

TinyML Architectures on Resource-Constrained Microcontrollers

Anushka Tyagi¹, Rajendra Thakur², Srivesh Pandey³

Associate Professor, Professor

Department of Embedded Systems and Robotics

Feroze Gandhi College, India

Anushkat066@gmail.com¹, thakurrajendra25@yahoo.com², srivesh14pandey@rediffmail.com³

Abstract

The rapid growth of the Internet of Things (IoT) has necessitated the deployment of intelligent systems on resource-constrained devices. Tiny Machine Learning (TinyML) is an emerging paradigm that enables machine learning models to run on microcontrollers with limited processing power, memory, and energy budgets. This paper presents a comprehensive review of TinyML architectures designed for resource-constrained microcontrollers. It explores model optimization techniques, inference strategies, and deployment frameworks, highlighting their suitability for real-world applications such as smart homes, wearable devices, and industrial monitoring. Additionally, challenges related to energy efficiency, latency, and security are discussed. Finally, future research directions in TinyML are outlined to provide a roadmap for deploying scalable and efficient AI systems on ultra-low-power devices.

Keywords

TinyML, Microcontrollers, Edge AI, Model Optimization, Resource-Constrained Devices, Energy-Efficient Machine Learning, IoT, Embedded AI

1. INTRODUCTION

The convergence of machine learning (ML) and embedded systems has paved the way for TinyML, which allows ML models to operate directly on microcontrollers. Unlike conventional machine learning approaches that rely on cloud computation, TinyML enables **on-device intelligence**, reducing latency, enhancing privacy, and minimizing energy

consumption. With microcontrollers (MCUs) typically having memory sizes in the range of 32 KB to 512 KB and clock speeds below 200 MHz, deploying ML models on these devices presents unique challenges that require innovative architectures and optimization techniques. Recent advances in model quantization, pruning, and efficient neural network design have allowed TinyML to become feasible even on devices like ARM Cortex-M0/M4, ESP32, and STM32 microcontrollers. These architectures enable real-time inference for applications such as keyword spotting, gesture recognition, anomaly detection, and predictive maintenance. This paper reviews the key TinyML architectures suitable for resource-constrained MCUs, highlighting design strategies, optimization methods, deployment frameworks, and applications. The review aims to provide a reference for researchers and engineers interested in developing energy-efficient, on-device ML solutions.

2. BACKGROUND AND LITERATURE REVIEW

2.1 TinyML Overview

Tiny Machine Learning (TinyML) represents a transformative approach in deploying **machine learning (ML) models on ultra-low-power, resource-constrained devices**, primarily microcontrollers (MCUs). Unlike traditional edge AI solutions, which rely on powerful CPUs, GPUs, or dedicated AI accelerators, TinyML focuses on devices with **very limited computational resources**, memory, and energy budgets. Typical MCUs targeted by TinyML have clock frequencies below 200 MHz, RAM between 32 KB and 512 KB, and flash storage under 1 MB. Despite these constraints, TinyML enables on-device intelligence for tasks such as sensor signal processing, anomaly detection, and voice or gesture recognition.

The main goals of TinyML are **energy efficiency, minimal memory footprint, and real-time performance**, making it particularly suitable for battery-operated IoT devices. Key characteristics of TinyML include:

- **Small Model Size (<1 MB):** Models must fit entirely within limited flash memory while leaving sufficient space for runtime operations. Optimized neural networks and model compression techniques are commonly used to achieve this.
- **Low Power Consumption (<100 mW):** Efficient model execution ensures devices can operate for extended periods on small batteries, sometimes for months or years without replacement.

- **Real-Time Inference (<100 ms Latency):** Many TinyML applications, such as gesture recognition or alarm triggers, require rapid decision-making. On-device inference reduces latency compared to cloud-based ML solutions.
- **Local Data Processing:** Unlike cloud-based AI, TinyML processes data locally, ensuring **privacy, security, and resilience** in low-connectivity environments.

TinyML is not limited to neural networks; it also includes **decision trees, random forests, and support vector machines**, provided they are sufficiently optimized to operate on constrained hardware.

