Dissertation

vorgelegt von

Matthias Wess

Estimation, Profiling and Modeling of DNNs for Embedded Systems

Zur Erlangung des akademischen Grades

Doktor der technischen Wissenschaften (Dr.techn.)

Wien, Austria, März 2025

Betreuer:	Univ.-Prof. Dr. Axel Jantsch
	Technische Universität Wien
Gutachter:	Univ.Prof. Dr.-Ing. Dipl.-Ing. Daniel Müller-Gritschneder
	Technische Universität Wien
Gutachter:	Prof. Mario Casu
	Politecnico di Torino

Contribution to Original Knowledge

- **ANNETTE Framework and Layer-Specific Optimization:** A novel Deep Neural Network (DNN) latency estimation framework, ANNETTE, is introduced, which separates the optimization phase from per-layer estimation. This separation enables more accurate latency estimation and facilitates efficient and targeted improvements to DNN architectures **[Paper I]** [1].
- **Refined Roofline Model and Mixed Statistical Modeling:** A refined roofline model is proposed for latency estimation, combined with mixed modeling techniques to enhance the selection of datapoints for statistical modeling. This method improves the accuracy of latency estimation by strategically choosing relevant datapoints while drastically reducing the number of required latency measurements **[Paper I]** [1].
- **Conformal Prediction for Latency Estimation:** Latency estimation inherently involves approximations, often yielding rough estimates of median or mean latency values rather than predictions for specific execution times due to sparse measurements or high variability in the data. To address these limitations, conformal prediction methods are applied to latency estimation, introducing confidence measures that quantify uncertainties and provide reliable ranges for the latency estimation. This approach enhances the robustness and trustworthiness of the latency estimation process **[Paper II]** [2].
- **Quantization with Weighted Quantization-Regularization (WQR) and Layer-Specific Precision Scaling:** The work adapts regularization-based methods for quantization (WQR), tailoring them to specific quantization functions. It demonstrates how combining these methods with per-layer precision scaling of the numeric format achieves higher compression quality reducing the required memory for execution without compromising accuracy of the DNN algorithm **[Paper III]** [3].
- **Power Profiling Methodology:** A new power profiling methodology is presented, highlighting that not all hardware platforms operate with equal efficiency across different layers. This showcases the variability in energy consumption and helps guide more informed hardware and layer-specific optimizations **[Paper IV]** [4].

List of Original Publications

This dissertation is based on the following original publications, which are referred to in the text by their Roman numerals:

Paper I Matthias Wess, Matvey Ivanov, Christoph Unger, Anvesh Nookala, Alexander Wendt, and Axel Jantsch. *Annette: Accurate neural network execution time estimation with stacked models. IEEE Access*, 9:3545–3556, 2021.

Paper II Matthias Wess, Daniel Schnöll, Dominik Dallinger, Matthias Bittner, and Axel Jantsch. *Conformal prediction based confidence for latency estimation of DNN accelerators: A black-box approach. IEEE Access*, 12:109847–109860, 2024

Paper III Matthias Wess, Sai Manoj Pudukotai Dinakarrao, and Axel Jantsch. *Weighted quantization-regularization in DNNs for weight memory minimization toward HW implementation. IEEE Transactions on Computer Aided Design & Integrated Circuits and Systems*, 37(11):2929–2939, 2018.

Paper IV Matthias Wess, Dominik Dallinger, Daniel Schnöll, Matthias Bittner, Maximilian Götzinger, and Axel Jantsch. *Energy profiling of DNN accelerators. 26th Euromicro Conference on Digital System Design (DSD)*, pages 53–60, 2023.

The following peer-reviewed papers and book chapter (Paper VIII) were already accepted or published in the course of the author's doctoral studies but are not included in this thesis:

Paper V

Matthias Wess, Sai Manoj Pudukotai Dinakarrao and Axel Jantsch. *Neural network based ECG anomaly detection on FPGA and trade-off analysis. IEEE International Symposium on Circuits and Systems*, pages 1–4, 2017.

Author's Contribution: Training, quantization and FPGA implementation of machine learning algorithm

Relevance to this Thesis: Detailed trade-off analysis on the relationship between bit-width, speed-up, and accuracy for scaling weights and features in a practical FPGA implementation

Paper VI

Bernhard Haas, Matthias Wess, Alexander Wendt and Axel Jantsch. *Neural Network Compression Through Shunt Connections and Knowledge Distillation for Semantic Segmentation Problems. IEEE International Symposium on Circuits and Systems*, pages 349–361, 2021.

Author's Contribution: Collaboration and discussions on combining Shunt Connection algorithm in with Knowledge Distillation, model execution time analysis

Relevance to this Thesis: Model compression techniques tailored for embedded platforms

Paper VII

Daniel Schnöll, Matthias Wess, Matthias Bittner, Maximilian Götzinger, and Axel Jantsch, *Fast, quantization-aware DNN training for efficient HW implementation. Euromicro Conference on Digital System Design*, pages 700–707, 2023.

Author's Contribution: Primarily contributed to writing and conceptual formulation, collaborative discussions on Batch Normalization for Quantization-Aware Training

Relevance to this Thesis: Explores hardware-efficient quantization methods for deep learning models

Paper VIII

Alexander Wendt, Horst Possegger, Matthias Bittner, Daniel Schnöll, Matthias Wess, Dušan Malić, Horst Bischof and Axel Jantsch, *A Pedestrian Detection Case Study for a Traffic Light Controller. Embedded Machine Learning for Cyber-Physical, IoT, and Edge Computing*, pages 75–96, 2023.

Author's Contribution: Collaborative discussions related to implementation challenges, provided expertise in inference frameworks and model conversion

Relevance to this Thesis: Real-world use case emphasizing the challenges of testing models on diverse platforms.

Paper IX

Daniel Schnöll, Dominik Dallinger, Matthias Wess, Matthias Bittner, Axel Jantsch, *Towards Optimal Implementations of Neural Networks on Micro-Controller, presented at Workshop on IoT, Edge, and Mobile for Embedded Machine Learning at ECML-PKDD*, 2024.

Author's Contribution: Collaborative discussions related to performance modeling

Relevance to this Thesis: Microcontrollers present a different target and other restrictions. Model Compression and Quantization remain relevant topic

Paper X	Matthias Bittner, Daniel Hauer, Matthias Wess, Dominik Dallinger, Daniel Schnöll, Konrad Diwold and Axel Jantsch, *Interpretable Load Forecasting with Structured State Space Neural Networks, presented at Workshop on Machine Learning for Sustainable Power Systems at ECML-PKDD*, 2024.

Author's Contribution: Research on related work and collaborative discussions

Relevance to this Thesis: Demonstrates the importance of model interpretability, efficient neural networks for forecasting

Paper XI	Matthias Bittner, Daniel Hauer, Matthias Wess, Daniel Schnöll, Konrad Diwold and Axel Jantsch, *Forecasting Load Profiles and Critical Overloads with Uncertainty Quantification for Low Voltage Smart Grids, presented at International Conference on System Reliability and Safety*, 2024.

Author's Contribution: Research on related work and collaborative discussions regarding uncertainty quantification

Relevance to this Thesis: Demonstrates a real-world use case with various uncertainty quantification methods

Paper XII	Axel Jantsch, Song Han, Lin Meng, Oliver Bringmann, Haotian Tang, Shang Yang, Hengyi Li, Matthias Wess and Martin Lechner, *Special Session: Estimation and Optimization of DNNs for Embedded Platforms*, *International Conference on Hardware/Software Codesign and System Synthesis*, pages 21–30, 2024.

Author's Contribution: Provided latency estimation results

Relevance to this Thesis: Estimation and optimization of deep learning models for embedded platforms

Contents

Chapter 1

Introduction

In the past decade, Deep Neural Networks (DNNs) have become crucial for solving complex tasks across various domains such as computer vision [5], time-series analysis [6], and natural language processing [7]. These algorithms typically involve two main phases: a computationally intensive training phase, where models learn from large datasets, and an inference phase, where trained models are deployed to make real-time predictions. This development has resulted in a significant increase in the availability of hardware optimized for both phases, not only in cloud environments but also in energy-constrained edge and embedded systems [8]. This dissertation focuses on inference and aims to improve the design flow from the original model to its real-world hardware implementation.

When tackling specific problems using DNNs, considering the final hardware solution from the beginning is often not feasible. When tackling specific problems using DNNs, considering the final hardware solution from the start is often impractical. Even if feasible, factors such as cost, scalability, and the continuous evolution of Artificial Intelligence (AI) models must be considered to ensure a viable implementation.

For many applications, the main challenge lies in developing a functional machine learning model that achieves satisfactory accuracy. In other cases, constraints such as latency, power consumption and cost necessitate the use of embedded platforms. Deploying trained algorithms on these platforms can be challenging due to limitations in memory and computing power.

In straightforward cases that fall into well-defined categories, such as object detection or image classification, developers can utilize existing tools, code repositories, and vendor-provided workflows to speed up the development process.

However, several real-world use cases do not fit neatly into these predefined tasks. Each problem presents unique intricacies that machine learning engineers must address before optimizing the system's performance on embedded platforms. Consequently, the path from training a DNN to its final hardware implementation is marked by several critical design decisions that ultimately determine the success of the project.

Fortunately, the rise in AI popularity has led to an abundance of tools that simplify downstream implementation and provide access to pre-trained DNNs. Resources such as Hugging Face [9], pre-trained models in TorchVision [10], Ultralytics' YOLO implementations [11], benchmarks such as MLPerf, exchange standards like Open Neural Network Exchange Format (ONNX) [12], and inference frameworks like TensorRT [13] and OpenVINO [14] have significantly accelerated developments across all areas. The increased accessibility of these tools enables researchers and developers to focus on addressing the unique challenges of their specific applications.

Despite the availability of these tools, many numerous challenges remain. With each new DNN model and hardware platform, the design space continually expands, making it increasingly difficult to navigate. As a result, benchmarks that were published just a few years ago quickly become outdated, and tedious experiments must be conducted for each new DNN architecture and hardware platform. To accelerate the design and implementation process of DNNs, this thesis aims to address key obstacles in the design flow, streamlining the search for the optimal implementation with the best combination of hardware and DNN architecture.

Section 1.1 provides an overview of the design and implementation flow of DNNs on embedded hardware. The Chapters 2, 3, and 4 offer more detailed insights. The chapters of this work align with the publications on DNN latency prediction ([Paper I] [1], [Paper II] [2]), quantization ([Paper III] [3]), and hardware profiling ([Paper II] [2],[Paper IV] [4]), addressing their respective challenges.

1.1 Design and Implementation Flow for Deep Neural Networks

Figure 1.1 provides a rough and incomplete overview of the design space. Within each of the indicated points are numerous subcategories,leading to an exponentially growing number of possible implementations to a specific problem.

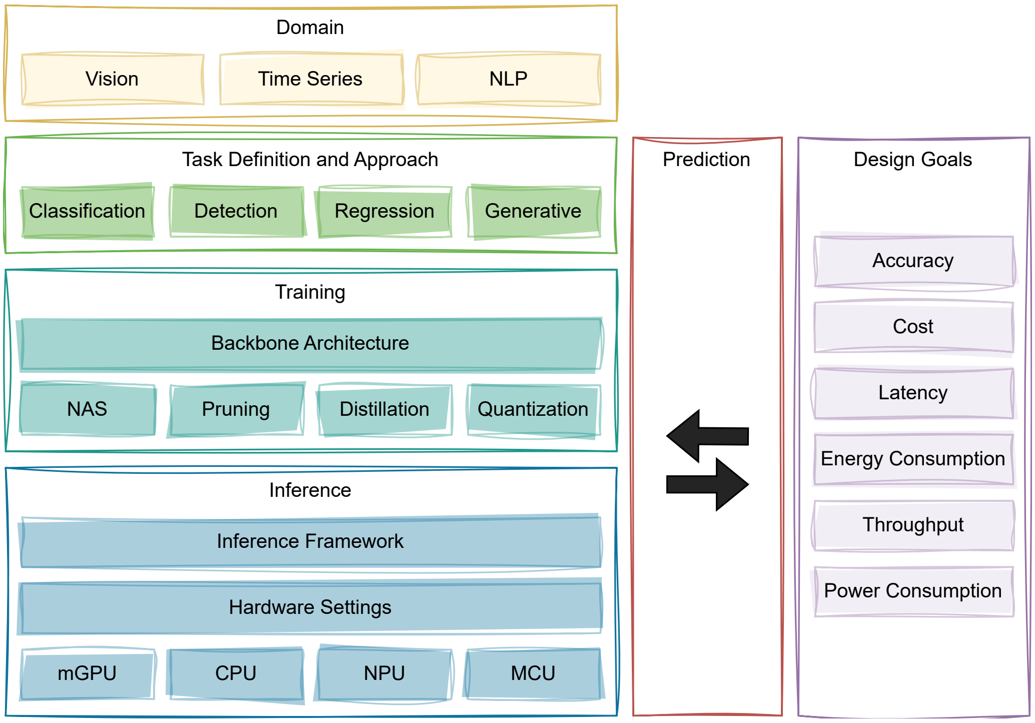


Figure 1.1: Overview of the design space for DNN design and hardware inference.

The domain of a machine learning task is usually determined by the application. However, there are cases where a problem can be approached from different perspectives, leading to a shift to another domain. This transformation is a critical design decision that can greatly impact the accuracy and performance of the solution. It often allows for the utilization of better-suited algorithms for the task. For example, converting raw audio data to spectrograms enables the use of image-focused DNNs to address tasks such as keyword spotting [15].

Next, the framing of the problem plays a crucial role in the design process. While some tasks, such as classification or object-detection, are well-defined, with established methods available to solve them, many real-world problems pose greater challenges. For instance, a problem that involves identifying objects could be framed either as a classification task (recognizing object types) or a detection task (locating objects). Selecting the right task and approach is critical, as it shapes the subsequent steps in machine learning model design and hardware selection [16].

Pretrained backbones can serve as a starting point, requiring only minor adaptations to address a specific machine learning task. However, since these backbones are trained on generic datasets, they are often not the optimal choice for specialized applications. To meet design goals, methods like Neural Architecture Search (NAS) can help automate the process of exploring and identifying the best architecture for the task [17]. Techniques such as pruning [18], and knowledge distillation [19] are commonly applied to optimize the model for accuracy and efficiency, particularly when targeting specific resource-constrained hardware platforms [20]. Similarly, quantization offers the opportunity to optimize DNNs for hardware friendly inference [3].

Finally, selecting the appropriate hardware for model inference is critical, as it must meet performance, resource and cost constraints. Each hardware platform supports specific inference frameworks, such as TensorRT [13] or OpenVINO [14] which offer different deployment options, compatibility with training frameworks, and vary in how efficiently they optimize model inference performance [21]. Additionally, different hardware platforms support various data types, which affects precision, speed, and compatibility [4], and are often optimized for specific application domains. Beyond these technical factors, considerations like product life-cycle, scalability, and future updates are important to ensure the solution remains adaptable and sustainable in the long term. While hardware and development costs are not technical constraints, they are crucial for economic viability. They impact production, scalability, and market success, making a technically feasible solution impractical if costs exceed budget limits. Thus, cost is key in product development but does not determine technical feasibility.

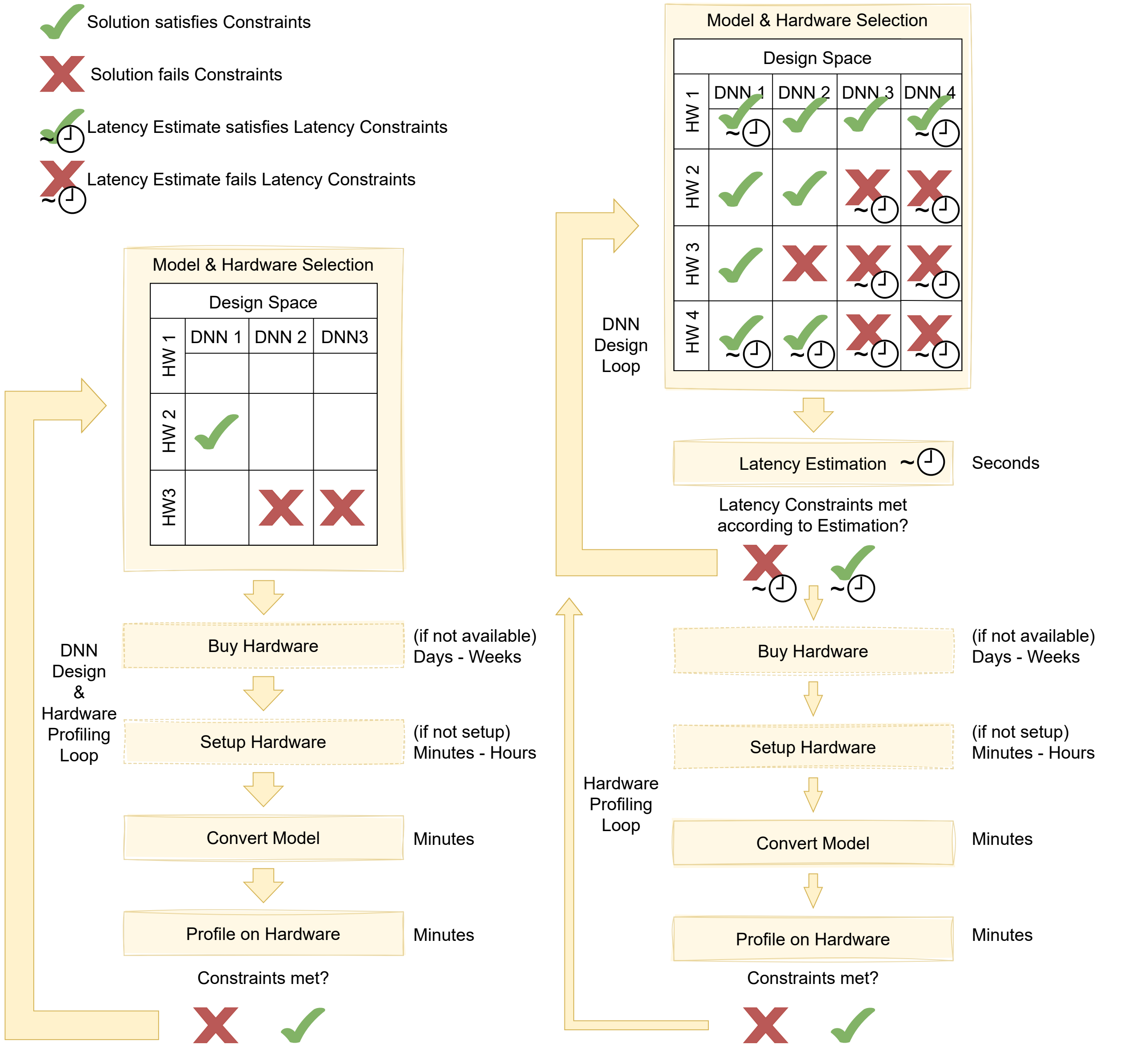
Resulting from all these considerations the different implementations can be compared with respect to accuracy, latency, throughput, power and energy consumption, cost and other metrics. For each of those design goals in isolation it is usually quite easy to define the optimal outcome:

1. **Accuracy** should be as high as possible (↑),
2. **Latency** should be as low as possible (↓),
3. **Power and energy consumption** should be as low as possible (↓),
4. **Cost** should be as low as possible (↓),

However, these goals are often interconnected, and additional factors such as available dataset size or hardware limitations also come into play. As a result, it becomes difficult to pinpoint a single target for the final solution. Given this complexity, it is essential to establish constraints for each design goal and parameter to guide the development process effectively and ensure a balanced solution [22].

Meeting design goals—such as optimizing accuracy, latency, and power consumption—requires careful balancing of trade-offs. In real-time applications, achieving low latency is crucial, as quick model responses are essential. Latency estimation can play a key role in early-stage design decisions by helping to rapidly narrow the design space and identify feasible solutions.

For example, consider a real-world scenario where a baseline solution addresses the problem with sufficient accuracy. From there, the next step is to define key constraints such as latency, throughput, power consumption, and energy consumption. Once these parameters are established, we attempt to run the model on the available hardware. If the model meets the performance requirements and the hardware is cost-effective, the process is complete. However, if the solution does not satisfy the constraints, we can either adapt the machine learning model, apply further optimizations, or select different hardware and adjust the hardware settings.



Solution satisfies Constraints
Solution fails Constraints
Latency Estimate satisfies Latency Constraints
Latency Estimate fails Latency Constraints
Model & Hardware Selection
Design Space
DNN 1
DNN 2
DNN3
HW 1
HW 2
HW3
DNN Design & Hardware Profiling Loop
Buy Hardware
(if not available) Days - Weeks
Setup Hardware
(if not setup) Minutes - Hours
Convert Model
Minutes
Profile on Hardware
Minutes
Constraints met?
Model & Hardware Selection
Design Space
DNN 1
DNN 2
DNN 3
DNN 4
HW 1
HW 2
HW 3
HW 4
DNN Design Loop
Latency Estimation
Seconds
Latency Constraints met according to Estimation?
Hardware Profiling Loop
Buy Hardware
(if not available) Days - Weeks
Setup Hardware
(if not setup) Minutes - Hours
Convert Model
Minutes
Profile on Hardware
Minutes
Constraints met?

Figure 1.2: Simplified DNN design flow comparing the traditional approach (left) and the optimized approach with latency estimation (right). Latency estimation speeds up the process, enabling broader exploration of viable solutions and informed decisions before hardware acquisition and testing.

This iterative process of testing various models with different hardware platforms and configurations, shown on the left side of Figure 1.2, can become cumbersome, especially when the necessary hardware is not readily available. While it is difficult to predict the accuracy of different algorithms, inference latency can be reliably estimated [2]. By applying latency estimation, the design space of feasible solutions can be rapidly narrowed, eliminating the need for a bruteforce approach and streamlining the development process. Figure 1.2 visualizes how latency estimation reduces the need for profiling by splitting the DNN design and hardware profiling loop. As applying latency estimation is only a matter of seconds, a broader and more complete design space can be explored before making informed decisions about which hardware to acquire, setup and test. This approach filters out solutions that fail to meet the defined constraints and supports the design of DNNs for hardware that may not yet be available. As a result, developers can focus on optimizing algorithms rather than repeatedly profiling them on hardware.

Going forward, Figure 1.3 sketches a more detailed design flow for DNN hardware implementation. For this work we view pruning similarly to knowledge distillation as a method to reach an optimal DNN architecture by adjusting the trade-off between accuracy and compute intensity [21]. The resulting design flow can be split into three phases: **Training**, **Hardware-specific Optimization**, and **Inference**.

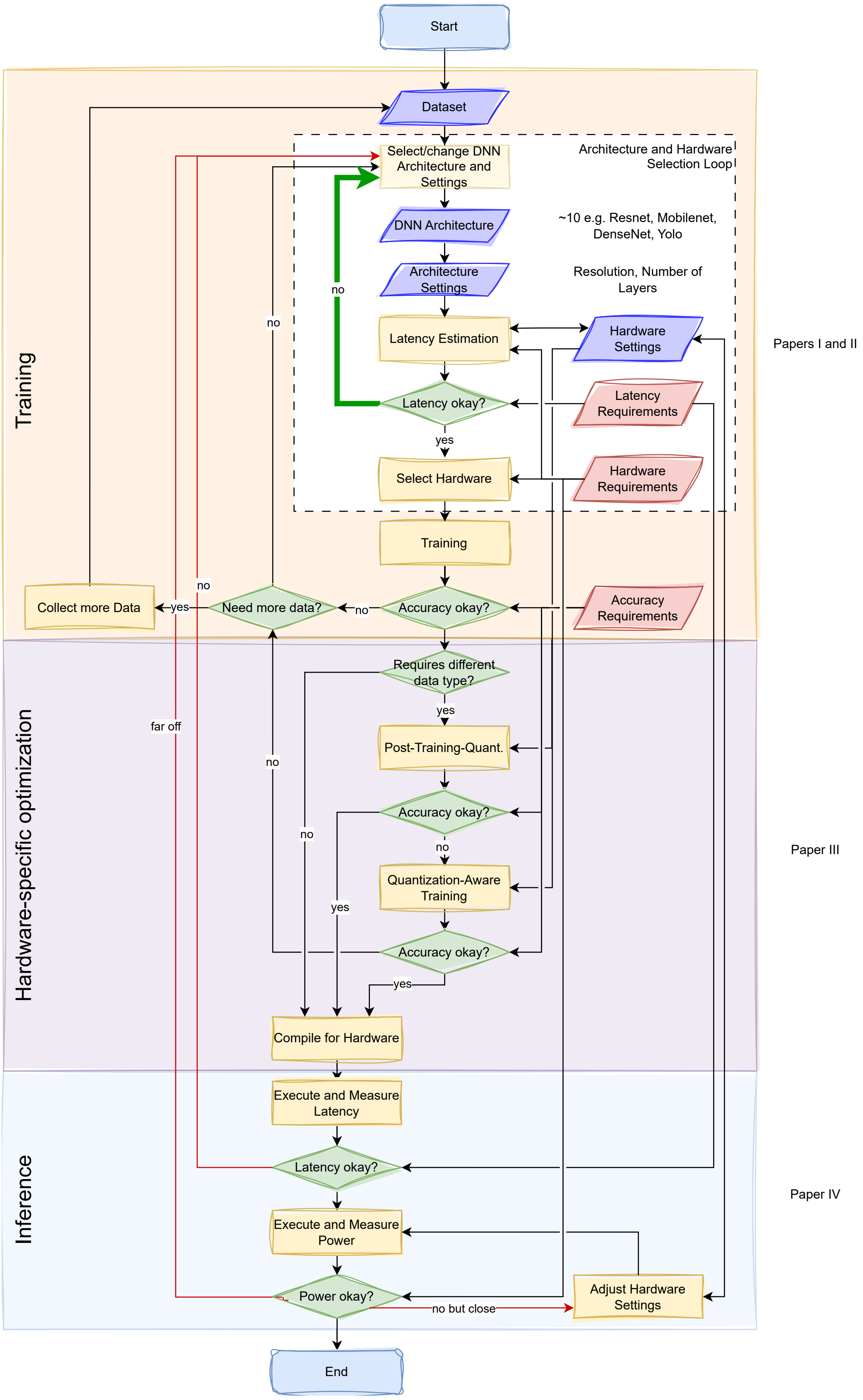


Figure 1.3: DNN design flow for hardware implementation

1.1.1 Training

The process begins with defining the task and gathering the relevant dataset. A network architecture, or a set of architectures, is then selected. Latency estimation is performed to provide an early indication of model suitability for deployment on specific hardware platforms. This step helps eliminate models that are computationally expensive or slow. Once promising models are identified, training proceeds with floating-point precision. After training, the model's accuracy is evaluated. If the required accuracy is achieved, the model proceeds to hardware-specific optimization.

1.1.2 Hardware-Specific Optimization

During this phase, the hardware's requirements are assessed, particularly regarding supported data types (e.g., FP32 vs. INT8). Firstly, Post-Training Quantization (PTQ) is applied, as it is a simpler and more efficient and low effort method that does not require retraining. However, if PTQ leads to a significant accuracy drop, Quantization-Aware Training (QAT) can be employed to recover the lost accuracy. Once the model reaches acceptable accuracy levels, it is compiled for the target hardware. If any hardware-specific adjustments, such as different architectures or datatypes, are necessary, the process loops back to the training phase.

1.1.3 Inference

The final phase involves compiling the model for the hardware and deploying it. Latency is measured to ensure predictions made prior to training the model are met. If the latency is acceptable, power profiling is carried out to confirm the model's energy efficiency. If latency exceeds the estimations, the model or hardware selection may require further adjustment.

1.2 Relevance of Publications

The publications presented in this thesis address critical challenges of the DNN design and implementation flow. In the **Training phase**, the research on latency estimation (*ANNETTE* [Paper I] [1]) and the application of *Conformal Prediction* [Paper II] [2] introduce methodologies to achieve accurate latency estimation and confidence intervals for performance predictions, improving the selection of network architectures. In the **Hardware-specific Optimization phase**, studies on *quantization and memory minimization* [Paper III] [3] help maintain model accuracy while optimizing for hardware limitations. Finally, in the **Inference phase**, the research on *black-box benchmarking* and *energy profiling* [Paper IV] [4] enables the evaluation of latency and power/energy consumption, ensuring that the DNN is optimized for both performance and energy efficiency on the target hardware platforms. These contributions collectively offer a comprehensive framework for optimizing DNNs across the entire design flow.

Chapter 2

Latency Prediction

In comparison to the vast amount of research available in the training and software areas of AI, the field of latency estimation for DNNs remains relatively small. Several factors contribute to this disparity:

Firstly, while AI algorithms themselves have seen groundbreaking advancements, the development and analysis of hardware for executing these algorithms has traditionally lagged behind. This gap is partly why Graphic Processing Units (GPUs) became central to AI development, despite being originally designed for graphics processing. The flexibility of GPUs allowed them to be quickly adapted for neural network training and inference, but the specialized development of hardware for DNNs has taken more time to mature [23].

Secondly, the initial surge in AI focused on applying algorithms to various datasets, offering abundant opportunities for researchers without necessitating immediate concern for hardware performance on embedded devices. Deploying these algorithms in real-time scenarios introduces additional complexity, requiring sophisticated optimization tools, including accurate latency estimation models [Paper I] [1]. The heterogeneity of hardware platforms further complicates this issue. DNNs are deployed on a wide range of hardware (e.g., GPUs, Field Programmable Gate Arrays (FPGAs), Neural Processing Units (NPUs)), each with unique characteristics [24]. This diversity makes it difficult to create generalized performance models.

Moreover, much of AI's early deployment occurred in cloud-based or offline environments where latency and real-time performance were not primary concerns. Only with the increasing demand for real-time applications such as autonomous driving, has the need for accurate performance estimation tools become critical [25]. Yet, many developers still rely on benchmarks or prior knowledge when selecting hardware platforms, instead of using dedicated tools for precise latency prediction.

2.1 Latency Estimation Methods

When comparing DNN architectures, metrics such as Floating Point Operation (FLOP) and parameter counts are commonly used, as they offer a straightforward way to quantify aspects like computational complexity and memory requirements. However, while these proxy metrics are useful for initial comparisons, they do not capture how the algorithms are actually mapped onto hardware [Paper I] [1]. The computational efficiency of different hardware platforms for the same network architecture can vary significantly depending on factors such as the utilized set of operations, the sequence in which these operations are executed, and the interconnections between them [23].

Analytical approaches attempt to improve upon this by estimating performance based on theoretical hardware capabilities [26]. For example, dividing the total number of operations by the hardware's theoretical peak number of Floating Point Operations per second (FLOPs/s) provides a rather optimistic latency estimate, as hardware vendors typically report peak performance numbers that assume perfect utilization of all compute resources, which is rarely achievable in practice. Additionally, factors like memory access patterns and parallelism overhead are not captured in such simplistic calculations.

The roofline model [26] offers an improvement by incorporating memory bandwidth alongside computational performance. It provides a visualization of performance bounds based on both computation and memory access, making it useful for understanding how well a DNN architecture utilizes available hardware.

While the roofline model itself is simple the challenge lies in properly applying it to DNNs for accurate latency estimation. For inference, the DNN is partitioned across the available hardware resources. As the roofline model fails to cover such aspects it can result in largely inaccurate predictions. Nonetheless, for compute-heavy layers such as convolutional layers, where these design challenges have a smaller impact, the roofline model can still yield reasonably accurate performance estimates [Paper I] [1].

Building upon the roofline model, several approaches have been developed to better account for hardware-specific details. Improvements in modeling the memory hierarchy and extending the roofline model to accommodate more realistic predictions by focusing on different aspects. For instance, the models introduced by *ANNETTE* [1] and *Blackthorn* [27] refine performance models by focusing on compute parallelism overhead. Other models [Paper IX] [28], [29] provide deeper insights by diving down to the level of basic operations and instructions, which is especially relevant for Microcontroller Units (MCUs) and other low-power devices.

Moreover, simulators like *SCALE-Sim* [30] and *SimPyler*[31] offer detailed simulation-based approaches to predict latency more accurately for specific hardware architectures. These simulators offer precise insights by modeling low-level hardware interactions. However, they can be time-consuming to use and often require extensive hardware knowledge, which may limit their practicality for rapid prototyping or when evaluating a vast design space.

Machine learning-based approaches have emerged as a promising alternative for latency prediction. In these methods, the target device is extensively benchmarked to collect execution time data for various network layers and configurations. This data is then used to train machine learning models that predict execution time based on network characteristics and hardware parameters. The core methodology across these approaches remains similar, with differences in feature engineering, applied machine learning algorithms, and data collection strategies [Paper I] [1], [Paper II] [2], [32].

