Efficient Deployment of Neural Networks for Thermal Monitoring on AURIX TC3xx Microcontrollers

Christian Heidorn[1][a], Frank Hannig[1][b], Dominik Riedelbauch[2][c], Christoph Strohmeyer[2][d] and Jürgen Teich[1][e]

[1]*Department of Computer Science, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany*

[2]*Schaeffler Technologies AG & Co. KG, Herzogenaurach, Germany*

Keywords: AURIX TriCore, Neural Networks, Thermal Monitoring.

Abstract: This paper proposes an approach for efficiently deploying neural network (NN) models on highly resource-constrained microcontroller architectures, particularly AURIX TC3xx microcontrollers. Here, compression and optimization techniques of the NN model are required to reduce execution time while maintaining accuracy on the target microcontroller. Furthermore, especially on AURIX TriCores that are frequently used in the automotive domain, there is a lack of support for automatic conversion and deployment of pretrained NN models. In this work, we present an approach that fills this gap, enabling the conversion and deployment of so-called thermal neural networks on AURIX TC3xx microcontrollers for the first time. Experimental results on three different NN types show that, when pruning of convolutional neural networks is applied, we can achieve a speedup of 2.7× compared to state-of-the-art thermal neural networks.

1 INTRODUCTION

Deploying neural networks (NNs) on microcontrollers allows AI applications to be run close to the sensors and increases the scope for future applications. For example, NNs can be used in electric vehicles to predict battery charge (Petersen et al., 2022) or implement thermal management of the electric motor (Kirchgässner et al., 2023). However, executing neural networks on microcontrollers is challenging because of scarce memory resources and very limited computing power.

For deploying NNs on microcontrollers, inference libraries are typically designed to efficiently manage the few available resources and fit the tight memory budget (Lin et al., 2020). Typically, those lightweight inference libraries are implemented in C or C++ in combination with a conversion workflow (David et al., 2020; Deutel et al., 2022; Lin et al., 2020) that generates library function calls for an NN. These workflows are often realized within popular machine learning (ML) frameworks, e.g., TensorFlow (Abadi et al., 2015) or PyTorch (Paszke et al., 2017), or relying on an exchange format description, such as ONNX (Open Neural Network Exchange, (Bai et al., 2019)).

Workflows such as TensorFlow Lite Micro (David et al., 2020) and microTVM (Chen et al., 2018) have drawbacks as they rely on an interpreter to execute the network graph at runtime. This adds memory and latency overhead (Lin et al., 2020). In addition, optimizations are performed merely at the layer level of an NN, which misses the potential of globally optimizing the overall NN graph, e.g., by layer fusion (Alwani et al., 2016; Heidorn et al., 2020). Another drawback of inference libraries is the effort required to integrate them into other standardized automotive architectures, such as AutoSAR (Automotive Open System Architecture) (Bunzel, 2011), 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. This often necessitates a complete redesign of the inference library and workflow.

When it comes to developing a new NN model, most of the existing workflows do not provide any information about how the model will perform on the target microcontroller, especially considering the execution time. Automotive systems typically have real-time requirements. However, during the development

[a] https://orcid.org/0009-0002-7557-0350
[b] https://orcid.org/0000-0003-3663-6484
[c] https://orcid.org/0000-0002-7937-4755
[d] https://orcid.org/0000-0003-3982-4499
[e] https://orcid.org/0000-0001-6285-5862

Heidorn, C., Hannig, F., Riedelbauch, D., Strohmeyer, C. and Teich, J.
Efficient Deployment of Neural Networks for Thermal Monitoring on AURIX TC3xx Microcontrollers.
DOI: 10.5220/0012563100003702
Paper published under CC license (CC BY-NC-ND 4.0)
In *Proceedings of the 10th International Conference on Vehicle Technology and Intelligent Transport Systems (VEHITS 2024)*, pages 64-75
ISBN: 978-989-758-703-0; ISSN: 2184-495X
Proceedings Copyright © 2024 by SCITEPRESS – Science and Technology Publications, Lda.

of the neural network model, a programmer or data scientist is typically unaware of whether the model can deliver predictions within the specified time constraints. With current workflows, this leads to time-consuming trial and error, as the model has to be adapted, trained, and deployed on the target device several times until it meets the time and memory constraints for the target hardware while still achieving acceptable accuracy.

Given the limitations of the methods mentioned above, we present an approach that paves the way to high throughput, accurate, and low memory footprint ML implementations for highly resource-constrained microcontroller targets used in automotive applications, especially for the AURIX TriCore 3xx family.

Our main contributions are:

- A versatile C code generator supporting TC3xx AURIX embedded microcontrollers, enabling the seamless deployment of different types of neural networks and integration into automotive systems.
- A flexible and extendible approach from neural network implementation to deployment on the target microcontroller integrating important optimization techniques (i.e., pruning, and approximation of special activation functions) to reduce the size and execution time of the neural network model.
- A case study exploring the interplay between the optimization techniques and their impact on the execution time and prediction error of thermal neural networks (TNNs) in comparison with temporal convolutional neural networks (TCNs) and recurrent neural networks (RNNs) on AURIX TriCore 387.

The remainder of this paper is organized as follows: Section 2 provides fundamentals on neural network compression and optimizations, and Section 3 discusses related work. In Section 4, we present our novel approach. Section 5 introduces the case study, as well as the dataset and models, which are used in the experimental evaluation (Section 6) before Section 7 concludes.

2 FUNDAMENTALS

Neural networks (NNs) consist of multiple layers of different types and can be represented as data flow graphs. NNs have trainable layers with weights, such as fully connected and convolutional layers. Typically, non-linear activation functions are applied after each of these layers, e.g., rectified linear unit (ReLU), *hyperbolic tangent* (tanh), or *sigmoid* functions. A broad range of compression methods exists for efficient deployment on microcontrollers, including pruning (Han et al., 2016) and approximations of expensive activation functions (Qian et al., 2023).

