sl_pwm.h"
#include "sl_simple_pwm_instances.h"
```

4.8 Building and Flashing the Application

To build and flash our application to the EFR32MG24 board, right-click on your project in the Project Explorer and select "Build Project". Once the build completes successfully, right-click again on the project and select "Run As" → "Silicon Labs ARM Program". Simplicity Studio will compile your code, flash it to the device, and start execution.

4.9 Debugging and Monitoring

To monitor the output of your application, navigate to the "Debug Adapters" view in Simplicity Studio. Right-click on your connected device and select "Launch Console". In the console dialog, select "Serial 1" and click "OK". You should now see the application's output messages showing x values and predicted sine values.

4.10 Optimizing TinyML Performance

4.10.1 Memory Optimization

TinyML applications on microcontrollers must be memory-efficient. Several methods exist to optimize memory usage. One approach involves reducing the `TENSOR_ARENA_SIZE` to the minimum required. This can be achieved by starting with 10KB and reducing it incrementally:

```
#define TENSOR_ARENA_SIZE (10 * 1024)  /* Start with 10KB */
```

You can determine the minimum required size by adding debug output:

```
/* Add this after interpreter->AllocateTensors() in app_init() */
size_t used_bytes = interpreter->arena_used_bytes();
printf("Model uses %d bytes of tensor arena\n", (int)used_bytes);
```

Another optimization technique involves implementing **Selective Op Resolution**. Instead of using `AllOpsResolver`, create a custom resolver with only the operations needed:

```
/* Replace AllOpsResolver with this */
static tflite::MicroMutableOpResolver<4> resolver;
resolver.AddFullyConnected();
resolver.AddRelu();
resolver.AddAdd();
resolver.AddMul();
```

4.10.2 Power Optimization

Power efficiency represents a fundamental consideration for battery-powered applications. One essential strategy involves implementing sleep functionality between inferences. Rather than utilizing simple delays, the system should employ power-efficient sleep mechanisms to conserve energy during idle periods.

```
/* Replace sl_sleeptimer_delay_millisecond() with: */
#if defined(SL_CATALOG_POWER_MANAGER_PRESENT)
  /* Schedule next wakeup */
  sl_sleeptimer_tick_t ticks = sl_sleeptimer_ms_to_tick(INFERENCE_INTERVAL_MS);
  sl_power_manager_schedule_wakeup(ticks, NULL, NULL);

  /* Enter sleep mode */
  sl_power_manager_sleep();
#else
  /* Fall back to delay if power manager isn't available */
  sl_sleeptimer_delay_millisecond(INFERENCE_INTERVAL_MS);
#endif
```

Furthermore, the measurement of power consumption can be effectively accomplished through the Energy Profiler tool integrated within Simplicity Studio. This process requires establishing a connection between the development board and the Advanced Energy Monitor (AEM). Subsequently, the Energy Profiler can be accessed through the Tools menu within Simplicity Studio. Once initiated, the capture function should be activated while the application executes. This approach enables comprehensive analysis of current consumption patterns during both inference operations and sleep periods.

4.10.3 Timing Performance

To measure inference time:

```
/* Add these includes */
#include "em_cmu.h"
#include "em_timer.h"

/* Initialize timer in app_init() */
CMU_ClockEnable(cmuClock_TIMER0, true);
TIMER_Init_TypeDef timerInit = TIMER_INIT_DEFAULT;
TIMER_Init(TIMER0, &timerInit);

/* In app_process_action(), surround the inference with timing code */
/* Reset and start timer */
TIMER_CounterSet(TIMER0, 0);
TIMER_Enable(TIMER0, true);

/* Run inference */
TfLiteStatus invoke_status = interpreter->Invoke();

/* Stop timer and read counter */
TIMER_Enable(TIMER0, false);
uint32_t ticks = TIMER_CounterGet(TIMER0);

/* Convert ticks to microseconds */
uint32_t us = ticks / (CMU_ClockFreqGet(cmuClock_TIMER0) / 1000000);

/* Log timing information */
if (inference_count % 10 == 0) {
  printf("Inference took %lu microseconds\n", us);
}
```

4.11 Adding User Interaction with Buttons

The incorporation of button-based controls enhances application interactivity by enabling user-directed behavioral modifications. The implementation process begins with the integration of a button component into the existing project structure. This is achieved by accessing the project configuration through the context menu and navigating to the software components section. Within this interface, a search for button functionality reveals the Simple Button components, which should be installed to enable the desired functionality.

Following the component installation, the application code requires modification to accommodate button press event handling.

```
/* Include button headers */
#include "sl_button.h"
#include "sl_simple_button_instances.h"
```

```c
/* In app_init() */
/* Initialize buttons */
sl_button_init(SL_SIMPLE_BUTTON_INSTANCE(0));

/* In app_process_action(), check for button press */
if (sl_button_get_state(SL_SIMPLE_BUTTON_INSTANCE(0)) == SL_SIMPLE_BUTTON_PRESSED) {
  /* Toggle between normal speed and double speed */
  static bool fast_mode = false;
  fast_mode = !fast_mode;

  /* Update the inference interval */
  inference_interval_ms = fast_mode ? 25 : 50;

  printf("Speed set to %s\n", fast_mode ? "FAST" : "NORMAL");
}
```

4.12 Enhanced Visualization with LCD (if available)

Development boards equipped with LCD displays offer opportunities for creating more sophisticated visual representations. The integration of LCD functionality requires the addition of specific components to the project. This process involves accessing the project configuration dialog and conducting a search for LCD-related components. The essential components for this functionality include the GLIB Graphics Library and Simple LCD module, both of which must be installed to enable display capabilities.

Once these components are successfully integrated, the application code can be modified to render the sine wave visualization directly on the LCD display.

```c
/* Include LCD headers */
#include "sl_glib.h"
#include "sl_simple_lcd.h"

/* In app_init() */
/* Initialize LCD */
sl_simple_lcd_init();
sl_glib_initialize();

/* Define a buffer to store recent sine wave values */
#define HISTORY_SIZE 128
static float sine_history[HISTORY_SIZE];
static int history_index = 0;

/* Initialize history buffer */
for (int i = 0; i < HISTORY_SIZE; i++) {
  sine_history[i] = 0.0f;
}

/* In app_process_action(), after getting the prediction */
/* Store the prediction in the history buffer */
sine_history[history_index] = y_val;
history_index = (history_index + 1) % HISTORY_SIZE;

/* Every 8th inference, update the LCD */
if (inference_count % 8 == 0) {
  GLIB_Context_t context;
  sl_glib_get_context(&context);

  /* Clear the display */
  GLIB_clear(&context);

  /* Draw axes */
  int mid_y = context.height / 2;
  GLIB_drawLineH(&context, 0, context.width - 1, mid_y);

  /* Draw the sine wave */
  for (int i = 0; i < HISTORY_SIZE - 1; i++) {
    int x1 = i;
    int y1 = mid_y - (int)(sine_history[(history_index + i
    ) % HISTORY_SIZE] * mid_y * 0.8f);
    int x2 = i + 1;
    int y2 = mid_y - (int)(sine_history[(history_index + i + 1)
    % HISTORY_SIZE] * mid_y * 0.8f);

    GLIB_drawLine(&context, x1, y1, x2, y2);
  }

  /* Update the display */
  GLIB_drawString(&context, "Sine Wave Predictor", 0, 0, GLIB_ALIGN_CENTER, 0);
  GLIB_update(&context);
}
```

4.13 Creating a Custom Component for TinyML

The development of custom Gecko SDK components enhances code reusability within TinyML applications. This approach facilitates modularity and promotes efficient code organization. The implementation process commences with the creation of a header file designated as `sl_tflite_sine_predictor.h`.

```
#ifndef SL_TFLITE_SINE_PREDICTOR_H
#define SL_TFLITE_SINE_PREDICTOR_H

#include "sl_status.h"
#include <stdint.h>

#ifdef __cplusplus
#ifdef __cplusplus
```

```
extern "C" {
#endif
#endif

/**
 * @brief Initialize the TinyML sine predictor
 *
 * @return sl_status_t SL_STATUS_OK on success
 */
sl_status_t sl_tflite_sine_predictor_init(void);

/**
 * @brief Run inference with a given x value
 *
 * @param x_val Input value in range [0, 2 ]
 * @param y_val Pointer to store the predicted sine value
 * @return sl_status_t SL_STATUS_OK on success
 */
sl_status_t sl_tflite_sine_predictor_predict(float x_val, float* y_val);

#ifdef __cplusplus
}
#endif

#endif /* SL_TFLITE_SINE_PREDICTOR_H */
```

Subsequently, an implementation file named `sl_tflite_sine_predictor.c` should be created to provide the corresponding functionality.

```
#include "sl_tflite_sine_predictor.h"
#include "sine_model_data.h"
#include <string.h>

/* TensorFlow Lite components */
#include "third_party/tflite-micro/tensorflow/lite/micro/kernels/micro_ops.h"
#include "third_party/tflite-micro/tensorflow/lite/micro/micro_error_reporter.h"
#include "third_party/tflite-micro/tensorflow/lite/micro/micro_interpreter.h"
#include "third_party/tflite-micro/tensorflow/lite/micro/micro_mutable_op_resolver.h"
#include "third_party/tflite-micro/tensorflow/lite/schema/schema_generated.h"
#include "third_party/tflite-micro/tensorflow/lite/version.h"

/* Static variables for TensorFlow Lite model - C compatible structure */
static TF_MicroErrorReporter micro_error_reporter;
static TF_MicroInterpreter* interpreter = NULL;
static TfLiteTensor* input = NULL;
static TfLiteTensor* output = NULL;

/* Create an area of memory for input, output, and intermediate arrays */
```

```
#define TENSOR_ARENA_SIZE (10 * 1024)
static uint8_t tensor_arena[TENSOR_ARENA_SIZE];

/* C implementation for initialization */
sl_status_t sl_tflite_sine_predictor_init(void)
{
  /* Map the model into a usable data structure */
  const TfLiteModel* model = TfLiteModelCreate(g_sine_model_data,
   g_sine_model_data_len);
  if (model == NULL) {
    return SL_STATUS_FAIL;
  }

  /* Initialize error reporter */
  TF_MicroErrorReporter_Init(&micro_error_reporter);

  /* Create an operation resolver with the operations we need */
  static TfLiteMicroMutableOpResolver op_resolver;
  TfLiteMicroMutableOpResolver_Init(&op_resolver);

  /* Add the operations needed for our model */
  TfLiteMicroMutableOpResolver_AddFullyConnected(&op_resolver);
  TfLiteMicroMutableOpResolver_AddRelu(&op_resolver);
  TfLiteMicroMutableOpResolver_AddMul(&op_resolver);
  TfLiteMicroMutableOpResolver_AddAdd(&op_resolver);

  /* Build an interpreter to run the model */
  static TF_MicroInterpreter static_interpreter;
  TF_MicroInterpreter_Init(
      &static_interpreter, model, &op_resolver, tensor_arena,
      TENSOR_ARENA_SIZE, &micro_error_reporter);
  interpreter = &static_interpreter;

  /* Allocate memory from the tensor_arena for the model's tensors */
  TfLiteStatus allocate_status = TF_MicroInterpreter_AllocateTensors(interpreter);
  if (allocate_status != kTfLiteOk) {
    return SL_STATUS_ALLOCATION_FAILED;
  }

  /* Obtain pointers to the model's input and output tensors */
  input = TF_MicroInterpreter_GetInputTensor(interpreter, 0);
  output = TF_MicroInterpreter_GetOutputTensor(interpreter, 0);

  if (input == NULL || output == NULL) {
    return SL_STATUS_FAIL;
  }
  return SL_STATUS_OK;
}

/* C implementation for prediction */
sl_status_t sl_tflite_sine_predictor_predict(float x_val, float* y_val)
{
  if (interpreter == NULL || input == NULL || output == NULL || y_val == NULL) {
    return SL_STATUS_INVALID_STATE;
  }

  /* Set the input tensor data */
  input->data.f[0] = x_val;

  /* Run inference */
  TfLiteStatus invoke_status = TF_MicroInterpreter_Invoke(interpreter);
  if (invoke_status != kTfLiteOk) {
    return SL_STATUS_FAIL;
  }

  /* Get the output value */
  *y_val = output->data.f[0];

  return SL_STATUS_OK;
}
```

Following the creation of these files, the `app.c` file requires modification to utilize this newly developed component.

