OpTC – A Toolchain for Deployment of Neural Networks on AURIX TC3xx Microcontrollers

Christian Heidorn[1], Frank Hannig[1], Dominik Riedelbauch[2],
Christoph Strohmeyer[2], and Jürgen Teich[1]

[1] Department of Computer Science,
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany
christian.heidorn@fau.de
[2] Schaeffler Technologies AG & Co. KG, Herzogenaurach, Germany

Abstract. The AURIX 2xx and 3xx families of TriCore microcontrollers are widely used in the automotive industry and, recently, also in applications that involve machine learning tasks. Yet, these applications are mainly engineered manually, and only little tool support exists for bringing neural networks to TriCore microcontrollers. Thus, we propose OpTC, an end-to-end toolchain for automatic compression, conversion, code generation, and deployment of neural networks on TC3xx microcontrollers. OpTC supports various types of neural networks and provides compression using layer-wise pruning based on sensitivity analysis for a given neural network. The flexibility in supporting different types of neural networks, such as multi-layer perceptrons (MLP), convolutional neural networks (CNN), and recurrent neural networks (RNN), is shown in case studies for a TC387 microcontroller. Automotive applications for predicting the temperature in electric motors and detecting anomalies are thereby used to demonstrate the effectiveness and the wide range of applications supported by OpTC.

Keywords: AURIX TriCore, Neural Networks, Pruning

1 Introduction

With the *Internet of Things* (IoT), the demand for deploying neural networks (NNs) on resource-constrained devices such as microcontrollers continues to grow. For example, neural networks can be used in electric vehicles to predict battery charge [1] or implement thermal management of the electric motor [2]. In recent years, several mature end-to-end ML frameworks (e.g., TensorFlow [3], PyTorch [4], Keras [5]) have emerged centered around GPUs as the workhorse. As a consequence, many neural network models come with high memory requirements and high computational complexity. This severely hampers their deployment on microcontrollers. Thus, compression must be used to reduce the memory footprint and speed up the computation, which typically comes at the cost of the neural network model's reduced prediction quality (e.g., accuracy, area under the curve).

A common compression technique is sparsification (a.k.a pruning [6]), which removes parts of the neural network to reduce computational complexity and, correspondingly, inference time. During the development of a neural network model, a programmer or data scientist is typically unaware of whether the implementation of the model will meet given execution time and memory constraints on some microcontroller target. There is a vast design space, especially when it comes to model compression. For example, when considering layer-wise pruning, thousands of possible combinations of pruning rates for the different layers with trainable weights exist. Typically, this leads to time-consuming trial and error, as the model must be compressed, re-trained, and deployed multiple times on the target device until finally meeting time and memory constraints on the target hardware while still achieving a high prediction quality.

In this work, we present a toolchain that paves the way for automated model compression and code generation to mitigate this manual trial-and-error process. It automatically returns pruned neural network models with trade-offs between memory footprint and execution time vs. prediction quality for highly resource-constrained microcontroller targets used in automotive applications, particularly for the AURIX TriCore 3xx family.

Our main contributions are:

- OpTC, a modular toolchain for deploying various types of neural networks (e.g., MLPs, CNNs, RNNs) on AURIX TriCore microcontrollers. Several optimizations are performed automatically, including operator fusion and model compression.
- Reduction of the vast design space by a *sensitivity analysis* [6] and exploration of the different design points (neural network compression rates) to identify trade-offs between execution time, memory requirements, and prediction quality.
- Evaluation of the toolchain for applications from the MLPerf Tiny benchmark [7] and a dataset for predicting the temperature in an electric motor [8] and showcasing OpTC when exploring these applications in terms of memory footprint (RAM and ROM), execution time, and prediction quality on an AURIX TriCore 387 microcontroller.

The remainder of this paper is organized as follows: Section 2 provides fundamentals on neural network pruning, Section 3 discusses related work, and Section 4 presents the novel toolchain approach. Section 5 introduces the case study, as well as the dataset and models, which are used in the experiments (Section 6) before Section 7 concludes.

2 Fundamentals

Neural networks consist of multiple layers of different types and can be represented as data flow graphs. NNs have trainable layers with weight tensors, such as fully connected and convolutional layers. Typically, non-linear activation functions are applied after each layer, e.g., *rectified linear unit* (ReLU), *hyperbolic tangent* (tanh), or *sigmoid* functions. For efficient deployment of neural networks on microcontrollers, pruning [9] has emerged as one of the most common compression techniques, which is explained below.

2.1 Pruning

Pruning reduces the number of neurons and their connections, thereby also reducing the number of weights and the number of floating point operations (FLOPs). *Structural pruning* is a technique where entire structures, e.g., output neurons, are set to zero and can be removed from computation [10]. For example, entire filters can be removed from convolutional layers, or rows and columns of the weight matrix can be eliminated in fully connected layers. This approach can effectively decrease execution time by reducing the number of loop iterations required to process the layers. Let a neural network be given that consists amongst other layers (e.g., activation and pooling layers) of a set V of layers with trainable weights. In the case of *global pruning*, a pruning rate p is defined with $0 \leq p < 1,\ p \in \mathbb{R}$. Further, let M_i denote the number of output neurons (or filters) of each corresponding layer $v_i \in V$. Then, the resulting number m_i of output neurons after pruning layer $v_i \in V$ is

$$m_i = \lceil M_i \cdot (1-p) \rceil. \tag{1}$$

In the case of *layer-wise pruning*, each layer v_i is assigned a pruning rate p_i where $0 \leq p_i < 1,\ p_i \in \mathbb{R}$. Typically, layer-wise pruning results in higher compression than global pruning for achieving a similar prediction quality. Here, the resulting number m_i of output neurons after pruning layer $v_i \in V$ is

$$m_i = \lceil M_i \cdot (1-p_i) \rceil. \tag{2}$$

The higher the pruning rate (p or p_i), the fewer weights have to be stored, and the fewer FLOPs are required. There exist several heuristics to determine which weights or weight structures should be removed. The most commonly used techniques are the ℓ^1 or ℓ^2 norm of the weights [11]. Here, the filters (or rows and columns) with the lowest ℓ^1 norm values are set to zero and removed.

