TinyML-Based Traffic Sign Recognition on MCUs

Aykut Emre Celen
a.e.celen@student.tudelft.nl
Delft University of Technology
The Netherlands

Ran Zhu
r.zhu-1@tudelft.nl
Delft University of Technology
The Netherlands

Qing Wang
qing.wang@tudelft.nl
Delft University of Technology
The Netherlands

Abstract

Real-time traffic sign recognition on microcontrollers (MCUs) introduces challenges due to MCUs' limited memory and processing capacity. In this work, we investigate the trade-offs between model size, classification accuracy, and inference latency within the hardware constraints of MCUs. We design an efficient CNN called **AykoNet** with two variants for traffic sign recognition on MCUs: **AykoNet-Lite**, which prioritizes model size and inference latency, and **AykoNet-Pro**, which prioritizes classification accuracy. We train AykoNet on the German Traffic Sign Recognition Benchmark (GTSRB) and specifically optimize it for deployment on the *Raspberry Pi Pico* platform, which is based on the MCU *RP2040*. AykoNet-Lite delivers 94.6% accuracy with only a 36.8 KB model size and 55.34 ms inference time, while AykoNet-Pro achieves 95.9% accuracy with an 80.18 KB model size and 87.13 ms inference time. Our design shows the effectiveness of our domain-specific preprocessing, class-aware data augmentation, depthwise separable convolutions, and hardware optimizations. The experimental results validate the feasibility of real-time traffic sign recognition in resource-constrained embedded systems. The source codes of this work are available here: https://github.com/aecelen/aykonet

CCS Concepts

• **Computing methodologies** → **Computer vision**; • **Computer systems organization** → **Embedded systems**.

Keywords

TinyML, traffic sign recognition, CNN, data augmentation, microcontrollers.

1 Introduction

Real-time traffic sign recognition enables autonomous navigation systems to interpret and respond to traffic regulations [12, 13]. Integrating such vision-based recognition capabilities into microcontrollers (MCUs) supports the development of low-cost intelligent transportation systems [1, 4]. However, deploying vision systems on MCUs introduces difficulties due to the limited memory and processing capacity of MCUs. The field of Tiny Machine Learning (TinyML) bridges this gap by adapting machine learning models to run efficiently on ultra-low-power MCUs [16]. Achieving real-time performance, on the other hand, requires optimization, as the fundamental challenge lies in balancing classification accuracy, model size, and inference latency within the constraints of hardware.

In this work, we address the following research question: *How can we design an efficient TinyML model for real-time traffic sign recognition on MCUs?* We explore the architectural trade-offs between model size, inference speed, and accuracy. We present **AykoNet**, an efficient network architecture for traffic sign recognition optimized specifically for deployment on the MCU Raspberry Pi Pico. Our approach achieves over 90% accuracy with inference times under 100 ms, demonstrating the feasibility of achieving real-time traffic sign recognition on embedded systems such as MCUs.

The rest of this paper is organized as follows. Section 2 reviews related work in efficient convolutional neural networks and traffic sign recognition on MCUs. Section 3 describes our AykoNet architecture and training methodology. Section 4 details our experimental setup and results. Section 5 discusses the implications and limitations of AykoNet. Section 6 concludes the paper and outlines future research directions.

2 Related Work

Recent literature has shown rising interest in designing lightweight and efficient convolutional neural networks for resource-constrained devices, e.g. [9, 10, 15]. This section presents two key architectures relevant to our work: MobileNets [8], for efficient general-purpose inference, and GiordyNet [6], a traffic sign recognition model specifically optimized for STM32 microcontrollers.

MobileNets, developed by Google in 2017, are a class of efficient convolutional neural networks specifically designed for mobile and embedded applications [8]. MobileNetV1, in particular, has been widely adopted in TinyML applications and is featured as a representative model for person detection on MCUs in [16]. The key innovation of MobileNets is the use of depthwise separable convolutions, which decompose standard convolutions with computational cost $D_K \times D_K \times M \times N \times D_F \times D_F$, where D_K is the kernel size, M is the number of input channels, N is the number of output channels, and D_F is the spatial dimension of the feature map. The decomposition splits this into two separate operations: a depthwise convolution with cost $D_K \times D_K \times M \times D_F \times D_F$ followed by a pointwise convolution with cost $M \times N \times D_F \times D_F$. This factorization reduces computational complexity to $(D_K \times D_K \times M + M \times N) \times D_F \times D_F$ [8]. Furthermore, MobileNets introduce a width multiplier (α) that allows the model to be scaled to match specific hardware constraints.

