Article

Optimizing Convolutional Neural Networks for Image Classification on Resource-Constrained Microcontroller Units

Susanne Brockmann * and Tim Schlippe *

IU International University of Applied Sciences, 99084 Erfurt, Germany
* Correspondence: susanne.e.brockmann@gmail.com (S.B.); tim.schlippe@iu.org (T.S.)

Abstract: Running machine learning algorithms for image classification locally on small, cheap, and low-power microcontroller units (MCUs) has advantages in terms of bandwidth, inference time, energy, reliability, and privacy for different applications. Therefore, TinyML focuses on deploying neural networks on MCUs with random access memory sizes between 2 KB and 512 KB and read-only memory storage capacities between 32 KB and 2 MB. Models designed for high-end devices are usually ported to MCUs using model scaling factors provided by the model architecture’s designers. However, our analysis shows that this naive approach of substantially scaling down convolutional neural networks (CNNs) for image classification using such default scaling factors results in suboptimal performance. Consequently, in this paper we present a systematic strategy for efficiently scaling down CNN model architectures to run on MCUs. Moreover, we present our CNN Analyzer, a dashboard-based tool for determining optimal CNN model architecture scaling factors for the downsampling strategy by gaining layer-wise insights into the model architecture scaling factors that drive model size, peak memory, and inference time. Using our strategy, we were able to introduce additional new model architecture scaling factors for MobileNet v1, MobileNet v2, MobileNet v3, and ShuffleNet v2 and to optimize these model architectures. Our best model variation outperforms the MobileNet v1 version provided in the MLPerf Tiny Benchmark on the Visual Wake Words image classification task, reducing the model size by 20.5% while increasing the accuracy by 4.0%.

Keywords: TinyML; image classification; microcontroller units

1. Introduction

In recent years, deep neural networks (DNNs), especially convolutional neural networks (CNNs), have surpassed human-level accuracy on a broad range of tasks, including image classification [1], object detection [2], and instance segmentation [3]. Following AlexNet [1], a trend of building deeper [4], wider [5], and more complex networks [6,7] has started to improve the accuracy of these models.

However, such large models do not fit within the memory and computational constraints of mobile devices such as mobile phones, autonomous robots, drones, and other intelligent systems with cameras [8]. Therefore, many mobile applications offload their computationally heavy machine learning inference to the cloud, which comes with drawbacks in terms of bandwidth, inference time, energy, economics, and privacy [9]. These issues, along with the need to run real-time inference on the edge, have initiated the development of new types of smaller neural networks such as SqueezeNet [10], MobileNets [11–13], and ShuffleNets [14,15] for image classification. However, these smaller neural networks still do not meet the resource constraints of many Internet of Things (IoT) devices [16], which often leads to discarding of captured sensor data. Consequently, there is a growing need for tiny models able to run on the microcontroller units (MCUs) embedded within IoT devices.

The new research field of TinyML is focused on deploying neural networks on small (~1 cm³), cheap (~$1), low-power (~1 mW), and widely available MCUs with random
access memory (RAM) sizes between 2 KB and 512 KB and read-only memory (ROM) storage capacities between 32 KB and 2 MB [9,17]. Examples of such IoT use cases include the processing of sensor data in smart manufacturing, personalized healthcare, automated retail, wildlife conservation, and precision agriculture contexts. In many of these fields, image classification plays an important role.

When seeking to obtain convolutional neural networks (CNNs) for image classification that fit the aforementioned constraints, CNNs for high-end edge devices are often ported to MCUs by reducing the input channels from RGB to grayscale [9], reducing the input resolution [9,18], or by drastically decreasing the default model architecture scaling factor of the model, such as the width multiplier $\alpha$ in MobileNets [11–13]. However, our analysis, which we will present in Section 6.2.1, shows that the naive approach of reducing the default model scaling factors leads to suboptimal results when substantially scaling down the model architecture.

Consequently, in this study we elaborate a systematic strategy to efficiently optimize CNN model architectures for running image classification on MCUs. Our goal was to optimize tiny models that fit the following MCU constraints, which are also recommended in the TinyML literature [18]:

- $\leq 250$ KB RAM
- $\leq 250$ KB ROM
- Inference cost $\leq 60$ M multiply–accumulate operations (MACs)

For our experiments, we used the Visual Wake Word (VWW) dataset [18] with a resolution of $96 \times 96 \times 3$ pixels. The VWW dataset was specifically designed for the MCU use case of classifying whether a person is present in an image and is an important part of the MLPerf Tiny Benchmark [19].

We developed our CNN Analyzer, a dashboard-based tool, to gain layer-by-layer insights into the model architecture scaling factors that have the potential to minimize model size, peak memory, and inference time. Using our strategy together with our CNN Analyzer, we were able to (1) locate the bottlenecks of the model; (2) introduce new model architecture scaling factors for MobileNet v1 [11], MobileNet v2 [12], MobileNet v3 [13], and ShuffleNet v2 [15]; and (3) optimize these model architectures. This would not have been possible with a neural architecture search (NAS) approach, as in [9,20–22], since NAS requires the definition of the search space in advance and does not provide layer-by-layer insights. In summary, our contributions are as follows:

- We investigated and developed a strategy to optimize existing CNN architectures for given resource constraints.
- We created the CNN Analyzer to inspect the metrics of each layer in a CNN.
- Our model implementations use the TensorFlow Lite for Microcontrollers inference library, as it can run on almost all available MCUs [23].
- We introduced new model architecture scaling factors to optimize MobileNet v1, MobileNet v2, MobileNet v3, and ShuffleNet v2.
- We have published the CNN Analyzer and its related code on our GitHub repository: https://github.com/subrockmann/tiny_cnn (accessed on 3 June 2024)

Our findings and developed tools are portable to other network architectures and can be combined with NAS approaches. While the goal of this paper is to increase performance of models that already fit the aforementioned MCU constraints, our strategy and the developed CNN Analyzer can also be applied to fit models into MCU constraints that originally require more resources.