2.2 Design Space

When applying global pruning to a given neural network model, the size of the design space Ω_{global} of all possible pruning options is the maximal number of output neurons and filters M_i, which results in $|\Omega_{\text{global}}| = \max_{i=0}^{|V|-1} M_i$. In the case of layer-wise pruning of a given neural network model, the size of the design space $\Omega_{\text{layer\_wise}}$ of all possible pruning combinations is thus $|\Omega_{\text{layer\_wise}}| = \prod_{i=0}^{|V|-1} M_i$. However, for many neural networks, this design space is excessively large, so it would take a prohibitively long time to explore and evaluate all design points. For example, consider the *Autoencoder* model for anomaly detection from the MLPerf Tiny benchmark [12]. The Autoencoder contains ten fully connected layers ($|V| = 10$), where the number of output neurons ranges from $M_i = 8$ to 128; then, the number of possible pruned configurations is nearly $|\Omega_{\text{layer\_wise}}| = 10^{20}$. Assessing the prediction quality and inference time for all configurations is not feasible since evaluating one pruned AE model requires in our experiments approximately 40 seconds (see Section 6.1). The evaluation time may even increase further if retraining is performed for each pruned configuration, which might be necessary to maintain the prediction quality of a pruned neural network.

3 Related Work

Conventional techniques for neural network deployment often ignore tight computational and memory constraints, which hinders their application to highly resource-constrained microcontrollers. Therefore, a dedicated approach is required when developing and compiling neural networks for resource-constrained microcontroller targets. It has to ensure that the computational and latency budget is within the device limits while still achieving a desired performance [13]. Recently, approaches from industry and academia have emerged that differ in their support of neural network types, input formats, and support of microcontrollers as summarized in the following.

3.1 Workflows for Deploying Neural Networks on Microcontrollers

Software development for microcontrollers is typically based on C or C++ programming. Thus, some ML workflows already come with an inference library [10, 14, 15, 16] developed in C or C++. Here, operator function calls required to compute the neural network layers are baked into C or C++ during code generation. Typically, those lightweight inference libraries are implemented in C or C++ combined with a conversion workflow that generates library function calls for a neural network. These workflows are often realized directly within popular machine learning (ML) frameworks, e.g., TensorFlow [3] or PyTorch [4], or relying on standard exchange format descriptions, such as ONNX (Open Neural Network Exchange, [17]) to ensure compatibility. Typically, the inference library is intended to be sufficiently generic to be used on any 32-bit microcontroller and, therefore, portable. Workflows such as TensorFlow Lite Micro [15] and microTVM [18] have drawbacks as they rely on an interpreter to execute the network graph at runtime. This adds memory and latency overhead [14]. In addition, optimizations are performed merely at the layer level of a neural network, which misses the potential of globally optimizing the overall neural network graph, e.g., by layer fusion [19, 20]. Another drawback of inference libraries is the effort required to integrate them into other standardized automotive architectures, such as AUTOSAR (Automotive Open System Architecture) [21], a software architecture for automotive systems, especially when managing multiple dependencies and ensuring compatibility with other parts. Within such automotive frameworks, the use of external libraries and hardware platforms (e.g., AURIX TriCore) is often restricted, necessitating a complete redesign of the inference library and workflow.

NNOM [16], TFLite Micro [15], and DNNruntime [10] are examples of generic inference libraries capable of generating ANSI C code. TFLite Micro and DNNruntime also support the floating point format. These libraries consist of C code that is compilable for TriCores. However, these libraries do not consider target-specific optimizations for layers, operators, and parallel execution on multiple cores. Implementing such target-specific optimizations (e.g., for TriCore TC3xx) would be effortful, and core-specific C code generation may be required. One would have to strongly alter the inference libraries and add the C code for each missing operator (or layer), and usually, the code generation (i.e., for the function calls) has to be refined correspondingly. An additional disadvantage of this approach comes when considering different optimizations. For example, if layer fusion shall be supported, for each combination of layers that can be fused, C code must be added to the inference library. As a consequence, this can add a lot of memory overhead on the target microcontroller.

Dory [22] comes without an inference library and generates C code for a given model that does not require linking to an inference library. Generating C code gives the possibility of seamless integration into other projects and provides the ability to fine-tune memory management for the target platform. For example, one can optimize memory usage based on the specific constraints of the target microcontroller and extend code generation for the target-specific memory hierarchy. However, Dory is dedicated to RISC-V targets and has a defined memory hierarchy. In addition, when it comes to model compression, Dory relies on having quantized neural networks as inputs and only supports integer (int) precision. However, we argue that on platforms such as AURIX embedded microcontrollers, which provide single-precision floating-point computation, it can still be used, especially for complex activation functions (e.g., *hyperbolic tangent*), or to support mixed-precision models that are known to maintain high prediction quality.

Finally, there is a C inference library available from Ekkono[3] for deploying neural network models on AURIX TriCore 2xx and 3xx microcontrollers. However, it appears that there is only a C++ inference library provided and no neural network compression is integrated into the workflow.

3.2 Integration of Compression Techniques into Workflows

Typically, for compressing a given neural network, there is some trial and error phase, where, e.g., a user defines a pruning rate and measures the resulting prediction quality of the neural network on the test dataset or a subset of the test dataset. If the model's prediction or the compression is not sufficient, this process has to be repeated several times. Some of the workflows listed in Section 3.1 already integrate compression techniques. For example, TFLite Micro is based on Tensorflow and Keras and provides quantization and weight clustering techniques. However, for workflows such as TFLite Micro optimization options have to be set manually and the use of different techniques is only possible to a very limited extent [23]. This is a major shortcoming, as the developer will only find out if the developed neural network meets the constraints of the target hardware after deployment. This may result in a large time overhead as the neural network has to be tuned, trained, and recompiled.

MCUNet [14] uses neural architecture search (NAS) to find networks that meet target platform constraints. If a model that meets the target platform constraints is found, the user can be sure that this model is deployable on the platform. However, the models have to be trained from scratch, which is time-consuming, especially when considering, that more models have to be trained in parallel, or one huge so-called "supernet" has to be trained to identify subnets afterward. Typically, there exist expert-designed pre-trained neural network models that already perform well on given datasets [12]. In this case, such a compute- and time-expensive search strategy should be avoided.