Pruning

Pruning (also referred to as sparsification) reduces the number of neurons and their connections in order to compress the original model. The general goal of this sparsification is to reduce the number of parameters — and, by that, the number of floating point operations (FLOPs) used for multiply-accumulate operations for the trainable layers in the NN. There are two general pruning strategies: *element-wise pruning* (Han et al., 2016; Han et al., 2015) and *structural pruning* (He et al., 2017; Li et al., 2017). Element-wise pruning removes connections from the computational graph of an NN and sets individual weights of a given layer to zero. By contrast, structural pruning sets entire structures of parameters to zero. For example, entire filters can be removed for convolutional layers, or rows and columns of the weight matrix can be eliminated in fully connected layers. This can reduce execution time by reducing the number of loop iterations required to process the layers. There are several heuristics to decide which weights or structures of weights to remove. The most common heuristics are ℓ^1 or ℓ^2 norm of the weights. Here, the filters (or rows and columns) with the lowest ℓ^1 values are set to zero and removed.

Approximation of Complex Activation Functions

LSTM cells or thermal NNs presented in the use case (see Section 5) for predicting the temperature in an electric motor require special functions, such as *hyperbolic tangent* and *sigmoid* functions. Since these functions involve exponential calculations, they are comparatively time-consuming to calculate on microcontrollers. As an alternative, the sigmoid and hyperbolic tangent can be replaced by the *Hardsigmoid* (Equation (1)) and *Hardtanh* (Equation (2)) functions, respectively, which are far less compute-intensive.

$$\text{Hardsigmoid}(x) = \begin{cases} 0 & \text{if } x \leq -3 \\ 1 & \text{if } x \geq 3 \\ \frac{x}{6} + \frac{1}{2} & \text{otherwise} \end{cases} \quad (1)$$

$$\text{Hardtanh}(x) = \begin{cases} -1 & \text{if } x < -1 \\ 1 & \text{if } x > 1 \\ x & \text{otherwise} \end{cases} \quad (2)$$

In Section 6, we evaluate how applying pruning and replacing the *tanh* and *sigmoid* functions by their respective approximation affects the prediction error and execution time on a target microcontroller for thermal neural networks.

3 RELATED WORK

Traditional deployment of neural networks has a high memory and computational footprint, which hinders its direct application on highly resource-constrained microcontrollers. This requires a dedicated approach for developing and compiling neural networks to resource-constrained microcontroller targets, ensuring that the computational and latency budget is within the device limits while still achieving the desired performance (Saha et al., 2022). Recently, approaches from industry and academia that differ in their support of neural network types[1], input formats, and support of microcontrollers have emerged. A brief overview is given in the following and is summarized in Table 1.

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 (David et al., 2020; Deutel et al., 2022; Lin et al., 2020; Ma, 2020) developed in C or C++. Here, operator function calls required to compute the neural network inference are baked into C or C++ during code generation. Typically, the inference library is intended to be sufficiently generic to be used on any type of 32-bit microcontroller and, therefore, portable. However, this also means that there is no integration with specific microcontroller families and vendor tools. Another drawback is the additional overhead involved in integrating the inference library into an existing project. This can require some effort, especially when managing multiple dependencies and ensuring compatibility with other software components. In addition, the library itself can add code size, especially if it needs to be extended for other microcontroller targets, and a runtime overhead due to the library abstraction and additional processing required for the dynamic execution of models. These overheads make deployment even more challenging as they partially offset the achieved speed-up and compression by NN compression techniques. One inference library available for AURIX TriCore 2xx and 3xx microcontrollers is from Ekkono[2]. Ekkono provides a C inference library for deploying NN models on AURIX platforms. However, there is no information about the workflow and which frameworks or types of NNs are supported.

[1]For example, multilayer perceptron (MLP), convolutional neural networks (CNNs), recurrent neural networks (RNNs)

Target-Specific Inference Libraries

Inference libraries often use low-level optimizations, such as those provided by ARM's CMSIS-NN library and data formats (ARM uses the Q-format[3]). In order to support targets such as AURIX TriCores, the library or kernel functions have to be extended (Lai and Suda, 2018). So far, inference libraries such as MCUNet (Lin et al., 2020) and microTVM (Chen et al., 2018) are based on CMSIS-NN and thus do not support other targets than ARM. To target the TriCore microarchitecture, all operators would need to be newly implemented and added to the library. In addition, the workflow and its code generation backend needs to be modified or needs to be implemented, as recently shown for microTVM (Liu et al., 2023).

Generic Inference Libraries for Microcontrollers

NNOM (Ma, 2020), TF Lite Micro (David et al., 2020), and DNNruntime (Deutel et al., 2022) are examples of generic inference libraries capable of generating ANSI C code. TF Lite Micro and DNNruntime also support the floating point format. These libraries are generally suitable to generate compilable C code for TriCores, which has not yet been done. Implementing target-specific (e.g., for TriCore TC3xx) optimizations for operators or parallel execution on multiple cores can also be cumbersome, and core-specific C code generation may be required. Again, one has to dive into the inference library and add the C code for the operator, and usually, the code generation also has to be refined. Glow (Rotem et al., 2018) is a neural network compiler that uses multiple levels of its own *intermediate representation* (IR) in the entire stack. For example, Glow's CPU backend executes low-level Glow instructions and calls in its own standard library kernels implemented in C++ and compiled with LLVM. The latter point especially makes Glow challenging to bring on TC3xx microcontrollers due to limited compiler support.

[2]Ekkono, "From Connected To Smart", https://www.ekkono.ai/, Date accessed: 01/10/2024

[3]ARM, "Compute the layer Q-formats", https://developer.arm.com/documentation/102591/2208/Compute-the-layer-Q-formats, Date accessed: 12/20/2023

Table 1: Overview of existing workflows to deploy neural networks on microcontrollers.

