This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2024.3439850

IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2023.0322000

Conformal Prediction based Confidence for Latency Estimation of DNN Accelerators: A Black-box Approach

MATTHIAS WESS[1,2], DANIEL SCHNÖLL[1,2], DOMINIK DALLINGER[1,2], MATTHIAS BITTNER[1,2] and AXEL JANTSCH[1,2]

[1]Institute of Computer Technology, TU Wien, 1040 Vienna, Austria
[2]Christian Doppler Laboratory for Embedded Machine Learning, Institute of Computer Technology, TU Wien, 1040 Vienna, Austria

Corresponding author: Matthias Wess (e-mail: matthias.wess@tuwien.ac.at).

This work was supported in part by the Austrian Federal Ministry for Digital and Economic Affairs, in part by the National Foundation for Research, Technology and Development, and in part by the Christian Doppler Research Association.

ABSTRACT Today, there exists a large number of different embedded hardware platforms for accelerating the inference of Deep Neural Networks (DNNs). To enable rapid application development, a number of prediction frameworks have been proposed to estimate the DNN inference latency on a wide range of hardware platforms. This work presents a novel smart padding benchmarking method, which allows the profiling of hardware platforms without requiring detailed per-layer reports. To mitigate the measurement inaccuracies inherent in the black-box approach, a confidence framework comprising three metrics has been developed. These metrics not only enhance the interpretation of prediction results but also significantly contribute to the refinement of the estimation framework itself, as they facilitate to improve the coverage of the training dataset for relevant layers and detect weaknesses in the training dataset. Empirical results demonstrate the method's robustness, with average prediction errors minimized to below 10% for smart padding benchmarking-based ANNETTE predictions for the Jetson Xavier, NXP i.MX93, and NXP i.MX8M+.

INDEX TERMS Estimation, Latency, Confidence, Neural Network Hardware, Conformal Prediction

I. INTRODUCTION

THE vast design space of optimization, pruning, quantization and mapping DNNs on embedded hardware platforms makes it almost impossible to quickly find the best fitting solution for an application. Neural Architecture Search (NAS) [1], [2] provides a means to achieve a DNN optimized with regards to certain requirements. Specifically in hardware-aware NAS the inference latency is often used as the target constraint and therefore needs to be computed or measured for each selected DNN architecture. To avoid the need to deploy each DNN on the requested platforms, various approaches have been proposed to predict the inference latency. Solutions to this problem range from the use of simple proxy metrics (such as the number of floating point operations) [3] and analytical models [4], [5] to Graph Convolutional Networks (GCNs) [6], [7]. Some solutions focus on specific design spaces to enable hardware-aware NAS and therefore provide limited generalization capabilities. Other methods (e.g. ANNETTE [5], nn-Meter [8]) aim to provide accurate predictions for a wide range of applications and cover the aspects of graph optimizations in a separate step to correctly model all steps in the deployment flow.

However, the vast amount of different hardware platforms available for DNN inference makes the adaption of estimation algorithms for each hardware platform cumbersome. Therefore, Metrics that provide additional information

Our goal is to address two challenges related to benchmarking and predicting the inference time of neural networks on constrained devices. First, benchmarking hardware platforms for specific DNNs is challenging due to layer fusion, dependencies on layer sequences, data loading effects, interference of profiling techniques with execution time, and other complications. Additionally, it is important to gain insights not only for entire networks but also at the per-layer level. Currently, achieving this level of detail necessitates the use of per-layer profiling results to accurately model execution time. However, there are situations where implementing per-layer profiling is not feasible or requires additional implementation effort and possibly generates additional profiling overhead. We tackle this challenge by developing an intelligent bench-

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2024.3439850

marking strategy that allows for the generation of per-layer abstraction models without relying on detailed insights.

Second, a latency estimate is only useful to designers if they know to which extent it can be trusted. How can we ensure the comparability of models and how can we trust models trained on limited data points? To address this concern, we propose three novel confidence metrics. These metrics provide quantitative measures of the reliability of our latency prediction models, enabling informed decision-making when selecting hardware platforms and DNN architectures for the application-specific DNN hardware implementation. Additional applications of latency prediction, such as hardware aware DNN compression [9], [10] and DNN offloading and partitioning [11], [12] can also potentially make use those prediction reliability measures.

Specifically, this paper makes the following contributions:

- We propose a method for profiling the latency of DNN inference on hardware with padded models;
- We propose a conformal prediction framework for DNN latency prediction to quantify the confidence of the predicted values.

II. RELATED WORK

a: Latency prediction

The goal of latency prediction is to estimate the total execution time of a network composed of a sequence of N layers $L = \{l_1, l_2, ..., l_N\}$. Each layer in the DNN has specific attributes and parameters that define its configuration, computation needs, and connections to other layers. These connections determine the data flow through the entire network. Current approaches for DNN latency prediction range from simple analytical models based on the roofline model [5] to elaborate Machine Learning (ML) based latency estimators [13]. These ML based prediction algorithms are trained on collected datasets $Z = \{(\vec{x_1}, y_1), (\vec{x_2}, y_2), ...\}$, where $\vec{x_i}$ are the feature vectors, describing layer i, and y_i are the values to be predicted. In the case of latency estimation, the target values can represent for example time or compute efficiency. As a result, ML based prediction algorithms are not limited to a specific hardware platform. They show good accuracy [14]–[16] but are mostly limited to the selected design space and are usually not designed for general network prediction.

Analytical prediction methods such as those presented in [17] and [18] provide high prediction accuracy for the target hardware platforms. However, they require in-depth hardware knowledge and are therefore not suitable when in-depth architecture details cannot be obtained due to confidentiality or when the required effort is excessive.

The latency prediction framework Blackthorn [4] encompasses analytical models constructed based on several measurement points. The focus of Blackthorn is on finding optimal measurement points to reduce the required amount of overall measurements to profile NVIDIA Graphic Processing Units (GPUs).

The framework ANNETTE [5] provides analytical models based on a refinement of the roofline model which, in addition to the compute and memory boundary, also takes into account the underlying compute architecture. In addition, ANNETTE relies on random forest regression models predicting the per-operator compute efficiency and also deploys decision trees to predict operator fusion rules.

Other similar approaches with iterative improvements and slightly different focus with regard to the profiled hardware [19], [20] have been proposed. nn-Meter [8] focuses on the prediction of mobile devices and deploys similar principles as ANNETTE relying on a larger training dataset. MAPLE-X [21] incorporates explicit prior knowledge of hardware devices to improve the prediction accuracy for newly benchmarked devices.

Finally, Graph Neural Networks (GNNs) offer the option to operate directly on the graph structure of the DNN to be predicted. Sectum [22] deploys a GNN to detect memory over-commitment in addition to an ANNETTE-like structure. While frameworks like DNNPerf [13] and GENNAPE [23] focus on the prediction of other DNN performance parameters (such as accuracy, training time, etc.), PerfSAGE [7] and DIPPM [24] rely on GNNs to predict latency, energy, and memory consumption and promise high prediction accuracy for different classes of network architectures. In both cases, the GraphSAGE architecture is deployed in different variants. Lastly, SLAPP [6] applies GNNs at sub-graph level to preserve the advantage of gained insights through per-operator prediction but still relying on a large number of data points.

The black-box approach using smart padding, presented in this work, can be a valuable method for most of the ML based latency estimation frameworks. Even though the technique does not replace detailed per-layer profiles, it enables in-model latency measurement of single layers or blocks of layers while decreasing the required effort for implementing overhead-free per-layer profiling tools.

b: Conformal prediction

The conformal prediction framework, introduced by Vovk, Grammerman and Shafer [25], [26] provides a general method for quantifying the uncertainty of predictions for arbitrary prediction algorithms and provides guarantees on the prediction error. Traditionally, confidence intervals are estimated using quantile regression [27], [28] or Bayesian methods [29]. In the context of this work, which leverages random forest regression [30], conformal prediction is particularly beneficial for uncertainty quantification, as it not only demonstrates good efficiency [31] but also ensures broad applicability across different machine learning algorithms. Furthermore, conformal prediction offers many additional advantages, such as its straightforward interpretability, model-agnostic nature, and adaptability.

For uncertainty quantification, conformal prediction relies on the computation of a nonconformity score α for each instance in a calibration set distinct from the initial training set. In regression, α is typically computed as the absolute error $\alpha_j = |\hat{y}_j - y_j|$ [31], where $\hat{y}$ is the predicted value. For the prediction of confidence intervals with significance δ, these

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2024.3439850

calculated nonconformity scores are used to formulate the prediction region for each instance j as $\hat{Y}_j^{\delta} = \hat{y}_j \pm \alpha(\delta)$ [31]. This means the predicted region $\hat{Y}$ will cover the true value of y with probability $p = 1 - \delta$. In the standard case, this results in confidence intervals of uniform width across all input feature vectors $\vec{x}$.

Thus, to minimize the average interval width, it is possible to implement normalized nonconformity functions [32]. Here, the nonconformity scores are scaled by σ, an estimate of the model's accuracy for the predicted instance. The resulting prediction regions are then computed as $\hat{Y}_j^{\delta} = \hat{y}_j \pm \alpha(\delta) \cdot \sigma_j$. This quality estimate can be obtained by various methods, such as predicting the errors with additional trained models or using the errors of the k nearest neighbors. Conformal prediction has been successfully applied in various domains, including medical diagnosis [33], face recognition [34], and financial risk prediction [35]. However, to our knowledge, this work presents the first approach to leverage conformal prediction for confidence estimation in latency estimation of DNNs.