GiordyNet [6], developed at ETH Zurich in 2020, offers a strong balance between accuracy and memory usage for traffic sign recognition on MCUs. The model is trained on the German Traffic Sign Recognition Benchmark (GTSRB) [7], a widely used dataset containing 43 classes of traffic signs. GTSRB is also employed by other traffic sign recognition systems, such as MASG-Net [5] and Advanced Driver Assistance System [2, 11]. GiordyNet combines domain-specific preprocessing with an efficient architecture tailored for MCUs. It processes grayscale images to reduce the number of input channels and applies photometric distortions to enhance robustness against varying lighting conditions. As a result, GiordyNet achieves a high accuracy of 94.7% on the GTSRB test set and uses significantly less RAM compared to SermaNet, the state-of-the-art neural network model from the GTSRB competition [6].

While MobileNets offer efficient inference through depthwise separable convolutions, they are general-purpose vision models and lack design optimizations tailored for traffic sign recognition. GiordyNet, on the other hand, demonstrates high accuracy on traffic signs due to its domain-specific preprocessing and architectural choices [6]. However, its use of standard convolutions results in computationally heavy inference. This highlights a gap in existing solutions: the need for a model that combines domain-specific optimizations with computational efficiency.

3 AykoNet

In this work, we present AykoNet, a novel architecture that addresses the gap in existing solutions. Our model incorporates GiordyNet's domain-specific preprocessing techniques while integrating MobileNet's efficient depthwise separable convolutions. This approach aims to reduce model size and inference time compared to standard convolutional architectures. Additionally, to maintain high classification accuracy despite these optimizations, we introduce a class-aware data augmentation strategy that handles the class imbalance in the GTSRB dataset.

3.1 Data Preprocessing

Aykonet is trained on the GTSRB dataset, which contains 39,166 training and 12,629 test images across 43 different traffic sign classes. Image sizes range from 15×15 to 250×250 pixels, and are captured under diverse lighting conditions, viewing angles, and weather scenarios, making the dataset a realistic benchmark for traffic sign recognition. We convert all images from RGB to grayscale during preprocessing, following the GiordyNet approach. This channel reduction from three colors to one decreases computational complexity and memory usage while preserving essential structural features like edges and shapes. For traffic signs, which are designed with high contrast and distinctive forms for visibility in varied conditions, the shape information captured in grayscale is often sufficient for accurate classification [6]. All images are then resized to 32×32 pixels using bilinear interpolation. This step standardizes input dimensions and reduces memory usage while preserving sufficient detail for accurate classification. The chosen resolution represents a balance between computational efficiency and representational fidelity, as demonstrated in GiordyNet's findings [6]. Figure 1 illustrates the preprocessing pipeline, demonstrating the transformation of an original high-resolution RGB image of a "Vehicles over 3.5 tons prohibited" traffic sign into its 32×32 grayscale representation, which will be used for our model training and inference.

3.2 Class-Aware Data Augmentation

The GTSRB dataset, consisting of 39,166 images distributed across 43 distinct classes, exhibits a significant class imbalance. In an ideally balanced scenario, each class would contain approximately 910 images. However, the actual distribution ranges from only 210 images to as many as 2250. This results in an imbalance ratio of roughly 1:11 between the least and most represented classes. Specifically, the least represented class is Speed limit (20 km/h) (ID: 0) with 210 images, while the most represented class is Speed limit (50 km/h) (ID: 2) with 2,250 images, deriving an imbalance ratio

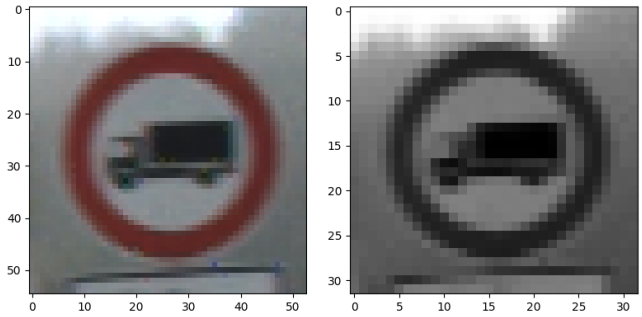

Figure 1: Preprocessing of the images: an example for the traffic sign "Vehicles over 3.5 tons prohibited".