2. Resource Constraints of Microcontroller Units

As recommended in the TinyML literature [18], our goal was to optimize tiny models that fit into 250 KB RAM and 250 KB ROM while having inference costs of less than 60 M MACs, as this would lead to inference times of less than 1 s.
Examples for high-end MCUs which require those constraints are ESP32 Xtensa LX6 (4 MB ROM, 520 KB RAM), Arduino Nano 33 Cortex-M4 (1 MB ROM, 256 KB RAM), Raspberry Pi Pico Cortex-M0+ (16 MB ROM, 264 KB), and STM32F746G-Disco board Cortex-M7 (1 MB ROM, 340 KB RAM), which we used for our experiments described in Section 6. Although the available ROM of these MCUs exceeds the 250 KB required to store the model, the storage overhead for the entire application utilizing the model must also be taken into account. Furthermore, these high-end MCUs "are used in a huge range of use cases, from sensing and IoT to digital gadgets, smart appliances and wearables. At the time of writing, they represent the sweet spot for cost, energy usage, and computational ability for embedded machine learning" [16].

For running inference of neural networks on MCUs, all static data, including program code and model parameters, have to fit into the ROM, while temporary data such as model activations must fit into the RAM. The RAM required for neural network inference varies throughout the layers, and is determined by intermediate tensors that must be stored for data transfer between layers. The largest sum of input and output tensors of an operation plus all other tensors that must be kept in the RAM for subsequent operations [24], is known as the peak memory. The amount of ROM needed for an application is the sum of the operating system size, the machine learning framework size, the neural network model size, and the application code size. The number of MACs or floating point operations (FLOPs) is used to measure the inference cost.

While the number of MACs and FLOPs has an impact on accuracy, inference time, and energy consumption, storage-related metrics such as the number of model parameters, which impacts the model size, and the peak memory, which determines the RAM requirements, are crucial metrics for running neural networks on resource-constrained MCUs. Consequently, it is relevant to achieve a trade-off between high accuracy, low inference time, minimal storage requirements, and low energy consumption. The authors of [18] used the number of model parameters as a proxy for model size, which requires 1 byte storage for each parameter using int-8 quantization. However, this neglects the additional storage requirements for metadata, computation graphs, and other information necessary for training and inference. Due to this relationship, models with fewer model parameters may have a higher model size than models with more model parameters. For example, the MLPerf Tiny Benchmark model of MobileNet v1 with scaling factor $\alpha = 0.25$ requires a total memory which is 1.36 times larger than the size for storing the model parameters alone. Consequently, in our strategy for optimizing CNN model architectures, we introduce the bytes/parameter ratio as a new evaluation metric to estimate the number of model parameters that have to be reduced to fit a model into the given constraints. For example, $\text{bytes/parameter ratio} = 1.3$ indicates that in order to reduce the model size by 1000 bytes, we need to reduce it by approximately 1300 model parameters.

As we will explain in Section 5.3, we capture the aforementioned metrics in our CNN Analyzer to derive optimization strategies.

3. Related Work

In this section, we will first describe techniques for reducing the size of neural networks and designing CNNs that require low computational resources (so-called efficient CNNs). Then, we will present efficient CNN architectures designed for mobile devices and MCUs.

3.1. Techniques for Reducing the Size of Neural Networks

Neural networks are usually highly over-parametrized, containing many redundant model parameters that do not contribute to the accuracy of the network [25]. Therefore, pruning [26–28] is used to remove less relevant model parameters. This reduces model size while preserving accuracy. However, the drawback of pruning is that it has the effect of creating a sparse model, and currently there are very few edge AI hardware and open-source software options that support the use of sparse models [16,29].
Another approach for reducing model size is quantization. Quantization maps high-precision weight values to low-precision weight values, reducing the number of bits needed for storing each weight. For example, [30] proposed full-integer quantization (int-8 quantization) of weights and activations to leverage integer-only hardware accelerators, which can improve inference time, computation, and power usage. The authors of [31] suggested knowledge distillation, which transfers knowledge from a large teacher model to a smaller student model by learning mappings from input to output vectors.

3.2. Techniques for Designing Efficient Convolutional Neural Networks

Convolutional layers are the core components of CNNs. These layers extract features from an input image using convolutional filters, which are small matrices that slide over the input image one patch at a time. Each filter performs an element-wise multiplication with the corresponding patch and sums the results to produce a single output value. In comparison to fully-connected layers, in convolutional layers each neuron only connects to the small rectangular input patch of the previous layer, which reduces the number of model parameters in the layer and makes them more efficient for image processing.

When developing efficient CNN architectures for edge devices from scratch, the standard convolutional layers can be replaced by less computationally complex convolutional layers, such as the depthwise separable convolutions [32] in [11–13], grouped convolutions [1] in [14,15], and factorization of convolutions [33,34].

Moreover, approaches that were originally designed to increase accuracy by increasing model size can also be used for model size reduction. The authors of [4] added additional layers to increase the model’s depth. Other approaches introduce model architecture scaling factors, which impacts the model’s width by increasing the number of channels [5,11–13] (e.g., the width multiplier $\alpha$ in MobileNets), increasing the image resolution [11–13] (e.g., the resolution multiplier $\rho$ in MobileNets), or increasing all three dimensions (model depth, number of channels, and image resolution) [35].

3.3. Efficient Convolutional Neural Networks

In the following section, we will present CNN architectures have been developed using the techniques mentioned in Section 3.2 to specifically run on mobile devices and MCUs.

3.3.1. Efficient Convolutional Neural Network Architectures for Mobile Devices

The first model specifically designed for image classification on mobile and edge devices was MobileNet v1 [11]. It uses depthwise separable convolutions instead of standard convolutional layers, thereby drastically reducing both computation and model size. MobileNet v2 [12] introduced inverted residuals and skip connections to improve accuracy while maintaining similar inference time and model size. MobileNet v3 [13] was further optimized through the use of neural architecture search, and introduced the efficient activation functions hard-swish and hard-sigmoid. All MobileNet architectures offer the width multiplier $\alpha$ and resolution multiplier $\rho$ as hyperparameters, which can be used to balance the trade-off between accuracy, model size, and inference time.

ShuffleNet v1 [14] replaces the standard convolutional layers with pointwise group convolution and channel shuffle, two operations that greatly reduce the computation cost while maintaining accuracy. ShuffleNet v2 [15] introduced a channel split operation, and adheres to design guidelines that promote equal channel width while avoiding excessive group convolution, network fragmentation, and element-wise operations.