DNNruntime scales down already defined neural networks using pruning and quantization until the memory requirements are met. The advantage here is that existing trained neural networks can be used for deployment on the target microcontroller. However, after compression, the model's prediction quality might be reduced. Again, the model has to be redesigned and trained, which is time-consuming. Especially in the case of *iterative pruning*, when retraining is applied after each pruning step, this process

[3] Ekkono, "From Connected To Smart", https://www.ekkono.ai/, Date accessed: 03/19/2024

becomes time-consuming, especially for neural nets that have multiple layers. Here, our toolchain proposed in the following determines pruning rates automatically based on sensitivity analysis and can reduce the vast design space to only a few evaluations (which can be decided by the user).

4 The OpTC Toolchain

According to Fig. 1, a sensitivity analysis is performed (Algorithm 1). Based on the pruning sensitivities of each layer v_i, initial pruning rates p_i^{init} are determined. Afterward, the design space is constrained by setting an upper bound $J \in \mathbb{N}$ of increasingly pruned versions of a given network model X. For each pruned model Y_j, the prediction quality on the test dataset, the execution time, and memory requirements when deployed on the target microcontroller are determined. The prediction quality, execution time, and memory requirements are collected, and Pareto-optimal solutions are finally output. Our approach integrates a fully automated loop consisting of neural network (NN) compression (Section 4.1), code generation (Section 4.2), flashing, and returning the measured memory utilization and execution time for the target architecture into one toolchain named OpTC. It is thus no longer necessary for developers to perform this cycle manually and search for the optimal tradeoff between pruning rates and NN prediction quality in a trial-and-error fashion. The toolchain supports various Python frameworks for specifying neural networks, including PyTorch and TensorFlow by relying on the ONNX standard [17]. ONNX is an exchange format describing neural net-

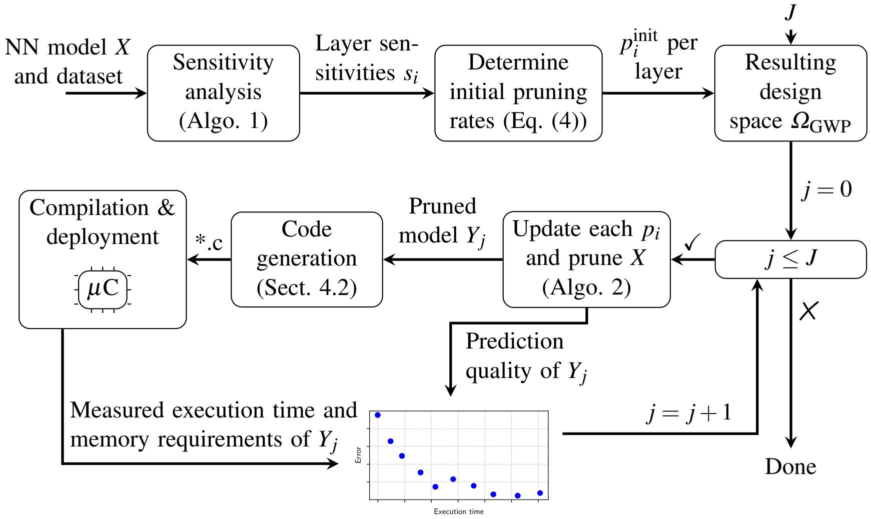

NN model X and dataset
Sensitivity analysis (Algo. 1)
Layer sensitivities s_i
Determine initial pruning rates (Eq. (4))
p_i^{init} per layer
J
Resulting design space Ω_{GWP}
$j = 0$
$j \leq J$
✓
✗
Update each p_i and prune X (Algo. 2)
Pruned model Y_j
Code generation (Sect. 4.2)
*.c
Compilation & deployment
μC
Prediction quality of Y_j
Measured execution time and memory requirements of Y_j
Error
Execution time
$j = j + 1$
Done

Fig. 1: Overview of our proposed approach to automatic compression by exploration of network configurations by iteratively increasing the degree of weighted global pruning of a neural network X for deployment on the target microcontroller (μC).

works as dataflow graphs, where each node represents a mathematical operation (e.g., element-wise addition or matrix multiplication). OpTC encompasses a template-based code generator that translates pre-trained neural networks stored in ONNX format to C code exploiting the static properties of trained neural networks (i.e., fixed layer configurations and parameters), such that no dependence on an inference library is required. For conversion from pre-trained Pytorch or TensorFlow neural networks to ONNX, the toolchain integrates the open-source conversion tools, tf2onnx, and the PyTorch-integrated ONNX exporter. This not only reduces the computational overhead at runtime but also allows for seamless integration into other projects or frameworks. In the following, the steps and algorithms shown in Fig. 1 are described in detail.

4.1 Automatic Compression

Typically, the workflow starts as follows: A user provides the model and the dataset as well as a search space for the pruning configuration as input. Each model is assumed to be pre-trained by an ML expert. For achieving high flexibility, different Python libraries, such as Microsoft Neural Network Intelligence (NNI)[4], can be integrated to compress the neural network. This provides different options for pruning a neural network X, resulting in a set of J pruned neural network configurations, with different trade-offs between prediction quality and execution time.

Sensitivity Analysis: Previous works [6, 24, 25] have shown that a layer could be differently important for the prediction quality of the overall neural network. We perform a sensitivity analysis (Algorithm 1) to determine the maximal pruning rates $p_i^{\max}$ for each layer $v_i \in V$ in a given neural network model X.

Algorithm 1 Sensitivity analysis

```
Input:  NN model $X$ with set $V$ of trained layers, test dataset, ascending sequence of
        pruning rates $P = [p^{\min}, \ldots, p^{\max}]$, threshold $T$
Output: Layer sensitivities $S$, i.e., $s_i \in S$ for each layer $v_i$
  for $v_i \in V$ do
      for $p \leftarrow p^{\min}$ to $p^{\max}$ do
          $Y \leftarrow$ prune layer $v_i$ of model $X$ with pruning rate $p$
          $a \leftarrow$ obtain prediction quality of pruned model $Y$ by evaluating it with given test dataset
          if $a < T$ then
              $s_i = 1 - p$                    ▷ maximal pruning rate $p_i^{\max}$ of layer $v_i$ found
              break
          end if
      end for
  end for
```

For analysis of sensitivity, we define a sequence of pruning rates, e.g., $P = [0.1, 0.2, \ldots, 0.9]$. For each layer v_i, the pruning rate $p \in P$ is increased while the other layers stay unpruned and the prediction quality of the overall neural network is measured. The

