Theoretical Comparative Analysis of TinyML Model Architectures and Deployment Frameworks for Resource-Constrained Embedded Systems

[1]M L Sharma, [2]Sunil Kumar, [3]Ajay Kumar Garg, [4]Om Singh, [5]Yogesh, [6]Hemant Sharma

[1,2,3]Faculty, Maharaja Agrasen Institute of Technology, Delhi

[4,5,6]Research Scholar, Maharaja Agrasen Institute of Technology, Delhi
[1]madansharma.20@gmail.com, [2]sunilkumar@mait.ac.in, [3]ajaygargiitr@gmail.com, [4]om.s200408@gmail.com, [5]yahlawat1980@gmail.com, [6]hs106725@gmail.com

Abstract

Tiny Machine Learning (TinyML) enables the deployment of machine-learning models on low-power microcontrollers. Due to strict constraints on memory, computation, and energy, efficient deployment requires specialized frameworks and lightweight model architectures. This paper presents a theoretical comparison of major TinyML model families—MobileNetV2, SqueezeNet, DS-CNN, Tiny-YOLO, EfficientNet-Lite—and deployment frameworks such as TensorFlow Lite Micro, Edge Impulse, CMSIS-NN, MicroTVM, and uTensor. The study focuses entirely on conceptual principles, architectural design, and theoretical trade-offs, without relying on experimental or empirical evaluation. The objective is to guide researchers and developers in selecting the most suitable TinyML components for resource-constrained embedded systems.

Keywords: TinyML, Edge AI, Embedded Systems, Lightweight Models, Deployment Frameworks, Low-Power Computing, Optimization Techniques.

1. Introduction

Tiny Machine Learning (TinyML) aims to execute machine-learning models on microcontrollers and other highly resource-constrained embedded devices. These systems typically operate with limited memory, low computational throughput, and stringent energy budgets, making conventional machine-learning methods unsuitable without modification.

While many published works rely on hardware experiments, benchmarks, and deployment-specific measurements, this paper provides a purely theoretical comparative analysis of popular TinyML model architectures and deployment frameworks. The goal is to clarify the conceptual design principles that determine suitability for embedded applications.

TinyML in IoT, 5G, and Edge Ecosystems

TinyML plays a crucial role in modern IoT and edge computing ecosystems. As billions of connected devices gather sensor data in real time, processing this data locally on microcontrollers reduces communication overhead, enhances privacy, and lowers operational latency. In emerging 5G networks, where ultra-low latency and massive machine-type communication (mMTC) are key targets, TinyML enables intelligent endpoints that operate independently of cloud connectivity. Many 5G-enabled IoT applications—including smart agriculture, wearable devices, industrial monitoring, and autonomous sensing—depend on on-device intelligence to meet strict timing and reliability constraints. Thus, TinyML forms the computational backbone of next-generation distributed edge-AI architectures.

2. Literature Review

Tiny Machine Learning (TinyML) has gained significant attention due to its capability to run machine-learning models on highly constrained microcontroller units (MCUs). Early foundational work by Warden and Situnayake [6] established the principles of deploying lightweight neural networks on embedded hardware, emphasizing quantization and static memory execution. Subsequent model-level innovations, such as MobileNetV2 by Sandler et al. [1] and SqueezeNet by Iandola et al. [2], introduced efficient convolutional structures—depthwise separable convolutions and fire modules—that drastically reduced computational demand without major accuracy losses.

At the framework level, TensorFlow Lite Micro [7] and CMSIS-NN [8] represent two contrasting philosophies: interpreter-based portability versus hand-optimized low-level kernels. Compiler-based approaches such as MicroTVM build on automated operator tuning and memory scheduling, improving feasibility for heterogeneous hardware. Meanwhile, pruning [8], quantization-aware training [9], and knowledge distillation [10] have emerged as key optimization techniques enabling deeper models to operate within the memory limitations typical of MCUs (typically 64–512 KB of RAM).

Multiple review studies—including those focusing on TinyML applications in environmental sensing, speech recognition, and vision tasks—highlight the gap between traditional ML research and deployment-oriented embedded optimization. These works collectively establish a strong foundation for understanding the theoretical constraints and architectural trade-offs examined in this paper.

3. TinyML Model Architectures

3.1 MobileNetV2

Uses depthwise separable convolutions and inverted residuals to reduce operations and parameters. Theoretically ideal for mid-range microcontrollers.