Approach	**Frameworks / Formats**	**Networks**			**Optimized μC Backend**	**Output**	**Precision**	
		MLP	CNN	RNN			int	float
MCUNet (Lin et al., 2020)	TensorFlow, TF Lite	✓	✓	✗	ARM	C++ inference lib. & C++ descriptors	✓	✗
TF Lite Micro (David et al., 2020)	TensorFlow, TF Lite	✓	✓	✓	ARM, RISC-V	C++ inference lib. & C++ descriptors	✓	✓
NNOM (Ma, 2020)	Keras	✓	✓	✓	ARM, RISC-V	C inference lib. & C descriptors	✓	✗
DNNruntime (Deutel et al., 2022)	PyTorch, ONNX	✓	✓	✗	ARM	C inference lib. & C descriptors	✓	✓
microTVM (Chen et al., 2018)	TensorFlow, PyTorch, ONNX	✓	✓	✗	ARM	C inference lib. & C descriptors	✓	✗
Glow (Rotem et al., 2018)	PyTorch, ONNX	✓	✓	✓	–	requires LLVM compilation	✓	✓
Dory (Burrello et al., 2021)	PyTorch, ONNX	✓	✓	✓	RISC-V	C code	✓	✗
Proposed	**PyTorch, ONNX**	✓	✓	✓	**TriCore**	**C code**	✓	✓

Code Generators

Dory (Burrello et al., 2021) 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, Dory relies on quantized neural networks as inputs when it comes to model compression and only supports integer precision. However, we argue that on platforms such as AURIX embedded microcontrollers provide single-precision floating-point computation. The support of floating-point computations is beneficial, especially for complex activation functions (e.g., *hyperbolic tangent*), or to support mixed-precision models that are known to maintain high accuracy.

Supported Input Formats and Networks

Workflows from the literature typically differ in the description format of the neural network model and the support of operators (see Table 1). For example, TF Lite Micro and MCUNet support TF Lite models developed in TensorFlow and Keras, respectively. DNNruntime and Dory support the Open Neural Network Exchange Format (ONNX) (Bai et al., 2019), giving them a more comprehensive range of supported machine learning frameworks such as PyTorch, MXNet (Chen et al., 2015), and TensorFlow. All the libraries examined support MLP and CNN models. However, we found that some libraries (e.g. DNNruntime) do not support RNNs, which are particularly useful for predicting time series data.

3.2 Microcontroller-Specific Compression

A current research trend is to adapt NN models for use in constrained platforms by compressing the NN topologies themselves (Sandler et al., 2018), either directly or by applying neural architecture search (Liberis et al., 2021; Lin et al., 2020). Orthogonally, techniques for pruning, post-training quantization, and quantization-aware fine-tuning can be used to reduce the cost of individual operations in terms of energy, and of individual parameters in terms of memory.

Some of the approaches listed in Section 3.1 already integrate compression techniques. For example, TF Lite Micro is based on TensorFlow and Keras and provides quantization and weight clustering techniques. However, workflows such as TF Lite Micro do not come with tools to estimate performance indicators such as inference time or RAM and ROM usage (Novac et al., 2021), or profiling when executed on the target microcontroller. This is a major shortcoming, as the developer will only find out if the developed NN meets the constraints of the target hard-

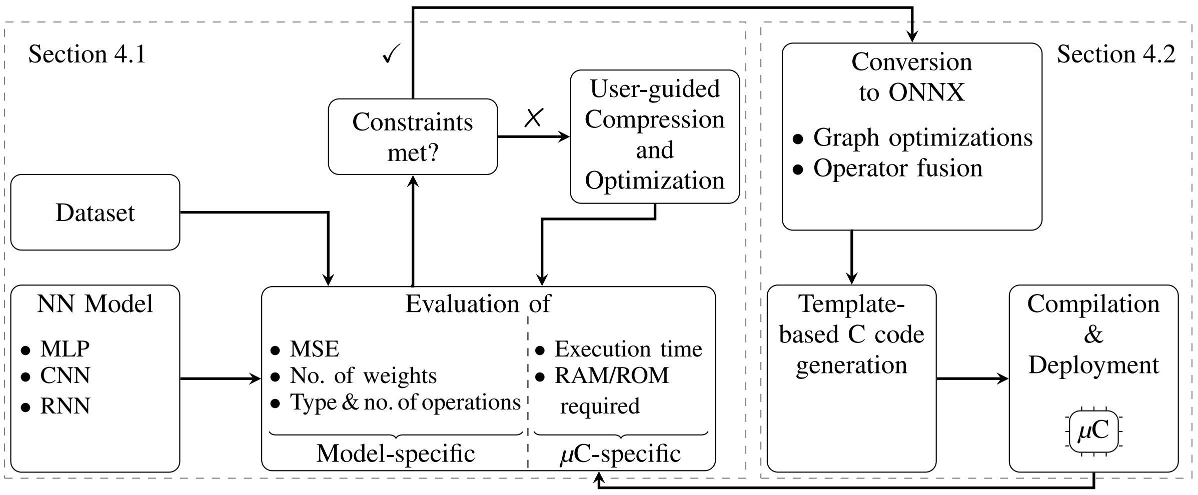


Figure 1: Overview of our proposed approach for seamless neural network deployment on the target microcontroller (μC). The arrows denote the path for deploying a given NN. If the constraints on execution time and memory consumption are met, the model is converted and code is generated. If the constraints are not met, the workflow offers the possibility to compress (prune) the model iteratively until the model satisfies the constraints.

ware after deployment. This results in a large time overhead as the NN has to be tuned, trained, and recompiled.

