Tiny Machine Learning for Embedded Systems

Real-Time Traffic Sign Recognition on Microcontrollers

Aykut Emre Celen[1]

Supervisor(s): Qing Wang[1]**, Ran Zhu**[1]

[1]**EEMCS, Delft University of Technology, The Netherlands**

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 22, 2025

Name of the student: Aykut Emre Celen
Final project course: CSE3000 Research Project
Thesis committee: Qing Wang, Ran Zhu, Ranga Rao Venkatesha Prasad

Abstract

Real-time traffic sign recognition on microcontrollers introduces challenges due to limited memory and processing capacity. This study investigates the trade-offs between model size, classification accuracy, and inference latency within hardware constraints. We present an efficient network architecture called AykoNet with two variants: AykoNet-Lite, prioritizing model size and inference latency, and AykoNet-Pro, prioritizing classification accuracy. We trained AykoNet on the German Traffic Sign Recognition Benchmark (GTSRB) and specifically optimized it for deployment on the Raspberry Pi Pico microcontroller. AykoNet-Lite delivers 94.60% accuracy with only a 36.80KB model size and 55.34ms inference time, while AykoNet-Pro achieves 95.90% accuracy with an 80.18KB model size and 87.13ms inference time. Our approach demonstrates the effectiveness of domain-specific preprocessing and architectural design, class-aware data augmentation, and the strategic use of depthwise separable convolutions. These results validate the feasibility of real-time traffic sign recognition in resource-constrained embedded systems. Specifically, AykoNet-Lite strikes an optimal balance between model size, classification accuracy, and inference latency for practical deployment in autonomous navigation applications.

1 Introduction

Real-time traffic sign recognition enables autonomous navigation systems to interpret and respond to traffic regulations [13; 14]. Integrating such vision-based recognition capabilities into microcontrollers supports the development of low-cost intelligent transportation systems [4; 1].

However, deploying vision systems on microcontrollers introduces difficulties due to limited memory and processing capacity. The field of Tiny Machine Learning (TinyML) bridges this gap by adapting machine learning models to run efficiently on ultra-low-power microcontrollers [19]. 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 hardware constraints.

This research addresses the question: How can we create an optimal TinyML model for real-time traffic sign recognition on microcontrollers? We explore the architectural trade-offs between model size, inference speed, and accuracy. This paper presents AykoNet, an efficient network architecture for traffic sign recognition specifically optimized for deployment on the Raspberry Pi Pico microcontroller. Our approach achieves over 90% accuracy with inference times under 0.1 seconds, demonstrating the feasibility of real-time traffic sign recognition on embedded systems.

Section 2 reviews related work in efficient convolutional neural networks and traffic sign recognition on microcontrollers. Section 3 describes the AykoNet architecture and training methodology. Section 4 details the experimental setup and results. Section 5 discusses the implications and limitations of our approach. Section 6 reflects the ethical aspects of this study. Section 7 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. [10; 12; 16]. This section presents two key architectures relevant to our work: MobileNets [9], 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 [9]. MobileNetV1, in particular, has been widely adopted in TinyML applications and is featured in the TinyML book as a representative model for person detection on microcontrollers [19]. The key innovation of MobileNets is the use of depthwise separable convolutions, which decompose standard convolutions with computational cost

$$D_K \times D_K \times M \times N \times D_F \times D_F \tag{1}$$

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 [9]

$$(D_K \times D_K \times M + M \times N) \times D_F \times D_F \tag{2}$$

Furthermore, MobileNets introduce a width multiplier (α) that allows the model to be scaled to match specific hardware constraints.

GiordyNet, developed by Marco Giordano at ETH Zurich in 2020, offers a strong balance between accuracy and memory usage for traffic sign recognition on microcontrollers [6]. The model is trained on the German Traffic Sign Recognition Benchmark (GTSRB), a widely used dataset containing 43 classes of traffic signs [8]. Sample images from different classes of the GTSRB dataset are shown in Figure 1. 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 microcontrollers. 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.

Figure 1: Sample images from the GTSRB dataset

3 AykoNet

In this work, we present AykoNet, a novel architecture that addresses the gap in existing solutions by integrating elements from both MobileNets and GiordyNet. Our model incorporates GiordyNet's domain-specific preprocessing techniques and architectural principles 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.

This section outlines the complete AykoNet training pipeline, detailing the dataset preparation methodology, augmentation strategy, architectural design, and training procedures.