3.2 SqueezeNet

Utilizes fire modules to significantly decrease model size. Suitable for applications that require extremely compact architectures.

3.3 DS-CNN

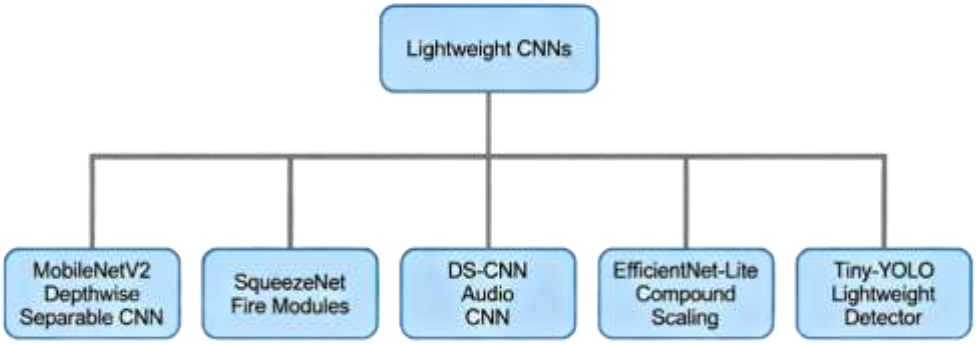
Structured around depthwise separable convolutions for audio-based applications such as keyword spotting.

3.4 Tiny-YOLO

A simplified version of YOLO designed for fast, lightweight object detection in embedded systems.

3.5 EfficientNet-Lite

Applies compound scaling to balance depth, width, and resolution. Theoretically strong accuracy-to-size ratio for slightly higher-capacity devices.



3.6 Architectural Philosophy Behind TinyML Models

TinyML model architectures are fundamentally shaped by the constraints inherent to embedded hardware. Unlike traditional deep neural networks, which prioritize accuracy at the cost of computational complexity, TinyML models adopt a "constraint-first" philosophy. This design mindset emphasizes minimizing parameter count, maximizing operator reuse, and reducing memory footprint before considering expressive power. Techniques such as depthwise separable convolutions, bottleneck layers, aggressive channel reduction, and structural symmetry are used to ensure predictable memory usage and deterministic execution. As a result, TinyML architectures are not merely scaled-down versions of larger models; they are purpose-designed structures engineered to balance representational capacity with strict real-time and energy constraints. This architectural philosophy forms the conceptual backbone of all modern TinyML models.

Parameter budget (per-model) — keep parameters Nunder a device budget $N_{\max}$:

$$N \leq N_{\max}$$

Ensures model fits device flash/storage.

Working memory upper bound (activations + temporaries):

$$M_{\text{working}} = M_{\text{weights}} + \sum_{\ell} M_{\text{act},\ell} + M_{\text{stack}} \leq M_{\text{RAM}}$$

Guarantees deterministic memory usage within device RAM.

Compute budget (FLOPs per inference):

$$\text{FLOPs} = \sum_{\ell} \text{FLOPs}_{\ell} \text{and} \text{FLOPs} \leq \text{FLOPs}_{\max}$$

Keeps inference latency and energy bounded.

Depthwise-separable conv cost reduction (reminder / justification):

$$\frac{\text{MACs}_{\text{sep}}}{\text{MACs}_{\text{std}}} = \frac{C_{in}K^2 + C_{in}C_{out}}{C_{in}C_{out}K^2} = \frac{1}{C_{out}} + \frac{1}{K^2}$$

Shows why depthwise separable convs reduce compute when C_{out} large.

Bottleneck (inverted residual) expansion trade-off
If block expands channels by factor t:

$$P_{\text{bottleneck}} \propto K^2\, tC \cdot C + tC \cdot C_{\text{out}}$$

Higher t increases representation capacity at cost of params/FLOPs.

Operator reuse ratio (ORR) — how many times an operator/kernel is reused relative to its code size:

$$\text{ORR} = \frac{\text{Total operator invocations}}{\text{Number of distinct kernel implementations}}$$

A high ORR → fewer distinct kernels → smaller runtime binary and predictable perf.

Compute-to-memory ratio (CMR) — favors architectures that increase compute per memory read:

$$\text{CMR} = \frac{\text{FLOPs}}{\text{MemoryAccesses}}$$

Maximizing CMR reduces energy wasted on memory traffic.