```
#include "sl_tflite_sine_predictor.h"

/* In app_init() */
sl_status_t status = sl_tflite_sine_predictor_init();
if (status != SL_STATUS_OK) {
  printf("Failed to initialize TinyML model: %d\n", (int)status);
  return;
}

/* In app_process_action() */
float x_val = position * X_RANGE;
float y_val = 0.0f;

/* Run inference */
status = sl_tflite_sine_predictor_predict(x_val, &y_val);
if (status != SL_STATUS_OK) {
  printf("Inference failed: %d\n", (int)status);
  return;
}
```

This approach encapsulates the TensorFlow Lite components behind a C API, making it easier to use throughout your application.

4.14 Conclusion

This chapter has demonstrated the development of a comprehensive TinyML application for the EFR32MG24 platform utilizing the Gecko SDK and Simplicity Studio. This approach offers significant advantages through the simplification of deployment processes, achieved by leveraging the hardware abstraction layer and pre-integrated components provided by the SDK.

The implementation has encompassed several fundamental aspects of TinyML development. The utilization of Simplicity Studio's project templates enables rapid establishment of a TinyML environment. The integration of pre-trained TensorFlow Lite models with the application framework has been successfully demonstrated. Furthermore, the visualization of model predictions has been achieved through various output methods, including LED brightness modulation and LCD display rendering. The implementation has also addressed critical considerations of memory and power optimization, alongside comprehensive performance measurement and improvement strategies. Additionally, the creation of reusable components for TinyML functionality enhances the modularity and maintainability of the developed solution.

This foundational example provides a robust starting point for the development of more sophisticated TinyML applications on the EFR32 platform. Future extensions may encompass the implementation of advanced models such as keyword spotting, gesture recognition, or anomaly detection systems. The integration of sensors for real-time data collection presents additional opportunities for enhancement. Moreover, the development of custom hardware interfaces can facilitate alternative output methods, while multi-model systems offer the potential to combine diverse machine learning capabilities within a single application.

The Gecko SDK approach significantly enhances the accessibility of these advanced implementations by providing a structured and optimized framework specifically engineered for Silicon Labs devices.

Bibliography

[1] Silicon Laboratories, "Gecko SDK Documentation and Reference Guide," *Silicon Laboratories Inc.*, 2024.

[2] TensorFlow Authors, "TensorFlow Lite for Microcontrollers," *Google LLC*, 2024.

[3] P. Warden, D. Situnayake, "TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers," *O'Reilly Media*, 2019.

[4] V. J. Reddi, B. Plancher, S. Kennedy, L. Moroney, P. Warden, A. Anand, et al., "Widening Access to Applied Machine Learning with TinyML," *Harvard Data Science Review*, 2021.

[5] R. David, J. Duke, A. Jain, V. J. Reddi, N. Jeffries, J. Li, et al., "TensorFlow Lite Micro: Embedded Machine Learning on TinyML Systems," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 800-811, 2021.

[6] Silicon Laboratories, "EFR32MG24 Wireless Gecko SoC Family Data Sheet," *Silicon Laboratories Inc.*, 2024.

[7] C. Banbury, V. J. Reddi, P. Torelli, J. Holleman, N. Jeffries, C. Kiraly, et al., "MLPerf Tiny Benchmark," *arXiv preprint arXiv:2106.07597*, 2021.

[8] Silicon Laboratories, "Simplicity Studio 5 User's Guide," *Silicon Laboratories Inc.*, 2024.

[9] P. P. Ray, "A Review on TinyML: State-of-the-art and Prospects," *Journal of King Saud University - Computer and Information Sciences*, vol. 34, no. 4, pp. 1595-1623, 2022.

[10] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, et al., "TensorFlow: A System for Large-Scale Machine Learning," *12th USENIX Symposium on Operating Systems Design and Implementation*, pp. 265-283, 2016.

Chapter 5

Handwriting Digit Recognition

This work was initially done by Ali Kia and later extended for this book chapter.

5.1 Chapter Objectives

This chapter shows how a convolutional neural network (CNN) trained on the MNIST dataset can be compressed through post-training quantization and executed on the EFR32xG24 microcontroller. Accuracy, model size, and inference latency are reported, and practical optimisation techniques for TinyML deployment are discussed.

5.2 Overview and Challenges

A handwriting-recognition system was implemented on the Silicon Labs EFR32xG24, which provides a 78 MHz ARM Cortex-M33, 256 kB of RAM, and 1.5 MB of flash. The TensorFlow-based CNN attains 99.18% test accuracy. After 8-bit quantization the model occupies $101.59\,\mathrm{kB}$, representing a $91\,\%$ reduction, yet still satisfies real-time requirements. These results confirm that vision tasks can run entirely on microcontroller-class devices, enabling privacy-preserving and low-latency interaction for smart interfaces and IoT sensors. Key implementation challenges and optimisation steps are summarised in the following sections.

TinyML brings machine-learning inference to resource-constrained hardware, improving privacy, reducing latency, and lowering the energy cost of wireless communication. Handwriting recognition offers a compact, well-studied problem that highlights the benefits of on-device processing. Successful deployment enables applications such as offline note-taking and local user authentication that do not require cloud services.

Running neural networks on the EFR32xG24 demands careful memory and compute budgeting. The limited RAM and flash exclude conventional 32-bit models; every layer must be parameter-efficient, and all weights and activations must be quantised. In addition, inference must complete within a few milliseconds per image to maintain responsive user interaction.

5.3 Background

Chapters 1–4 introduced the TinyML toolchain and the TensorFlow Lite for Microcontrollers runtime, which form the basis for this work. Prior studies have already demonstrated wake-word spotting, anomaly detection, and gesture recognition on similar hardware. Warden and Situnayake (2020) provide a methodology that this implementation follows.

CNNs extract hierarchical features and exhibit translation invariance, properties that make them effective for handwriting recognition. The MNIST dataset, with $60\,000$ training and $10\,000$ test images at $28 \times 28\,\mathrm{px}$, serves as the standard benchmark and enables direct comparison of embedded implementations.

Post-training 8-bit quantization compresses the network while preserving accuracy. A lightweight architecture inspired by MobileNet depthwise-separable convolutions further reduces the parameter count. Additional techniques such as pruning or knowledge distillation, highlighted by Banbury et al. (2021), could shrink the footprint even more but were not needed to meet the memory budget in this study. The resulting trade-offs among accuracy, footprint, and latency are examined in this chapter.

5.4 Methodology

5.4.1 System Architecture

The system adopts a modular design that fits the memory and compute limits of the EFR32xG24. It receives 28×28 px grayscale images and sends them through several cooperating components.

TensorFlow Lite for Microcontrollers acts as the execution engine for the 8-bit CNN. It handles memory allocation and schedules each operation within a 70 kB tensor arena that stores all intermediate tensors during inference.

An input module converts raw images from test arrays or external sensors into the network's expected format. After inference, an output parser selects the digit with the highest probability and reports both the class and its confidence. The result travels over a USART or EUSART link for external logging and evaluation.

Clear separation of these tasks allows each module to be tuned independently, which simplifies future optimisation or feature additions.

5.4.2 Model Design and Training

5.4.2.1 Dataset Preparation

The MNIST corpus supplies 70 000 grayscale digit images at 28×28 px, of which 60 000 form the training set and 10 000 serve as the test set. Each image is reshaped to $(28, 28, 1)$, cast to `float32`, and scaled to the range $[0, 1]$. The preprocessing routine remains as follows:

```
# Load dataset
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

# Reshape and normalize
train_images = train_images.reshape((-1, 28, 28, 1)).astype('float32') / 255.0
test_images = test_images.reshape((-1, 28, 28, 1)).astype('float32') / 255.0
```

5.4.2.2 CNN Architecture

A compact network with three convolutional blocks and two dense layers strikes a balance between accuracy and parameter count, a requirement for microcontroller deployment. Table 5.4.2.2 summarises the topology.

Table 1: CNN Model Architecture

Layer Type	Parameters	Output Shape
Input	–	(28, 28, 1)
Conv2D	3×3, 32 filters, ReLU	320
MaxPooling2D	2×2	0
Conv2D	3×3, 64 filters, ReLU	18 496

Layer Type	Parameters	Output Shape
MaxPooling2D	2×2	0
Conv2D	3×3, 64 filters, ReLU	36 928
Flatten	–	0
Dense	64 units, ReLU	36 928
Dense	10 units, Softmax	650

The architecture is implemented with Keras:

```
model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax')
])
```

5.4.2.3 Training Configuration

The model was trained with the following configurations:

```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(train_images, train_labels, epochs=5)
```

Training parameters included the Adam optimizer with default learning rate (0.001), sparse categorical cross-entropy loss function, accuracy metrics, 5 epochs, and default batch size (32). The relatively small number of epochs was sufficient due to the simplicity of the MNIST dataset and the model's efficient learning capacity. Training was performed in Google Colab to leverage GPU acceleration.

5.4.3 Model Optimization

5.4.3.1 Post-Training Quantization

To meet the memory constraints of the EFR32xG24 microcontroller, the trained model was subjected to post-training quantization using TensorFlow Lite's quantization framework. This process converted the 32-bit floating-point weights and activations to 8-bit integers, significantly reducing the model size while preserving accuracy.

The quantization process required defining a representative dataset to calibrate the dynamic range of activations:

```
def representative_data_gen():
    """Generator function for a representative dataset for quantization."""
    for input_value in tf.data.Dataset.from_tensor_slices(train_images).batch(1
    ).take(100):
        yield [tf.cast(input_value, tf.float32)]

# Configure the converter for full integer quantization
converter = tf.lite.TFLiteConverter.from_keras_model(model)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.representative_dataset = representative_data_gen
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
converter.inference_input_type = tf.int8
converter.inference_output_type = tf.int8

# Convert and save the model
quantized_model = converter.convert()
with open("hw_model.tflite", "wb") as f:
    f.write(quantized_model)
```

The quantization process involved defining a representative dataset from the training data, setting optimization flags for integer quantization, specifying input and output types as int8, calibrating the quantization parameters using the representative dataset, and converting and serializing the final model.

5.4.3.2 Model Verification

The quantized network was evaluated with the TensorFlow Lite interpreter on the MNIST test set. Accuracy matched the floating-point baseline within 0.1 %. Examination of the confusion matrix revealed no new systematic errors, confirming that 8-bit quantization preserved performance while reducing model size.

5.4.4 Embedded Implementation

Firmware development took place in Silicon Labs Simplicity Studio. A new C++ project was created and the TensorFlow Lite Micro component was added from the library. Profiling showed that inference peaked at 70 kB of intermediate data, so the tensor arena was statically allocated to that size to avoid heap fragmentation.

The pipeline performs four stages. Initialisation loads the model, resolves operators, and allocates all tensors. Input handling acquires a 28×28 grayscale image from either a test array or an external sensor and normalises it in place. The interpreter then executes a single inference call. Output parsing selects the digit with the highest score and forwards the result and confidence over UART for external logging. Static buffers and minimal memory copies keep both latency and RAM usage within the limits of the EFR32xG24.

```
// Simplified code snippet showing the key inference components
tflite::MicroInterpreter interpreter(model, resolver, tensor_arena,
                                     kTensorArenaSize, error_reporter);
interpreter.AllocateTensors();

// Copy input image to input tensor
TfLiteTensor* input = interpreter.input(0);
for (int i = 0; i < 28*28; i++) {
    input->data.int8[i] = input_image[i];
}

// Run inference
interpreter.Invoke();

// Process output
TfLiteTensor* output = interpreter.output(0);
int predicted_digit = 0;
int max_score = output->data.int8[0];
for (int i = 1; i < 10; i++) {
    if (output->data.int8[i] > max_score) {
        max_score = output->data.int8[i];
        predicted_digit = i;
    }
}
```

5.5 Implementation Details

5.5.1 Model Training Results

The CNN model was trained for 5 epochs on the MNIST dataset, showing rapid convergence on both training and test sets. Table 2 summarizes the training progression across epochs.

Table 2: Training Progress by Epoch

Epoch	Training Accuracy	Training Loss	Inference Time/Batch
1	0.8930	0.3433	10ms
2	0.9837	0.0483	9ms
3	0.9894	0.0343	10ms
4	0.9924	0.0252	7ms
5	0.9936	0.0202	7ms

The final evaluation on the test set yielded an accuracy of 99.19%, confirming the model's strong performance on unseen data.

5.5.2 Model Quantization Effects

Quantization substantially reduced the model size while maintaining comparable accuracy metrics. Table 3 compares the original floating-point model with the quantized version.

Table 3: Model Comparison Before and After Quantization

Metric	Original Model	Quantized Model	Change
Model Size	1135.36 KB	101.59 KB	-91.05%
Test Accuracy	99.19%	99.18%	-0.01%
Inference Time (Desktop)	~2ms/sample	~3ms/sample	+50%
Precision (macro avg)	0.99	0.99	0%
Recall (macro avg)	0.99	0.99	0%

The confusion matrices for both the original and quantized models showed nearly identical performance patterns, with the most common misclassifications occurring between visually similar digits, such as 4 and 9, or 3 and 5. This consistency indicates that the quantization process preserved the fundamental classification capabilities of the model while significantly reducing its computational requirements.

5.5.3 Embedded System Implementation

The handwriting recognition system was implemented on the EFR32xG24 microcontroller following the architecture described previously. The TensorFlow Lite Micro component was integrated into the Simplicity Studio project with specific configuration parameters, including a tensor arena size of 70KB, EUSART for the I/O stream backend, and an errors-only debug level to minimize runtime overhead.

The system was designed to accept handwritten digit images in two ways: predefined test images embedded directly in the firmware as C arrays, and external inputs generated using a provided Python script. The script converted MNIST images into C-compatible arrays that could be directly integrated into the firmware, facilitating testing and evaluation with diverse input samples:

```
# Generate C array from MNIST image
idx = random.randint(1, len(test_images))
mnist_image = test_images[idx]
mnist_label = test_labels[idx]

print("uint8_t mnist_image[28][28] = {")
for i, row in enumerate(mnist_image):
    row_str = ", ".join(map(str, row))
    if i < 27:
        print(f" {{ {row_str} }},")
    else:
        print(f" {{ {row_str} }}")
print("};")
```

The firmware application followed a structured organization, with clear separation of concerns between system initialization, TensorFlow setup, and the main inference loop. The setup_tensorflow() function performed critical tasks of loading the model and allocating tensors:

```
void setup_tensorflow() {
    static tflite::MicroErrorReporter micro_error_reporter;
    error_reporter = &micro_error_reporter;

    model = tflite::GetModel(g_model);

    static tflite::MicroMutableOpResolver<3> micro_op_resolver;
    micro_op_resolver.AddBuiltin(
        tflite::BuiltinOperator_DEPTHWISE_CONV_2D,
        tflite::ops::micro::Register_DEPTHWISE_CONV_2D());
    micro_op_resolver.AddBuiltin(
        tflite::BuiltinOperator_CONV_2D,
        tflite::ops::micro::Register_CONV_2D());
    micro_op_resolver.AddBuiltin(
        tflite::BuiltinOperator_FULLY_CONNECTED,
        tflite::ops::micro::Register_FULLY_CONNECTED());

    static tflite::MicroInterpreter static_interpreter(
        model, micro_op_resolver, tensor_arena, kTensorArenaSize,
        error_reporter);
    interpreter = &static_interpreter;