of 10.71x. Such disparity can lead to biased models that perform poorly on underrepresented classes. This is particularly problematic in safety-critical applications, such as autonomous driving, where misclassifying rare traffic signs could result in hazardous outcomes. Ensuring balanced performance across all classes is therefore essential for building robust and reliable traffic sign recognition systems.

3.2.1 Augmentation Methods. We develop a data augmentation pipeline that employs the following four methods to simulate real-world variations:

- **Rotation:** Randomly rotates images within a range of $[-15°, 15°]$ to account for camera tilts and vehicle angles.
- **Translation:** Shifts images horizontally and vertically by up to 5 pixels to simulate sign displacement within the frame.
- **Shearing:** Applies a horizontal shear transformation using a random factor from $[-0.2, 0.2]$ to simulate perspective distortions caused by angled views.
- **Gamma Correction:** Adjusts image brightness and contrast using a gamma value randomly sampled from the range $[0.4, 1.5]$, simulating varying lighting conditions.

Figure 2 illustrates the effects of our data augmentation methods applied to a preprocessed image of the "Vehicles over 3.5 tons prohibited" traffic sign. The original image, resized to 32×32 pixels and converted to grayscale (as shown in Figure 1), is used as the input for these augmentations to demonstrate their impact.

3.2.2 Augmentation Strategy. We develop a tiered data augmentation strategy that applies augmentation proportionally to each class's sample count:

- Classes with $\leq$*300* images: we apply *three augmentation methods per image*, quadrupling the class size.
- Classes with *301–450* images: we apply *two augmentation methods per image*, tripling the class size.
- Classes with *451–780* images: we apply *one augmentation methods per image*, doubling the class size.
- Classes with *781–1500* images: we augment *30% of the images with one method*, increasing the class size.
- Classes with $>$*1500* images: *no augmentation* is applied, as these classes are already well represented in the dataset.

3.2.3 Results. Our class-aware data augmentation increases the total dataset size from 39,209 to 61,726, representing a 57.4% increase. More importantly, it reduces the class imbalance ratio from

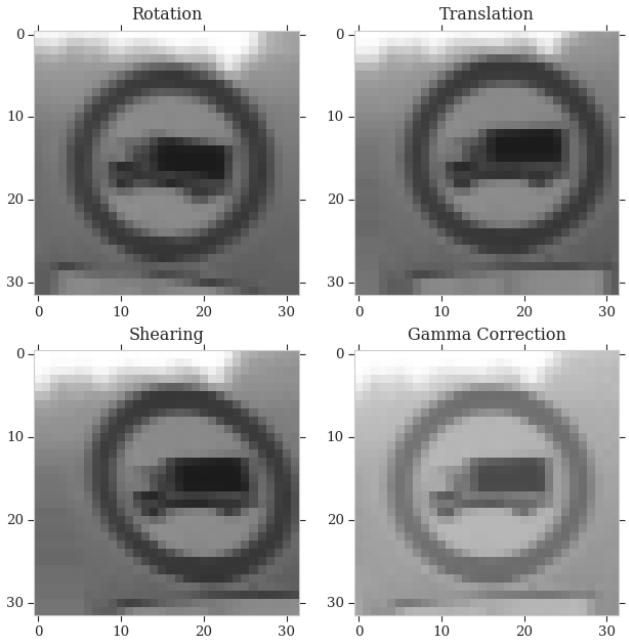

Figure 2: The used data augmentation methods.

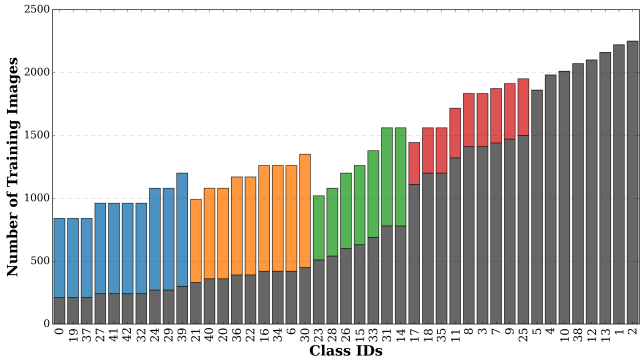

Figure 3: Class distribution after augmentation.