As we decided to experiment with models which are originally smaller than 250 KB, as explained in Section 5.1, we did not consider the following model architectures for the experimental part of our research: SqueezeNet [10], SqueezeNext [36], CondenseNet [37], NASNet-A [20], PNASNet [38], MnasNet-A1 [39], EfficientNet-B0 [35], AmoebaNet-A [40], DARTS [41], FBNet-A [42], and GhostNet [43]. Additionally, our analyses of vision transformers demonstrated that even those optimized for mobile devices exceed the MCU constraint of 250 KB model size. For example, MobileViT-XXS [44] has 1.9 M model param-
eters leading to a model size of more than 1.9 MB, as each model parameter uses 1 byte of storage using int-8 quantization, as described in Section 2.

3.3.2. Efficient Convolutional Neural Networks for Microcontrollers

The first efforts to use existing efficient CNN architectures on MCUs were conducted by [18]; they reported an accuracy of less than 80% with 208 K model parameters (MobileNet v1 [11]) and less than 85% with 290 K (MobileNet v2 [12]) and 400 K (MnasNet [39]) model parameters on the VWW dataset with a resolution of 96 × 96 × 3. In all three cases, they did not report the model sizes; however, if we use the number of model parameters as a proxy for model size, which requires 1 byte of storage for each model parameter using int-8 quantization, only MobileNet v1 fits our model constraint of model size <250 KB. However, it does not reach a minimum accuracy of 80%.

Several efficient CNN architectures have been explicitly designed to run on MCUs. For example, in order to reduce computational complexity, Effnet [33] separates 3 × 3 kernels into depthwise kernels and introduces separable pooling. The model architecture is designed for an input resolution of 32 × 32 × 3 pixels, which does not match our use case of 96 × 96 × 3 pixels, as explained in Section 5.1. Therefore, we omitted EffNet from our experiments.

IoTNet [34] is another CNN architecture specifically designed for IoT devices. Unlike EffNet, IoTNet uses a sequence of 1 × 3 and 3 × 1 standard convolutions instead of depthwise convolutions. As this model is also only designed for a small input resolution of 32 × 32 × 3 pixels, and no code implementation is provided, we excluded IoTNet from our experiments.

MicroNets [9] were developed by combining differential architecture search (DARTS) [41], quantization-aware training [45], and knowledge distillation [31]. The authors used a MobileNet v2 backbone [12] and the VWW dataset [18], the same dataset that we used to optimize CNNs on MCUs in our experiments (see Section 5.1). Unfortunately, their paper [9] provides neither the model nor details about the MicroNet model architecture; hence, we could not include the MicroNet architectures in our experiments.

Another method to produce efficient CNNs for MCUs is Sparse Architecture Search (SpArSe) [21]. SpArSe uses a combination of neural architecture search, pruning, and network morphism. Currently, very few edge AI hardware and open-source software options support sparse models generated by pruning [16,29]. Therefore, we did not use SpArSe or other methods for pruning in our experiments.

The model parameters of MCUNet v1 [22] are determined using a two-stage neural architecture search method (TinyNAS) that first optimizes the search space based on MnasNet [39] according to the MCU constraints, then trains a super network that contains all the possible sub-networks through weight sharing. To run the resulting MCUNet v1 models on MCUs, [22] developed the specific memory-efficient TinyEngine inference library. MCUNet v2 [46] extended the work of MCUNet v1 and introduced patch-based inference and receptive field redistribution for the memory-intensive layers to overcome the RAM bottleneck in the first layers. Although MCUNet v1 and MCUNet v2 reach more than 90% accuracy on the VWW dataset, they do not meet our model size constraints, as they require significantly more than 250 KB ROM; specifically, MCUNet v1 requires 1007 KB, and MCUNet v2 requires 1010 KB. Furthermore, our goal was to use the TensorFlow Lite for Microcontrollers (TFLM) inference library, which runs on most MCUs [23]; however, these models are not compatible with TensorFlow.

µNAS [47] is a neural architecture search method that uses aging evolution and dynamic model pruning to find network architectures with low computational requirements of up to 64 KB of ROM and RAM. However, the model search was computationally too expensive for our use case of 96 × 96 × 3 pixels; for instance, finding an optimal model with µNAS took [47] 23 GPU days on CIFAR10 with a 32 × 32 × 3 pixels image resolution.
3.3.3. Comparison to Our Work

As described above, many recent papers on optimizing CNNs for image classification on MCUs have applied a combination of (1) the creation of a CNN architecture (e.g., a specialized model architecture \[33,34\] or neural architecture search \[9,21,22,46,47\]) and (2) optimization steps, e.g., a unique training procedure \[22,46\], model compression techniques such as quantization \[9,22,46\] and pruning \[21,47\], and sometimes even a specialized inference framework \[22,46\].

In contrast, our focus was to find a solution for the first issue, i.e., optimal CNN model architectures for the MCU use case. Consequently, we did not apply any of the optimization steps from (2) apart from int-8 quantization. However, the model variations created through our optimization strategy can be further enhanced through the aforementioned optimization steps.

4. Our Strategy for Optimizing CNNs on MCUs

To compare and optimize CNNs on MCUs, we developed a strategy by which each model architecture is evaluated according to the process depicted in Figure 1. In the next subsections, we will describe the steps of our strategy in detail.

![Figure 1. Our strategy for optimizing CNNs on MCUs.](image)

4.1. Step 1: Create Model Variations

We create model variations in two ways: (1) We create untrained model variations using default model architecture scaling factors (e.g., different values for the width multiplier $\alpha$ used in MobileNet v1 \[11\]). (2) If we find new model architecture scaling factors in step 2, we create new model variations using the new model architecture scaling factors.

4.2. Step 2: Analyze Model Variations with CNN Analyzer

(1) We use our CNN Analyzer to check whether the model variations fit our constraints. A model variation that fits the constraints is sent to step 3 for training. (2) If the distance between at least one model metric and constraint is above a threshold, that model variation...
is discarded. (3) For each of the remaining model variations, we investigate how to make the
model variation fit our constraints by changing the model architecture scaling factors. Then,
we proceed to step 1 to build a new model variation with these new model architecture
scaling factors.