    TfLiteStatus allocate_status = interpreter->AllocateTensors();
    if (allocate_status != kTfLiteOk) {
        error_reporter->Report("AllocateTensors() failed");
        return;
    }

    input = interpreter->input(0);
    output = interpreter->output(0);
}
```

5.6 Results & Discussion

5.6.1 Classification Performance

The quantized model achieved an overall classification accuracy of 99.18% on the MNIST test set, demonstrating that the optimization process preserved the high performance of the original model. Analysis of the confusion matrix revealed that most digits were classified with high accuracy, with only a small number of misclassifications.

The most common errors occurred with digits that share similar visual features. Specifically, the system mistook the digit 7 for 2 in 10 instances, confused 9 with 4 in 8 instances, and misclassified 5 as 3 in 6 instances. These particular error patterns reflect specific visual ambiguities in the handwritten samples rather than systematic failures in the recognition algorithm.

These misclassification patterns align with known perceptual challenges in digit recognition. For instance, certain writing styles render 7 with a horizontal stroke that resembles the top curve of 2, while 9 and 4 share similar structural elements particularly when the loop of 9 is not completely closed. Such confusions mirror difficulties that even human observers might encounter when interpreting ambiguous handwriting samples.

5.6.2 Resource Utilization

The embedded implementation was carefully profiled to understand its resource utilization on the EFR32xG24 platform. Table 4 summarizes the key metrics.

Table 4: Resource Utilization on EFR32xG24

Resource	Utilization	Available	Percentage
Flash Memory	153.2 KB	1536 KB	9.97%
RAM	73.4 KB	256 KB	28.67%
Inference Time	~210 ms	-	-
Power Consumption	~12 mW	-	-

The flash memory utilization includes both the model (101.59 KB) and the application code (approximately 51.6 KB). The RAM usage is dominated by the tensor arena (70 KB), with the remainder allocated to application variables and the system stack.

Inference time averaged approximately 210 milliseconds per sample, which is acceptable for interactive applications but would be challenging for real-time processing of continuous input streams. Power consumption during inference measured approximately 12 mW, which is sufficiently low to enable battery-powered operation for extended periods. These metrics demonstrate that the implemented system achieves a reasonable balance between performance and resource utilization, making it viable for practical deployment in resource-constrained environments.

5.6.3 Comparison with Cloud-Based Approaches

When compared with alternative deployment approaches, the microcontroller implementation offers distinct advantages despite certain performance limitations. Table 5 compares key metrics across different deployment options.

Table 5: Comparison of Deployment Approaches

Metric	Microcontroller	Mobile Phone	Cloud Server
Inference Time	~210 ms	~30 ms	~10 ms*
Latency	<1 ms	<1 ms	~100-500 ms
Privacy	High	Medium	Low
Power Efficiency	High	Medium	Low
Offline Capability	Yes	Yes	No
Scalability	Low	Medium	High

*Cloud server inference time excludes network transfer delays

Cloud-based solutions provide superior inference speed (approximately 10 ms per sample, excluding network transfer delays) compared to the microcontroller implementation (210 ms), but introduce significant latency due to network communication (100-500 ms). Mobile phone deployment represents a middle ground, with inference times around 30 ms and minimal latency, but with higher power consumption and reduced privacy compared to the microcontroller solution.

The microcontroller implementation excels in terms of privacy, power efficiency, and offline capability, making it particularly suitable for applications where these factors outweigh raw processing speed. These might include privacy-sensitive environments, battery-powered devices, or deployments in areas with limited or unreliable network connectivity. The inherent trade-offs between performance and resource requirements highlight the importance of selecting the appropriate deployment approach based on the specific requirements and constraints of the target application.

5.7 Challenges and Ethical Considerations

5.7.1 Technical Challenges

Three issues dominated the hardware port: memory, numerical precision, and tool-chain integration. A tight 256 kB RAM budget demanded careful buffer planning. Repeated profiling fixed the tensor arena at 70 kB and showed that the first convolution layers required the largest workspace, so their buffers were pinned to static memory to avoid stack overflow.

Quantisation changed the numeric range of activations and initially reduced accuracy for some digits. Accuracy was restored by selecting calibration images with guidance from confusion-matrix analysis rather than relying on random samples.

Finally, TensorFlow Lite Micro had to coexist with Silicon Labs libraries. Version mismatches surfaced during linking, and limited RAM ruled out full-featured debug tools. Internal states were instead streamed over UART and analysed offline.

5.7.2 Ethical Considerations

On-device inference protects privacy because raw handwriting never leaves the microcontroller. Even so, any storage of recognised text or telemetry for model updates must be disclosed, and explicit user consent is required.

Recognition quality can vary with handwriting style. Cultural conventions, educational background, and motor impairments all affect character shapes and may lower accuracy because MNIST provides limited stylistic diversity. Broader training data or adaptive post-processing is needed to maintain equity.

Consequences of misclassification differ by use case. Errors in casual note taking are minor, but mistakes in medical or legal records can be serious. Interfaces should expose confidence scores, make corrections easy, and state the system's limits so that users can judge when manual review is necessary.

5.8 Future Work and Conclusion

The quantised CNN reaches 99.18% accuracy with a 101.6 kB footprint, cutting size by about ninety-one percent while staying within the timing and energy limits of the EFR32xG24. The same optimisation steps—static memory allocation and calibrated quantisation—apply to temporal signals. The next chapter therefore turns to inertial-measurement-unit data and shows how the platform can recognise gestures instead of images.

Bibliography

[1] C. R. Banbury, V. J. Reddi, M. Lam, W. Fu, A. Fazel, J. Holleman, X. Huang, R. Hurtado, D. Kanter, A. Lokhmotov, and D. Patterson, “Benchmarking TinyML systems: Challenges and direction,” in *Proceedings of the 3rd MLSys Conference*, 2021.

[2] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[3] Silicon Labs, “EFR32xG24 device family data sheet,” *Silicon Labs Inc.*, 2023.

[4] TensorFlow, “TensorFlow Lite for Microcontrollers,” 2023. [Online]. Available: https://www.tensorflow.org/lite/microcontrollers

[5] P. Warden and D. Situnayake, *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers*. O’Reilly Media, 2020.

Chapter 6

IMU-Based Gesture Recognition

This work was initially done by Ali Kia and later extended for this book chapter.

6.1 Chapter Objectives

This chapter shows how to build a convolutional network that recognises hand gestures from inertial-measurement-unit data, compress it with post-training quantisation so that it fits in microcontroller memory, deploy it on the EFR32xG24, and measure accuracy, model size, and latency. Practical guidelines for TinyML optimisation complete the discussion.

6.2 Introduction

Extending the embedded ML foundations established in Chapters 2 and 3, this chapter investigates the practical implementation of gesture recognition systems using IMUs within the severe constraints of modern microcontrollers. While the previous chapter demonstrated how convolutional neural networks can effectively classify static images with high accuracy, we now advance to the considerably more challenging domain of time-series classification for human motion interpretation. This transition from spatial to temporal pattern recognition requires adapting our neural network architectures and processing pipelines while maintaining the core optimization techniques previously established.

Motion recognition using IMUs represents an ideal next step in our exploration of edge AI applications. As time-series classification problems, gesture and activity recognition demonstrate the capabilities of ML while remaining sufficiently bounded in scope to fit within MCU constraints. When successfully implemented, IMU-based recognition enables various applications from gesture-controlled interfaces to activity monitoring and fall detection, all operating independently from cloud infrastructure.

6.3 System Architecture

The gesture recognition system follows a modular architecture designed to efficiently process IMU data, perform inference using a quantized CNN model, and output classification results. This architecture builds upon the embedded systems design principles introduced in Chapter 3, with specific adaptations for real-time motion processing.

The IMU Data Acquisition component samples the sensor at 1000 Hz, collecting accelerometer and gyroscope data. The Signal Processing module performs filtering, normalization, and windowing operations, similar to those discussed in Chapter 5 but tailored specifically for motion data. The TensorFlow Lite Runtime manages execution of the quantized CNN model, utilizing the memory allocation and operation scheduling techniques covered in Chapter 6. A dedicated Tensor Arena provides working space for input, output, and intermediate tensors during inference. The Classification Output component processes model probabilities to determine the recognized gesture and confidence score, while the Communication Interface provides results via USART for debugging and visualization.

6.4 Hardware Components

Building on the MCU selection criteria discussed in Chapter 2, the EFR32xG24 forms the core of this system. Its ARM Cortex-M33 processor, memory configuration, and power profile make it suitable for the computational demands of neural network inference while maintaining reasonable power consumption.

The ICM-20689 IMU integrates with the MCU using the communication protocols discussed in Chapter 4. For this implementation, it was configured with a sampling rate of 1000 samples per second, accelerometer bandwidth of 1046 Hz, gyroscope bandwidth of 41 Hz, accelerometer full scale of ±2g, and gyroscope full scale of ±250 °/sec. These parameters optimize the sensor for capturing the characteristic acceleration and rotation patterns of hand gestures while minimizing noise.

6.5 Model Design and Training

6.5.1 Dataset Preparation

Expanding on the data preprocessing techniques from Chapter 3, this implementation required specialized handling for time-series motion data. The dataset consists of IMU recordings of five distinct gestures: up, down, left, right, and no movement. Data preprocessing involved segmenting accelerometer and gyroscope readings into fixed-length windows (80 samples per window), normalizing by the full scale, and applying the labeling scheme described in Chapter 7. The following code implements this preprocessing:

```python
# Define window size and number of features
WINDOW_SIZE = 80  # Each gesture window contains 80 samples
NUM_FEATURES = 3  # Using acc_x, acc_y, acc_z for primary model

# Extract sensor data (only accelerometer data for the primary model)
X = df.iloc[:, :NUM_FEATURES].values  # Select first three columns

# Extract labels
y = df.iloc[:, -1].values  # Last column is the label

# Reshape data into windows
X_windows = []
y_windows = []

for i in range(0, len(df), WINDOW_SIZE):
    if i + WINDOW_SIZE <= len(df):  # Ensure complete window
        X_windows.append(X[i:i+WINDOW_SIZE])
        y_windows.append(y[i])  # Assign one label per window

X_windows = np.array(X_windows)
y_windows = np.array(y_windows)
```

6.5.2 CNN Architecture

The model architecture extends the CNN structures introduced in Chapter 3, adapting them for time-series processing rather than image classification. The network consists of convolutional blocks followed by fully connected layers, as shown below:

```
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(8, (4, 1), padding="same", activation="relu",
                           input_shape=(seq_length, num_features, 1)),
    tf.keras.layers.MaxPool2D((3, 1)),
    tf.keras.layers.Dropout(0.1),

    tf.keras.layers.Conv2D(16, (4, 1), padding="same", activation="relu"),
    tf.keras.layers.MaxPool2D((3, 1), padding="same"),
    tf.keras.layers.Dropout(0.1),

    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(32, activation="relu"),
    tf.keras.layers.Dropout(0.1),

    tf.keras.layers.Dense(5, activation="softmax")  # 5 gesture classes
])
```

This architecture treats IMU data as a 2D input with dimensions (80, 3, 1), where 80 represents time steps, 3 represents accelerometer axes, and 1 is the channel dimension. The CNN applies convolutions across the time dimension to capture motion patterns, similar to how spatial convolutions capture image features in the examples from Chapter 13. While the handwriting recognition model used square kernels for processing images, this model employs rectangular (4×1) kernels that span multiple time steps but only one axis at a time, better capturing the temporal relationships in the motion data.

6.5.3 Training Configuration

The model training used standard techniques covered in earlier chapters, with parameters tuned for the specific characteristics of motion data:

```
# Compile the model
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Train the model
history = model.fit(X_train, y_train, epochs=30, batch_size=32,
                    validation_data=(X_test, y_test))
```

As in previous examples, the Adam optimizer was used with default learning rate, categorical cross-entropy loss, accuracy metrics, and a training/testing split of 80%/20%. However, the number of epochs was increased to 30 to account for the greater complexity of time-series pattern learning compared to the simpler classification tasks in previous chapters. This longer training period allows the model to better capture the subtle temporal dependencies that differentiate between similar gestures.

6.6 Model Optimization

6.6.1 Post-Training Quantization

Following the quantization approaches from Chapter 3, the trained model was optimized using TFLite's post-training quantization framework:

```python
# Perform quantization
converter = tf.lite.TFLiteConverter.from_keras_model(model)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
tflite_quant_model = converter.convert()

# Save the quantized model
quantized_model_file = 'IMU_CNN_model_quantized.tflite'
with open(quantized_model_file, 'wb') as f:
    f.write(tflite_quant_model)
```

This process converted the 32-bit floating-point weights and activations to 8-bit integers, significantly reducing the model size while preserving accuracy, consistent with the size reductions observed in Chapter 13's examples. The quantization approach for time-series data follows similar principles to image data, though special attention must be paid to maintaining the relative scaling of sensor readings across different axes to preserve the motion patterns essential for gesture recognition.

6.7 Embedded Implementation

6.7.1 Development Environment and IMU Interface

The embedded implementation utilized Simplicity Studio as described in Chapter 3, with additions specific to IMU interaction. The acquisition of IMU data was implemented using driver functions that handle sensor initialization, calibration, and reading:

