Proceedings of the 5th International Conference on Signal Processing and Machine Learning
DOI: 10.54254/2755-2721/133/2025.20598

Optimization Strategies for Low-Power AI Models on Embedded Devices

Tianqi Zhen[1,a,*]

[1]*Washington State University, School of Electrical Engineering & Computer Science, Pullman, WA, 99163, United States*
a. tianqi.zhen@wsu.edu
**corresponding author*

Abstract: With the growing demand for IoT devices, developing low-power AI models on embedded systems has become increasingly important. However, the efficient implementation of AI models within the computational and battery limitations of these devices remains a significant challenge. This study addresses how model pruning and quantization compression techniques can reduce power consumption without significantly compromising model accuracy. The research method optimizes the performance of the three-color recognition model, organizes a dataset consisting of red, yellow, and green classification images, and pre-processes them to standardize the resolution and format. The research object is to use pruning and quantization techniques in combination to optimize memory and computational efficiency further. Experimental evaluation was performed on an Arduino Nano 32 with a camera model, TensorFlow Lite for Microcontrollers for deployment, and a power measurement tool to record energy consumption. The results demonstrate that these methods significantly reduce energy consumption while maintaining acceptable accuracy for real-time applications. This study provides practical optimization strategies for deploying TinyML on resource-constrained devices, offering valuable insights for low-power AI development in IoT and edge computing applications.

Keywords: TinyML, ultra-low-power, model optimization, quantization, pruning

1. Introduction

With the advent of Tiny Machine Learning, deploying optimized ML models on constrained battery-less Internet of Things (IoT) devices with minimal energy availability has become increasingly viable[1]. However, achieving real-time performance for tasks such as image recognition remains a significant challenge due to limited computational power and battery life. Current research in the field has explored various optimization techniques, such as pruning and quantization, to address these limitations. While these methods have demonstrated success in reducing power consumption and model size, challenges remain in balancing efficiency with accuracy, particularly for real-time, low-power applications.

This study focuses on optimizing AI models for ultra-low-power embedded systems by employing pruning and quantization techniques. Specifically, it targets a traffic light color recognition task where an Arduino Nano 32 embedded device, equipped with a camera module, identifies red, yellow, and green colors and triggers feedback upon detecting red. The research workflow begins with organizing

DOI: 10.54254/2755-2721/133/2025.20598

a dataset of red, yellow, and green samples. These samples are preprocessed into a standardized format using OpenCV, resized to 32×32 pixels for subsequent easier use, and normalized to improve model training efficiency. A lightweight convolutional neural network is trained to classify these colors. Pruning is then applied during training using TensorFlow's 'tensorflow_model_optimization' library, which removes less important model weights to reduce computational complexity. Subsequently, full integer quantization is performed, converting the model's parameters to 8-bit integers using TensorFlow Lite, further reducing memory usage and increasing inference speed. The optimized model is deployed and tested on an Arduino Nano 32 embedded device, where metrics such as accuracy, power consumption, and inference time are evaluated using measurement tools to verify the effectiveness of the optimization.

The significance of this research lies in its potential to advance low-power AI deployment in IoT and edge computing. By addressing the balance between energy efficiency and accuracy, this study provides practical strategies for TinyML applications and sets a foundation for future developments in real-time intelligent systems, such as traffic light recognition for smart cars. This analysis addresses the core challenges of TinyML deployment and provides insights that can be extended to other resource-constrained AI applications in IoT and edge computing. Additionally, it also highlights the need for further exploration of hardware-aware optimization to enhance model performance and scalability.

2. About TinyML

2.1. Background of TinyML

As the Internet of Things and edge devices become ubiquitous daily, the need for efficient and low-power AI on embedded devices has become a thorny issue. TinyML brings machine learning capabilities to these constrained environments, enabling real-time inference on tasks. However, unlike large systems, embedded devices face significant challenges due to their unique computational power, memory, and battery limitations, such as huge energy consumption, privacy issues, network and processing latency, and reliability issues [2]. The inability of devices to support complex computations or high energy requirements makes traditional deep-learning models difficult to implement.