4.3. Step 3: Train and Evaluate Model Variations

(1) We train all remaining model variations on our dataset using the same setup for
training. (2) We evaluate them on the same test set. (3) From each model architecture, we
select the five best-performing model variations that exceed an accuracy of 80% on the test
set to proceed to step 4.

4.4. Step 4: Evaluate Model Variations on MCU

(1) We convert the model variation to a c-byte array, (2) compile the model and
evaluation code, (3) flash the resulting compiled code on the MCU, and (4) measure the
inference time on the MCU.

5. Experimental Setup

In this section, we will first introduce the dataset we used to optimize and test our
model variations. Second, we will explain how we used TensorFlow, TensorFlow Lite and
TensorFlow Lite for Microcontrollers to transfer CNN models on an MCU. Then, we will
present the CNN Analyzer which we implemented to determine the optimal CNN model
architecture scaling factors for our down-scaling strategy. Finally, we will describe how we
tested our best models on a real MCU.

5.1. Dataset

Commonly used datasets for image classification include ImageNet [48] and CI-
FAR10 [49]. However, ImageNet [48], with 1000 classes, is not an appropriate dataset
for our MCU use case [18]. Furthermore, the resolution of the CIFAR10 images [49]
(32 × 32 × 3 pixels) is too small for most real-world IoT use cases [18].

Consequently, for our experiments we used the VWW dataset [18], which consists of
109,620 images (80% training, 10% validation, 10% test) with a resolution of 96 × 96 × 3 pix-
els. The VWW dataset was specifically designed for the MCU use case of classifying
whether a person is present in an image, and is an important part of the MLPerf Tiny
Benchmark [19]. Following this benchmark, we used the constraints defined in Section 2
and a minimum accuracy of 80%. Our goal was to find a model variation that reaches
maximum accuracy on the VWW test set while staying within these resource constraints.

5.2. Running CNNs on MCUs

To keep our research platform-independent, we use the open source TensorFlow frame-
work for model creation, TensorFlow Lite for optimization, and the TensorFlow Lite for
Microcontrollers inference runtime [23] for running the models on the MCU. Consequently,
our work is not restricted to a specific MCU type and allows for a portable deployment
of models across different hardware platforms. To implement a CNN model that runs on
MCUs, we need to apply the following steps:

1. Build a TensorFlow model representation: To build the model variation with our model
architecture scaling factors, we used the TensorFlow framework.
2. Convert to TensorFlow Lite model representation: To optimize the model for inference
on mobile devices.
3. Convert to TensorFlow Lite for Microcontrollers model representation: The optimized
TensorFlow Lite model representation is compiled to a c-byte array, which is necessary
in order to run it on MCUs.

In each of the three steps, we retrieve metrics and tabular data for further analysis in
the CNN Analyzer, as described in Section 5.3.2.
5.3. CNN Analyzer: A Dashboard-Based Tool for Determining Optimal CNN Model Architecture Scaling Factors

To determine optimal model architecture scaling factors for given constraints such as accuracy, model size, peak memory, and inference time in steps 1–3 of our optimization strategy (described in Section 4), we developed the CNN Analyzer. This toolkit allows TensorFlow models to be built with different model architecture scaling factors, and enables the storage, analysis, visualization, comparison, and optimization of the model variations.

5.3.1. Model Scorecard

As shown in Figure 2, CNN Analyzer displays metrics, layer-wise visualizations, and tabular data in a scorecard. Example metrics include the number of model parameters, model size, peak memory, inference time, and accuracy. Layer-wise visualizations display the input height, number of output channels, model parameters, and number of MACs and FLOPs. The layer-wise visualizations are based on the tabular data, which are also displayed for more detailed exploration.

![Model scorecard from our CNN Analyzer.](image)

5.3.2. Implementation

CNN Analyzer is powered by a collection of existing and self-developed analytical tools that analyze and benchmark the model representations created for each model variation. In an interactive Jupyter notebook, the user can choose the model architecture, define the model architecture scaling factors, and begin building, conversion, and analysis of model variations. The extracted information of the different model variations, including its compound model name, is logged in the model database of CNN Analyzer to keep track of...
all the different model architectures and model architecture scaling factors. The machine learning operations (MLOps) tool Weights & Biases (https://www.wandb.com, accessed on 4 July 2024) is used to log all model architecture scaling factors as well as to track and visualize all model training runs. All metrics and tabular data of each model variation are retrieved from the TensorFlow, TensorFlow Lite, and TensorFlow Lite for Microcontrollers representations, which are described in Section 5.2.

*TensorFlow* provides the `tf.model.summary` method (https://www.tensorflow.org/api_docs/python/tf/keras/Model#summary, accessed on 4 July 2024) for generating a layer-wise summary report with layer names, layer types, number of channels, output shape, and number of model parameters, as well as a summary of the total MACs and FLOPs of the model variation.

To capture the layer-wise RAM requirements and peak memory of the model variation, we used `tflite-tools` (https://github.com/eliberis/tflite-tools, accessed on 4 July 2024) created by [24] to analyze the TensorFlow Lite model representations. Additionally, we utilized the TensorFlow Lite native benchmarking binary (https://www.tensorflow.org/lite/performance/measurement#native_benchmark_binary, accessed on 4 July 2024), which can run on Linux, Mac OS, and Android devices and creates a report with average inference time on the CPU and a breakdown of the inference time per layer.

To measure the inference time on MCUs, the TensorFlow Lite model representation has to be compiled into a c-byte array. Since compiling the model representation together with its corresponding runtime code and uploading it to the MCU for inference time profiling requires many manual steps, we first simulated the inference using a hardware simulator. To simulate the inference, we used the Silicon Labs Machine Learning Toolkit (MLTK) (https://siliconlabs.github.io/mltk, accessed on 4 July 2024), which provides a model profiler that uses a hardware simulator to estimate the inference time and CPU cycles per layer (based on the ARM Cortex-M33). To compile the model and flash it on the MCU for the final inference time evaluation, we used STM32.Cube.AI (https://stm32ai.st.com/stm32-cube-ai, accessed on 4 July 2024). The STM32.Cube.AI software framework supports profiling of TensorFlow Lite models on locally connected hardware such as the STM32F746G-Disco board, which we used for our experiments. STM32.Cube.AI creates detailed reports including the model size, peak RAM, and inference time as well as a layer-wise breakdown of the MACs, number of model parameters, and inference time on the MCU.

