Development and training of a deep learning approach to estimate latency of deep neural network inference

Lilit Beglaryan
Akian College of Science & Engineering
American University of Armenia (AUA)
Yerevan, Armenia
lilit_beglaryan@edu.aua.am

Christopher Ringhofer
Embedded Systems Group
University of Duisburg-Essen
Duisburg-Essen, Germany
christopher.rinhghofer@uni-due.de

***Abstract*— This academic paper introduces a deep learning estimator model designed to predict the latency of the inference stage in fully connected deep neural networks (DNNs) running on laptop CPUs. As the size and computational requirements of DNNs continue to increase over time, it becomes crucial to estimate their performance measures in order to optimize them effectively. The primary objective of this paper is to estimate one of the key performance measures of DNNs, which is latency.**

This research and the proposed prediction model contribute to the broader field of optimizing modern neural network architectures, specifically in the context of hardware-aware neural architecture search (HW-NAS). The paper draws comparisons between existing latency prediction methodologies, focusing on those that are relevant for integration into HW-NAS algorithms based on their exhibited results. This work represents a small step towards addressing the urgent issue of latency optimization.

This research achieved strong performance metrics and loss values (MSE=0.0007,MAE=0.0211,RMSE=0.0261). The model showed a small average absolute deviation of latency (0.02) between predicted and actual latency values. Experiments focused on estimating latencies of DNN architectures with up to 5 layers with ReLU activation (up to 50 neurons per layer), limiting generalizability to similar setups. The study exclusively targeted laptop CPUs, restricting applicability to similar hardware platforms.

Overall, this paper contributes to the field by offering a deep learning estimator model for predicting latency in fully connected DNNs running on laptop CPUs, and by discussing its implications within the context of HW-NAS.

Keywords—latency estimation, deep FCNN, HW-NAS (NAS), FLOPs, Prediction model, Analytical model, Real-time measurement, Look-Up table, GCN, CPU, GPU, Gradient Boosting, MSE, ReLU, RMSE, MAE

I. PAPER STRUCTURE

To begin, the paper introduces the topic of latency and its significance in optimizing modern neural networks. It emphasizes the importance of accurately predicting inference or execution latency in various platforms.

Next, the paper delves into an examination of existing approaches for estimating latency. It explores different methods and techniques employed to predict inference or execution

Lilit Beglaryan, American University of Armenia, ©2023

time, shedding light on their applications as well as their limitations.

Following the exploration of existing approaches, the paper introduces the proposed approach in detail. It provides a comprehensive description of the design and implementation of the latency estimator prediction model. This section of the paper highlights the unique aspects of the proposed approach and how it addresses the limitations of previous methods.

Finally, the paper presents the results obtained from implementing the proposed approach. It showcases the performance and effectiveness of the latency estimator model, demonstrating its ability to accurately predict inference latency on CPU-based platforms.

Overall, the paper follows a structured flow, starting with an introduction to the topic, reviewing existing approaches, introducing the proposed approach, and concluding with the results obtained. This approach allows readers to gain a thorough understanding of the research context, the proposed model, and its potential impact in optimizing neural networks by predicting latency.

II. INTRODUCTION

In recent decades, with the growth of available data, complex technologies, and new business demands, Machine Learning (ML) models have been growing in their sizes, and complexities, and started consuming more resources such as inference latency, training or even development time, hardware (HW), such as memory (RAM), storage (HDD/SSD), processing units (CPU (Central Processing Unit), GPU (Graphics Processing Unit), TPU (Tensor Processing Unit),etc.), electricity **[1]**. Historically, ML engineers and researchers have focused more on increasing the prediction performance (for example, accuracy, precision, recall) of models ignoring their inefficient consumption of computing power and required time. The fact that these models are trained and deployed multiple times makes these inefficiencies more troublesome. The natural urge to make the ML models more efficient and scalable in these resource-constrained devices has led to different approaches, for instance, quantization and pruning as well as special network architectures optimized for efficiency.

One approach to design such optimized architectures is called HW-NAS (HardWare-aware Neural Architecture Search) **[2]**. Since the previous way of manually improving models was time-consuming and not so efficient, HW-NAS, appearing in 2017, came up with "state of the art (SOTA) results in resource-constrained environments" **[2, p. 2]**. HW-NAS automates the neural network (NN) design process to achieve optimality from the very beginning of the model construction, making trade-offs between the above-mentioned resource constraints, while enabling intelligent computations and functionalities of complex neural nets on average resource-constrained devices [1].

The concept of HW-NAS aims to optimize neural network architectures based on a predefined cost function. To construct such a cost function, objective cost values are required as samples, which can then be used to develop a search strategy [2]. One possible approach is to utilize example cost measurements to train a HW cost estimator. Consequently, the herewith proposed latency estimator could be integrated into HW-NAS methodologies to estimate and enhance the latencies of the targeted architectures.

III. PROBLEM STATEMENT

A. *What is the latency measurement?*

Latency refers to the real-time performance measurement of an ML model, representing the time required to process a single unit of data under the assumption that data processing occurs within a single time unit. Typically, latency is measured in seconds. It is important to reduce the latency due to its impact on how fast the ML model performs in real-time [3].

Achieving improved latency is a challenging task that necessitates a deep understanding of both the specific application and the target ML model. Additionally, it depends on the underlying system on which the ML model executes, such as the CPU/GPU/FPGA (Field Programmable Gate Arrays), the number of cores, processing units, and the memory's hierarchical structure. Moreover, the choice of ML frameworks

utilized during the development and deployment of the model also impacts latency [4]. Latency is not only hardware-dependent but also depends on the data fed to the model [2].