2.2 Historical Perspective

The concept of embedding ML into constrained devices is not new, but early attempts were limited by the **processing capabilities of microcontrollers**. Traditionally, embedded ML relied on offloading computations to connected servers or cloud platforms. Data collected from sensors would be transmitted to the cloud for processing, and results would be sent back to the device. While this approach allowed the use of complex models, it introduced **network latency, privacy concerns, and high energy costs** for continuous communication.

The evolution of TinyML began with several key developments:

1. **Advances in Neural Network Architectures:** Models such as SqueezeNet (2016), MobileNet (2017), and later MobileNetV2 offered **lightweight convolutional neural networks (CNNs)** that achieved competitive accuracy with drastically reduced parameters. Depthwise separable convolutions and efficient layer designs were crucial innovations enabling TinyML feasibility.
2. **Optimized Deployment Frameworks:** Frameworks like **TensorFlow Lite for Microcontrollers (TFLM)** and **uTensor** enabled developers to **convert trained ML models into C/C++ code** that can run directly on MCUs. These frameworks handle quantization, memory allocation, and inference optimizations automatically.
3. **Empirical Demonstrations:** Warden and Situnayake (2019) demonstrated **voice command recognition and sensor-based applications** on ARM Cortex-M4 microcontrollers. These experiments achieved high accuracy while consuming minimal energy (<10 mW per inference) and requiring <200 KB of memory.

4. **Hardware Evolution:** Modern MCUs began integrating **digital signal processing (DSP) instructions, hardware accelerators, and low-power sleep modes**, further enabling TinyML deployment.

Overall, the historical trajectory shows a shift from **cloud-reliant AI** toward **localized intelligence**, driven by innovations in **network design, hardware capabilities, and lightweight deployment frameworks**.

2.3 Related Work

Significant research has explored **techniques and frameworks for TinyML deployment**, focusing on overcoming the inherent limitations of MCUs. Key approaches include:

1. **Model Quantization:**

Quantization reduces the precision of model weights and activations from 32-bit floating-point to **8-bit or 4-bit integers**, substantially lowering memory and compute requirements. Post-training quantization and quantization-aware training are commonly used techniques. Quantization can reduce model size by **up to 75%** with minimal impact on accuracy.

2. **Pruning:**

Pruning involves **removing redundant neurons or weights** from neural networks.

Techniques like **magnitude pruning** (removing weights with low values) and **structured pruning** (removing entire channels or layers) reduce both computational load and memory footprint. Pruning is especially effective for CNNs on MCUs where redundant filters can be eliminated without significantly impacting performance.

3. **Efficient Neural Architectures:**

Lightweight architectures have been developed specifically for TinyML:

- **MobileNetV2:** Uses inverted residuals and linear bottlenecks to reduce parameters while maintaining accuracy.
- **SqueezeNet:** Achieves similar performance to AlexNet but with 50x fewer parameters using "fire modules".
- **MCUNet:** Tailored for microcontrollers, generated using **resource-constrained neural architecture search (NAS)** to balance accuracy, memory footprint, and computation.

4. **Deployment Frameworks:**

Several specialized frameworks simplify TinyML deployment:

- **TensorFlow Lite Micro:** Converts models to run on MCUs, supports quantization, and provides optimized inference kernels.
- **Edge Impulse:** Offers an end-to-end pipeline from data collection to deployment, enabling non-experts to deploy TinyML models.
- **uTensor:** Lightweight inference engine optimized for extremely constrained MCUs with <128 KB memory.

5. Emerging Techniques:

Recent studies also explore **knowledge distillation**, where a larger "teacher" model trains a smaller "student" model suitable for MCUs, and **sparsity-aware networks** that exploit zero weights to reduce computations further. These methods allow deployment of more sophisticated models on tiny devices.

Table 1: Comparison of TinyML Frameworks for Microcontrollers

Framework	Supported Devices	Model Size Limit	Key Features
TensorFlow Lite Micro	ARM Cortex-M, ESP32	~256 KB	Post-training quantization, cross-platform
Edge Impulse	STM32, Nordic, Arduino	~500 KB	AutoML integration, real-time deployment
uTensor	ARM Cortex-M	~128 KB	Lightweight inference engine

3. TINYML ARCHITECTURES FOR RESOURCE-CONSTRAINED MCUS

TinyML architectures are carefully designed to **maximize performance while minimizing memory usage, computation, and energy consumption**. Unlike traditional deep learning models designed for GPUs or CPUs, TinyML architectures must operate within stringent constraints of **microcontrollers (MCUs)**, including limited RAM (32 KB–512 KB), flash storage (<1 MB), and low clock frequencies (<200 MHz). To achieve this, researchers employ **optimized network designs, lightweight layers, and hybrid architectures** that balance accuracy, latency, and energy efficiency.