III. METHODOLOGY

Currently, for latency estimation of DNN hardware accelerators, we encounter two primary challenges:

1. Across the broad spectrum of available DNN accelerators the availability of knowledge, insight, and profiling tools varies widely. This diversity necessitates tailored benchmarking and modeling approaches for each type.
2. The accuracy of latency prediction models varies widely due to variations in benchmarking methodologies, dataset size (e.g. limited due to the compilation time), and hardware architectures. These issues compromise the reliability of latency estimates and affect the coverage of the DNN design space.

The following sections address the identified challenges in estimating latency for DNN hardware. Section III-A provides an overview of the model generation process, highlighting the additions to the latency estimation framework. To tackle issue (1), Section III-B presents a flexible methodology that allows us to profile hardware platforms based on a minimal requirement on the available hardware insights and profiling possibilities. Lastly, to address the diverse latency prediction model quality (2), in Section III-C we propose the application of conformal prediction methods as measures for the confidence of the per-layer and per-network estimation.

A. OVERVIEW

Figure 1 depicts the usual stages to compile a trained Neural Network (NN) for hardware inference:

- The trained **DNN model** is exported from a training framework such as Tensorflow or Pytorch to an intermediate exchange format (e.g. ONNX, TFlite).
- **Backend independent optimizations** are applied to optimize the graph for inference. These can include removing layers from the graph that are only required for training (e.g. Dropout), or fusing layers while still maintaining mathematical equivalency (e.g. Batch Normalization). While most inference frameworks apply this step automatically, it is still recommended to make use of tools such as NVIDIA's ONNX-GraphSurgeon [1] or ONNX-simplifier [2] in a separate step. Hence, this step can similarly be applied in the latency estimation flow.
- **Backend dependent optimizations** represent the changes applied to the DNN model, that are either required or beneficial with regard to latency and/or efficiency, and which are not executable on all hardware platforms. Since each hardware platform provides a different set of operations and possibly allows for multiple operations in a pipelined manner (*composite layers*) to reduce data transfer these optimizations need to be considered in the estimation framework [5], [8].
- Lastly, the model is compiled and executed on the hardware platform using the hardware-specific inference **backend**. Some compilers provide different optimization targets (e.g. latency, memory) or optimize the workload for a specific hardware setting. It has to be considered that, with the current methods, each prediction model can only provide the predictions for one specific combination of compiler and hardware settings.

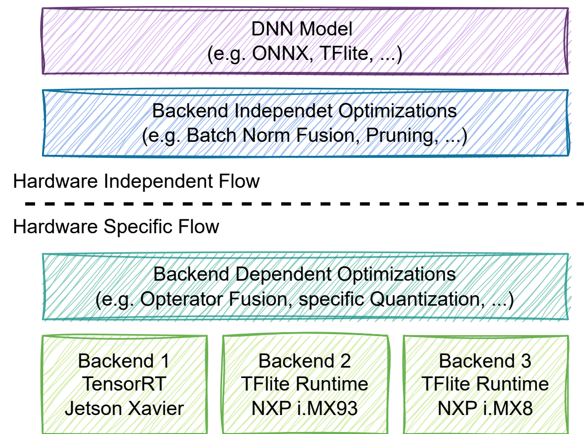

FIGURE 1. Overview of the compilation flow for inference on embedded hardware platforms.

From the inference workflow, there are different levels of insights that can be gathered and used for the latency estimation framework:

- DNN graph before and after backend-dependent optimization
- Per layer latencies
- Overall network latency

Ideally, these insights not only include the hardware mapping of the computational graph but also precise timing for each layer. This would allow for the development of accurate

[1] https://docs.nvidia.com/deeplearning/tensorrt/onnx-graphsurgeon
[2] https://github.com/daquexian/onnx-simplifier

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2024.3439850

latency estimation models and the identification of further optimizations, such as combining individual layers into *composite layers*. In this context, *Composite layers* refer to the fusion of multiple neural network operations (e.g., Conv2D + ReLU + MaxPool) into a single operation executed as one unit, enhancing processing efficiency and reducing latency.

However, the level of detail available in profiling data can vary significantly across different hardware platforms. Some allow for more detailed analysis than others. Additionally, the generation of per-layer reports can also lead to additional overhead resulting in inaccurate latency measurements. In cases where direct profiling at this level is not feasible, alternative methods, like employing GNNs for overall network latency estimation or block-wise estimation [15], have been explored. However, these approaches do not provide insights at the layer-level and are limited in their coverage of the design space, as they cannot account for all possible blocks and network configurations in the training dataset. The experiments conducted in this study demonstrate that simply benchmarking each layer type through single-layer measurements (profiling NNs consisting of only one layer) does not yield the required level of measurement accuracy. This is due to the overhead associated with data transfer at the start and end of the execution.

While each hardware architecture presents unique complexities, the smart padding method introduced in this work enables benchmarking across a wide array of current hardware platforms. This technique accounts for the data transfer overhead during the measurement process, thus isolating the actual computation time within a DNN with multiple layers. However, it operates under the assumption that the hardware platform performs computations on a per-layer basis. This assumption aligns with the operational characteristics of most modern hardware architectures, where parallel execution of layers typically does not yield substantial performance gains.

As an example, when taking a closer look at the block diagram of an ARM Ethos Neural Processing Unit (NPU) (see Figure 2) and the attached main components of the NXP i.MX93 (Main CPU and DDR Memory), we can gain insight into the underlying cause of this overhead. During the computation of the NN the intermediate feature maps are stored in a shared buffer, which is tightly coupled to the compute units. This setup enables fast data transfer and optimal compute efficiency. However, at the start and end of the NN inference, data must be transferred via the AXI-bus into or from this shared buffer. Additionally, potential data reordering or similar steps can further impede the speed of this process.

Considering the benchmarked hardware as a black-box, without in-depth knowledge of the specific relationships between the amount of transferred or processed data and the resulting latency, the developed methodology therefore needs to be able to account for this overhead. Furthermore, the implemented confidence metrics should reflect the added estimation uncertainty stemming from the black-box approach.

For this work, we build on Accurate Neural Network Exectution Time Estimation (ANNETTE) [5], an open-source framework for NN latency estimation on embedded hardware platforms. Figure 3 provides an overview of the modules of ANNETTE and the components that are added for this work.

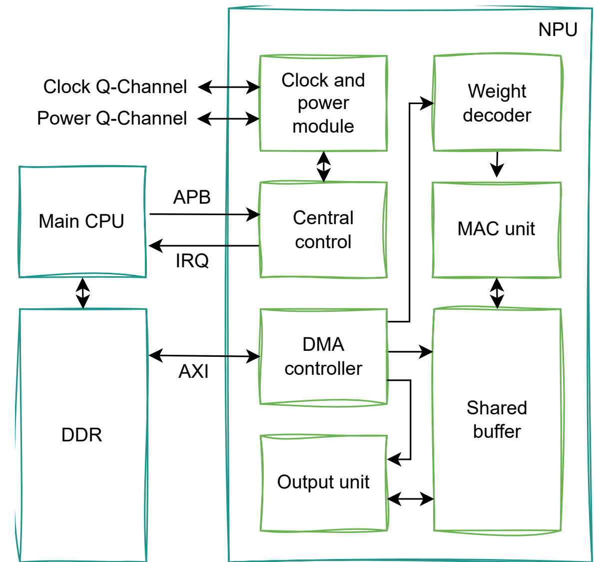

FIGURE 2. Blockdiagram of the ARM Ethos NPU [36] and connections to the main CPU and DDR.

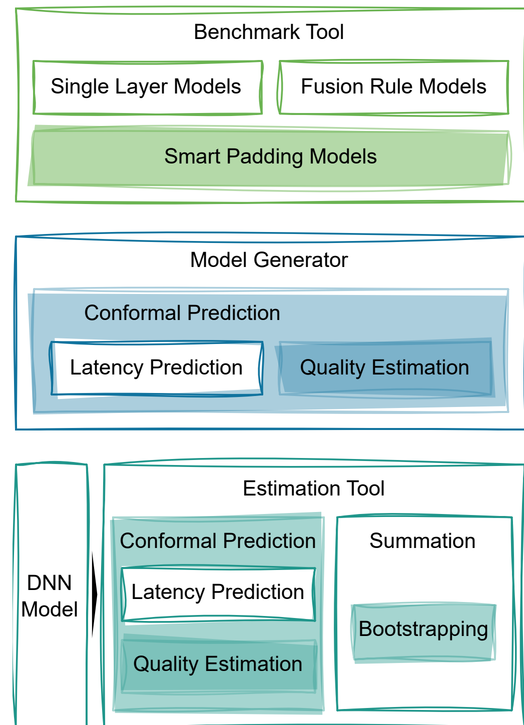

FIGURE 3. Overview of the Components in ANNETTE. The color-shaded components are added in this work.