10.7× to 2.7×, corresponding to an approximately 75% reduction in disparity between the largest and smallest classes. The most underrepresented classes grow from 210 samples to 840 samples, ensuring adequate representation during training. Figure 3 illustrates the original class distribution and augmentation targets for each class. The blue column indicates +300%, the orange column +200%, the green column +100%, and the red column +30%.

3.3 Architecture

To explore the architectural trade-offs, we develop two variants of AykoNet, each optimized for different objectives:

- **AykoNet-Lite:** Prioritizes minimal model size and fast inference for real-time applications.
- **AykoNet-Pro:** Prioritizes classification accuracy while maintaining deployability on MCUs.

3.3.1 AykoNet-Lite. We design AykoNet-Lite with aggressive size and speed optimizations for real-time MCU deployment. Table 1 shows AykoNet-Lite's complete architecture.

The designed model follows a power-of-two channel progression of 8→16→32→64→128. This scaling strategy optimizes memory access patterns on the Raspberry Pi Pico's ARM Cortex-M0+ architecture, enabling efficient memory alignment for 32-bit memory bus access, simplified address calculation during convolution operations, and improved vectorization efficiency in TensorFlow Lite Micro kernels [3]. This contrasts with GiordyNet's irregular channel progression of 10→50→100, which may result in less efficient memory utilization. Except for the initial feature extraction layer, all convolutional layers employ depthwise separable convolutions to reduce computational cost and parameter count. This approach differs from GiordyNet, which does not incorporate any depthwise operations, and from MobileNet, which uses depthwise separable convolutions even in its initial feature extraction. The classifier consists only of GlobalAveragePooling2D followed by dropout and a single dense output layer. This minimalist design avoids the computational overhead of intermediate dense layers while preserving sufficient representational capacity for traffic sign classification.

3.3.2 AykoNet-Pro. While AykoNet-Lite is designed for minimal resource usage, AykoNet-Pro prioritizes higher classification accuracy within feasible deployment constraints.

The model uses 16→32→64→128→128 channels, starting with a higher initial capacity compared to AykoNet-Lite (16 vs 8) and maintaining maximum channels in the final block. This is designed to provide richer feature representations throughout the network. In contrast to AykoNet-Lite, the classifier includes an intermediate Dense(128) layer with ReLU activation, providing additional representational capacity for complex feature combinations. This architectural choice trades a modest increase in size for improved classification performance.

4 Experimental Setup and Results

To evaluate AykoNet's performance and validate our design choices, we benchmark it against three baseline models: MobileNetV1_25 (width multiplier of α=0.25), MobileNetV1_20 (α=0.20), and GiordyNet. We choose them to highlight the trade-offs between the general-purpose efficiency of MobileNets and the domain-specific optimization of GiordyNet. We evaluate all the models in terms of the model size, classification accuracy, and inference latency.

Our experimental pipeline follows the standard TinyML workflow. The process begins with model training on Google Colab, followed by conversion to the TensorFlow Lite format with full integer quantization. The final models are then deployed on our target platform, Raspberry Pi Pico, which is based on the RP2040 MCU. It features a dual-core ARM Cortex-M0+ processor running at 133 MHz, with 264 KB of SRAM and 2 MB of flash memory.

4.1 Evaluation Metrics

4.1.1 Model Size. We measure the post-quantization model size in kilobytes (KB), representing the final memory footprint required for deployment. This metric directly impacts the feasibility of deployment on memory-constrained microcontrollers. Based on the guidelines of TensorFlow Lite for MCU, we assume that models exceeding

Table 1: AykoNet-Lite's body architecture.