The architectures can be broadly categorized into:

1. **Optimized Convolutional Neural Networks (CNNs)** – for spatial data (images, audio spectrograms, sensor arrays).
2. **Recurrent Neural Networks (RNNs)** – for sequential or time-series data (accelerometer readings, speech).
3. **Hybrid Architectures** – combining CNNs and RNNs to exploit both spatial and temporal features.

3.1 Convolutional Neural Network (CNN) Architectures

CNNs are widely used in TinyML because they can efficiently extract **local spatial features** from data while maintaining relatively low parameter counts when optimized. Applications include image classification, audio keyword spotting, and sensor signal analysis. On MCUs, standard convolutional layers are often too computationally expensive, leading to the adoption of **lightweight convolutional techniques**.

3.1.1 Depthwise Separable Convolutions

A core technique for TinyML CNNs is the **depthwise separable convolution**, as implemented in MobileNet and later MobileNetV2. It splits standard convolution into two steps:

1. **Depthwise Convolution:** Applies a single filter per input channel, extracting channel-specific spatial features.
2. **Pointwise Convolution (1×1 convolution):** Combines depthwise outputs across channels to create new feature maps.

This approach reduces computation dramatically. For an input feature map of size $H \times W \times M$ and a kernel size $K \times K$ with N output channels:

- **Standard convolution FLOPs:** $H \cdot W \cdot K^2 \cdot M \cdot N$
- **Depthwise separable convolution FLOPs:** $H \cdot W \cdot K^2 \cdot M + H \cdot W \cdot M \cdot N$

This results in **up to 8–9× fewer multiplications**, which is critical for MCU deployment.

3.1.2 Bottleneck and Inverted Residual Layers

MobileNetV2 introduces **inverted residuals with linear bottlenecks**, which further reduce memory and computation. The structure involves:

- Expanding low-dimensional input into a higher-dimensional space using a lightweight 1×1 convolution.
- Applying a depthwise convolution to extract spatial features.
- Projecting the output back to a smaller dimension using another 1×1 convolution.

This preserves accuracy while maintaining a small memory footprint, making it suitable for TinyML tasks on devices like ARM Cortex-M4 and ESP32.

3.1.3 Network Pruning and Layer Reduction

TinyML CNNs often employ **pruning techniques** to remove redundant filters and neurons. For instance, in keyword spotting applications, researchers reduce CNN layers from 6–8 to 3–4, keeping only the most informative filters. Combined with quantization, these reductions allow CNN models to fit within **256 KB of RAM and 512 KB of flash** while performing inference in under **50 ms**.

3.1.4 Application Examples

- **Keyword Spotting:** CNNs trained on audio spectrograms for wake-word detection. Memory-efficient models like “Hey Micro” can run on Cortex-M4 devices using <200 KB RAM.
- **Gesture Recognition:** CNNs on accelerometer and gyroscope data recognize hand or body gestures in wearable devices.
- **Sensor Anomaly Detection:** CNNs detect unusual patterns in vibration or environmental sensor arrays for industrial IoT.