The ANNETTE workflow comprises two phases: the characterization phase and the estimation phase. Initially, in the characterization phase, **Benchmark Tool** (Fig. 3) executes the benchmarks on the hardware, by autonomously measuring the latencies for a set of parametric dummy network models

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2024.3439850

TABLE 1. Definitions of Time-Related Symbols

Symbol	Definition
t	Latency
T	Latency Interval
$t_{\text{data_in}}$ / $t_{\text{data_out}}$	Data transfer latency
t_{comp}	Computation latency
LUM	Layer Under Measurement
$\hat{t}$	Predicted Latency
$\hat{T}$	Predicted Latency Interval
$\mathcal{T}$	Measured Latency

and stores the results in a data frame. Subsequently, the **Model Generator** utilizes this data to generate prediction models for the assessed layer types and fusion rules. Predominantly, the end user interacts with the **Estimation Tool**, which loads a DNN model description in ONNX format and predicts the latency using the previously generated models. This work relies on the random forest-based estimation models of ANNETTE. However, it is possible to apply the same methodology to other latency estimation frameworks, such as nn-Meter or PerfSAGE since they are compatible with the conformal prediction approach [37].

To facilitate the proposed black-box benchmarking approach, the set of benchmarks is expanded with smart padding models (described in Section III-B). To enable uncertainty quantification for latency prediction, we apply conformal prediction methods to the random forest regression models (see Section III-C). This requires modifications to both the **Model Generator** and the **Estimation Tool** to support the conformal prediction framework. Specifically, the **Model Generator** is extended to include support for training the quality estimators and calculating non-conformity scores. Updates to the **Estimation Tool** enable the inference of conformal prediction, including the quality estimators, and the use of bootstrapping to compute the per-network confidence metrics.

B. BLACK-BOX BENCHMARKING

This section describes the techniques used to achieve per-layer prediction models for hardware with limited profiling capabilities. The notation used in this section is outlined in Table 1.

As mentioned in Section III-A, when measuring individual layers, there is additional overhead due to data transfer times, complicating the accurate assessment of execution times. The measured execution time $\mathcal{T}$ of a NN on hardware that executes layer by layer is determined by the computation time $t_{\text{comp},i}$ per layer, as well as the additional data transfer times $t_{\text{data_in}}$ and $t_{\text{data_out}}$.

$$\mathcal{T} = t_{\text{data_in}} + t_{\text{data_out}} + \sum_{i=1}^{\text{layers}} t_{\text{comp},i} \tag{1}$$

As a result, when benchmarking single-layer models based on the measured latency of the entire model, the estimator will overestimate the execution time of multi-layer models (see Section IV). Figure 4 illustrates a model with three layers. The effects of pipelining result in the overlaps of the compute and actual data read and data write times ($t'_{\text{data_in}}$ and $t'_{\text{data_out}}$) since the compute unit can start computation without having all the data available. While in most cases $t'_{\text{data_in}}$ and $t'_{\text{data_out}}$ are proportional to the amount of data to be transferred, due to the irregular pipelining effects, estimating $t_{\text{data_in}}$ and $t_{\text{data_out}}$ is more complicated and requires a different approach.

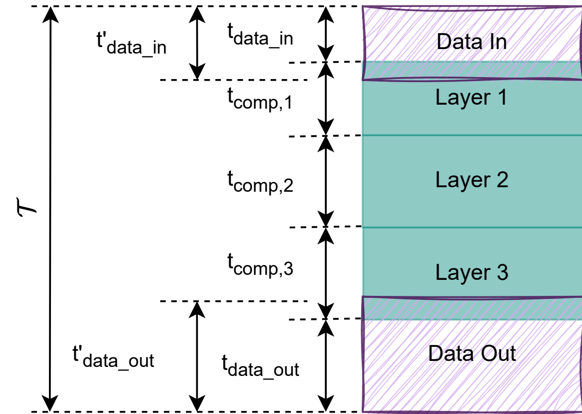

FIGURE 4. Relationship between measured, compute and data transfer times for a DNN with three layers.

At this point, the main challenge lies in disentangling the data read/write times from the computation latency of the Layer Under Measurement (LUM) t_{LUM}. At first glance, this task may seem straightforward; however, the intricate relationship between layer configuration and the dimensions of the resulting input and output feature maps requires a smart approach. To address this issue, we propose a smart padding strategy to measure the computation latency of the LUM within a multi-layer model. Figure 5 depicts the models for the smart padding strategy (Figure 5b,c), alongside a simple single-layer model (Figure 5a). Our strategy is based on two key concepts: firstly, to reduce data transfer times ($t_{\text{data_in}}$ and $t_{\text{data_out}}$) to a bare minimum, thereby mitigating their impact on the latency measurements. Secondly, we independently measure the execution latency of a padding-only model. This enables the calculation of T_{LUM} by subtracting the times of the padding-only models (Figure 5b) from the padded layer model (Figure 5c).

According to Equation 1, the measured latency $\mathcal{T}_a$ for the single layer model includes the data transfer times ($t_{\text{data_in}}$, $t_{\text{data_out}}$) and the actual computation time t_{LUM}. The padded model consists of the LUM padded by an input and output padding 1×1 2D convolution layer with c_{in}=1 input channels, for the input padding layer and c_{out}=1 output channels for the output padding layer. However, to calculate t_{LUM} accurately, those padding layers also need to be benchmarked separately. To minimize the error when calculating the latency of the LUM, we construct those padding-only models by pairing input and output padding convolution layers with matching dimensions. Therefore, each measured padding-only model consists of two convolution layers with the same number

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2024.3439850

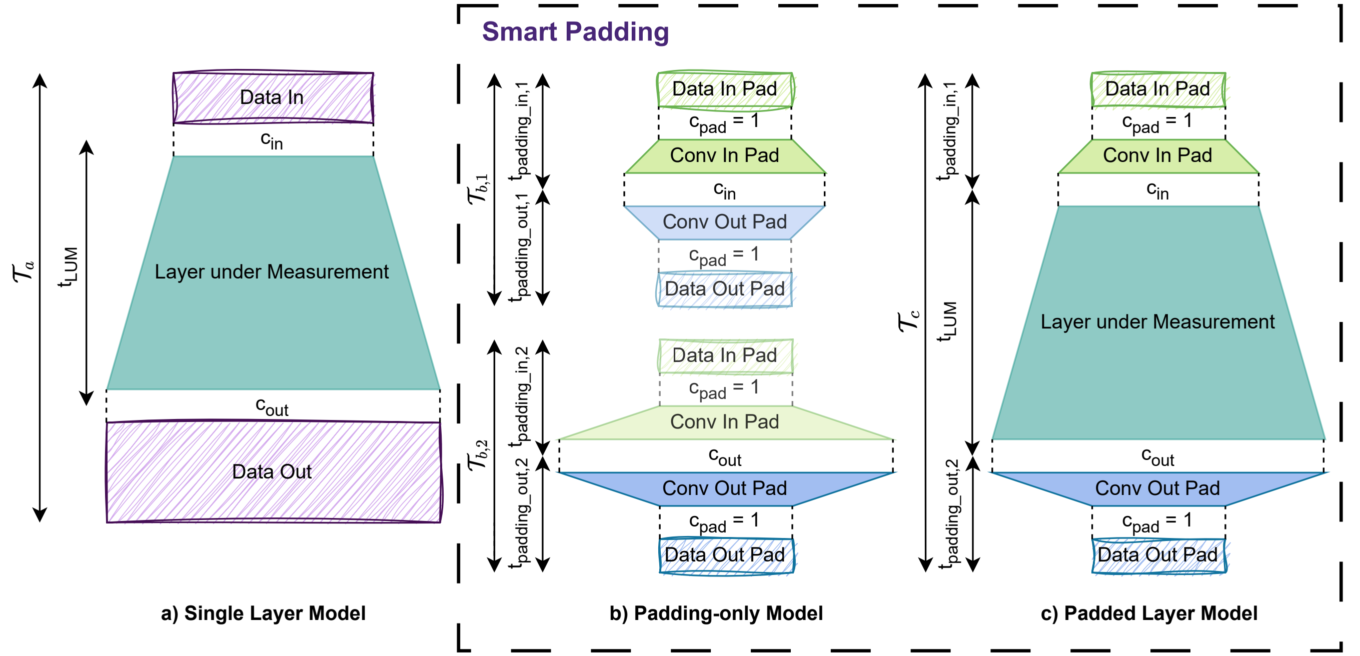

FIGURE 5. The three models used for benchmarking the different platforms. The Single Layer Model (a) is the simplest way but does not provide accurate measurements of t_{LUM}. The black-box benchmarking method makes use of padding-only (b) and padded layer models (c) to solve this problem.

Algorithm 1 Smart Padding for Latency Benchmarking

Initialize look-up table for padding-only models
for each required combination of padding-only model **do**
  Construct the padding-only models as in Fig. 5b
  Measure total latency $\mathcal{T}_{b,1}$
  Measure total latency $\mathcal{T}_{b,2}$
  Store $\mathcal{T}_{b,1}$ and $\mathcal{T}_{b,2}$ in the look-up table
end for
for each LUM to be measured **do**
  Construct a padded layer model as in Fig. 5c
  Measure total latency $\mathcal{T}_c$
  Load correct $\mathcal{T}_{b,1}$ and $\mathcal{T}_{b,2}$ from look-up table
  Compute T_{LUM} using the Equation 4
end for

of operations and equal data input and output dimensions. Equations 2 and 3 describe $\mathcal{T}_{b,n}$ and $\mathcal{T}_c$.

$$\mathcal{T}_{b,n} = t_{\text{padding_in},n} + t_{\text{padding_out},n} \tag{2}$$

$$\mathcal{T}_c = t_{\text{padding_in},1} + t_{\text{LUM}} + t_{\text{padding_out},2} \tag{3}$$

Padding the LUM with convolutional layers at the input and output offers two major advantages: Firstly, it reduces the amount of input data transfer to a minimum since c_{in} and c_{out} can be set to 1. Consequently, we only need to determine the latency of the padding layers including the data transfer times. Secondly, using padding layers allows us to profile layers with different input and output dimensions (e.g. convolution layers with stride) compared to other solutions such as repeating the same layer multiple times.