Layer Type	Output Shape	Parameters
Initial Feature Extraction		
Conv2D	(32, 32, 8)	80
BatchNormalization	(32, 32, 8)	32
ReLU	(32, 32, 8)	0
First Depthwise Separable Block		
DepthwiseConv2D	(16, 16, 8)	80
BatchNormalization	(16, 16, 8)	32
ReLU	(16, 16, 8)	0
Conv2D	(16, 16, 16)	144
BatchNormalization	(16, 16, 16)	64
ReLU	(16, 16, 16)	0
Second Depthwise Separable Block		
DepthwiseConv2D	(8, 8, 16)	160
BatchNormalization	(8, 8, 16)	64
ReLU	(8, 8, 16)	0
Conv2D	(8, 8, 32)	544
BatchNormalization	(8, 8, 32)	128
ReLU	(8, 8, 32)	0
Third Depthwise Separable Block		
DepthwiseConv2D	(4, 4, 32)	320
BatchNormalization	(4, 4, 32)	128
ReLU	(4, 4, 32)	0
Conv2D	(4, 4, 64)	2,112
BatchNormalization	(4, 4, 64)	256
ReLU	(4, 4, 64)	0
Fourth Depthwise Separable Block		
DepthwiseConv2D	(4, 4, 64)	640
BatchNormalization	(4, 4, 64)	256
ReLU	(4, 4, 64)	0
Conv2D	(4, 4, 128)	8,320
BatchNormalization	(4, 4, 128)	512
ReLU	(4, 4, 128)	0
Classification Head		
GlobalAveragePooling2D	(128)	0
Dropout	(128)	0
Dense (Softmax)	(43)	5,547
Total parameters:		**19,419**
Trainable parameters:		**18,683**
Non-trainable parameters:		**736**

250KB would not fit within the Pico's deployment constraints, as this threshold ensures sufficient memory remains available for the system components [14].

4.1.2 Classification Accuracy. We evaluate classification accuracy using quantized TensorFlow Lite models with 8-bit integer precision, as these models are intended for MCU deployment. We evaluate 1,000 randomly selected images from the official GTSRB test set (which contains 12,629 images).

4.1.3 Inference Latency. We evaluate the real-time performance by measuring the inference time of quantized models on the MCU

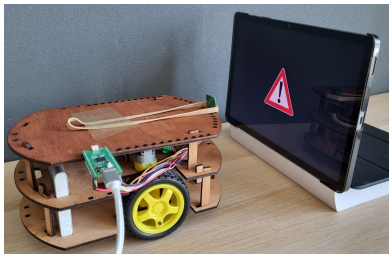

Figure 4: Experimental setup: the used Raspberry Pi Pico Zero MCU and the HM01B0 camera.

RP2040 with an HM01B0 camera. Experiments are conducted in a static setup, where both the camera and the traffic signs remain stationary. The camera operates at an exposure rate of 300, and the distance between the camera and the signs is fixed at 15 cm, as shown in Figure 4. To ensure consistency, we evaluate all the deployable models using a standardized protocol with 43 test images from the GTSRB dataset (one per traffic sign class). We record inference times using Pico's timers (`get_absolute_time()`), specifically measuring the duration of the TensorFlow Lite Micro interpreter's `Invoke()` call. This isolates neural network computation time from the overhead of preprocessing and post-processing. We sum the inference times over all 43 images and average them to derive representative latency metrics.

4.2 Results

We evaluate the model's performance based on the three key metrics listed above: classification accuracy, model size, and inference latency. The results are summarized in Tables 2-4.

4.2.1 Model Size. We measure model sizes using the file size of the TensorFlow Lite models as described in Section 4.1.1. We only report quantized versions, as they represent the deployable models. Table 2 lists the results.

We can observe that, despite using depthwise separable convolutions for efficiency, MobileNetV1 models have the largest sizes due to their general-purpose architecture, which is designed for diverse image classification tasks. These models process RGB inputs (3 channels) and employ 13 convolutional blocks, resulting in a higher parameter count compared to domain-specific architectures.

MobileNetV1_25 exceeds 300 KB. Although it surpasses the 250 KB threshold recommended in the TFLite Micro repository [14], it may be deployable since TensorFlow Lite Micro's tensor arena requirement (`kTensorArenaSize`) is typically smaller than the full model size, as not all model weights need to be cached simultaneously during inference. However, following the deployment guidelines from the TFLite Micro repository [14], we consider models

Table 2: Model size.