Optimization Challenge

One of the core challenges in performance estimation lies in correctly capturing and modeling the mapping of layers of a neural network to the specific hardware resources. The per-layer approach remains a valid technique, where the performance of each individual layer is estimated separately. However, this method struggles to account for inter-layer optimizations, such as operation fusions and shared memory usage between layers. Tools like *ANNETTE* [Paper I] [1] address this by using decision trees to detect possible fusions between layers. Similarly, *nn-Meter* [33] includes specific kernels for different combinations of layers and benchmarks them to find the best configuration. The methodology begins by identifying available kernels and optimizations, then benchmarks those combinations to achieve accurate performance estimates. For example, care must be taken when measuring multiple layers simultaneously. To avoid skewed results ANNETTE and nn-Meter utilize Random Forests for per-kernel predictions after performing a layer fusion rule detection. PerfSAGE [34], DIPPM [35], SLAPP [36] rely on Graph Neural Networks (GNNs) to capture cross-layer optimizations. In more complex cases, such as those involving graph neural networks [32], data collection can be more comprehensive or restricted to a smaller design space, capturing information for possible layer configurations.

Required Data for Accurate Modeling

A second key challenge is balancing the need for accurate performance predictions while minimizing the amount of data required to train the models. Since the design space is vast, collecting exhaustive data for all possible configurations is impractical. Therefore, it becomes essential to identify the most representative data points. Hybrid approaches, combining empirical measurements with analytical models, are often employed to achieve this balance [Paper I] [1] [37]. ANNETTE, for instance, performs parameter sweeps and selects specific points to measure, focusing on scenarios where the compute architecture is optimally utilized. By combining the machine learning models, trained on this data, with analytical models, ANNETTE is able to deliver robust predictions with fewer data points. Performance representatives [37] further improved this concept by using integer division to reduce machine model complexity. *Blackthorn* [27] uses a model-in-the-loop approach to dynamically adjust the required measurement points during runtime. This approach significantly reduces the number of required data points but may not capture the exact behavior in all scenarios due to the underlying model assumptions.

Hardware Heterogenity

Handling heterogeneous hardware platforms, such as NVIDIA devices where different parts of the network can be assigned to different cores (e.g., GPU vs. Deep Learning Accelerator cores) or mobile SoCs, remains a significant challenge [38]. Often, this mapping is done manually, with developers specifying which network components should be assigned to specialized compute resources. While statistical estimation models can benchmark layers on specific devices and automate parts of this process, there is no universally applicable solution to handling hardware heterogeneity. The problem is particularly challenging in scenarios where layers need to be dynamically mapped to different cores based on real-time performance considerations.

2.2 Challenges Addressed in this Work

The primary contribution of this research lies in the development of systematic approaches, most notably *ANNETTE* [Paper I] [1] and its underlying models. *ANNETTE* has significantly advanced the establishment of a structured methodology for benchmarking, particularly on hardware with varying computational efficiencies. By combining analytical and statistical models, ANNETTE strikes an effective balance between minimizing the number of measurements required and maintaining high accuracy. Additionally, the integration of layer-level optimizations, such as layer-fusion [39], has improved prediction accuracy at the network level. As a result, the system achieved prediction errors within a range of approximately 5-15% [Paper I] [1].

Building on this, we introduced a confidence framework as an extension to *ANNETTE*, leveraging *conformal predic-*

tion [Paper II] [2]. This framework serves two key purposes: first, it provides a mechanism to evaluate whether the prediction model sufficiently covers the entire design space. If the model cannot confidently predict performance for certain regions, developers are notified, allowing them to either refine the model or avoid architectures that are unlikely to meet the required specifications. This enhances the ability to interpret results and make more informed decisions when selecting hardware or DNN architectures, particularly in edge cases where the model might be less reliable. Additionally, the framework aids in the assessment and refinement of the generated models. In parallel with the development of this framework, a smart benchmarking approach was introduced [Paper II] [2], enabling the generation of *ANNETTE* models without requiring per-layer benchmarks, further streamlining the design process.

2.3 Impact on the State of the Art

The ANNETTE framework laid the foundation for structured performance modeling of DNNs. The ACADL-based automated performance modeling framework [29] builds upon ANNETTE's stacked modeling approach by introducing a formalized architecture description language for hardware accelerators, thereby extending on the architectural description of the refined roofline model. The Performance Representatives method [37] refines ANNETTE's benchmarking strategy, reducing the number of required training samples while maintaining estimation accuracy. The SLAPP framework [36] extends ANNETTE's operator-level and layer-wise estimations by leveraging graph-based learning techniques to model execution times at a subgraph level. Together, these works demonstrate the lasting impact of ANNETTE on scalable, efficient, and hardware-aware DNN performance estimation.

Chapter 3

Quantization

Quantization is a crucial technique in optimizing DNNs for efficient deployment across a wide range of devices [40]. Unlike other optimization methods which primarily focus on reducing the size or complexity of neural network architectures, quantization aims to reduce the data type size of weights and feature maps within the already defined architecture. As a result quantization is commonly applied in addition to architecture optimization techniques such as pruning [41]. The transition to smaller data types not only reduces the amount of data that needs to be moved between memory and compute units, which is a critical bottleneck in many DNN applications, but also simplifies operations [40]. Integer operations, for instance, are less complex and consume less power compared to floating-point operations, but this comes with a trade-off in precision [42]. Quantization, therefore, must be carefully managed to minimize the loss of accuracy, particularly as lower precision formats are adopted [43].

In today's landscape, many embedded and specialized devices designed to run DNNs are equipped with compute units that are optimized for specific data types. For example, Nvidia GPUs are tailored for FP32 and FP16 but also support INT8 operations to enhance performance [44]. Similarly, Central Processing Units (CPUs) commonly support several datatypes but also integrate Single Instruction, Multiple Data (SIMD) engines and instructions optimized for FP16 and INT8 [45]. NPUs, on the other hand, are often focused on executing INT8 computations due to the balance they strike between precision and efficiency [46]. Furthermore, newer hardware architectures are introducing support for even more specialized data types such as BrainFloat16 or even INT4 and INT2, pushing the boundaries of efficiency in DNN inference [47, 44].

Applying quantization is especially valuable in scenarios where resource constraints are a significant consideration, such as in mobile devices, autonomous systems, and real-time edge computing applications [48, 49, 50]. By reducing the bit-width of weights and activations, quantization helps decrease the computational and memory demands of neural networks, resulting in improved speed, energy efficiency, and memory usage [51]. As the demand for deploying DNNs in real-world, resource-limited environments grows, quantization remains an essential tool for achieving both high performance and energy efficiency [23].

3.1 DNN Quantization-Specific Considerations

In general, quantization describes the well-established concept to map most often continuous input values to a set of output values by applying a quantization function f, typically achieved with rounding and truncation [51]. Interestingly, DNNs bring some new opportunities and challenges to the problem of quantization [51].

3.1.1 Over-Parameterization

Firstly, DNNs are essentially large constructs composed primarily of matrix multiplications and non-linear activation functions. Therefore, both training and inference are computationally intensive tasks that can be challenging to handle. Consequently, optimizing data types and operations can lead to significant gains in energy-efficiency and throughput [23, 39]. However, despite enormous efforts of researchers to develop slim and efficient DNN architectures, these networks remain heavily over-parameterized, especially when models optimized for standard datasets like ImageNet are applied to real-world use cases with a limited set of classes [52, 53]. This over-parameterization enables techniques such as pruning and quantization to achieve high compression rates at minimal to no accuracy loss [18, 54].

3.1.2 Evaluating Quantization Quality

Secondly, DNNs do not necessarily solve well-posed or well-conditioned problems, but are usually trained by minimizing the result of a loss function which approximates the prediction error of the DNN on a training dataset. Due to the over-parameterized nature of DNNs there are multiple different models that optimize the prediction accuracy sufficiently [55]. As a result, it is possible to have a high quantization error between the original and the quantized model with still very good prediction and generalization performance [56, 41]. In some cases, minimizing quantization error might be important, for instance to maintain the exact behaviour of the original DNN. However, in general, DNNs can be treated as black-boxes where only the resulting accuracy matters. This contrasts with traditional quantization methods, which focus on preserving the original signal as closely as possible [57].

3.1.3 Quantization Granularity

Besides these two aspects, which provide additional degrees of freedom compared to standard quantization, other important considerations make the quantization of DNNs even more complex. The layered structure of neural networks allows for the application of varying quantization schemes and compression rates at different granularity levels, such as per-model, per-layer [Paper III] [3] or even per-channel quantization [51]. However, to fully exploit the benefits of these approaches, the hardware must be capable of supporting variable precision operations across layers and channels. Consequently, no single solution has emerged as universally optimal.

3.1.4 Quantization and Training

The final key consideration is how quantization interacts with the training process. As floating-point precision is essential for gradient-based optimization methods, which rely on small, incremental updates to weights, specifically extreme quantization poses a problem. To avoid this, PTQ [58, 59] separates the training from the quantization step, by applying the quantization after the model has been fully trained. Usually PTQ is fast and resource-efficient as it requires no additional training, but it can result in significant accuracy reduction since the model has not been exposed to the quantization effects during training. Thus, as an alternative, QAT [56, 42] integrates the quantization process. By simulating the effects of quantization during training (fake-quantization), the model can adapt its weight representations to be more robust to lower precision, reducing accuracy loss when the final quantized model is deployed.

A major challenge in QAT is the non-differentiability of quantization functions, such as rounding, which interrupts the gradient flow needed for backpropagation. Several strategies have been proposed to address this issue [51, 42, 3, 60, 61, 62]. Straight-Through Estimation (STE) is a common approach that bypasses the rounding operation by approximating the gradient of the quantization function as the identity function [51].

Further quantization techniques involve the introduction of randomness into the rounding process, rather than deterministically rounding values to the nearest quantization level (stochastic quantization) [42] or encouraging weights to settle into values that are easier to quantize, by applying regularization during training (weight regularization) [3]. Alternatively, making quantization parameters dynamic and learnable during training, as in PACT and LSQ, which allow quantization ranges and step sizes to adapt during training, results in more flexible and precise quantization, improving the final model's performance [60, 61, 62].

3.1.5 Quantization Schemes

When applying quantization, several schemes must be considered. We can distinguish between symmetric and asymmetric, and uniform and non-uniform quantization schemes. It must be considered, that not all presented methods fit all of the quantization schemes [Paper III] [3]. Therefore the target quantization schemes have to be taken into consideration when choosing the quantization methodology. For INT8 quantization, for example simple PTQ can be sufficient [63], depending on the task, DNN architecture and target accuracy.

3.2 Challenges Addressed in this Work

We addressed the challenge of balancing memory compression and accuracy retention in neural networks, particularly for deployment on devices with limited memory and computational resources [Paper III] [3]. Based on the previous discussions, the work makes use of different quantization bit-widths at the layer granularity while enforcing quantization through an additional regularization term. Furthermore, the method is evaluated using two quantization schemes: Dynamic Fixed Point (DFP) and Power-of-Two (Po2).

The main contribution of this work is the introduction of WQR, a technique that enhances quantization by adding a regularization term to the loss function during training. This term forces the weights to gravitate toward quantization levels, thereby reducing quantization error and improving accuracy after quantization. The results show that adapting the regularization term, to the applied quantization function can increase the achievable accuracy. Additionally, Layerwise Precision Scaling using a greedy algorithm is deployed to ensure that more critical layers retain higher precision, while less sensitive layers are quantized more aggressively.

The results of the method are compared for two quantization schemes: DFP and Po2. The findings show that DFP performs well at higher bit-widths, while Po2, which simplifies operations into bit-shifts, is preferred for lower bit-widths. It is demonstrated that WQR achieves significant compression ratios (up to 9.33x for the SVHN dataset) with minimal accuracy degradation, making it a highly efficient technique for hardware implementations [Paper III] [3].

At the time this method was introduced, it represented a step forward in quantizing DNNs for resource-constrained environments. As deep learning has progressed and workloads have grown more complex, the exact method may not be entirely sufficient on its own. Nonetheless, the core principles remain sound, and WQR can be effectively combined with other quantization or optimization methods. The results achieved in this work address several recurring challenges as DNN models continue to grow:

- Datatypes: Current trends favor smaller integer and low-bitwidth floating-point types [64]. Since Po2 based schemes lack flexibility and their effectiveness depends on the weight distribution, they are unlikely to gain broader adoption.

- Granularity: Adjusting precision at the layer or block level is currently the most efficient approach to optimize DNNs [65]. For instance, Nvidia's platforms allow for the use of different datatypes in different parts of the network, illustrating the growing importance of fine-grained control.

- QAT Approach: At 8-bit precision, STE is usually sufficient for maintaining accuracy. However, for lower precision levels, it will be interesting to see which quantization technique becomes dominant. A combination of techniques, including STE, stochastic rounding, and quantization regularization, is theoretically viable. For fine-tuning, the regularization-based approach may better preserve the original behaviour of the DNN.

3.3 Impact on the State of the Art

The WQR approach presented in this work has influenced research in QAT and hardware-efficient deep learning models. Singhal et al. [66] extend WQR by incorporating non-uniform quantization with learnable bit multipliers, improving flexibility and fault tolerance in low-bit neural networks. [67] builds upon WQR's structured regularization principles to optimize weight representations, reducing hardware complexity through efficient sub-expression sharing. QFALT [68] directly applies Quantization Regularization to enhance fault tolerance in quantized models, ensuring weight robustness in unreliable hardware environments.

Chapter 4

Hardware Profiling

After successful implementation and optimization, the final step in the general DNN implementation workflow is profiling the execution on the target platform [Paper IV] [4]. For this we assume that the resulting accuracy is already evaluated and meets the requirements. In contrast to accuracy evaluation, inference profiling is mostly used to verify the fulfillment of latency and throughput requirements. However, there are plenty of other reasons why a developer might want to assess performance metrics of an application and record a detailed profile. Consequently, the approach taken heavily depends on the deployment requirements of the DNN application and the expected insights. This section outlines the different goals of profiling, the specific metrics of interest, and the granularity at which profiling can be applied. It also explores some of the unique challenges associated with profiling DNNs.

4.1 Different Goals of Profiling

The specific objective of profiling a DNN determines the profiling strategy and the types of insights sought. We can identify several distinct goals for profiling, each with a particular focus [23]:

- General Profiling and Bottleneck Identification: Profiling can be used to gain an overall understanding of the entire application, identify specific bottlenecks that limit performance, and detect inefficient layers within the network [69]. This involves analyzing the entire pipeline to identify stages or layers that are costly in terms of computation, memory, or data transfer. The gathered knowledge can be used for further optimization efforts such as quantization, pruning, or hardware-specific tuning.

- Hardware Configuration Optimization: Profiling helps identify the best hardware settings for optimal performance or other design goals, which may include adjusting clock speeds or determining the ideal number of active and used hardware resources, such as processor cores [70].

- Software and Compiler Configuration Optimization: Use profiling to identify optimal software configurations and compiler settings for optimizing performance. This includes adjusting compiler optimization flags, selecting efficient libraries or frameworks, and fine-tuning application-specific parameters [28]. Examples are optimizing for throughput or latency, improving memory usage, and determining the ideal number of threads for parallel execution.

- Data Collection for Modeling: Profiling can also be used to gather data about resource usage, performance, and hardware behavior to model the behavior of DNNs executed on hardware. This data can be used to generate estimation models that help optimize deployment for future tasks [34].

4.2 Metrics to Profile

When profiling a DNN, several inference metrics are commonly analyzed to gain insights into the performance of the actual execution on hardware. The most crucial metrics for real-time applications are **latency**, which measures the time taken to execute the DNN from input to output, and **throughput**, which quantifies the number of inferences the system can process over a given period [23]. For battery-powered or energy-constrained devices, understanding the **power** and **energy consumption** of the hardware during the execution of the DNN becomes essential [Paper IV] [4]. Finally, **hardware resource utilization** is key to understanding how well the DNN architecture leverages the underlying hardware [34].

4.3 Profiling Granularity

Finally, the profiling can occur at different levels, starting with the processing pipeline level, where the entire application process, including stages like input, pre-processing, core DNN computation, and decision-making, is analyzed to provide an overview of latency and resource utilization across all components. At a more focused level, profiling can target specific parts of the network, such as the backbone or task-specific heads, to understand how each section contributes to the computational load and identify areas for optimization. Finally, profiling can be performed on individual kernels, offering insights into specific operations that could benefit from optimization or more efficient implementation [Paper IV] [4].

4.4 Challenges in Profiling DNNs

Although, most hardware vendors provide tools to enable hardware profiling, several open challenges remain. The granularity at which profiling is conducted significantly influences the complexity of acquiring the desired metrics. While in-depth profiling, such as per-kernel latency analysis, is available through tools like Nvidia's Nsight Systems [71] for their Jetson mobile Graphic Processing Unit (mGPU) platform, these tools often introduce overhead, which can skew the results. Nevertheless, relative measurements are still valuable for understanding trends and identifying inefficiencies. On other platforms, such as the NXP i.MX93 development board (i.MX93) [72], extracting similar metrics proves much more difficult, making it challenging to develop tools that work uniformly across different hardware architectures.

Furthermore, obtaining accurate power consumption and hardware resource usage data presents an even greater challenge. Although systems generally have built-in sensors to monitor power usage for system protection, their sampling rates are frequently limited [Paper IV] [4]. Similarly, vendors are often reluctant to disclose detailed hardware utilization information to avoid exposing their internal architectures to reverse engineering.

Lastly, when profiling DNNs, it is essential to consider adjustable settings that can fine-tune both hardware and software. Hardware settings, such as clock speeds, the number of active cores, and power modes, significantly impact key metrics like latency, throughput, and energy consumption [Paper IV] [4] [70]. A thorough understanding and adjusting these settings enables optimal tuning of the hardware to meet performance requirements. On the software side, execution strategies, including, parallelism, scheduling, and the specific optimization goal (e.g. maximizing throughput, reducing latency, or minimizing power consumption) are equally important. As a result, even for one specific hardware and DNN combination the design space remains vast.

4.5 Challenges Addressed in this Work

To tackle these challenges we presented two main approaches: Our first approach involved using power side-channel analysis to study the power implications of different operations in the DNN. This method involved monitoring the power consumption of the device while running different layers of a neural network and extracting insights from the power profiles. By correlating the power traces with the known network structure, we were able to understand which layers and operations were the most power-intensive. This approach is especially useful in scenarios where direct access to profiling tools is unavailable or restricted, since it also allowed for extracting per-kernel latency numbers. The method was applied on different neural network accelerators, such as the Intel Neural Compute Stick 2 (NCS2) [73], the Coral Edge Tensor Processing Unit (Edge TPU) [74], and the NXP i.MX8M+ development board (i.MX8M+) [75], revealing the energy efficiency of specific layers under various hardware and software settings like clock frequency or parallel execution threads. The results showed that for the three DNN accelerators, the relationship between power consumption and energy per image can be optimized by adjusting either the number of parallel inference requests or the clock frequency. In both scenarios, power consumption rises when shifting to higher throughput modes, but still leads to a reduction in energy consumed per image overall [Paper IV] [4]. While energy consumption strongly correlates with latency, variations in dynamic power consumption can be attributed to additional factors such as memory access patterns and data movement overheads [76]. Furthermore, the analysis and comparison of different layer types revealed that depth-wise separable convolution layers exhibit lower compute efficiency than standard convolution layers. The systematic benchmarking approaches applied in [Paper I] [1] and [Paper II] [2] provide a solid foundation for conducting a more in-depth analysis, enabling further investigation into underlying performance bottlenecks and improvements in energy consumption modeling.

While the power measurements were useful, a more robust technique was required for gathering single layer benchmark data in cases where per-layer latency figures were unavailable. To address this, we introduce a *smart padding* method [Paper II] [2], which involves using padded versions of specific layers to isolate their performance and measure their impact accurately. The padding involves adding dummy input and output operations, which helps to minimize the overhead from data transfer and makes it easier to isolate the computation of a particular layer. The smart padding approach enables the creation of per-layer abstraction models even when profiling insights are limited. It compensates for data transfer and pipeline inefficiencies, enabling more accurate latency prediction while also significantly reducing the complexity of layer-specific benchmarks. We demonstrated the effectiveness of this approach on the NVIDIA Jetson Xavier AGX (Jetson Xavier) [77], i.MX93 [72], and i.MX8M+ [75] platforms. The empirical results showed that smart padding reduced prediction errors to below 10% for the tested devices [Paper II] [2].

Chapter 5

Conclusions and Outlook

This thesis has made contributions toward building a structured design and implementation flow for DNN, addressing key challenges in estimation, quantization, and profiling. Through the development of new methodologies and frameworks such as *ANNETTE*, this work has introduced solutions that enhance the efficiency of DNN deployment on a range of hardware platforms, also highlighting the areas that require further development to achieve the overall goal of a generalizable design and implementation flow.

In the area of estimation, this thesis successfully implemented latency estimation [Paper I] [1] to create a more holistic understanding of DNN behavior on embedded hardware. The combination of analytical and stochastic models proved to be crucial in this field. The introduction of a confidence framework [Paper II] [2] further improves the usability of the prediction results, allowing for more informed performance tuning across a wide range of architectures.

The next logical step is to connect latency estimation with resource utilization and power consumption models, bringing them together into a unified framework. Such a model would not only offer more comprehensive performance predictions but also make use of benchmarking across diverse hardware settings. This could also enable accurate predictions for upcoming hardware generations, leveraging architectural insights. Moreover, refining the confidence methods to include optimizations that occur during graph compilation, will further increase the trustworthiness of the predictions. An additional focus will need to be on an automated analysis and estimation whether a network can be inferred on a target device. This way developers can ensure that the selected or designed network contains only supported layers and fits within the memory constraints of the target hardware.

Quantization plays a crucial role in optimizing DNNs for deployment on resource-constrained devices. The development of WQR provides a method for dynamically adjusting layer precision based on its criticality. However, quantization remains a challenging and evolving area. There is no single method that is perfect for all networks and hardware platforms, and given the rapid pace at which DNN architectures evolve, this is unlikely to change. The field is moving toward smaller integer datatypes, such as INT4 and INT2 [78], but striking the right balance between compression and accuracy will continue to be a key focus.

As the landscape of DNN architectures shifts, future research should aim to improve the flexibility of quantization techniques, ensuring that they can adapt to the demands of new architectures. Although post-training quantization will likely remain the standard due to its simplicity, methods for QAT will be essential in cases where precision must be maintained despite aggressive quantization. Standards and support for different quantization types will also need to evolve, providing developers with more tools to handle the increasing complexity of DNNs.

Profiling is another area where this thesis has made progress. The use of power side-channel analysis provided valuable insights into the power and latency performance of DNNs at a granular level [Paper IV] [4]. This allows for a detailed understanding of how individual operations impact overall performance, particularly when deployed on embedded platforms. However, despite these advancements, the complexity of the post-processing required for power side-channel analysis prevented the full automation of these measurements. This presents a challenge for scaling the methodology to broader use cases. Additionally, the introduction of the smart padding technique enabled accurate layer isolation for latency profiling, even in the absence of detailed hardware transparency [Paper II] [2].

Future efforts should aim to combine both approaches to enable power consumption and resource estimation. The smart padding could potentially increase the reliability of the power measurements. Automation will also be key to making profiling more scalable, enabling real-time assessments across different hardware platforms, and streamlining the benchmarking process. As hardware architectures continue to diversify, automated profiling tools will be crucial for ensuring that performance predictions remain accurate and consistent across platforms.

As the focus of AI development increasingly shifts toward Large Language Models (LLMs) like GPT [79] and BERT [80], the methodologies developed in this thesis take on new relevance. While this work primarily Convolutional Neural Networks (CNNs), the growing prominence of LLMs presents a set of unique challenges that must be addressed in future research. LLMs are typically much larger than traditional CNNs for vision applications, and their resource requirements, particularly in terms of memory and power consumption, are substantially higher. The resource and power estimation techniques developed here provide a strong foundation, but they will need to be adapted to the specific needs of LLMs. The primary difficulty lies in the fact that many LLM implementations are still highly customized, relying on hand-optimized code to achieve maximum performance [81, 82]. Unlike CNNs, where inference frameworks have become widely adopted, LLMs lack such standardization, making their optimization more complex.

Looking ahead, it will be essential to develop more sophisticated resource and power estimation models that fit specifically to LLMs. Hardware support for LLMs is also still evolving, and future research should explore how to optimize these models for next-generation hardware platforms, ensuring that both efficiency and performance are maximized. Standards for LLM optimization will need to mature, and tools that can automate the benchmarking and profiling of these large-scale models will be crucial for their successful deployment.

As AI models and hardware continue to evolve, the need for adaptable, flexible, and efficient design methodologies will only grow. The contributions made in this thesis provide a strong foundation for this evolution, offering valuable insights and tools that will help drive further advancements in DNN optimization. By continuing to refine these methods and adapting them to new technologies, the field will move closer to realizing a fully automated, generalizable DNN design and implementation flow.

In general, there are several major areas to address in extending this research, although their priority will depend on developments in both the community and industry. The following items provide a high-level overview, focusing on situations where resources are limited:

- **Latency Prediction:**
  - *ANNETTE* has recently been adapted to work with ONNX, enabling experiments on a variety of state of the art DNNs. Moving beyond CNNs to include transformer-based and other architectures will likely unveil new challenges, and resolving them should further improve latency estimation accuracy.
  - In addition to latency, predicting the usage of memory and other resources is crucial. These factors can determine whether a DNN is actually deployable on a given platform. This is especially important for the TinyML domain.
  - With the rise of LLMs, time-series models are also gaining traction. Compared to most computer vision workloads, the temporal dependency in frames or samples introduces distinct inference optimization and scheduling strategies, which latency estimation tools should account for, including the different execution scenarios
  - Cross-platform prediction that leverages knowledge of prior hardware generations and different architectures is essential. Learning how efficiently one operator performs relative to another across architectures can streamline early design decisions.
- **Quantization:**
  - Develop hardware-aware, mixed-precision strategies with a focus on automated and general methods. This involves detecting layers that are highly sensitive to quantization and recommending suitable precision assignments and hardware mappings based on that sensitivity.
  - Investigate lightweight approaches to QAT (e.g., partial QAT), balancing compute overhead with accuracy requirements, and ensuring these methods align with supported execution standards on common hardware.
  - Extend quantization efforts to newer or emerging architectures such as State Space Models (SSMs), which may have different sensitivity patterns and numerical properties than CNNs or transformer-based networks. [83]
- **Hardware Profiling:**
  - Expand the smart padding (black-box) approach to capture inter-layer interactions and events, improving granularity in performance analysis, but also providing additional insights for estimation.
  - Combine vendor-specific insights (e.g., profiling tools, architecture documentation) with black-box measurement approaches to build a more holistic profiling framework.
  - Apply the smart padding approach to power profiling, aiming to provide more accurate energy measurements with minimal post-processing overhead.

Acknowledgments

This dissertation would not have been possible without the support of many people. I would like to express my sincere gratitude to:

... my supervisor Axel Jantsch for his excellent guidance, professional advice, and the many inspiring discussions that have significantly shaped this work.

... Herbert Taucher, Hannes Muhr, and Martin Matschnig, who made this work possible by initiating the cooperation with TU Wien helped me align my research with practical relevance.

... my colleagues at the CD Laboratory, whose company made even the hardest problems — and the occasional bite into granite — surprisingly enjoyable.

... all the people at ICT, ISAS, and Siemens for the opportunities to learn far beyond my own field — and for ensuring that there was never a shortage of good conversations, creative ideas, or freshly brewed coffee.

... my brothers and friends, for being fellow travelers — not always in the same direction, but always with enough drive to race across the seven seas, or at least up the next summit.

... my parents, Maria and Wolfgang, for their constant support — and for helping me grow into someone who doesn't always pick the *mittlere*, but often chooses the scenic route instead.

... my partner Miriam, whose support and energy got me through every crux this journey had to offer.

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.

Bibliography

[1] M. Wess, M. Ivanov, C. Unger, A. Nookala, A. Wendt, and A. Jantsch, "ANNETTE: accurate neural network execution time estimation with stacked models," *IEEE Access*, vol. 9, pp. 3545–3556, 2021.

[2] M. Wess, D. Schnöll, D. Dallinger, M. Bittner, and A. Jantsch, "Conformal prediction based confidence for latency estimation of DNN accelerators: A black-box approach," *IEEE Access*, vol. 12, pp. 109847–109860, 2024.

[3] M. Wess, S. M. P. Dinakarrao, and A. Jantsch, "Weighted quantization-regularization in dnns for weight memory minimization toward HW implementation," *IEEE TCAD*, vol. 37, no. 11, pp. 2929–2939, 2018.

[4] M. Wess, D. Dallinger, D. Schnöll, M. Bittner, M. Götzinger, and A. Jantsch, "Energy profiling of DNN accelerators," in *DSD*, pp. 53–60, IEEE, 2023.

[5] S. V. Mahadevkar, B. Khemani, S. Patil, K. Kotecha, D. R. Vora, A. Abraham, and L. A. Gabralla, "A review on machine learning styles in computer vision—techniques and future directions," *IEEE Access*, vol. 10, pp. 107293–107329, 2022.

[6] N. Mohammadi Foumani, L. Miller, C. W. Tan, G. I. Webb, G. Forestier, and M. Salehi, "Deep learning for time series classification and extrinsic regression: A current survey," *ACM Computing Surveys*, vol. 56, pp. 1–45, Apr. 2024.

[7] D. W. Otter, J. R. Medina, and J. K. Kalita, "A survey of the usages of deep learning for natural language processing," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 32, pp. 604–624, Feb. 2021.

[8] P. Dhilleswararao, S. Boppu, M. S. Manikandan, and L. R. Cenkeramaddi, "Efficient hardware architectures for accelerating deep neural networks: Survey," *IEEE Access*, vol. 10, pp. 131788–131828, 2022.

[9] Hugging Face, "Hugging face: The ai community building the future." https://huggingface.co. Accessed: 2024-12-08.

[10] TorchVision Contributors, "Torchvision: Datasets, transforms, and models for computer vision." https://pytorch.org/vision/, 2024. Accessed: 2024-12-08.

[11] Ultralytics, "Ultralytics yolo: State-of-the-art object detection models." https://github.com/ultralytics/yolov5, 2024. Accessed: 2024-12-08.

[12] O. Community, "Onnx: Open neural network exchange." https://onnx.ai, 2024. Accessed: 2024-12-08.

[13] N. Corporation, "Nvidia tensorrt: High-performance deep learning inference." https://developer.nvidia.com/tensorrt, 2024. Accessed: 2024-12-08.

[14] I. Corporation, “Openvino toolkit: Optimize and deploy ai inference.” https://www.intel.com/content/www/us/en/developer/tools/openvino-toolkit.html, 2024. Accessed: 2024-12-08.

[15] V. Franzoni, “Cross-domain synergy: Leveraging image processing techniques for enhanced sound classification through spectrogram analysis using cnns,” *Journal of Autonomous Intelligence*, vol. 6, Aug. 2023.

[16] J. Xu, L. Zhao, S. Zhang, C. Gong, and J. Yang, “Multi-task learning for object keypoints detection and classification,” *Pattern Recognition Letters*, vol. 130, pp. 182–188, 2020. Image/Video Understanding and Analysis (IUVA).

[17] T. Elsken, J. H. Metzen, and F. Hutter, “Neural architecture search: A survey,” in *JMLR*, 2018.

[18] S. Han, H. Mao, and W. J. Dally, “Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding,” in *ICLR*, 2016.

[19] B. Haas, A. Wendt, A. Jantsch, and M. Wess, “Neural network compression through shunt connections and knowledge distillation for semantic segmentation problems,” in *AIAI*, vol. 627 of *IFIP Advances in Information and Communication Technology*, pp. 349–361, Springer, 2021.

[20] T.-J. Yang, Y.-H. Chen, and V. Sze, “Designing energy-efficient convolutional neural networks using energy-aware pruning,” *CVPR*, July 2017.

[21] O. Bekhelifi and N.-E. Berrached, “On optimizing deep neural networks inference on cpus for brain-computer interfaces using inference engines,” in *2024 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5, IEEE, May 2024.

[22] H. Cai, L. Zhu, and S. Han, “ProxylessNAS: Direct neural architecture search on target task and hardware,” in *ICLR Poster*, 2019.

[23] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, “Efficient processing of deep neural networks: A tutorial and survey,” *Proceedings of the IEEE*, vol. 105, pp. 2295–2329, Dec. 2017.