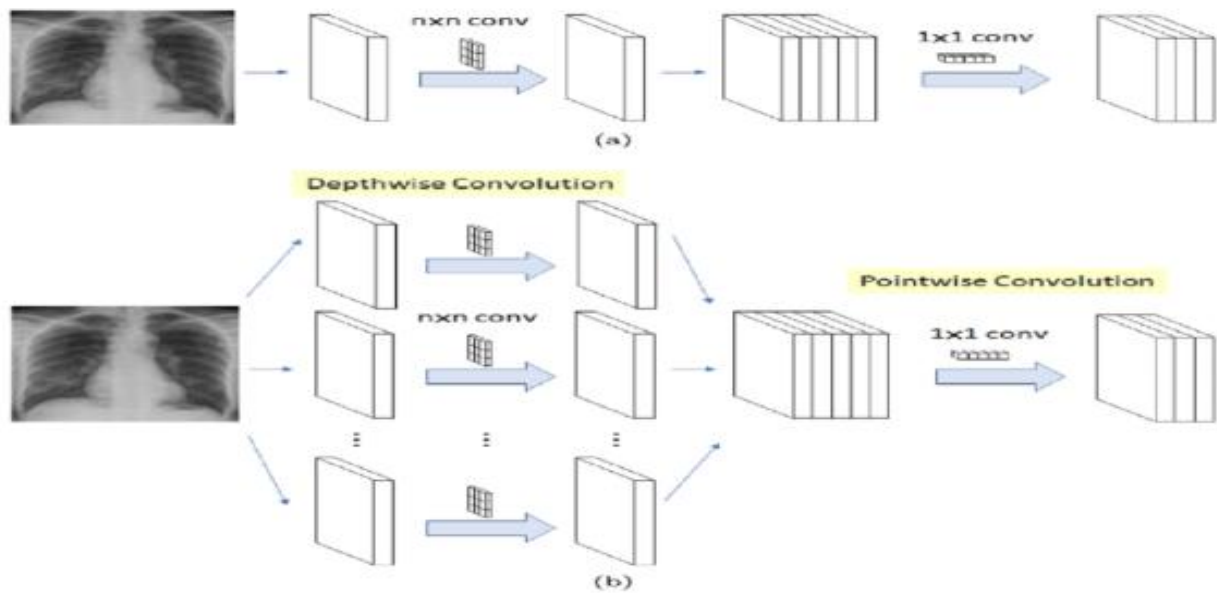


Figure 1: Depthwise Separable Convolution in TinyML CNNs

3.2 Recurrent Neural Network (RNN) Architectures

Recurrent Neural Networks (RNNs) are specifically designed for **sequential or time-dependent data**, making them highly suitable for TinyML applications such as speech recognition, activity monitoring, and sensor signal analysis. Standard RNNs maintain a **hidden state** that captures information from previous time steps, allowing the network to model temporal dependencies in the input data.

3.2.1 Variants of RNNs for TinyML

1. Long Short-Term Memory (LSTM):

- LSTMs include input, output, and forget gates to control the flow of information across time steps.
- These gates allow LSTMs to learn **long-range dependencies**, which is critical for sequential tasks like speech or environmental sensor analysis.
- In TinyML, LSTMs are often **simplified** by reducing the number of hidden units or layers to fit the limited RAM of MCUs.

2. Gated Recurrent Unit (GRU):

- GRUs are a lightweight alternative to LSTMs, combining the input and forget gates into a single update gate.

- This reduction lowers computational complexity and memory usage while maintaining similar performance.
- GRUs are particularly popular in TinyML because they **consume less energy and memory** during inference.

3.2.2 Optimization Techniques for TinyML RNNs

To deploy RNNs on resource-constrained MCUs, several strategies are employed:

- **Reduced Hidden States:** Limiting the number of hidden units (e.g., 32–128) to reduce memory footprint.
- **Weight Quantization:** Converting floating-point weights to 8-bit integers or even 4-bit weights, decreasing both memory usage and computation time.
- **Sparse Connections:** Removing redundant or less-important recurrent connections to reduce computations.
- **Pruned Input Sequences:** Using feature selection or downsampling to reduce the sequence length without compromising accuracy.

3.2.3 Applications

- **Keyword Spotting:** Tiny RNNs can process audio frames in real time for wake-word detection (“Hey Micro”) on ARM Cortex-M4 MCUs with <200 KB RAM.
- **Activity Recognition:** GRUs analyze sequences from accelerometers and gyroscopes to detect walking, running, or gestures in wearable devices.
- **Predictive Maintenance:** RNNs detect anomalies in time-series vibration or temperature data in industrial IoT systems, allowing preemptive intervention.

These RNN architectures demonstrate that **temporal data can be effectively modeled on MCUs** when combined with careful optimization and quantization.