Model	Size (KB)
MobileNetV1_25-int8	307.59
MobileNetV1_20-int8	217.79
GiordyNet-int8	106.87
AykoNet-Lite-int8	36.80
AykoNet-Pro-int8	80.18

Table 3: Classification accuracy.

Model	Accuracy
MobileNetV1_25-int8	87.50%
MobileNetV1_20-int8	79.80%
GiordyNet-int8	95.50%
AykoNet-Lite-int8	94.60%
AykoNet-Pro-int8	95.90%

exceeding 250 KB as incompatible with Pico's memory constraints. Reducing the width multiplier α from 0.25 to 0.20 yields a 29.2% size reduction (from 307.59 KB to 217.79 KB), highlighting the effectiveness of width multipliers for compressing and scaling models.

GiordyNet (106.87 KB) and AykoNet-Pro (80.18 KB) achieve similar model sizes, indicating similar architectural complexity. AykoNet-Lite (36.8 KB) represents an aggressive minimization approach, achieving a 54% reduction compared to AykoNet-Pro.

4.2.2 Classification Accuracy. We evaluate the quantized TensorFlow Lite models on the test set, as described in Section 4.1.2. The results are presented in Table 3.

MobileNetV1_25 achieves an accuracy of 87.50% despite its general-purpose design, demonstrating adaptability to domain-specific classification tasks. Reducing the width multiplier α from 0.25 to 0.20 results in an 8.8% drop in accuracy (from 87.50% to 79.80%) in MobileNetV1_20. Compared to the corresponding 29.2% reduction in model size (from 307.59 KB to 217.79 KB), the accuracy loss is disproportionately smaller, suggesting that the width multiplier α is an effective mechanism for balancing model efficiency and performance. However, since MobileNetV1_20 achieves less than 80% accuracy, it is considered unsuitable for traffic sign recognition in safety-critical autonomous systems.

AykoNet-Pro achieves the highest accuracy at 95.90%, closely matching the performance of GiordyNet (95.50%). Both models benefit from domain-specific architectures and preprocessing techniques. AykoNet-Lite demonstrates exceptional efficiency, achieving 94.60% accuracy while requiring only 36.8 KB of storage. Compared to GiordyNet, AykoNet-Lite is approximately one-third the size while sacrificing only 0.9 percentage points in accuracy, validating the effectiveness of our architectural optimizations.

In addition to comparisons with other models, the confusion matrices of AykoNet-Lite (Figure 5) and AykoNet-Pro (Figure 6) illustrate their performance across 43 traffic sign classes. Darker blue squares along the diagonal represent correct predictions, while lighter off-diagonal squares indicate misclassifications. Both models show difficulty with classes 19, 21, 27, 30, 37, and 40, suggesting that these specific traffic sign types are inherently more challenging to classify. This may also be due to the lower number of training samples for these classes, as shown by the dataset imbalance in Figure 3. In contrast, classes such as 13, 16, 32, 33, 35, 39, and 42 consistently exhibit dark diagonal squares in both matrices, indicating they are reliably well-classified regardless of the model architecture.

4.2.3 Inference Latency. We measure the inference time of the models on the Raspberry Pi Pico MCU as described in Section 4.1.3. We exclude MobileNetV1_25-int8 as its size exceeds the 250 KB threshold (see Section 4.1.1). Table 4 shows the latency results.

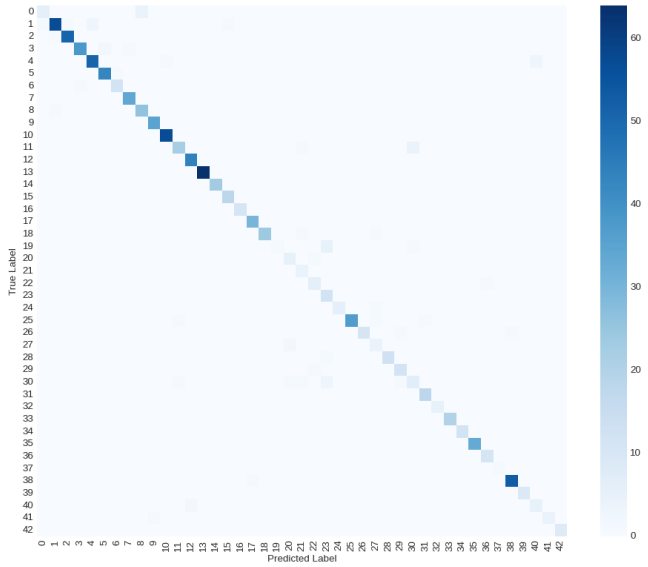

Figure 5: Confusion matrix of AykoNet-Lite *(Colors indicate the prediction count for each traffic sign type).*