The algorithm for computing all T_{LUM} is summarized in Algorithm 1. Firstly, we measure the latency of the padding-only models with configuration sets characterized by the height (h), width (w), and channel dimensions (c_{in} and c_{out}). These measurements allow the creation of an exhaustive look-up table that accounts for any combination of input and output padding dimensions required for the padded layer models. Secondly, the latencies for the padded layer models are measured. However, by using Equations 2 and 3 it is neither possible to determine t_{LUM} nor the distribution between the input and output padding layers ($t_{padding_in,1}$ and $t_{padding_out,2}$).

For layers where the dimensions of the input padding layer and the output padding layer are not identical, without per-layer profiling, the exact numbers for $t_{padding_in,1}$ and $t_{padding_out,2}$ are not obtainable. However, it is possible to compute the upper and lower bound of the latency interval of the LUM with:

$$\begin{aligned} t_{\text{LUM_upper}} &:= \mathcal{T}_c - \min_{n\in\{1,2\}}(\mathcal{T}_{b,n}) \\ t_{\text{LUM_lower}} &:= \mathcal{T}_c - \max_{n\in\{1,2\}}(\mathcal{T}_{b,n}) \\ T_{\text{LUM}} &:= [t_{\text{LUM_lower}}, t_{\text{LUM_upper}}] \end{aligned} \tag{4}$$

Compared to alternative methods, smart padding drastically reduces the interval width of T_{LUM}. This is due to the small number of additional operations and minimized data transfer of the padding layers. For example, when measuring the single layer model of a 2D convolution layer with a stride of 1, the time required for data transfer is proportional to $w \cdot h \cdot (c_{in} + c_{out})$. Here, w and h represent the width and height of the layer, while c_{in} and c_{out} refer to the number of input and

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2024.3439850

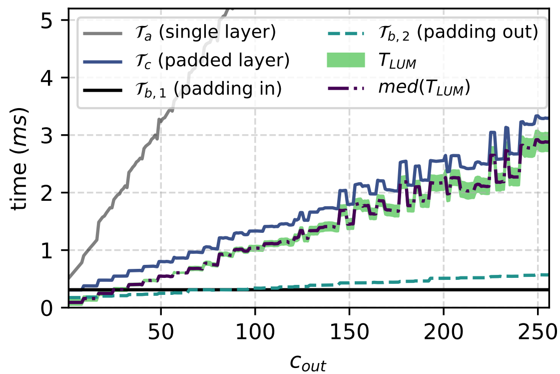

FIGURE 6. **Computed and measured times for the three models of a 2D convolution layer with $c_{in} = 64$, $c_{out} \in [1, 256]$, $h = 64$, $w = 64$ for the i.MX93**

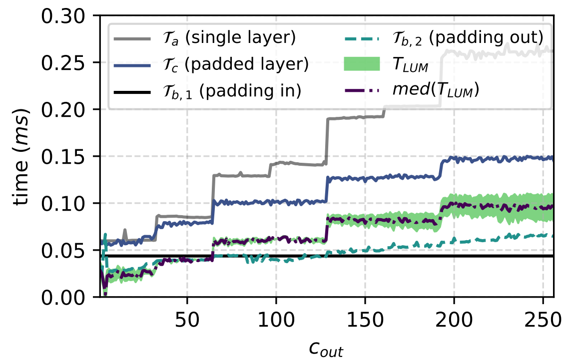

FIGURE 7. **Computed and measured times for the three models of a 2D convolution layer with $c_{in} = 64$, $c_{out} \in [1, 256]$, $h = 64$, $w = 64$ for the Jetson Xavier**

output channels, respectively. The computation time scales with $w \cdot h \cdot c_{\text{in}} \cdot c_{\text{out}} \cdot k_{\text{h}} \cdot k_{\text{w}}$, where k_{h} and k_{w} are the kernel height and width. In contrast, when considering the padded layer model, the data transfer time remains the same for the input and output padding. However, the computation time for the entire model is now determined by the computation time of the LUM and an additional term that accounts for the computation time padding layers.

$$t_{\text{comp_padded}} \propto w \cdot h \cdot c_{\text{in}} \cdot c_{\text{out}} \cdot k_{\text{h}} \cdot k_{\text{w}} + w_{\text{in}} \cdot h_{\text{in}} \cdot c_{\text{in}} + w_{\text{out}} \cdot h_{\text{out}} \cdot c_{\text{out}} \tag{5}$$

This means that the resulting width of the possible latency interval is the difference between $\mathcal{T}_{\text{b},1}$ and $\mathcal{T}_{\text{b},2}$. As a result, there are three major possible outcomes:

1) $c_{\text{in}} = c_{\text{out}}, w_{\text{in}} = w_{\text{out}}, h_{\text{in}} = h_{\text{out}}$: $t_{\text{LUM_upper}} = t_{\text{LUM_lower}}$ as a result of $\mathcal{T}_{\text{b},1} = \mathcal{T}_{\text{b},2}$
2) $c_{\text{in}} \neq c_{\text{out}}, w_{\text{in}} = w_{\text{out}}, h_{\text{in}} = h_{\text{out}}$: The error margin is dominated by the difference in computation time of the input and output padding layers
3) $c_{\text{in}} \neq c_{\text{out}}, w_{\text{in}} \neq w_{\text{out}}, h_{\text{in}} \neq h_{\text{out}}$: The error margin is composed of the difference in computation time and data transfer time of the input and output padding layers

As an example, Figures 6 and 7 depict the computed T_{LUM} for the case 2 ($c_{\text{in}} \neq c_{\text{out}}, w_{\text{in}} = w_{\text{out}}, h_{\text{in}} = h_{\text{out}}$). We note that the computed median and error interval of t_{LUM} are magnitudes smaller than the measured $\mathcal{T}_{\text{a}}$ for the single layer model on the NXP i.MX93 development board (i.MX93) and the NVIDIA Jetson Xavier AGX (Jetson Xavier).

For the final dataset generation, T_{LUM} is computed for each individual padded model measurement alongside the calculated interval. Using this method, we can utilize the smart padding benchmarks for the layer model generation as described in [4], [5], [8]. In general, the decision to use 2D convolution layers (including a non-linearity) as padding layers is motivated by two main factors.

Firstly, unlike when padding with slicing or concatenation operations, it ensures that there is no possibility for the compiler to further simplify the computation graph. Secondly, the same procedure can be applied to 1D and 3D convolutions while still achieving a similar reduction in operations and data transfer. Lastly, based on our understanding, the presented method of smart padding could be applied to other operators that meet those specific requirements.

TABLE 2. **Layer Feature Specifications and Sample Sizes**

Layer	Features
Conv2D + relu	$h, w, c_{\text{in}}, c_{\text{out}}, k_w, k_h$, stride$_h$, stride$_w$, FLOPs, params
DWConv2d + relu	h, w, c, k_w, k_h, stride$_h$, stride$_w$, FLOPs, params
FC	$c_{\text{in}}, c_{\text{out}}$, FLOPs, params
maxpool	h, w, c, k_w, k_h, stride$_h$, stride$_w$
avgpool	h, w, c, k_w, k_h, stride$_h$, stride$_w$
add + relu	h, w, c_{in}
concat	$h, w, c_{\text{in1}}, c_{\text{in2}}$

Table 2 contains a list of all used features for the benchmarked layers. The width and height of the images are limited to 1024 and the kernel sizes (k_{h}, k_{w}) are limited to 11. The resulting parameter space is sub-sampled randomly and then balanced across the FLOPs dimension, to ensure a balanced dataset. The resulting number of data points is around 1000 per layer type.

C. LATENCY ESTIMATION WITH CONFIDENCE

The primary target of latency estimation frameworks is to accurately predict the application of optimization strategies layer execution time of DNNs. However, incorporating confidence metrics into latency prediction frameworks significantly improves their interpretability and practical usefulness. This enhancement not only provides insights into the precision of the predictions but also informs downstream decision-making processes by flagging areas of low certainty. The implementation of confidence metrics should improve the usability of the predictors for two primary applications:

- **Hardware Platform Selection:** Since the modeling process does not work with the same accuracy for each

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2024.3439850

hardware platform, providing a confidence level helps in selecting the most appropriate hardware platform.
- **Network Architecture Comparison:** When comparing DNNs with different layer types and configurations, it is important to understand which layers are outside the distribution of the training datasets and therefore not correctly predicted by the estimator.