Instead, DNNruntime and MCUnet have the capability to estimate if the model meets the memory requirements (RAM and ROM usage) of the microcontroller. DNNruntime scales down already defined neural networks by means of 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 prediction quality of the model might be reduced. Again, the model has to be redesigned and trained, which is time-consuming. MCUNet focuses on neural architecture search (NAS) to find networks that meet target platform constraints. Here, the models have to be trained from scratch. If a model that meets the target platform constraints is found, the user can be sure that this model is deployable on the platform.

4 WORKFLOW

We have seen in Section 3 that existing workflows lack support for either (a) recurrent neural networks, (b) a TriCore backend, or (c) floating-point computations, and most come with an inference library. In contrast, our proposed workflow displayed in Fig. 1 targets all these problems. It integrates a conversion tool that maps pretrained NNs stored in ONNX format to C code, exploiting the static properties of trained NNs (i.e., fixed layer configurations and parameters) such that no dependence on an inference library is required. This not only reduces the computational overhead at runtime but also allows for seamless integration into other projects or frameworks. Moreover, our workflow gives options to apply various optimizations on a given neural network model, both at the graph level (e.g., by fusing successive operators or loop fusion) and at the operator level. In the following, we elaborate on the two main parts of the workflow: Section 4.1 presents the supported frameworks for NN development and compression. Section 4.2 explains the integrated code generator.

4.1 NN Model and Compression

Our workflow supports various Python frameworks for specifying neural networks, such as PyTorch or TensorFlow, by relying on the ONNX standard. The models are assumed to be predefined and pretrained by an ML specialist. Due to our modular concept, it is possible to integrate different Python libraries for compressing a predefined neural network, such as Microsoft Neural Network Intelligence (NNI)[4]. This provides options for pruning a predefined neural network to meet desired constraints (e.g., execution time). Here, the user guides the compression by selecting the strategy, either global or layer-wise pruning, and the respective pruning ratio(s). We showcase the workflow functionality by the example of global pruning in our experiments in Section 6 to

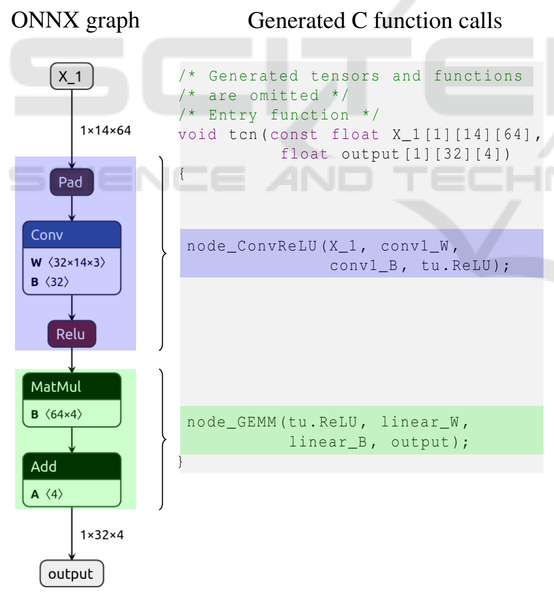
[4]Microsoft, "Neural Network Intelligence (NNI)", https://github.com/microsoft/nni, Date accessed: 01/15/2024

show the effect of pruning on the error in predicting engine temperatures from time series data. Integrating further compression techniques (e.g., quantization) is left for future work.

After the compression step, the models are converted to ONNX format. Python frameworks such as PyTorch and MXNet (Chen et al., 2015) have built-in functionality for ONNX export. To support the TensorFlow model, we use the tf2onnx[5] library.

4.2 Template-Based C Code Generation

ONNX (Bai et al., 2019) is an exchange format describing neural networks as dataflow graphs, where each node represents a mathematical operation, such as element-wise addition or matrix multiplication. One way to compile an ONNX graph into an executable would be to translate each layer operation directly into C code, containing loops and possibly other low-level instructions. However, it is beneficial to optimize the graph itself, e.g., by fusing the operational nodes of the graph, as C or C++ compilers typically do not perform these optimizations (Rotem et al., 2018). Notable optimization techniques include



```
/* Generated tensors and functions
/* are omitted */
/* Entry function */
void tcn(const float X_1[1][14][64],
              float output[1][32][4])
{

 node_ConvReLU(X_1, conv1_W,
                    conv1_B, tu.ReLU);


 node_GEMM(tu.ReLU, linear_W,
               linear_B, output);
}
```

Figure 2: An exemplary ONNX representation (left) and the C function calls generated by our workflow (right). The ONNX graph consists of a convolutional layer (Conv), an activation layer (ReLU), and a linear layer (MatMul and Add). The dimensions of the input (*batch_size* $\times$ *channels* $\times$ *feature_vector_length* $= 1 \times 14 \times 64$) and output tensor ($1 \times 32 \times 4$) are given at the arrows. Often, padding (zero-padding in this case) is applied to maintain the output dimension after the convolutional layer.

[5]ONNX, "tf2onnx", https://github.com/onnx/tensorflow-onnx, Date accessed: 01/15/2024

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 operations can be removed from the computational graph of an NN (e.g., in the case of zero-padding).

Currently, our workflow supports convolutions, linear transformations, pooling operations, and activation functions. It also supports more complex operators, such as long short-term memory cells, and applies approximations of activation functions.