2.2. Challenges of TinyML

The primary challenge in TinyML lies in reducing model size and energy consumption without compromising task accuracy. For real-time feedback systems, such as color recognition for real-time feedback systems, maintaining fast processing time and accuracy is critical. In smart environments, the essence of TinyML lies in this balancing act between model complexity and the limited computational resources of tiny devices [3]. Therefore, model optimization techniques are essential to deploy effective AI on embedded platforms. Pruning reduces the computational burden of the model by sparsely connecting the network, while quantization minimizes memory usage by converting model parameters to lower precision. The combination of these techniques makes it possible to run AI models with reasonable accuracy and energy efficiency on devices with minimal hardware resources.

This study applies these techniques to a basic TinyML color recognition task, where the model must recognize red, yellow, and green through a camera module on an Arduino Nano 32 (embedded device) and trigger feedback when red is detected. A balanced approach is shown by implementing a combination of pruning and quantization, which meets low power requirements without significantly affecting the inference quality. This analysis addresses the core challenges of TinyML deployment

DOI: 10.54254/2755-2721/133/2025.20598

and provides insights that can be extended to other resource-constrained AI applications in IoT and edge computing.

3. Data Preparation and Model Training

3.1. Data Collection and Labeling

The accuracy of a machine learning model depends on the quality of its dataset. A high-quality dataset accurately represents real-world phenomena, which is comprehensive and free from biases. The quality of the dataset can have a significant impact on the accuracy and effectiveness of the ML model [4]. The goal of this study is to develop a color recognition model capable of identifying red, yellow, and green. To construct a dataset that supports diverse analysis, each of the three color categories—red, yellow, and green—contains 100 images, sourced from online image repositories.

To ensure data quality, each image was collected under similar lighting during the screening process to minimize the impact of changes in brightness and shadows, thereby improving the stability of the model under different lighting conditions. The images were labeled with its corresponding color: red is assigned as class 0, yellow as class 1, and green as class 2. This structured labeling is critical to correctly distinguish colors and accurately analyze categories when the model is subsequently reasoned on an embedded device.

3.2. Data Preprocessing

Due to the limited memory and computational resources of the Arduino Nano 32, preprocessing is a critical step in optimizing the model for efficient deployment on this embedded platform. The goal of preprocessing is to reduce the image size and standardize the input format, ensuring efficient model performance without sacrificing necessary color information. Each image is resized to a resolution of 32x32 pixels, which strikes a balance between computational efficiency and sufficient color differentiation details. This small but sufficient resolution allows the model to capture and distinguish the color patterns required for red, yellow, and green without wasting too much computational overhead. Adjusting to this input size ensures that each image consume minimal memory, making it manageable on low-power hardware. Using OpenCV, pixel values are divided by 255, and images are resized and normalized to the [0, 1] range, enhancing the model's ability to handle intensity changes and avoiding errors caused by being too sensitive to brightness. The images are converted to RGB format to be compatible with the ‘TensorFlow’ model, and each processed image is stored in a NumPy array as a dataset. The appropriate size for training, validation, and test sets in machine learning model development is a often overlooked but crucial factor [5]. Therefore, to effectively train a compact model for embedded color recognition, the processed dataset is split into training and validation sets with a ratio of 8:2.

4. Optimization

4.1. Pruning Methods for Model

There are two main types of pruning strategies: structured and unstructured, each with different characteristics in terms of implementation, computational complexity, and effectiveness. Structured pruning removes entire structures from the network,which ensures that the pruned network retains a regular structure, making it more efficient to implement on hardware. Since structured pruning retains the regular structure of the network, it aligns better with hardware acceleration and embedded device constraints, resulting in faster inference time and lower power consumption. However, structured

DOI: 10.54254/2755-2721/133/2025.20598

pruning is more aggressive as it removes entire units. This can result in extreme accuracy degradation if not carefully tuned.

On the other hand, unstructured pruning removes individual weights based on their magnitude, creating sparse connections in the network. This approach can achieve more fine-grained compression by targeting the weights that contribute the least to the model output. It allows for finer pruning, resulting in higher compression rates without noticeable accuracy loss. However, t often incurs additional overhead regarding memory access patterns and computational load, especially on resource-constrained devices. In In summary, structured pruning offers faster models with potential accuracy trade-offs, while unstructured pruning maintains higher accuracy at the cost of more complexity and potentially lower efficiency. Structured pruning is easier to implement, as it focuses on larger network blocks, while unstructured pruning is more precise but adds complexity to the optimization process[6].