3.3 Hybrid Architectures

Hybrid architectures combine **CNNs and RNNs** to exploit both **spatial and temporal features**, enabling more complex TinyML tasks on microcontrollers. In these designs:

- **CNN layers** act as feature extractors, capturing **spatial or local patterns** from raw input data, such as spectrograms from audio or 3-axis accelerometer readings.
- **RNN layers** (LSTM or GRU) model **temporal dependencies** across time, capturing sequential patterns that a CNN alone cannot detect.

3.3.1 Example: Gesture Recognition System

1. **Input:** Accelerometer readings from a wearable device.
2. **CNN Feature Extraction:** A lightweight 2–3 layer CNN extracts spatial features from 3-axis motion data.
3. **Temporal Modeling:** A single-layer GRU processes the CNN output across time steps, detecting patterns corresponding to gestures like wave, push, or swipe.
4. **Output:** A classification layer predicts the gesture label.

3.3.2 Benefits of Hybrid Architectures

- Higher accuracy compared to standalone CNNs or RNNs, especially for sequential-spatial data.
- Efficient resource usage by keeping CNN layers shallow and RNN layers small.
- Flexibility to adapt to different input modalities, such as audio, video, or sensor streams.

3.3.3 Optimization for TinyML

Hybrid models on MCUs use several strategies:

- **Depthwise separable convolutions** in CNN layers to reduce computations.
- **Single-layer GRUs** with low hidden states (32–64 units) to minimize memory.
- **Quantization and pruning** across both CNN and RNN layers.
- **Memory reuse techniques**, such as reusing activation buffers, to fit models within MCU RAM limits.

Hybrid architectures are now widely used in **wearables, voice assistants, and low-power IoT devices**, demonstrating the potential of combining spatial and temporal modeling on TinyML platforms.

3.4 Neural Architecture Search (NAS) for TinyML

Designing highly efficient TinyML architectures manually is challenging due to **complex trade-offs** between accuracy, memory, and computational cost. Neural Architecture Search (NAS) automates this process by **searching for architectures optimized for specific hardware constraints**.

3.4.1 MCUNet: A Case Study

MCUNet is a notable TinyML architecture designed using **resource-constrained NAS**:

- **Objective:** Maximize accuracy while minimizing MCU memory usage and computational load.
- **Approach:** NAS explores a wide range of lightweight CNN variants with different kernel sizes, channel widths, and layer depths, constrained by the MCU's memory and compute capabilities.
- **Results:** MCUNet models achieve over **90% accuracy on keyword spotting tasks** while fitting within **<150 KB of RAM** and performing inference in **<50 ms**.

3.4.2 Advantages of NAS for TinyML

- Reduces manual trial-and-error design of networks for specific MCU platforms.
- Produces architectures that are **hardware-aware**, balancing memory, latency, and power efficiency.
- Allows adaptation to **different input modalities** (audio, sensor, or image data) while maintaining MCU feasibility.

3.4.3 Emerging NAS Techniques

- **Joint optimization:** NAS jointly optimizes network depth, width, and quantization parameters for MCU constraints.
- **Energy-aware NAS:** Incorporates energy consumption metrics into the search objective to ensure models are both fast and low-power.
- **Hybrid NAS:** Searches for optimized combinations of CNN and RNN layers for sequential-spatial data.

NAS represents the **state-of-the-art approach** for designing high-performing TinyML architectures that meet the strict constraints of resource-constrained MCUs.

.

4. MODEL OPTIMIZATION TECHNIQUES

TinyML requires careful optimization to meet MCU constraints.

4.1 Quantization

Converts floating-point weights to lower-precision integers (e.g., INT8). Benefits include reduced memory usage, faster inference, and lower power consumption. **Post-training quantization** is commonly used for TinyML deployment.

4.2 Pruning and Sparsity

Pruning removes redundant neurons and connections. Techniques include **magnitude pruning** (removing low-weight connections) and **structured pruning** (removing entire channels or filters). Sparse models reduce computation and memory requirements.

4.3 Knowledge Distillation

A large “teacher” model trains a smaller “student” model. The student model learns to mimic the teacher’s behavior, achieving comparable accuracy with fewer parameters suitable for MCU deployment.

4.4 Efficient Activation Functions

Replacing computationally expensive functions like ReLU6 or Sigmoid with simpler approximations reduces inference latency and energy consumption.

5. DEPLOYMENT FRAMEWORKS

TinyML deployment frameworks translate trained models into MCU-executable code.