Using an intermediate representation describing the operators in Python, for each operator (e.g., GEMM), we use different predefined, parameterized C code templates for code generation. Here, we apply code optimizations (e.g., operator or loop fusion) to reduce execution time and memory requirements, as shown for an example in Fig. 2. The example shows a convolutional layer with zero-padding applied, followed by an activation layer (here, ReLU) and a fully connected linear layer. The zero-padding operator can be directly integrated and removed from the convolutional layer. In our workflow, patterns (e.g., convolution followed by ReLU) are recognized, and a new operator (here, the ConvReLU operator) is generated while retaining the correct functionality. The linear layer consists of a matrix multiplication of the input matrix with a weight matrix, followed by a vector addition of the bias, which is fused to a GEMM operator. For the intermediate tensors, tensor unionization is performed to wrap the tensors into unions to help the compiler reuse heap memory (visualized by "tu." in Fig. 2). Unlike a runtime library, where new operators have to be added to an existing library and the workflow must be modified, it only takes one step to integrate new templates into our workflow. The final result of code generation is a single C file containing all the necessary operation nodes required for the model's inference. This file can be easily incorporated into other software projects, e.g., one can integrate the C code into Matlab (Matlab Embedded Coder) (MathWorks, 2022).

5 CASE STUDY

Temperature estimation tasks are necessary for electric drives for automotive as well as for 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. Advancing the trend of fitting models on empirical data, machine-learning-based thermal modeling, with neural networks was proposed recently.

NNs neither require any domain expert's knowledge nor geometry or material information for their design at the same level of accuracy as compared to formerly used lumped parameter thermal networks (LPTNs), which model equivalent circuit diagrams approximating inner heat transfer based on thermodynamic theory (Kirchgässner et al., 2021a). However, an increased computational demand for NNs in real-time is required. Kirchgässner et al. have tuned NNs to estimate temperatures of interest in a *Permanent Magnet Synchronous Motor* (PMSM) (Kirchgässner et al., 2021a). Here, recurrent neural networks and convolutional neural networks are used to predict the temperature profile in the stator teeth, winding, and yoke as well as the rotor's permanent magnets. Ground truth data is available from test bench runs.

Neural networks have already proven well for classification tasks (e.g., in image processing), and many works to compress them have been proposed. However, in the case of thermal monitoring, the neural networks are typically designed to solve a regression problem, minimizing a loss function, e.g., the Mean Squared Error (MSE, Eq. (3)) between the predicted value and the ground truth.

In the following, we consider a time series of samples i, with $\hat{y}_i$ denoting the predicted value, y_i the actual value, and N the number of samples. The MSE to be minimized is defined as

$$\mathrm{MSE} = \sum_{i=1}^{N} \frac{(\hat{y}_i - y_i)^2}{N}. \tag{3}$$

In our experiments (Section 6), we show how to compress neural networks designed for thermal prediction properly to fit the memory requirements of an AURIX TC3xx target and how pruning affects the predictions on the different targets, which, to the best of our knowledge, is done for the first time.

5.1 Dataset

For the training and evaluation, we used the publicly available *Electric Motor Temperature* (EMT) dataset from (Kirchgässner et al., 2021b), consisting of temperature sequence data at different positions in a 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 data set consists of 185 hours of recordings and integrates 69 different profiles.

5.2 Models

In our experiments, we compare three different models trained on the EMT dataset:

Thermal Neural Network (TNN) is a publicly available Neural Network from (Kirchgässner et al., 2023). It consists of three multi-layer perceptrons (MLPs), working as function approximators for modeling three ordinary differentiable equations[6]. The activation functions resulting in the lowest MSE found with an extensive design space exploration are *hyperbolic tangent* and *sigmoid*.

Temporal Convolutional Neural Network (TCN) is a type of convolutional neural network (CNN) designed explicitly for sequence modeling tasks. TCNs consist of 1-dimensional convolutional layers, where the trained filters are convolved over the time dimension. They showed promising results when applied to sequence data and are also known to perform better in terms of MSE than, for example, recurrent neural networks (RNNs) when applied to the problem of thermal prediction (Kirchgässner et al., 2021a). As we do not have access to the exact CNN model used by (Kirchgässner et al., 2021a), we reconstructed a comparable CNN by ourselves. In Kirchgässner et. al, the model consisted of two convolutional blocks of 16 kernels with a kernel size of 2. Our CNN consists of two 1-D convolutional layers of 32 kernels, each with a kernel size of 3, with rectified linear units (ReLU) layers used as activation functions, and one fully connected layer with four output neurons (corresponding to the four target temperatures). The intention behind our larger design is to give more opportunity to prune the TCN, which in turn reduces the number of kernels and may lead to better performance after pruning and retraining.

Recurrent Neural Network (RNN) has also proven to be a valuable type of neural network for sequence modeling tasks. Although it has already been shown in (Kirchgässner et al., 2021a) that RNNs result in a higher MSE, we have integrated the RNN to also

[6]One function approximator outputs thermal conductances, another the inverse thermal capacitances, and the last one the power losses generated within the components.

Table 2: Comparison of thermal neural network (TNN), TNN with approximated activations functions (TNN-$H_{S,T}$, TNN-H_S, TNN-H_T), temporal convolutional neural network (TCN) with different pruning rates, and recurrent neural network (RNN) wrt. to mean squared error on the test data set, the number of parameters, and the measured execution times on the TC387.

Model Type	Pruning Rate	No. of Parameters	MSE in °C^2				Exec. Time on
			Permanent Magnet	Stator Yoke	Stator Tooth	Stator Winding	TC387 in ms
TNN	0%	552	4.04	2.49	2.92	4.80	0.81
TNN-$H_{S,T}$	0%	552	10.82	2.64	5.65	9.67	0.26
TNN-H_S	0%	552	4.01	2.04	3.26	4.41	0.43
TNN-H_T	0%	552	5.31	2.05	3.36	5.85	0.66
TCN	0%	4,612	0.03	0.21	0.24	0.10	2.66
	50%	1,540	0.13	1.89	1.10	2.07	0.89
	60%	1,135	0.37	2.45	0.88	2.20	0.67
	70%	784	2.03	2.31	1.11	2.00	0.47
	75%	580	2.93	3.13	1.13	2.79	0.30
	80%	487	14.95	5.17	1.20	7.38	0.26
	90%	244	59.71	4.50	3.67	15.37	0.15
RNN	0%	2,116	2.25	2.47	2.77	3.83	35.5

provide execution time measurements for comparison with the previous two neural networks. Similarly, our RNN consists of a long short-term memory cell with a hidden size of 16 and, again, one fully connected layer with four output neurons.