[24] L. Mei, H. Liu, T. Wu, H. E. Sumbul, M. Verhelst, and E. Beigne, “A uniform latency model for dnn accelerators with diverse architectures and dataflows,” in *DATE*, IEEE, Mar. 2022.

[25] S. Miraliev, S. Abdigapporov, V. Kakani, and H. Kim, “Real-time memory efficient multitask learning model for autonomous driving,” *IEEE Trans. Intell. Veh.*, vol. 9, no. 1, pp. 247–258, 2024.

[26] S. Williams, A. Waterman, and D. A. Patterson, “Roofline: an insightful visual performance model for multicore architectures,” *Commun. ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[27] M. Lechner and A. Jantsch, “Blackthorn: Latency estimation framework for cnns on embedded nvidia platforms,” *IEEE Access*, vol. 9, pp. 110074–110084, 2021.

[28] D. Schnöll, D. Dallinger, M. Wess, M. Bittner, and A. Jantsch, “Towards optimal implementations of neural networks on micro-controller,” in *Presented at Workshop on IoT, Edge, and Mobile for Embedded Machine Learning at ECML-PKDD*, 2024.

[29] K. Lübeck, A. L. Jung, F. Wedlich, M. M. Müller, F. N. Peccia, F. Thömmes, J. Steinmetz, V. Biermaier, A. Frischknecht, P. P. Bernardo, and O. Bringmann, "Automatic generation of fast and accurate performance models for deep neural network accelerators," *CoRR*, vol. abs/2409.08595, 2024.

[30] A. Samajdar, J. M. Joseph, Y. Zhu, P. N. Whatmough, M. Mattina, and T. Krishna, "A systematic methodology for characterizing scalability of DNN accelerators using scale-sim," in *ISPASS*, pp. 58–68, IEEE, 2020.

[31] Y. Braatz, D. S. Rieber, T. Soliman, and O. Bringmann, "Simpyler: A compiler-based simulation framework for machine learning accelerators," in *ASAP*, pp. 213–220, IEEE, 2023.

[32] K. G. Mills, F. X. Han, J. Zhang, F. Chudak, A. S. Mamaghani, M. Salameh, W. Lu, S. Jui, and D. Niu, "GENNAPE: towards generalized neural architecture performance estimators," in *AAAI*, pp. 9190–9199, AAAI Press, 2023.

[33] L. L. Zhang, S. Han, J. Wei, N. Zheng, T. Cao, Y. Yang, and Y. Liu, "nn-meter: towards accurate latency prediction of deep-learning model inference on diverse edge devices," in *MobiSys*, MobiSys '21, pp. 81–93, ACM, June 2021.

[34] Y. Chai, D. Tripathy, C. Zhou, D. Gope, I. Fedorov, R. Matas, D. Brooks, G.-Y. Wei, and P. Whatmough, "Perfsage: Generalized inference performance predictor for arbitrary deep learning models on edge devices," *CoRR*, vol. abs/2301.10999, 2023.

[35] K. Panner Selvam and M. Brorsson, "DIPPM: A deep learning inference performance predictive model using graph neural networks," in *Lecture Notes in Computer Science* (J. Cano, M. D. Dikaiakos, G. A. Papadopoulos, M. Pericàs, and R. Sakellariou, eds.), vol. 14100 of *Lecture Notes in Computer Science*, pp. 3–16, Springer Nature Switzerland, 2023.

[36] Z. Wang, P. Yang, L. Hu, B. Zhang, C. Lin, W. Lv, and Q. Wang, "Slapp: Subgraph-level attention-based performance prediction for deep learning models," *Neural Networks*, vol. 170, pp. 285–297, Feb. 2024.

[37] A. L. Jung, J. Steinmetz, J. Gietz, K. Lübeck, and O. Bringmann, "It's all about PR - smart benchmarking AI accelerators using performance representatives," *CoRR*, vol. abs/2406.08330, 2024.

[38] S. Liu, W. Zhou, Z. Zhou, B. Guo, M. Wang, C. Fang, Z. Lin, and Z. Yu, "Deep learning inference on heterogeneous mobile processors: Potentials and pitfalls," in *Workshop on Adaptive AIoT Systems at MobiSys*, pp. 1–6, ACM, 2024.

[39] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Q. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: an automated end-to-end optimizing compiler for deep learning," in *OSDI* (A. C. Arpaci-Dusseau and G. Voelker, eds.), pp. 578–594, USENIX Association, 2018.

[40] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. G. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *CVPR*, pp. 2704–2713, Computer Vision Foundation / IEEE Computer Society, 2018.

[41] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural networks," in *NIPS*, pp. 1135–1143, 2015.

[42] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *NIPS*, pp. 3123–3131, 2015.

[43] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, "HAQ: hardware-aware automated quantization with mixed precision," in *CVPR*, pp. 8612–8620, Computer Vision Foundation / IEEE, 2019.

[44] H. Wu, P. Judd, X. Zhang, M. Isaev, and P. Micikevicius, "Integer quantization for deep learning inference: Principles and empirical evaluation," *CoRR*, vol. abs/2004.09602, 2020.

[45] V. Vanhoucke, A. W. Senior, and M. Z. Mao, "Improving the speed of neural networks on cpus," in *Workshop on deep learning and unsupervised feature learning at NIPS*, 2011.

[46] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *ISCA*, pp. 1–12, ACM, 2017.

[47] D. D. Kalamkar, D. Mudigere, N. Mellempudi, D. Das, K. Banerjee, S. Avancha, D. T. Vooturi, N. Jammalamadaka, J. Huang, H. Yuen, J. Yang, J. Park, A. Heinecke, E. Georganas, S. Srinivasan, A. Kundu, M. Smelyanskiy, B. Kaul, and P. Dubey, "A study of BFLOAT16 for deep learning training," *CoRR*, vol. abs/1905.12322, 2019.

[48] M. Cococcioni, F. Rossi, E. Ruffaldi, S. Saponara, and B. D. de Dinechin, "Novel arithmetics in deep neural networks signal processing for autonomous driving: Challenges and opportunities," *IEEE Signal Process. Mag.*, vol. 38, no. 1, pp. 97–110, 2021.

[49] W. Chen, H. Qiu, J. Zhuang, C. Zhang, Y. Hu, Q. Lu, T. Wang, Y. Shi, M. Huang, and X. Xu, "Quantization of deep neural networks for accurate edge computing," *JTEC*, vol. 17, no. 4, pp. 54:1–54:11, 2021.

[50] T. Zebin, P. J. Scully, N. Peek, A. J. Casson, and K. B. Ozanyan, "Design and implementation of a convolutional neural network on an edge computing smartphone for human activity recognition," *IEEE Access*, vol. 7, pp. 133509–133520, 2019.

[51] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, "A survey of quantization methods for efficient neural network inference," *CoRR*, vol. abs/2103.13630, 2021.

[52] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *CVPR*, pp. 770–778, IEEE, June 2016.

[53] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient convolutional neural networks for mobile vision applications," *ICLR*, 2017.

[54] Y. Cheng, D. Wang, P. Zhou, and T. Zhang, "Model compression and acceleration for deep neural networks: The principles, progress, and challenges," *IEEE Signal Processing Magazine*, vol. 35, no. 1, pp. 126–136, 2018.

[55] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals, "Understanding deep learning (still) requires rethinking generalization," *Commun. ACM*, vol. 64, no. 3, pp. 107–115, 2021.

[56] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *JMLR*, vol. 18, no. 1, pp. 6869–6898, 2017.

[57] R. Gray and D. Neuhoff, "Quantization," *IEEE Transactions on Information Theory*, vol. 44, no. 6, pp. 2325–2383, 1998.

[58] M. Nagel, R. A. Amjad, M. van Baalen, C. Louizos, and T. Blankevoort, "Up or Down? Adaptive Rounding for Post-Training Quantization," in *ICML*, vol. 119 of *Proceedings of Machine Learning Research*, pp. 7197–7206, PMLR, 2020.

[59] Z. Liu, Y. Wang, K. Han, W. Zhang, S. Ma, and W. Gao, "Post-training quantization for vision transformer," in *NIPS* (M. Ranzato, A. Beygelzimer, Y. N. Dauphin, P. Liang, and J. W. Vaughan, eds.), pp. 28092–28103, 2021.

[60] S. K. Esser, J. L. McKinstry, D. Bablani, R. Appuswamy, and D. S. Modha, "Learned step size quantization," in *ICLR*, OpenReview.net, 2020.

[61] J. Choi, Z. Wang, S. Venkataramani, P. I. Chuang, V. Srinivasan, and K. Gopalakrishnan, "PACT: parameterized clipping activation for quantized neural networks," *CoRR*, vol. abs/1805.06085, 2018.

[62] D. Schnöll, M. Wess, M. Bittner, M. Götzinger, and A. Jantsch, "Fast, quantization aware DNN training for efficient HW implementation," in *DSD*, pp. 700–707, IEEE, 2023.

[63] S. Kim, G. Park, and Y. Yi, "Performance evaluation of INT8 quantized inference on mobile gpus," *IEEE Access*, vol. 9, pp. 164245–164255, 2021.

[64] R. Gong, X. Liu, S. Jiang, T. Li, P. Hu, J. Lin, F. Yu, and J. Yan, "Differentiable soft quantization: Bridging full-precision and low-bit neural networks," in *ICCV*, pp. 4851–4860, IEEE, 2019.

[65] W. Chen, P. Wang, and J. Cheng, "Towards mixed-precision quantization of neural networks via constrained optimization," in *ICCV*, pp. 5330–5339, IEEE, 2021.

[66] R. Singhal, A. Biswas, S. Elangovan, and S. Sabnis, "Learning bit multipliers for non-uniform quantization," 2024.

[67] E. Kavvousanos, I. Kouretas, V. Paliouras, and T. Stouraitis, "A regularization approach to maximize common subexpressions in neural network weights," in *30th IEEE International Conference on Electronics, Circuits and Systems, ICECS 2023, Istanbul, Turkey, December 4-7, 2023*, pp. 1–4, IEEE, 2023.

[68] A. Biswas and U. Ganguly, "QFALT: quantization and fault aware loss for training enables performance recovery with unreliable weights," in *International Joint Conference on Neural Networks, IJCNN 2024, Yokohama, Japan, June 30 - July 5, 2024*, pp. 1–6, IEEE, 2024.

[69] E. Li, L. Zeng, Z. Zhou, and X. Chen, "Edge AI: on-demand accelerating deep neural network inference via edge computing," *IEEE Trans. Wirel. Commun.*, vol. 19, no. 1, pp. 447–457, 2020.

[70] S. Wu, H. Yang, X. You, R. Gong, Y. Liu, Z. Luan, and D. Qian, "Proof: A comprehensive hierarchical profiling framework for deep neural networks with roofline analysis," in *ICPP*, vol. 4 of *ICPP '24*, pp. 822–832, ACM, Aug. 2024.

[71] K. Iyer and J. Kiel, "GPU Debugging and Profiling with NVIDIA Parallel Nsight," 2016.

[72] NXP Semiconductors, "i.MX 93 Family of Applications Processors." https://www.nxp.com/products/processors-and-microcontrollers/arm-processors/i-mx-applications-processors/i-mx-9-processors/i-mx-93-applications-processor-family-arm-cortex-a55-ml-acceleration-power-efficient-mpu:i.MX93, 2022. Accessed: 2024-12-20.

[73] Intel Corporation, “Intel Neural Compute Stick 2.” https://www.intel.com/content/www/us/en/developer/tools/neural-compute-stick/overview.html. Accessed: 2024-12-20.

[74] Google, “Coral Edge TPU.” https://coral.ai/products/accelerator/. Accessed: 2024-12-20.

[75] NXP Semiconductors, “i.MX 8 Series Applications Processors.” https://www.nxp.com/products/processors-and-microcontrollers/arm-processors/i.mx-applications-processors/i.mx-8-processors:IMX8-SERIES. Accessed: 2024-12-20.

[76] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, “Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks,” *JSSC*, vol. 52, pp. 127–138, Jan. 2017.

[77] NVIDIA Corporation, “NVIDIA Jetson AGX Xavier.” https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-agx-xavier/. Accessed: 2024-12-20.

[78] Y. Chai, J. Gkountouras, G. G. Ko, D. Brooks, and G. Wei, “INT2.1: towards fine-tunable quantized large language models with error correction through low-rank adaptation,” *CoRR*, vol. abs/2306.08162, 2023.

[79] T. B. Brown *et al.*, “Language models are few-shot learners,” in *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual* (H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, eds.), 2020.

[80] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: pre-training of deep bidirectional transformers for language understanding,” in *NAACL-HLT* (J. Burstein, C. Doran, and T. Solorio, eds.), pp. 4171–4186, Association for Computational Linguistics, 2019.

[81] Ollama Developer Community, “Ollama.” https://github.com/jmorganca/ollama, 2023. Accessed: 2024-12-21.

[82] ggerganov and Contributors, “llama.cpp.” https://github.com/ggerganov/llama.cpp, 2023. Accessed: 2024-12-21.

[83] Z. Xu, Y. Yue, X. Hu, Z. Yuan, Z. Jiang, Z. Chen, J. Yu, C. Xu, S. Zhou, and D. Yang, “Mambaquant: Quantizing the mamba family with variance aligned rotation methods,” 2025.

Erklärung zur Verfassung der Arbeit

Hiermit erkläre ich, dass die vorliegende Arbeit gemäß dem Code of Conduct – Regeln zur Sicherung guter wissenschaftlicher Praxis (in der aktuellen Fassung des jeweiligen Mitteilungsblattes der TU Wien), insbesondere ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel, angefertigt wurde. Die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet. Die Arbeit wurde bisher weder im In– noch im Ausland in gleicher oder in ähnlicher Form in anderen Prüfungsverfahren vorgelegt.

Vienna, Austria, 19.03.2025

Matthias Wess

IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received December 11, 2020, accepted December 19, 2020, date of publication December 24, 2020,
date of current version January 7, 2021.

Digital Object Identifier 10.1109/ACCESS.2020.3047259

ANNETTE: Accurate Neural Network Execution Time Estimation With Stacked Models

MATTHIAS WESS[1,2], MATVEY IVANOV[1,2], CHRISTOPH UNGER[1], ANVESH NOOKALA[1], ALEXANDER WENDT[1,2], (Member, IEEE), AND AXEL JANTSCH[1,2], (Senior Member, IEEE)
[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 (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 With new accelerator hardware for Deep Neural Networks (DNNs), the computing power for Artificial Intelligence (AI) applications has increased rapidly. However, as DNN algorithms become more complex and optimized for specific applications, latency requirements remain challenging, and it is critical to find the optimal points in the design space. To decouple the architectural search from the target hardware, we propose a time estimation framework that allows for modeling the inference latency of DNNs on hardware accelerators based on mapping and layer-wise estimation models. The proposed methodology extracts a set of models from micro-kernel and multi-layer benchmarks and generates a stacked model for mapping and network execution time estimation. We compare estimation accuracy and fidelity of the generated mixed models, statistical models with the roofline model, and a refined roofline model for evaluation. We test the mixed models on the ZCU102 SoC board with Xilinx Deep Neural Network Development Kit (DNNDK) and Intel Neural Compute Stick 2 (NCS2) on a set of 12 state-of-the-art neural networks. It shows an average estimation error of 3.47% for the DNNDK and 7.44% for the NCS2, outperforming the statistical and analytical layer models for almost all selected networks. For a randomly selected subset of 34 networks of the NASBench dataset, the mixed model reaches fidelity of 0.988 in Spearman's ρ rank correlation coefficient metric.

INDEX TERMS Analytical models, estimation, neural network hardware.

I. INTRODUCTION

Deep Neural Networks have become key components in many AI applications, including autonomous driving [1], medical diagnosis [2], [3] and machine translation [4]. The computational intensity of some AI applications based on DNNs prevents their use on embedded system platforms, as these algorithms often have to meet latency and performance requirements to fulfill their purpose.

Attempting to close the gap between the computational intensity of DNNs and the available computing power, a wide variety of hardware accelerators for DNNs and other AI workloads have emerged in recent years. A considerable amount of research has improved the efficiency of DNNs and reduced their memory consumption by applying methods such as pruning [5], [6], quantization [7]–[9], and factorization [10], [11]. Alternatively, a network architecture that is expected to work efficiently on the target device can be designed and trained directly. Networks like MobileNet [12] and ShuffleNet [13] are specifically designed to reduce the number of Multiply-Accumulate operations (MACs), but they contain specific layer types that are not necessarily optimal for all hardware types. In addition, computational efficiency depends largely on the specific architectural parameters of each layer and the hardware platform used [14].

Finally, also the mapping toolchain optimizing the original network graph for the selected hardware platform has to be considered since many hardware accelerators allow specific combinations of layers to be fused together to reduce inter-layer data transfer and/or to optimize data flow. Therefore, when optimizing the network architecture towards ''direct metrics'' such as latency or energy consumption, ''indirect metrics'' such as Floating Point Operation (FLOP) or memory footprint can serve as a starting point but do not take into account the platform-specific non-linearities. As a result, networks optimized towards ''direct metrics''

The associate editor coordinating the review of this manuscript and approving it for publication was Barbara Masini.

considerably outperform ''indirect'' optimized architectures in terms of the selected metrics [14], [15]. On the one hand, the enormous design space for neural network architectures makes it difficult to design a network that runs at high efficiency on all hardware architectures. On the other hand, not all networks work with the same efficiency on a given platform. For example, Fig. 1 shows the effective compute performance when running 12 networks used for evaluation in this paper on a ZCU102 Xilinx MPSoC evaluation board. Furthermore, the computational roofline shows the maximum reachable, effective compute performance.

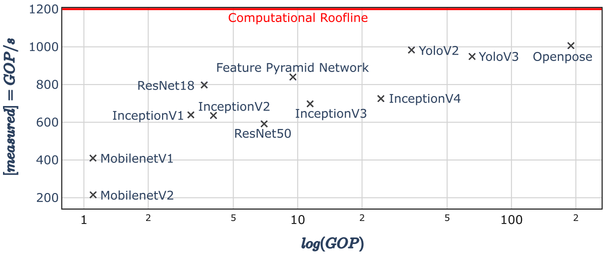


FIGURE 1. Effective compute performance when inferring the DNNs from Table 2 on the Xilinx ZCU102 evaluation board.

We can see the high variance of the effective compute performance for a variety of different network architectures when executed on the same hardware. Due to the large differences in effective compute performance, we can conclude that it is not sufficient to divide the number of operations of a network by the peak compute performance of the target device to achieve a satisfying estimation of the network execution time. So when aiming for field deployment, it is difficult to choose a specific hardware platform before deciding on the network architecture. As a result, there have been some recent attempts to predict network latency and performance on different hardware platforms. However, most of the work targets either Graphic Processing Units (GPUs) [16], [17] server or the embedded Central Processing Units (CPUs) [18], [19], leaving out a wide range of hardware accelerators such as Field Programmable Gate Arrays (FPGAs) and hardware specifically designed for AI tasks e.g. Xilinx ZCU102 and Intel NCS2. In this work we aim to model performance of such DNN hardware accelerators. Also, the existing work does not take into account the graph optimizations undertaken by the compiler, which leads to changes in the accuracy of the prediction.

Therefore, we propose a framework for the generation of stacked, mapping models and layer models to estimate the network execution time. To our knowledge, this is also the first work in which the different approaches to modeling layer execution time and mapping models are systematically investigated and evaluated on a broad range of network architectures.

This paper makes the following key contributions:

- We introduce Accurate Neural Network Exectution Time Estimation (ANNETTE), a time estimation framework that allows predicting the execution time of Deep Neural Networks on hardware accelerators based on a stacked modeling approach of mapping models and layer-wise estimation models
- We propose mixed models for layer execution modeling to decrease the necessary model complexity of the statistical models to cover also computational utilization inefficiencies
- We propose a methodology to extract mapping models and layer execution models from micro-kernel and multi-layer benchmarks. Our evaluation of the generated mapping models and layer models on a set of 12 state-of-the-art models show a mean absolute percentage error of 3.41% for the ZCU102
- We compare mixed layer models with statistical layer models, the roofline model, and a refined roofline model in terms of accuracy and fidelity

II. RELATED WORK

Several studies have been performed to measure how well certain DNNs perform on different hardware. Their purpose is to explore the design space and to get the highest efficiency out of the hardware. In EmBench [20], common DNNs like ResNet, ShuffleNet, and MobileNet were tested on a wide range of hardware, ranging from power consuming server hardware like the NVIDIA GeForce RTX 2080 Ti GPU to mobile devices like the Intel NCS2. A key finding in EmBench was the Pareto curve of accuracy and latency of different networks on the hardware devices. Often, it depends on the type of layers used in the respective networks. While they tested all different combinations of networks and hardware, our work takes another approach. For each hardware, we provide a method that measures latency for each layer type and then estimates the latency of a whole composed network. Together with known accuracies of the architectures in NASBench [21], we can then explore the Pareto curve of a specific hardware platform without further measurements.

MLPerf [22] is an attempt by over 30 organizations to create an industry-wide standard benchmark to assess the vast number of machine learning software and hardware combinations, while DAWNBench [23] is led by academia. MLPerf limits the problem space by defining a set of scenarios, datasets, libraries, frameworks, and metrics. Additionally, it specifies prohibited operations to enhance comparability under equal terms. For our statistical model, MLPerf could provide additional measurements to align it for new hardware and enhance our measurements. However, the available data does not suffice to construct accurate mapping models and layer models.

Besides characterizing accelerator hardware, hardware optimized neural architecture search (NAS) is becoming increasingly popular and powerful. While handcrafted cells of ResNet and Inception lie close to the Pareto optimum at GPUs [21], the design space for mobile devices is very large [24]. It offers potential for automated architecture search, especially when the demand for customized networks rises. FBnetV3 [25], SqueezeNAS [26] and

Proxyless NAS [15] focus on low-latency network architecture search for mobile devices. They were developed to replace costly redesign DNNs for certain tasks on certain platforms. While SqueezeNAS focus on semantic segmentation, FBNetV3 and Proxyless NAS focus on classification tasks. Both tools show superior latency-accuracy tradeoffs compared to MobileNet. SqueezeNAS, as well as Proxyless NAS, first generate a super network, in which each cell is selected from a search space. They approximate latencies by building look-up tables for the selected blocks within the design space to save time. All three works could profit from a uniform estimation framework that accurately predicts performance for multiple platforms. In NetAdapt [14], empirical measurements on a Google Pixel 1 CPU are used to construct layer-wise look-up tables to shrink a pre-trained MobileNetV1 until the resource constraints are met to optimize DNNs for inference on mobile devices. FBNetV3 uses multi-use predictors to power their neural architecture search algorithm by predicting architecture statistics such as accuracy and the proxy metrics FLOPS and number of parameters.

NeuralPower [17] is an attempt to estimate execution latency, power, and as a result, overall energy consumption based on layer-wise sparse polynomial regression for GPU platforms. In terms of execution time estimation, NeuralPower achieves an average accuracy of 88.24% on the networks VGG-16, AlexNet, NIN, Overfeat, CIFAR10-6conv. In addition to the layer-wise time estimation, the same modeling method is also applied to estimate power and finally energy consumption with even higher accuracy. Fast-DeepIoT [18] uses execution time models based on linear model trees to predict the layer execution time on the devices Nexus 5 and Galaxy Nexus to finally compress VGGNet for both devices and reduce the neural network execution time by 48% to 78% and energy consumption by 37% to 69% compared with the state-of-the-art compression algorithms. In PreVIous [19], the execution time models are based on linear regression, and for the devices, Raspberry 3 and Odroid-XU4 reaches about 96% average accuracy for the layer-wise estimation. These results lead us to believe that the task of estimating layer execution times for task optimized computing architectures is significantly more challenging than for CPUs. Therefore, we propose a methodology for generating stacked mapping and layer execution time models for hardware accelerators and systematically compare the prediction accuracy of different modeling approaches. Other than that, MLPAT [27] and DNN-Chip Predictor [28] propose white box approaches to estimate timing, power and energy. MLPAT reports only 10% error when predicting the power of the TPU-v1. DNN-Chip Predictor's predicted performance differs from those of measurements of FPGA/ASIC implementation by no more than 17.6% when evaluated for two DNNs on three accelerator architectures.

III. ARCHITECTURE

Fig. 2 shows an overview of the proposed framework, allowing us to generate abstraction models for the hardware platform and the mapping toolchain. The **Benchmark Tool** generates networks, which are then optimized by the provided mapping toolchain for the selected platform. During the benchmark phase, we execute the generated models on the target device and extract detailed layer execution times. We rely on the provided platform tools for mapping, inference, and profiling. With the collected profiling information, the **Model Generator** can create abstraction models of the graph optimizations and the different layer types. These abstraction models are used in our **Estimation Tool** to predict the performance of a network without compiling and executing the model. Furthermore, detailed insights are gained to produce efficient networks for the modeled hardware devices.

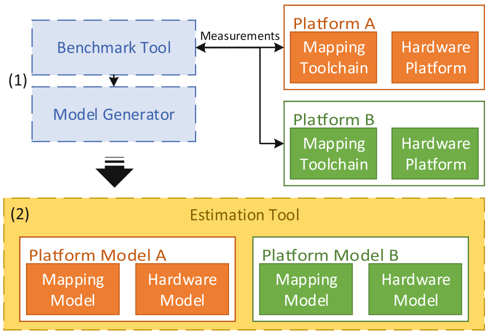


FIGURE 2. Overview of the Annette architecture: In the benchmark phase (1), first the platform benchmarks are performed and then the platform models are generated. In the estimation phase (2), the Estimation Tool reads a network description graph, and provides an estimated network execution time, a detailed layer-wise execution time prediction table, and a predicted execution graph.

IV. BENCHMARK TOOL

For platform characterization, we make use of two kinds of benchmarks: micro-kernel benchmarks and multi-layer benchmarks. We aim to characterize the computational efficiency of a hardware platform when executing only a specific layer with the micro-kernel benchmarks. On the other hand, multi-layer benchmarks give us a deeper understanding of which kind of layers are executed separately and which layers can be fused, reducing the off-chip data movement. Fig. 3 depicts the workflow of the Benchmark Tool.

We define a benchmark as one parametric network graph profiled several times on the hardware with different input resolutions or kernel-sizes. Each generated network architecture stays the same for each benchmark, while only the layer parameter settings (e.g. number of channels and kernel size) are changed according to the configuration file. The input for each benchmark is a configuration and a graph description file. The configuration file defines the parameter settings of each measurement. The graph description file defines the architecture of the benchmarked dummy network. The Graph Generator module builds the network models based on the description and configuration information and feeds it to the hardware-specific modules. In each hardware module, the network graph is initially optimized and compiled by

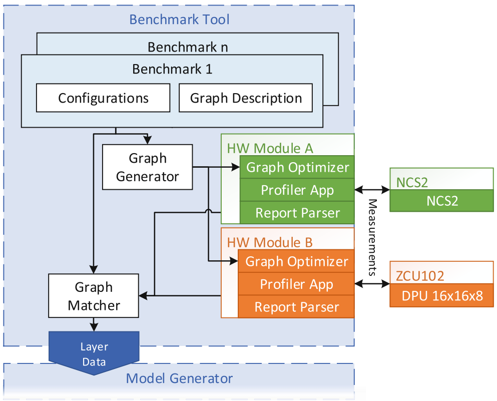


FIGURE 3. The Benchmark Tool profiles the provided set of benchmark models with different configuration settings on the hardware platforms and generates layer data files.

the platform mapping toolchain. The optimized graph is then inferred on the target device in a platform specific benchmark application. Then, a report is generated with the help of the platform profiling tool. Finally, the report is parsed into a standard format, and the **Graph Matcher** compares the collected layer data with the original input network.

Running each benchmark separately is a time-consuming task of about three to five days per benchmark, as each model must go through the entire compilation toolchain before the desired measurements can be made. Therefore, we have developed some network models that allow us to measure several kernels within a benchmark run. In this case, the models must be constructed in a particular manner so that the compiler cannot fuse layers or that the computational effort for an operation is not increased. It would result in measuring more than just the desired micro-kernel. Those linear network graphs still count as micro-kernel benchmarks since we still measure the execution time of each layer individually. It must be taken into account that when using graphs with more than one layer for micro-kernel benchmarking, the maximum allowable layer size may be smaller than when measuring a single layer. The choice of configuration parameters has an additional influence on the benchmark wall time. It also influences the insights that can be gained from the collected data and on the understanding of how to model the accelerator This topic is discussed in Section V.

We use the same mechanism for the multi-layer benchmarks, with the difference that they have more configuration parameters. The goal of these benchmarks is primarily to model the mapping toolchain and understand which optimizations the graph optimization toolchain can perform, but also to be able to benchmark multiple layers at the same time.

The Graph Generator builds the network models, which are benchmarked on the target platform, based on the graph description and the configurations table. It iterates through the configurations table generating one network model per parameter setting. We apply micro-kernel benchmarks for 2D convolution, 2D depth-wise separable convolution, max pooling, average pooling, and fully connected layers, with values in the range from 8 to 2048 for height (h), width (w), number of input channels (c), number of filters (f), input and output neurons, kernel sizes (k_h, k_w) 1, 3, 5, and 7, and pooling sizes from 2 to 10, resulting in a total of about 35k measurements per layer. Figure 4 illustrates the network architectures used for the multi-layer benchmarks. All convolution layers are followed by batch normalization and ReLU layers.

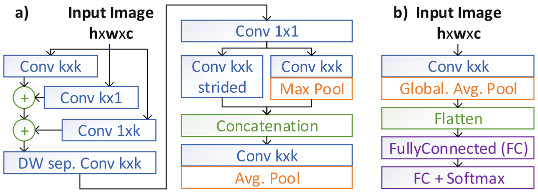


FIGURE 4. The multi-layer benchmark networks (a) ANNETTE ConvNet for characterizing convolution and pooling layers; (b) ANNETTE FCNet for benchmarking global average pooling and fully connected layers.

The **Hardware Modules** are simple scripts that automatically call the platform optimization (Graph Optimizer) and compilation toolchain to prepare the benchmark models for inference. In the case of DNNDK, as a developing framework for the hardware module Deep Neural Network Processing Unit (DPU) on the ZCU102 MPSoC board, optimization and compilation functionality are provided through the Deep Compression Tool (DECENT) and the Deep Neural Network Compiler (DNNC) respectively [29]. In the case of NCS2 the graph is optimized and compiled by the OpenVINO Toolkit [30]. Similarly, we rely on provided execution and the platform specific profiler applications (Profiler App) to extract the layer execution times for the compiled networks. To avoid measurement errors, we average the results of 20 iterations. Finally, a Report Parser extracts the layer-specific information and maps it back to the original graph, comparing the executed layers with the original layers by their names. Therefore, the Profiler App must provide execution times and layer names. The execution information is stored in a standardized format so that the Graph Matcher can process the provided data in the same way for each platform. These encapsulated hardware modules make it easy to add future hardware to the benchmarking tool.

In addition to the Report Parser, the Graph Matcher extracts information about the differences between the original input graph and the final net graph executed on the target device. While the parser merely ensures that there are no changes to the original naming scheme and provides a standardized output, the Graph Matcher extracts additional information about the optimization behavior of the mapping toolchain. The Graph Matcher creates a layer result file for each executed layer and an optimization mapping file for the entire benchmark. The layer result files contain information about the layer parameters, e.g., height, width, number of input channels, and the resulting execution times. To track the behavior of the mapping toolchain, we also store ternary variables that successive layers have been fused with the measured layer. This merging variable can store the following states: not-fused, fused, and possibly-fused. Possibly-fused is used because, it is not possible to detect where the layer has been merged or not for layers with multiple inputs.

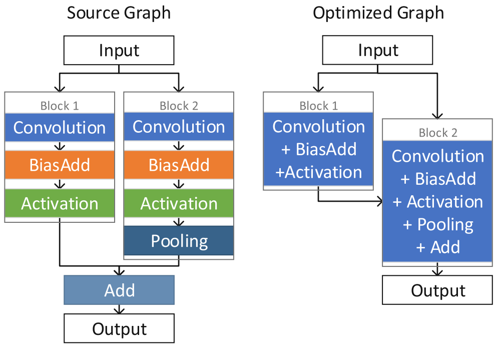


FIGURE 5. Graph optimization.