[4] Microsoft, “Neural Network Intelligence (NNI)”, https://github.com/microsoft/nni

sensitivity analysis usually requires just iterating over the test dataset without retraining. If the prediction quality a of the partially pruned neural network model drops below a given threshold T (e.g., the accuracy of the unpruned model), the pruning rate applied at this point defines the maximum pruning rate $p_i^{\max}$ for layer v_i[5]. The layer sensitivity s_i is then defined as

$$s_i = 1 - p_i^{\max}. \tag{3}$$

Global Weighted Pruning (GWP): For a given layer v_i, the initial pruning rate p_i^{init} is determined based on the layer sensitivity s_i and by introducing a positive integer number J called steps, which defines how many differently pruned network configurations to be evaluated:

$$p_i^{\text{init}} = \frac{1 - s_i}{J} = \frac{p_i^{\max}}{J} \qquad \forall i\ :\ v_i \in V \tag{4}$$

Note that if $s_i = 1$, p_i^{init} turns to zero, such that the respective layer v_i will stay unpruned. We call this technique *global weighted pruning*, short GWP, where each layer $v_i \in V$ gets assigned the initial pruning rate p_i^{init}, which is incremented in each iteration j, $0 \leq j \leq J$ that is input to Algorithm 2.

Algorithm 2 Global Weighted Pruning

```
Input:  NN model $X$ with set $V$ of trained layers, initial pruning rate $p_i^{\text{init}}$ per layer $v_i$,
        iteration $j$ with $0 \leq j \leq J$
Output: pruned model $Y_j$
  for $i \leftarrow 0$ to $|V| - 1$ do
      $p_i = p_i^{\text{init}} \cdot j$                         ▷ Determine pruning rate $p_i$ of layer $v_i$
  end for
  $Y_j \leftarrow$ compress model $X$ by pruning each layer $v_i$ with pruning rate $p_i$
```

By definition, $j = 0$ denotes the initial (unpruned) neural network model X. Compared to [11], where a global step size for each layer was defined, p_i^{init} can also be interpreted as a layer-specific step size, which is kept constant, and the pruning rate p_i is increased in each step j by p_i^{init}. The number of remaining filters m_i after pruning for each iteration j is obtained as

$$m_i = \lceil M_i \cdot (1 - p_i^{\text{init}} \cdot j) \rceil. \tag{5}$$

The design space is, therefore, of size $|\Omega_{\text{GWP}}| = J$, where J can be freely chosen.

4.2 Template-based C Code Generation

One way to compile an ONNX graph into an executable is to translate each layer directly into C code. However, it is beneficial to optimize the graph itself, e.g., by fusing the operation nodes of the graph, as C or C++ compilers typically do not perform

[5] Note that for some datasets, the prediction quality of a neural network is determined by measuring an error (e.g., the mean squared error). In this case, $a < T$ is replaced by $a > T$ in Algorithm 1, where a is the prediction error of the pruned model, and $p_i^{\max}$ is determined if the error is above the threshold error T of the unpruned model.

these optimizations [26]. Our code generator [27] includes optimization techniques for merging activation functions into preceding convolutional or matrix multiplication operations. Another example of graph optimization is the transformation of a matrix-matrix multiplication followed by a vector addition into a general matrix multiplication (GEMM), which is often required for multi-layer perceptrons, as it seems that the built-in exporters in PyTorch often model a linear layer as two separate operations (matrix-matrix multiplication with the weights, and vector addition on the resulting output by a bias). In addition, some nodes can be removed from the computational graph of a neural network (e.g., zero-padding). Currently, OpTC supports convolutions, linear transformations, pooling operations, and activation functions. It also supports complex operators such as long short-term memory cells and applies approximations to activation functions. Internally, a Python-based intermediate representation (IR) is used, which describes the neural network operators and their interactions (dataflow). During code generation, the IR is traversed in a valid order, and for each operator (e.g., GEMM), a corresponding predefined, parameterized C template is instantiated and code emitted. OpTC applies code optimizations (e.g., operator or loop fusion) to reduce execution time and memory requirements. The toolchain recognizes patterns (e.g., convolution followed by ReLU), fuses them into functionally equivalent operators, and emits fused C loops, which are beneficial concerning performance. For the intermediate tensors, tensor unionization [27] is performed to wrap the tensors into unions to help the compiler reuse heap memory.

5 Benchmarks

Neural networks have already proven their value for classification tasks (e.g., in image processing), and many works to compress them have been proposed. However, in the case of detecting anomalies or thermal monitoring, the neural networks are typically designed to solve a regression problem, minimizing a loss function, e.g., the Mean Squared Error between the predicted value and the ground truth.

In our experiments (Section 6), we show the effects of compression for different benchmark neural networks different benchmarks and also demonstrate that our toolchain supports a broad range of different models. For benchmarking, we use datasets and models provided by the *MLPerf Tiny* benchmark suite, an open-source benchmark suite for TinyML systems [12], and a publicly available *Electric Motor Temperature* (EMT) dataset from [8], consisting of temperature sequence data at different positions in an electric motor taken from a testbench. The used datasets and neural network models are described below.

5.1 Autoencoder (AE) for Anomaly Detection

Analogously to MLPerf Tiny, we use the “toy car” sub-dataset from the ToyADMOS dataset [28]. For training, sound recordings of seven different regularly operating toy cars are provided, each consisting of 1000 audio samples mixing the machine sound with environmental noise. The baseline architecture is an Autoencoder model consisting of an encoder (four fully connected layers) and a decoder (five fully connected layers).

Each part comprises fully connected layers (128 neurons) with ReLU activation. The encoder and decoder are connected by a so-called bottleneck layer, which is also a fully connected layer with eight neurons. The input and output layers of the model each have 640 neurons, respectively. The model itself is not applied directly to the 10-second audio data. The audio data is preprocessed into a log-mel spectrogram with 128 bands and a frame size of 32 ms. Then, the model is used repeatedly over a sliding window of five frames (hence the 640 input size), and the MSE of the resulting reconstruction error is averaged over the central 6.4 second part of the spectrogram, providing an overall anomaly score [12].