6 EXPERIMENTS

The three models introduced in Section 5.2 have been implemented in PyTorch and trained on the EMT dataset (66 profiles were used for training) for 100 epochs, using the Adam optimizer, with the goal of minimizing the Mean Squared Error (MSE) for four respective target temperatures (permanent magnet, stator yoke, stator tooth, stator winding). We used the remaining three profiles (one example is visualized in Fig. 3) for each target temperature as test dataset. To determine the MSE, we calculate the average MSE over the three profiles for each target temperature. For deployment, PyTorch models are exported in single-precision floating-point to ONNX and converted to C code using our workflow depicted in Fig. 1. The resulting C code was compiled using the HighTec GCC compiler[7] 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. 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. All models and input samples fit into the scratchpad memory. The TCN consumes the most RAM (25 kilobytes, 11% of available RAM) because of the large size of the intermediate tensors (feature maps). The TNN (188 bytes) and the RNN (1.2 kilobytes) use less than 1% of the available RAM. For all three models, the available ROM (10 megabytes), where the weights and the program are stored, was used by less than 1%. The execution of the model is bare-metal, i.e., without any operating system in between. The number of clock ticks (cycles) was measured using `Ifx_TickTime` functions as part of the IFX low-level driver library (iLLD), which accesses the performance counter of the TriCore. More specifically, the cycles from calling the first operator of the NN until writing back the resulting output (vector with the predicted target temperatures) were counted. Each experiment was conducted 100 times to calculate the average number of cycles of one inference, from which we derived the average execution time corresponding to the clock frequency of the TriCore.

Table 2 gives an overview of the MSE for the unpruned (i.e., pruning rate equals 0%) TNN, TCN, and RNN. The TNN and RNN show a significantly higher MSE for the four target temperatures compared to the unpruned TCN on the test data. We explain this difference by the fact that the unpruned TCN consists of many more parameters (and operations) than the TNN. For an input sequence of 64 samples (which is used as the input length during training), the compu-

[7]HighTec, "Tricore Development Platform v4.9.3.0-infineon-1.0", https://hightec-rt.com/en/products/development-platform, Date accessed: 01/15/2024

tation of the TCN involves 290K floating-point operations (FLOPs). In contrast, the computation of the TNN involves only 33K FLOPs, which is also visible in the higher execution times for the unpruned TCN in Table 2. However, for the RNN, which requires about 124K operations, the execution time is much higher than for the TCN. The main reason for this is that compute-intensive *tanh* and *sigmoid* computations are required, and the implementation of the LSTM cell[8] is much more complex than that of a convolutional or linear layer. This may result in fewer optimization possibilities for the compiler. In the following, we prune the TCN to reduce the execution time (see Section 6.1). Furthermore, in Section 6.2, we propose to optimize the TNN by replacing the costly *tanh* and *sigmoid* computations to see how the execution time and the MSEs for the target temperatures are affected.

6.1 Pruning

The TCN model has the highest execution time due to its large number of parameters but also has a significantly lower MSE for all four target temperatures. Therefore, we applied structural pruning of the TCN to reduce the execution time. For this experiment, we prune the TCN iteratively using the ℓ^1 norm and start with a pruning rate of 50%. The pruning rate indicates how many filters are removed from each convolutional layer (the higher the pruning rate, the more filters are removed). After pruning the filters, we applied retraining for 50 epochs to allow the remaining filter weights to adjust properly. The results in Table 2 show that the TCN with a pruning rate of 60% has a lower execution time on the TriCore and still has a lower MSE for all four target temperatures compared to the TNN. In Fig. 3, the temperature predictions of the 75% pruned TCN and the TNN are compared with the ground truth values from one profile of the test set. Here, it can be seen that the pruned TCN can still predict the three target temperatures, stator tooth, and stator winding, as well as the TNN, with slightly less deviation when predicting the temperature of the permanent magnet. When the 75% pruned TCN predicts the stator yoke, there is some deviation from the ground truth. For a pruning rate of 75%, the average MSE of the stator yoke (3.13°C^2) is slightly higher than for the TNN (2.49°C^2).

One possible improvement could be to include an additional weighting for the target temperature when calculating the loss during retraining of the pruned ar-