3.1 Data and Preprocessing

Aykonet is trained on the GTSRB dataset, which contains 39,166 training and 12,629 test images across 43 different traffic sign classes (see Appendix A for the class labels). 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 2 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 used for model training and inference.

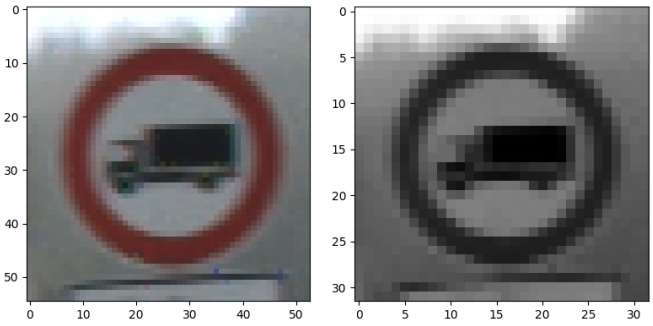

Figure 2: Preprocessing of an image

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 of 10.71x. Figure 3 illustrates this imbalance across all 43 traffic sign categories, sorted by frequency.

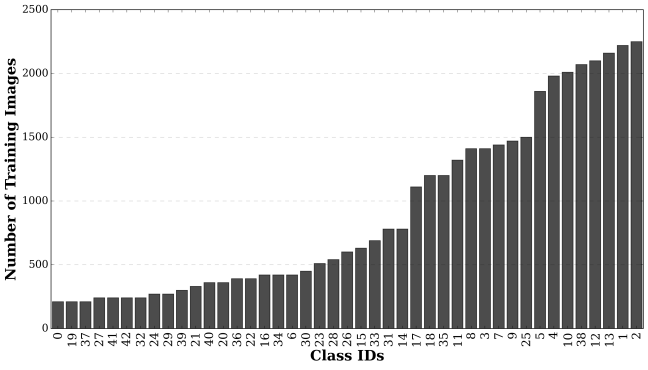

Figure 3: Class distribution in the GTSRB training dataset

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.

Augmentation Techniques

To address this class imbalance, we developed a data augmentation pipeline that employs these four techniques 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 4 illustrates the effects of our data augmentation techniques 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 2), is used as the input for these augmentations to demonstrate their impact.

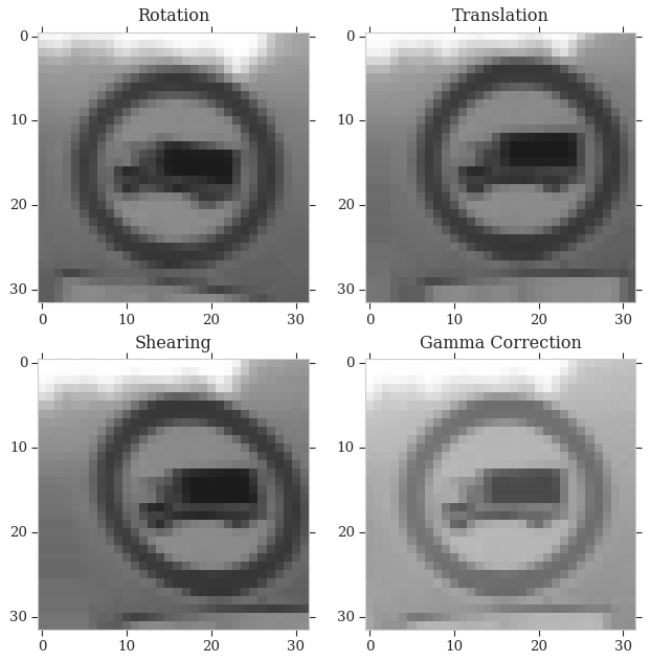

Figure 4: Effects of data augmentation techniques applied to a preprocessed image

Augmentation Strategy

We developed a tiered data augmentation strategy that applies augmentation proportionally to each class's sample count:

- Classes with $\leq$**300** images: we apply **three augmentation techniques per image**, quadrupling the class size (+300%).
- Classes with **301–450** images: we apply **two augmentation techniques per image**, tripling the class size (+200%).
- Classes with **451–780** images: we apply **one augmentation technique per image**, doubling the class size (+100%).
- Classes with **781–1500** images: we augment **30% of the images with one technique**, increasing the class size by 30%.
- Classes with $>$**1500** images: **no augmentation** is applied, as these classes are already well represented in the dataset.