Fig. 5 shows an example of how a graph could be optimized by the mapping toolchain. In the specific example we set the fused flags for *BiasAdd, Activation* operation in the *Convolution* layer of block 1 and 2 to *fused*. In block 2, the fused flag for the *Pooling* operation is set to *fused* as well. Here, it is important to note that since the pooling layer also has a set of parameters, i.e., pooling height, pooling width, pooling stride, and pooling type that define its execution policy, we also need to add those parameters to the already existent stored parameters for the convolution layer. It enables the graph optimizer modeler to extract rules that define in the case of which parameter combinations the layers can be fused. Since the element-wise addition layer may have been matched to either block 1 or block 2, the fused flag for the *Add* operation is set to *possibly-fused* in both blocks.

The generated layer data consists of a table for each layer type that for each measurement contains the parameter settings of the layer e.g. height, width, channels, kernel size as well as the measured execution time. This data is then fed to the Model Generator to extract optimization and layer models for the final estimation step.

V. MODEL GENERATOR

This section explains how we model the graph optimizations of the mapping toolchain and the computational efficiency of the hardware platforms to achieve better overall latency estimation accuracy. As depicted in Fig. 1, not all networks are computed with the same efficiency when compared to the number of operations in the convolution layers. There are two leading causes of the non-linear nature of the relationship between the number of operations and execution time. First, the non-convolutional layers cannot be neglected. They are not considered in the commonly claimed number of operations, such as element-wise addition, concatenation, activation, or pooling. It is crucial for the execution time of these layers, whether they are executed in isolation or connection with a convolution layer [31]. The second factor is that the utilization of computational resources for the same layer can depend on the parameter settings on a specific layer (e.g., height, width). It means that two compute-bound layers with the same number of operations but with differently shaped input and weight tensors are not necessarily computed with the same efficiency [18].

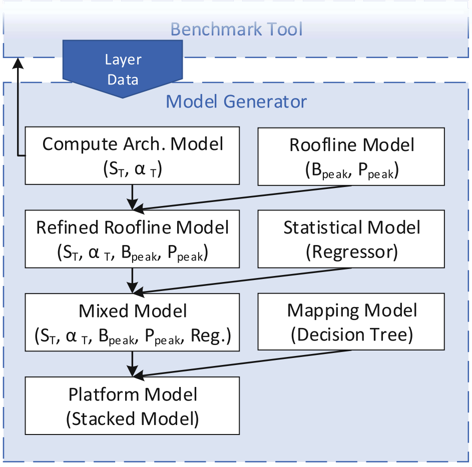

Benchmark Tool
Layer Data
Model Generator
Compute Arch. Model (S_T, α_T)
Roofline Model (B_{peak}, P_{peak})
Refined Roofline Model (S_T, α_T, B_{peak}, P_{peak})
Statistical Model (Regressor)
Mixed Model (S_T, α_T, B_{peak}, P_{peak}, Reg.)
Mapping Model (Decision Tree)
Platform Model (Stacked Model)

FIGURE 6. The model generator, extracts a stacked model consisting of mapping models and layer models.

To cover all these aspects, we propose a stacked model approach to model the overall network execution time accurately. Fig. 6 shows how the different models are fused for the generation of the platform model. Tab. 1 describes the parameters for the models extracted from the benchmarks. The first performed benchmarks are input parameter sweeps to determine the unrolling parameters $\vec{s}$ and $\vec{\alpha}$. These parameters describe the amount of parallel performed multiplications in per dimension of the compute architecture and the parallelization efficiency. With the help of these two parameter vectors, we can construct a model that describes the utilization efficiency of several compute architectures (e.g. systolic arrays). Additionally, preliminary values of P_{peak} and B_{peak} are determined, which describe the peak performance and the peak off-chip bandwidth.

TABLE 1. Model parameters.

Parameter	Description
HW par.	
P_{peak}	Peak performance of the Architecture (ops/sec)
B_{peak}	Peak off-chip bandwidth (byte/sec)
$\vec{s}$	Spatial unrolling parameter vector of the Architecture
$\vec{\alpha}$	Spatial unrolling efficiency coefficient vector
SW par.	
$\vec{x}_n$	Input feature vector describing layer n
f_n	Number of operations in layer n (ops)
D_n	Data transferred in layer n (bytes)
Interm.	
u_{eff_n}	Utilization efficiency of comp. resources in layer n
u_{stat_n}	Statistically computed efficiency in layer n
P_{eff_n}	Effective performance in layer n (ops/sec)
Predictions	
$\hat{T}_n$	Estimated time of layer n (sec)

These parameters are determined automatically based on measurements or knowledge of the computing architecture. Once determined, the parameters are fed back to the Benchmark Tool to adjust the parameter settings for the succeeding benchmarks. The rest of the micro-kernel benchmark results are used to generate the **Roofline Model** by deducing the final values of P_{peak} and B_{peak}, which together with the previously determined unrolling parameters, construct the **Refined Roofline Model**. We combine the **Statistical Model** and the **Refined Roofline Model** in the **Mixed Model**. For the final **Platform Model**, we add the **Mapping Model**, which covers optimizations performed on the graph before the actual execution.

A. LAYER EXECUTION TIME MODELS

For the construction of layer-level execution time models, we rely on the measurements performed in the benchmarks. We construct parametric analytical models for the *convolution*, the *depth-wise separable convolution*, the *fully connected*, and *pooling layer*. The selection of these layers is motivated, similarly as in the works [16], [17], by the fact that these are the most computational intense layers and, therefore, most critical. However, we will also show that it is also crucial for more complex network architectures to model different layers to achieve accurate results with high fidelity. While the simple roofline model describes most layers with satisfying accuracy, we refine the roofline model for the convolution layer to increase the estimation accuracy.

1) ANALYTICAL MODELS

For the estimation framework to always work with at least the most simple model, we implement the roofline model [32] for all layer types as a fallback solution. In the roofline model for each layer n, smallest achievable execution time is either limited by the peak computational performance P_{peak} or the maximal bandwidth B_{peak}. In layer n, with the data to be transferred D_n and the number of operations f_n give us the estimated execution time $\hat{T}_{roof_n}$ with the effective computation performance P_{eff} equal to P_{peak}

$$\hat{T}_{roof_n}(f_n, D_n) = max(\frac{f_n}{P_{peak}}, \frac{D_n}{B_{peak}}). \tag{1}$$

Keeping in mind that for fused layers the term of D_n has to be corrected (see Section V-B), this formulation of the roofline model can be applied to the four named layer types and will be denoted in the experimental section as roofline model.

However, as mentioned earlier, computational efficiency also depends on how the shapes of the input-, weight- and output tensors are mapped on the computing architecture. When incorporating the reduced utilization efficiency u_{eff} in equation (1) we obtain

$$\hat{T}_{ref_n}(f_n, D_n) = max(\frac{f_n}{P_{peak}u_{eff_n}}, \frac{D_n}{B_{peak}}) \tag{2}$$

Next we aim to describe the utilization efficiency of a general compute architecture with an array of Processing Elements (PEs). The number of spatial dimensions A and the number of PEs alongside each dimension $\vec{s} \in \mathbb{N}^A$ define the compute architecture. For example an array could be described with $A = 2$ and $\vec{s} = \begin{pmatrix}16 & 12\end{pmatrix}$, which amounts to a total of 192 PEs. When computing a layer, the operations have to be mapped onto the array either spatially or temporally. With the parameter settings of the layer as the feature vector $\vec{x}$ we can approximate the utilization efficiency with

$$u_{eff}(\vec{x}) = \prod_{i=1}^{A} \frac{x_i/s_i}{\lceil x_i/s_i \rceil}. \tag{3}$$

Hereby the size of the vector $\vec{x}$ does not have to match the size of the vector $\vec{s}$ as the operations can also be mapped in the temporal dimension. For example, when mapping a 2D 1 × 1 convolution layer with a 12 × 6 × 128 input feature map and 256 output channels, the feature vector describing the layer could be any permutation of $\begin{pmatrix}12 & 6 & 128 & 256 & 1 & 1\end{pmatrix}$ depending on the mapping of the layer onto the array. With equation (3), for the presented example case and the input feature map height and width mapped spatially onto the 16x 12 array, we would get $u_{eff} = 0.375$.

It has to be mentioned that equation (3) neglects the overhead of control units and warming up as well as possible input parameter augmentation for $x_i < s_i$. For example, since the first layer in most DNNs has three input channels ($x_i = 3$), channel augmentation can often improve performance in the first layer of the neural network. To allow for further adjustment of the model to different efficiencies for each element of $\vec{s}$ we add the unrolling efficiency vector $\vec{\alpha}$ to get the final utilization efficiency of the refined roofline model

$$u_{eff}(\vec{x}) = \prod_{i=1}^{A}(\alpha_i + \frac{\lceil x_i/s_i \rceil}{x_i/s_i}(1 - \alpha_i))^{-1} \tag{4}$$

where $\vec{\alpha} \in \mathbb{R}^A \mid 0 \leq \alpha_i \leq 1$. The coefficients α_i adjust the impact of the spatial unrolling. According to the terminology used in [33], α_i allows us to adjust the impact of *spatial* and *temporal fragmentation* on the overall utilization efficiency. So far, we have identified no other method to derive the values of $\vec{\alpha}$ from the system architecture than by measurement.

This refined version of the roofline model allows us to model not only the reduced utilization efficiency of n-D convolutions due to the mapping restrictions of existing compute architectures. It can also be used to model jumps in utilization efficiency caused by higher-level features such as the number of input parameters, weights, or outputs.

We apply the simple roofline model with separately measured data throughput rate and peak performance to the pooling, depth-wise separable, and fully connected layers, respectively, under the presumption that accuracy does not have to be as high as for the convolutional layers. However, it is still important to also capture the execution time of those layers. Furthermore, for fused layers, we define the first term of equation (2) as the sum of the execution time of the convolution layer and the following fused layer. For the second term, we adjust the number of transferred data to the overall amount of the fused layer. For example, a convolution layer with a succeeding pooling layer with a stride greater than one has a reduced number of output parameters.

Within the modeling framework, we determine model parameters P_{peak}, B_{peak}, $\vec{s}$ and $\vec{\alpha}$ for all layers automatically based on the measurements of the Benchmark Tool. At first, we perform sweep benchmarks to measure the layer execution time while sweeping each of the parameters describing the layer. For example, in one sweep for a 2D convolution layer, we measure the execution time, incrementing the number of input channels in each measurement. These sweeps are performed for each parameter at multiple points, while the other layer parameters are set to the same value for the entire sweep. Based on these measurements, we can extract the preliminary values of P_{peak} and B_{peak} by finding the maximum performance and data throughput values. Next we determinate the values of s_i and α_i, by fitting equation (3) to the collected data using mean square minimization, with the conditions $\vec{\alpha} \in \mathbb{R}^A \mid 0 \leq \alpha_i \leq 1$ and $\vec{s} \in \mathbb{N}^A$. Lastly with the determined values of $\vec{s}$ and $\vec{\alpha}$ we perform the rest of the benchmarks using preferably layer settings with (3) to determine the final values of P_{peak} and B_{peak}.

2) STATISTICAL MODELS

Apart from the analytical estimation model, we also generate statistical regression models to estimate the performance for all benchmarked layer types. In general, we found that the statistical models produce more precise results when predicting utilization efficiency rather than the resulting execution time. We estimate the utilization efficiency $u_{stat} = f(\vec{x})$ where $u_{stat} \in \mathbb{R} \mid 0 < u_{stat} \leq 1$ for each layer separately based on a feature vector $\vec{x}$ describing the layer's parameter settings. Similar to [17], [18] we include higher-level features such as the number of input parameters and the number of operations. For example, for the 2D convolutional layer we select the feature vector $\vec{x} = (h, w, c, f, k_h, k_w, stride, \#ops, \#in, \#out, \#weights)$.

We applied random forest regression for the statistical models of the network layers, which worked best for the data collected in the benchmarks. Although tree-based regression methods generally do not extrapolate well, they have the useful property that the output values do not explode but remain constant when the input values are outside the training data range. In the case of the u_{stat} estimate, this behavior does not degrade the quality of the estimate. For the final prediction of the layer execution time, we then apply the roofline model with statistically computed utilization efficiency:

$$\hat{T}_{mix_n}(f_n, D_n) = max(\frac{f_n}{P_{peak} u_{stat_n}}, \frac{D_n}{B_{peak_n}}) \tag{5}$$

Due to the large number of architectural parameters for the convolution layer, we have to carefully select for which configuration parameter settings to perform the measurements. This is important since the points of measurement influence the quality of the resulting statistical models. To find the best points of measurement for our statistical model, we generate three datasets. For the first dataset, we aim to model the surface of points with the best utilization efficiency. Therefore, we reduce the space of measurements to points with utilization efficiency equal to 1. For the generation of the second dataset, we add Gaussian noise to the parameters with $s_i > 1$ to also cover cases with utilization efficiency < 1. The third dataset is the union of datasets 1 and 2.

The experimental results show that, depending on the selected statistical model, too large amounts of measurement points would be required to model the entire surface of dataset 3 correctly. Therefore, we use dataset 1 for the generation of the statistical models and follow a third approach. We combine the generated statistical models with the refined roofline model from Section V-A1 to achieve higher accuracy for the points with utilization efficiency < 1.

3) MIXED MODELS

To combine the advantages of the statistical and analytical models, we also implement a mixed modeling approach by stacking the statistical model and the refined roofline model. The execution time of the mixed model $\hat{T}_{mix}$ for the layer n can be expressed as

$$\hat{T}_{mix_n}(f_n, D_n) = max(\frac{f_n}{P_{peak} u_{eff_n} u_{stat_n}}, \frac{D_n}{B_{peak}}) \tag{6}$$

Decoupling the modeling of u_{eff} and u_{stat} has the advantage that the necessary model complexity for the estimation of u_{stat} is reduced, as the model only needs to correctly estimate the points with $u_{eff_n} = 1$. Fig. 7 shows how combining the statistical model and the refined roofline model results in the mixed model.

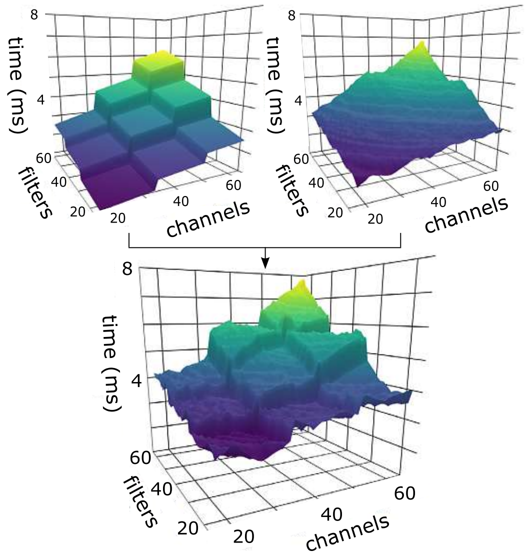


FIGURE 7. An example of predicted execution time surfaces for the refined roofline model (top left), statistical model (top right) and mixed model (bottom). The plane of the mixed model is an overlay of the refined roofline model and the statistical model.

The analytical part of the model, namely the refined roofline model, covers the step-wise linear shape of the target surface. Based on the refined roofline model, we can determine at which points we want to perform measurements for the statistical model. As mentioned in section V-A2, we only select points with $u_{eff} = 1$ for computing the regression model for u_{stat}. Therefore, the refined roofline model improves the statistical model twofold: by refining the area with $u_{eff} \neq 1$ and regarding the selection of points for the measurements.

Due to the better choice of data points, the statistical model will produce a better result with a lower risk of overfitting. This also explains why the regression model based on dataset 1 is outperforming the models with additional data points. However, thanks to the analytical part of the mixed model, we can still model the local shape of the surface. We can say that while the analytical part is responsible for modeling inefficiencies of the computational architecture, the statistical model covers the memory architecture.

B. MAPPING MODELS

The last estimation module we present is the mapping model. The main objective is to predict whether two successive layers have been fused or not. This is important for cases where $T_{total} \neq T_1 + T_2$, where T_{total} is the total execution time of layers 1 and 2; T_1 and T_2 are the execution times of the two layers when executed separately. As mentioned above, this difference is mainly due to reduced off-chip data transfer and pipelining effects. For the generation of the mapping models, we use the input feature vectors $\vec{x}$ previously defined for the statistical model and aim to predict the values of the *fused flags* extracted by the Graph Matcher in Section IV). We rely on *Decision Tree Classifiers* to determine the rules for the mapping prediction. For example, Fig. 8 shows a simplified version of the decision tree for the fusion of a convolution layer followed by a max-pooling layer. We can see that in the example shown, the decision if the two layers are merged or not depends mainly on whether a certain number of *channels* and *filters* in the convolution layer is exceeded or not. We apply the same concept to all fused layer combinations we were able to find in our evaluation networks in Tab. 2.

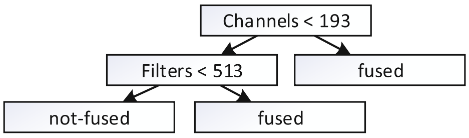


FIGURE 8. Sample decision tree for fusing pooling and convolution on NCS2.

VI. ESTIMATION

For the network level estimation, we apply the stacked model presented in Section III on a network description graph. At first, we apply the mapping models to reconstruct the mapping of the platform mapping toolchain. For this, we iterate through all directly connected layers and check whether they should be fused or not. Afterwards, we apply the layer level models on each remaining layer of the optimized graph. The network execution time estimation $\hat{T}_{total}$ is the sum of all estimated layers $\hat{T}_n$.

Because of the different models available for each layer, we implement the estimation framework in a way that we can select the preferred model type but always use the roofline model as a fallback solution so that the highest possible number of layers execution times is always estimated.

VII. RESULTS AND PERFORMANCE ANALYSIS

To quantify the accuracy of the latency estimation methods presented in Section III, we compare the estimated results to measured times for 12 state-of-the-art DNNs listed in Tab. 2 from Xilinx Model Zoo [34] and a randomly selected subset of 34 networks from the models generated in NASBench [21] on target devices.

TABLE 2. Networks used to evaluate estimation accuracy.

Network	Dataset	Operations
InceptionV1	Imagenet	3.2G
InceptionV2	Imagenet	4.0G
InceptionV3	Imagenet	11.4G
InceptionV4	Imagenet	24.5G
Resnet18	Imagenet	3.7G
Resnet50	Imagenet	7.7G
Feature Pyramid Network (FPN)	Cityscapes	8.9G
Openpose	AIChallenger	189.7G
MobilenetV1	Imagenet	1.1G
MobilenetV2	Imagenet	1.2G
YoloV2	VOC	34.0G
YoloV3	VOC	65.4G

A. EXPERIMENTAL SETUP

All experiments were performed with batch size 1 to achieve the lowest possible latency, but by adding the batch-size as an additional input parameter for the benchmark dataset and by adding the batch size to the input feature vector of the estimation models, it would also be possible to extend the method to larger batch sizes. For Xilinx DPU, we used a ZCU102 evaluation board with a DPU configuration of 4096 MAC units. Measurements on the NCS2 were performed with an Intel i5-4590 3.3 GHz host processor equipped with 16 GB of RAM in synchronous mode. For both platforms, we used the provided tools for mapping and compilation. To assess the estimator performance, we use two test sets. **Test set 1** contains the 12 DNNs listed in Table 2, and we use it to evaluate in detail the performance for commonly used networks. With **Test set 2**, we aim to understand whether ANNETTE could be used for a hardware-oriented neural architecture search. Therefore, we randomly select 34 models of the NASBench [35] neural architecture search dataset, which contains a large variety of different architectures with similar sizes, and evaluate the accuracy and fidelity of our estimator.

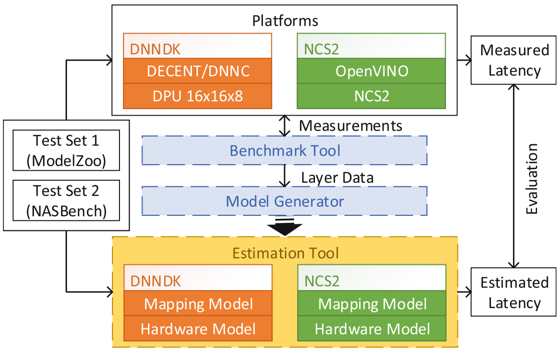


FIGURE 9. The experimental setup for prediction accuracy evaluation.

Figure 9 shows the experimental setup. In the first phase, the benchmarks from Section IV are executed on the target platforms. The execution times of the layers are extracted using the provided profiling tools and stored together with the configuration files of the benchmarks. With the Model Generator (Section V), the mapping and hardware abstraction models are derived and made available to the Estimation Tool (Section VI). In the second phase, the network graphs are fed into the estimator. For evaluation, the resulting estimated times are compared with the execution times measured on the target device. The detailed information provided by the profiling tools allows us to compare not only the total execution times of the networks but also the execution time of each layer.

B. LAYER EXECUTION TIME MODELS

First, we evaluate the accuracy of the previously presented layer execution time models. Tab. 3 reports the Mean-Absolute-Error (MAE), the Mean-Absolute-Percentage-Error (MAPE) and the Root-Mean-Square-Percentage-Error (RMSPE) of the different layer models for all convolution layers of the networks in Table 2. The results were estimated and measured for both the NCS2 and the ZCU102 SoC-board. Additionally, we also report the accuracy of other state-of-the-art execution time prediction methods [16], [17].

TABLE 3. Layer execution time model evaluation for all convolution layers of the networks in Table 2.

Work	Device	Model Type	MAE(ms)	RMSPE	MAPE
[16]	Titan X	Analytical	-	58.29%	-
[17]	Titan X	Statistical	-	39.97%	-
This	NCS2	Roofline	0.783	63.64%	32.58%
		Ref. Roof.	0.730	61.42%	31.69%
		Statistical	0.402	44.13%	15.59%
		Mixed	**0.360**	**42.60%**	**15.57%**
This	ZCU102	Roofline	0.100	18.52%	39.67%
		Ref. Roof.	0.066	15.45%	34.57%
		Statistical	0.064	12.95%	16.69%
		Mixed	**0.036**	**10.55%**	**12.71%**

The mixed model outperforms the other model types for both platforms in terms of MAE, MAPE and RMSPE. It is noticeable that for the ZCU102, the refined roofline model has a lower MAE than the statistical model. Since the MAE is a non-weighted error metric, we conclude that for the ZCU102, the refined roofline model predicts larger layers more accurately than the statistical model.

For fair comparison to other state-of-the-art works, it has to be mentioned that the reported numbers were measured on a different set of networks[1] and for a different set of target devices. While the Paleo [16] and NeuralPower [17] target server GPUs (Titan X), our work targets prediction for specific accelerators for neural networks. However, even in this case, the statistical prediction method outperforms the analytical model. Nevertheless, analytical models are easier to understand and can be easily adapted to similar architectures, whereas a statistical model can only be based on measurements. Additionally, we applied the NeuralPower estimation method with our collected data for the NCS2 and ZCU102, but we were not able to produce any useful results with a MAPE lower than 1000%, so we don't list the results of this approach in Tab. 3. To our mind, these results are a consequence of the bad extrapolation behavior of polynomial functions, which are used for estimation in NeuralPower.

C. MAPPING MODELS

We evaluate the performance of the mapping models on the dataset consisting of the layers from the example networks generated by the Benchmark Tool. For the training data set, we consider only the layer pairs that contain the target layer, e.g., for training the decision tree that predicts whether a pooling layer is fused or not, we include only layer pairs in the data set, at least one of which is a pooling layer. Then we select 80% of the samples for training and 20% for validation. Tab. 4 shows the F_1 score and the Matthews Correlation Coefficient (MCC) for the fusing of element-wise addition and pooling layers.

TABLE 4. Mapping model evaluation for fusing pooling and element-wise addition with a preceding convolution layer.

Device	Layer Type	Total Samples	F1 Score	MCC
ZCU102	Pooling	31733	0.973	0.871
	ElemwiseAdd	6079	0.990	0.923
NCS2	Pooling	14628	0.824	0.831
	ElemwiseAdd	21942	0.792	0.733

Since the F_1 score ignores true negatives, the MCC, which depends on all four confusion matrix categories, should be preferred for the evaluation of the binary classification [36]. It can be seen that the mapping prediction works quite well for both platforms. However, the prediction for the DNNDK (ZCU102) for both layer types achieves a higher F_1 score and MCC than the prediction for the NCS2. We assume that the reasons for this are that the DNNDK is generally more capable of merging several layers and that the optimization behavior of the OpenVINO toolkit depends more on the

[1] Paleo and Neuralpower on VGG-16, AlexNet, NIN, Overfeat, CIFAR10-6conv

architecture of the whole network than only on the parameter settings of the individual layers.

D. EVALUATION FOR TEST SET 1

For evaluation of the generated platform models of the NCS2 and DNNDK, we perform the mapping and layer-wise estimation for the models listed in Table 2. Then, we compare the predicted network execution time with the measured time. Table 5 shows the MAE and MAPE of all presented models for the executed networks for the ZCU102 and NCS2.

Fig. 10 and Fig. 11 show the estimation accuracy of the platform models. Due to moderate parallelization effects on the NCS2, the roofline model and the refined roofline model have similar performance. However, in some cases, the refined roofline model provides slightly better predictions. Also for the NCS2, the statistical and the mixed model achieve almost almost similar performance with a MAPE of 7.92% and 7.44%, respectively. Overall, the mixed model consistently performs the best for the NCS2. Similarly, for the ZCU102, the mixed model provides the most accurate predictions with a MAPE of only 3.47%. Interestingly, in the case of the ZCU102, for some of the networks, the refined roofline model estimates the network execution time more accurately than the statistical model. Since the refined roofline model mainly covers reduced utilization efficiency due to the computational architecture, we can conclude that for those cases, the main inefficiency lies in the low utilization efficiency of computational resources due to a parameter not aligning with the number of available multiplier resources (see Seciton V-A1). The comparison to other state-of-the-art execution time estimators, which are also denoted in Tab. 2, is difficult since the necessary complexity of the model and the resulting accuracy highly depends on the target device. In addition, the evaluation performed in this work includes more complex and larger networks with several different layer types than in other works.

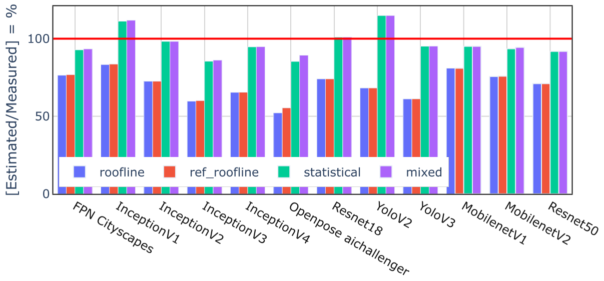


FIGURE 10. Accuracy of the estimated latency for the selected of Table 2 networks on NCS2.

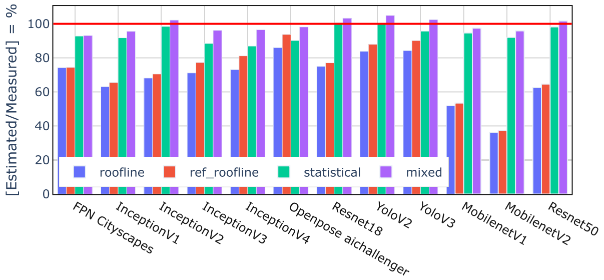


FIGURE 11. Accuracy of the estimated latency for the selected Table 2 on DNNDK.

TABLE 5. Network execution time estimation evaluation for all the networks in Tab. 2. The mixed model outperforms the other models for both platforms in MAE and MAPE.

Work	Device	Model Type	MAE (ms)	MAPE
[16]	Titan X GPU	Analytical	23.13	27.61%
[17]	Titan X GPU	Statistical	5.11	7.96%
This	NCS2	Roofline	67.79	29.95%
		Ref. Roofline	64.93	29.56%
		Statistical	18.52	7.92%
		Mixed	**14.95**	**7.44%**
This	ZCU102	Roofline	6.12	30.89%
		Ref. Roofline	4.08	27.24%
		Statistical	2.49	5.97%
		Mixed	**0.87**	**3.47%**

E. EVALUATION FOR TEST SET 2

To evaluate the accuracy of the estimations for design space exploration, we perform the estimation for a randomly selected subset of 34 network architectures generated for the NASBench dataset. We select this dataset since it contains several networks with similar sizes that were constructed for the same task. Therefore it is more appropriate to evaluate the fidelity of the estimation tool on Test Set 2. We assess the performance on Test Set 2 for the NCS2, which was performing worse on Test Set 1. Table 6 provides the MAE, MAPE and Spearman's rank correlation coefficient ρ as fidelity metric. A perfect Spearman correlation of $+1$ occurs when the variables are a perfect monotonically increasing function of each other. This property makes ρ a valid measure for fidelity [37].

TABLE 6. Fidelity and accuracy metrics for Test Set 2.

	Spearman's ρ	MAE (ms)	MAPE
Roofline / Ref. Roofline	0.971	3.50	33.38%
Statistical / Mixed	0.988	0.53	9.65%

Fig. 12 shows the resulting estimated and measured time in milliseconds for the NCS2. Due to the selected resolutions, there is no difference between the results of the roofline and the refined roofline model. Hence, also the statistical

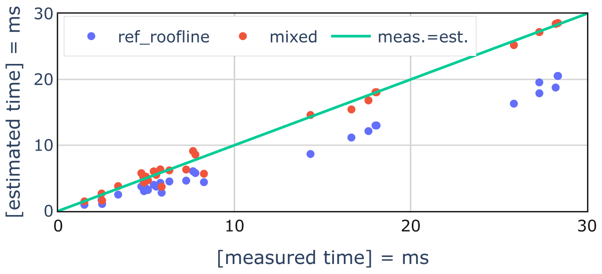


FIGURE 12. NCS2 estimation performance for Test Set 2.

and mixed models achieve the same results. For Test Set 2, the mixed/statistical modeling approach reaches almost a Spearman's rank correlation coefficient of +1 and outperforms the analytic models by more than 20 percentage points in MAPE.

VIII. CONCLUSION

We propose a framework for execution time estimation for neural network hardware accelerators. It is based on stacked models, consisting of mapping models and mixed layer models. We generate the models based on micro-kernel and multi-layer benchmark results and evaluate the performance on two sets of networks for two selected hardware accelerators. Overall, the mixed models perform best. For a set of 12 state-of-the-art DNNs, the estimation with mapping models and mixed models reach a MAPE of only 3.47% on the Xilinx ZCU102 SoC and 7.44% on the Intel NCS2 when estimating total network execution times. For the use case of design space exploration, we evaluate the fidelity of the generated models by applying the estimation method on a randomly selected subset of 34 models of the NASBench dataset. The estimation with mapping models and mixed layer models reaches fidelity of 0.988 in Spearman's ρ rank correlation coefficient metric. The evaluation demonstrates the advantages of applying mixed models for the selected hardware platforms. In the future, we aim to extend the evaluation to additional embedded hardware, such as the Nvidia Jetson platform, to gain additional insights for a different class of accelerators.

Due to the large parameter space of DNNs, one crucial point for the development of the estimation framework is to make assumptions about the computing architecture to exclude as many non-meaningful measurement points as possible. An essential clue is the step-wise linear nature of architecture resources, such as an array of multipliers or caches. They follow a linear performance trend until the cache or the multiplier array is fully allocated. Besides, for a precise estimation, it is important to consider not only the individual layers in isolation but also how they are executed in the overall context.

We are confident that accurate estimation methods can significantly facilitate informed making of decisions. Nevertheless, it is in the area of neural architecture search where estimation can make a critical contribution to a hardware-specific search or the right choice of networks and hardware in advance of the development of applications.

REFERENCES

[1] C. Fruhwirth-Reisinger, G. Krispel, H. Possegger, and H. Bischof, "Towards data-driven multi-target tracking for autonomous driving," in *Proc. 25th Comput. Vis. Winter Workshop (CVWW)*. Slovenia, Balkans, 2020, pp. 27–36.

[2] P. Rajpurkar, A. Y. Hannun, M. Haghpanahi, C. Bourn, and A. Y. Ng, "Cardiologist-level arrhythmia detection with convolutional neural networks," *CoRR*, vol. abs/1707.01836, pp. 1–9, Jul. 2017.

[3] M. Wess, P. D. Sai Manoj, and A. Jantsch, "Neural network based ECG anomaly detection on FPGA and trade-off analysis," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2017, pp. 1–4.

[4] J. Zhang and C. Zong, "Deep neural networks in machine translation: An overview," *IEEE Intell. Syst.*, vol. 30, no. 5, pp. 16–25, Sep. 2015.

[5] F. Tung and G. Mori, "CLIP-Q: Deep network compression learning by in-parallel pruning-quantization," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 7873–7882.

[6] S. Srinivas and R. V. Babu, "Data-free parameter pruning for deep neural networks," *CoRR*, vol. abs/1507.06149, pp. 1–12, Jul. 2015.