5.2 Convolutional Neural Network (CNN) on Keyword Spotting

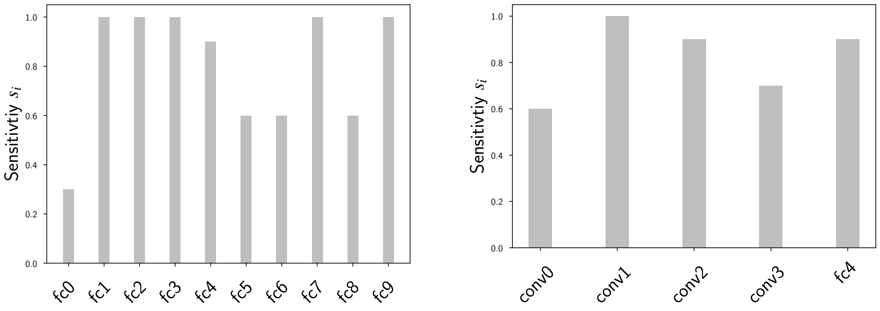
Recognition of specific words and brief phrases, known as keyword spotting, is one of the primary use cases of ultra-low-power machine learning. Wake word detection is a specific case of keyword spotting wherein a detector continuously monitors for a specific word or phrase to enable a larger processor. It requires continuous operation; therefore, low power consumption is key. For this benchmark, we use the Speech Commands v2 dataset [29]. The dataset contains 30 words and a collection of background noises and is divided into training, validation, and test subsets such that any individual speaker only appears in one subset. For the model, we use the CNN model described in [30] for processing raw audio data. The CNN consists of five 1-dimensional convolutional layers, where the trained filters are convolved over the time dimension. After each convolutional layer, there is a pooling layer and a ReLU activation layer. One fully connected layer at the end outputs a vector, where each element represents the probability for the respective class. The CNN is trained on a subset of words for 12 output classes [12]. To evaluate the accuracy, we randomly selected 1000 utterances from the speech commands for the test dataset.

5.3 Temporal Convolutional Neural Network (TCN) on Electric Motor Temperature

Temperature estimation tasks are necessary for electric drives in automotive and industrial automation applications. As a motivating example, having fast and accurate estimations for the rotor temperature helps to manufacture motors with fewer sensors while still enabling control strategies to utilize the motor to its maximum capability. For the training and evaluation, we use the publicly available *Electric Motor Temperature* (EMT) dataset from [8], consisting of temperature sequence data at different positions in a *Permanent Magnet Synchronous Motor* (PMSM) motor taken from a test bench. A total of ten input features are used, such as the speed of the electric motor, torque, current, ambient temperature, etc. The regression targets are the temperatures of the permanent magnet, stator yoke, stator tooth, and stator winding. In total, the dataset consists of 185 hours of recordings and integrates 69 different profiles. For the model, the CNN of [27] is chosen. It consists of two 1-D convolutional layers with 32 1×3 kernels, ReLU layers as activation functions, and one fully connected layer with four output neurons corresponding to the four target temperatures.

6 Experiments

The three datasets and models introduced in Section 5 have been implemented in PyTorch. Each model is trained on the respective dataset for 100 epochs, using the Adam optimizer, to minimize the Mean Squared Error (MSE) (anomaly detection and electric motor temperature), or the cross entropy (keyword spotting). A subset of the entire dataset not used for training was used for testing. For training and testing each explored neural network, we used an Nvidia RTX 2080 Ti GPU. For all benchmarks, we pruned the models using the ℓ^1 norm and setting $J = 10$. This results in 10 gradually stricter pruned model configurations. For deployment, the PyTorch models were exported in single-precision floating-point to ONNX and converted to C code (see Section 4.2). The resulting C code was compiled using the HighTec GCC compiler[6], using `-O3` optimization as an argument. Execution times were measured on the AURIX TC387_3.3 V_TFT evaluation board with a TriCore 387. The TriCore 387 supports floating-point computations running at a frequency of 300 MHz. It consists of 4 cores (CPU0, CPU1, CPU2, CPU3), which differ in the size of the scratchpad RAM, i.e., the size of scratchpad RAM of CPU0 and CPU1 is 240 kilobytes, whereas CPU2 and CPU3 are 96 kilobytes[7]. The TC387 has a ROM of size 10 megabytes, where the weights of the model and the program are stored. For a fair comparison, all models were run on a single core, with 240 kilobytes of scratchpad RAM, and the floating-point format was chosen.



(a) Autoencoder on anomaly detection

(b) CNN on keyword spotting

Fig. 2: Pruning sensitivities s_i for respective layer v_i, which serves to determine the initial pruning rate. The sensitivities for the ten fully connected layers of the Autoencoder (a) and for the four convolutional layers and the fully connected layer of the convolutional neural network for keyword spotting (b) are visualized.

[6] HighTec, “Tricore Development Platform v4.9.3.0-infineon-1.0”, https://hightec-rt.com/en/products/development-platform, Date accessed: 03/19/2024

[7] Infineon, “AURIX TC38x User Manual”, https://www.infineon.com, Date accessed: 03/19/2024

6.1 Anomaly Detection

In the first stage of the toolchain, the sensitivity analysis is conducted for the considered model on the respective test dataset. As a measure of the accuracy, we consider the area under the curve (AUC), which is the quality target for the anomaly detection dataset in the MLPerf tiny benchmark suite [12]. When the AUC exceeds the threshold of the unpruned model ($T = 0.86$) on the test dataset ("anomaly_id_01"), the maximum pruning rate $p_i^{\max}$ of the layer is determined (see Algorithm 1).

In Fig. 2a, the layer sensitivities (see Eq. (3)) for the anomaly detection model are reported. Remarkably, the first fully connected layer (fc0) has the lowest sensitivity, meaning that, at a maximum pruning rate of $p_0^{\max} = 0.7$, still the AUC of the unpruned model can be achieved. The other layers of the encoder network (fc0 to fc3) have high sensitivities and, the fully connected layers fc1 to fc3 have to stay unpruned in the following exploration loop. The decoder (fc5 to fc9), has high sensitivity at the output layer, which is typical for neural networks [6]. However, fc7 is an interesting case, which is highly sensitive compared to the other three layers, which is an interesting finding for this type of model. Based on the sensitivities s_i, and setting $J = 10$, according to Eq. (4), the initial pruning rates p_i^{init} for each layer are determined. As example for fc0 $p_0^{\text{init}} = \frac{1-s_0}{J} = \frac{1-0.3}{10}$. With $M_0 = 128$ output neurons, for $j = 1$, the number of output neurons m_0 computes to $m_0 = \lceil M_0 \cdot (1 - p_0^{\text{init}} \cdot j) \rceil = \lceil 128 \cdot (1 - \frac{1-0.3}{10} \cdot 1) \rceil = 120$, meaning the 8 output neurons with the lowest ℓ^1 norm are pruned. For each pruned model Y_j, C code is generated and the execution time as well as the memory utilization (ROM and scratchpad RAM) on the TriCore 387 are measured.