Results

Our class-aware data augmentation substantially improved the balance of the GTSRB dataset (see Appendix B for the full list of augmented classes):

- 10 classes were augmented with 3 techniques per image (+300%)
- 9 classes were augmented with 2 techniques per image (+200%)
- 7 classes were augmented with 1 technique per image (+100%)
- 9 classes were augmented with 1 technique applied to 30% of images (+30%)
- 8 classes with sufficient samples were not augmented

Figure 5 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%.

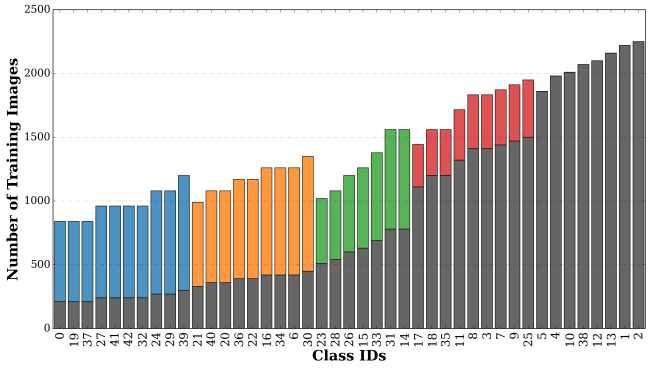

Figure 5: Class distribution in the GTSRB training dataset after data augmentation

This approach increased the total dataset size from 39,209 to 61,726 images, representing a 57.4% increase. More importantly, it reduced the class imbalance ratio from 10.7× to

2.7×, corresponding to an approximately 75% reduction in disparity between the largest and smallest classes. The most underrepresented classes grew from 210 samples to 840 samples, ensuring adequate representation during training.

3.3 Architecture

To explore the architectural trade-offs, we developed 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 microcontrollers

AykoNet-Lite

We designed AykoNet-Lite with aggressive size and speed optimizations for real-time microcontroller deployment. Table 1 shows the complete architecture.

The 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.

AykoNet-Pro

While AykoNet-Lite is designed for minimal resource usage, AykoNet-Pro prioritizes higher classification accuracy within feasible deployment constraints. Table 2 shows the complete architecture.

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.

3.4 Training

AykoNet was trained using the Adam optimizer with a learning rate of 0.001 and a sparse categorical cross-entropy loss function. The training process employed a batch size of 32 and was configured to run for a maximum of 50 epochs with a 20% validation split from the training data.

To prevent overfitting, early stopping was implemented with patience of 15 epochs, monitoring validation accuracy. This callback automatically restored the model weights to the best-performing epoch when validation accuracy stopped improving.

Table 1: AykoNet-Lite 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**

3.5 Post-training Quantization

The TinyML workflow as described in the TinyML book was followed [19]. After training, the model underwent post-training quantization to reduce its size and meet the computational requirements of 8-bit microcontrollers.

The trained Keras model was first converted to TensorFlow Lite format with float32 precision, then integer-only quanti-

Table 2: AykoNet-Pro Body Architecture

Layer Type	Output Shape	Parameters
Initial Feature Extraction		
Conv2D	(16, 16, 16)	160
BatchNormalization	(16, 16, 16)	64
ReLU	(16, 16, 16)	0
First Depthwise Separable Block		
DepthwiseConv2D	(16, 16, 16)	160
BatchNormalization	(16, 16, 16)	64
ReLU	(16, 16, 16)	0
Conv2D	(16, 16, 32)	544
BatchNormalization	(16, 16, 32)	128
ReLU	(16, 16, 32)	0
Second Depthwise Separable Block		
DepthwiseConv2D	(8, 8, 32)	320
BatchNormalization	(8, 8, 32)	128
ReLU	(8, 8, 32)	0
Conv2D	(8, 8, 64)	2,112
BatchNormalization	(8, 8, 64)	256
ReLU	(8, 8, 64)	0
Third 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
Fourth Depthwise Separable Block		
DepthwiseConv2D	(4, 4, 128)	1,280
BatchNormalization	(4, 4, 128)	512
ReLU	(4, 4, 128)	0
Conv2D	(4, 4, 128)	16,512
BatchNormalization	(4, 4, 128)	512
ReLU	(4, 4, 128)	0
Classification Head		
GlobalAveragePooling2D	(128)	0
Dense	(128)	16,512
ReLU	(128)	0
Dropout	(128)	0
Dense (Softmax)	(43)	5,547
Total parameters:		**54,539**
Trainable parameters:		**53,323**
Non-trainable parameters:		**1,216**

zation was applied using 500 representative samples from the training set [7]. This process converts all weights and activations from 32-bit floating-point to 8-bit integers.