5.3.3. Naming Conventions for the Analyzed Models

Within our CNN Analyzer, all model variations are named according to the following scheme: `<base model> <α> <image resolution> <input channels> <classes> <variation code>`. The `<variation code>` combines a short code (`l` for `loop_length`, `ll` for `last_layer_channels`, `pl` for `penultimate_layer_channels` and `b` for `β`) and the corresponding value for the model architecture scaling factor.

6. Experiments and Results

Based on the literature review in Section 3, we identified MobileNet v1 [11], MobileNet v2 [12], MobileNet v3 [13], ShuffleNet v1 [14], and ShuffleNet v2 [15] as suitable candidate architectures to optimize for running on MCUs within our constraints. In total, we created 397 different model variations, of which 269 were trained and 63 were deployed to our MCU (STM32F746G-Disco board (https://www.st.com/en/evaluation-tools/32f746discovery.html, accessed on 4 July 2024) for inference time profiling (step 4). In the following subsections, we present examples describing our optimization strategy with MobileNet v1 [11]. The process is similar for other models.

6.1. Benchmark Model

First, we used the original MobileNet v1 [11] implementation code from the MLPerf Tiny Benchmark [19] repository to create a model variation with the benchmark’s scaling factor of `α = 0.25`, which scales the model width, and trained it once for 50 epochs on the VWW
dataset [18] with an image resolution of 96 × 96 × 3 pixels (mobilenetv1_0.25_96_c3_o2). This model, which serves as our benchmark model, has an int-8 quantized model size of 293.8 KB, uses 54.0 KB peak memory, requires 66.4 ms for inference on the MCU, and reaches 85.4% accuracy.

6.2. Optimization of MobileNet v1 in Detail

In the following subsections, we will provide an example describing our optimization strategy with MobileNet v1. We will first delineate the optimization with the default model architecture scaling factors, followed by the optimization by introducing new model architecture scaling factors.

6.2.1. Optimization with Default Model Architecture Scaling Factors \(\alpha\) and \(l\)

The MobileNet v1 model architecture consists of several stacked MobileNet blocks that replace the standard convolutional layers. The width multiplier \(\alpha\) uniformly thins each network layer by multiplying the number of output channels with \(\alpha\), if \(\alpha < 1\).

As the architecture repeats a MobileNet block with identical input and output dimensions five times, it is possible to vary this number of repetitions without breaking the architecture, which is displayed in Table 1. The authors of [11] experimented with this part of the model architecture for model optimization. Therefore, we implemented a model architecture scaling factor, which we name \(\text{loop\_length} (l)\) (default value: \(l = 5\)), for fine-tuning the architecture.

Table 1. MobileNet v1 architecture and optimizations.

<table>
<thead>
<tr>
<th>Input VWW (Our Use Case)</th>
<th>Operator</th>
<th>Scaling Factors</th>
<th>Benchmark (\alpha = 0.25)</th>
<th>Optim. 1 (\alpha = 0.25)</th>
<th>Optim. 2 (\alpha = 0.3)</th>
<th>Optim. 3 (\alpha = 0.7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channels:</td>
<td>Channels:</td>
<td>Channels:</td>
<td>Channels:</td>
<td>Channels:</td>
<td>Channels:</td>
<td>Channels:</td>
</tr>
<tr>
<td>96 × 96 × 3</td>
<td>conv2d 3 × 3</td>
<td>(\alpha)</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>22</td>
</tr>
<tr>
<td>48 × 48 × 32</td>
<td>mobilenet/s1</td>
<td>(\alpha)</td>
<td>16</td>
<td>16</td>
<td>19</td>
<td>44</td>
</tr>
<tr>
<td>48 × 48 × 64</td>
<td>mobilenet/s2</td>
<td>(\alpha)</td>
<td>32</td>
<td>32</td>
<td>38</td>
<td>89</td>
</tr>
<tr>
<td>24 × 24 × 128</td>
<td>mobilenet/s1</td>
<td>(\alpha)</td>
<td>32</td>
<td>32</td>
<td>38</td>
<td>89</td>
</tr>
<tr>
<td>24 × 24 × 128</td>
<td>mobilenet/s2</td>
<td>(\alpha)</td>
<td>64</td>
<td>64</td>
<td>76</td>
<td>179</td>
</tr>
<tr>
<td>12 × 12 × 256</td>
<td>mobilenet/s1</td>
<td>(\alpha)</td>
<td>64</td>
<td>64</td>
<td>76</td>
<td>179</td>
</tr>
<tr>
<td>12 × 12 × 256</td>
<td>mobilenet/s2</td>
<td>(\alpha \times \beta)</td>
<td>128</td>
<td>128</td>
<td>153</td>
<td>107</td>
</tr>
<tr>
<td>6 × 6 × 512</td>
<td>mobilenet/s1</td>
<td>(\alpha \times \beta)</td>
<td>128</td>
<td>128</td>
<td>153</td>
<td>107</td>
</tr>
<tr>
<td>6 × 6 × 512</td>
<td>mobilenet/s2</td>
<td>(\alpha \times \beta)</td>
<td>128</td>
<td>128</td>
<td>153</td>
<td>107</td>
</tr>
<tr>
<td>6 × 6 × 512</td>
<td>mobilenet/s1</td>
<td>(\alpha \times \beta)</td>
<td>128</td>
<td>128</td>
<td>153</td>
<td>107</td>
</tr>
<tr>
<td>3 × 3 × 1024</td>
<td>mobilenet/s1</td>
<td>(\text{pl})</td>
<td>256</td>
<td>256</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>3 × 3 × 1024</td>
<td>global avgpool</td>
<td></td>
<td>256</td>
<td>256</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>1024</td>
<td>dense (k)</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>k (k = 2)</td>
<td>softmax</td>
<td></td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Acc (%)</td>
<td></td>
<td></td>
<td>85.4</td>
<td>85.1</td>
<td>86.1</td>
<td>88.8</td>
</tr>
<tr>
<td>Model Size</td>
<td></td>
<td></td>
<td>293.8 KB</td>
<td>244.6 KB</td>
<td>243.4 KB</td>
<td>243.9 KB</td>
</tr>
<tr>
<td>&lt;250 KB</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
To understand the impact of the width multiplier $\alpha$ for small numbers, we created model variations with $\alpha \in \{0.1, 0.2, 0.25, 0.35\}$ and $l \in \{1, 2, 3, 4, 5\}$ and evaluated the impact of these model architecture scaling factors on the int-8 quantized model size and accuracy.