The results are depicted in Fig. 3. Each dot represents one explored pruned configuration. Note that the respective y-axis denotes the error, which for the Autoencoder benchmark is the area under the curve (AUC) subtracted from 1 (Error=1-AUC). The measurement of one pruned configuration requires about 40 seconds, with approximately 30 seconds required for determining the AUC for the test dataset and 10 sec-

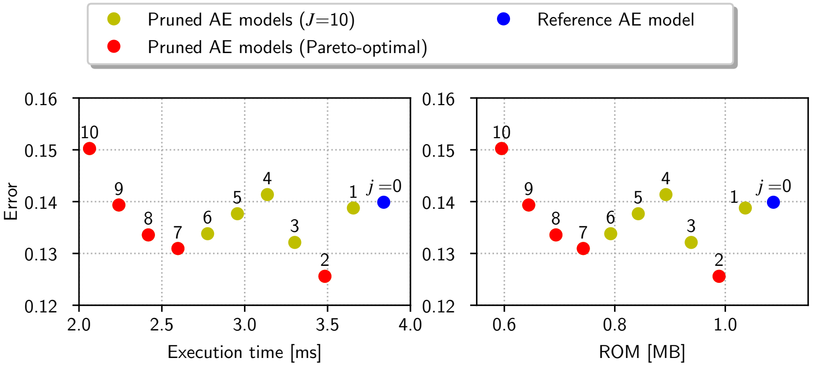


Fig. 3: J explored configurations based on global weighted pruning for the Autoencoder (AE) on anomaly detection benchmark. Each dot is numbered with the corresponding iteration j ($0 \leq j \leq 10$).

onds for C code generation, compilation, and deployment on the TC387 on the TriCore. Overall, the exploration for $J = 10$ requires only 7 minutes. The blue dot represents the unpruned baseline ($j = 0$), which requires 1.1 MB ROM (11% of the available ROM), 50 kB scratchpad RAM (21% of available RAM of CPU0), and has an inference time of 3.8 ms. Note that with increasing j and thus increasing degree of the global weighted pruning, the number of parameters (ROM) as well as the execution time is reduced due to a reduction in the number of floating point operations required to compute the inference for the model.

The red dots represent pruned AE models that are Pareto-optimal, i.e., none of these design points is dominated by any other explored point. Mathematically, a design point is not dominated if there is no point in the design space that is at least equally good in all objectives. As an example, point $j = 7$ dominates all points 0 and $3-6$, as error, ROM image, and execution time are all smaller than for any of these points. However, $j = 7$ does not dominate point $j = 2$, and vice versa.

As the successive layers fc2 to fc4 stay unpruned and as these define the maximum of RAM to allocate, the RAM requirements cannot be reduced by pruning for this specific neural network model. In Fig. 3, the non-dominated configurations $j = 2$ as well as $7-9$ all dominate the initial baseline model.

Regarding the ROM requirements, OpTC also explores if a pruned network fits the memory requirements of other microcontroller targets. For example, the TC32x microcontroller has only a ROM of 1 MB and for instance, the baseline Autoencoder model would not fit into the ROM. Here, OpTC provides pruned configurations (e.g., all configurations $j \geq 2$ in Fig. 3) that are deployable also on TC32x microcontrollers.

6.2 Keyword Spotting

Analogously to the Autoencoder, a sensitivity analysis of the convolutional neural network for the keyword spotting benchmark (see Section 5.2) was performed. The sensitivities in Fig. 2b indicate that also the first layer (conv0) is the least sensitive. However,

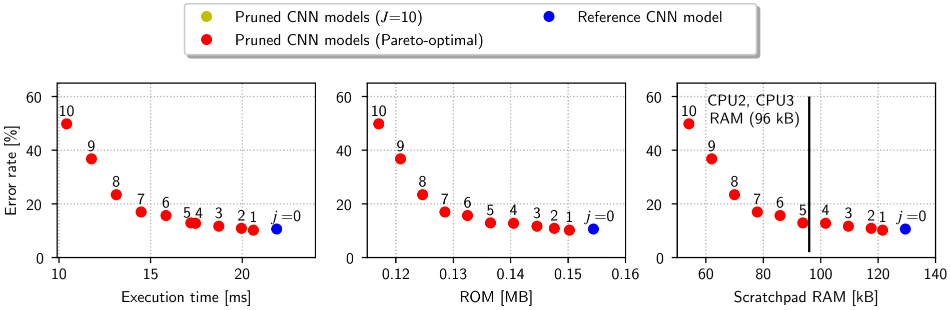


Fig. 4: J explored configurations based on global weighted pruning for the convolutional neural network for the keyword spotting benchmark. The vertical line in the rightmost plot indicates the available scratchpad RAM for CPU2 and CPU3. Here, all configurations $j \geq 5$ meet the scratchpad (RAM) requirements.

compared to the Autoencoder, it still has a higher sensitivity. Here, one can already see that the sensitivities differ significantly between different models and even an expert may have to carry out a lot of iterations to find which pruning rates for which layer yield the best trade-off between performance and prediction quality. The number of pruning configurations to be explored is set again to $J = 10$. The error rate is computed to determine the error rate of the convolutional neural network for keyword spotting. It denotes the number of misclassified samples (false predicted keyword) divided by the number of samples in the test dataset.

In contrast to the Autoencoder, we see that with increasing j (stricter pruning), the error rate increases as well (Fig. 4). Here, only configuration $j = 1$ dominates the baseline, as typically pruning a network with low pruning rates shows slightly lower error rate due to better generalization[8].

Considering the model's RAM utilization, the baseline model requires 130 kB RAM, which is 54% of the available RAM for CPU0, CPU1, respectively. However, the baseline CNN for keyword spotting exceeds the available RAM of CPU2 and CPU3 of TC387 (96 kB, marked by the black vertical line in Fig. 4). Here, each of the determined pruned configurations requires only 94 kB, which fits into CPU2 and CPU3, while the prediction error increases marginally by 2.7%. Based on the results of OpTC, the user can decide on which CPU of the TC387 the model should be deployed and if the resulting trade-off in error still meets his constraints.