4.2. Pruning Data and Analysis

Structured pruning was chosen for this study because it is consistent with the hardware capabilities of the Arduino Nano 32. Unstructured pruning is less practical here because the device lacks advanced computational capabilities for sparse matrix operations. Once the model is trained, it can be compressed with a small loss in accuracy. Pruning is a model compression technique that systematically removes unimportant weights from a neural network, thereby reducing the size and computational requirements of the model. It removes the connections below a certain threshold since these provide low or no utility to the output and may even lead to overfitting [7].

For the color classification model, pruning was implemented using TensorFlow's 'tensorflow_model_optimization' library, which allows pruning during training according to a specified pruning schedule.

The polynomial decay schedule starts with an initial sparsity of 20% and gradually increases the sparsity to 80% at the end of training. This gradual approach helps ensure that important weights are retained, minimizing the impact on model accuracy.

Table 1: Pruning data

	Before Pruning	After Pruning
Model Size	160 KB	100 KB
Accuracy	95.1%	87.4%
Inference Speed	61ms	40ms

The 'sparsity.prune_low_magnitude' (pruning wrapper) automatically prunes each layer during training by identifying and setting the weights with the lowest magnitude to 0, which has the least impact on the model. After pruning is complete, the model contains sparse connections with a significant portion of the weights set to 0. To prepare the model for deployment, remove any pruning configuration to strip out pruning-specific metadata and hooks, making it ready for conversion to TensorFlow Lite format. Pruned models are more compact and efficient, which directly benefits embedded deployment. The results are shown in Table 1, pruning reduced the model size by about 37% with little impact on accuracy. The pruned model performed comparable to the original model on a color classification task, with only a slight drop in accuracy of about 8%. The reduction in model size means lower memory usage and faster inference, making it ideal for real-time applications on resource-constrained devices. The reduction in model size and computational requirements after pruning significantly increased the model's suitability for deployment on an Arduino Nano 32, demonstrating the effectiveness of pruning techniques for TinyML applications.

DOI: 10.54254/2755-2721/133/2025.20598

4.3. Combination of Pruning and Quantization

The combination of pruning and quantization achieves maximum optimization of the model and supports low-power applications on embedded devices. Pruning affects model performance by removing neurons with less significance. Quantization areduces data type precision, which affects accuracy. When they are done individually, pruning and quantization result in a considerable loss in model accuracy since the model is incapable of learning from and correcting any quantization errors promptly [8]. The combined method integrates the advantages of the two techniques and optimizes the model size, inference speed, and energy consumption by using their complementary characteristics while ensuring the impact on accuracy. Although both pruning and quantization result in a slight loss of accuracy, the combined model can still achieve an accuracy of 81.6%. Pruning provides a basis for scarification, reduces unnecessary parameters and computation, and creates conditions for further optimization of quantization. Quantization reduces the demand for storage and computing resources by reducing data precision.

4.4. Quantization for Low-Power Inference

Quantization is another model optimization technique that reduces the precision of model parameters. It is a key technology for optimizing machine learning models for resource-constrained devices. Different quantization strategies create trade-offs between model size, computational efficiency, and accuracy. It is important to choose the most appropriate method according to the situation. The two main types of quantization are floating-point quantization and fixed-point quantization. Floating-point quantization reduces the weights to a lower-precision 8-bit or 16-bit floating-point format. This method retains higher precision than fixed-point quantization, making it suitable for complex models where accuracy is the goal. However, compared with integer quantization, its advantages in power and memory efficiency are less significant. While floating-point quantization can maintain higher accuracy due to its wider range and reduced quantization error, fixed-point quantization can still perform well if managed properly [9]. It converts all weights of the model to 8-bit integers, which significantly reduces memory usage and computational costs because integer operations are much faster than floating-point operations and consume less power. The main disadvantage of this method is that it may lead to accuracy loss due to reduced precision in models with a large dynamic range.

4.5. Quantization Data and Analysis

Full integer quantization was chosen because of its alignment with the constraints and capabilities of the target hardware. The quantization process involved the mapping of weights stored in high-precision values to lower-precision data types [10]. This reduces the memory footprint and computational requirements of the model, which is particularly beneficial for the Arduino Nano 32, where efficiency is critical in embedded devices. By converting to a smaller integer-based format, quantization reduces the energy required for inference, making it suitable for real-time, low-power applications. This study applies full integer quantization to the color classification model after pruning. This process quantizes the weights and activations so that all computations during inference are performed using 8-bit integer values instead of more power-hungry 32-bit floating-point values. Full integer quantization is particularly beneficial for TinyML applications because integer operations do not require as much power and memory, allowing the device to run longer on battery.