The model size in KB for the model variations with different $\alpha$ and $l$ are displayed in Figure 3, sorted by $\alpha$ and $l$. The figure consolidates the data stored in our CNN Analyzer. The horizontal line marks the model size constraint of 250 KB. It can be observed that the model size expands with increasing $\alpha$ and $l$. $\alpha$ significantly effects the model size, as more channels per layer require more model weights, which increases the storage requirements. The model with the highest accuracy (85.1%) that fits into our peak memory constraint is MobileNet v1, with $\alpha = 0.25$ and $l = 3$ (mobilenetv1_0.25_96_c3_02_l3).

Figure 3. MobileNet v1 model size in KB for different $\alpha$ and $l$.

Figure 4 shows the accuracy of our model variations. The horizontal line marks our 80% accuracy threshold. It demonstrates that the accuracy significantly decreases with decreasing $\alpha$. As our goal was to reduce model size while maintaining high accuracy, we looked for other methods to reduce the number of model parameters and introduced new additional model architecture scaling factors.

Figure 4. MobileNet v1 test accuracy for different $\alpha$ and $l$.

6.2.2. Layer-Wise Optimization with New Model Architecture Scaling Factors $pl$ and $ll$

Figure 5 shows (in blue) the layer-wise visualization of the number of model parameters in MobileNet v1 with $\alpha = 0.25$. It can be observed that the penultimate convolutional layer (consisting of 33 K parameters) and the last convolutional layer (consisting of 66 K parameters) are the biggest model parameter contributors, leading to a model size of 293.8 KB,
which exceeds our 250 KB constraint. The MobileNet v1 [11] architecture was designed for ImageNet [48] classification with 1000 classes, unlike our use case with only two classes. Therefore, we hypothesized that the model size could be significantly reduced by lowering the number of model parameters in the penultimate and last convolutional layers without incurring a significant negative impact on accuracy.

Figure 5. MobileNet v1: Benchmark vs. Optimization with \( pl \) and \( ll \).

To test this hypothesis, we introduced our two new model architecture scaling factors: \textit{penultimate\_layer\_channels} (\( pl \)) determines the number of channels in the penultimate convolutional layer, while \textit{last\_layer\_channels} (\( ll \)) specifies the number of channels in the last convolutional layer. We investigated the impact of varying these model architecture scaling factors. For our best MobileNet v1 variation (\textit{mobilenetv1\_0.3\_96\_c3\_o2\_l5ll32pl256}), the reduction of \( pl \) from 1,024 to 256 and \( ll \) from 1024 to 32 were optimal and decreased the number of model parameters significantly. As illustrated with the red bars in Figure 5, the number of model parameters in the penultimate convolutional layer dropped to 11.7k, while the number of model parameters in the last convolutional layer dropped to 0.7k. This reduced model size allowed us to increase the width multiplier \( \alpha \) to 0.3. The resulting best model variation uses width multiplier \( \alpha = 0.3 \), has a 17.2% decreased model size of 243.4 KB that fits the \( \leq 250 \) KB ROM constraint, and even shows 0.8% increased accuracy of 86.1% in comparison to the benchmark model.

6.2.3. Layer-Wise Optimization with New Model Architecture Scaling Factor \( \beta \)

As empirically shown in Section 6.2.1, a higher width multiplier \( \alpha \) is highly correlated with higher accuracy; thus, our design goal was to develop a model variation with the largest \( \alpha \) that could still fit into our 250 KB model size and 250 KB peak memory constraints.

The layer-wise visualization of model parameters in our CNN Analyzer, as shown in Figure 6 with the red bars, reveals that our best optimized MobileNet v1 [11] model variation \textit{mobilenetv1\_0.3\_96\_c3\_o2\_l5ll32pl256} still has a high number of model parameters in certain layers. The five convolutional layers that are repeated with the model architecture scaling factor \( l \) and the preceding layer are the layers with the most model parameters (the rectangle in Figure 6); therefore, we introduced our new model architecture scaling factor \( \beta \) to control the number of channels in these layers in proportion to the overall width multiplier \( \alpha \). Our new model architecture scaling factor \( \beta \) reduces the impact of these six layers on the overall model size, allowing us to further increase \( \alpha \) to 0.7 and thereby increase the model’s accuracy to 88.8%, using a model size of 243.9 KB and a peak memory of 148.5 KB. In total, we obtained a relative reduction in model size of 20.5% while increasing relative accuracy by 4.0%.
6.3. Summary of Benchmark MobileNet v1 Optimization

Table 1 provides a layer-by-layer summary of the optimal MobileNet v1 model variations that we obtained by inducing and optimizing additional model architecture scaling factors. The first two columns represent the input resolutions of each layer together with the operators leading to the resolution of the next layer. The third column shows the model architecture scaling factors that contributed to reductions in the number of channels in the corresponding layer. The fourth column displays the number of channels of the benchmark model from MLPerf Tiny Benchmark. The remaining columns show the optimizations, which are explained in detail in Section 6.2.

While our benchmark model (Benchmark) did not fulfill the constraint on model size of <250 KB, we were able to produce a model of 244.6 KB by retrieving optimal values for the default model architecture scaling factors width multiplier $\alpha$ and loop length $l$ (Optim. 1), however with poorer accuracy (85.1%) compared to Benchmark (85.4%). Looking at the channels (Channels), demonstrates that the model reduction was achieved by lowering the number of channel repetitions from five repetitions to three repetitions without breaking the architecture.

However, by introducing new model architecture scaling factors which reduce the channels in the penultimate layer ($pl$) and the last ($ll$) convolutional layer (Optim. 2), we were able to tackle the biggest model parameter contributors, which were located in the last two convolutional layers. This allowed us to fit the model variation within the 250 KB model size constraint even with $ll = 5$, leading to a higher accuracy of 86.1%.