6.3 Electric Motor Temperature

For evaluating the prediction quality of the temporal convolutional neural network for electric motor temperature prediction (see Section 5.3), we evaluated the average mean squared error (MSE) of four target temperatures.

The sensitivity analysis revealed that only the last fully connected layer is highly sensitive; Therefore, the layer stays unpruned during the exploration. As in [27], we

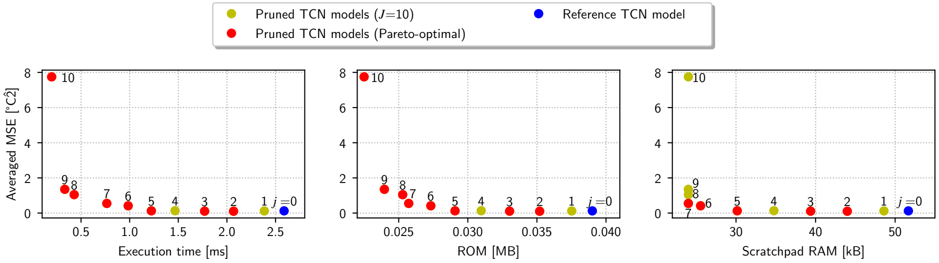


Fig. 5: J explored configurations based on global weighted pruning for the temporal convolutional neural network (TCN) trained on the electric motor temperature dataset.

[8] In terms of neural networks, *generalization* means that a neural network tends to overfit the training dataset. With methods such as pruning, typically the overfitting on the training data can be reduced, and neural networks show better prediction quality on the test dataset.

applied retraining of 50 epochs for each pruned model. In Fig. 5, the key performance indicators (i.e., execution time, MSE, and memory utilization) of the resulting pruned configurations are shown. Also in this benchmark, one can see that with increasing j and thus stricter prunings, the execution time decreases as well as ROM requirements due to the reduction of the number of parameters (and FLOPs). The baseline model requires 2.6 ms for inference and about 52 kB of RAM. With retraining, the mean-squared error does not increase in the first steps. Compared to the unpruned baseline, a pruned configuration with the same MSE with an execution time of 1.2 ms was found, resulting in a speedup of 2.2×. Notably, for the RAM, after steps $j = 7$, no further reduction could be achieved. At this point, the input and output features of the fully connected layer, which is left out from pruning, define the maximum of intermediate values to be stored.

7 Conclusion

In this work, we presented OpTC, a toolchain for automated neural network model compression and C code generation for AURIX TriCore microcontrollers. Our toolchain reduces the vast design space of pruning configurations by sensitivity analysis and is able to explore the reduced design space within minutes. For applications from the MLPerf Tiny benchmark on TC3xx targets that are commonly used in automotive systems, we showcased the efficiency of OpTC, providing speedups or reducing the memory footprint to make models deployable also on targets with a small capacity of scratchpad (RAM) memory. In the case of the anomaly detection and electric motor temperature benchmarks, OpTC provides pruned configurations showing speedups of 1.7× and 2.2×, respectively, without increasing the error over the baseline model.

Furthermore, we showed that the explorations using OpTC also enable increasing the range of microcontroller targets to which a given neural network can be deployed. For example, the TC32x microcontroller has only a ROM size of 1 MB. Here, the baseline Autoencoder model from the MLPerf Tiny Benchmark would not fit into the ROM. OpTC provides pruned configurations that are deployable on TC32x microcontrollers. In the case of the keyword spotting benchmark, we demonstrated how OpTC determines configurations that can be run on cores with reduced RAM size.

Acknowledgements: This work was supported by the Schaeffler Hub for Advanced Research at Friedrich-Alexander-Universität Erlangen-Nürnberg (SHARE at FAU).

References

[1] P. Petersen, T. Rudolf, and E. Sax. “A Data-driven Energy Estimation based on the Mixture of Experts Method for Battery Electric Vehicles”. In: *In Proceedings of the 8th International Conference on Vehicle Technology and Intelligent Transport Systems (VEHITS)*. SCITEPRESS, 2022, pp. 384–390. DOI: 10.5220/0011081000003191.

[2] W. Kirchgässner, O. Wallscheid, and J. Böcker. “Thermal neural networks: Lumped-Parameter Thermal Modeling with State-Space Machine Learning”. In: *Engineering Applications of Artificial Intelligence* 117 (2023), p. 105537. DOI: 10.1016/J.ENGAPPAI.2022.105537.

[3] M. Abadi, A. Agarwal, P. Barham, et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. 2015. URL: https://www.tensorflow.org.

[4] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. "Automatic differentiation in PyTorch". In: *In Proceedings of NIPS Autodiff Workshop*. OpenReview.net, 2017. URL: https://openreview.net/forum?id=BJJsrmfCZ.

[5] F. Chollet et al. *Keras*. https://keras.io. 2015.

[6] S. Han, J. Pool, J. Tran, and W. J. Dally. "Learning both Weights and Connections for Efficient Neural Network". In: *In Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS)*. 2015, pp. 1135–1143.

[7] C. Banbury, V. J. Reddi, P. Torelli, J. Holleman, N. Jeffries, C. Kiraly, P. Montino, D. Kanter, S. Ahmed, D. Pau, et al. "MLPerf Tiny Benchmark". In: *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks* (2021).

[8] W. Kirchgässner, O. Wallscheid, and J. Böcker. *Electric Motor Temperature*. 2021. DOI: 10.34740/KAGGLE/DSV/2161054. URL: https://www.kaggle.com/dsv/2161054.

[9] S. Han, H. Mao, and W. J. Dally. "Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding". In: *In Proceedings of 4th International Conference on Learning Representations (ICLR)*. 2016. DOI: 10.48550/arXiv.1510.00149.

[10] M. Deutel, P. Woller, C. Mutschler, and J. Teich. "Energy-efficient Deployment of Deep Learning Applications on Cortex-M based Microcontrollers using Deep Compression". In: *The Computing Research Repository (CoRR)* (2022). arXiv: 2205.10369 [cs.LG].

[11] C. Heidorn, N. Meyerhöfer, C. Schinabeck, F. Hannig, and J. Teich. "Hardware-Aware Evolutionary Filter Pruning". In: *Embedded Computer Systems: Architectures, Modeling, and Simulation - 22nd International Conference, SAMOS 2022, Samos, Greece, July 3-7, 2022, Proceedings*. Vol. 13511. Lecture Notes in Computer Science. Springer, 2022, pp. 283–299. DOI: 10.1007/978-3-031-15074-6_18.