DOI: 10.54254/2755-2721/133/2025.20598

Table 2: Quantization data

	Before Quantization	After Quantization
Model Size	160KB	63KB
Accuracy	95.1%	81.6%
Inference Speed	100mW	69mW

The quantization process uses a 'TFLiteConverter' with specific quantization settings. The representative dataset helps the model calibrate its dynamic range during the quantization process, which guarantees the model's accuracy. The calibration step adjusts the value range of each layer to fit the 8-bit integer range. This specifies the data type of the model as int8, indicating that both weights and activations will use 8-bit integers. The quantized model is deployed on an Arduino Nano 32 for real-time color recognition. Since integers are less computationally intensive, the quantized model runs faster and consumes less power than the original floating-point model. The results are shown in Table 2, by quantizing both weights and activations to 8-bit integers, power consumption during inference was reduced by about 31%, which is very beneficial for battery-powered devices. The accuracy difference is small, with the quantized model being about 81.6% accurate compared to 95.1% for the original. Although the accuracy loss is not substantial, full integer quantization still resultes in a slight decline in performance for more complex tasks. Additionally, quantization requires the use of a representative dataset to calibrate the dynamic range, which adds more complexity.

5. Discussion

5.1. Finding

The results of this study demonstrate the effectiveness of pruning and quantization in optimizing a simple color classification model for deployment on an Arduino Nano 32. By removing unimportant weights using pruning techniques, the model size was reduced by approximately 37% (from 160 KB to 100 KB) and the inference speed was improved from 61 milliseconds to 40 milliseconds. The structured pruning approach was particularly effective in aligning with the hardware capabilities of the target device, although it resulted in a slight decrease in accuracy of 7.7% (from 95.1% to 87.4%). The increased efficiency and reduced computational requirements make the pruned model more suitable for resource-constrained environments.

Quantization applied after pruning further reduced the model size by 63% (from 160 KB to 63 KB) and reduced power consumption during inference by approximately 31% (from 100 mW to 69 mW). The full integer quantization technique converts all weights and activations to 8-bit integers. This approach maintains reasonable accuracy, achieving 81.6% accuracy for the quantized model compared to 95.1% accuracy for the original model. The trade-off between reduced accuracy and computational efficiency highlights the value of quantization for low-power applications in energy-constrained devices.

5.2. Remaining Challenges

Despite significant improvements achieved by combining pruning and quantization, several challenges remain in optimizing machine learning models for embedded systems. One of the main challenge is the trade-off between model compression and accuracy; structured pruning without careful tuning can lead to significant accuracy degradation, which is a major problem for tasks that require high accuracy. Although quantization reduces power consumption and memory usage, it introduces quantization noise, limiting its effectiveness in complex applications.

DOI: 10.54254/2755-2721/133/2025.20598

Another challenge is the dependence on hardware capabilities, as embedded devices like the Arduino Nano 32 lack advanced hardware support for sparse matrix operations, making unstructured pruning difficult to implement. Furthermore, if the hardware does not natively support integer arithmetic, the quantized model may not achieve its full computational efficiency. These limitations highlight the limitations of developing optimization strategies for specific hardware platforms.

However, when used together, pruning and quantization together enhance the model's suitability for deployment by reducing size, increasing inference speed, and reducing energy consumption while maintaining acceptable accuracy for real-time color recognition tasks. It is clear that pruning and quantization can effectively reduce model size by removing redundant connections or reducing parameter precision. This reduction improved computational efficiency, resulting in speedy inference times and increased throughput, increasing the possibility of deployment of deep neural networks on resource-constrained devices, and facilitating real-time applications [11]. These findings demonstrate the practicality of combining these optimization techniques for TinyML applications. The significance of this study is not limited to the specific task of color recognition. The combined use of pruning and quantization highlights a scalable approach for deploying TinyML applications in IoT scenarios such as environmental monitoring, industrial automation, and wearable devices. However, the observed accuracy trade-offs emphasize the need for task-specific fine-tuning to balance efficiency and performance. This study also highlights the importance of hardware-aware optimizations, as the success of these techniques depends on the capabilities of the target platform. This study demonstrates a practical framework that can advance the role of TinyML in achieving sustainable, low-power AI solutions.