With the help of CNN Analyzer’s visualizations of the number of model parameters in each layer, we were able to induce a new width multiplier $\beta$ and apply it to the six layers with the highest number of model parameters (Optim. 3). Using $\beta = 0.3$ allowed us to reach the 250 KB model size constraint despite significantly increasing $\alpha$ to 0.7. The best model architecture was achieved with $\alpha = 0.7$, $l = 5$, $pl = 64$, $ll = 32$, and $\beta = 0.3$, resulting in an accuracy of 88.8%.

6.4. Leveraging Visualizations to Find Optimal Model Architecture Scaling Factors

During our experiments, we observed that in order to achieve high accuracy it is important to choose a large width multiplier $\alpha$, as shown in Figure 4. However, this leads to a higher number of model parameters, which increases the model size, as demonstrated in Figure 3. As the relationships between the width multiplier $\alpha$, the number of model parameters, and the model size are not linear, the magnitude of increasing $\alpha$ is not intuitive. For example, slightly increasing $\alpha$ for MobileNet v1 from 0.2 to 0.25 increases the model size by 42%, from 207.5 KB to 294.2 KB.

For MobileNet v1 with $\alpha = 0.25$, our layer-wise visualizations showed a peak in model parameters in the penultimate layer (consisting of 33K parameters) and the last convolutional layer (consisting of 66K parameters) (see Figure 5). These two layers contribute 45%
of the 222K parameters of the model variation. Without the layer-wise visualization of our CNN Analyzer, the introduction of new model architecture scaling factors to control these layers would not have been possible.

Since the CNN Analyzer displays the number of model parameters, model size, bytes/parameter ratio, and layer-wise visualizations of channels and model parameters side-by-side, the user can derive ideas on how to optimize specific scaling factor values. These visualizations are even more important when several model architecture scaling factors influence the same layer and the model parameter distribution shifts.

6.5. Optimizations of Further Models

We used the same strategy for the other model architectures and adapted it for their specific model architecture scaling factors.

6.5.1. MobileNet v2

MobileNet v2 [12] also provides a width multiplier $\alpha$, which we varied in our experiments. Additionally, we exposed the expansion factor $t$, which scales the number of channels inside the bottleneck block, as a model architecture scaling factor ($t \in [1, 6]$, default value $t = 6$). In the default implementation, $\alpha$ does not scale the last convolutional layer with 1,280 channels. Consequently, we also introduced our new model architecture scaling factor $last\_layer\_channels$ to control and significantly reduce the number of model parameters in this layer.

Since the architecture contains only one convolutional layer after the bottleneck blocks, we could not introduce $penultimate\_layer\_channels$ for the MobileNet v2 architecture.

The best MobileNet v2 model variation (mobilenetv2_0.25_96_c3_o2_15l256) uses $\alpha = 0.25$, $t = 5$, $last\_layer\_channels = 256$, has an int-8 quantized model size of 248.0 KB, uses 56.3 KB peak memory, requires 59.5 ms inference time on the MCU, and reaches an accuracy of 84.1%, which is below the accuracy of the benchmark model.

6.5.2. MobileNet v3


The best architecture within our constraints uses $\alpha = 0.05$, has an int-8 quantized model size of 197.1 KB, peak memory of 75.3 KB, 41.7 ms inference time on the MCU, and reaches 83.5% accuracy, which is below the accuracy of our benchmark model.

Since model variations with higher width multipliers $\alpha$ exceeded our peak memory constraint of 250 KB, we used the same approach as [18], who removed the squeeze-and-excitation modules inside the MobileNet v3 architecture to lower the peak memory. The model variations without the squeeze-and-excitation module are indicated by the suffix NSQ (“no squeeze”). We also introduced our new model architecture scaling factors $penultimate\_layer\_channels (pl)$ and $last\_layer\_channels (ll)$ to significantly reduce the number of model parameters in these layers.

Our best MobileNet v3 model variation without the squeeze-and-excitation module is mobilenetv3smallNSQ_0.3_96_c3_o2_l32pl128. It uses $\alpha = 0.3$, $penultimate\_layer\_channels = 128$, $last\_layer\_channels = 32$, has an int-8 quantized model size of 172.8 KB, uses a peak memory of 110.6 KB, requires 118.8 ms inference time on the MCU, and reaches an accuracy of 86.1%, slightly outperforming our benchmark model’s accuracy of 85.4%.

6.5.3. ShuffleNet v1

The ShuffleNet v1 [14] architecture uses pointwise group convolutions instead of costly $1 \times 1$ convolutions to reduce computational cost while maintaining accuracy.

The model can be scaled by controlling the number of groups in the pointwise convolutions with the ShuffleNet-specific default model architecture scaling factor $g \in \{1, 2, 3, 4, 8\}$, which controls the connection sparsity, and a ShuffleNet-specific default model architecture scaling factor $a \in \{0.25, 0.5, 1, 1.5\}$, which scales the number of channels per
layer. Since the number of filters in each shuffle unit block must be divisible by g, only a limited number of valid model variations can be created.

Due to architectural constraints and the downsampling strategy of ShuffleNet v1, we could not introduce new model architecture scaling factors to further optimize the model.

The best model variation of ShuffleNet v1 (shufflenetv1_0.25_96_c3_o2_g1) with \( \alpha = 0.25 \) and \( g = 1 \) has an int-8 quantized model size of 175.2 KB, 81 KB of peak memory, 69.6 ms inference time on our MCU, and 85.1% accuracy, which is below the accuracy of our benchmark model.

6.5.4. ShuffleNet v2

In ShuffleNet v2 [15], the number of channels \( c \) in the first ShuffleNet v2 block is controlled by the ShuffleNet-specific default model architecture scaling factor \( \alpha \in \{0.5, 1, 1.5, 2\} \). We extended the range of \( \alpha \) to also include \( \alpha \in \{0.05, 0.1, 0.2, 0.25, 0.35\} \). It is important to take into account that the number of output channels of the first block must be an even number in order to allow for the channel split operation.

To further optimize the ShuffleNet v2 architecture, we introduced our new model architecture scaling factor last_layer_channels to significantly reduce the model parameters in this layer. Since the architecture contains only one convolutional layer after the ShuffleNet blocks, we could not introduce penultimate_layer_channels for the ShuffleNet v2 architecture.

Our best ShuffleNet v2 model variation (shufflenetv2_0.1_96_c3_o2_l128) with \( \alpha = 0.1 \) and last_layer_channels = 128 achieved 83.3% accuracy using 78.8 KB of peak memory and had a model size of 167.8 KB. This optimized architecture does not reach the accuracy of our benchmark model.