4 Experimental Setup and Results

To evaluate AykoNet's performance and validate our design choices, we benchmarked it against three baseline models. We conducted a comparative experiment against these models: MobileNetV1_25 (width multiplier of α=0.25), MobileNetV1_20 (width multiplier of α=0.20), and GiordyNet. We chose these baselines to highlight the trade-offs between the general-purpose efficiency of MobileNets and the domain-specific optimization of GiordyNet. This comparison positions AykoNet's approach within the existing literature. To ensure a fair analysis, all models underwent identical training procedures and post-training quantization.

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 also on Google Colab. The final models were then deployed on our target hardware platform, a Raspberry Pi Pico microcontroller, for real-world performance testing.

We evaluated all models against these three metrics:

- Model Size
- Classification Accuracy
- Inference Latency

This methodology provides a comprehensive assessment of the trade-offs between model complexity, accuracy, and performance that are critical in TinyML applications. The following subsections will detail the training and deployment of the models, explain the evaluation metrics, and present the final results.

4.1 MobileNets

MobileNetV1 was selected as a baseline because it serves as the canonical model in the "person detection" examples within both the official TinyML book [19] and the TensorFlow Lite for Microcontrollers GitHub repository [15]. This established use case demonstrates its suitability for microcontroller applications, making it a relevant benchmark for our work.

MobileNetV1_25

We began with the MobileNetV1_25, with a width multiplier of α=0.25. This specific version was chosen as it directly corresponds to the architecture used in the aforementioned TinyML book [19] and TensorFlow Lite for Microcontrollers [15] examples.

Unlike AykoNet's preprocessing pipeline, images were resized to 32×32 pixels but kept in their original 3-channel RGB format, as the standard MobileNet architecture expects this input. TensorFlow's built-in `preprocess_input` function was used for normalization.

The model was constructed by loading the MobileNet base architecture from `tensorflow.keras.applications.mobilenet` with random weights and adding a GlobalAveragePooling2D layer followed by a final Dense layer with a softmax activation for the 43 traffic sign classes. We adapted training configurations from the TensorFlow Lite for Microcontrollers GitHub repository's person detection example [15]. The repository authors note that hyperparameter tuning is non-trivial [15]. For this reason, we adopted their specified parameters.

Specifically, we used an Adam optimizer with a scheduled learning rate, starting at 0.045 and decaying every 2.5 epochs by a factor of 0.98. The model was trained with a batch size of 96, using categorical cross-entropy loss with a label smoothing factor of 0.1. To prevent overfitting, we implemented an early stopping callback to monitor validation loss with patience of 10 epochs, restoring the best-performing weights.

After training, the model was converted to the TensorFlow Lite format and underwent full 8-bit integer quantization, following the guidelines in the TinyML book and the TensorFlow Lite Micro repository [19; 15].

MobileNetV1_20
Upon quantization, we discovered that the memory footprint of the MobileNetV1_25 model was too large for the Raspberry Pi Pico's memory constraints. To obtain a deployable MobileNet variant, we trained a smaller version with the width multiplier reduced to α=0.20. All other aspects of the preprocessing, model architecture, training, and quantization process were identical to those described for MobileNetV1_25.

4.2 GiordyNet

As a domain-specific baseline, we implemented GiordyNet based on the architecture and methodology described in its original report [6]. We trained the model on the GTSRB dataset, with all images preprocessed into 32×32 grayscale images, where the report identifies 32x32 image size as optimal for balancing detail and efficiency.

For data augmentation, we followed the report's suggestion of applying photometric distortions to handle variations in lighting. Specifically, we applied gamma correction with a random value between 0.4 and 1.5. A key difference from AykoNet is how this augmentation was applied; in line with the GiordyNet methodology, it was applied randomly to 30% of the entire training set, rather than being used as part of a class-aware strategy to rebalance the dataset. This differs from AykoNet's class-aware strategy because it applies augmentation proportionally based on class frequency to address dataset imbalance more systematically.