Based on these two major use-cases, the *confidence metrics* should demonstrate several key properties at both the layer and accelerator levels. The confidence metric should:

1) Take into account the method of data acquisition, providing insights into the reliability of the data, especially in cases where the black-box measurement method from Section III-B is deployed.
2) Enable comparison of confidence in estimation at the levels of per-layer compute efficiency and per-layer latency.
3) Assess the coverage of the benchmark dataset and identify configurations of layers that are outside of the benchmarked design space.

To implement such confidence metrics, we rely on the conformal prediction framework which offers various options to generate statistically valid prediction regions for any underlying point predictor [25], [26]. As a result, we implement three confidence metrics that enable the comparison of DNNs prediction results and the underlying prediction models, on layer and network level:

- Confidence Metric Throughput Variance (CM_{TV})
- Confidence Metric Latency Variance (CM_{LV})
- Confidence Metric Outliers (CM_{O})

For these confidence metrics, the concepts of quality estimation for conformal prediction are used to estimate different systematic uncertainties in the latency prediction models. Additional confidence metrics could be easily integrated by following the same principles. In this work, the primary emphasis is on the prediction confidence of layer time predictors. Although the prediction of model optimizations performed by the optimization toolchain is also crucial in accurately estimating total network execution times, it is not the central focus of this study. The motivation behind this decision is that the correct prediction of fusion rules represents a simpler challenge than the per-layer latency prediction, due to the limited amount of possible and useful combinations of layers, in comparison to the myriad configurations of each layer type. Nevertheless, the presented concepts have the potential to be applied to the model optimization predictors in future work. Furthermore, the following methodology requires that all occurring layer types within the investigated networks are benchmarked and modeled with the statistical method of ANNETTE.

a: Inference

Figure 8 depicts an overview of the confidence estimation extension for ANNETTE. The network topology is described by a set of N layers where each layer is described by a feature

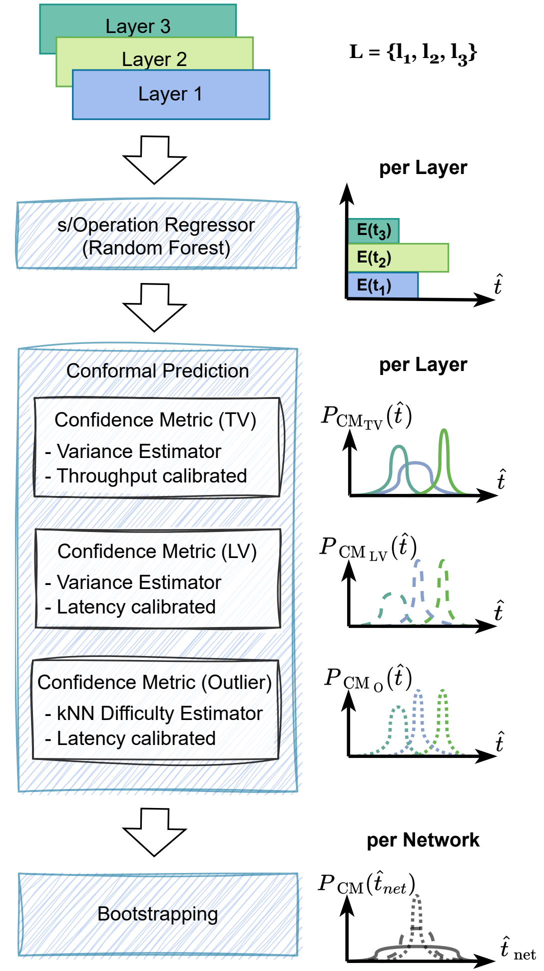

FIGURE 8. Overview of the confidence prediction methodology.

vector $\vec{x}$ which is composed of the configuration parameters describing the layer. Furthermore, $\vec{x}$ also includes additional high-level features such as the number of parameters, number of input features, etc.:

$$L = \{l_1, l_2, l_3 ... l_N\} \text{ where } l_i = \vec{x_i} \tag{6}$$

Based on the measured times for each layer type an individual random forest regressor is trained. For this work, the target value is the time per operation t_{op} instead of the number of operations per second. This adaption is required to avoid potential zero division for broad confidence intervals since the final computation time of a layer t_{comp} is now computed with $t_{comp} = t_{op} \cdot \text{num}_{ops}$. Based on the predictions of the regressors the expected value for the computation time of the entire network is computed as the sum of the predicted computation times of all layers:

$$\hat{t}_{net} = \sum_{i}^{N} \text{num}_{ops,i} \cdot \hat{t}_{op,i} \tag{7}$$

Consequently, from a probabilistic perspective if $P_{CM}(\hat{t}_i)$ with $i \in \{1, 2, ..., N\}$ are the probability distributions for all

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2024.3439850

layers of the network computed by the three different conformal interval predictors, the probability distribution for the total computation time for each interval predictor is computed by the convolution of the probability distributions. With $*$ as the notation of the convolution operator this results in the Equation:

$$P_{\mathrm{CM}}(\hat{t}_{\mathrm{net}}) = P_{\mathrm{CM}}(\hat{t}_1) * P_{\mathrm{CM}}(\hat{t}_2) * \ldots * P_{\mathrm{CM}}(\hat{t}_N) \quad (8)$$

To ensure that $P_{\mathrm{CM}}(\hat{t}_{\mathrm{net}})$ is computed correctly in all cases, we apply bootstrapping. This helps overcome limitations in the case that only a few data points are used in the calibration step for the uncertainty quantification.

b: Training

For the training of the conformal regressors this work relies on the techniques implemented in CREPES [38] a Python package for generating conformal regressors and predictive systems. To ensure the robustness of our latency prediction models, the predictors are specifically trained on the median of the measured times, focusing on the quantification of the predictor's uncertainty rather than variations in latency for the same network. For each trained regressor CREPES provides a multitude of methods for the generation of confidence intervals. Firstly, to avoid splitting the training data into calibration and proper training dataset, we apply out-of-bag calibration. In contrast to standard non-normalized conformal regressors, which predict constant confidence intervals for all instances, normalized conformal regressors produce instance-specific confidence intervals based on difficulty estimates.

As mentioned in Section II there are several ways to perform the difficulty estimate. For $\mathrm{CM_{TV}}$ and $\mathrm{CM_{LV}}$, variance-based difficulty estimation is applied. For $\mathrm{CM_O}$, k-nearest neighbors (k-NN)-based difficulty estimation is used. Additionally, while the difficulty estimation in $\mathrm{CM_{LV}}$ is calibrated based on the absolute prediction error of the layer latency, for $\mathrm{CM_{TV}}$ it is calibrated based on the absolute prediction error of the layer efficiency (s/operation). The difficulty estimation for $\mathrm{CM_O}$ is solely based on the feature vectors $\vec{x}$ of the calibration data.

The effects of applying the three different normalization methods are depicted in Figure 9, which shows the 95% confidence intervals around the predicted value for the measurements performed on the Jetson Xavier from Section III-B Figure 7.

The confidence interval for $\mathrm{CM_{TV}}$ is depicted in Figure 9a. Since the confidence interval estimation is calibrated via the absolute error of the time per operation, the resulting confidence intervals increase with the number of operations. Hence, this $\mathrm{CM_{TV}}$ is more useful when comparing the prediction quality of the compute efficiency rather than the overall layer execution time.

To address this limitation, we introduce $\mathrm{CM_{LV}}$. For this measure, the confidence intervals are computed based on the residuals of the computed layer execution time (see Figure 9b). It is worth noting that the confidence intervals for this measure closely align with the error interval extracted earlier, as shown in Figure 7. As a result, this $\mathrm{CM_{LV}}$ is most useful for comparing the confidence intervals of the overall layer execution times.

For $\mathrm{CM_O}$ (Fig. 9c), it can be observed that the width of the confidence intervals increase towards the boundary values of c_{out} within the example dataset. This is because the distance to the k-Nearest Neighbor data points increases for predictions in those regions. This indicates a sparse local coverage by the benchmark data, which may compromise the prediction accuracy. Thus, $\mathrm{CM_O}$ serves as a tool to pinpoint predictions for layers with feature vectors that are not well covered by the training dataset. For feature vectors far beyond the dataset's scope, the resulting confidence intervals might extend to negative values. However, as it is unrealistic for a layer to be computed in negative time, such wide confidence intervals should rather be viewed as indicators of underrepresented areas in the dataset than as precise latency ranges.

IV. RESULTS

For the evaluation of the methodology presented in Section III we conduct a series of experiments. First, we compare the smart padding (see Section III-B) benchmarking method with padded models to simple single-layer benchmarking in terms of overall prediction quality. Secondly, to evaluate the confidence prediction method, we perform a series of experiments to determine if the desired properties listed in Section III-C are met. The experiments include the results for three different hardware platforms: the NVIDIA Jetson Xavier AGX, NXP i.MX 93, and NXP i.MX8M+ development board (i.MX8M+). The Jetson Xavier was operated at maximum power setting with TensorRT as the inference runtime, using integer 8-bit precision and offering 22 TOPs, not considering the Deep Learning Accelerator (DLA) cores. The i.MX93 utilized TensorFlow Lite with 8-bit quantization and the TensorFlow Lite inference runtime delegate, providing up to 1 TOPS using the ARM Ethos U65 microNPU. The i.MX8M+ employed the VeriSilicon VIP9000 NPU, delivering up to 2.3 TOPS also using the TensorFlow Lite inference runtime.