6. Conclusion

This study explores the effectiveness of pruning and quantization model compression techniques in optimizing color classification models for low-power real-time inference on embedded devices. These techniques were investigated to address key challenges in deploying machine learning models on resource-constrained hardware. Pruning successfully reduced the model's size by about 37% with little loss in accuracy, making it more compact and efficient for real-time applications. Quantization further improved energy efficiency by converting 32-bit floating point values to 8-bit integers, which reduced the model's memory footprint and inference time. Despite an 8% decrease in accuracy, quantization significantly reduced power consumption by 31%, demonstrating that model optimization can improve energy efficiency while maintaining adequate performance.

The significance of this study lies in its practical approach to deploying AI models on low-power devices, enabling TinyML applications to run autonomously for long periods of time. The methods in this study offer a solid foundationfor optimizing other models, laying the foundation for the wider application of TinyML in IoT devices such as smart sensors and environmental monitors.

However, this study has limitations. The accuracy of the model is slightly reduced due to quantization, while nor drastic, may affect more complex or more precision-dependent tasks. Additionally, this study focused on a single task, color classification, and performed only two optimization methods, pruning, and quantization, so the results may not translate directly to more complex or computationally demanding models. Future research could explore other methods for model optimization to get the most appropriate choice to maximize the performance of the model. Investigating the impact of these techniques on more complex models and diverse tasks would provide deeper insights into their effectiveness across various applications. By improving and adapting these optimization techniques, more powerful AI applications for embedded systems may be possible in the future.

DOI: 10.54254/2755-2721/133/2025.20598

References

[1] *Sabovic, A. Aernouts, M. Subotic, D. Fontaine, J. DePoorter, E. Famaey, J. (2023). Towards energy-aware tinyML on battery-less IoT devices. Internet Things. 22: 100736. https://doi.org/10.1016/j.iot.2023.100736*

[2] *Partha, P.R. (2022). A review on TinyML: State-of-the-art and prospects. Journal of King Saud University-Computer and Information Sciences. 34: 1595-1623. https://doi.org/10.1016/j.jksuci.2021.11.019*

[3] *n.d. (2024). TinyML: Exploring the world of tiny machine learning. LinkedIn. https://www.linkedin.com/pulse/tinyml-exploring-world-tiny-machine-learning-scdte/#:~:text=The%20essence*

[4] *Youdi, G. Guangzhen, L. Yunzhi, X. Rui, L. Lingzhong, M. (2023). A survey on dataset quality in machine learning. Information and Software Technology. 162: 107268. https://doi.org/10.1016/j.infsof.2023.107268*

[5] *Braka, O. (2023). On Common Split for Training, Validation, and Test Sets in Machine Learning. https://pub.towardsai.net/breaking-the-mold-challenging-the-common-split-for-training-validation-and-test-sets-in-machine-271fd405493d*

[6] *n.d. (2024). Understanding the difference: Structured vs. unstructured neural pruning. Medium. https://medium.com/@agi_63938/understanding-the-difference-structured-vs-unstructured-neural-pruning-76292b9384b7#:~:text=Unstructured%20pruning%20removes%20individual%20weights%20from%20a%20neural, zeros.%20Unstructured%20pruning%20is%20easier%20than%20structured%20pruning.*

[7] *Francisco, C. (2021).TinyML models: What's happening behind the scenes. https://medium.com/marionete/tinyml-models-whats-happening-behind-the-scenes-5e61d1555be9*

[8] *Bofeng, J. Jun, C. Yong, L. (2023). Single-shot pruning and quantization for hardware-friendly neural network acceleration. Engineering Applications of Artificial Intelligence. 126: 106816. https://doi.org/10.1016/j.engappai.2023.106816*

[9] *n.d. (2024). Comparison of Fixed-point Versus Floating-point Quantization in Neural Networks. https://peerdh.com/blogs/programming-insights/comparison-of-fixed-point-versus-floating-point-quantization-in-neural-networks*

[10] *Kartik, T. (2024). A Guide to Quantization in LLMs. Symbl.ai. https://symbl.ai/developers/blog/a-guide-to-quantization-in-llms/.*

[11] *Ummara, B. Mahrukh, M. Dilshad, S. Muhammad, F.U.B. Ali, H. Mustansar, A.G. Arshad, A.K. Wadood, A. (2024). Advances in Pruning and Quantization for Natural Language Processing. IEEE Access. 12: 139113 - 139128. https://ieeexplore.ieee.org/document/10685352*