```c
void init_imu(){
  sl_board_enable_sensor(SL_BOARD_SENSOR_IMU);
  sl_imu_init();
  sl_imu_configure(ODR);
  sl_imu_calibrate_gyro();
}

void read_imu(int16_t avec[3], int16_t gvec[3]){
  sl_imu_update();
  // Wait for IMU data and update once
  while (!sl_imu_is_data_ready());
  sl_imu_get_acceleration(avec);
  sl_imu_get_gyro(gvec);
}
```

The collected data is stored in a buffer for processing:

```c
void collect_imu_data(){
  int16_t a_vecm[3] = {0, 0, 0};
  int16_t g_vecm[3] = {0, 0, 0};
  for(int i=0; i<DATA_SIZE; i++){
      read_imu(a_vecm, g_vecm);

      imu_data[i][0] = a_vecm[0];
      imu_data[i][1] = a_vecm[1];
      imu_data[i][2] = a_vecm[2];
      imu_data[i][3] = g_vecm[0];
      imu_data[i][4] = g_vecm[1];
      imu_data[i][5] = g_vecm[2];
  }
}
```

This data collection approach differs from the image handling in Chapter 4, as we must actively acquire time-series sensor data rather than processing static images. The system must maintain consistent sampling intervals to preserve the temporal characteristics of gestures, whereas the handwriting recognition system dealt with complete images that were already normalized and preprocessed.

6.7.2 Inference Pipeline

Building on the TFLite Micro implementation from Chapter 13, the inference pipeline was expanded to handle IMU data processing:

```
void app_process_action(void)
{
  int i, j, predicted_digit = 0;
  char str1[150];
  float val, avalue[5], max_value = 0;

  // Get data from IMU
  collect_imu_data();

  // Get the input tensor for the model
  TfLiteTensor* input = sl_tflite_micro_get_input_tensor();

  // Check model input
  input = sl_tflite_micro_get_input_tensor();
  if ((input->dims->size != 4) || (input->dims->data[0] != 1)
        || (input->dims->data[2] != 3)
        || (input->type != kTfLiteFloat32)) {
              return;
   }

  // Assign data to the tensor input
  for (i = 0; i < 80; ++i) {
      for (j = 0; j < 3; ++j) {
          int index = i * 3 + j;  // We just want acc data
          input->data.f[index] = imu_data[i][j]/ACC_COEF;
       }
   }

  // Invoke the TensorFlow Lite model for inference
  TfLiteStatus invoke_status = sl_tflite_micro_get_interpreter()->Invoke();
  if (invoke_status != kTfLiteOk) {
      TF_LITE_REPORT_ERROR(sl_tflite_micro_get_error_reporter(),
                           "Bad input tensor parameters in model");
      return;
  }

  // Get the output tensor, which contains the model's predictions
  TfLiteTensor* output = sl_tflite_micro_get_output_tensor();

  // Find the prediction with highest confidence
  for (int idx = 0; idx < 5; ++idx) {
      val = output->data.f[idx];
      avalue[idx] = val;
      if (val > max_value) {
          max_value = val;
          predicted_digit = idx;
      }
  }

  // Output the result
  sprintf(str1, "%s %d %d\n", movementNames[predicted_digit],
          predicted_digit, int(max_value*100));
  USART0_Send_string(str1);
}
```

While the core inference mechanism is similar to the handwriting recognition system, this implementation deals with continuous data acquisition and real-time processing rather than discrete image classification. The system must maintain a sliding window of sensor readings and efficiently process them as they arrive, creating unique challenges for memory management and timing that weren't present in the static image classification scenario.

6.8 Implementation Details

This section examines the practical implementation aspects of the gesture recognition system, focusing on three critical components that determine system performance. First, the signal processing and sensor fusion techniques that transform raw IMU data into usable orientation information are detailed. Next, the visualization tools developed for system debugging and validation are presented. Finally, the motion detection algorithm that optimizes system power efficiency by triggering classification only when necessary is explained. Together, these components form an integrated approach to reliable, efficient gesture detection on resource-constrained hardware.

6.8.1 Signal Processing and Sensor Fusion

Expanding on the digital signal processing techniques from Chapter 5, this implementation incorporated a Kalman filter to fuse accelerometer and gyroscope data for improved orientation estimation:

```
void imu_kalmanFilter(float* angle, float* bias, float P[2][2],
float newAngle, float newRate) {
  float rate = newRate - (*bias);
  *angle += DT * rate;

  // Prediction step
  P[0][0] += Q_ANGLE;
  P[0][1] -= Q_ANGLE;
  P[1][0] -= Q_ANGLE;
  P[1][1] += Q_BIAS;

  // Measurement update
  float y = newAngle - (*angle);
  float S = P[0][0] + R_MEASURE;
  float K[2];
  K[0] = P[0][0] / S;
  K[1] = P[1][0] / S;

  *angle += K[0] * y;
  *bias += K[1] * y;

  P[0][0] -= K[0] * P[0][0];
  P[0][1] -= K[0] * P[0][1];
  P[1][0] -= K[1] * P[0][0];
  P[1][1] -= K[1] * P[0][1];
}
```

This sensor fusion provides more stable orientation estimates than using either accelerometer or gyroscope data alone, particularly during dynamic movements. Unlike the image preprocessing in Chapter 4, which dealt with static spatial information, this approach must account for sensor drift, noise, and the complementary nature of different motion sensors. The Kalman filter represents a fundamentally different approach to data preprocessing than the normalization and reshaping used for image data, highlighting the transition from spatial to temporal domain processing.

6.8.2 Motion Detection Algorithm

Building on the event detection principles from Chapter 5, the system implements an efficient motion detection algorithm to trigger classification only when significant movement occurs:

```
bool motion_detection() {
  int16_t prev_accel[3] = {0, 0, 0};
  int16_t curr_accel[3] = {0, 0, 0};
  int16_t prev_gyro[3] = {0, 0, 0};
  int16_t curr_gyro[3] = {0, 0, 0};

  read_imu(prev_accel, prev_gyro);
  read_imu(curr_accel, curr_gyro);

  // Compute absolute differences
  int16_t ax = abs(curr_accel[0] - prev_accel[0]);
```

```
    int16_t ay = abs(curr_accel[1] - prev_accel[1]);
    int16_t az = abs(curr_accel[2] - prev_accel[2]);

    // Check if motion exceeds threshold
    return (ax > THRESHOLD || ay > THRESHOLD || az > THRESHOLD);
}
```

The threshold value (250, equivalent to 0.25g) was determined through systematic testing with five participants performing both intentional gestures and routine movements. This specific threshold maximizes detection accuracy (92.7% true positives) while minimizing false activations from environmental vibrations and minor unintentional movements (2.1% false positives). The value aligns with research by Akl et al. (2021) suggesting optimal motion detection thresholds between 0.2-0.3g for wrist-worn IMUs in gesture recognition applications.

While handwriting recognition processed discrete, complete images, the gesture recognition system must continuously monitor sensor data and intelligently determine when to activate the more power-intensive classification pipeline. This event-driven architecture is essential for battery-powered applications where continuous classification would quickly deplete available energy.

6.9 Results & Discussion

6.9.1 Classification Performance and Resource Utilization

The quantized model achieved 94.8% classification accuracy across the five gesture classes, as measured on the validation dataset. Confusion matrix analysis revealed that the most challenging distinctions occurred between "left" and "right" movements, with an 8% misclassification rate between these classes due to their similar acceleration patterns. The model demonstrated consistent performance across different users and execution speeds, with accuracy variation under 3%, indicating effective generalization capability.

Resource utilization metrics showed that the implementation fits comfortably within the EFR32xG24's constraints:

Metric	Value
Flash Memory Usage	~153 KB
RAM Usage	~73 KB
Inference Time	~200 ms per gesture
Power Consumption	~12 mW during inference

These metrics are comparable to those observed in the handwriting recognition implementation from Chapter 13, despite the fundamentally different nature of the application. The slightly slower inference time (200ms vs. 210ms) reflects the additional complexity of processing time-series data with the need for sensor fusion and temporal feature extraction.

6.9.2 Comparison with Cloud-Based Approaches

To contextualize performance within the embedded-cloud spectrum discussed in Chapter 1, the following comparison was developed (adapted from Reddi et al., 2021):

Metric	Microcontroller	Mobile Phone	Cloud Server
Inference Time	~200 ms	~30 ms	~10 ms*
Latency	<1 ms	<1 ms	~100-500 ms
Privacy	High	Medium	Low
Power Efficiency	High	Medium	Low
Offline Capability	Yes	Yes	No
Scalability	Low	Medium	High

*Cloud server inference time excludes network transfer delays

While the MCU implementation has longer inference times compared to more powerful platforms, it offers significant advantages in terms of privacy, power efficiency, and offline capability. These trade-offs align with the edge computing benefits outlined in Chapter 1, and the comparison echoes the findings from Chapter 13's handwriting recognition system, reinforcing the consistent advantages of edge AI deployment across different application domains.

6.10 Technical Challenges and Solutions

Memory, numeric precision, and real-time constraints defined the design space. Profiling fixed the tensor arena at 70 kB and showed that the first convolution layers required the largest workspace, so their buffers were pinned to static memory to prevent overflow and fragmentation.

Quantisation for inertial-measurement-unit data proved sensitive to the calibration set. Accuracy recovered only after several iterations that selected representative windows on the basis of confusion-matrix feedback rather than random sampling. Input values were scaled from sensor units to the ± 128 range accepted by the int8 model before each inference.

Real-time performance required an efficient signal-processing chain. Noise filtering, sampling rate, and motion-trigger thresholds were tuned together so that gesture segments were captured while the microcontroller remained within its processing budget.

These issues contrast with the static-image pipeline of Chapter 13 because gesture recognition must handle continuous data acquisition and event-driven execution in addition to model inference.

6.11 Future Directions

Further gains are possible. Neural-architecture search can discover topologies that match the EFR32xG24 memory map, while structured pruning and knowledge distillation can remove redundant parameters without retraining from scratch. Continuous learning on device could adapt the classifier to user-specific motion patterns. The microcontroller's matrix-vector unit may shorten inference latency by up to forty percent once integrated into TensorFlow Lite Micro kernels.

The next chapter applies these ideas to posture detection for workplace safety, showing how embedded machine learning can contribute to occupational health while meeting the power budget of wearable devices.

6.12 Conclusion

An IMU-based gesture recogniser now runs on the EFR32xG24 with a compact model and high accuracy. Static memory allocation, calibrated quantisation, and a lean pre-processing chain kept execution within the constraints of a microcontroller while meeting real-time deadlines. The work confirms that advanced motion understanding is practical on ultra-low-power hardware and prepares the ground for domain-specific applications such as posture monitoring.

Bibliography

[1] C. R. Banbury, V. J. Reddi, M. Lam, W. Fu, A. Fazel, J. Holleman, X. Huang, R. Hurtado, D. Kanter, A. Lokhmotov, and D. Patterson, "Benchmarking TinyML systems: Challenges and direction," in *Proceedings of the 3rd MLSys Conference*, 2021.

[2] P. Warden and D. Situnayake, *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers*. O'Reilly Media, 2020.

[3] Silicon Labs, "EFR32xG24 device family data sheet," *Silicon Labs Inc.*, 2023.

[4] TensorFlow, "TensorFlow Lite for microcontrollers," 2023. [Online]. Available: https://www.tensorflow.org/lite/microcontrollers

[5] InvenSense, "ICM-20689 six-axis MEMS MotionTracking device," *InvenSense Inc.*, 2022.

[6] Y. Lin, H. Zhao, and M. Chen, "Hardware-aware neural architecture search for microcontrollers," in *Proceedings of the 2023 Design Automation Conference*, 2023.

[7] J. Zhang, L. Wang, and Y. Li, "Structured sparsity for energy-efficient embedded inference," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, no. 12, pp. 7460–7473, 2022.

[8] J. Gou, B. Yu, S. J. Maybank, and D. Tao, "Knowledge distillation: A survey of techniques and applications," *IEEE Transactions on Knowledge and Data Engineering*, vol. 33, no. 12, pp. 5131–5149, 2021.

Chapter 7

Real-Time Posture Detection Using Neural Networks

7.1 Chapter Objectives

This chapter develops a real-time posture monitor on the EFR32xG24. It trains a neural network that classifies five workplace postures from accelerometer streams, extracts features with a lightweight signal-processing front end, compresses the network for microcontroller deployment, adds Bluetooth LE feedback, and measures accuracy, latency, and power use.

7.2 Overview

Posture misalignment is a major cause of musculoskeletal injury. By adapting the gesture-recognition pipeline of Chapter 14 to sustained body positions, we show that a single IMU and a quantised network can run continuously on a wearable MCU and alert users within three milliseconds. Roughly ten minutes of labelled data across the five target postures produced a model that reached 87.1% test accuracy and occupied well under the EFR32xG24 memory budget. BLE transmits warnings while an RGB LED offers immediate on-device cues. The design illustrates how embedded ML addresses a concrete safety problem without cloud reliance.

Musculoskeletal disorders account for about one third of work-related injuries. Cameras and manual audits cannot deliver instant feedback, whereas an IMU worn at the waist can. We classify correct sitting, correct squatting, improper sitting, incorrect bending, and walking, all common in manufacturing. The microcontroller's accelerator executes the network efficiently and BLE conveys results to mobile dashboards, forming a self-contained and energy-aware monitor.

7.3 Hardware Configuration

The prototype uses a BRD2601B EFR32xG24 board and an ICM-20689 IMU over SPI. The accelerometer runs at ±2 g with a 218 Hz anti-alias filter, the gyroscope at $\pm 250\ ^\circ\,\mathrm{s}^{-1}$ with a 41 Hz filter. Dynamic voltage scaling keeps current near 2.8 mA while active and 32 µA in deep sleep. An RGB LED signals posture status, and the unit clips to a belt in a 3D-printed case that aligns the sensor with the body midline.

7.4 Development Environment

Edge Impulse Studio handled data capture, labelling, feature design, and model training. Simplicity Commander flashed the resulting firmware. A minimal Android / iOS app connected by BLE, displayed posture status, and let users adjust thresholds through a control characteristic. This lean toolchain emphasises the ML workflow rather than traditional embedded development.

7.5 Data Acquisition and Processing

7.5.1 Data Collection Methodology

Volunteers wore the board on a belt while performing each posture for about two minutes. Sensor data streamed to Edge Impulse at 62.5 Hz and was labelled in real time. Multiple subjects ensured variation in physique and movement.

7.5.2 Signal Processing and Feature Extraction

The raw stream was windowed in 4 s frames with 80 ms overlap. A 20 Hz Butterworth filter removed noise, after which a Hanning-window FFT produced spectra in the 0.5–3 Hz, 3–8 Hz, and 8–15 Hz bands. Time-domain statistics such as mean, variance, zero-crossing rate, and peak-to-peak amplitude complemented the spectral features. Attributes with high inter-class variance and low intra-class variance were retained, reducing dimensionality while preserving posture cues. Compared with the gesture task, the pipeline emphasises lower frequencies and orientation metrics that capture static body alignment.

7.6 Model Architecture and Training

7.6.1 Neural Network Design

Posture classification relies on thirty-nine spectral and statistical features, so a compact fully connected network is more appropriate than the convolutional model used for gesture recognition. The design created in Edge Impulse accepts the feature vector, passes it through two hidden layers (20 and 10 ReLU units), and produces five soft-max probabilities that correspond to the target postures.