A. BLACK-BOX BENCHMARKING

The goal of the following experiments is to compare the quality of the collected smart padding benchmark data with the single-layer benchmark data and assess how well they serve as ground-truth data for prediction models. For the presented results, we generate ANNETTE prediction models using both the single-layer and smart padding methods. These generated prediction models are then compared in terms of total network latency against the measured network latencies. Additionally, we compare the results to the predictions provided by the ARM Vela compiler [3] for the Ethos U65 NPU on the i.MX93.

Table 3 shows the prediction accuracy for a set of state-of-the-art DNNs for the i.MX93. In the case of the i.MX93, the

[3] https://pypi.org/project/ethos-u-vela

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2024.3439850

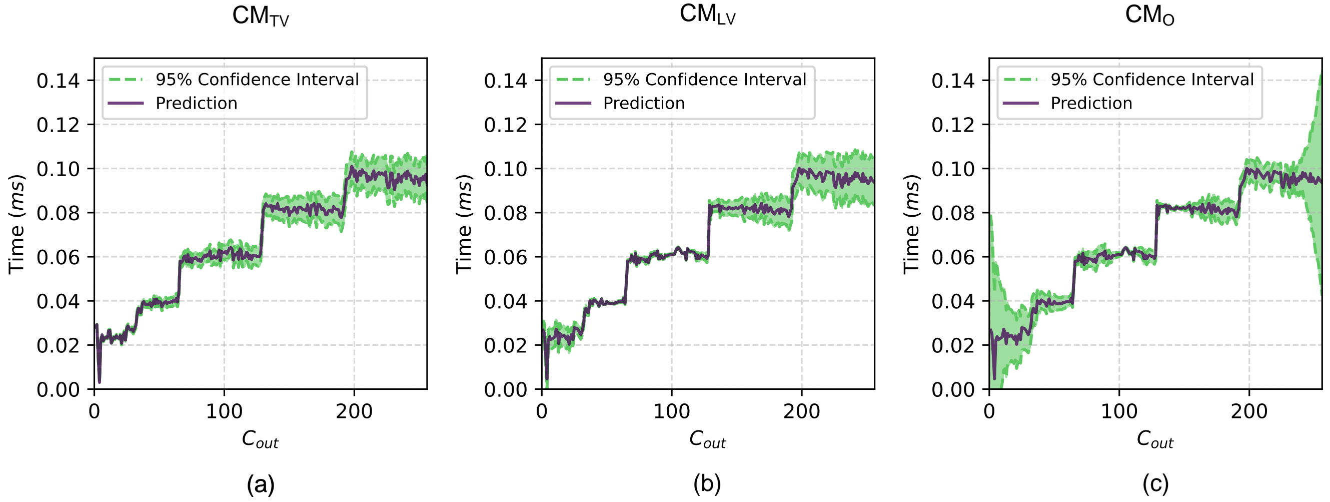

FIGURE 9. Overview of predicted Confidence Intervals for 2D convolution layer with $c_{in} = 64$, $c_{out} \in [1, 256]$, $h = 64$, $w = 64$ on the Jetson Xavier: (a) Confidence s/Operations Normalized, (b) Confidence Normalized with respect to time, (c) Confidence with k-NN difficulty estimation

Network	Measured [ms]	Vela Compiler	Single Layer	Smart Padding
YOLOv5s	103.6	+30.5%	+292.5%	**-9.5%**
YOLOv5m	213.0	+21.1%	+260.8%	**-3.6%**
YOLOv5l	383.1	+22.2%	+233.2%	**-0.8%**
YOLOv8n	67.6	+104.1%	+210.6%	**-3.9%**
YOLOv8s	138.9	+83.0%	+192.4%	**-5.1%**
YOLOv8m	294.8	+64.6%	+172.5%	**-1.3%**
YOLOv8l	486.0	+77.4%	+200.6%	**+5.8%**
YOLOv8x	763.8	+56.1%	+173.6%	**+0.5%**
MobilenetV1	4.97	+78.8%	+759.6%	**+24.6%**
InceptionV4	66.4	+40.5%	+492.4%	**-1.5%**
Avg. error		57.8%	298.8%	**5.7%**

TABLE 3. Percentage prediction errors for the ANNETTE models for the i.MX93 in comparison to the Vela compiler estimates

Network	Measured [ms]	Single Layer	Smart Padding
YOLOv5s	87.3	+96.7%	**-6.6%**
YOLOv5m	165.3	+88.4%	**-5.7%**
YOLOv5l	260.1	+84.8%	**-3.2%**
YOLOv8n	54.4	+56.2%	**-2.5%**
YOLOv8s	100.9	+43.7%	**-2.5%**
YOLOv8m	186.4	+37.1%	**-6.2%**
YOLOv8l	286.1	+36.3%	**-4.7%**
YOLOv8x	363.0	+36.6%	**-7.3%**
MobilenetV1	3.69	+305.2%	**+2.4%**
InceptionV4	63.2	+110.8%	**-51.7%**
Avg. abs. error		89.6%	**9.3%**

TABLE 5. Percentage prediction errors for the ANNETTE models for the i.MX8M+

Network	Measured [ms]	Per-Layer Profiling	Single Layer	Smart Padding
YOLOv5s	4.6	**+0.1%**	+55.3%	-2.1%
YOLOv5m	9.4	**+0.0%**	+65.9%	-4.8%
YOLOv5l	14.2	+12.1%	+79.1%	**-7.4%**
YOLOv8n	5.5	-21.9%	**+17.9%**	-18.3%
YOLOv8s	7.29	**-2.5%**	+53.1%	-3.1%
YOLOv8m	13.5	**-3.6%**	+64.3%	-6.5%
YOLOv8l	19	+9.9%	+80.5%	**+4.2%**
YOLOv8x	28.9	-1.6%	+53.9%	**-9.7%**
MobilenetV1	0.45	-17.5%	+59.3%	**-6.1%**
InceptionV4	4.82	-14.4%	+116.2%	**+6.0%**
Avg. abs. error		8.4%	64.6%	**6.9%**

TABLE 4. Percentage prediction errors for the ANNETTE models for the Jetson Xavier in comparison to the ANNETTE model based on the per-layer profiling

smart padding-based ANNETTE prediction demonstrates superior performance compared to the single-layer ANNETTE prediction and the Vela estimates, achieving higher prediction accuracy across all networks. The average prediction errors for the smart padding-based ANNETTE prediction, single layer-based ANNETTE prediction, and Vela estimates are 5.7%, 298.8%, and 57.8% respectively. Further in-depth analysis revealed that benchmarking individual layers on the i.MX93 results in additional time overhead due to an extra quantization step. This leads to a more substantial improvement than expected, thanks to the smart padding method. Likewise, for the smart padding-based and single layer-based ANNETTE prediction, the average percentage errors are 6.9% and 64.6% for the Jetson Xavier, and 9.2% and 89.6% for the i.MX8M+. The detailed results for the Jetson Xavier and i.MX8M+ are shown in Tables 4 and 5, respectively.

Notably, only in 3 instances, the smart padding-based ANNETTE estimation errors are larger than 10%. These errors can be explained by the limited dataset used for this work, which does not cover a stride different than 1 and asymmetric convolution kernels. These limitations result in not optimal prediction results for InceptionV4, MobilenetV1, and YOLOv8n but also allow us to evaluate the confidence metrics.

To compare the accuracy of the smart padding strategy with

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2024.3439850

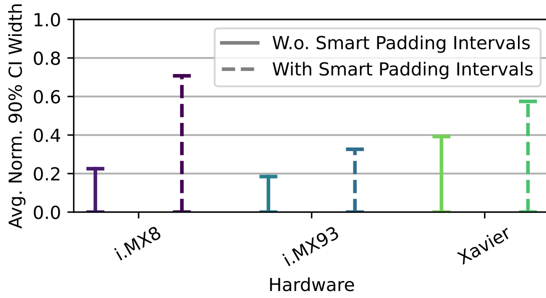

FIGURE 10. Average normalized 90% confidence intervals for CM_{TV} on the all tested networks

per-layer profiling, we utilized the built-in per-layer profiling from the Nvidia benchmarking script for the Jetson Xavier. The Pearson correlation coefficient between the Nvidia per-layer profiling and the computed mean value of T_{LUM} was found to be 0.975. The ANNETTE predictions based on per-layer profiling data for the Jetson Xavier showed similar accuracy to those from the smart padding-based ANNETTE models. Therefore, we conclude that smart padding benchmarking provides profiling accuracy comparable to per-layer benchmarking, while significantly reducing implementation efforts. The overhead associated with the smart padding benchmarking strategy is limited to measuring 854 padding-only models.

B. CONFIDENCE METRICS

For the evaluation of the *confidence metrics*, we display the results on model, network, and layer levels. Firstly, since CM_{TV} is throughput calibrated, it mostly serves to compare the normalized per-layer confidence interval size for different models. This can, for example, be used to compare the overall confidence of the previously computed models.

a: Model-Level Comparison

Figure 10 displays the average normalized 90% confidence interval size for the generated models for all tested networks. To evaluate the influence of the smart padding method on the generated latency prediction models, we also generate models based on the mean value without including the previously computed intervals (see Section III-B).