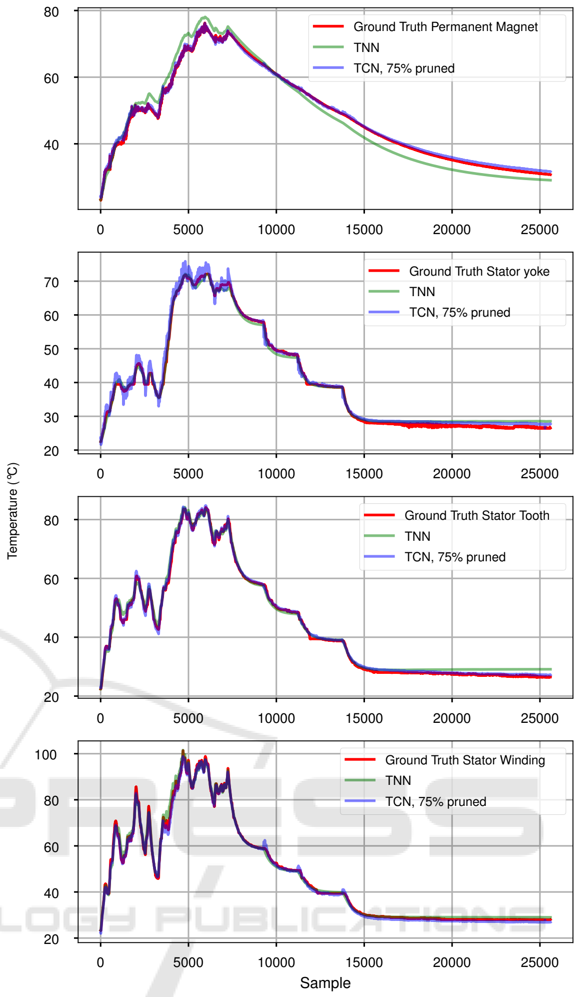


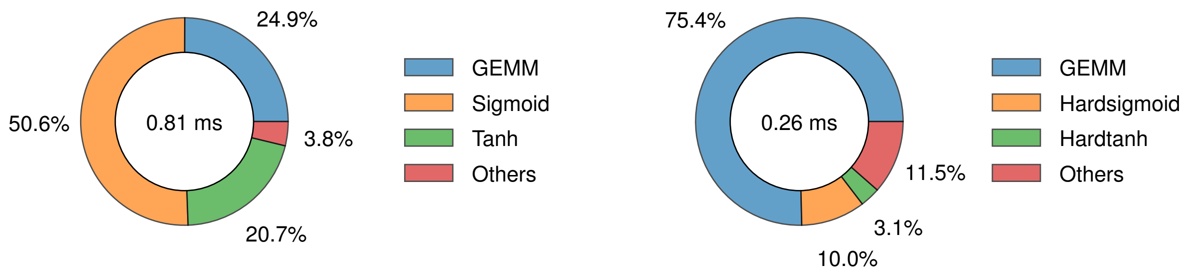
Figure 3: Predictions of TNN and 75% pruned TCN for an unseen temperature profile from the test data for the four target temperatures.

chitecture, but this is beyond the scope of this work. However, the average MSE over all four targets of the TCN is still below that of the TNN; and at the pruning rate of 75%, it achieves a speedup of 2.7× compared to the TNN.

6.2 Approximation of Activation Functions

Although the TNN requires very few parameters and FLOPs, and still has comparable error rates to a 75% pruned TCN, it takes much longer to compute. The reason for this becomes apparent when looking at the different types of operations used in the TNN (see Fig. 4). The hyperbolic tangent and the sigmoid functions take the most time (almost 70% of the overall execution time). This is despite the fact that we

[8]PyTorch Documentation, LSTM, https://pytorch.org/docs/stable/generated/torch.nn.LSTM.html, Date accessed: 01/16/2024



(a) TNN (b) TNN-$H_{S,T}$

Figure 4: Breakdown of the execution times for the required operations for the (a) TNN and (b) TNN-$H_{S,T}$ on a TC387. The TNN operations hyperbolic tangent (tanh) and sigmoid take more than 70% of the computation time. With the approximated activation functions (TNN-$H_{S,T}$), the execution time is drastically reduced.

have integrated the `float.h` library to calculate the exponential functions. We have therefore optimized the TNN by replacing the hyperbolic tangent and sigmoid functions with the *hardtanh* and *hardsigmoid* functions supported by the pipeline, which is declared TNN-$H_{S,T}$ in Fig. 4. This drastically speeds up the execution by a factor of 3.1. However, this speedup comes with a trade-off, as for some target temperatures, the MSE increases significantly. Therefore, we propose two additional models, TNN-H_T and TNN-H_S, where we only approximate the *tanh* or *sigmoid*, respectively. With this, the MSE is on par with TNN while achieving a speedup of 1.23× for TNN-H_T and 1.88× for TNN-H_S. Nevertheless, the 75% pruned TCN model still achieved a lower MSE and execution time. In the future, one option to achieve a lower MSE for approximated TNN-$H_{S,T}$ might be to increase the model size to compensate for the loss in accuracy of the approximated activation functions.

7 CONCLUSION

In this work, we presented a new approach for deploying different types of neural networks on AURIX TriCore (TC3xx) targets that are commonly used in automotive systems. Apart from supporting not only a restricted set of NNs such as CNNs, our approach provides means to compress and optimize a given model to adjust memory consumption properly, as well as to reduce the execution time during deployment. In addition, we provide insights for developers designing a neural network and planning to deploy it on an automotive microcontroller. In a case study on thermal electric motor management, we showed that a conventional approach using a thermal neural network (TNN), despite its expert design, has disadvantages in terms of accuracy and especially execution time when deployed on a target platform. NNs can be used to provide accurate temperature estimations from time series. In this regard, we evaluated TCNs and TNNs using the workflow. The TNN consists of many nodes with expensive activation functions (hyperbolic tangent and sigmoid), which require about 70% of the execution time on the TriCore. The considered TCNs do not require these types of activation functions and still have the potential to reduce execution time further by applying pruning. The 75% pruned TCN achieves a speedup of 2.7× compared to the TNN. However, it is possible to additionally optimize the TNNs, e.g., by approximating the activation functions, which leads to a speedup of 3.1× but increases the MSE significantly. A future direction here could be to balance complex activation functions and their less accurate counterparts, as well as to retrain the model or increase the model size. In the future, we plan to integrate cost models into the workflow so that the developer can get an estimate of the execution time before deploying the models on a given AURIX TriCore. This also paves the way for investigating automated search techniques (e.g., (Heidorn et al., 2022; Heidorn et al., 2024)) for neural architectures that aim for NNs with low execution time on the microcontroller while maintaining low error rates.

ACKNOWLEDGEMENTS

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

REFERENCES

Abadi, M., Agarwal, A., Barham, P., et al. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems.

Alwani, M., Chen, H., Ferdman, M., and Milder, P. A. (2016). Fused-layer CNN accelerators. In *In Proceed-*

ings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 22:1–22:12. IEEE Computer Society.

Bai, J., Lu, F., Zhang, K., et al. (2019). ONNX: Open neural network exchange.