The GiordyNet architecture consists of three sequential convolutional blocks, each containing a Conv2D layer followed by MaxPooling2D, BatchNormalization, and ReLU activation [6]. The feature maps are then flattened and passed through two Dense layers, with the final layer using a softmax activation for classification.

We trained the model using an Adam optimizer and an early stopping callback that monitored validation loss with patience of 10 epochs, restoring the best weights. After training, we converted the model to TensorFlow Lite and quantized it following the workflow outlined in the TinyML book [19].

4.3 AykoNet

We trained AykoNet on Google Colab using the augmented GTSRB dataset, following the complete methodology outlined in Section 3.

4.4 Deployment

Hardware Platform
We deployed the models on the Raspberry Pi Pico which features a dual-core ARM Cortex-M0+ running at 133 MHz, with 264 KB of SRAM and 2 MB of flash. Due to its constrained memory architecture, the Pico introduces challenges for TinyML deployment, particularly as its limited SRAM must hold intermediate activations and temporary buffers during inference [18].

Software Framework
We used the official TensorFlow Lite Micro port for the Raspberry Pi Pico [17] for model deployment. This framework provides optimized inference routines specifically designed for ARM Cortex-M0+ architecture. We integrated quantized models into the TensorFlow Lite Micro runtime. Model weights were stored in flash memory, while intermediate computations were executed in SRAM during inference.

4.5 Evaluation Metrics And Results

Model Size
We measured 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 TensorFlow Lite for Microcontrollers guidelines, we assumed that models exceeding 250KB would not fit within the Pico's deployment constraints, as this threshold ensures sufficient memory remains available for system components [15].

Classification Accuracy
We evaluated classification accuracy using quantized TensorFlow Lite models with 8-bit integer precision, as these models represent the intended microcontroller deployment. We performed the evaluation on 1,000 randomly selected images from the official GTSRB test set (which contains a total of 12,629 images).

Inference Latency
We evaluated real-time performance by measuring the inference time of quantized models on the hardware platform. To ensure consistency, we evaluated all deployable models using a standardized protocol with 43 test images from the GTSRB dataset (one per traffic sign class). We recorded inference times using the Pico's timers (`get_absolute_time()` and `absolute_time_diff_us()`), specifically measuring the TensorFlow Lite Micro interpreter's `Invoke()` call duration. This isolates neural network computation time from preprocessing and post-processing overhead. We summed inference times over all 43 images and averaged them to derive representative latency metrics.

4.6 Results

We evaluated model performance based on three key metrics: classification accuracy, model size, and inference latency. The results are summarized in the tables below.

Model Size
We measured model sizes using the file size of the TensorFlow Lite models as described in Section 4.5. We only report quantized versions, as they represent the deployable models. Table 3 lists the results.

Despite using depthwise separable convolutions for efficiency, MobileNetV1 models have the largest sizes due to their general-purpose architecture 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.

Table 3: 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

MobileNetV1_25 exceeds 300KB. Even though it surpasses the 250KB threshold recommended in the TFLite Micro repository [15], 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 [15], we consider models exceeding 250KB 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 307.59KB to 217.79KB), highlighting the effectiveness of width multipliers for compressing and scaling models.

GiordyNet (106.87KB) and AykoNet-Pro (80.18KB) achieve similar model sizes, indicating similar architectural complexity. AykoNet-Lite (36.80KB) represents an aggressive minimization approach, achieving a 54% reduction compared to AykoNet-Pro.

Classification Accuracy

We evaluated the quantized TensorFlow Lite models on the test set, as described in Section 4.5. The results are presented in Table 4.

Table 4: 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%

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 307.59KB to 217.79KB), 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 perfomance 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 36.80KB 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.

Inference Latency

We measured inference time on the Raspberry Pi Pico as described in 4.5. We excluded MobileNetV1_25-int8 as its size exceeds the 250KB threshold (see Section 4.5). Table 5 shows the latency results.

Table 5: Inference Latency

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

We exclude MobileNetV1_25 from on-device testing due to its 307.59KB size exceeding the established 250KB deployment threshold for Raspberry Pi Pico, as detailed in Section 4.5.