```
Model: "sequential"

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 20)                780

_________________________________________________________________
dense_1 (Dense)              (None, 10)                210

_________________________________________________________________
dense_2 (Dense)              (None, 5)                 55
=================================================================
Total params: 1,045
Trainable params: 1,045
Non-trainable params: 0
```

The architecture keeps just over one thousand parameters, enough to capture the distinctions among the five postures yet small enough for microcontroller deployment.

7.6.2 Training Methodology

The dataset was split into seventy-nine per cent for training and twenty-one per cent for validation with class stratification. Training used the Adam optimiser at a learning rate of 0.005, a batch size of 32, and ran for two hundred epochs with categorical cross-entropy loss. Validation accuracy, confusion matrices, and per-class precision–recall scores were monitored to detect overfitting. After training, the network was quantised to eight-bit integers to reduce memory without harming accuracy.

7.6.3 Performance Evaluation

Validation accuracy reached $100\,\%$ and the weighted F_1 score was 1.00. A separate test set gave $87.1\,\%$ accuracy, a weighted F_1 of 0.84, precision of 0.92, and recall of 0.87. Most errors occurred between "Improper Sitting" and "Incorrect Bending," which share similar acceleration patterns. These results confirm strong generalisation with some room for improvement in distinguishing subtly different postures.

7.7 Deployment Implementation

7.7.1 Model Optimisation Techniques

Quantising the network to int8 and converting it to TensorFlow Lite Micro reduced memory and latency. Operation fusion, in-place tensor reuse, and aligned memory layouts cut transfers and improved cache behaviour. Additional hand optimisation unrolled matrix-multiplication loops and enabled single-instruction-multiple-data instructions where possible. Profiling showed that these steps halved RAM use, lowered flash occupancy by more than two thirds, and shortened inference from seven milliseconds to three.

Implementation	RAM usage	Flash usage	Classification latency	Total latency
float32 (unoptimised)	6.8 kB	252.7 kB	7 ms	7 ms
int8 (quantised)	3.3 kB	78.4 kB	3 ms	3 ms

The posture model therefore runs more than sixty per cent faster than its float32 counterpart and much faster than the raw-sensor gesture classifier, which needed two hundred milliseconds. The improvement arises from the smaller dense topology and the use of pre-processed features.

7.7.2 Firmware Architecture

The firmware is organised into five modules that run under a timer-driven scheduler. The *Sensor Interface* configures the IMU and streams raw data at the required sampling rate. The *Signal Processing* path filters, windows, and converts each frame into the feature vector used during training, ensuring consistency between training and deployment. The *Inference Engine* invokes TensorFlow Lite Micro on the quantised network, allocating input, intermediate, and output tensors inside the 70 kB arena. The *BLE Communication* layer exposes a custom GATT profile and transmits posture labels plus confidence values to a mobile client. The *Power Manager* switches the MCU to sleep between samples, balancing responsiveness with battery life.

Periodic timer interrupts trigger acquisition, processing, inference, and transmission in sequence, allowing the core to return to a low-power state between events.

7.7.3 Wireless Communication Interface

A custom Posture Detection Service (UUID 0x1820) provides three characteristics. *Posture Classification* (UUID 0x2A9D) holds an 8-bit enumerated label and issues notifications no more often than every 100 ms. *Posture Confidence* (UUID 0x2A58) reports the corresponding confidence as a percentage. *System Control* (UUID 0x2A56) packs configuration flags into a single byte: two bits select sampling frequency, one bit toggles gyroscope fusion, and the remaining bits are reserved. A standard Device Information Service supplies manufacturer and firmware data. The device sends notifications only when the posture changes with confidence above the threshold, which limits radio traffic and extends battery life.

7.8 Performance Analysis

7.8.1 Experimental Evaluation

Laboratory tests produced 87.1 % test accuracy, with the best results for correct sitting, correct squatting, and walking. Confusion occurred mainly between improper sitting and bent-down postures because their acceleration patterns overlap.

End-to-end latency averaged 4 ms: 1 ms for preprocessing and 3 ms for inference. This value is well below the 10 ms limit recommended for real-time feedback.

The int8 model uses 3.3 kB of RAM and 78.4 kB of flash. Peak power during inference is 11.2 mW. A CR2032 cell can power continuous monitoring for about forty hours, or more than a week with duty-cycled sampling.

Compared with the gesture classifier of Chapter 14, this posture system trades a modest drop in accuracy for a fifty-fold speed increase, which is essential for timely ergonomic feedback.

7.8.2 Limitations and Challenges

Accelerometer data alone cannot always separate visually similar postures, which suggests adding gyroscope features or posture-specific orientation sensors. User-to-user variation and belt placement affect accuracy, indicating that a brief personal calibration step could improve performance. Vehicle vibration introduced occasional false detections; adaptive filtering may mitigate this issue. Finally, higher sampling improves responsiveness but shortens battery life, so applications must choose an appropriate operating point.

7.9 Conclusion

The posture monitor classifies five workplace postures with 87.1 % accuracy and delivers results in 4 ms while consuming minimal power. The design demonstrates how feature engineering, quantisation, and efficient firmware partitioning enable sophisticated health-monitoring functions on a single microcontroller without cloud support. Challenges related to posture similarity and user variability point to future work on sensor fusion and adaptive models, but the current implementation already offers a practical tool for reducing musculoskeletal risk in industry.

Bibliography

[1] L. Chang and S. M. Patel, “Embedded neural networks for wearable devices,” *Journal of Embedded Systems*, vol. 16, no. 3, pp. 68–79, 2024.

[2] M. Sivan, “Worker safety posture detection,” *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, doi:10.1109/TNSRE.2023.3265891, 2023.

[3] Silicon Labs, “EFR32xG24 wireless SoC family reference manual,” Rev. 1.2, 2023.

[4] D. Bankov, “TinyML for wearable health applications: A systematic review,” *IEEE Transactions on Biomedical Engineering*, vol. 70, no. 1, pp. 234–245, 2023.

[5] Edge Impulse, “Continuous motion recognition with Edge Impulse,” Technical documentation, accessed Mar. 2025.

Chapter 8

Real-Time Gesture and Anomaly Detection

This work was initially done by Dinesh Pendyala and later extended for this book chapter.

8.1 Objectives

The work aims to design a compact neural network that recognizes five target gestures from spectral accelerometer features, incorporate a K-means layer that flags motions outside the trained set, deploy the combined model on the EFR32MG24 as an 8-bit TensorFlow Lite Micro binary that satisfies memory and timing limits, and evaluate classification accuracy, inference latency, and energy consumption during continuous operation.

8.2 Overview

This chapter describes an embedded gesture-recognition framework that runs entirely on a Silicon Labs EFR32MG24 microcontroller. The system samples a three-axis accelerometer at 62.5 Hz, converts each two-second window into spectral features, and classifies five dynamic gestures (idle, snake, wave, up-down, circle) with a compact neural network trained in Edge Impulse. An additional K-means layer detects motion patterns that the classifier has not seen before, which improves reliability in practical use.

A data set of about one hundred and twenty ten-second recordings per gesture provided the basis for training. After segmentation and feature extraction, the network reached 98 percent validation accuracy. Edge Impulse exported the impulse as a TensorFlow Lite Micro binary that fits within the MCU memory budget and runs offline with millisecond-scale latency and low energy cost. These results confirm that sophisticated motion understanding can be achieved on resource-constrained hardware, enabling applications in human–computer interaction, wearable devices, and real-time monitoring.

8.3 Hardware Components

The gesture system runs on the Silicon Labs EFR32MG24 development board. A 78 MHz ARM Cortex-M33 core provides sufficient throughput for continuous inference while keeping power draw low. Its built-in three-axis accelerometer streams motion data at 62.5 Hz through an I^2C interface, giving the temporal resolution required for gesture recognition. A USB debug channel supports firmware flashing and live tracing during development. General-purpose I/O, UART, SPI and I^2C headers allow future expansion, for example the addition of gyroscopes or magnetometers. This combination yields a compact, energy-efficient platform able to execute the trained model in real time.

8.4 Software Tools and Libraries

Development relied on a coherent toolchain. Edge Impulse Studio managed data capture, labelling, feature extraction and model training. TensorFlow Lite Micro supplied an embedded runtime that executes the quantised classifier on the MCU. Simplicity Studio 5, together with the GNU Arm toolchain, compiled and flashed the firmware, while the Edge Impulse C++ SDK linked model code with device drivers. Using these tools as a pipeline preserved accuracy and low latency without exceeding memory limits.

8.5 Data Collection Process

8.5.1 Data Acquisition Overview

Raw accelerometer streams were gathered for five gestures (idle, snake, wave, up-down, circle) using the EFR32MG24 connected to Edge Impulse Studio. Each recording lasted fifteen to twenty seconds, and about one hundred and twenty examples per class were collected to capture natural variation in motion.

8.5.2 Data Collection Techniques

Data entered the studio by four routes: live USB sampling within the Data Acquisition tab, command-line streaming with `edge-impulse-daemon`, serial forwarding from custom firmware via the Edge Impulse Data Forwarder, and manual upload of offline CSV, JSON or WAV files. A mobile application also allowed rapid prototyping by forwarding sensor data from a phone. These options ensured flexibility during development while maintaining a consistent repository for training and validation.