[12] C. R. Banbury, V. J. Reddi, P. Torelli, N. Jeffries, C. Király, J. Holleman, P. Montino, D. Kanter, P. Warden, D. Pau, U. Thakker, A. Torrini, J. Cordaro, G. D. Guglielmo, J. M. Duarte, H. Tran, N. Tran, W. Niu, and X. Xu. "MLPerf Tiny Benchmark". In: *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*. Ed. by J. Vanschoren and S. Yeung. 2021. URL: https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/da4fb5c6e93e74d3df8527599fa62642-Abstract-round1.html.

[13] S. S. Saha, S. S. Sandha, and M. B. Srivastava. "Machine Learning for Microcontroller-Class Hardware: A Review". In: *The Computing Research Repository (CoRR)* (2022). arXiv: 2205.14550 [cs.LG].

[14] J. Lin, W. Chen, Y. Lin, J. Cohn, C. Gan, and S. Han. "MCUNet: Tiny Deep Learning on IoT Devices". In: *In Proceedings of the Annual Conference on Neural Information Processing Systems (NeurIPS)*. 2020.

[15] R. David, J. Duke, A. Jain, V. J. Reddi, N. Jeffries, J. Li, N. Kreeger, I. Nappier, M. Natraj, S. Regev, R. Rhodes, T. Wang, and P. Warden. "TensorFlow Lite Micro: Embedded Machine Learning on TinyML Systems". In: *The Computing Research Repository (CoRR)* (2020). arXiv: 2010.08678 [cs.AI].

[16] J. Ma. *A higher-level Neural Network library on Microcontrollers (NNoM)*. Version v0.4.2. Oct. 2020. DOI: 10.5281/zenodo.4158710.

[17] J. Bai, F. Lu, K. Zhang, et al. *ONNX: Open Neural Network Exchange*. 2019. URL: https://github.com/onnx/onnx.

[18] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Q. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy. "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning". In: *In Proceedings of the 13th USENIX Symposium on Op-*

erating Systems Design and Implementation (OSDI). USENIX Association, 2018, pp. 578–594.

[19] M. Alwani, H. Chen, M. Ferdman, and P. A. Milder. “Fused-Layer CNN Accelerators”. In: *In Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (Taipei, Taiwan). IEEE Computer Society, 2016, 22:1–22:12. DOI: 10.1109/MICRO.2016.7783725.

[20] C. Heidorn, F. Hannig, and J. Teich. “Design Space Exploration for Layer-parallel Execution of Convolutional Neural Networks on CGRAs”. In: *Proceedings of the 23rd International Workshop on Software and Compilers for Embedded Systems (SCOPES)*. ACM, 2020, pp. 26–31. DOI: 10.1145/3378678.3391878.

[21] S. Bunzel. “AUTOSAR – The Standardized Software Architecture”. In: *Informatik Spektrum* 34.1 (2011), pp. 79–83. DOI: 10.1007/S00287-010-0506-7.

[22] A. Burrello, A. Garofalo, N. Bruschi, G. Tagliavini, D. Rossi, and F. Conti. “DORY: Automatic End-to-End Deployment of Real-World DNNs on Low-Cost IoT MCUs”. In: *IEEE Transactions on Computers* 70.8 (2021), pp. 1253–1268. DOI: 10.1109/TC.2021.3066883.

[23] P. Novac, G. B. Hacene, A. Pegatoquet, B. Miramond, and V. Gripon. “Quantization and Deployment of Deep Neural Networks on Microcontrollers”. In: *Sensors* 21.9 (2021), p. 2984. DOI: 10.3390/s21092984.

[24] M. Sabih, A. Mishra, F. Hannig, and J. Teich. “MOSP: Multi-Objective Sensitivity Pruning of Deep Neural Networks”. In: *13th IEEE International Green and Sustainable Computing Conference, IGSC 2022, Pittsburgh, PA, USA, October 24-25, 2022*. IEEE, 2022, pp. 1–8. DOI: 10.1109/IGSC55832.2022.9969374. URL: https://doi.org/10.1109/IGSC55832.2022.9969374.

[25] C. Heidorn, M. Sabih, N. Meyerhöfer, C. Schinabeck, J. Teich, and F. Hannig. “Hardware-Aware Evolutionary Explainable Filter Pruning for Convolutional Neural Networks”. In: *International Journal of Parallel Programming* (2024). DOI: 10.1007/s10766-024-00760-5.

[26] N. Rotem, J. Fix, S. Abdulrasool, S. Deng, R. Dzhabarov, J. Hegeman, R. Levenstein, B. Maher, N. Satish, J. Olesen, J. Park, A. Rakhov, and M. Smelyanskiy. “Glow: Graph Lowering Compiler Techniques for Neural Networks”. In: *The Computing Research Repository (CoRR)* (2018). arXiv: 1805.00907 `[cs.PL]`.

[27] C. Heidorn, F. Hannig, D. Riedelbauch, C. Strohmeyer, and J. Teich. “Efficient Deployment of Neural Networks for Thermal Monitoring on AURIX TC3xx Microcontrollers”. In: *Proceedings of the 10th International Conference on Vehicle Technology and Intelligent Transport Systems (VEHITS)* (Angers, France). SciTePress, May 2–4, 2024.

[28] Y. Koizumi, S. Saito, H. Uematsu, N. Harada, and K. Imoto. “ToyADMOS: A Dataset of Miniature-Machine Operating Sounds for Anomalous Sound Detection”. In: *Proceedings of the IEEE Workshop on Applications of Signal Processing to Audio and Acoustics (WASPAA)* (New Paltz, NY, USA). IEEE, Oct. 20–23, 2019, pp. 313–317. DOI: 10.1109/WASPAA.2019.8937164.

[29] P. Warden. “Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition”. In: *The Computing Research Repository (CoRR)* (2018). arXiv: 1804.03209 `[cs.CL]`.

[30] W. Dai, C. Dai, S. Qu, J. Li, and S. Das. “Very deep convolutional neural networks for raw waveforms”. In: *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (New Orleans, LA, USA). IEEE, Mar. 5–9, 2017, pp. 421–425. DOI: 10.1109/ICASSP.2017.7952190.