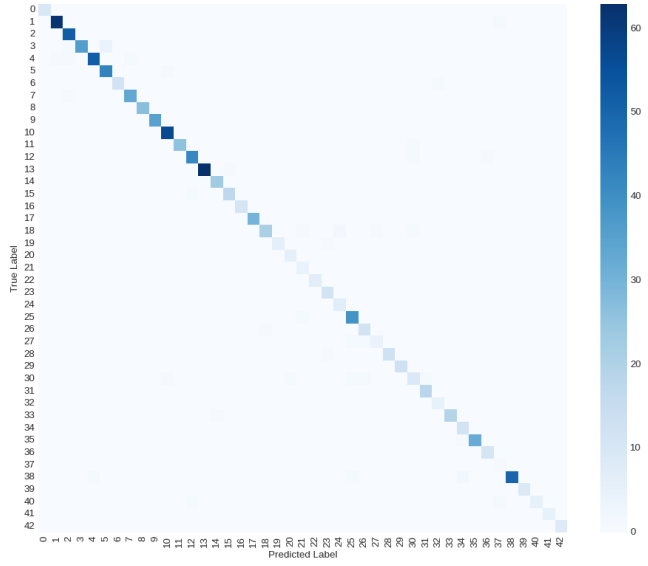

Figure 6: Confusion matrix of AykoNet-Pro *(Colors indicate the prediction count for each traffic sign type).*

GiordyNet exhibits the highest inference latency (204.08 ms), primarily attributed to its exclusive use of standard convolution layers instead of depthwise separable convolutions. Furthermore, its irregular channel progression of 10⟶50⟶100 likely leads to misaligned memory access patterns on the 32-bit ARM Cortex-M0+ processor. The combination of computationally intensive standard convolutions and suboptimal memory access patterns contributes to its substantial latency.

MobileNetV1_20 achieves efficient inference (77.29 ms) despite processing RGB images (3 channels) and having a model size of

Table 4: Inference latency.

Model	Time (ms)
MobileNetV1_20-int8	77.29
GiordyNet-int8	204.08
AykoNet-Lite-int8	55.34
AykoNet-Pro-int8	87.13

217.79 KB. Its exclusive use of depthwise separable convolutions throughout the network, combined with optimizations specifically designed for TensorFlow Lite Micro's interpreter [3, 8], demonstrates the effectiveness of hardware-aware architectural design for embedded deployment on resource-constrained microcontrollers.

AykoNet-Pro records an inference time of 87.13 ms, demonstrating a 12.7% increase compared to MobileNetV1_20, despite processing grayscale images. This performance overhead is attributed to the inclusion of standard convolution layers in the initial feature extraction, which introduces greater computational complexity than a purely depthwise separable approach. AykoNet-Lite delivers the fastest inference time (55.34 ms), validating its lightweight design. Although it also employs standard convolutions in the initial feature extraction, its minimalist classification head and reduced initial channel capacity compared to AykoNet-Pro (8 vs 16) significantly decrease the total computational operations, resulting in optimal inference performance for the target hardware platform.

5 Discussions

By developing and evaluating AykoNet, we identified key principles for effective embedded machine learning deployment, exploring the architectural trade-offs that influence model size, classification accuracy, and inference latency. Our results reveal distinct performance rankings across these metrics: AykoNet-Pro achieves the highest accuracy (95.90%), while AykoNet-Lite delivers the smallest model size (36.8 KB) and the fastest inference time (55.34 ms). Critically, no single model dominates all metrics, highlighting the fundamental trade-offs in TinyML and real-world deployment.

The evaluation of AykoNet variants against MobileNetV1_20 underscores the advantages of domain-specific preprocessing and architecture design in achieving high classification accuracy. While MobileNetV1_20 offers a deployable model size and applicable inference latency for real-time systems, its sub-80% accuracy renders it unsuitable for safety-critical applications. AykoNet addresses this gap by employing a targeted domain-specific approach, achieving the high accuracy required for robust autonomous systems.