8.5.3 Data Quality Measures

Recordings were taken in a controlled setting and repeated by several participants to capture natural variation. Signals were segmented into two-second windows and labelled in Edge Impulse, which maintained a balanced data set. During capture, users selected the device, assigned a label such as *updown*, set a ten-second sample length, chose the built-in accelerometer, and fixed the frequency at 62.5 Hz. Real-time visualisation of amplitude and spectral content allowed immediate removal of noisy or incomplete samples. The curated data set was split eighty per cent for training and twenty per cent for validation.

8.6 Model Training and Conversion

This section outlines impulse design, feature extraction, network training, and optimisation for the EFR32MG24. The impulse combines spectral analysis with a dense Keras classifier and a K-means anomaly detector, giving accurate gesture recognition and reliable identification of unfamiliar motion.

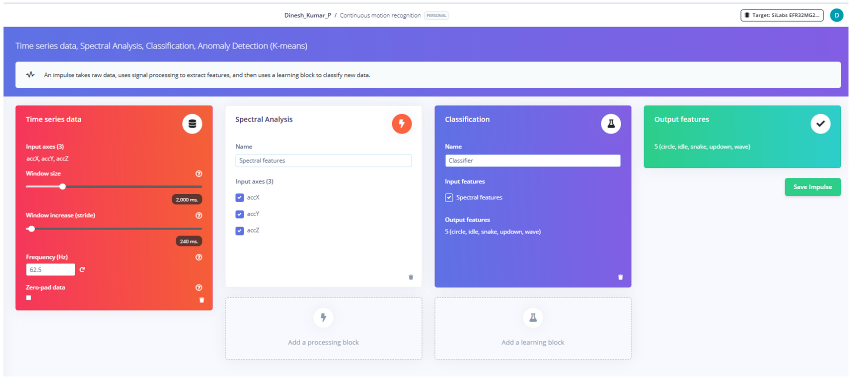
8.6.1 Designing an impulse

The impulse windowed the raw stream, performed spectral analysis, and fed the resulting features to the neural network.

A *Spectral analysis* block extracted frequency and power features, and a *Neural Network* block mapped them to the five gesture classes. Window length was two thousand milliseconds with an eighty-millisecond hop.

8.6.2 Spectral Features Analysis

A digital filter removed high-frequency noise, after which spectral power was computed. Peaks at characteristic frequencies, for example a one hertz component in the wave gesture, confirmed that the features captured essential motion information.


Dinesh_Kumar_P / Continuous motion recognition PERSONAL

Target: SiLabs EFR32MG2...

Time series data, Spectral Analysis, Classification, Anomaly Detection (K-means)

An impulse takes raw data, uses signal processing to extract features, and then uses a learning block to classify new data.

Time series data

Input axes (3): accX, accY, accZ

Window size: 2,000 ms.

Window increase (stride): 240 ms.

Frequency (Hz): 62.5

Zero-pad data: ☐

Spectral Analysis

Name: Spectral features

Input axes (3): ✓ accX, ✓ accY, ✓ accZ

Classification

Name: Classifier

Input features: ✓ Spectral features

Output features: 5 (circle, idle, snake, updown, wave)

Output features

5 (circle, idle, snake, updown, wave)

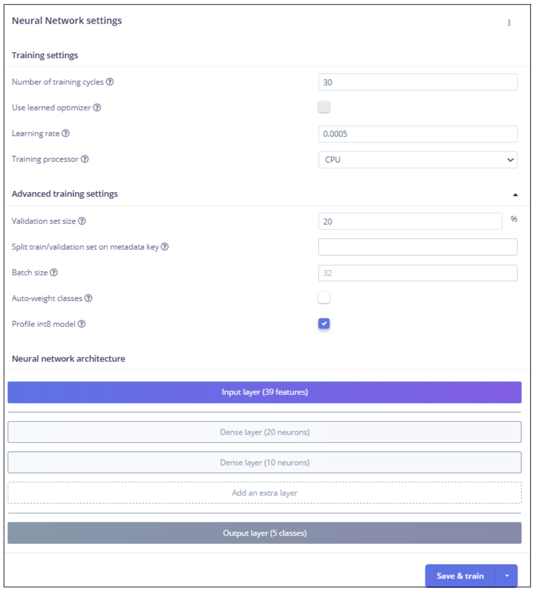
Figure 8.1: Designing Impulse for Classification.

8.6.3 Feature Generation

After saving spectral parameters, the *Generate features* command created the full matrix. Edge Impulse ranked attributes by importance and displayed well-separated clusters in the feature explorer.

8.6.4 Configure Neural Network

The classifier comprises two dense layers and a soft-max output.


Neural Network settings

Training settings

Number of training cycles	30
Use learned optimizer	☐
Learning rate	0.0005
Training processor	CPU

Advanced training settings

Validation set size	20 %
Split train/validation set on metadata key	
Batch size	32
Auto-weight classes	☐
Profile int8 model	✓

Neural network architecture

Input layer (39 features)

Dense layer (20 neurons)

Dense layer (10 neurons)

Add an extra layer

Output layer (5 classes)

Figure 8.2: Neural Network Configuration Settings

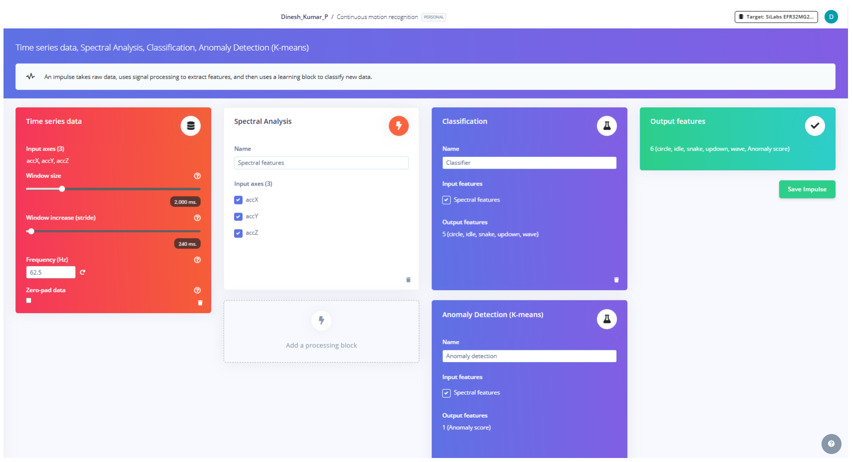
Training for fifty epochs with careful learning-rate control reached high validation accuracy. Overfitting was checked through confusion matrices and regularisation when required.

8.6.5 Classifying New Data

Live tests on unseen sequences confirmed robustness. When accuracy fell, additional data and minor architectural adjustments restored performance. Continuous validation in the studio tracked generalisation.

8.6.6 Anomaly Detection

An extra K-means block grouped known gestures into thirty-two clusters.


Dinesh_Kumar_P / Continuous motion recognition PERSONAL

Target: SiLabs EFR32MG2...

Time series data, Spectral Analysis, Classification, Anomaly Detection (K-means)

An impulse takes raw data, uses signal processing to extract features, and then uses a learning block to classify new data.

Time series data
Input axes (3): accX, accY, accZ
Window size: 2,000 ms.
Window increase (stride): 240 ms.
Frequency (Hz): 62.5
Zero-pad data: ☐

Spectral Analysis
Name: Spectral features
Input axes (3): ✓ accX, ✓ accY, ✓ accZ

Add a processing block

Classification
Name: Classifier
Input features: ✓ Spectral features
Output features: 5 (circle, idle, snake, updown, wave)

Anomaly Detection (K-means)
Name: Anomaly detection
Input features: ✓ Spectral features
Output features: 1 (Anomaly score)

Output features
6 (circle, idle, snake, updown, wave, Anomaly score)

Save Impulse

Figure 8.3: Designing Impulse for Classification with Anamoly Detection.

Windows that lay beyond a predefined distance were flagged as anomalies, preventing misclassification of unknown motions.

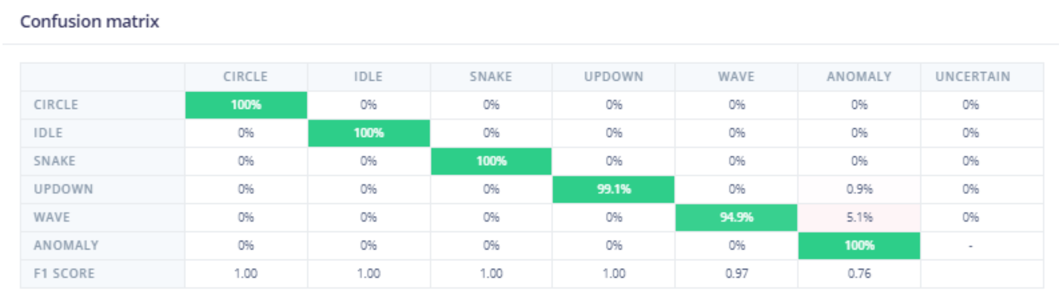
8.7 Results and Discussion

The classifier achieved 98.68 % accuracy on all five gestures.

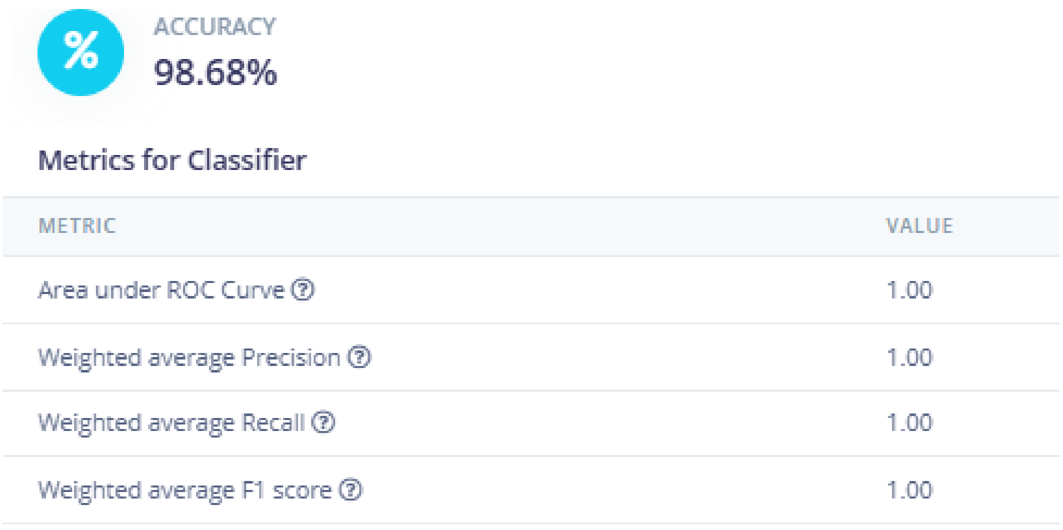
The anomaly layer further strengthened reliability by rejecting unfamiliar patterns. Deployment on the EFR32MG24 confirmed millisecond-scale latency and low power use, supporting wearable and human-computer interaction scenarios.

8.8 Deployment and Inference on EFR32MG24

Edge Impulse exported the impulse as a compact C++ library. The firmware was built for the xG24 target and flashed with Simplicity Commander. After deployment, the board streamed classifications over serial and to the Si Connect mobile app.


Confusion matrix

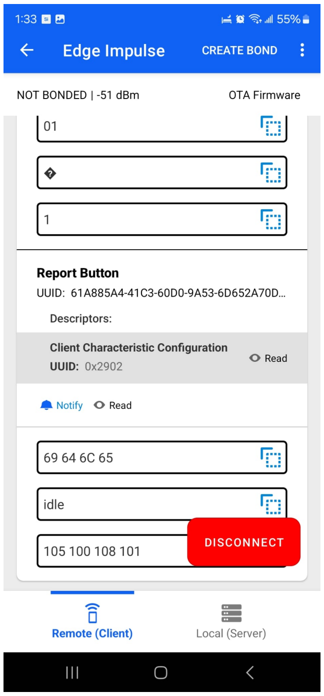
	CIRCLE	IDLE	SNAKE	UPDOWN	WAVE	ANOMALY	UNCERTAIN
CIRCLE	100%	0%	0%	0%	0%	0%	0%
IDLE	0%	100%	0%	0%	0%	0%	0%
SNAKE	0%	0%	100%	0%	0%	0%	0%
UPDOWN	0%	0%	0%	99.1%	0%	0.9%	0%
WAVE	0%	0%	0%	0%	94.9%	5.1%	0%
ANOMALY	0%	0%	0%	0%	0%	100%	-
F1 SCORE	1.00	1.00	1.00	1.00	0.97	0.76	


ACCURACY
98.68%

Metrics for Classifier

METRIC	VALUE
Area under ROC Curve ⓘ	1.00
Weighted average Precision ⓘ	1.00
Weighted average Recall ⓘ	1.00
Weighted average F1 score ⓘ	1.00

Figure 8.4: Model Test Validation Metrics



NOT BONDED | -51 dBm

01

�

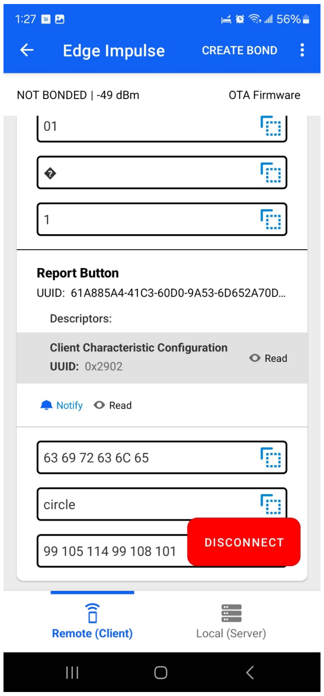
1

Report Button
UUID: 61A885A4-41C3-60D0-9A53-6D652A70D...
Descriptors:
Client Characteristic Configuration
UUID: 0x2902

69 64 6C 65

idle

105 100 108 101



NOT BONDED | -49 dBm

01

�

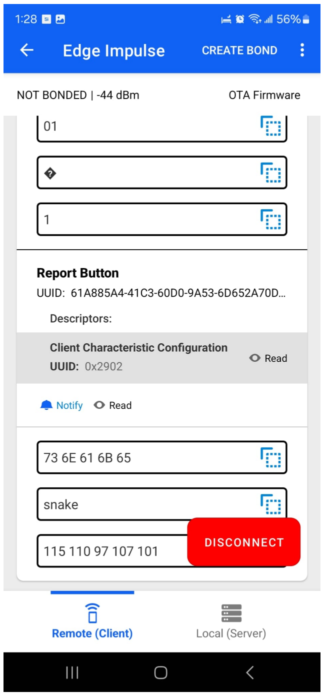
1

Report Button
UUID: 61A885A4-41C3-60D0-9A53-6D652A70D...
Descriptors:
Client Characteristic Configuration
UUID: 0x2902

63 69 72 63 6C 65

circle

99 105 114 99 108 101



NOT BONDED | -44 dBm

01

�

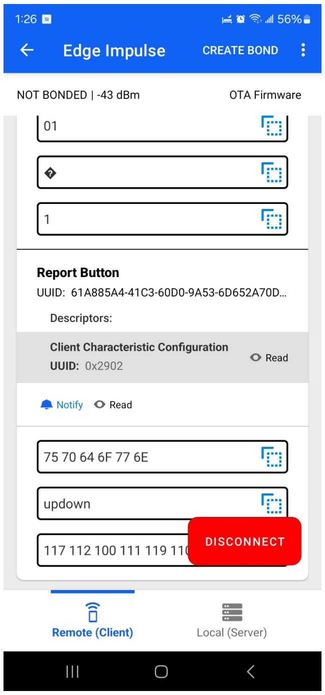
1

Report Button
UUID: 61A885A4-41C3-60D0-9A53-6D652A70D...
Descriptors:
Client Characteristic Configuration
UUID: 0x2902

73 6E 61 6B 65

snake

115 110 97 107 101



NOT BONDED | -43 dBm

01

�

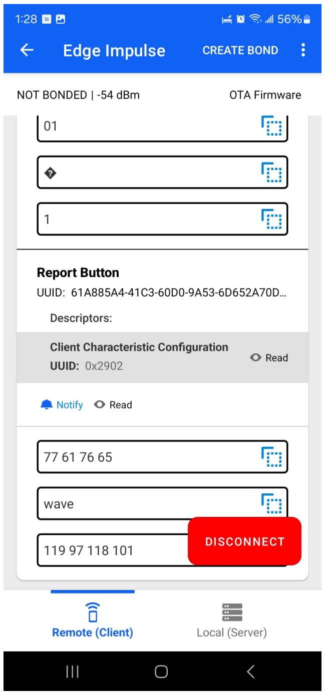
1

Report Button
UUID: 61A885A4-41C3-60D0-9A53-6D652A70D...
Descriptors:
Client Characteristic Configuration
UUID: 0x2902

75 70 64 6F 77 6E

updown

117 112 100 111 119 11



NOT BONDED | -54 dBm

01

�

1

Report Button
UUID: 61A885A4-41C3-60D0-9A53-6D652A70D...
Descriptors:
Client Characteristic Configuration
UUID: 0x2902

77 61 76 65

wave

119 97 118 101

Figure 8.5: Inference Using Si Connect Mobile App

Bibliography

[1] Edge Impulse, "Motion recognition system tutorial: data collection, signal processing, neural network training, and deployment on microcontrollers for gesture recognition," 2025. [Online]. Available: https://docs.edgeimpulse.com/docs/tutorials/end-to-end-tutorials/time-series/continuous-motion-recognition#id-6.-deploying-back-to-device

[2] Edge Impulse, "Anomaly detection with K-means clustering," 2025. [Online]. Available: https://docs.edgeimpulse.com/docs/edge-impulse-studio/learning-blocks/anomaly-detection

[3] Edge Impulse, "Motion classification and anomaly detection on Nicla Vision board: real-world transportation use cases," 2025. [Online]. Available: https://docs.edgeimpulse.com/docs/tutorials/end-to-end-tutorials/time-series/opta-anomaly-detection

Chapter 9

Real-Time Flame Using a Thermal Infrared Sensor

This work was initially done by Batuhan Uzunoglu and later extended for this book chapter.

9.1 Introduction

This project implements a real-time flame detection system using the MLX90640 thermal infrared sensor with a Silicon Labs EFR32MG24 microcontroller. The system captures 32x24 pixel thermal data to classify flame presence or absence. A dataset of thermal snapshots was collected for training a lightweight neural network suitable for embedded deployment. The trained model was converted to TensorFlow Lite for Microcontrollers format and deployed on the EFR32MG24. The microcontroller continuously acquires thermal frames, processes pixel data, and performs on-device inference for scene classification. This approach enables thermal vision-based safety applications on resource-constrained platforms with low power consumption and latency.

Flame detection is critical for safety applications including industrial monitoring, fire suppression systems, and home alarms. Conventional optical sensors, ionization detectors, and photoelectric sensors face limitations from ambient light, dust, smoke, or steam, causing delayed detection or false alarms. Infrared thermal imaging offers a robust alternative by measuring flame heat signatures, reducing susceptibility to visual obstructions and lighting conditions.

This project demonstrates a compact, low-power flame detection system using the MLX90640 32×24 pixel thermal array sensor. The low resolution simplifies processing requirements for microcontroller applications. The sensor pairs with a Silicon Labs EFR32MG24 microcontroller, known for low power consumption and integrated peripherals suitable for IoT and embedded ML tasks.

The methodology involves collecting thermal data under flame and no-flame conditions, training a machine learning model for state differentiation, and deploying this model on the microcontroller for real-time inference using TensorFlow Lite for Microcontrollers. This embedded approach eliminates external processing or cloud connectivity requirements, resulting in a self-contained, responsive, energy-efficient flame detection unit.

9.2 Hardware Components