[7] M. Wess, S. M. P. Dinakarrao, and A. Jantsch, "Weighted quantization-regularization in DNNs for weight memory minimization toward HW implementation," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 11, pp. 2929–2939, Nov. 2018.

[8] S. Shin, K. Hwang, and W. Sung, "Fixed-point performance analysis of recurrent neural networks," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, Mar. 2016, pp. 976–980.

[9] D. Miyashita, E. H. Lee, and B. Murmann, "Convolutional neural networks using logarithmic data representation," 2016, *arXiv:1603.01025*. [Online]. Available: http://arxiv.org/abs/1603.01025

[10] M. Jaderberg, A. Vedaldi, and A. Zisserman, "Speeding up convolutional neural networks with low rank expansions," in *Proc. Brit. Mach. Vis. Conf.*, 2014, pp. 1–12.

[11] C. Tai, T. Xiao, and X. Wang, "Convolutional Neural Networks with Low-Rank Regularization," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2016, pp. 1–11.

[12] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted residuals and linear bottlenecks," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 4510–4520.

[13] X. Zhang, X. Zhou, M. Lin, and J. Sun, "ShuffleNet: An extremely efficient convolutional neural network for mobile devices," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 6848–6856.

[14] T.-J. Yang, A. Howard, B. Chen, X. Zhang, A. Go, M. Sandler, V. Sze, and H. Adam, "NetAdapt: Platform-aware neural network adaptation for mobile applications," in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, 2018, pp. 285–300.

[15] H. Cai, L. Zhu, and S. Han, "ProxylessNAS: Direct neural architecture search on target task and hardware," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2019, pp. 1–13.

[16] H. Qi, E. R. Sparks, and A. Talwalkar, "Paleo: A performance model for deep neural networks," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2017, pp. 1–10.

[17] E. Cai, D. Juan, D. Stamoulis, and D. Marculescu, "NeuralPower: Predict and deploy energy-efficient convolutional neural networks," *CoRR*, vol. abs/1710.05420, pp. 1–16, Oct. 2017.

[18] S. Yao, Y. Zhao, H. Shao, S. Liu, D. Liu, L. Su, and T. Abdelzaher, "FastDeepIoT," in *SenSys*. New York, NY, USA: ACM Press, 2018.

[19] D. Velasco-Montero, J. Fernandez-Berni, R. Carmona-Galan, and A. Rodriguez-Vazquez, "PreVIous: A methodology for prediction of visual inference performance on IoT devices," *IEEE Internet Things J.*, vol. 7, no. 10, pp. 9227–9240, Oct. 2020.

[20] M. Almeida, S. Laskaridis, I. Leontiadis, S. I. Venieris, and N. D. Lane, "EmBench," in *Proc. 3rd Int. Workshop Deep Learn. Mobile Syst. Appl.*, 2019, pp. 7–13.

[21] C. Ying, A. Klein, E. Real, E. Christiansen, K. Murphy, and F. Hutter, "NAS-bench-101: Towards reproducible neural architecture search," in *Proc. Int. Conf. Mach. Learn. (ICML)*, vol. 97, Jun. 2019, pp. 7105–7114.

[22] V. J. Reddi, C. Cheng, D. Kanter, and P. Mattson, "Mlperf inference benchmark," *CoRR*, vol. abs/1911.02549, pp. 1–5, Dec. 2019.

[23] C. Coleman, D. Narayanan, D. Kang, T. Zhao, J. Zhang, L. Nardi, P. Bailis, K. Olukotun, C. Ré, and M. Zaharia, "Dawnbench: An end-to-end deep learning benchmark and competition," *Training*, vol. 100, no. 101, p. 102, 2017.

[24] B. Wu, K. Keutzer, X. Dai, P. Zhang, Y. Wang, F. Sun, Y. Wu, Y. Tian, P. Vajda, and Y. Jia, "FBNet: hardware-aware efficient ConvNet design via differentiable neural architecture search," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2019, p. 10.

[25] X. Dai, A. Wan, P. Zhang, B. Wu, Z. He, Z. Wei, K. Chen, Y. Tian, M. Yu, P. Vajda, and J. E. Gonzalez, "FBNetV3: Joint architecture-recipe search using neural acquisition function," 2020, *arXiv:2006.02049*. [Online]. Available: http://arxiv.org/abs/2006.02049

[26] A. Shaw, D. Hunter, F. Landola, and S. Sidhu, "SqueezeNAS: Fast neural architecture search for faster semantic segmentation," in *Proc. IEEE/CVF Int. Conf. Comput. Vis. Workshop (ICCVW)*, Seoul, South Korea, 2019, pp. 2014–2024.

[27] T. Tang and Y. Xie, "Mlpat: A power area timing modeling framework for machine learning accelerators," in *Proc. DOSSA Workshop*, 2018, pp. 1–3.

[28] Y. Zhao, C. Li, Y. Wang, P. Xu, Y. Zhang, and Y. Lin, "DNN-chip predictor: An analytical performance predictor for DNN accelerators with various dataflows and hardware architectures," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, Barcelona, Spain, May 2020, pp. 1593–1597, doi: 10.1109/ICASSP40776.2020.9053977.

[29] Xilinx. (2020). *Xilinx Deep Neural Network Development Kit*. Accessed: Apr. 17, 2020. [Online]. Available: https://www.xilinx.com/products/design-tools/ai-inference/edge-ai-platf%orm.html#dnndk

[30] Intel. (2018). *OpenVINO Toolkit*. Accessed: Dec. 12, 2018. [Online]. Available: https://software.intel.com/en-us/openvino-toolkit

[31] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer CNN accelerators," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2016, pp. 1–12.

[32] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[33] Y. Chen, T. Yang, J. S. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE J. Emerg. Sel. Topics Circuits Syst.*, vol. 9, no. 2, pp. 292–308, Jun. 2019, doi: 10.1109/JETCAS.2019.2910232.

[34] Xilinx. (2020). *Xilinx AI-Model-Zoo*. Accessed: Apr. 17, 2020-04-17. [Online]. Available: https://github.com/Xilinx/AI-Model-Zoo

[35] C. Ying, A. Klein, E. Real, E. Christiansen, K. Murphy, and F. Hutter, "Nas-bench-101: Towards reproducible neural architecture search," *CoRR*, vol. abs/1902.09635, pp. 1–15, May 2019.

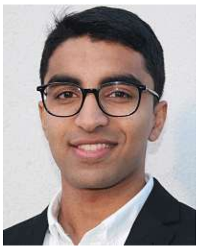
[36] D. Chicco and G. Jurman, "The advantages of the matthews correlation coefficient (MCC) over f1 score and accuracy in binary classification evaluation," *BMC Genomics*, vol. 21, no. 1, Jan. 2020.

[37] H. Javaid, A. Ignjatovic, and S. Parameswaran, "Fidelity metrics for estimation models," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2010, pp. 1–8.

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. He is part of the Christian Doppler Laboratory for Embedded Machine Learning at TU Wien, Austria. His current research interests include hardware acceleration of deep neural networks and energy-efficient machine learning

MATVEY IVANOV is currently pursuing the bachelor's degree with the Faculty of Electrical Engineering and Information Technology, TU Wien, Austria. Since 2019, he has been part of the Christian Doppler Laboratory for Embedded Machine Learning at TU Wien.

CHRISTOPH UNGER received the B.Sc. degree in computer engineering and the M.Sc. degree in automation and control from TU Wien, Vienna, Austria, in 2015 and 2020, respectively, where he is currently pursuing the Ph.D. degree with the Automation and Control Institute (ACIN). He is currently a Researcher with ACIN, TU Wien. His work is focused on the topics of machine intelligent control as well as skill transfer learning in the area of robotics. His research interests include robotics, generative deep learning, and intelligent and optimal-based control.



ANVESH NOOKALA received the Bachelor of Science degree in electrical engineering from TU Wien, in 2019, where he is currently pursuing the master's degree in embedded systems, with a focus on a range of topics such as mechatronics, machine vision, computer systems, and electronics design. Parallel to his studies, he is part of the Siemens Electronics Research Group, Vienna, where his work is focused on hardware for artificial intelligence and related topics.

ALEXANDER WENDT (Member, IEEE) received the degree in technical physics, in 2007, and the Ph.D. degree in decision making in artificial intelligence, in 2016. He is currently a Research Coordinator with the Christian Doppler Laboratory for Embedded Machine Learning, TU Wien, Austria. After successfully completing his degree, he worked as a Safety Engineer with Frequentis AG. Until 2020, he focused on software architectures for smart grids and cognitive architectures as control systems in buildings. Since 2020, his research focus is on the characterization and optimization of neural networks for embedded hardware. He has published more than 30 articles, acted as the session chair in sessions about machine learning and cognitive architectures.

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.

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. 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.

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. We tackle this challenge by developing an intelligent benchmarking 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 comparing hardware platforms using prediction algorithms.

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 $L = \{l_1, l_2, ..., l_\mathrm{N}\}$. 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 [9]. 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 [10]–[12] 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 [13] and [14] 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 [15], [16] 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 [17] 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 [18] deploys a GNN to detect memory over-commitment in addition to an ANNETTE-like structure. While frameworks like DNNPerf [9] and GENNAPE [19] focus on the prediction of other DNN performance parameters (such as accuracy, training time, etc.), PerfSAGE [7] and DIPPM [20] 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 [21], [22] 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 [23], [24] or Bayesian methods [25]. In the context of this work, which leverages random forest regression [26], conformal prediction is particularly beneficial for uncertainty quantification, as it not only demonstrates good efficiency [27] 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 $\alpha_j = |\hat{y}_j - y_j|$ [27], where $\hat{y}$ is the predicted value. For the prediction of confidence intervals with significance δ, these calculated nonconformity scores are used to formulate the prediction region for each instance j as $\hat{Y}_j^\delta = \hat{y}_j \pm \alpha(\delta)$ [27]. 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 [28]. 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 [29], face recognition [30], and financial risk prediction [31]. 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.

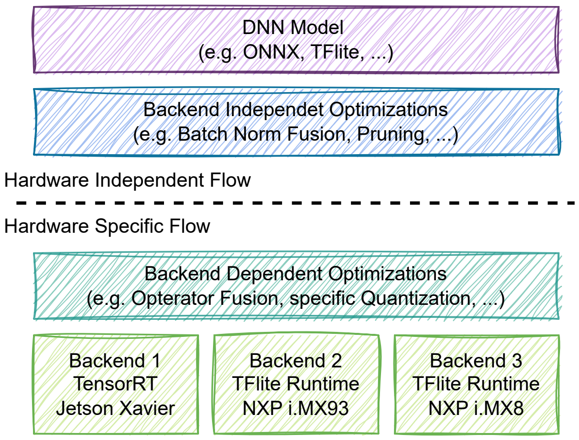

DNN Model
(e.g. ONNX, TFlite, ...)
Backend Independet Optimizations
(e.g. Batch Norm Fusion, Pruning, ...)
Hardware Independent Flow
Hardware Specific Flow
Backend Dependent Optimizations
(e.g. Opterator Fusion, specific Quantization, ...)
Backend 1
TensorRT
Jetson Xavier
Backend 2
TFlite Runtime
NXP i.MX93
Backend 3
TFlite Runtime
NXP i.MX8

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

- **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.

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 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 addi-

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

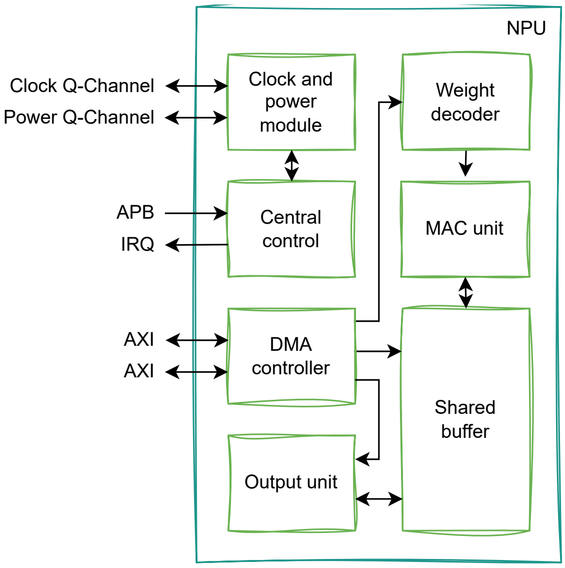


FIGURE 2. Blockdiagram of the ARM Ethos NPU [32].

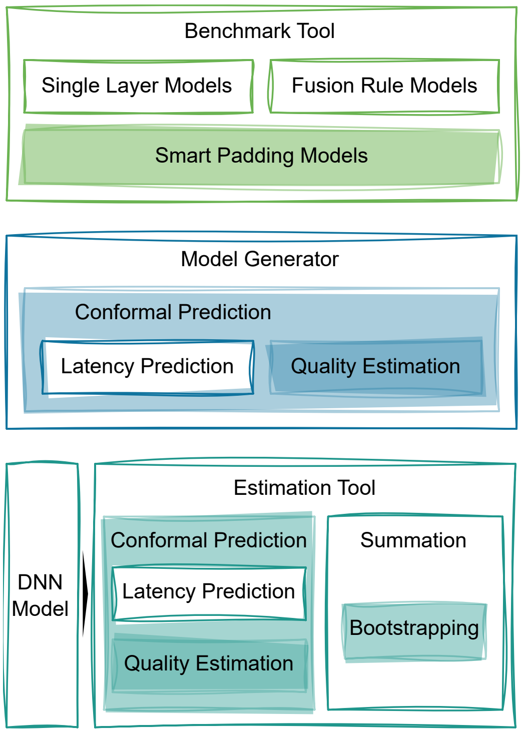


FIGURE 3. Overview of the Components in ANNETTE. The color-shaded components are added in this work.

tional 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 [11], 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.

Taking a closer look at the block diagram of an ARM Ethos Neural Processing Unit (NPU) (see Figure 2), 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.

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 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 [33].

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 **Esti-**

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

mation 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 $t_{\text{data\_out}}$.

$$\mathcal{T} = t_{\text{data\_in}} + t_{\text{data\_out}} + \sum_{i=1}^{\text{layers}} t_{\text{comp},i} \qquad (1)$$

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.

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).

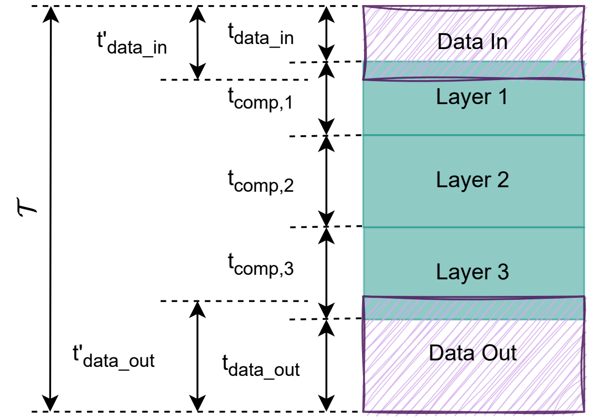


FIGURE 4. Relationship between measured, compute and data transfer times for a DNN with three layers.

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

$$\mathcal{T}_{\text{b},n} = t_{\text{padding\_in},n} + t_{\text{padding\_out},n} \qquad (2)$$

$$\mathcal{T}_{\text{c}} = t_{\text{padding\_in},1} + t_{\text{LUM}} + t_{\text{padding\_out},2} \qquad (3)$$

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.

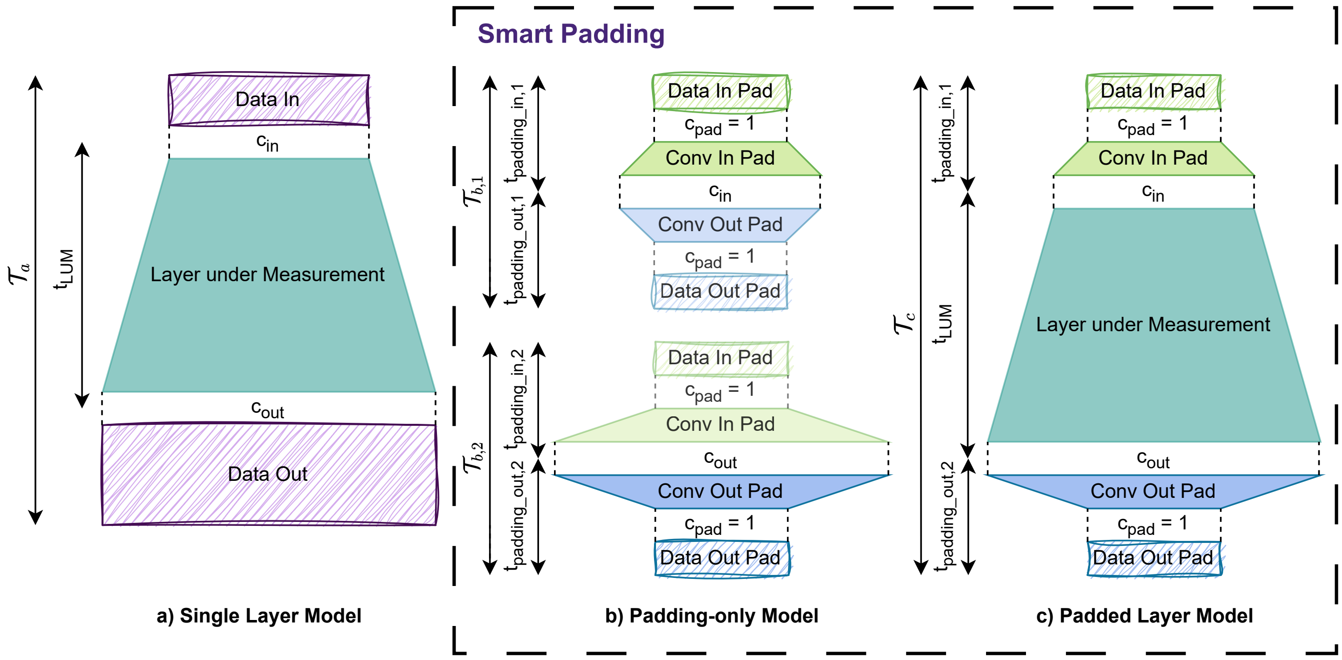


FIGURE 5. The three models used for benchmarking the different platforms. The Single Layer Model (a) is the simplest way but does not provide accurate measurements of t_{LUM}. The black-box benchmarking method makes use of padding-only (b) and padded layer models (c) to solve this problem.