GiordyNet exhibits the highest inference latency (204.08ms), 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 (77.29ms) despite processing RGB images (3 channels) and having a model size of 217.79KB. Its exclusive use of depthwise separable convolutions throughout the network, combined with optimizations specifically designed for TensorFlow Lite Micro's interpreter [9; 3], demonstrates the effectiveness of hardware-aware architectural design for embedded deployment on resource-constrained microcontrollers.

AykoNet-Pro records an inference time of 87.13ms, 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 (55.34ms), 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 Discussion

By developing and evaluating AykoNet, we aimed to identify 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 (36.80KB) and the fastest inference time (55.34ms). 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, rendering it unsuitable for autonomous systems. AykoNet overcomes this limitation by integrating efficient depthwise separable convolutions. To maintain high classification accuracy despite this optimization, AykoNet applies class-aware data augmentation on the GTSRB dataset, which led AykoNet-Pro to achieve higher accuracy than GiordyNet while being smaller and faster.

Our experimental design had several limitations. To address the class imbalance, we developed a data augmentation pipeline without exploring undersampling approaches, which could offer valuable comparisons. Additionally, the specific contribution of class-aware data augmentation remains unclear since we did not evaluate AykoNet variants without it. Lastly, we evaluated the models on a subset of 1000 images rather than the complete GTSRB test set, limiting the statistical significance.

Based on our findings, designing an optimal TinyML model for real-time traffic sign recognition on the Raspberry Pi Pico using the GTSRB dataset requires three key principles. First, **domain-specific preprocessing and architectural design**, as demonstrated by the superior performance of both AykoNet and GiordyNet compared to MobileNetV1_20. Second, **class-aware data augmentation**, implemented in AykoNet to address dataset imbalance and enhance accuracy. Third, **the strategic use of depthwise separable convolutions**, which enabled AykoNet to achieve significant efficiency gains over GiordyNet while maintaining high accuracy. For practical deployment, **AykoNet-Lite** emerges as the optimal solution, providing the best balance between classification accuracy (94.60%), model size (36.80KB), and inference latency (55.34ms) within hardware constraints.

6 Responsible Research

6.1 Ethical Considerations

Traffic sign recognition is a low-risk domain. While the dataset we use may incidentally capture individuals or vehicles in the background, it is publicly available and widely used in literature. Our research focuses solely on the traffic signs, and any background elements are not analyzed or used in any way.

6.2 Reproducibility

We provided detailed documentation of our hardware, software, training procedures, and evaluation protocols to ensure reproducibility. The dataset used is publicly available, and model settings are specified with fixed random seeds used for consistent results. All code and configuration files will be made available in a public repository to enable replication and support further research.

7 Conclusion

This research addressed the question: How can we create an optimal TinyML model for real-time traffic sign recognition on microcontrollers?, where optimal is defined as balancing model size, classification accuracy, and inference latency within the hardware constraints. We proposed a new model architecture called AykoNet with two variants: one prioritizing classification accuracy (AykoNet-Pro) and one prioritizing model size and inference latency (AykoNet-Lite). We investigated some of the important design decisions leading to an optimal model. We then compared AykoNet variants with existing architectures relevant to our work. We concluded by demonstrating AykoNet's effectiveness by highlighting the importance of image preprocessing, data augmentation, and depthwise separable convolution layers. As a next step, we plan to integrate AykoNet into an autonomous navigation system with camera integration, enabling real-time traffic sign recognition and autonomous vehicle response.

A GTSRB Class Labels

The GTSRB dataset contains 43 different traffic sign classes as follows:

0: Speed limit (20km/h)
1: Speed limit (30km/h)
2: Speed limit (50km/h)
3: Speed limit (60km/h)
4: Speed limit (70km/h)
5: Speed limit (80km/h)
6: End of speed limit (80km/h)
7: Speed limit (100km/h)
8: Speed limit (120km/h)
9: No passing
10: No passing for vehicles over 3.5 metric tons
11: Right-of-way at intersection
12: Priority road
13: Yield
14: Stop
15: No vehicles
16: Vehicles over 3.5 tons prohibited
17: No entry
18: General caution
19: Dangerous curve left
20: Dangerous curve right
21: Double curve
22: Bumpy road
23: Slippery road
24: Road narrows on the right
25: Road work
26: Traffic signals
27: Pedestrians
28: Children crossing
29: Bicycles crossing
30: Beware of ice/snow
31: Wild animals crossing
32: End of all speed and passing limits
33: Turn right ahead
34: Turn left ahead
35: Ahead only
36: Go straight or right
37: Go straight or left
38: Keep right
39: Keep left
40: Roundabout mandatory
41: End of no passing
42: End no passing vehicles over 3.5 tons