Bunzel, S. (2011). AUTOSAR – The standardized software architecture. *Informatik Spektrum*, 34(1):79–83.

Burrello, A., Garofalo, A., Bruschi, N., Tagliavini, G., Rossi, D., and Conti, F. (2021). DORY: Automatic end-to-end deployment of real-world DNNs on low-cost IoT MCUs. *IEEE Transactions on Computers*, 70(8):1253–1268.

Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., and Zhang, Z. (2015). MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. *The Computing Research Repository (CoRR)*.

Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E. Q., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., Guestrin, C., and Krishnamurthy, A. (2018). TVM: An automated end-to-end optimizing compiler for deep learning. In *In Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 578–594. USENIX Association.

David, R., Duke, J., Jain, A., Reddi, V. J., Jeffries, N., Li, J., Kreeger, N., Nappier, I., Natraj, M., Regev, S., Rhodes, R., Wang, T., and Warden, P. (2020). TensorFlow Lite Micro: Embedded machine learning on TinyML systems. *The Computing Research Repository (CoRR)*.

Deutel, M., Woller, P., Mutschler, C., and Teich, J. (2022). Energy-efficient deployment of deep learning applications on Cortex-M based microcontrollers using deep compression. *The Computing Research Repository (CoRR)*.

Han, S., Mao, H., and Dally, W. J. (2016). Deep compression: Compressing deep neural network with pruning, trained quantization and Huffman coding. In *In Proceedings of 4th International Conference on Learning Representations (ICLR)*.

Han, S., Pool, J., Tran, J., and Dally, W. J. (2015). Learning both weights and connections for efficient neural network. In *In Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS)*, pages 1135–1143.

He, Y., Zhang, X., and Sun, J. (2017). Channel pruning for accelerating very deep neural networks. In *In IEEE International Conference on Computer Vision (ICCV)*, pages 1398–1406. IEEE Computer Society.

Heidorn, C., Hannig, F., and Teich, J. (2020). 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)*, pages 26–31. ACM.

Heidorn, C., Meyerhöfer, N., Schinabeck, C., Hannig, F., and Teich, J. (2022). 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*, volume 13511 of *Lecture Notes in Computer Science*, pages 283–299. Springer.

Heidorn, C., Sabih, M., Meyerhöfer, N., Schinabeck, C., Teich, J., and Hannig, F. (2024). Hardware-aware evolutionary explainable filter pruning for convolutional neural networks. *International Journal of Parallel Programming*.

Kirchgässner, W., Wallscheid, O., and Böcker, J. (2021a). Data-driven permanent magnet temperature estimation in synchronous motors with supervised machine learning: A benchmark. *IEEE Transactions on Energy Conversion*, 36(3):2059–2067.

Kirchgässner, W., Wallscheid, O., and Böcker, J. (2021b). Electric motor temperature.

Kirchgässner, W., Wallscheid, O., and Böcker, J. (2023). Thermal neural networks: Lumped-parameter thermal modeling with state-space machine learning. *Engineering Applications of Artificial Intelligence*, 117:105537.

Lai, L. and Suda, N. (2018). Enabling deep learning at the IoT edge. In *In Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, page 135. ACM.

Li, H., Kadav, A., Durdanovic, I., Samet, H., and Graf, H. P. (2017). Pruning filters for efficient ConvNets. In *In Proceedings of the 5th International Conference on Learning Representations (ICLR)*. OpenReview.net.

Liberis, E., Dudziak, L., and Lane, N. D. (2021). μNAS: Constrained neural architecture search for microcontrollers. In *In Proceedings of the 1st Workshop on Machine Learning (EuroMLSys)*, pages 70–79. ACM.

Lin, J., Chen, W., Lin, Y., Cohn, J., Gan, C., and Han, S. (2020). MCUNet: Tiny deep learning on IoT devices. In *In Proceedings of the Annual Conference on Neural Information Processing Systems (NeurIPS)*.

Liu, C., Jobst, M., Guo, L., Shi, X., Partzsch, J., and Mayr, C. (2023). Deploying machine learning models to ahead-of-time runtime on edge using MicroTVM. *The Computing Research Repository (CoRR)*.

Ma, J. (2020). A higher-level Neural Network library on Microcontrollers (NNoM).

MathWorks (2022). Statistics and machine learning toolbox.

Novac, P., Hacene, G. B., Pegatoquet, A., Miramond, B., and Gripon, V. (2021). Quantization and deployment of deep neural networks on microcontrollers. *Sensors*, 21(9):2984.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. (2017). Automatic differentiation in PyTorch. In *In Proceedings of NIPS Autodiff Workshop*. OpenReview.net.

Petersen, P., Rudolf, T., and Sax, E. (2022). 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)*, pages 384–390. SCITEPRESS.

Qian, C., Ling, T., and Schiele, G. (2023). Energy efficient LSTM accelerators for embedded FPGAs through parameterised architecture design. In *In Proceedings of the 36th International Conference on Architecture of Computing Systems (ARCS)*, pages 3–17. Springer.

Rotem, N., Fix, J., Abdulrasool, S., Deng, S., Dzhabarov, R., Hegeman, J., Levenstein, R., Maher, B., Satish, N., Olesen, J., Park, J., Rakhov, A., and Smelyanskiy, M. (2018). Glow: Graph lowering compiler techniques for neural networks. *The Computing Research Repository (CoRR)*.

Saha, S. S., Sandha, S. S., and Srivastava, M. B. (2022). Machine learning for microcontroller-class hardware: A review. *The Computing Research Repository (CoRR)*.

Sandler, M., Howard, A. G., Zhu, M., Zhmoginov, A., and Chen, L. (2018). MobileNetV2: Inverted residuals and linear bottlenecks. In *In IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4510–4520. Computer Vision Foundation / IEEE Computer Society.