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 $T_{\text{LUM}}$ using the Equation 4
end for
```

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:

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

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 output channels, respectively. The computation time scales with $w \cdot h \cdot c_{\text{in}} \cdot c_{\text{out}} \cdot k_{\text{height}} \cdot k_{\text{width}}$, 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.

$$
\begin{aligned}
t_{\text{comp\_padded}} &\propto w \cdot h \cdot c_{\text{in}} \cdot c_{\text{out}} \cdot k_{\text{h}} \cdot k_{\text{w}} \\
&+ w_{\text{in}} \cdot h_{\text{in}} \cdot c_{\text{in}} + w_{\text{out}} \cdot h_{\text{out}} \cdot c_{\text{out}}
\end{aligned} \tag{5}
$$

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 $\mathcal{T}_{\text{b},1} = \mathcal{T}_{\text{b},2}$

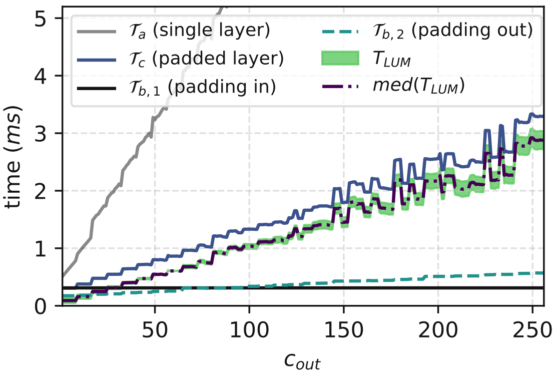


FIGURE 6. Computed and measured times for the three models of a 2D convolution layer with $c_{in} = 64$, $c_{out} \in [1, 256]$, $h = 64$, $w = 64$ for the i.MX93

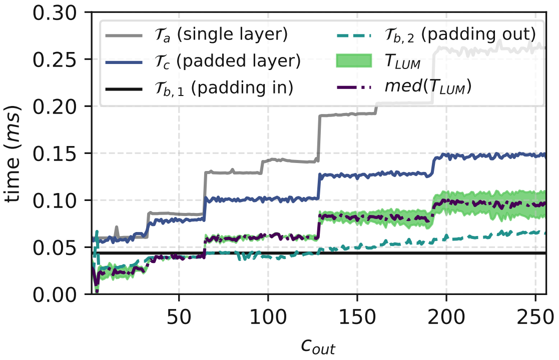


FIGURE 7. Computed and measured times for the three models of a 2D convolution layer with $c_{in} = 64$, $c_{out} \in [1, 256]$, $h = 64$, $w = 64$ for the Jetson Xavier

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.

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 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 [21], [22]. 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})

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

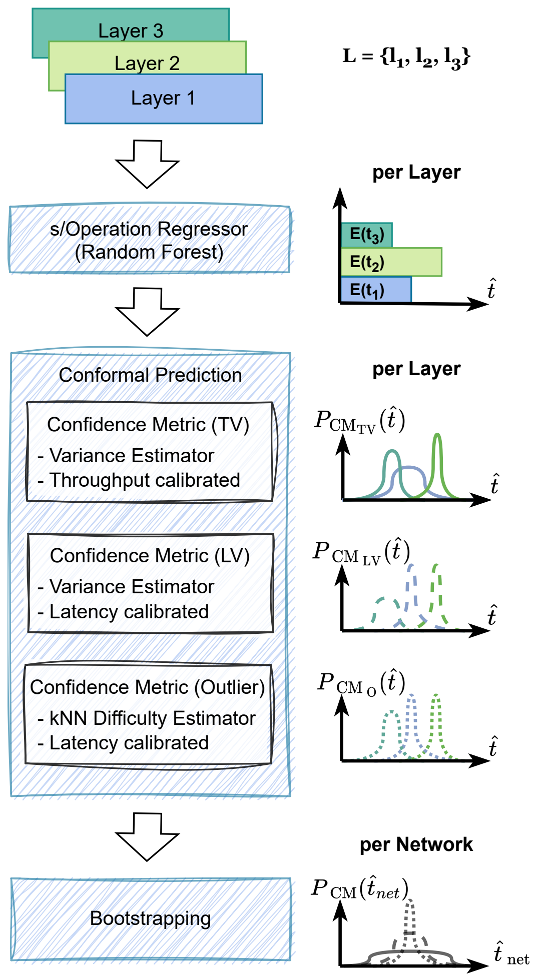


FIGURE 8. **Overview of the confidence prediction methodology.**

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 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.:

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

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 $t_\mathrm{comp} = t_\mathrm{op} \cdot \mathrm{num}_\mathrm{ops}$. 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:

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

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

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

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 [34] a Python package for generating conformal regressors and predictive systems. 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 CM_TV and CM_LV, variance-based difficulty estimation is applied. For CM_O, k-NN-based difficulty estimation is used. Additionally, while the difficulty estimation in CM_LV is calibrated based on the absolute prediction error of the layer latency, for CM_TV it is calibrated based on the absolute prediction error of the layer efficiency (s/operation). The difficulty estimation for 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.

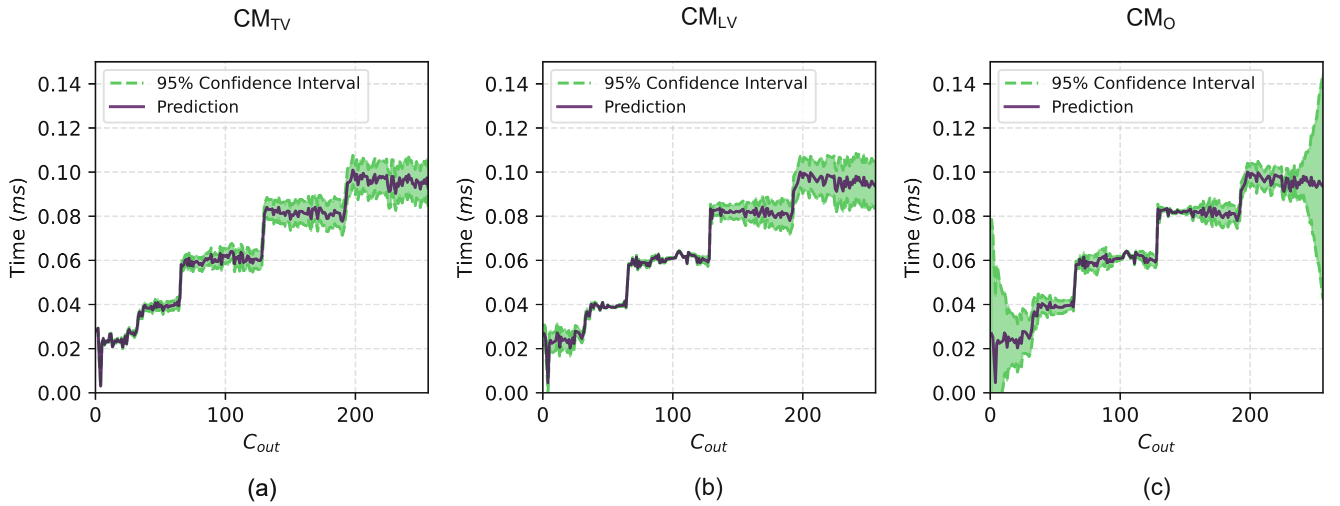


FIGURE 9. Overview of predicted Confidence Intervals for 2D convolution layer with $c_{in} = 64$, $c_{out} \in [1, 256]$, $h = 64$, $w = 64$ on the Jetson Xavier: (a) Confidence s/Operations Normalized, (b) Confidence Normalized with respect to time, (c) Confidence with k-NN difficulty estimation

The confidence interval for 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 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 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 CM_{LV} is most useful for comparing the confidence intervals of the overall layer execution times.

For 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, 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 2 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 smart padding-based ANNETTE prediction demonstrates superior performance compared to the single-layer ANNETTE prediction and the Vela estimates, achieving higher predic-

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

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 2. 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 4. Percentage prediction errors for the ANNETTE models for the i.MX8M+

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

TABLE 3. Percentage prediction errors for the ANNETTE models for the Jetson Xavier

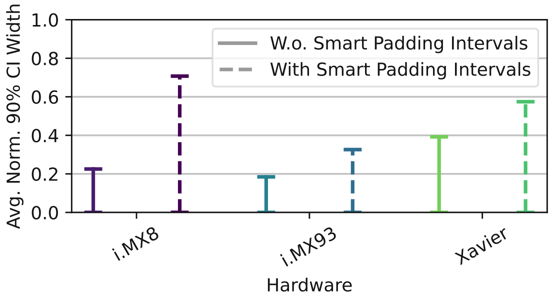


FIGURE 10. Average normalized 90% confidence intervals for CM_{TV} on the all tested networks

tion 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 3 and 4, 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.

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. 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 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.

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

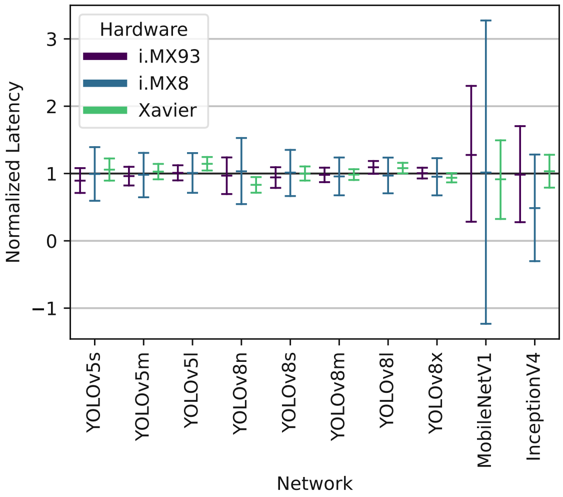


FIGURE 11. ***CM_{LV} for the tested networks.***

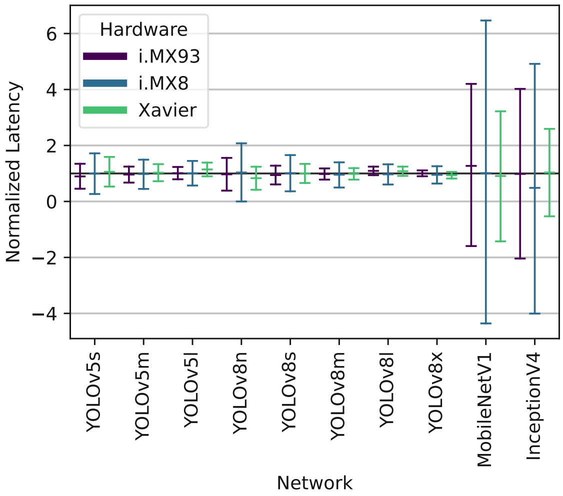


FIGURE 12. ***CM_O for the tested networks.***

confidence interval for $\mathrm{CM}_{\mathrm{LV}}$ indicates sub-optimal prediction accuracy due to high variances in the dataset within the prediction region. 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

$\mathrm{CM}_{\mathrm{LV}}$ 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. Additionally, the $\mathrm{CM}_{\mathrm{LV}}$ interval widths hint at some layers in prediction regions with high variance of measured latencies.

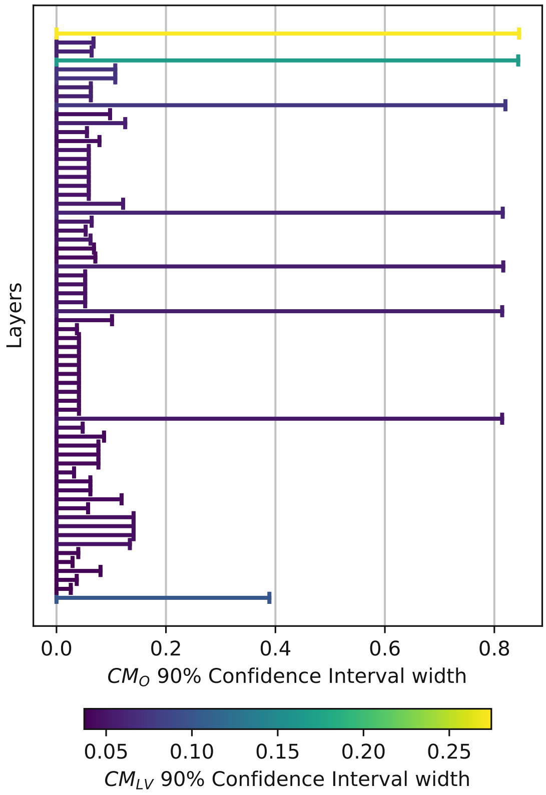


FIGURE 13. **YOLOv8n per-layer interval widths of CM_{LV} and CM_O**

V. CONCLUSION

This study introduces a novel approach for benchmarking DNN accelerators that eliminates the need for per-layer profiling. 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.

REFERENCES

[1] Han Cai, Ligeng Zhu, and Song Han. ProxylessNAS: Direct neural architecture search on target task and hardware. In *ICRL*, 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.

[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] 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.

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

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

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

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

[19] 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.

[20] 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.

[21] 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.

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

[23] Nicolai Meinshausen. Quantile regression forests. *J. Mach. Learn. Res.*, 7:983–999, 2006.

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

[25] 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.

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

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

[28] 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.

[29] 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.

[30] 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.

[31] 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.

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

[33] 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.

[34] 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, *Conformal and Probabilistic Prediction with Applications, 24-26 August 2022, Brighton, UK*, volume 179 of *Proceedings of Machine Learning Research*, pages 24–41. PMLR, 2022.

MATTIAS 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.

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.

2023 26th Euromicro Conference on Digital System Design (DSD)

Energy Profiling of DNN Accelerators

Matthias Wess, Dominik Dallinger, Daniel Schnöll, Matthias Bittner,
Maximilian Götzinger and Axel Jantsch
Christian Doppler Laboratory for Embedded Machine Learning, Institute of Computer Technology
TU Wien, 1040 Vienna, Austria
{firstname}.{lastname}@tuwien.ac.at

***Abstract*—This paper introduces a novel methodology for assessing the energy efficiency of neural network accelerators at both layer and network granularity. The approach involves extracting per-layer timing reports from recorded power profiles. The power and energy consumption of three prominent neural network accelerators, namely the *Intel Neural Compute Stick 2*, the *Coral Edge TPU*, and the *NXP i.MX8M Plus* is evaluated for three different Deep Neural Networks (DNNs) using this method.**

The study investigates the relationship between decreasing sampling frequencies and the average error, as well as the detailed energy consumption of individual DNN layers and layer types. The findings reveal that latency outperforms the number of operations per layer as a predictor for both overall and dynamic energy, with errors of 10 % and 100 % respectively.

The main conclusions are: a sampling frequency of 200 kHz is necessary to achieve an average error of 5 %; the number of operations is an inadequate predictor of energy consumption; and specific hardware settings significantly influence power and energy consumption, emphasizing the need for their consideration in estimation.

***Index Terms*—Power analysis, Deep Neural Networks, Hardware accelerators**

I. Introduction

Along with the increasing usage of Machine Learning (ML) for solving complex tasks such as computer vision, there has been plenty of research on hardware architectures to execute such algorithms at reduced power and energy budgets. These hardware accelerators reduce the overall power consumption and allow integration of the data processing into the edge devices. Specifically, the inference of ML algorithms on embedded devices has proven efficient in terms of performance and energy. Among several one-board solutions, hardware vendors also provide external USB and PCI-based inference accelerators to support the processing system for ML workloads [1]–[3].

Assessing the power and energy requirements of the various embedded devices is crucial for understanding their performance and optimizing their use in various applications. Obtaining precise measurements that reveal the energy consumption of individual DNN layers is challenging, primarily due to their short execution time. This task becomes even more difficult in the presence of other power consumers in connected electronics, which can obscure the relevant figures.

Our interest is the power consumption of specific DNNs as well as their specific layers to analyze the energy efficiency of different layer types. Moreover, we attempt to understand the accuracy of proxy metrics for estimating power and energy consumption.

This paper makes the following key contributions:

- We perform experiments on the required measurement frequency to achieve accurate single-layer measurements.
- We propose a method to extract layer times from the measured power profiles by iteratively removing the last layer of the DNN.
- We profile DNNs on *Neural Compute Stick 2* (NCS2) and *Coral Edge TPU* (edge TPU) at single-layer granularity, gaining insights about the execution efficiency of different layer types on different devices.

Our primary focus is to investigate the energy efficiency of three distinct DNN accelerator architectures. We achieve this by monitoring power consumption at a high sampling frequency. Furthermore, we develop a methodology that enables us to automatically capture the power behavior during the inference phase of DNNs on these accelerators. This methodology also allows us to extract valuable information concerning the power consumption of individual layers within the Neural Network (NN). We accomplish this by correlating detailed latency reports from the accelerators with the recorded power profiles. In cases where detailed latency reports are unavailable, we employ an iterative approach where we remove the last executed layer of the network. To extract layer-specific power consumption data, we then compare the resulting power profiles. Lastly, we conduct an extensive analysis on the inference options available on the *NCS2*, *edge TPU*, and *NXP i.MX8M Plus Development Kit* (i.MX8M+).

II. Related Work

In response to the growing need to execute machine learning algorithms on embedded hardware platforms, numerous efforts have been made to compare the performance of different hardware platforms. Cantero et al. [4] compare the performance of the *edge TPU Coral Dev Board* and the *Variscite i.MX8M PLUS Board* across five distinct model architectures in various resolution settings. The results revealed that the *Coral Dev Board*, with 4 Tera Operations per Second (TOPS), achieved faster computation than the *i.MX8M Board*, which reaches up to 2.3. However, the *i.MX8M* demonstrates more efficient

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.

2771-2508/23/$31.00 ©2023 IEEE
DOI 10.1109/DSD60849.2023.00018

resource utilization and closely approaches the performance level of the *Coral Dev Board*.

Several benchmarks have been developed to enable a fair and extensive comparison of compute capabilities [5], [6]. These benchmarks rely on a diverse set of workloads. Notably, the MLCommons consortium[1] offers the MLPerf benchmarks [7] for inference and training across diverse domains such as object detection, medical imaging, speech-to-text, and natural language processing. These benchmarks are categorized into data center, edge, mobile, and tiny platforms, representing the broad spectrum of hardware used in machine learning.

Highlighting the importance of comparing power consumption for specific hardware platforms on certain benchmark applications is the fact that the MLCommons consortium released the MLPerf Tiny benchmark [6], which also focuses on power consumption. MLPerf Tiny [6] includes several benchmark tasks: Keyword Spotting, Visual Wake Words, Image Classification, and Anomaly Detection. By providing standardized evaluation criteria, researchers can compare and analyze the energy efficiency of different DNN accelerators and architectures. However, one limitation is that the benchmark results are presented in a summarized form, which provides a broad overview but may lack detailed insights. To conduct these evaluations, the EnergyRunner[2] framework is employed.

Furthermore, leveraging the MLPerf benchmark set, Libutti et al. [8] have measured the power consumption and performance of the edge TPU (adopted from [9]) and Intel NCS2 [1]. They explored various inference modes and settings, yet their findings were limited to reporting overall energy consumption per network without providing detailed insights at a more granular level.

Blott et al. [10] measure the latency and power consumption of many devices, including FPGA, GPU, edge TPU, and VLIW processors for inference of diverse DNNs they focus on gaining a better understanding of the design space with regards to pruning and quantization.

Finally, there have been efforts to estimate the energy consumption of hardware platforms execution NNs. Reif et al. present *Precious* [11], an approach for estimating the energy consumption of ML models based on linear and random forest regressors. In particular, their implementation estimates execution times as well as the power draw of Convolutional Neural Networks (CNNs) on embedded accelerator hardware for NNs (i.e., Google Coral edge TPU [2]). However, it is restricted to a few layertypes and limited layer settings. Other more accurate approaches [12], [13] require in-depth knowledge of the accelerator design and target the hardware design space exploration domain.

In contrast to other works which report the lump latency, power, and energy for inference, we aim to gain a deeper understanding of the energy efficiency of single layers to provide the basis for energy-efficient neural architecture search and layer energy estimation. We, therefore, analyze the power consumption at much finer granularity to obtain latency, power, and energy measurements for individual layers. To do so, we develop a methodology to extract layer times based on the gathered power profiles. This granularity resolution allows us to relate latency and energy of specific layer types and draw conclusions about their efficiency of execution on the given architecture.

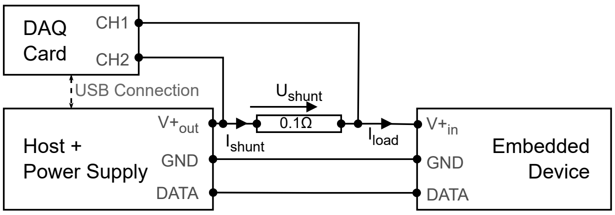


Fig. 1: Power Measurement Circuit. The shunt resistance $R_{\text{shunt}} = 0.1\,\Omega$. U_{shunt} and I_{shunt} are voltage drop and current at the shunt, respectively.

III. Experimental Setup

The measurements are performed on a host computer with an Intel-i7-8565U at 1.80 GHz with 16 GB of RAM and an NCS2 as well as an edge TPU, both connected as ML co-processors connected via USB3.0. For the i.MX8M+, which is also powered via USB 3.0, we measure the power of the entire system. For the measurement setup, we select a USB-1608GX Data Acquisition (DAQ) card due to the good programmability, high sampling rate and flexible measurement ranges. To gather the voltage drop across a shunt resistor was placed on the power line of the Device Under Test (DUT) (see Fig. 1) with a maximum sampling frequency of 500 kHz.

We then compute the power consumption of the DUT as

$$P_{\text{total}} = (U_{\text{supply}} - U_{\text{shunt}}) \cdot (U_{\text{shunt}}/R_{\text{shunt}}), \qquad (1)$$

where R_{shunt} is the shunt resistance, and U_{shunt} is the voltage drop at the shunt. Due to the high input resistance, we can neglect the current through the DAQ card and assume $I_{\text{shunt}} = I_{\text{load}}$. Important to note is that the power supply lines of the USB cable from the host to the DUT have been cut, and the DUT is powered exclusively by the dedicated power supply.

A. Intel NCS2

The NCS2 USB 3.0 ML accelerator implements a MYRIAD X Visual Processing Unit (VPU) with 16 Streaming Hybrid Architecture Vector Engine (SHAVE) processors operating at 700 MHz. For the computation of a DNN, the NCS2 provides a maximum nominal performance of 1 TOPS in Floating-Point 16 (FP16) format. Inference on the accelerator is performed via the OpenVino toolkit, supporting up to 4 parallel inference requests (see section V-E).

[1] https://mlcommons.org, accessed: 2023-05-25

[2] https://github.com/eembc/energyrunner, accessed: 2023-05-25

B. Coral Edge TPU

The Google Coral machine-learning accelerator co-processor is a USB device embedding the Google Edge edge TPU ASIC. The Coral accelerator performs inference operations on TensorFlow Lite[3] models with a peak current of 900 mA at 5 V. The accelerator maintains a computational throughput of 4 TOPS at a computational efficiency of 2 TOPS per Watt [2]. The edge TPU supports two operational modes: the standard mode and the maximum frequency, which are discussed in section V-E.

C. I.MX8M Plus

The i.MX8M+ processor is a heterogeneous multi-core processor developed by NXP. The processor incorporates an embedded Vivante VIP8000 Neural Processing Unit (NPU) that provides 2.3 TOPS of computing power. To optimize the data exchange between the computing units, the NPU shares the high-speed internal memory bus with the Central Processing Units (CPUs). Similar to the edge TPU, the Vivante VIP8000 performs 8-bit Integer (INT8) operations, accelerating the TensorFlow Lite execution.

D. Software

For the measurements of the NCS2, we make use of the OpenVino toolkit [14], which offers many convenient tools for converting, optimizing, benchmarking, and analyzing neural networks. The model optimization and conversion tools support networks generated in popular deep learning frameworks/formats such as PyTorch[4], TensorFlow[5], and ONNX[6] as source workloads. Performance metrics such as latency and throughput of individual layers can be extracted with the help of the API benchmark application.
For the measurements of the edge TPU and the i.MX8M+, the models are converted into TensorFlow Lite format and post-training quantized to INT8 format. The inference on the edge TPU and the i.MX8M+ is driven through the TensorFlow Lite inference engine, offloading the workload to the accelerators via the delegate functionality. Both, the edge TPU and the i.MX8M+ only support fully-quantized 8-bit TensorFlow Lite models. For the edge TPU the models need to be specifically compiled (using the edge TPU Compiler) [15]. For the i.MX8M+ no compilation is necessary prior to the execution. To access the DAQ card, we use the MCC Universal Library[7].

E. The workload

As workload for the measurements, we selected three different DNNs for image classification and object detection, namely MobileNetV2 [16], YOLOv3, and YOLOv3-tiny [17].

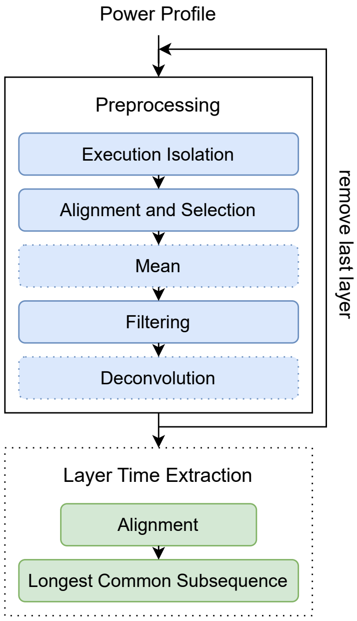


Fig. 2: The signal processing pipeline used for the power profile extraction and layer time extraction

IV. Methodology

This section describes the proposed methodology and signal processing pipeline designed to facilitate precise benchmarking and profiling. We provide a detailed description of the pipeline, which enables the semi-automatic annotation of power and energy consumption for each executed DNN on the hardware platform.

When per-layer timing reports are available on the hardware platforms, the times can be annotated into the power profile. In that case, the major challenge is to identify the start of the first layer. Alternatively, we can retrieve the per-layer timing information by iteratively removing the network layers one by one and comparing the recorded power profiles of the modified and the original network. This approach ensures synchronization between the layer latency annotations and the power profile. Also, some devices operate less efficiently when in profiling mode, which we can avoid. Our method, can generate layer-wise execution time reports for potentially any hardware with a characteristic per-layer power profile.

Fig. 2 depicts the implemented signal processing pipeline. The pipeline consists of two sub-pipelines: pre-processing and layer-profile extraction. For the benchmarking, we also record the power profile of the initialization, the warm-up phase, and multiple iterations of the DNN executed on the target device.

The pre-processing is critical to isolate a single execution power profile from the recorded data. To reach this target, we first detect the single executions by applying a sliding window with the width according to the execution time of the DNN. To ensure the correct cutout of the actual execution

[3] https://www.tensorflow.org/lite, accessed: 2023-05-25
[4] https://pytorch.org, accessed: 2023-05-25
[5] https://www.tensorflow.org, accessed: 2023-05-25
[6] https://onnx.ai/, accessed: 2023-05-25
[7] https://github.com/mccdaq/uldaq, accessed: 2023-05-25

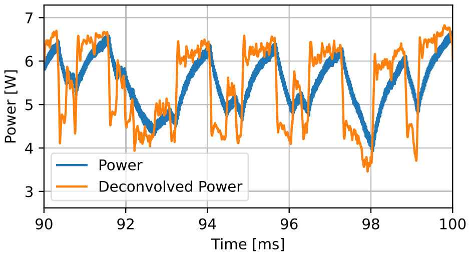


Fig. 3: The recorded power profile for the i.MX8M+ exhibits smoothing effects in comparison to the deconvolved power profile

profiles, we compute the maxima and minima of the filtered signal and select the detected peaks with similar inter-peak distances as center points for the cutouts. Additional adaptive thresholds and edge detection ensure clean cutouts of the execution profiles. Next, the cutouts are pairwise iteratively aligned by minimizing the Euclidean distance between the cutout signals. We use the isolation forest algorithm [18] to filter out anomalies in the cutouts. In an optional next step, we compute the means of the remaining cutouts to reduce noise on the measured signal. To obtain accurate power profiles of the DNN executed on a target device, we need to address the effect of the capacitors within the power supply circuit. Capacitors act as short-term energy storage devices, smoothing the signal but distorting the actual power consumption in the time domain. to the device during the execution of the DNN. These distortions can introduce measurement inaccuracies, potentially leading to the misattribution of energy consumed by a previous layer to the next layer. To solve this problem, we perform a deconvolution with the impulse response of the normalized capacitor discharge curve. The value of τ of the discharge curve is empirically determined as part of the pre-processing step, and the deconvolution is applied to each recorded power profile. The effect of this deconvolution step is shown in Fig. 3.

For our semi-automated layer time extraction, we iteratively remove the last layer of the network and record the profile of the resulting network. The resulting profile usually exhibits a shorter execution time and possibly visible artifacts in the power profile after the output layer (which can be denoted to reading the output data). To neglect these artifacts, we determine the longest common sub-sequence between the aligned power profiles of the current sub-network and the previously measured networks. Computing the differences between the identified longest common sub-sequences we retrieve the execution times of each layer.

Fig. 4 shows the layer times extracted with our pipeline compared to the reports generated by OpenVino for the NCS2. Based on this method, we achieve an average error of $42\mu s$ for the execution of MobileNetV2 on the NCS2. At an average layer time of $311\mu s$, this means that we have to consider that for short layers, the error of the extracted layer times is rather large. However, since we are mainly interested in the efficiency of the large layers which make up for most of the energy consumption, we proceed with this approach.

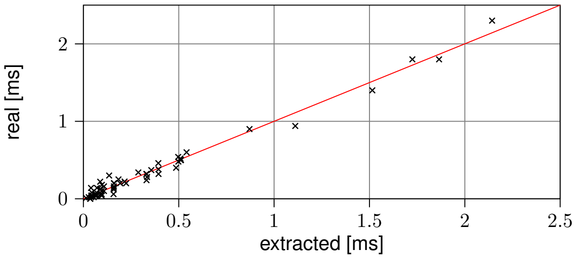


Fig. 4: Comparison of the extracted layer times vs. measured layer times for MobileNetV2 on NCS2

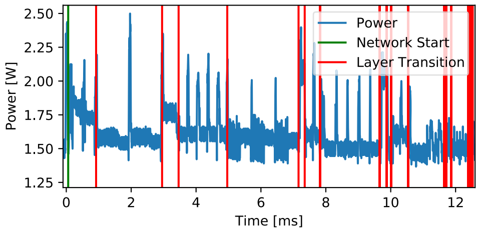


Fig. 5: The power profile of MobileNetV2 executed on NCS2 with annotated layer transitions

To verify the method, we first visualize the recorded measurements to analyze the power profile at 500 kHz sampling frequency. We can now visualize the timestamps of the transitions between the layers. Fig. 5 shows excerpts of such measurements of MobileNetV2 executed with batch size one on the NCS2 in synchronous mode.

Based on our observations, we find that both the NCS2 with $P_{\text{base}} \sim 1.4\,\text{W}$ and the edge TPU with $P_{\text{base}} \sim 1.1\,\text{W}$ exhibit a relatively high base power consumption after the network is loaded in comparison to the dynamic power P_{dyn} which is consumed additionally during the execution of the DNNs. The i.MX8M+ has an even higher idle power consumption with a $P_{\text{base}} \sim 2.9\,\text{W}$. As in this case we are not only measuring the NPU but also the CPU and memory components

Therefore, we denote the total power consumption as

$$P_{\text{total}} = P_{\text{base}} + P_{\text{dyn}} \tag{2}$$

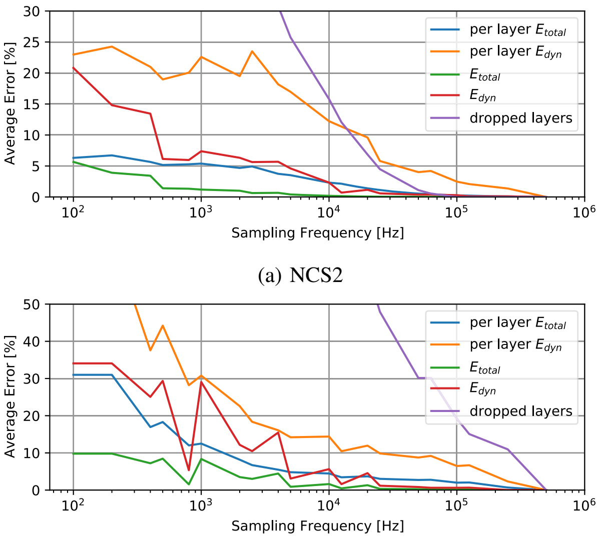
for the further experiments. We approximate the energy consumption of the entire network and the single layers with

$$E \approx t \cdot \overline{P}, \tag{3}$$

where t is the reported execution time and $\overline{P}$ is the mean power consumption of the executed layers and the entire network. With (2) and (3), we can compute the energies consumed due to base power P_{base} and dynamic power P_{dyn}.

V. Results

This section summarizes the results of applying the power profiling methodology presented in section IV.



(a) NCS2

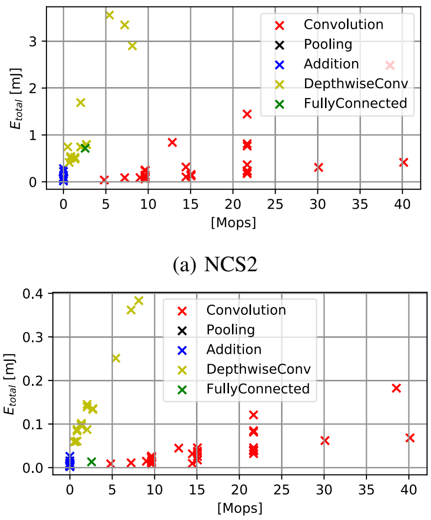
(b) edge TPU

Fig. 6: The error of measured energy and percentage of dropped layers in % for MobileNetV2 on the NCS2 and edge TPU with respect to 500 kHz sampling frequency

A. Sampling frequency

We first study the influence of the sampling frequency on the resulting errors with regards to the number of profiled layers, the energy consumption per layer, and the energy consumption for the entire network to gain an understanding for the required measurements frequencies for future devices.

Fig. 6a shows the error at a given sampling frequency compared to the results obtained with a sampling frequency of 500 kHz for the NCS2. The error of the total energy E_{total} is below 5 % at sampling rates above 100 Hz because the base power is a relatively major component. The error for the dynamic energy E_{dyn} grows faster with decreasing sampling frequency. The results show that a fine-grained understanding of the per-layer energy consumption requires a 10 kHz sampling frequency or higher for the NCS2. The denoted error per layer considers only the layers actually recorded during measurement. Notably, with decreasing sampling frequency, the number of dropped layers (layers with execution time shorter than the sampling window) quickly grows and amounts to 15 % at 10 kHz sampling frequency.The results for repeating the same measurements on the edge TPU are depicted in Fig. 6b. Compared with the NCS2, the edge TPU executes MobileNetV2 7 × faster, which leads to higher relative errors at the same sampling frequency. As expected the average network and per-layer error scales with the compute performance of the device and the latency of the network layers.



(a) NCS2

(b) edge TPU

Fig. 7: Total energy vs. number of operations for MobileNetV2 on the NCS2 and edge TPU

B. Energy versus the number of operations

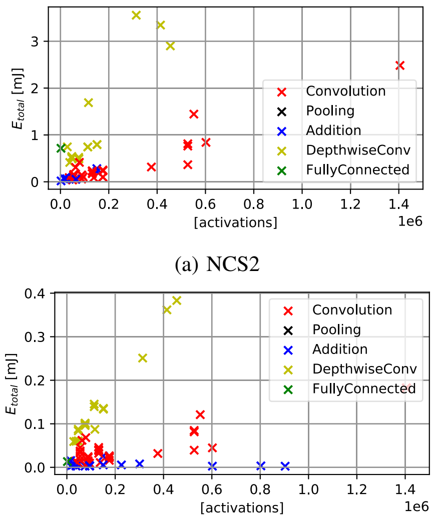
Comparing the measured energy consumption of the layers with the number of operations, we note that for both accelerators, not all layers are computed with equal energy efficiency. Fig. 7 shows the energy consumed for specific numbers of operations of different layers in MobileNetV2. Naturally, more operations lead to higher energy consumption, but the slope differs quite for different layer types. Depth-wise convolution layers (DepthwiseConv) have fewer operations but consume 4-6 times more energy per operation than ordinary convolutions. This behavior is explainable by the lower computational efficiency and the longer layer execution times of the NCS2 as well as the edge TPU computing depth-wise separable convolution layers. Moreover, within a layer type, there are significant differences. The energy efficiency difference between different convolution layers is up to 3× for the edge TPU and 4× for the NCS2. Thus, the number of operations is a poor proxy for estimating energy consumption for both accelerators.

C. Energy versus the number of activations

Considering the number of activations (Fig. 8), we see similar patterns for the NCS2 and the edge TPU, as the depth-wise convolution layers are computed with lower energy efficiency. Again, not only between different layer types but also within a layer type, there are significant differences in terms of energy efficiency per activation.

D. Energy versus latency

Fig. 9a and 9c depict the total energy and the dynamic vs. the runtime for the layers in MobileNetV2 on the NCS2. The total energy consumption correlates well with the runtime



(a) NCS2

(b) edge TPU

Fig. 8: Total energy vs. number of activations (output feature map size) for MobileNetV2 on the NCS2 and edge TPU

per layer (see Fig. 9a and 9b). One of the main reasons for this is the relatively high base power consumption of the NCS2 and the edge TPU compared to the dynamic power. When increasing the number of parallel inference requests for the NCS2 up to 4, the relation between dynamic and base energy shifts more towards dynamic energy. Thus, we also investigate the correlation between the dynamic energy per layer and the runtime. We find that the correlation is still fairly good; a layer that needs more time also needs more dynamic energy. However, the data points in Fig. 9c and 9d are not lined up on a straight line with a constant slope, which means the correlation is imperfect. Some layers consume up to 2 times more dynamic energy per time unit (more dynamic power) than others. Interestingly, on the NCS2, depth-wise separable convolution layers tend to consume less power than convolution layers.

Combining this observation with what we see in Fig. 7, depth-wise separable convolution layers are less efficiently executed by the hardware than convolution layers: they have fewer operations and require relatively more time but less power. It seems that they cannot keep the hardware as busy as ordinary convolution layers. One possible explanation is that depth-wise convolution layers are limited by the memory hierarchy rather than the computational capabilities of the NCS2 and the edge TPU. We found that the remaining layer types used in MobileNetV2 generally consume a higher amount of energy per operation than convolutional layers but are only responsible for an almost negligible portion of the total energy consumed per network.

E. Hardware settings

The NCS2 can be operated in two different modes, synchronous and asynchronous. In summary, synchronous means purely sequential execution while asynchronous allows pipelined execution. The *nireq* factor in Table I gives the number of requested parallel inferences. The inference latency slightly increases when pipelining inference requests. In contrast, the throughput, measured in frames per second, improves fairly significantly: between 33 % and 77 % when moving from sync to async-2 mode and between 5 % and 25 % when moving from async-2 to async-3. No improvement is found for the async-4 mode because the maximal amount of nireq has already been reached at async-3.

A better hardware utilization in the async-2 and async-3 modes, compared to sync-1, is reflected in the increased average power consumption, increasing by 17 % to 43 %.

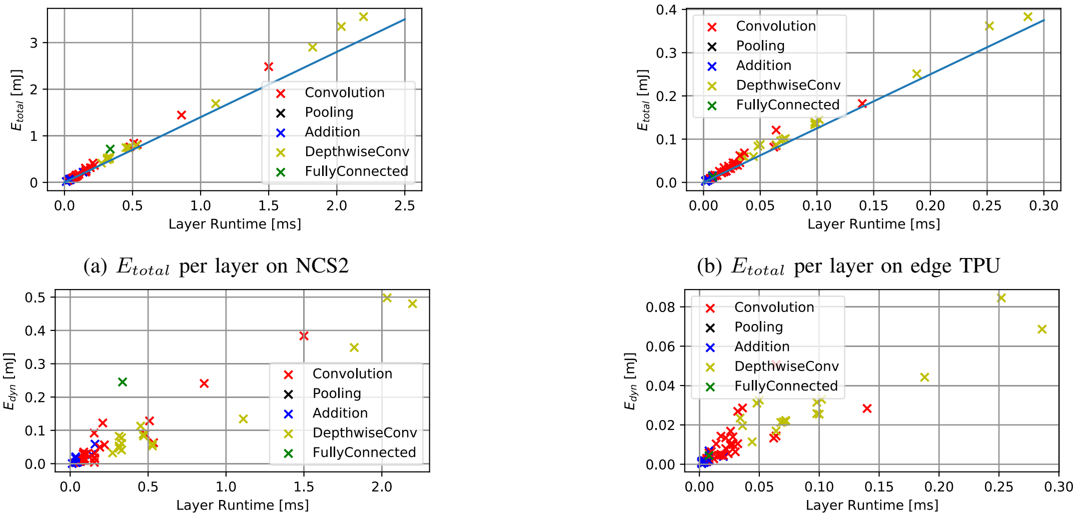
While the async modes increase the hardware utilization and power consumption, the total energy per inference is reduced, which reflects an overall better usage of hardware and energy resources. Comparing the energy values for the base and dynamic energy for the different execution modes, we can see that the total energy reduction per inference comes mostly from amortizing the high base energy over several inferences. For example, the dynamic energy per inference for executing YOLOv3 with 4 inference requests is almost twice as high as the base energy.

In contrast to the NCS2, the edge TPU does not allow multiple parallel inference requests but can be operated at two different clock frequency settings. Table I lists the frequency settings standard (std) and maximum (max).

The three test networks gain a speedup of between 10 % and 50 % when switching from std to max frequency mode. Due to the lower execution time with maximum frequency the base energy decreases while the dynamic energy increases for all three networks. On the other hand, with maximum frequency, the total energy decreases for all three networks while the mean power consumption increases by up to 15 %.

Depending on the selected hardware settings and the network architecture, the edge TPU is 2-10 times more efficient regarding total energy consumed per parameter and operation than the NCS2. This difference can be explained by the difference in data types the two operators are operating on (INT8 and FP16).

Lastly, the i.MX8M+ NPU also only supports INT8 execution. For comparison we also provide the numbers for execution on the CPU. As the measurements for the i.MX8M+ also include the power consumption of the CPU and other components, the base power consumption is significantly higher for the i.MX8M+ than for the edge TPU and the NCS2. Interestingly, for YOLOv3 and YOLOv3-tiny the i.MX8M+ outperforms the edge TPU in terms of latency. However, even though the execution time is smaller E_{total} and E_{dyn} are still higher than for the edge TPU. Additionally, we can see that for MobileNetV2 the edge TPU outperforms the i.MX8M+. We assume that this is due to the fact that the edge TPU is a better fit for depthwise convolution.



(a) E_{total} per layer on NCS2

(b) E_{total} per layer on edge TPU

(c) E_{dyn} per layer on NCS2

(d) E_{dyn} per layer on edge TPU

Fig. 9: Total and dynamic energy for the layers of MobileNetV2; the blue line denotes the base energy consumed by the edge TPU during the layer runtime

VI. Conclusion

We presented a methodology on how to perform power measurements an how to extract additional information of the power profiles. The method allows to gain additional insights with reduced additional development effort per platform. However, the method has some limitations regarding accuracy of the extracted data. It has to be considered, that the shorter the execution time of a NN the less accurate this method works. Therefore, we expect difficulties of applying this method for too powerful platforms or for small networks.

From our experiments with three networks and the accelerator platforms, we draw the following main conclusions, some expected and others less conspicuous.

- To obtain an average power measurement error lower than 15 %, a sampling frequency of 10 kHz or higher is required. For an error below 5 %, a sampling frequency of 200 kHz is recommended for the two USB accelerators.
- The number of operations is a poor predictor of energy consumption. Latency is a much better predictor with, in our experiments, an expected error margin of around 10 %. However, the correlation between latency and energy usage for individual layers can vary by up to 2× for dynamic energy, which means dynamic energy estimation based only on latency may be off by up to 100 %. Which means that for correctly predicting dynamic energy, additional factors would have to be considered.
- Settings of the hardware can significantly influence latency, throughput, and energy consumption. Specifically, the async-3 mode on NCS2 improves throughput by up to 100 % and energy per inference by up to 35 %, at a cost of increasing latency by up to 48 %.
- On the NCS2, increasing the number of parallel inference requests shifts the relation between base power and dynamic power towards the latter. As a further consequence, this aggravates the task of estimating the total energy due to the non-linear nature of the relationship between layer runtime and dynamic energy.
- For both USB accelerators, we can adjust the relationship between power consumption and energy per image through either the number of parallel inference requests or the clock frequency. In both cases, the power consumption increases when switching to the higher throughput modes, but with the overall result of a lower energy per image.

With our study, we wanted to obtain a better understanding of power and energy usage in specific state-of-the-art networks with diverse layer combinations on a given hardware platform. We conclude that latency can be used as a first-order estimate for power and energy consumption for a network and individual layers. However, if a more detailed understanding of a network and its layers is required for system energy budgeting or network optimizations, more detailed and precise measurements are required because the dependencies and influences are often non-linear and non-intuitive. In addition to providing insight into what can be derived from accurate power profiles of DNN accelerators, the understanding gained in this study can also be applied to power consumption estimation and energy-aware neural architecture search.

References

[1] Intel. Neural compute stick documentation. https://www.intel.com/content/www/us/en/developer/articles/guide/get-started-with-neural-compute-stick.html, 2019. Accessed: 2023-05-25. 1, 2

TABLE I: Speedup comparison of different Networks. *nireq* denotes the number of parallel inference requests. *Freq* denotes the frequency setting for the edge TPU.

HW	Network	*nireq*	F_{thr}(fps)	T_{lat}(ms)	P (mW)	E_{total}(mJ)	E_{base}(mJ)	E_{dyn}(mJ)	E/Gop(mJ)	$E/Mpar$(mJ)
NCS2 w.o. Host FP16	YOLOv3-tiny	1	21.2	41	2165	101.93	65.91	36.02	18.32	11.52
		2	35.3	52	2670	75.55	39.61	35.94	13.58	8.54
	5.6 Gop	3	43.1	46	2995	69.42	32.45	36.97	12.47	7.85
	8.8 Mpar	4	43.1	44	2954	68.54	32.48	36.06	12.32	7.75
	YOLOv3	1	2.6	363	2505	960.92	537.04	423.88	14.69	15.61
		2	4.4	400	3413	769.61	315.69	453.92	11.76	12.50
	65.8 Gop	3	4.7	425	3615	764.89	296.22	468.67	11.69	12.42
	61.6 Mpar	4	4.9	390	3604	742.50	288.43	454.07	11.35	12.06
	MobileNetV2	1	49.3	21	1806	36.60	28.37	8.23	60.84	10.55
		2	87.2	23	2118	24.29	16.06	8.23	40.38	7.00
	0.6 Gop	3	90.4	31	2164	23.95	15.49	8.46	39.81	6.90
	3.4 Mpar	4	92.4	53	2162	23.39	15.15	8.24	38.88	6.74

HW	Network	*Freq*	F_{thr}(fps)	T_{lat}(ms)	P (mW)	E_{total}(mJ)	E_{base}(mJ)	E_{dyn}(mJ)	E/Gop(mJ)	$E/Mpar$(mJ)
Edge TPU w.o. Host INT8	YOLOv3-tiny	std	46.3	22.3	1407	30.40	22.28	8.12	5.46	3.44
		max	51.0	19.6	1528	29.95	20.21	9.73	5.38	3.39
	YOLOv3	std	6.3	158.3	1519	240.50	163.27	77.23	3.68	3.91
		max	7.0	142.0	1657	235.36	147.29	88.06	3.60	3.82
	MobileNetV2	std	331.3	3.0	1422	4.29	3.11	1.18	7.13	1.24
		max	512.3	1.9	1658	3.23	2.02	1.21	5.37	0.93

HW	Network	*Freq*	F_{thr}(fps)	T_{lat}(ms)	P (mW)	E_{total}(mJ)	E_{base}(mJ)	E_{dyn}(mJ)	E/Gop(mJ)	$E/Mpar$(mJ)
i.MX8M+ INT8	YOLOv3-tiny	npu	102.8	9.7	4398	42.78	26.32	16.47	7.64	4.84
		cpu	1.3	758.5	3917	2971.33	2421.60	549.73	534.41	335.93
	YOLOv3	npu	9.5	105.0	4788	502.51	289.93	212.58	7.64	8.16
		cpu	0.1	7706.6	3434	26462.35	20632.56	5829.79	402.16	429.58
	MobileNetV2	npu	97.1	10.3	3807	39.21	28.11	11.11	65.35	11.53
		cpu	8.4	119.7	3375	403.97	327.90	76.07	673.28	118.81

[2] Coral. Usb accelerator datasheet. https://coral.ai/docs/accelerator/datasheet/, 2021. Accessed: 2023-05-25. 1, 2, 3

[3] NXP. i.mx 8m plus documentation. https://www.nxp.com/products/processors-and-microcontrollers/arm-processors/i-mx-applications-processors/i-mx-8-applications-processors/i-mx-8m-plus-arm-cortex-a53-machine-learning-vision-multimedia-and-industrial-iot:IMX8MPLUS, 2023. Accessed: 2023-05-25. 1

[4] David Cantero, Iker Esnaola-Gonzalez, José Miguel-Alonso, and Ekaitz Jauregi. Benchmarking object detection deep learning models in embedded devices. *Sensors*, 22(11), 2022. 1

[5] Cody Coleman, Daniel Kang, Deepak Narayanan, Luigi Nardi, Tian Zhao, Jian Zhang, Peter Bailis, Kunle Olukotun, Christopher Ré, and Matei Zaharia. Analysis of dawnbench, a time-to-accuracy machine learning performance benchmark. *ACM SIGOPS Oper. Syst. Rev.*, 53(1), 2019. 2

[6] Colby R. Banbury et al. Mlperf tiny benchmark. In Joaquin Vanschoren and Sai-Kit Yeung, editors, *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks*, 2021. 2

[7] V. J. Reddi et al. Mlperf inference benchmark. In *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020. 2

[8] Leandro Libutti, F. Igual, L. Piñuel, Laura C. De Giusti, and M. Naiouf. Benchmarking performance and power of USB accelerators for inference with MLPerf. In *1st Workshop on Accelerated Machine Learning (AccML)*, 2020. 2

[9] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017. 2

[10] Michaela Blott et al. Evaluation of optimized cnns on heterogeneous accelerators using a novel benchmarking approach. *IEEE Trans. Computers*, 70(10), 2021. 2

[11] Stefan Reif, Benedict Herzog, Judith Hemp, Timo Hönig, and Wolfgang Schröder-Preikschat. Precious: Resource-demand estimation for embedded neural network accelerators. In *First International Workshop on Benchmarking Machine Learning Workloads on Emerging Hardware*, 2020. 2

[12] Yannan Nellie Wu, Joel S. Emer, and Vivienne Sze. Accelergy: An architecture-level energy estimation methodology for accelerator designs. In *Proceedings of the International Conference on Computer-Aided Design, ICCAD*, 2019. 2

[13] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David M. Brooks. Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA*, 2014. 2

[14] Intel. OpenVINO Toolkit. https://software.intel.com/en-us/openvino-toolkit, 2021. 3

[15] Coral. Tensorflow models on the edge tpu. https://coral.ai/docs/edgetpu/models-intro/, 2021. 3

[16] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, 2018. 3

[17] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *CoRR*, abs/1804.02767, 2018. 3

[18] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. Isolation forest. In *Proceedings of the 8th IEEE International Conference on Data Mining (ICDM)*, 2008. 4

Weighted Quantization-Regularization in DNNs for Weight Memory Minimization towards HW Implementation

Matthias Wess, Sai Manoj Pudukotai Dinakarrao, *Member, IEEE,* and Axel Jantsch, *Member, IEEE*

***Abstract*—Deployment of Deep Neural Networks (DNNs) on hardware platforms is often constrained by limited on-chip memory and computational power. The proposed weight quantization offers the possibility of optimizing weight memory alongside transforming the weights to hardware friendly data-types. We apply *Dynamic Fixed Point* and *Power-of-two quantization* in conjunction with *Layer-wise Precision Scaling* to minimize the weight memory. To alleviate accuracy degradation due to precision scaling, we employ quantization-aware fine-tuning. For fine-tuning, *quantization-regularization* and *weighted quantization-regularization* are introduced to force the trained quantization by adding the distance of the weights to the desired quantization levels as a regularization term to the loss-function. While Dynamic Fixed Point quantization performs better when allowing different bit-widths for each layer, Power-of-two quantization in combination with retraining allows higher compression rates for equal bit-width quantization. The techniques are verified on an All-Convolutional Network. With accuracy degradation of 0.10 percentage points, for Dynamic Fixed Point with Layer-wise Precision Scaling we achieve compression ratios of 7.34 for CIFAR-10, 4.7 for CIFAR-100 and 9.33 for SVHN dataset.**

***Index Terms*—Convolutional Neural Networks, Quantization, Regularization, Memory minimization**

I. Introduction

STARTING with AlexNet [1] Deep Convolutional Neural Networks (DCNN) have been gaining attention by delivering impressive results on challenging problems, such as object recognition on ImageNet dataset [2] or facial recognition [3]. The adaptation of such DCNNs and deep neural networks (DNNs) in various applications including autonomous driving, medical diagnosis [4], [5] and machine translation [6] led to an ever increasing amounts of data to process under high performance requirements.

Most of these applications can be described as supervised learning tasks, split into training phase and inference. In the training phase, the algorithm is optimized to solve a certain task for the training data. The architecture of a DCNN or DNN is defined by the number of layers and their functionality (e.g. Convolutional, Fully Connected, Pool, Batch-Normalization) and the layer-specific parameters which define the dimensions and behavior of the layer in forward and backward-propagation. To train the defined architecture on a given training data, the labeled data is fed through the network and in back-propagation, the layer-specific weights are adjusted to decrease the error between the output and original label. For inference, DNN employs the model derived during training phase on the test or unknown data. The ability to correctly process the new data based on training data is called generalization ability.

Despite the state-of-the-art DNNs taking one or several high-end GPUs and up to several days to train, inference can be performed on a broad spectrum of platforms including CPUs, GPUs, FPGAs, and ASICs. With the increasing size of DNNs (e.g. ResNet [7] up to 152 layers), even the complexity of inference is also exacerbating due to more critical requirements and constraints such as limited power consumption, high throughput or hard real-time processing. There are several challenges that hinder the efficient deployment and inference of the State-of-the-art Deep Neural Networks (DNN) on embedded resource constrained platforms. The two biggest challenges are the large size of the networks and the total number of necessary operations in feed-forward computation, since a hardware accelerator design can be bound either by the limit of parallel operations, or by the memory interface transmission rate [8], [9]. As a consequence, model compression and increasing the efficiency of computations, are two legitimate ways to reach hardware requirements.

Recent works [10], [11] have proven the robustness of DNNs to compression of weights and simplification of activation functions with high number of parameters and the resulting redundancy [5], [10], [12], [13], [14]. This enables several techniques including weight sharing [10], [15], pruning [16] and Huffman Encoding [10] to reduce external memory access. Pruning not only reduces the memory footprint of a DNN model, but also allows skipping of multiplications with 0, thus reducing the amount of total multiplications [17], [18]. To reduce also power consumption within operations, the model parameters have to be quantized in specific formats a dedicated hardware can make use of. Dynamic fixed point [14], [19] and power of two quantization [11] are two hardware friendly formats that enable performing multiplications either as low-precision multiplications or simple shift operations.

There are several approaches on how to best prepare a DNN for inference with low precision data types. On one side when employing the state-of-the-art DNNs it is desirable to directly make use of pre-trained models without architectural

Matthias Wess is with ICT, TU Wien, Vienna, Austria, and Siemens AG, Vienna, Austria (e-mail: matthias.wess@student.tuwien.ac.at).

Sai Manoj P.D. is with George Mason University, Fairfax, VA, United States (e-mail: saimanoj.p.2013@ieee.org).

Axel Jantsch is with ICT, TU Wien, Vienna, Austria (e-mail: axel.jantsch@tuwien.ac.at)

This article was presented in the International Conference on Hardware/Software Codesign and System Synthesis 2018 and appears as part of the ESWEEK-TCAD special issue

adjustments. [13] and [20] propose methods for layer-wise bit-width optimization without retraining but not for bit-width optimization followed by retraining. Furthermore, to fully leverage optimized hardware accelerators for efficient inference (e.g. [21]) it can be desirable to force certain quantization [11], compression or pruning schemes [17] in an additional fine-tuning step. [11] proposes incremental weight quantization while incorporating a Power-of-two datatype and achieve almost lossless quantization for several DNNs. Other works such as [22], [23] employ stochastic quantization methods to during training. In stochastic training the algorithm stores a floating point value and the quantized value at the same time and for each feed forward computation the quantized weights are newly computed on a stochastic basis.

This paper makes the following contributions:

- We propose weighted quantization-regularization (WQR), a method for trained low precision quantization of weights in Neural Networks to any given quantization scheme.
- We combine Layer-wise Precision Scaling [20] with weighted quantization-regularization to reduce the loss in classification performance while increasing the compression rate.
- We analyze the benefits of Power-of-2 (Po2) and Dynamic Fixed Point (DFP) based quantization in our setting and in combination with weighted quantization-regularization and layer-wise bit-width optimization.

Aiming at highly efficient implementation in FPGAs, we perform evaluation for quantization-regularization for dynamic fixed point [23] and power of two quantization [11] schemes on CNNs. We apply the proposed algorithm on SVHN CIFAR-10 and CIFAR-100 dataset for two different quantization schemes and show that weighted quantization-regularization decreases loss in classification performance in comparison to direct weight quantization for All-Convolutional Network on CIFAR-10 from 1.5% to 0%. The results suggest that the proposed algorithm reduces accuracy loss due to quantization.

II. Motivational Case Study

Figure 1 explains with a simple example the two main parts of the paper. Assuming a two-layer neural network with two layers with 600 and 900 weights respectively, we want to achieve model compression by reducing the number of bits stored per weight and specific quantization of the weights to enhance the computational energy efficiency.

First note, that layer 2 has a stronger impact on the size of the weight memory, as it contains more weights. Thus, it is beneficial in terms of memory footprint to reduce the bit-width of its weights more than those of layer 1. However, quantization also negatively affects the accuracy of the algorithm, due to weight quantization errors. Therefore we apply Layer-wise Precision Scaling (Fig. 1a) to find the best trade-off between compression due to quantization and accuracy degradation. While for the example in figure 1 uniform 3-bit quantization leads to 4.5kbit weight memory, with Layer-wise Precision Scaling applied according to figure 1a only 3.6kbit weights need to be stored, allowing us to increase compression ratio by a factor 1.25.

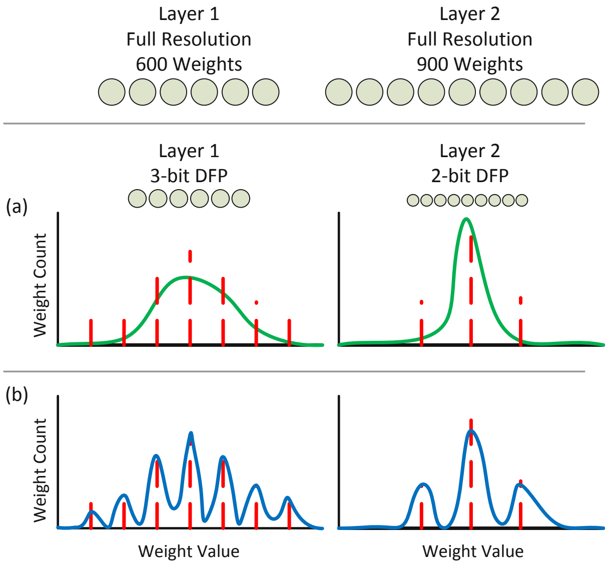


Fig. 1. An example Network with two layers demonstrating (a) Layer-wise Precision Scaling (b) Retraining with Quantization-regularization. Starting with two layer network at first (a) the bit-widths of both layers are adjusted by defining a lower precision format with the quantization levels marked as dotted lines in the histogram charts. To reduce the quantization error, (b) retraining with additional regularization, decreasing the average distance of the weights to the quantization levels, is performed.

To alleviate the accuracy degradation, performing trained quantization by applying additional regularization with the goal of reducing the weight quantization error results in an increase the accuracy. For state-of-the-art DNNs Layer-wise Precision Scaling shows even higher efficiency due to the higher variation of numbers of weights per layer (table III).

III. Proposed Method

Figure 2 illustrates the entire quantization flow for learned model compression which can be separated into three steps:

1) **Quantization Scheme Evaluation:** We define and analyze two quantization strategies in terms of their effectiveness for hardware-friendly execution their advantages and disadvantages during fine-tuning and the resulting performance.
2) **Layer-wise Precision Scaling:** To increase the model compression ratio we apply layer-wise precision scaling, meaning that for each layer different bit-widths are used for weights. Thereby we study the influence of selecting different bit-widths per layer on the resulting classification accuracy.
3) **Retraining with WQR and QR:** The last task focuses on reducing accuracy degradation occurring due to quantization. As loss of accuracy is induced due to the change of weight magnitudes when approximating them by rounding to the nearest quantization level, we aim to force weights to reduce their distance to such quantization levels in retraining, thus increasing classification accuracy of the quantized network.

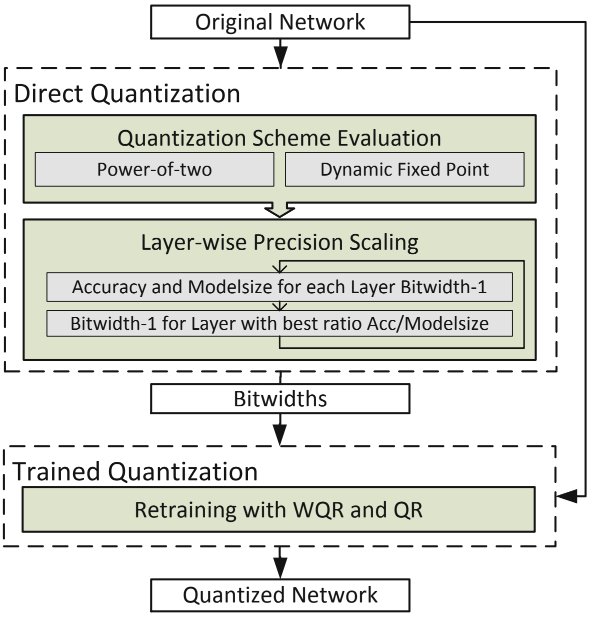

Original Network
Direct Quantization
Quantization Scheme Evaluation
Power-of-two
Dynamic Fixed Point
Layer-wise Precision Scaling
Accuracy and Modelsize for each Layer Bitwidth-1
Bitwidth-1 for Layer with best ratio Acc/Modelsize
Bitwidths
Trained Quantization
Retraining with WQR and QR
Quantized Network

Fig. 2. Learned Weight Quantization step by step

While the first two steps serve for finding the best quantization method and bit-width for each layer when applying quantization without retraining, in the third step we perform retraining aiming to reduce the accuracy loss caused by quantization. In our flow, the weights are not first quantized and then retrained, but we always start from the high accuracy model, fine-tune weights with modified loss-functions and then perform quantization. This approach has the advantage that the network parameters are trained in full precision but with the additional regularization terms which cause the weights to reduce their distance to the desired quantization levels before performing the actual quantization step. To better distinguish we use the term direct quantization for quantization without any fine-tuning. Trained quantization on the other hand consists of fine-tuning, followed by the actual quantization step. Input for the quantization process is a Deep Neural Network (DNN) with N convolutional and/or fully connected layers and weight-tensors $W_n, 0 < n < N$ of arbitrary resolution. Details on the network used for evaluation can be found in table III. Table I lists the variables used in this work.

A. Direct Quantization

In direct quantization, the original network M is expressed as M_q where the weights W_n of each layer are represented as W_{q_n}. The values of W_{q_n} are determined by rounding each element of W_n to the quantization level with the smallest absolute distance of a defined quantization scheme Q.

1) Quantization Scheme Evaluation: Here, we present Power-of-two [11] and Dynamic Fixed Point [23], [19], two different quantization schemes and compare their properties for direct and trained quantization.

TABLE I
VARIABLES USED IN THIS WORK

Variable	Comment
M	Original network model
M_q	Quantized network model
N	Number of layers
W_n	Original Weights
W_{q_n}	Quantized weights
b_n	Bit-widths for layers $n = 0...N$
Q_n	Quantization schemes for layers $n = 0...N$
Q_{p2}	Power-of-2 quantization scheme
n_1	Maximum exponent for Power-of-2 quantization
n_2	Minimum exponent for Power-of-2 quantization
s	Maximum absolute weight within the selected layer
Q_{dfp}	Dynamic Fixed Point quantization scheme
B	Unscaled Dynamic Fixed Point quantization scheme
acc_M	Classification accuracy of the original network
acc_{M_q}	Classification accuracy of the quantized network
Δacc	Accuracy degradation due to quantization
W_{mem}	Weight memory bits
λ_1	Quantization-Regularization scale factor
QR	Quantization-Regularization Term
λ_2	Weighted Quantization-Regularization scale factor
WQR	Weighted Quantization-Regularization term

a) Power-of-two quantization: We implement Power-of-two (Po2) quantization similar as in [11]. Q_{p2} is given as

$$Q_{p2} = \{\pm 2^{n_1}, ..., \pm 2^{n_2}, 0\}. \tag{1}$$

n_1 and n_2 are integers with

$$\begin{aligned} n_1 &= \lfloor log_2 \frac{4s}{3} \rfloor \quad (2) \\ s &= max(abs(W)). \quad (3) \end{aligned}$$

For a given bit-width b and n_2 are defined by

$$n_2 = n_1 - (2^{b-1} - 1). \tag{4}$$

Thus, the quantization levels depend on the distribution of weights, especially on the weight with the highest absolute value. By adding ‘0’ as a quantization level, we enable power-of-two quantization to also serve as a pruning mechanism when applied to weight matrices, as small weights are rounded to zero. In experiment symmetrical quantization schemes lead to higher classification accuracies for the quantized networks, therefore we only use $2^b - 1$ of 2^b possible quantization levels.

b) Dynamic Fixed Point: Dynamic fixed point (DFP) data type is successfully used in several works for either direct quantization or retrained model compression [23], [19]. For DFP quantization, we first define a set of $2^b - 1$ equidistant quantization levels:

$$B = \{\pm 2^{b-1} - 1, \pm 2^{b-1} - 2, ..., 0\}. \tag{5}$$

Similar to Po2 quantization, we prefer a symmetric quantization scheme. Next B is normalized and scaled, depending on the distribution of weights:

$$Q_{dfp} = \frac{B}{2^{b-1}} * 2^{n_1}. \tag{6}$$

Figure 3 depicts the distribution of weights for an example layer of a CNN, before and after quantization. While figure 3(a) shows the distribution for Po2 quantization, figure 3(b) illustrates the distribution for DFP quantization. As can be seen that po2 has much irregular quantization values compared to DFP, and also considers the values close to '0' which might help to retain the information with lower weights and aid in improving the accuracy.

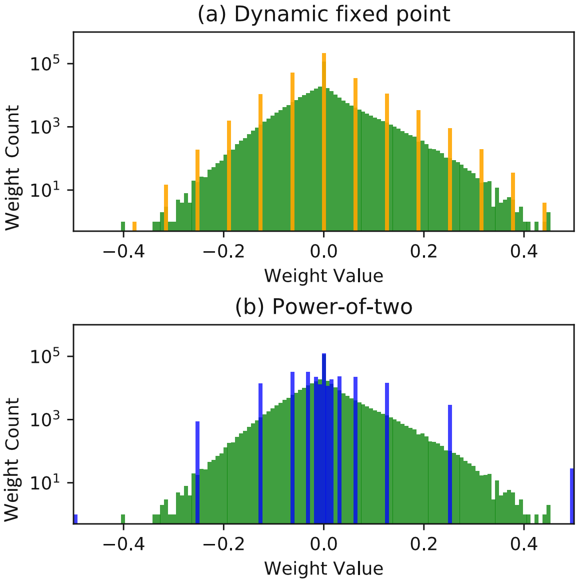


Fig. 3. Distributions before and after direct weight quantization for (a) Dynamic Fixed Point and (b) Power-of-two quantization

As accuracy degradation of the quantized model M_q in comparison to the original model M is a result of the quantization error, it is necessary to understand the relation between bit-width and quantization error for both datatypes.

With DFP quantization, the mean square error (MSE) can be reduced with increasing bit-width, since every additional bit divides the intervals in half (see fig. 3a). Meanwhile when increasing bit-width in Po2 quantization, the new quantization levels are always added close to '0' (see fig. 3b). As a consequence with Po2 quantization, the quantization error can only be reduced to a certain extent. Figure 4 shows the resulting mean square errors for one weight-tensor of an example layer when applying different bit-widths.

In addition, we consider the amount of pruned weights as an important factor for model compression. In comparison to DFP, Po2 quantization decreases sparsity within the weight matrices as a result of quantization, due to the higher density of levels close to '0'. Therefore to fully benefit from the advantages of sparsity, an additional pruning step before retraining is recommended. In figure 5 the number of pruned weights depending on the selected bit-width is shown for Po2 and DPF quantization.

Figures 4 and 5 suggest that for direct Po2 quantization bit-widths higher than 4-bit do not further decrease the Δacc but 4-bits in comparison to 5-bits slightly increase sparsity. On the other hand, by scaling the bit-width of DFP the resulting MSE can be reduced exponentially (Fig. 4) meaning that even bit-widths higher than 8 bit deliver more accurate results. In terms of sparsity, DFP prunes more weights than Po2 for bit-widths of four and higher.

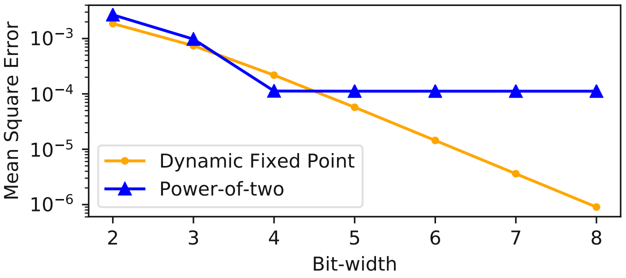


Fig. 4. Mean square error for an example layer when applying Power-of-two and Dynamic Fixed Point quantization with different bit-widths. While DFP quantization decreases the quantization error exponentially with increasing bit-width, with Po2 quantization the quantization error reaches the minimum already at bit-width 4.

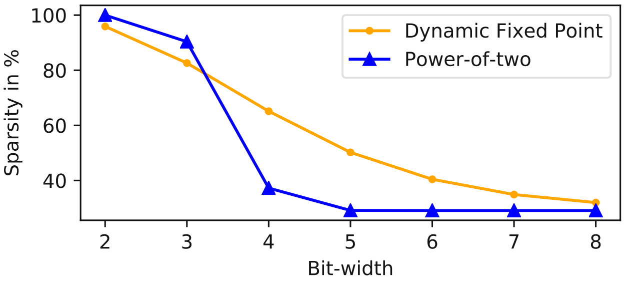


Fig. 5. With increasing bit-widths the sparsity due to quantization decreases for Dynamic Fixed Point and Power-of-two quantization. Sparsity denotes the amount of weights that are 0 relative to the total amount of Weights

Based on these observations we expect that for direct DFP quantization Δacc can be reduced to almost 0, based on Layer-wise Precision Scaling. For direct Po2 quantization we expect a higher Δacc and no significant increase of accuracy for bit-widths higher than five. It can be seen in figures 6 and 10 that these expectations are confirmed.

2) Layer-wise Precision Scaling: Secondly, to further reduce the model-size, we apply different quantization schemes per layer by optimizing bit-widths. In comparison to choosing equal bit-width for each layer, due to the varying amount of parameters and varying distribution of weights between layers, selecting fitting quantization schemes for each layer can enable lower bit-widths per layer without reducing the resulting accuracy. For the experiments we assume either DFP or Po2 quantization. For an arbitrary network M with accuracy acc_M applying weight quantization with the set of bit-widths b_n leads to accuracy acc_{Mq} and weight memory bits

$$W_{\text{mem}} = \sum_{n}^{N} \text{card}(W_n) * b_n \tag{7}$$

where $\text{card}(A)$ denotes the cardinality of set A. For each b_n we compute the resulting accuracy degradation

$$\Delta acc = acc_M - acc_{Mq} \tag{8}$$

and iteratively decrease the bit-width of the layer where a lower bit-width leads to the smallest product of $\Delta acc * W_{mem}$ (see algorithm 1).

Algorithm 1 Layer-wise Precision Scaling