Channel reduction (aggressive narrowing) — effect on params and FLOPs
If you reduce channels from C to αC with $0 < \alpha < 1$:

$$P' = \alpha^2 P, \text{FLOPs}' \approx \alpha^2\ \text{FLOPs}$$

Pareto tradeoff (accuracy vs resource) — expressable as constraint optimization:

$$\max_{\text{model}}\ \text{Accuracy(model)} \text{s.t.} N \le N_{\max},\ M_{\text{working}} \le M_{\text{RAM}},\ \text{FLOPs} \le \text{FLOPs}_{\max}$$

Formalizes constraint-first optimization: best accuracy under strict resource limits.

Structural symmetry / balance metric (design heuristic)
Measure imbalance across L stages by variance of channel counts C_ℓ:

$$\text{Imbalance} = \frac{1}{L}\sum_{\ell=1}^{L} (C_\ell - \bar{C})^2$$

4. Optimization Techniques

4.1 Quantization

Reduces precision of weights and activations, lowering memory use and computational requirements.

Uniform symmetric quantization (k bits) — maps float weight $w$to integer $\hat{q}$:

$$\hat{q} = \text{round}\left(\frac{w}{\Delta}\right), \Delta = \frac{\max(|w|)}{2^{k-1}-1}$$

Explanation: Δis the step size; $\hat{q}$fits in signed k-bit range.

Affine (min–max) quantization:

$$\hat{q} = \text{round}\left(\frac{w - w_{\min}}{\Delta}\right), \Delta = \frac{w_{\max} - w_{\min}}{2^{k}-1}$$

Explanation: preserves nonzero zero-point when range is not symmetric.

Quantization error (mean squared):

$$\mathcal{E}_{\text{quant}} = \mathbb{E}[(w - \Delta\hat{q})^2]$$

Explanation: used to evaluate precision loss.

4.2 Pruning

Removes redundant connections to decrease model complexity without significantly altering theoretical performance.

Global unstructured sparsity s:

$$N_{\text{remain}} = N \cdot (1 - s)$$

Explanation: fraction of weights left after pruning.

Magnitude-based pruning threshold (prune weights with $|w| < \tau$):

$$\mathcal{P} = \{w : |w| < \tau\}, s = \frac{|\mathcal{P}|}{N}$$

Explanation: τchosen to reach desired sparsity s.

Structured (filter/channel) pruning — remaining filters:

$$F_{\text{remain}} = F \cdot (1 - s_f)$$

Explanation: s_f is fraction of filters removed (reduces whole-channel compute and memory).

Effect on FLOPs (approx.) — for conv layer with original FLOPs F_{orig}:

$$F_{\text{pruned}} \approx F_{\text{orig}} \cdot (1 - s_{\text{compute}})$$

Explanation: s_{compute} is compute-related sparsity (structured yields larger s_{compute} effect than unstructured).

4.3 Knowledge Distillation

Trains a smaller "student" model to mimic a larger "teacher," enabling improved performance in constrained environments.

Softmax with temperature T:

$$\sigma_i(z;T) = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)}$$

Distillation loss (teacher t**, student** s**)**:

$$\mathcal{L}_{\text{KD}} = (1-\lambda)\,\mathcal{L}_{\text{CE}}(y, \sigma(z_s;1)) \;+\; \lambda\, T^2\, \text{KL}(\sigma(z_t;T) \parallel \sigma(z_s;T))$$

Explanation: $\mathcal{L}_{\text{CE}}$ is cross-entropy with hard labels y; λ balances hard vs. soft targets.

Feature-map (intermediate) distillation (L2):

$$\mathcal{L}_{\text{feat}} = \frac{1}{M}\sum_{m=1}^{M} \parallel F_t^{(m)} - \phi(F_s^{(m)}) \parallel_2^2$$

Explanation: $F^{(m)}$ are intermediate feature tensors, $\phi(\cdot)$ aligns dimensions (e.g., linear projection), M layers distilled.

4.4 Compiler-Level Optimization

Techniques such as operator fusion, graph rewriting, and memory scheduling reduce inference overhead on microcontrollers.

Operator fusion — reduction in memory traffic (conceptual):

$$M_{\text{traffic,fused}} \approx M_{\text{traffic, A}} + M_{\text{traffic, B}} - M_{\text{overlap}}$$

Explanation: fusing ops A and B removes intermediate writes/reads, reducing M_{overlap}.

Latency model (approx.):

$$t_{\text{inference}} \approx \sum_{\ell} \frac{\text{FLOPs}_{\ell}}{\text{Perf}_{\ell}} + \sum_{\ell} t_{\text{mem},\ell}$$

Explanation: per-layer compute time plus memory transfer time; compiler optimizations aim to lower $t_{\text{mem},\ell}$ and operator overhead.