As outlined in Section III-B, the CM_{TV}, which is used for this comparison, is calibrated with regard to the layer throughput. As a result, this metric provides a measure for comparing the confidence for the compute efficiency predictions across the tested hardware platforms and all tested layers. As expected, including the smart padding intervals in the calibration of the confidence metrics leads to larger confidence intervals. Notably, the increase of the confidence interval widths differs for the different hardware platforms. We conclude that CM_{TV} can be used to determine which hardware platforms would profit the most from implementing per-layer profiling and for which hardware platforms, the smart padding method is sufficient. Furthermore, CM_{TV} can

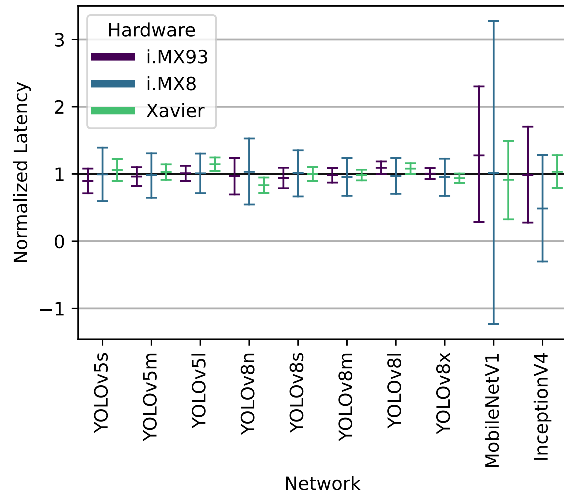

FIGURE 11. *CM_{LV} for the tested networks.*

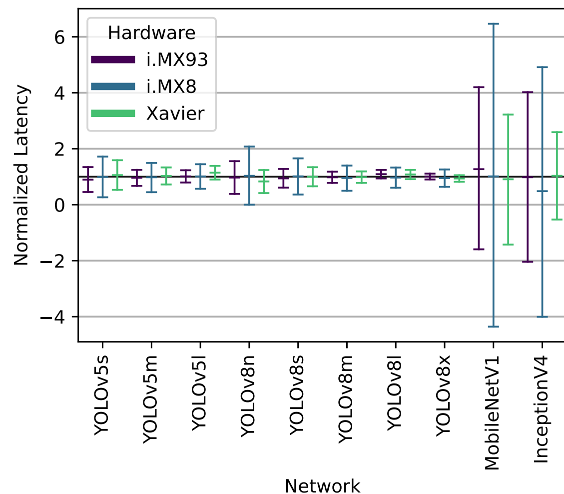

FIGURE 12. *CM_{O} for the tested networks.*

guide engineers in situations where a platform may show consistently high throughput variance across different network layers. By using CM_{TV}, engineers can prioritize hardware that demonstrates lower throughput variance, suggesting more stable performance across diverse workloads.

b: Network-Level Comparison

For the network-level comparison, the 90% confidence intervals of CM_{LV} and CM_{O} are displayed for all networks in Figure 11 and 12 respectively. As mentioned in Section IV-A, these confidence metrics provide a deeper understanding of the predictions performed for each individual network. It is noticeable that the confidence intervals for MobilenetV1 and InceptionV4 are particularly large, which aligns with the occurrence of inaccurate predictions in certain cases. A large confidence interval for CM_{LV} indicates sub-optimal prediction accuracy due to high variances in the dataset within the

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2024.3439850

prediction region. To address this, engineers can use CM_{LV} as an indicator to refine the training datasets by incorporating more diverse network configurations that mirror the operational settings. This can help in reducing the confidence intervals and thus improving the accuracy of latency predictions Conversely, a large confidence interval for CM_O suggests inadequate coverage of one or more layers in the collected dataset, potentially leading to inaccurate prediction results. In this case, we can go one step further and analyze the prediction results on a layer level.

c: Layer-Level Comparison

CM_{LV} and CM_O provide insights into the root causes of potentially inaccurate prediction on a layer level. Figure 13 displays the confidence interval widths for the latency predictions of YOLOv8n on the Jetson Xavier. The layers with large confidence interval widths almost exclusively have a stride of 2 which is not covered well in the example benchmark dataset. This can be detected by the large confidence interval widths of the CM_O for those specific layers.

On the other hand, the CM_{LV} interval widths for hint layers 1 and 4 at prediction regions with high variance of measured latencies. However, compared with the per-network CM_{LV} and CM_O, we see that the prediction confidence for InceptionV4 for the Jetson Xavier could be improved by extending the dataset so that the required layer configurations are well covered.

For example, in a hardware-aware NAS process, the CM_O can prevent the NAS from settling on a seemingly optimal architecture that performs poorly in untested conditions. Meanwhile, CM_{TV} and CM_{LV} ensure the chosen architecture consistently meets performance expectations across a range of architecture variations, thereby avoiding costly misestimations of network efficiency

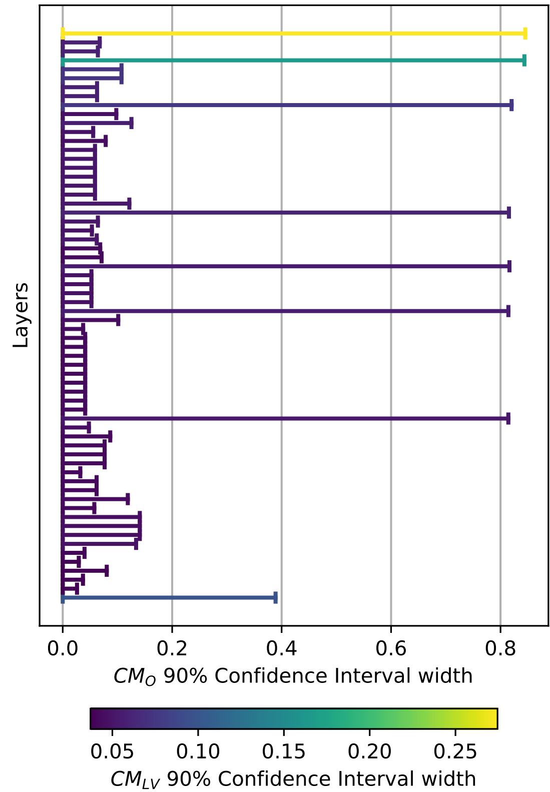

FIGURE 13. YOLOv8n per-layer confidence interval widths of CM_{LV} and CM_O for predictions for the Jetson Xavier

V. CONCLUSION

This study introduces a novel approach for benchmarking DNN accelerators that eliminates the need for per-layer profiling for existing latency estimation frameworks. As a result, the setup for benchmarking new hardware is simplified, and the potential profiling overhead can be eliminated. The experiments underscore the method's effectiveness across three distinct hardware platforms (Jetson Xavier, i.MX8M+ and i.MX93), improving the latency prediction accuracy by a large margin in comparison to single-layer benchmarking and outperforming the latency prediction of the ARM Vela compiler. Furthermore, this study integrates three confidence metrics to improve the usability and interpretability of latency prediction frameworks.

From the perspective of developers, the introduction of smart padding not only decreases the implementation effort when benchmarking new hardware platforms but also allows benchmarking without profiling overhead. Furthermore, the adoption of our confidence framework has already yielded significant insights into the prediction models for certain hardware platforms. With the guidance of the confidence metrics, we were able to precisely identify and correct inaccuracies in layer-specific predictions.

For end-users, the introduced confidence metrics offer a more informed basis for selecting hardware and network models for DNN deployment.

Future research could further refine the smart padding method by exploring its application across diverse network layers and other domains like time series and 3D modeling. Expanding the use of conformal prediction methods to include GNN based latency prediction methods and developing automated benchmark point selection based on confidence levels are also promising directions.

REFERENCES

[1] Han Cai, Ligeng Zhu, and Song Han. ProxylessNAS: Direct neural architecture search on target task and hardware. In *ICLR*, 2019.
[2] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V. Le. Mnasnet: Platform-aware neural architecture search for mobile. In *CVPR*, pages 2820–2828, 2019.
[3] Andrew Anderson, Jing Su, Rozenn Dahyot, and David Gregg. Performance-oriented neural architecture search. In *HPCS*, pages 177–184. IEEE, 2019.
[4] Martin Lechner and A. Jantsch. Blackthorn: Latency estimation framework for cnns on embedded nvidia platforms. *IEEE Access*, 9:110074–110084, 2021.
[5] M. Wess, Matvey Ivanov, C. Unger, A. Nookala, A. Wendt, and A. Jantsch. Annette: Accurate neural network execution time estimation with stacked models. *IEEE Access*, 9:3545–3556, 2021.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2024.3439850

[6] Zhenyi Wang, Pengfei Yang, Linwei Hu, Bowen Zhang, Chengmin Lin, Wenkai Lv, and Quan Wang. Slapp: Subgraph-level attention-based performance prediction for deep learning models. *Neural Networks*, 170:285–297, February 2024.

[7] Yuji Chai, Devashree Tripathy, Chu Zhou, Dibakar Gope, Igor Fedorov, Ramon Matas, D. Brooks, Gu-Yeon Wei, and P. Whatmough. Perfsage: Generalized inference performance predictor for arbitrary deep learning models on edge devices. *CoRR*, 2023.