B Data Augmentation Results

Class ID	Before	After	Final
0	210	630	840
19	210	630	840
37	210	630	840
27	240	720	960
41	240	720	960
42	240	720	960
32	240	720	960
24	270	810	1080
29	270	810	1080
39	300	900	1200
21	330	660	990
40	360	720	1080
20	360	720	1080
36	390	780	1170
22	390	780	1170
16	420	840	1260
34	420	840	1260
6	420	840	1260
30	450	900	1350
23	510	510	1020
28	540	540	1080
26	600	600	1200
15	630	630	1260
33	689	689	1378
31	780	780	1560
14	780	780	1560
17	1110	333	1443
18	1200	360	1560
35	1200	360	1560
11	1320	396	1716
8	1410	423	1833
3	1410	423	1833
7	1440	432	1872
9	1470	441	1911
25	1500	450	1950
5	1860	0	1860
4	1980	0	1980
10	2010	0	2010
38	2070	0	2070
12	2100	0	2100
13	2160	0	2160
1	2220	0	2220
2	2250	0	2250

References

[1] Michael Breiter, Adrian Fazekas, Tobias Volkenhoff, and Markus Oeser. Video based intelligent transportation systems – state of the art and future development. *Transportation Research Procedia*, 14:4495–4504, 12 2016.

[2] Tejas Chaudhari, Ashish Wale, Amit Joshi, and Suraj Sawant. *Traffic Sign Recognition Using Small-Scale Convolutional Neural Network*, 2020.

[3] Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Shlomi Regev, Rocky Rhodes, Tiezhen Wang, and Pete Warden. *TensorFlow Lite Micro: Embedded Machine Learning on TinyML Systems*, 2020.

[4] Esma Dilek and Murat Dener. Computer vision applications in intelligent transportation systems: A survey. *Sensors*, 23(6):2938, 2023.

[5] Chunhui Du, Shenglan Su, Cheng Lin, et al. A lightweight network for traffic sign detection via multiple scale context awareness and semantic information guidance. *Scientific Reports*, 15:10110, 2025.

[6] Marco Giordano. *Traffic Sign Recognition, CNN on Microcontrollers*, 2020.

[7] Google AI Edge Authors. *Post-training Integer Quantization*, 2024.

[8] Sebastian Houben, Johannes Stallkamp, Jan Salmen, Marc Schlipsing, and Christian Igel. Detection of traffic signs in real-world images: The German Traffic Sign Detection Benchmark. In *International Joint Conference on Neural Networks*, number 1288, 2013.

[9] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[10] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and ¡0.5mb model size, 2016.

[11] Ida Syafiza Binti Md Isa, Choy Ja Yeong, and Nur Latif Azyze bin Mohd Shaari Azyze. Real-time traffic sign detection and recognition using raspberry pi. *International Journal of Electrical and Computer Engineering (IJECE)*, 12(1), 2022.

[12] Jonghoon Jin, Aysegul Dundar, and Eugenio Culurciello. Flattened convolutional neural networks for feedforward acceleration, 2015.

[13] Girish Kumar N G, Ashish Kishore, and Aaditya Krishna. Real-time traffic sign recognition and autonomous vehicle control system using convolutional neural networks. *Multimedia Tools and Applications*, pages 1–36, 04 2025.

[14] A. Radha Rani, Y. Anusha, S.K. Cherishama, and S. Vijaya Laxmi. Traffic sign detection and recognition using deep learning-based approach with haze removal for autonomous vehicle navigation. *e-Prime - Advances in Electrical Engineering, Electronics and Energy*, 7:100442, 2024.

[15] Yuan Tang. *TensorFlow Lite for Microcontrollers*.

[16] Min Wang, Baoyuan Liu, and Hassan Foroosh. Design of efficient convolutional layers using single intra-channel convolution, topological subdivisioning and spatial "bottleneck" structure, 2017.

[17] Pete Warden. *Pico TensorFlow Lite Port*.

[18] Pete Warden. *Understanding the Raspberry Pi Pico's Memory Layout*, 2024.

[19] Pete Warden and Daniel Situnayake. *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers*. O'Reilly Media, 2019.