```
procedure LAYER-WISE PRECISION SCALING($M$)
    initialize $b_n$
    while $\Delta acc < \epsilon$ do
        for all $n$ in layers do
            bitwidth of $layer_n$ - 1
            Compute $Acc_{M_q}$, $\Delta acc$ and $W_{mem}$
            bitwidth of $layer_n$ + 1
        end for
        Decrease bitwidth of layer with min($\Delta acc * W_{mem}$)
    end while
end procedure
```

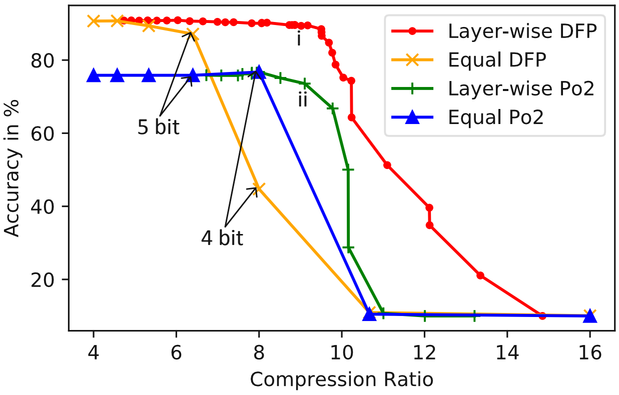


Fig. 6. Layer-wise Precision Scaling compared with equal bit-width quantization for DFP and Po2 quantization. Compression ratio is the ratio between 32bit weight memory and the weight memory for the quantized network. Point i indicates DFP with $b_n = [7\ 7\ 7\ 4\ 4\ 3\ 3\ 7\ 7]$, point ii indicates Po2 with $b_n = [4\ 4\ 4\ 4\ 3\ 3\ 4\ 4\ 4]$.

Figure 6 shows the results for Layer-wise Precision Scaling performed by algorithm 1 on All-Convolutional Network [24] for CIFAR-10. We can deduce that while DFP quantization also allows direct quantization, whereas for power-of-two quantization almost always an additional fine-tuning step is necessary to achieve high accuracy results.

B. *Trained Quantization*

The used datasets (CIFAR-10, CIFAR-100 and SVHN) are already divided into **test data** and **training data**. While with Layer-wise Precision Scaling as described in section III-A2 focuses on decreasing $\Delta acc * W_{mem}$ without retraining, we can additionally reduce Δacc by retraining the original network on the training data with the goal of increasing accuracy of the classifier on test data. As a consequence we use the performance metrics in table II.

1) Quantization-Regularization: To decrease Δacc for a selected set of bit-widths b_n, we need to find the best set of W_n so that approximation with Wq_n achieves a maximum of acc_{Mq}. As shown in [13], [25] the degradation of classification accuracy of a DNN due to quantization is directly related to the signal to quantization-noise ratio (SQNR) and the amount of weights per layer, as both influence the SQNR of the intermediate layer outputs and as a consequence the resulting network outputs. Therefore retraining network weights to achieve lower SQNR without reducing acc_M, leads to an increased accuracy of the quantized network. To enforce weight quantization during the training phase we define the quantization-regularization (QR) term as

TABLE II
PERFORMANCE METRICS USED FOR FINE-TUNING

Variable	Metrics	Comment
acc_{tr}	Training Accuracy	Accuracy of the network on the training data
acc_M	Test Accuracy	Accuracy of the network on the test data
acc_{M_q}	Quantized Test Accuracy[1]	Accuracy of the network with quantized weights on the test data
acc_{init}	Init. Quantized Test Accuracy	Accuracy of the network with direct quantized initial weights on the test data
CR	Compression Ratio	Ratio W_{Mem} of original model to W_{Mem} of quantized model

$$QR = \sum_{n}^{N} \sum_{i}^{\text{card}(W_n)} \frac{\mid W_{n_i} - Wq_{n_i} \mid}{\max(Q_n) * \text{card}(W_n)} \tag{9}$$

which expresses the mean of the absolute weight distances of each weight to the corresponding quantized value.

By adding the QR-term to the loss function (eq. 10) weights are forced closer to the quantization levels during retraining.

$$\text{Modified Loss} = \text{Loss} + \lambda_1 * QR \tag{10}$$

During fine-tuning with the parameter λ_1 the trade-off between $min(Loss)$ and $min(QR)$ and, as a consequence, between $min(\Delta accuracy)$ and $max(acc_M)$ can be adjusted. For the experiments we applied fixed λ_1 and linearly increasing λ_1 (e.g. $\lambda_1 = 10*epoch$). Figure 7 depicts the trained quantization process. With each epoch the weights are pulled closer to the quantization levels, thus decreasing QR and Δacc.

While choosing a high (> 1000) λ_1 leads to fast quantization with strong accuracy degradation, a low λ_1 value (< 1) does not enforce quantization. Either way, much like weight decay low λ_1 values can help avoiding overfitting during training.

2) Weighted Quantization-Regularization: While in normal QR each weight within one layer is considered equally important for reaching high classification accuracy, the efficiency of pruning [16], [17], [15] shows that especially weights with small magnitudes can be changed without reducing the accuracy of the network. Similarly to [8], the weights can be divided into two disjoint subsets, where QR is applied on one of the subsets while the other weights are being retrained without QR. Going one step further we can multiply the QR value of each weight with the absolute magnitude of the weight[2]. This strategy forces quantization stronger on

[1] For the computation of Quantized Test Accuracy, the weights of the network are directly quantized after each epoch.

[2] Previously the sum of weights is normalized to 1 for each layer

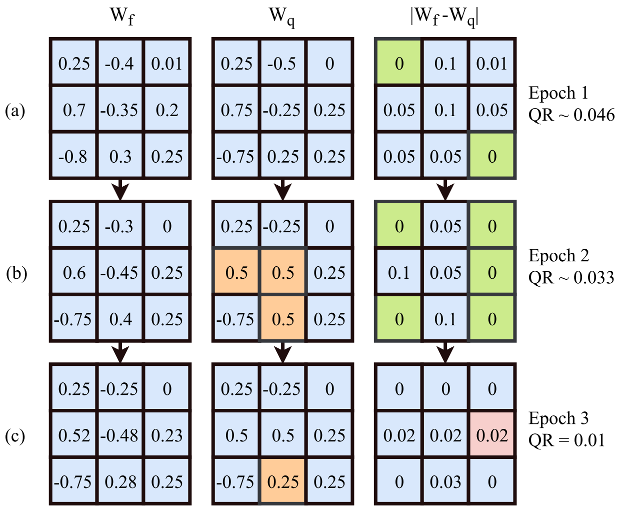


Fig. 7. Illustration of Quantization-Regularization. The first column shows the floating point values of the weights on which the actual training is performed. The second column shows the quantized weights, and the third column the element-wise absolute difference of floating point and quantized weights. Changes of quantized weights are marked in the second column. Weight update for W_f is performed based on backpropagation of the modified loss function (eq. 10). Successfully quantized weights are marked in the third column. After Epoch 1 (a) the weights are hardly regularized and QR is relatively large. Epoch 2 (b) shows that due to regularization, weights are pulled closer to the quantization levels $\mathbf{Q_n} = \{\mathbf{0}, \pm\mathbf{0.25}, \pm\mathbf{0.5}, \pm\mathbf{0.75}\}$ and QR gets smaller. For three of the weights the resulting quantization level changed, and weights decrease their distance to the next quantization level. Epoch 3 (c) shows further reduction of the QR term.

weights with higher magnitudes which can be especially useful for Po2 quantization, where density of quantization levels decreases with increasing weight values. Therefore we define the Weighted Quantization-Regularization (WQR) term as

$$WQR = \sum_{n}^{N} \sum_{i}^{\text{card}(W_n)} (\frac{\mid W_{n_i} - W_{q_{n_i}} \mid\mid W_{n_i} \mid}{\max(Q_n)^2 * \text{card}(W_n)} \tag{11}$$

and similarly to equation 10 we can weight the trade-off between accuracy and weight regularization with λ_1 and λ_2 (eq. 12)

$$\text{Modified Loss} = \text{Loss} + \lambda_1 * QR + \lambda_2 * WQR \tag{12}$$

Again during training the parameters λ_1 and λ_2 have to be adjusted carefully to reach the desired improvement of Acc_{Mq}, without at the same time decreasing Acc_M. In our experiments we found a linear increasing λ_2 to work best (e.g. $\lambda_2 = 10 *$ epoch). For fine-tuning, we use algorithm 2.

Figure 8 illustrates the fine-tuning process for 4-bit equal bit-width Po2 quantized All-Convolutional Net for CIFAR-10. At the beginning of the fine-tuning process the term $\lambda_2 * WQR$ increases due to the increasing λ_2 value, while the WQR-Term decreases exponentially. At Epoch 200 learning rate is decreased from $1e^{-4}$ to $1e^{-5}$ leading to the drop of $\lambda_2 * WQR$. This can be explained by fact that a larger learning rate leads to larger weight changes. If the weights are already close to the quantization levels a lower learning rate can lead to better approximation of the weights to the quantization levels.

Algorithm 2 Fine-tuning with Quantization-Regularization and Weighted Quantization Regularization