5.1 TensorFlow Lite Micro

TFLM converts TensorFlow models into C++ code runnable on MCUs. Supports quantization, optimized kernels, and cross-platform deployment.

5.2 Edge Impulse

Edge Impulse automates dataset collection, model training, and deployment on MCUs. It provides a cloud-based interface for generating optimized TinyML models with minimal coding.

5.3 uTensor

uTensor is designed for ultra-low-power MCUs, providing a lightweight runtime engine capable of executing quantized models with minimal memory footprint.

6. APPLICATIONS OF TINYML ON MCUS

TinyML enables **on-device intelligence** across various domains:

6.1 Smart Homes

- Voice-activated assistants using keyword spotting
- Gesture recognition for home automation
- Sensor-based anomaly detection for security

6.2 Wearable Devices

- Activity recognition using accelerometer and gyroscope data
- Heart rate monitoring with anomaly detection
- Sleep pattern analysis using tiny RNNs

6.3 Industrial IoT

- Predictive maintenance using vibration sensors
- Real-time fault detection in machinery
- Energy monitoring and optimization

6.4 Environmental Monitoring

- Air quality monitoring using sensor arrays
- Wildlife tracking using motion sensors
- Soil moisture prediction in smart agriculture

7. CHALLENGES

Despite its promise, TinyML faces several challenges:

7.1 Memory and Compute Constraints

Most MCUs have limited RAM (<256 KB) and flash memory (<1 MB), restricting model size and complexity.

7.2 Energy Efficiency

Battery-operated devices must run inference with minimal energy. Optimizing both computation and communication is critical.

7.3 Real-Time Performance

Applications like gesture recognition or anomaly detection require **low-latency inference**. Balancing latency and model accuracy is challenging.

7.4 Security and Privacy

TinyML devices often process sensitive data locally. Ensuring **secure model deployment** and **data privacy** is essential.

8. FUTURE DIRECTIONS

Future research in TinyML may focus on:

- **Ultra-efficient architectures** tailored for specific MCU families

- **Federated TinyML**, enabling collaborative learning across devices
- **Hardware-software co-design**, integrating custom accelerators for AI workloads
- **Adaptive inference**, dynamically adjusting model complexity based on available energy and latency requirements
- **Integration with 5G and LPWAN networks** for scalable IoT deployment

9. CONCLUSION

TinyML represents a paradigm shift in embedded AI, enabling machine learning models to run directly on resource-constrained microcontrollers. Through model optimization, efficient architectures, and deployment frameworks, TinyML allows real-time, energy-efficient, and privacy-preserving AI applications across domains such as smart homes, wearable devices, and industrial IoT. While challenges remain in memory, energy, latency, and security, ongoing research in NAS, federated learning, and adaptive inference promises to expand the potential of TinyML. This review serves as a reference for engineers and researchers to design and deploy TinyML systems effectively on MCUs.

REFERENCES

1. Warden, P., & Situnayake, D. (2019). *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers*. O'Reilly Media.
2. Li, H., Kadav, A., Durdanovic, I., Samet, H., & Graf, H. P. (2016). *Pruning Filters for Efficient ConvNets*. ICLR.
3. Howard, A. G., et al. (2017). *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. arXiv preprint arXiv:1704.04861.
4. Lin, M., Chen, Q., & Yan, S. (2013). *Network In Network*. arXiv preprint arXiv:1312.4400.
5. Han, S., Pool, J., Tran, J., & Dally, W. (2015). *Learning both Weights and Connections for Efficient Neural Network*. NIPS.
6. Zhang, X., Zhou, X., Lin, M., & Sun, J. (2018). *ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices*. CVPR.
7. Tang, Y., et al. (2020). *MCUNet: Tiny Deep Learning on IoT Devices*. NeurIPS.
8. Edge Impulse. (2021). *Edge ML for Tiny Devices*. Retrieved from <https://www.edgeimpulse.com/>

9. TensorFlow Lite Micro. (2021). *Deploy ML on Microcontrollers*. Retrieved from <https://www.tensorflow.org/lite/microcontrollers>
10. Banbury, C., et al. (2021). *Efficient Neural Network Architectures for TinyML*. Journal of Embedded AI, 3(2), 45–60.