6.5.5. Summary of Model Optimizations

Table 2 lists the best results of our five examined model architectures MobileNet v1, MobileNet v2, MobileNet v3, ShuffleNet v1, and ShuffleNet v2, plus the MLPerf Tiny Benchmark [19] inference model (mobilenetv1_0.25_96_c3_o2), sorted by accuracy. Using our strategy and the CNN Analyzer, we were able to obtain two models that significantly outperformed the MLPerf Tiny inference benchmark model. The MobileNet v1 model architecture with variation mobilenetv1_0.7_96_c3_o2_l5ll16pl32b0.25 outperformed all other evaluated architectures for our model constraints.

<table>
<thead>
<tr>
<th>Model</th>
<th>Acc. (%)</th>
<th>Model Size (KB)</th>
<th>Peak Memory (KB)</th>
<th>Inference on MCU (ms)</th>
<th>MACs</th>
<th>Params</th>
<th>Bytes/Param</th>
</tr>
</thead>
<tbody>
<tr>
<td>mobilenetv1_0.7_96_c3_o2_l5ll16pl32b0.3</td>
<td>88.8</td>
<td>243.9</td>
<td>148.5</td>
<td>181.7</td>
<td>21,893,563</td>
<td>171,743</td>
<td>1.454</td>
</tr>
<tr>
<td>mobilenetv3smallNSQ_0.3_96_c3_o2_l32pl128</td>
<td>86.1</td>
<td>172.8</td>
<td>110.6</td>
<td>118.8</td>
<td>6,191,720</td>
<td>78,664</td>
<td>2.249</td>
</tr>
<tr>
<td>mobilenetv1_0.25_96_c3_o2 (benchmark)</td>
<td>85.4</td>
<td>293.8</td>
<td>54.0</td>
<td>66.4</td>
<td>7,489,664</td>
<td>221,794</td>
<td>1.356</td>
</tr>
<tr>
<td>shufflenetv1_0.25_96_c3_o2_g1</td>
<td>85.1</td>
<td>175.2</td>
<td>81.0</td>
<td>69.6</td>
<td>3,184,560</td>
<td>71,030</td>
<td>2.526</td>
</tr>
<tr>
<td>mobilenetv2_0.25_96_c3_o2_l5l256</td>
<td>84.1</td>
<td>248.0</td>
<td>56.3</td>
<td>59.5</td>
<td>3,886,352</td>
<td>138,366</td>
<td>1.835</td>
</tr>
<tr>
<td>shufflenetv2_0.1_96_c3_o2_l128</td>
<td>83.3</td>
<td>167.8</td>
<td>78.8</td>
<td>57.4</td>
<td>2,741,080</td>
<td>56,058</td>
<td>3.065</td>
</tr>
</tbody>
</table>

All models were developed and employed using the following downscaling and optimization processes to optimize our candidate CNN model architecture:
• Build model variations with different width multipliers $\alpha$ and check the model size and peak memory. Find a model variation where only one of those constraints is not met.
• If the peak memory constraint is not met, choose a smaller width multiplier $\alpha$.
• If the model size requirement is not met, create a layer-wise visualization of the model parameters and identify the layers with the most model parameters.
• Reduce the number of channels in the layers that have the most model parameters.
• Finally, try to increase the width multiplier $\alpha$ as much as possible while keeping the model variation within the constraints.

7. Conclusions and Future Work

Our research focused on optimizing CNN architectures for MCUs by systematically scaling down the architectures with (1) existing model architecture scaling factors and (2) new model architecture scaling factors, which we induced with the help of our optimization strategy and our developed CNN Analyzer. Our experiments revealed that using the original default model architecture scaling factors leads to suboptimal results when significantly scaling down models, as this approach is too coarse and detrimental to accuracy. Our research also considered the actual model size, which accounts for the overhead required to store the model architecture. Furthermore, to estimate the number of model parameters needed to fit a model within the given constraints, we introduced the bytes/parameter ratio as a new evaluation metric.

By applying our model optimization strategy, we successfully enhanced the performance of established efficient architectures such as MobileNet v1 [11], MobileNet v2 [12], MobileNet v3 [13], and ShuffleNet v2 [15]. Our model variations outperformed the benchmark model from the MLPerf Tiny Benchmark [19], reducing the relative model size by 20.5% while increasing relative accuracy by 4.0%. The CNN Analyzer and its related code are available on our GitHub repository, allowing the research community to further develop and improve CNN model optimization for resource-constrained MCUs. While we applied CNN Analyzer for a specific MCU use case where extreme constraints had to be met, it is generally applicable for scenarios with less strict constraints as well, e.g., microprocessor units that require the adaptation of CNN model architectures to hardware constraints.

For future work, we suggest increasing the accuracy of the best model variations through knowledge distillation [31]. Since the VWW training set consists of less than 100,000 images, we recommend pretraining the model variations on a larger dataset, then fine-tuning the model variations on the VWW dataset. Additionally, the best model variations can be trained for other binary classification tasks with a resolution of $96 \times 96 \times 3$ pixels. Our findings and developed tools are portable to other network architectures, and can be combined with state-of-the-art NAS approaches.

Author Contributions: Conceptualization, methodology, software, validation, resources, writing, visualization: S.B. and T.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: We have released our CNN Analyzer and all related code open-source on our GitHub repository, empowering the research community to explore and build upon our work and fostering further advancements in the field of CNN model optimization for resource-constrained MCUs: https://github.com/subrockmann/tiny_cnn (accessed on 1 July 2024).

Conflicts of Interest: The authors declare no conflicts of interest.

References


34. Lawrence, T.; Zhang, L. IoTNet: An Efficient and Accurate Convolutional Neural Network for IoT Devices. *Sensors* 2019, 19, 5541. [CrossRef]


47. Liberis, E.; Dudziak, L.; Lane, N.D. μNAS: Constrained Neural Architecture Search for Microcontrollers. In Proceedings of the 1st Workshop on Machine Learning and Systems, New York, NY, USA, 26 April 2021; EuroMLSys ’21; pp. 70–79. [CrossRef]


**Disclaimer/Publisher’s Note:** The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.