```
procedure TRAINED QUANTIZATION($M$, $b_n$, $\lambda_1$, $\lambda_2$)
    for epochs do
        for all Mini Batches do                ▷ Train on Training Data
            for $n >= N$ do                    ▷ Quantize all Layers
                $W_{q_n}$ = Quantize($W_n$,$Q_n$)
            end for
            Loss+$\lambda_1 * QR(W_n, W_{q_n}) + \lambda_* WQR(W_n, W_{q_n})$
            Backpropagation(Loss,QR,WQR,$\lambda_1$,$\lambda_2$)
        end for
        Compute $Acc_M$                        ▷ Test Accuracy
        Compute $Acc_{M_q}$                    ▷ Quantized Test Accuracy
    end for
    return $M_q$                               ▷ Return Quantized Model
end procedure
```

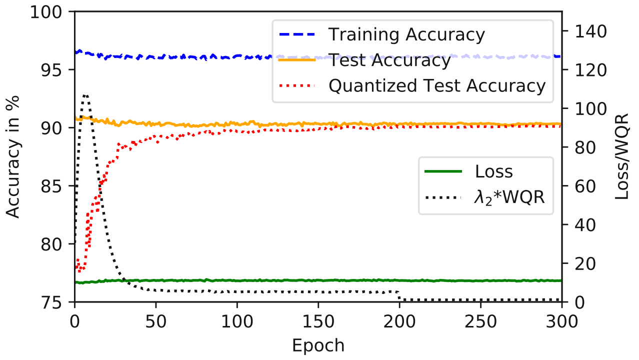


Fig. 8. Fine-tuning of All-Convolutional Net for CIFAR-10 with linear increasing $\boldsymbol{\lambda_2}$ for 4-bit equal bit-width Po2 quantization. $\boldsymbol{\Delta}$**Acc** is decreased from $\mathbf{14.09}\%$ to $\mathbf{0.14}\%$ resulting in Quantized Test Accuracy of $\mathbf{90.18}\%$ in comparison to initial Floating Point Test Accuracy $\mathbf{90.83}\%$.

C. Summary and Analysis

In combination, the discussed techniques for quantization (sec. III-A), Layer-wise Precision Scaling (sec. III-A2) and fine-tuning with QR (sec. III-B1) and WQR (sec. III-B2) facilitate DNN weight compression.

The above discussed two quantization schemes behave differently during the quantization process and require different quantization strategies. For bit-widths higher than 7 bit DFP can be applied without any retraining and still achieves almost floating point accuracy. For equal bit-width DFP quantization with 7 bit and less, Δacc increases and fine-tuning is necessary to reach the accuracy of the original network. On the other hand Po2 quantization always requires retraining, as even the use of bit-widths higher than five reduce the quantization error only to a certain extent.

To increase the compression ratio when applying DFP quantization, layer-wise precision scaling is an effective method, since not all layers require the same the bit-width for high accuracy. For instance, in modern convolution-only networks, layers with fewer parameters require larger bit-widths [13]. As a result, when applying layer-wise precision scaling the weights of the output and input layers are kept at high

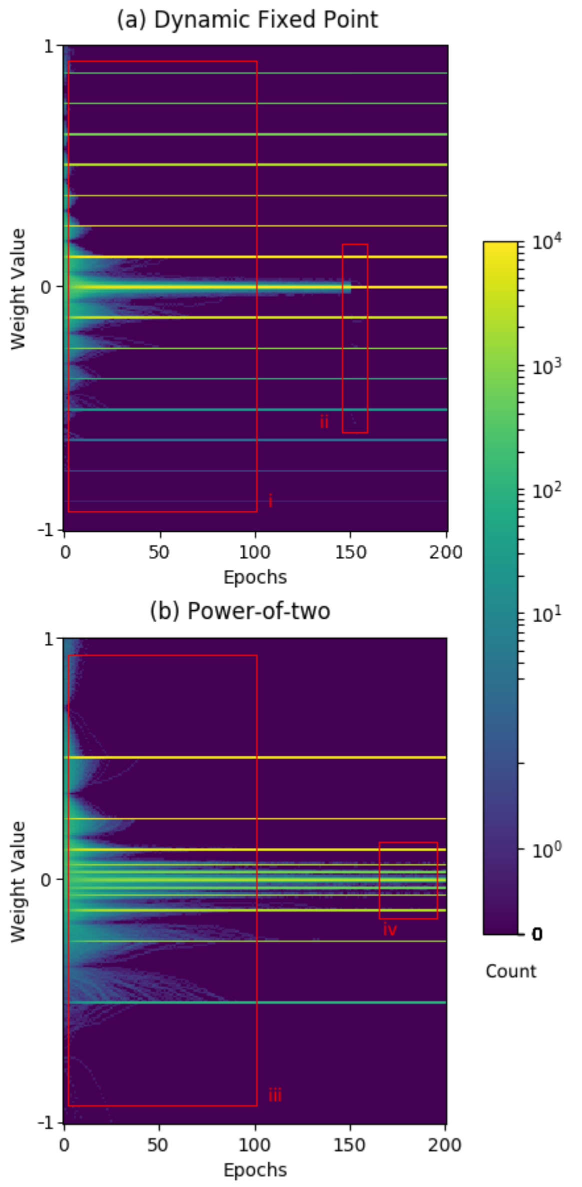


Fig. 9. Distribution of weights in layer 5 of AllConvNet (CIFAR-10) during fine-tuning with WQR and QR to (a) 4-bit Dynamic Fixed Point and (b) 4-bit Po2. DFP (a) is trained with $\lambda_2 = \mathbf{epoch} * \mathbf{10}$. Region i shows the faster quantization of weights with high magnitudes and slower quantization of weights close to 0. At epoch 150 QR is applied with $\lambda_1 = \mathbf{100}$ to quantize also weights with smaller values (ii). Po2 (b) is trained with $\lambda_2 = \mathbf{epoch} * \mathbf{10}$. Region iii also shows slower quantization for small magnitude weights. Due to the distribution of quantization levels, weights around 0 are already close to the next quantization level (iv).

precision, as they usually have the fewest parameters. In comparison to DFP, for Po2 quantization, layer-wise precision scaling does not proof to be as effective, as almost the same bit-width is recommended throughout the network to achieve best accuracy.

For fine-tuning of Po2 and DFP quantized networks, QR and WQR can be added to the loss function as regularization terms to force quantization during retraining. While the QR term forces all weights equally to reduce the distance to the next quantization level, the WQR term is reduced for weights with small magnitudes. Therefore, WQR operates less restrictive than QR, as the QR value is multiplied with a factor from 0 to 1 (normalized weight magnitude). As a consequence it is an effective strategy to apply WQR followed by QR fine-tuning.

Figure 9 shows the distribution of weights in layer 5 during (a) 4-bit DFP and (b) 4-bit Po2 quantization of the All-Convolutional Network for the CIFAR-10 dataset. For fine-tuning to 4-bit DFP quantization (Fig.9a) λ_2 is increased linearly by a factor of 10 with each epoch. As QR scale factor λ_1 we use 0 before and 100 starting at epoch 150. Region i in figure 9a shows the decelerated quantization of small magnitude weights. Starting with epoch 150, also weights close to 0 are quantized, due to the application of QR (Fig. 9a,ii). For 4-bit Po2 quantization only WQR is necessary ($\lambda_2 = epoch * 10$) to achieve quantization. Due to the the non-equidistant distribution of quantization levels in Po2 quantization, the weights with high magnitudes take longer than in DFP quantization to reach the quantization levels (Fig. 9b,iii). Additional QR is not necessary since, due to the high density of quantization levels close to 0, the weights with smaller magnitudes induce a very small quantization error (Fig. 9b,iv).

QR and WQR enable trained quantization to improve performance in comparison to direct quantization. Regularization-based quantization is very simple to implement and applicable for any quantization scheme. As a consequence QR and WQR could be employed alongside other effective quantization techniques such as stochastic quantization. Besides the simplicity of the approach, it also allows a deeper analysis of quantization schemes by recording the weight distribution during training.

While enabling compression for more efficient inference, during training QR and WQR add overhead due to the mandatory quantization step after each mini-batch and the necessary computations for calculation of QR and WQR. The fact that all weights have to be stored as floating point values and quantized values, increases the weight memory during training by a factor of 2.

IV. Experimental Results

The following section describes the experimental results for direct and trained quantization of All-CNNs on three datasets.

A. *Experimental Setup*

For experimental evaluation, we apply the proposed methods on All-Convolutional Networks for the Datasets CIFAR-10 [26], CIFAR-100 and SVHN [27]. We use All-Convolution Network model All-CNN-C from [24] for evaluation. The CNN Architecture summed up in table III consists of nine convolution layers and a global average pooling layer. Due to the similar filter-width and height the number of weights in the convolution layers mainly depends on the filter-depths. The number of operations also depends on the layer-output dimensions. However, it needs to be noted that the proposed technique is neither architecture nor dataset bound.

TABLE III
ALL-CNN-C ARCHITECTURE AND THE NUMBER OF WEIGHTS AND MAC-OPERATIONS FOR ONE FORWARD COMPUTATION WITH BATCH-SIZE ONE.

Layer (WxH)	Output Dim. (WxHxD)	#Weights	#MACs
Input	32x32x3	-	-
Conv. 3x3	32x32x96	2K	2.3M
Conv. 3x3	32x32x96	83K	74.6M
Conv. 3x3	32x32x96	83K	74.6M
Pooling 2x2	16x16x96	-	-
Conv. 3x3	16x16x192	165K	32.5M
Conv. 3x3	16x16x192	332K	65M
Conv. 3x3	16x16x192	332K	65M
Pooling 2x2	8x8x192	-	-
Conv. 3x3	8x8x192	332K	11.9M
Conv. 1x1	8x8x192	37K	2.4M
Conv. 1x1	8x8x10	2K	0.1M
Pooling 8x8	10	-	-
Total	-	**1386K**	**329.7M**

For the three datasets, we use the predefined training and test sets. For the floating point baselines we trained the CNN for 350 epochs with initial learning rate 10^{-3} multiplied by a fixed multiplier after epochs 200 and 300. To avoid overfitting, we use dropout with dropout rate 0.5 after layers. In contrast to the original All-CNN paper [24], the models are not regularized by weight decay to avoid interfering with the studied regularization methods. In terms of data augmentation we only apply horizontal flipping and random shifting by a maximum of 3 pixels.

B. *Performance Analysis*

We evaluate the performance of the proposed method comparing the test accuracy of the original network with the resulting accuracies after direct and trained quantization. For the tables V, IV and VI, we make use of the abbreviations in table II. We aim to achieve high compression rates in terms of weight memory while maintaining high test accuracy. In addition to bit-width reduction we also consider the resulting sparsity as an important factor for possible further compression. Assuming skipping of multiplications with 0 weights, sparsity also reduces the amount of required MAC-operations for forward computation[3]. For all experiments except the two marked, bit-width for activations is 32-bit Fixed Point.

1) CIFAR-100: CIFAR-100 is an image classification dataset consisting of a training set of 50000 and a test set of 10000 32×32 color images representing 100 different categories such as airplanes, automobiles, birds, cats, deers,dogs, frogs, horses, ships and trucks [26]. The training batches contain exactly 5000 images from each class. Table V shows the resulting accuracies after fine-tuning with 200 epochs of WQR with $\lambda_2 = epoch \times 10$. In addition λ_1 is set to 100 starting at epoch 150. This setup is not optimal for all configurations, as in some cases training with only QR would be sufficient, but it allows using the same parameterization for each iteration increasing comparability. Compared to a full implementation (32-bit), the proposed layer-wise quantization (DFP-lw) saves ∼4-8× in terms of weight memory and has an accuracy loss by ∼0.1-4.5%. Compared to the reduced equal bit-width quantization, the proposed layer-wise quantization has higher sparsity and smaller weight memory.

In addition, figure 10 depicts the results of trained quantization in comparison to direct quantization. We can see that retraining with QR and WQR in all cases increases classification accuracy in comparison to direct quantization.

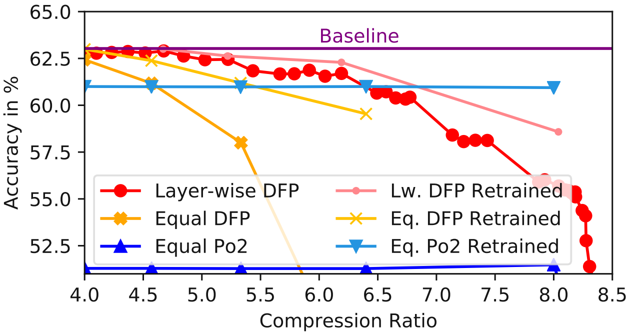


Fig. 10. Results for direct and trained DFP and Po2 quantization with equal bit-widths on ALL-CNN for CIFAR-100. For DFP also Layer-wise Precision Scaling with direct and trained quantization is illustrated.

Comparing equal bit-width quantization with Layer-wise Precision Scaling for DFP datatype, we can see that for similar compression ratios, equal bit-width (DFP-eq) never reaches the accuracy of Layer-wise Precision Scaling (DFP-lw), even when retraining with WQR/QR is applied. For Po2 quantization we found equal 4 bit quantization (Po2-eq) the most effective method as higher bit-widths did not increase accuracy and Layer-wise Precision Scaling for lower than 4 bit leads to drastic accuracy drop.

2) CIFAR-10: CIFAR-10 is a benchmark image classification dataset equal to CIFAR-100 in terms of image and dataset sizes, which instead of 100 classes divides the images into 10 classes. We use the same training method as for CIFAR-100. The results for trained quantization of All-CNN for CIFAR-10 are shown in table V. To allow comparing to other works for the two marked configurations, we also quantized the activations to 8-bit DFP. In contrast to ALL-CNN for CIFAR-100, for this dataset higher compression rates can be achieved. For Compression Ratio ∼ 8, DFP with Layer-wise Precision Scaling (DFP-lw) gives lowest accuracy degradation of 0.51 percentage points. For equal bit-with quantization Po2 outperforms DFP by 1.55 percentage points. For Compression Ratio 7.38 classification accuracy of DFP with Layer-wise Precision Scaling is only 0.01 percentage points below the floating point baseline.

3) SVHN: The SVHN image classification dataset consists of 694K 32×32 color images for training and 26K images for testing. The images represent digits form 0 to 9. Similarly to CIFAR-100 and CIFAR-10 we perform Layer-wise Precision Scaling and retraining with WQR and QR for model compression. The results are shown in table VI and figure 11. For the SVHN dataset at Compression Ratio ∼ 8, Po2 quantization

[3] The number of skipped MACs due to a pruned weight depends on the layer-input dimensions. Therefore sparsity in weights is unequal to sparsity in MACs.

TABLE IV
RESULTS IN COMPARISON TO FLOATING POINT BASELINE OF ALL-CNN FOR CIFAR-100 (CR=COMPRESSION RATIO)

b_n	Type	$W_{mem}[Bit]$	CR	Sparsity[%]	non-0 MACs	MAC Sparsity[%]	acc_{tr}[%]	acc_{init}[%]	acc_{M_q}[%]
32 bit	float	44344K	1	0	329.7M	0	83.24	63.03	**63.03**
8 bit	DFP-eq	11086K	4	5.8	304.8M	7.6	83.5	62.43	**62.99**
7 bit	DFP-eq	9700K	4.6	11.5	280.3M	15	82.65	61.86	**62.38**
6 bit	DFP-eq	8314K	5.3	21.9	234.1M	29	81.02	58	**61.19**
5 bit	DFP-eq	6928K	6.4	38.9	161M	51.2	76.99	43.72	**59.54**
[9 9 9 9 6 5 7 9 9]	DFP-lw	9485K	4.7	14.6	289.5M	12.2	83.44	62.90	**62.93**
[9 9 9 9 5 5 5 9 9]	DFP-lw	8490K	5.2	23.2	279.2M	15.3	82.45	62.45	**62.64**
[9 9 9 5 4 4 5 9 9]	DFP-lw	7163K	6.2	36.4	243.4M	26.2	80.76	61.70	**62.29**
[9 6 5 5 3 3 4 7 9]	DFP-lw	5513K	8.0	60.4	133.6M	59.5	73.76	55.69	**58.58**
4 bit	Po2-eq	5543K	8	18.3	255.2M	22.6	79.47	51.27	**60.94**

TABLE V
RESULTS IN COMPARISON TO FLOATING POINT BASELINE OF ALL-CNN FOR CIFAR-10

b_n	Type	W_{mem}[Bit]	CR	Sparsity[%]	non-0 MACs	MAC Sparsity[%]	acc_{tr}[%]	acc_{init}[%]	acc_{M_q}[%]
32 bit	float	43791K	1	0	329.7M	0	96.69	90.83	**90.83**
8 bit	DFP-eq[4]	10947K	4	4.2	303.5M	7.6	96.64	90.65	**90.85**
4 bit	DFP-eq	5474K	8	42.3	140.7M	57.2	94.4	44.47	**88.63**
[8 8 8 5 4 4 3 8 8]	DFP-lw[4]	5929K	7.38	36.8	243M	26	96.49	90.14	**90.81**
[7 7 7 4 4 3 3 7 7]	DFP-lw	5432K	8.1	46.1	203.9M	37.9	96.01	90.07	**90.32**
4 bit	Po2-eq	5474K	8	13.2	255.9M	22.1	96.13	76.73	**90.18**

TABLE VI
RESULTS IN COMPARISON TO FLOATING POINT BASELINE OF ALL-CNN FOR SVHN

b_n	Type	W_{mem}[Bit]	CR	Sparsity[%]	non-0 MACs	MAC Sparsity[%]	acc_{tr}[%]	acc_{init}[%]	acc_{M_q}[%]
32 bit	float	43791K	1	0	329.7M	0	97.70	95.84	**95.84**
4 bit	DFP-eq	5474K	8	43.5	166.8M	49.2	96.27	86.17	**95.37**
[6 4 4 3 3 3 4 5 6]	DFP-lw	4690K	9.3	62.7	126.6M	63.2	96.43	95.03	**95.89**
[5 4 4 3 3 3 3 3 3]	DFP-lw	4276K	10.23	70.5	116.7M	64.4	95.52	89.86	**95.36**
4 bit	Po2-eq	5474K	8	17.0	272.8M	16.9	97.02	91.93	**96.02**

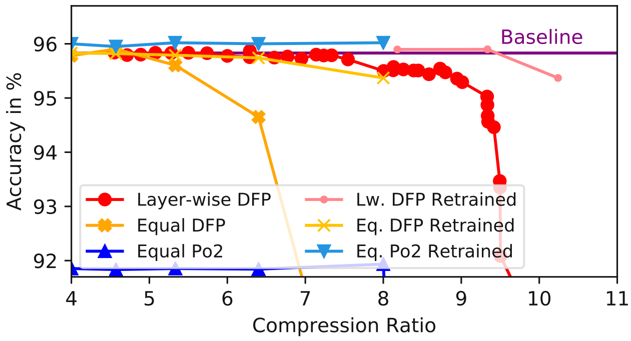


Fig. 11. Results for direct and trained DFP and Po2 quantization with equal bit-widths on ALL-CNN for SVHN. For DFP also Layer-wise Precision Scaling with direct and trained quantization is illustrated.

and DFP with Layer-wise Precision Scaling both outperform the original baseline model.

Comparing the results for the datasets CIFAR-100, CIFAR-10 and SVHN, we can conclude that the attainable Compression Ratio for lossless trained quantization with QR/WQR not only depends on the selected datatype and bit-widths, but also on the selected datasets. While for CIFAR-10 and SVHN, lossless trained quantization achieves Compression Ratio 8.0 and 9.4 respectively, for CIFAR-100 the maximum Compression Ratio is 4.7 for lossless compression. In the case of CIFAR-100 further increasing Compression Ratio up to 8.0 reduces accuracy by 2.09 percentage points. In contrast to this trade-off, stronger compression for CIFAR-10 and SVHN immediately leads to drastic accuracy reduction. We suspect that this difference can be explained by the relation between complexity of the dataset and the selected network. To better understand this relation, in future the techniques have to be applied for further datasets and network topologies.

While with DFP lossless compression can always be achieved, Po2 can sometimes lead to slight performance degradation. Layer-wise Precision Scaling turns out to be more effective for DFP than for Po2. Po2 still reaches maximum accuracy at uniform 4-bit bit-width while achieving higher accuracy than uniform 4-bit and even 5-bit DFP.

V. COMPARISON WITH RELATED WORK

The proposed method of Weighted Quantization-Regularization presents a novel technique for trained quantization of DNNs. However in some works performing weight-binarization similar regularization methods are used to achieve weights with values ‘+1’ or ‘−1’ [28]. Today trained quantization is mostly performed by stochastic rounding during training [14], [29], [22], [30]. Gupta et al. [30] apply stochastic training for CIFAR-10 dataset to achieve Fixed Point quantization to 16-bit and 12-bit. Their accuracy

[4] 8-bit DFP for activations

is reduced by 0.8 and 4.2 percentage points respectively compared to the floating point baseline. In comparison to that with our method we reach 8-bit DFP quantization without any performance drop.

Courbariaux et al. [14] perform quantization based on stochastic round for 10-bit Dynamic Fixed Point weights and activations. Their accuracy drops in comparison to the baseline networks 3.14 percentage points for CIFAR-10 and 2.58 percentage points for SVHN since in contrast to us, they also perform weight-update with 12-bit DFP which decreases comparability. For our CIFAR-10 and SVHN network we experimentally also applied 8-bit DFP for weights and activations, and found that accuracy even increased after QR/WQR retraining. Gysel et al. [22] use their CAFFE-based tool Ristretto for Layer-wise Precision Scaling and fine-tuning with stochastic rounding. On CIFAR-10 their accuracy lies 0.3% below the floating point baseline accuracy, when quantizing not only weights but also activations to 8 bit DFP. By applying Layer-wise Precision Scaling we are able to increase compression ratio from 4 to 7.38 and after QR/WQR-retraining achieve equal to baseline accuracy while inducing higher sparsity due to the stronger compression. Other than fine-tuning with stochastic rounding, Zhou et al. [11] presented an incremental retraining method to perform Power-of-two weight quantization. They achieve lossless 5-bit/4-bit quantization for several DNNs for the ImageNet dataset. Even for lower bit-rates incremental quantization achieves state-of-the-art results. Even though this method seems highly promising, in contrast to our work, it is only verified to work for power-of-two quantization.

VI. Conclusion

We propose Quantization-Regularization (QR) and Weighted QR (WQR) as techniques for improving accuracy after bit-width reductions inflicted by a quantization scheme. WQR/QR allow fine-tuning for weights in any quantization scheme and also works for non-uniform bit-widths (Layer-wise Precision Scaling). For ALL-CNN with the CIFAR-10 benchmark WQR reaches lossless compression up to a ratio of 7.38x with DFP and Layer-wise Precision Scaling. Compared to the 32bit floating point baseline in the All-CNN network, WQR with DFP obtains a weight memory compaction of 8.0×-10.23× and a MAC sparsity of 37.9%-64.4% in the benchmark tasks. For these cases with maximal compaction we observe between 0.48-4.45 percentage points reduction of classification accuracy. A high MAC sparsity benefits HW implementations because it potentially reduces the number of multiply-accumulate operations.

Note, that WQR/QR is not a stand-alone tool, but can be combined with other techniques and is typically complementary to those. We have observed, that it has some limitations when applied to very low bit-widths; hence, its combination with stochastic rounding methods is considered as future work.

Furthermore, we have studied two quantization schemes, Dynamic Fixed Point (DFP) and Power-of-two (Po2), and we find that DFP is usually preferable to Po2 because it is as good as or better than Po2 in most cases, it can reach floating point accuracy when increasing bit-width, and it does not necessarily require retraining (retraining improves accuracy but for Po2 it is absolutely necessary). However, in a few special cases with low bit-width Po2 is slightly better and it might be preferred for HW implementations because it requires only shift operations instead of multiplications. Thus, when an optimized HW implementation is developed, Po2 could be considered as a useful option.

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[2] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255.

[3] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf, "Deepface: Closing the Gap to Human-Level Performance in Face Verification," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2014, pp. 1701–1708.

[4] P. Rajpurkar, A. Y. Hannun, M. Haghpanahi, C. Bourn, and A. Y. Ng, "Cardiologist-level Arrhythmia Detection with Convolutional Neural Networks," *arXiv preprint arXiv:1707.01836*, 2017.

[5] M. Wess, P. D. S. Manoj, and A. Jantsch, "Neural network based ECG anomaly detection on FPGA and trade-off analysis," in *Proceedings of the IEEE International Symposium on Circuits and Systems*, 2017, pp. 1–4.

[6] J. Zhang and C. Zong, "Deep Neural Networks in Machine Translation: An Overview," *IEEE Intelligent Systems*, vol. 30, no. 5, pp. 16–25, 2015.

[7] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, 2016, pp. 770–778.

[8] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2015, pp. 161–170.

[9] P. D. S. Manoj, J. Lin, S. Zhu, Y. Yin, X. Liu, X. Huang, C. Song, W. Zhang, M. Yan, Z. Yu, and H. Yu, "A scalable network-on-chip microprocessor with 2.5D integrated memory and accelerator," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 64, no. 6, pp. 1432–1443, June 2017, https://ieeexplore.ieee.org/document/7819521/.

[10] S. Han, H. Mao, and W. J. Dally, "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding," in *International Conference on Learning Representations*, 2016.

[11] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen, "Incremental Network Quantization: Towards Lossless CNNs with Low-Precision Weights," *International Conference on Learning Representations*, 2017.

[12] X. Chen, X. Hu, H. Zhou, and N. Xu, "FxpNet: Training a Deep Convolutional Neural Network in Fixed-Point Representation," in *International Joint Conference on Neural Networks*. IEEE, 2017, pp. 2494–2501.

[13] D. D. Lin, S. S. Talathi, and V. S. Annapureddy, "Fixed Point Quantization of Deep Convolutional Networks," in *Proceedings of the International Conference on Machine Learning*, 2016, pp. 2849–2858.

[14] M. Courbariaux, Y. Bengio, and J.-P. David, "Training deep neural networks with low precision multiplications," *arXiv preprint arXiv:1412.7024*, 2014.

[15] X. Dong, S. Chen, and S. Pan, "Learning to Prune Deep Neural Networks via Layer-wise Optimal Brain Surgeon," in *Advances in Neural Information Processing Systems*, 2017, pp. 4860–4874.

[16] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both Weights and Connections for Efficient Neural Networks," in *Advances in Neural Information Processing Systems*, 2015, pp. 1135–1143.

[17] T.-J. Yang, Y.-H. Chen, and V. Sze, "Designing Energy-Efficient Convolutional Neural Networks using Energy-Aware Pruning," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

[18] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing DNN Pruning to the Underlying Hardware Parallelism," in *Proceedings of the Annual International Symposium on Computer Architecture*. ACM, 2017, pp. 548–560.

[19] P. Gysel, "Ristretto: Hardware-Oriented Approximation of Convolutional Neural Networks," Master's thesis, University of California Davis, 2016.

[20] Y. Zhou, S.-M. Moosavi-Dezfooli, N.-M. Cheung, and P. Frossard, "Adaptive Quantization for Deep Neural Network," *AAAI Conference on Artificial Intelligence*, 2018.

[21] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient Inference Engine on Compressed Deep Neural Network," in *International Symposium on Computer Architecture*. IEEE, 2016, pp. 243–254.

[22] P. Gysel, M. Motamedi, and S. Ghiasi, "Hardware-oriented Approximation of Convolutional Neural Networks," *arXiv preprint arXiv:1604.03168*, 2016.

[23] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations," *Journal of Machine Learning Research*, 2018.

[24] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller, "Striving for Simplicity: The All Convolutional Net," in *International Conference on Learning Representations*, 2015.

[25] S. Shin, Y. Boo, and W. Sung, "Fixed-point Optimization of Deep Neural Networks with Adaptive Step Size Retraining," *IEEE International Conference on Acoustics, Speech, and Signal Processing*, 2017.

[26] A. Krizhevsky, V. Nair, and G. Hinton. (2009, mar) Cifar-10 and Cifar-100 Datasets. [Online]. Available: https://www.cs.toronto.edu/ kriz/cifar.html

[27] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng, "Reading Digits in Natural Images with Unsupervised Feature Learning," in *NIPS workshop on Deep Learning and Unsupervised Feature Learning*, vol. 2011, no. 2, 2011, p. 5.

[28] W. Tang, G. Hua, and L. Wang, "How to Train a Compact Binary Neural Network with High Accuracy?" in *AAAI Conference on Artificial Intelligence*, 2017, pp. 2625–2631.

[29] M. Courbariaux, Y. Bengio, and J.-P. David, "BinaryConnect: Training Deep Neural Networks with Binary Weights during Propagations," in *Advances in Neural Information Processing Systems*, 2015, pp. 3123–3131.

[30] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep Learning with Limited Numerical Precision," in *International Conference on Machine Learning*, 2015.

Matthias Wess recieved the BSc and MSc degrees from Department of Electrical Engineering, TU Wien, Vienna, Austria in 2013 and 2017, respectively, where he is currently pursuing the PhD degree with the Institute for Computer Technology.

His current research interests include hardware acceleration of deep neural network inference.

Sai Manoj P. D. is a Research Assistant Professor at George Mason University (GMU), Fairfax, VA, United States. Prior to joining GMU, Dr. Manoj worked as a post-doctoral research fellow at TU Wien, Austria. He received his PhD in Electrical and Electronic Engineering from Nanyang Technological University, Singapore, in 2015. His research interests include Adversarial learning, digital design for machine learning, cyber-security for embedded processors, self-aware SoC design, machine learning for on-chip data processing, and security in IoT networks. He is a recipient of 'A. Richard Newton Research Young Research Fellow Award' in DAC 2013. He is a Member of the IEEE.

Axel Jantsch Axel Jantsch is Professor of Systems on Chip at the Institute of Computer Technology at TU Wien, Austria. His research interests include embedded machine learning and self-awareness in SoCs and embedded systems. Jantsch has a PhD in computer science from TU Wien. He is a Member of the IEEE.