[8] Li Lyna Zhang, Shihao Han, Jianyu Wei, Ningxin Zheng, Ting Cao, Yuqing Yang, and Yunxin Liu. nn-meter: towards accurate latency prediction of deep-learning model inference on diverse edge devices. In *MobiSys*, MobiSys '21. ACM, June 2021.

[9] Muhammad Sabih, Ashutosh Mishra, Frank Hannig, and Jürgen Teich. MOSP: multi-objective sensitivity pruning of deep neural networks. In *IGSC*, pages 1–8. IEEE, 2022.

[10] Zailong Chen, Chubo Liu, Wangdong Yang, Kenli Li, and Keqin Li. Lap: Latency-aware automated pruning with dynamic-based filter selection. *Neural Networks*, 152:407–418, August 2022.

[11] Yuzhe Xu, Thaha Mohammed, Mario Di Francesco, and Carlo Fischione. Distributed assignment with load balancing for dnn inference at the edge. *JIOT*, 10(2):1053–1065, January 2023.

[12] Yubin Duan and Jie Wu. Optimizing job offloading schedule for collaborative dnn inference. *TMC*, 23(4):3436–3451, April 2024.

[13] Yanjie Gao, Xi Gu, Hongyu Zhang, Haoxiang Lin, and Mao Yang. Runtime performance prediction for deep learning models with graph neural network. *ICSE-SEIP*, May 2023.

[14] Thomas C. P. Chau, L. Dudziak, M. Abdelfattah, Royson Lee, Hyeji Kim, and N. Lane. Brp-nas: Prediction-based nas using gcns. *CoRR*, July 2020.

[15] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. Once-for-all: Train one network and specialize it for efficient deployment. In *ICLR*. OpenReview.net, 2020.

[16] Hayeon Lee, Sewoong Lee, S. Chong, and S. Hwang. Help: Hardware-adaptive efficient latency predictor for nas via meta-learning. 2021.

[17] Konstantin Lübeck, Alexander Louis-Ferdinand Jung, Felix Wedlich, and Oliver Bringmann. Work-in-progress: Ultra-fast yet accurate performance prediction for deep neural network accelerators. In *CASES*, pages 27–28. IEEE, 2022.

[18] Linyan Mei, Huichu Liu, Tony Wu, H. Ekin Sumbul, Marian Verhelst, and Edith Beigne. A uniform latency model for dnn accelerators with diverse architectures and dataflows. In *DATE*. IEEE, March 2022.

[19] Jinyang Li, Runyu Ma, Vikram Sharma Mailthody, Colin Samplawski, Benjamin Marlin, Songqing Chen, Shuochao Yao, and Tarek Abdelzaher. Towards an accurate latency model for convolutional neural network layers on gpus. In *MILCOM*. IEEE, 2021.

[20] Saeejith Nair, Saad Abbasi, Alexander Wong, and Mohammad Javad Shafiee. Maple-edge: A runtime latency predictor for edge devices, 2022.

[21] Saad Abbasi, Alexander Wong, and M. Shafiee. Maple-x: Latency prediction with explicit microprocessor prior knowledge. *CoRR*, 2022.

[22] Yan Li, Junming Ma, Donggang Cao, and Hong Mei. Sectum: Accurate latency prediction for tee-hosted deep learning inference. In *ICDCS*. IEEE, 2022.

[23] Keith G. Mills, Fred X. Han, Jialin Zhang, Fabian Chudak, Ali Safari Mamaghani, Mohammad Salameh, Wei Lu, Shangling Jui, and Di Niu. Gennape: towards generalized neural architecture performance estimators. In *AAAI*, AAAI'23/IAAI'23/EAAI'23. AAAI Press, 2023.

[24] Karthick Panner Selvam and Mats Brorsson. *DIPPM: A Deep Learning Inference Performance Predictive Model Using Graph Neural Networks*, pages 3–16. Springer Nature Switzerland, 2023.

[25] Alexander Gammerman, Volodya Vovk, and Vladimir Vapnik. Learning by transduction. In Gregory F. Cooper and Serafín Moral, editors, *UAI*, pages 148–155. Morgan Kaufmann, 1998.

[26] Vladimir Vovk, Alexander Gammerman, and Glenn Shafer. *Algorithmic learning in a random world*, volume 29. Springer, 2005.

[27] Nicolai Meinshausen. Quantile regression forests. *Journal of Machine Learning Research*, 7:983–999, 2006.

[28] Tilmann Gneiting. Quantiles as optimal point forecasts. *International Journal of Forecasting*, 27(2):197–207, April 2011.

[29] Luiz Hespanhol, Caio Sain Vallio, Lucíola Menezes Costa, and Bruno T Saragiotto. Understanding and interpreting confidence and credible intervals around effect estimates. *Brazilian Journal of Physical Therapy*, 23(4):290–301, July 2019.

[30] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[31] Ulf Johansson, Henrik Boström, Tuve Löfström, and Henrik Linusson. Regression conformal prediction with random forests. *Mach. Learn.*, 97(1-2):155–176, 2014.

[32] Henrik Boström, Henrik Linusson, Tuve Löfström, and Ulf Johansson. Accelerating difficulty estimation for conformal regression forests. *Ann. Math. Artif. Intel.*, 81(1–2):125–144, March 2017.

[33] Charles Lu, Andréanne Lemay, Ken Chang, Katharina Höbel, and Jayashree Kalpathy-Cramer. Fair conformal predictors for applications in medical imaging. In *AAAI*, pages 12008–12016. AAAI Press, 2022.

[34] Charalambos Eliades and Harris Papadopoulos. Conformal prediction for automatic face recognition. In Alex Gammerman, Vladimir Vovk, Zhiyuan Luo, and Harris Papadopoulos, editors, *COPA*, volume 60 of *Proceedings of Machine Learning Research*, pages 62–81. PMLR, 2017.

[35] Wojciech Wisniewski, David Lindsay, and Siân Lindsay. Application of conformal prediction interval estimations to market makers' net positions. In Alexander Gammerman, Vladimir Vovk, Zhiyuan Luo, Evgueni N. Smirnov, Giovanni Cherubin, and Marco Christini, editors, *COPA*, volume 128 of *Proceedings of Machine Learning Research*, pages 285–301. PMLR, 2020.

[36] ARM. Arm Ethos NPU Technical Reference Manual. https://developer.arm.com/documentation/102420/0200/Functional-description/Functional-blocks-, 2024. Accessed: 2024-03-01.

[37] Soroush H. Zargarbashi, Simone Antonelli, and Aleksandar Bojchevski. Conformal prediction sets for graph neural networks. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *ICML*, volume 202 of *Proceedings of Machine Learning Research*, pages 12292–12318. PMLR, 2023.

[38] Henrik Boström. crepes: a python package for generating conformal regressors and predictive systems. In Ulf Johansson, Henrik Boström, Khuong An Nguyen, Zhiyuan Luo, and Lars Carlsson, editors, *COPA*, volume 179 of *Proceedings of Machine Learning Research*, pages 24–41. PMLR, 2022.

MATTHIAS WESS received the B.Sc. and M.Sc. degrees from the Department of Electrical Engineering, TU Wien, Vienna, Austria, in 2013 and 2017, respectively, where he is currently pursuing the Ph.D. degree with the Institute for Computer Technology. As a member of the Christian Doppler Laboratory for Embedded Machine Learning, his research is primarily focused on the latency estimation of deep neural networks and enhancing the energy efficiency of machine learning algorithms.

DANIEL SCHNÖLL received the M.Sc. degree in embedded systems at TU Wien, Vienna, Austria, in 2021. He is part of the Christian Doppler Laboratory for Embedded Machine Learning at TU Wien, Austria, where he is currently pursuing a Ph.D. degree with the Institute for Computer Technology. His current research interests include TinyML and optimization of deep neural networks for embedded inference.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2024.3439850

DOMINIK DALLINGER received the Bachelor of Science degree in electrical engineering from TU Wien, Viennam, Austria, in 2021. He is now pursuing a Master's degree in Embedded Systems at TU Wien, with a broad focus on mechatronics, machine vision, computer systems, and electronics design. He is also engaged with the Christian Doppler Laboratory for Embedded Machine Learning, focusing his research on TinyML.

MATTHIAS BITTNER received the M.Sc. degrees in automation and control at TU Wien, Vienna, Austria, and artificial intelligence at Johannes Kepler University, Linz, Austria in 2021 and 2024, respectively. He is affiliated with the Christian Doppler Laboratory for Embedded Machine Learning at TU Wien, where he is pursuing a Ph.D. degree at the Institute for Computer Technology. His research interests include energy-efficient machine learning for time-series applications and leveraging artificial intelligence for sustainability.

AXEL JANTSCH (Senior Member, IEEE) received the Dipl.Ing. degree and the Ph.D. degree in computer science from TU Wien, Vienna, Austria, in 1987 and 1992, respectively. From 1997 to 2002, he was an Associate Professor with KTH Royal Institute of Technology, Stockholm. From 2002 to 2014, he was a Full Professor in electronic systems design at KTH. Since 2014, he has been a Professor of systems on chips with the Institute of Computer Technology, TU Wien. His current research interests include systems on chips, self-aware cyber-physical systems, and embedded machine learning. He has published five books as an editor and one as an author and over 300 peer-reviewed contributions in journals, books, and conference proceedings. He has given over 100 invited presentations at conferences, universities, and companies.

• • •

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4