Memory peak after activation scheduling:

$$M_{\text{peak}} = \max_{t} \sum_{\ell \in \mathcal{A}(t)} M_{\text{act},\ell}$$

Explanation: $\mathcal{A}(t)$is set of activations alive at time t; scheduling reduces the maximum by reordering and freeing activations earlier.

Graph rewriting effect on op count (example: replace sequence of ops with single fused op):

$$\#\text{ops}_{\text{rewritten}} = \#\text{ops}_{\text{orig}} - \Delta_{\text{fuse}}$$

Explanation: compiler matches patterns to reduce op count and kernel launch overhead; $\Delta_{\text{fuse}}$is ops removed.

---

5. TinyML Deployment Frameworks

5.1 TensorFlow Lite Micro

Portable interpreter with static memory allocation. Theoretically reliable across many hardware platforms.

5.2 Edge Impulse

Generates optimized C/C++ inference code and integrates data acquisition, training, and deployment tools.

5.3 CMSIS-NN

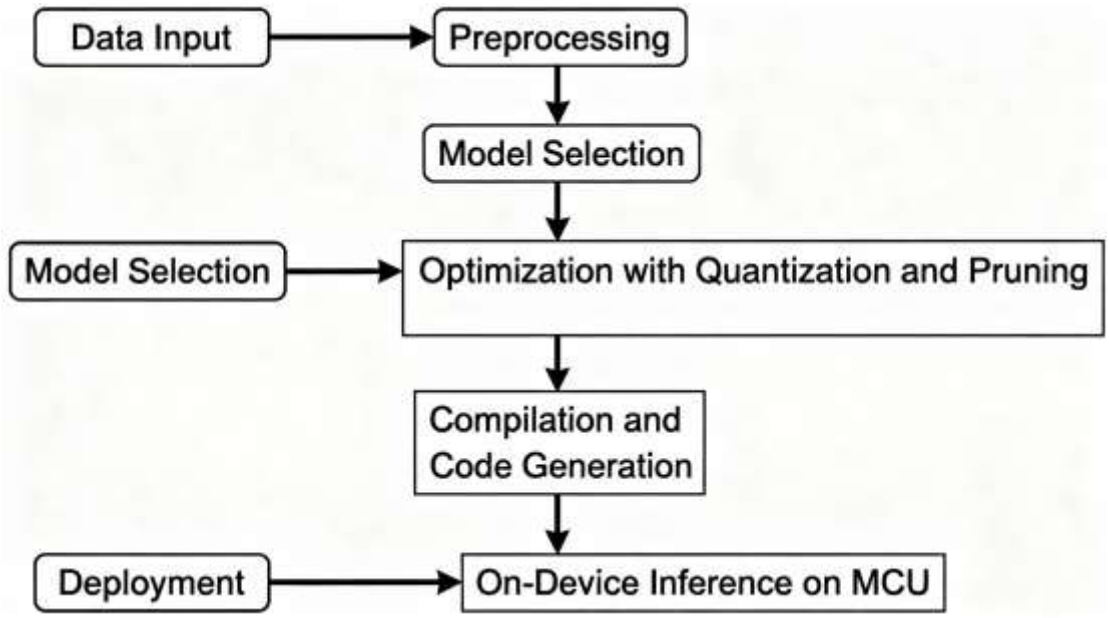
Provides highly optimized kernels for ARM Cortex-M processors, maximizing theoretical efficiency.

5.4 MicroTVM

Compiler-driven ahead-of-time optimization targeting embedded platforms.

5.5 uTensor

Lightweight runtime with modular architecture. Suitable for small-scale TinyML deployments.



6. Theoretical Comparative Table

Framework / Model	Strength	Limitation	Best Use Case
MobileNetV2	Balanced efficiency	Moderate memory need	Vision tasks
SqueezeNet	Very compact	Lower expressive power	Light classification
DS-CNN	Efficient for audio	Narrow domain	Keyword spotting
Tiny-YOLO	Fast detection	Lower accuracy	Basic object detection
EfficientNet-Lite	High accuracy ratio	Larger size	Mid-range MCUs
TFLM	Portable, stable	Interpreter overhead	General purpose
Edge Impulse	Automated workflow	Less low-level control	Rapid deployment
CMSIS-NN	Maximum efficiency	ARM-only	Cortex-M devices
MicroTVM	Compiler optimized	Complex	Performance-critical
uTensor	Lightweight	Limited support	Simple systems

7. Discussion

Each model and framework offers unique theoretical advantages and trade-offs. Lightweight CNN architectures are suitable for classification tasks under constrained memory, while more complex architectures such as MobileNetV2 or EfficientNet-Lite require stronger microcontroller capabilities.