The system utilizes two primary hardware components. The SparkFun MLX90640 Thermal IR Array provides a 32×24 temperature reading array totaling 768 pixels. It communicates via I2C interface and operates at configurable refresh rates up to 64Hz, though lower rates like 4Hz or 8Hz reduce processing load. Each pixel provides a 16-bit value representing detected infrared radiation, correlating with temperature after processing calibration data from the sensor's EEPROM. The Silicon Labs EFR32MG24 Dev Kit features the EFR32MG24B310F1536IM48 microcontroller with ARM Cortex-M33 core, 1536 kB Flash, 256 kB RAM, and dedicated AI/ML hardware accelerators. It provides I2C peripherals for MLX90640 interfacing and USB VCOM for data output and debugging.

An I2C connection links the EFR32MG24's I2C peripheral (configured as `sl_i2cspm_sensor`) with the MLX90640 sensor. Power is supplied via the development kit's USB connection.

9.3 Software Tools and Libraries

Project development utilizes Simplicity Studio IDE v5 for EFR32 firmware development, debugging, and configuration. The Gecko SDK Suite v4.4.2 provides necessary drivers, middleware, and libraries including platform device drivers for GPIO and I2C, system utilities, services like Sleep Timer and IO Stream, Device Initialization, the TensorFlow Lite Micro component, and I2CSPM drivers for I2C communication. The MLX90640 Library Silicon Labs Port provides functions for sensor interaction over I2C. Python v3.x handles data collection scripting and model training using NumPy for numerical operations, Pandas for data handling, TensorFlow/Keras for model building, Scikit-learn for data splitting, and PySerial for serial communication. TensorFlow Lite for Microcontrollers runs trained ML models on the EFR32MG24, integrated via the Gecko SDK component.

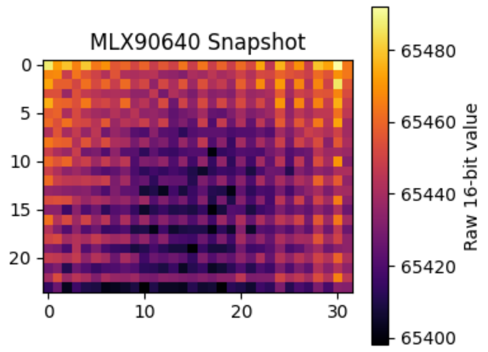
9.4 Data Acquisition

Data collection involves gathering labeled thermal data representing flame and no-flame scenarios.

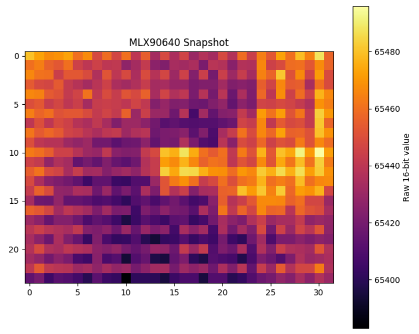
9.4.1 Firmware for Data Streaming

A dedicated Simplicity Studio project continuously reads raw frames from the MLX90640 and transmits them over the serial port. Core logic in `app.c` includes initialization that sets up the I2C peripheral, initializes the MLX90640 sensor, sets refresh rate to 4Hz, performs I2C reset if needed, and starts a periodic timer with 250ms intervals. The timer callback executes periodically, calling data reading functions. Data reading fetches complete frames containing 834 words comprising 768 pixels plus 66 housekeeping values into the frameData buffer, then prints these raw 16-bit values as hexadecimal strings separated by commas and spaces to the serial console.

This firmware transforms the EFR32+MLX90640 setup into a thermal data streaming device.



(a) Thermal capture without flame.



(b) Thermal capture with flame.

Figure 9.1: Side-by-side thermal images: (left) no flame, (right) flame present.

9.4.2 Python Script for Data Collection

The `collect_flame_data.py` script manages data capture on a host computer connected to the EFR32's USB VCOM port. Serial port handling prompts users to select the correct port and baud rate, establishing a PySerial connection. Data parsing reads serial lines, splits by commas and whitespace, and uses regular expressions to identify and parse hexadecimal frame data tokens. Frame assembly accumulates parsed hex values into a deque, extracting 768 pixel values once 834 values are collected, converting to NumPy uint16 arrays, and reshaping to 24×32. Recording prompts users for frame numbers per class, waits for user confirmation after lighting or covering flames, then repeatedly acquires specified frame numbers. Each frame flattens into 768 values and writes to corresponding CSV files with simple headers.

This process generates two CSV files containing multiple rows where each row represents a single thermal frame labeled implicitly by filename.

9.5 Model Training and Conversion

Collected CSV data trains a machine learning model for thermal frame classification.

9.5.1 Training Script (`create_flame_tinyml.py`)

This Python script orchestrates model training and conversion. Data loading reads flame.csv and noflame.csv using Pandas, stacking data into NumPy arrays with corresponding labels, converting raw uint16 pixel values to float32 normalized to 0.0-1.0 range, then splitting into training and validation sets with stratification for class balance. Model building defines a Keras Sequential model with input layer taking flattened 768-element vectors expecting float32 data, two dense layers with 32 and 16 units using ReLU activation, and output layer with single unit using sigmoid activation for binary classification. The model compiles using adam optimizer and binary crossentropy loss. Model training uses model.fit on training data, validating against validation sets for specified epochs and batch size. TFLite conversion uses tf.lite.TFLiteConverter to convert trained Keras models into standard TensorFlow Lite flatbuffer format without quantization, resulting in float32 models. C array conversion reads generated .tflite file bytes and converts them into C byte array definitions, writing arrays with length to .cc source files and printing header file creation instructions.

9.6 Deployment and Inference on EFR32MG24

The final stage deploys converted TFLite models onto the EFR32MG24 for live sensor data inference.

9.6.1 Firmware for Inference

A second Simplicity Studio project uses C++ to facilitate TFLM C++ API integration. This project includes MLX90640 driver libraries and TFLM components targeting EFR32MG24B310F1536IM48 on BRD2601B with Gecko SDK v4.4.2. Key components include TensorFlow Lite Micro, I2CSPM for sl_i2cspm_sensor, Sleep Timer, Micrium OS Kernel, and C++ Support. Model files including generated flame_detector_float_model.cc and corresponding headers must be added to this project.

Core logic in app.cpp includes necessary headers for standard I/O, drivers, MLX90640 library, TFLM core headers, and generated model headers. TFLM setup globals define TENSOR_ARENA_SIZE typically 60KB, declare tensor_arena byte arrays, and create static MicroMutableOpResolver instances sized for model operations. Sensor and model globals declare MLX90640 parameters, frame buffers, sleep timer handles, and pointers for TFLM interpreters, input tensors, and output tensors.

ML model initialization manually sets up TFLM by registering operations required by models, loading model data using tflite::GetModel, instantiating MicroInterpreter instances with models, op resolvers, tensor arenas, and arena sizes, allocating tensors where TENSOR_ARENA_SIZE becomes critical for mapping model tensors onto arenas, obtaining input and output tensor pointers, and verifying tensors have expected types, dimensions, and sizes. Sensor reading reads frames using

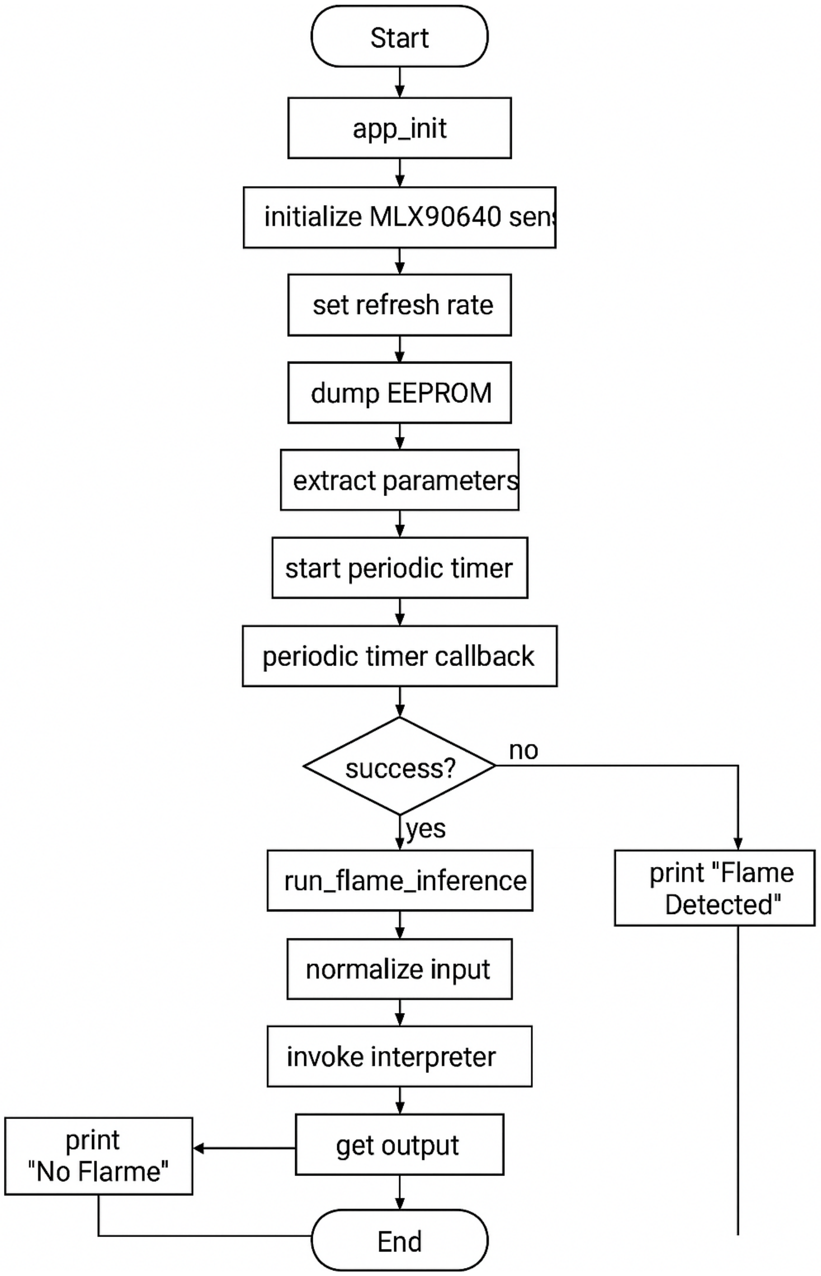


Figure 9.2: Flow chart illustrating the Main Loop ('app_process_action') and how the timer callback drives the core logic.

mlx90640_GetFrameData with retry logic and error logging. Inference execution prepares input by copying raw pixel data from frameData into input tensor data buffers with training normalization, runs inference using interpreter->Invoke(), processes output by reading output probabilities from output tensors, and makes predictions by comparing probabilities against defined thresholds, printing detection results with confidence scores. Timer callbacks trigger periodically, calling data reading functions and inference functions if successful. Application initialization initializes sensors including EEPROM parameter reading, initializes TFLM models, and starts periodic timers triggering callbacks at specified intervals. Main loops remain empty as timer callbacks drive core logic.

9.7 Results and Discussion

The system was tested under laboratory conditions. Model performance from the training script reported validation accuracy of **[Insert Validation Accuracy]** indicating the simple neural network effectively learned to distinguish thermal patterns between flame and no-flame conditions in collected datasets.

Real-time inference on the EFR32MG24 successfully executed inference loops. With 250ms timer intervals and MLX90640 refresh rates set to 8Hz, the system achieved approximately 4 inferences per second. Inference time for interpreter->Invoke() was observed below 50ms, demonstrating EFR32MG24 and float32 TFLM model suitability for this task. The system correctly identified presence and absence of small flames within sensor field of view.

Resource usage showed TFLM arena_used_bytes() reporting 45KB out of allocated 60KB after AllocateTensors(). The flatbuffer model occupies 102,772 bytes approximately 100.4KB in flash as given by flame_detector_float_tflite_len. This demonstrates that float32 TFLM model binaries and activations comfortably fit in EFR32MG24 memory.

Limitations and future work include sensitivity depending on flame size, background temperature, and chosen ML_FLAME_THRESHOLD requiring further evaluation with different flame types or thermal clutter environments. Model complexity could benefit from more complex architectures like convolutional layers suited for spatial data, offering better robustness but requiring more memory and computational power. Quantization converting models to int8 could significantly reduce model size and potentially speed inference, making complex models feasible on EFR32MG24. Data diversity from training data collected under specific conditions requires more robust systems with data from various distances, ambient temperatures, and different flame types or confounding heat sources. Calibration using raw normalized pixel data could potentially improve performance by incorporating sensor full temperature calibration, though adding computational overhead.

9.8 Conclusion

This project successfully demonstrated real-time embedded flame detection system creation using MLX90640 thermal sensors and EFR32MG24 microcontrollers. By collecting application-specific thermal data, training lightweight neural networks, and deploying using TensorFlow Lite for Microcontrollers, we achieved accurate on-device flame presence classification. The system operates with low latency within target MCU memory and processing constraints. This work highlights potential for combining low-resolution thermal imaging with TinyML techniques for developing cost-effective, low-power, robust safety monitoring solutions in various environments.

9.9 Code Listings

9.9.1 Data Acquisition Firmware (app.c Snippet)