The comparative analysis between AykoNet variants and GiordyNet highlights the distinct advantages of depthwise separable convolution layers and class-aware data augmentation. Although GiordyNet achieves strong classification accuracy, its reliance on computationally intensive standard convolutions results in problematic inference latency for real-time applications. AykoNet overcomes this limitation by integrating efficient depthwise separable convolutions. To maintain high classification accuracy despite this optimization, we apply class-aware data augmentation on the dataset, which led AykoNet-Pro to achieve higher accuracy than GiordyNet while being smaller and faster.

Our experimental design has several limitations. To address class imbalance, we implemented a data augmentation pipeline without exploring undersampling strategies, which could have provided useful comparative insights. Additionally, the specific contributions of individual design decisions—such as grayscale conversion and data augmentation—are not quantified, as we did not evaluate AykoNet variants without these components. Lastly, both the camera and traffic signs remain stationary during the experiments. This setup does not fully reflect real-world conditions, where the system must operate on moving vehicles. The primary reason for this limitation is the HM01B0 camera used in our setup, which has a very low resolution of 160×120 pixels and poor image quality during motion. When the camera moves, the resulting images become significantly blurred due to its limited dynamic range and slow shutter response, making it unsuitable for capturing clear frames in dynamic scenarios.

6 Conclusion

In this work, we have studied how to design an effective TinyML model for real-time traffic sign recognition on MCUs. We proposed a new network architecture called AykoNet with two variants: one prioritizing classification accuracy (AykoNet-Pro) and one prioritizing model size and inference latency (AykoNet-Lite). We investigated several key design decisions that led to an effective model. We compared AykoNet with existing architectures. The results demonstrated AykoNet's effectiveness, highlighting the impact of domain-specific preprocessing, class-aware data augmentation, depthwise separable convolutions, and hardware optimizations. For future work, we plan to integrate AykoNet into an autonomous navigation system with camera integration, enabling real-time recognition of traffic signs and autonomous vehicle response.

References

[1] Michael et al. Breiter. 2016. Video-Based ITS – State of the Art. *Transp Res Procedia* (2016).
[2] Tejas Chaudhari, Ashish Wale, Amit Joshi, and Suraj Sawant. 2020. *Traffic Sign Recognition Using Small-Scale CNN.*
[3] Robert et al. David. 2020. *TensorFlow Lite Micro: Embedded ML on TinyML Systems.*
[4] Esma Dilek and Murat Dener. 2023. Computer Vision in ITS: A Survey. *Sensors* (2023).
[5] Chunhui Du, Shenglan Su, and Cheng et al. Lin. 2025. A lightweight network for traffic sign detection via multiple scale context awareness. *Sci Rep* (2025).
[6] Marco Giordano. 2020. *Traffic Sign Recognition, CNN on Microcontrollers.*
[7] Sebastian Houben, Johannes Stallkamp, Jan Salmen, Marc Schlipsing, and Christian Igel. 2013. Detection of Traffic Signs: GTSRB Benchmark. In *IJCNN.*
[8] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *arXiv:1704.04861* (2017).
[9] Forrest N. et al. Iandola. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer params.
[10] Jonghoon Jin, Aysegul Dundar, and Eugenio Culurciello. 2015. Flattened CNNs for Feedforward Acceleration.
[11] Ida Syafiza Binti Md Isa, Choy Ja Yeong, and Nur Latif Azyze. 2022. *Real-time traffic sign detection on Raspberry Pi.*
[12] Girish Kumar et al. N G. 2025. Real-time traffic sign recognition for autonomous vehicles. *Multimedia Tools Appl* (2025).
[13] A. et al. Radha Rani. 2024. Traffic sign detection with haze removal. *e-Prime* (2024).
[14] Yuan Tang. [n. d.]. *TensorFlow Lite for Microcontrollers.*
[15] Min Wang, Baoyuan Liu, and Hassan Foroosh. 2017. Design of Efficient Conv Layers with Intra-channel Conv.
[16] Pete Warden and Daniel Situnayake. 2019. *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers.*