Frameworks differ in execution strategy: interpreter-based systems maximize portability, while compiler-driven approaches maximize theoretical performance. Kernel-optimized libraries such as CMSIS-NN are ideal when operating on supported hardware.

8. Conclusion

This paper provides a purely theoretical comparative analysis of TinyML model architectures and deployment frameworks. By describing conceptual principles without relying on practical experimentation, the work offers a clear perspective on how different TinyML components align with the constraints of embedded systems. Such a theoretical foundation can guide developers and researchers when selecting architectures and frameworks for TinyML applications.

9. Challenges and Future Research Directions

Despite rapid advances, TinyML continues to face several fundamental challenges. The most significant limitation is the strict memory and computational budget of microcontrollers, which restricts model depth and diversity. Quantization and pruning techniques help reduce complexity, but maintaining accuracy under aggressive compression remains a major research problem. Another challenge lies in the lack of hardware standardization; TinyML deployments vary widely across ARM Cortex-M, RISC-V, DSP-enhanced MCUs, and custom edge accelerators, complicating portability and optimization pipelines.

Real-time performance under energy constraints is another active research area. Many IoT devices operate on battery power, making energy-efficient inference essential for long-term deployment. Furthermore, building robust on-device learning capabilities—where models adapt to new users or environments without cloud assistance—remains largely unexplored. Future research is expected to focus on compiler-driven optimization, ultra-lightweight transformer architectures, biologically inspired neural models, and standardized benchmarking platforms (e.g., MLPerf Tiny) to ensure fair comparison across systems. The integration of TinyML with emerging technologies such as IoT, 5G, and edge computing is poised to redefine the computational landscape for smart embedded systems.

10. References

1. Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., & Chen, L. C. (2018). MobileNetV2: Inverted Residuals and Linear Bottlenecks. In IEEE/CVF Conference on Computer Vision and Pattern Recognition (pp. 4510-4520).

2. Iandola, F. N., Han, S., Moskewicz, M. W., Ashraf, K., Dally, W. J., & Keutzer, K. (2016). SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and 0.5MB model size. arXiv preprint arXiv:1602.07360.

3. Tan, M., & Le, Q. V. (2019). EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. In International Conference on Machine Learning (pp. 6105-6114).

4. Redmon, J., & Farhadi, A. (2016). YOLO9000: Better, Faster, Stronger. In IEEE Conference on Computer Vision and Pattern Recognition (pp. 7263-7271).

5. Sainath, T. N., Parada, C., & Parekh, S. (2015). Convolutional Neural Networks for Small-footprint Keyword Spotting. In Sixteenth Annual Conference of the International Speech Communication Association (pp. 1478-1482).

6. Warden, P., & Situnayake, D. (2019). TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers. O'Reilly Media.

7. TensorFlow Lite Team. (2023). TensorFlow Lite: On-Device Machine Learning Inference. Retrieved from https://www.tensorflow.org/lite

8. Han, S., Pool, J., Tran, J., & Dally, W. (2015). Learning both Weights and Connections for Efficient Neural Network. In Proceedings of Advances in Neural Information Processing Systems (pp. 1135-1143).

9. Jacob, B., Kalenichenko, D., Sivakumar, H., et al. (2018). Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. In IEEE/CVF Conference on Computer Vision and Pattern Recognition (pp. 2704-2713).

10. Hinton, G., Vanhoucke, V., & Dean, J. (2015). Distilling the Knowledge in a Neural Network. In Advances in Neural Information Processing Systems (pp. 3581-3589).

11. David, R., Duke, J., Jain, A., Reddi, V. J., Jeffries, N., et al. (2021). *TensorFlow Lite Micro: Embedded Machine Learning on TinyML Systems*. arXiv:2010.08629.

12. Lin, J., Chen, W. M., Lin, Y., Cohn, J., Gan, C., & Han, S. (2020). *MCUNet: Tiny Deep Learning on IoT Devices*. Advances in Neural Information Processing Systems (NeurIPS).

13. Lai, L., Suda, N., & Chandra, V. (2018). *CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs*. arXiv:1801.06601.