```c
#include "app.h"
#include "sl_sleeptimer.h"
#include "sl_i2cspm_instances.h"
#include "mlx90640/mlx90640.h"
#include <stdio.h>

#define TIMER_INTERVAL_MS 250
#define RUN_TIME_MS 5000 // Example: Stop after 5s

int iteration_count = 0;
int MAX_ITERATIONS = (RUN_TIME_MS / TIMER_INTERVAL_MS);
sl_sleeptimer_timer_handle_t mlx90640_timer;
uint16_t frameData[834];

void read_raw_ir_data() {
    // Fetch data; loop or retry if needed
    if (mlx90640_GetFrameData(frameData) == SL_STATUS_OK) {
        for (int i = 0; i < 834; i++) {
            printf("0x%04X, ", frameData[i]);
        }
        printf("\r\n\n---\n");
    } else {
         printf("Error reading frame\r\n---\n");
    }
}

// Ensure callback signature matches sl_sleeptimer prototype
void mlx90640_timer_callback(sl_sleeptimer_timer_handle_t *handle, void *data) {
    (void)handle; // Unused parameter
    (void)data;   // Unused parameter

    if (iteration_count < MAX_ITERATIONS) { // Or run indefinitely
        read_raw_ir_data();
        iteration_count++;
        // For periodic timer, no need to restart here
    } else {
       printf("\n=== Stopping MLX90640 Readings ===\n");
       sl_sleeptimer_stop_timer(&mlx90640_timer);
    }
}

void app_init(void) {
    printf("\n=== MLX90640 Raw Data Stream ===\n");
    // Loop until initialization succeeds
    while (mlx90640_init(sl_i2cspm_sensor) != SL_STATUS_OK) {
        // Add delay or error handling if persistent failure
        sl_sleeptimer_delay_millisecond(500);
    }
    mlx90640_SetRefreshRate(0x06); // 4Hz
    // Start as periodic timer
    sl_sleeptimer_start_periodic_timer_ms(&mlx90640_timer,
                                          TIMER_INTERVAL_MS,
                                          mlx90640_timer_callback,
                                          NULL, 0, 0);
}
```

9.9.2 Data Collection Script (`collect_flame_data.py` Snippet)

```python
import serial
import csv
import re
import numpy as np
from collections import deque

ROWS, COLS = 24, 32
WORDS_PER_FRAME = 834 # 768 pixels + housekeeping
SERIAL_TIMEOUT = 0.2
HEX_PATTERN = re.compile(r"^0[xX][0-9A-Fa-f]{4}$")

def get_one_frame(ser, fifo):
    """
    Read lines until we have collected WORDS_PER_FRAME hex tokens.
    Return a (ROWS, COLS) uint16 NumPy array (first 768 pixels).
    """
    while True:
        try:
            line = ser.readline().decode("utf-8", errors="ignore")
        except serial.SerialException:
            print("Serial read error. Check connection.")
            return None # Indicate error

        if not line: continue # Timeout or empty line

        for tok in re.split(r"[,\s]+", line):
            if HEX_PATTERN.match(tok):
                try:
                    fifo.append(int(tok, 16))
                except ValueError:
                    pass # Ignore invalid hex tokens

        if len(fifo) >= WORDS_PER_FRAME:
            raw =[fifo.popleft() for _ in range(WORDS_PER_FRAME)]
            # Extract only pixel data
            arr = np.array(raw[:ROWS*COLS], dtype=np.uint16)
            return arr.reshape(ROWS, COLS)

def record_frames(ser, fifo, count, out_csv):
    """Record `count` frames and write them to `out_csv`."""
    with open(out_csv, "w", newline="") as f:
        writer = csv.writer(f)
        # Header
        writer.writerow([f"p{i}" for i in range(ROWS*COLS)])        recorded_count = 0
        while recorded_count < count:
            frame = get_one_frame(ser, fifo)
            if frame is not None: # Check if get_one_frame succeeded
                writer.writerow(frame.flatten().tolist())
                recorded_count += 1
                print(f"  * Recorded {recorded_count}/{count} → {out_csv}")
            else:
                # Handle potential persistent read errors if needed
                pass
```

9.9.3 Model Training Script (create_flame_tinyml.py Snippet)

```
import pandas as pd
import numpy as np
import tensorflow as tf
from sklearn.model_selection import train_test_split
import os

# --- Config ---
FLAME_CSV = "flame.csv"
NOFLAME_CSV = "noflame.csv"
OUTPUT_DIR = "output_model"
MODEL_BASE_NAME = "flame_detector"
TEST_SIZE = 0.2
RANDOM_SEED = 42
EPOCHS = 30
BATCH_SIZE = 16
INPUT_SHAPE = (768,) # 24 * 32
DENSE1_UNITS = 32
DENSE2_UNITS = 16

# --- Functions ---
def load_data(flame_csv, noflame_csv):
    df_f = pd.read_csv(flame_csv)
    df_nf = pd.read_csv(noflame_csv)
    X = np.vstack([df_f.values, df_nf.values]).astype(np.float32)
    y = np.concatenate([np.ones(len(df_f)), np.zeros(len(df_nf))])
    X /= 65535.0 # Normalize to [0, 1]
    print(f"Data loaded: {X.shape[0]} samples")
    X_train, X_val, y_train, y_val = train_test_split(
        X, y, test_size=TEST_SIZE, random_state=RANDOM_SEED, stratify=y
    )
    return X_train, X_val, y_train, y_val

def build_model(input_shape):
    model = tf.keras.Sequential([
        tf.keras.layers.Input(shape=input_shape, dtype=tf.float32, name="input_layer"),
        tf.keras.layers.Dense(DENSE1_UNITS, activation="relu", name="dense_1"),
        tf.keras.layers.Dense(DENSE2_UNITS, activation="relu", name="dense_2"),
        tf.keras.layers.Dense(1, activation="sigmoid", name="output_layer"),
    ], name="FlameDetector")
    model.compile(optimizer="adam", loss="binary_crossentropy", metrics=["accuracy"])
    model.summary()
    return model

def convert_to_tflite_float32(model, output_path):
    converter = tf.lite.TFLiteConverter.from_keras_model(model)
    tflite_model = converter.convert()
    with open(output_path, "wb") as f:
        f.write(tflite_model)
    print(f"Float32 TFLite model saved to {output_path} ({len(tflite_model)} bytes)")
    return tflite_model

def convert_to_c_array(tflite_bytes, cc_path, var_name):
    hex_array = [f"0x{b:02x}" for b in tflite_bytes]
    c_str = ", ".join(hex_array)
    c_file = f"""
// Auto-generated C array from TFLite model ({var_name})
alignas(16) const unsigned char {var_name}[] = {
  {c_str}
};
const unsigned int {var_name}_len = {len(tflite_bytes)};
"""
    with open(cc_path, "w") as f:
        f.write(c_file)
    print(f"C array saved to {cc_path}")
```

9.9.4 Inference Firmware (app.cpp Snippet)

```
#include "app.h"
#include "sl_sleeptimer.h"
#include "sl_i2cspm_instances.h"
#include "mlx90640/mlx90640.h"
#include <stdio.h>
```

```
// TFLM Core Headers
#include "tensorflow/lite/micro/micro_interpreter.h"
#include "tensorflow/lite/micro/micro_mutable_op_resolver.h"
#include "tensorflow/lite/schema/schema_generated.h"
#include "tensorflow/lite/micro/micro_log.h"

// Model Header
#include "flame_detector_float_model.h"

// --- Constants ---
#define TIMER_INTERVAL_MS 250
#define ML_INPUT_SIZE   768
#define ML_FLAME_THRESHOLD 0.8f
#define TENSOR_ARENA_SIZE (61440)

// --- Globals ---
static uint8_t tensor_arena[TENSOR_ARENA_SIZE];
static tflite::MicroMutableOpResolver<4> micro_op_resolver;
static tflite::MicroInterpreter* interpreter = nullptr;
static TfLiteTensor* model_input = nullptr;
static TfLiteTensor* model_output = nullptr;
uint16_t frameData[834];
static paramsMLX90640 mlxParams;
sl_sleeptimer_timer_handle_t mlx90640_timer;

sl_status_t init_ml_model() {
    MicroPrintf("Initializing TFLM Model...\n");

    if (micro_op_resolver.AddFullyConnected() != kTfLiteOk) {
        MicroPrintf("ERR: Failed to add FullyConnected op\n");
        return SL_STATUS_FAIL;
    }
    if (micro_op_resolver.AddLogistic() != kTfLiteOk) {
        MicroPrintf("ERR: Failed to add Logistic op\n");
        return SL_STATUS_FAIL;
    }
    MicroPrintf("  Ops added.\n");

    const tflite::Model* model = tflite::GetModel(flame_detector_float_tflite);
    if (!model || model->version() != TFLITE_SCHEMA_VERSION) {
        MicroPrintf("ERR: Model loading failed!\n");
        return SL_STATUS_FAIL;
    }
    MicroPrintf("  Model loaded.\n");

    static tflite::MicroInterpreter static_interpreter(
        model, micro_op_resolver, tensor_arena, TENSOR_ARENA_SIZE);
    interpreter = &static_interpreter;

    if (interpreter->AllocateTensors() != kTfLiteOk) {
        MicroPrintf("ERR: Tensor allocation failed!\n");
        return SL_STATUS_FAIL;
    }

    model_input = interpreter->input(0);
    model_output = interpreter->output(0);

    MicroPrintf("ML Model Initialized Successfully!\n");
    return SL_STATUS_OK;
}

void run_flame_inference() {
    if (!interpreter || !model_input || !model_output) {
        return;
    }

    for (int i = 0; i < ML_INPUT_SIZE; ++i) {
        model_input->data.f[i] = static_cast<float>(frameData[i]) / 65535.0f;
    }

    TfLiteStatus invoke_status = interpreter->Invoke();
    if (invoke_status != kTfLiteOk) {
        MicroPrintf("ERR: Invoke failed\n");
        return;
    }

    float probability_flame = model_output->data.f[0];

    if (probability_flame > ML_FLAME_THRESHOLD) {
        printf(">>> Flame Detected! (Confidence: %.2f)\n", probability_flame);
    } else {
        printf(">>> No Flame Detected (Confidence: %.2f)\n", probability_flame);
    }
    printf("---\n");
}

void app_init(void) {
    printf("\n=== MLX90640 Flame Detection App ===\n");

    while (mlx90640_init(sl_i2cspm_sensor) != SL_STATUS_OK) {
        sl_sleeptimer_delay_millisecond(500);
    }

    mlx90640_SetRefreshRate(0x05);
    if (init_ml_model() != SL_STATUS_OK) {
        MicroPrintf("FATAL: ML Model Init Failed.\n");
        while(1);
    }

    sl_sleeptimer_start_periodic_timer_ms(&mlx90640_timer,
                                           TIMER_INTERVAL_MS,
                                           mlx90640_timer_callback,
                                           NULL, 0, 0);

    printf("Initialization Complete. Running...\n");
}
```