Directly measuring the inference time for each candidate architecture in the large search space of NAS can be extremely time-consuming. As a result, utilizing latency estimators or proxies is a more efficient approach. However, selecting an appropriate measure to serve as a proxy for latency is a critical consideration. Some approaches opt for measuring the speed observed on the target device during inference, while others prefer to rely on the number of FLOPs (floating-point operations per second), although with the advancement of HW technologies (multithreading, cashing, optimization, parallelism, etc.) FLOPs are no more representative of the latency measure [5, p. 3].

In Fig. 1, you can see the curves of latency estimations taking FLOPs as a proxy measurement of the same type of convolution operators (1x1) on two types of HW platforms. While with the increase of the number of input and output channels of CNNs (Convolutional Neural Network) the number of FLOPs does increase, however FLOPs cannot account for other critical factors that impact latency. It is evident that the surfaces depicted in the Fig. 1 do not exhibit any common patterns, which allows us to conclude that FLOPs "may not capture desired hardware characteristics" [6, p. 5]. Consequently, relying solely on FLOPs as a measure to estimate the latency of a model may not be reliable.

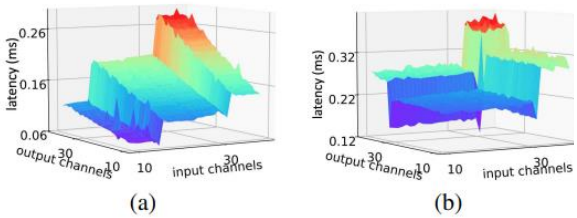

Figure 1: Latency vs. #Channels for a 1×1 convolution on an input image size of 56×56 and stride 1 on (a) Snapdragon 835 CPU and (b) Hexagon v62 DSP. Red (blue) color indicates high (low) latency. *[6]*

B. *Targetted ANN (Artificial Neural Network) architectures in this study*

In general, throughout history, there have been numerous efforts to optimize the latency of various types of NN architectures, targeting different hardware platforms and specifications. Examples include optimizing for mobile GPUs [6] and desktop CPUs, TPUs (Tensor Processing Units), and DSPs (Digital Signal Processors) [4, p. 2].

Specifically, the focus of this work is to introduce a latency estimator for fully connected neural networks (FCNNs) during the inference stage, with a specific emphasis on the laptop CPU as the target HW platform. The goal is to develop an estimation approach that accurately predicts the latency of FCNN models when executed on laptop CPUs. By understanding and estimating the latency, developers and researchers can make informed decisions about the performance of FCNNs on CPU-based systems.

FCNNs are a specific type of artificial neural network (ANN) in which every neuron in each layer is connected to all the neurons in the subsequent layer. Deep FCNNs possess additional complexity. They consist of at least two layers and are often referred to as deep nets. Unlike traditional ANNs, deep neural networks (DNNs) incorporate multiple hidden layers, which offer advantages in terms of model performance but also introduce additional complexity and latency overhead.

C. *What is the objective of this project?*

The objective of this project is to develop and train a deep FCNN latency estimator, which in its turn will take as input encoded architectures of other deep FCNNs in a predefined format and generate estimated latencies for those during the inference stage (not training) on a local CPU. The DNNs for which the latencies will be estimated will accept one-dimensional sequential data as input (as a proxy for time series data).

IV. Related Research

In this chapter, we will explore and discuss existing methodologies that address the problem of estimating the latency of various types of neural networks running on different

HW technologies. These methodologies have been developed to provide insights into the performance and efficiency of neural networks, enabling researchers and practitioners to make informed decisions and optimizations.

A. *ANNETTE: Accurate Neural Network Execution Time Estimation with Stacked Models*

The research paper [7], authored by a team of six individuals from Vienna University of Technology (TU Wien), introduces a novel framework called ANNETTE (Accurate Neural Network Execution Time Estimation). This framework focuses on modeling the execution latency of different type of Deep Neural Networks (DNNs),including CNNs, FCNNs and RNNs (Recurrent Neural Network), encompassing both the forward and backward passes involved in training and inference processes. ANNETTE specifically targets hardware accelerators and aims to provide precise estimations of DNN execution time on previously unseen architectures.

The research emphasizes the importance of addressing the disparity between resource intensive DNNs, specifically convolutional neural networks (CNNs), and the limited computing resources available on embedded platforms and mobile devices. In order to achieve this, the ANNETTE method introduces embedded estimation models that are specifically designed for inference on resource-constrained devices [7].

These embedded estimation models are developed and evaluated using the DNN Development Kit (DNNDK) provided by Xilinx. DNNDK is a development framework that is based on FPGAs, allowing for efficient deployment and optimization of DNN models. By utilizing DNNDK and FPGAs, ANNETTE aims to provide accurate and efficient latency estimation for CNNs on embedded platforms, enabling effective model selection and optimization in resource-constrained environments [7].

Extensive research efforts have been dedicated to CNN models under resource constraints, such as those found in embedded platforms and mobile devices. It is important to recognize that relying solely on the target hardware platform for latency estimations may lead to inaccurate results. This is because latency is influenced not only by the hardware specifications but also by other factors such as the network architecture, optimization techniques, and input data characteristics. Modern hardware accelerators employ various optimization techniques, such as processing multiple neural layers simultaneously to enhance latency and data flow between layers. This implies that relying solely on indirect measurements like FLOPs (Floating Point Operations) or memory footprint is insufficient, as they fail to account for platform-specific complexities and lack cross-platform independence. Furthermore, variations in the clock speed of target devices, like the CPU clock frequency, can lead to varying latencies even for the same model on identical hardware platforms. Hence, the approach of dividing the number of FLOPs by the peak compute performance of the target device to estimate execution time is not dependable.

To summarize, although efforts are being made to optimize DNNs in order to overcome computational limitations, relying solely on hardware-specific factors or indirect measurements can result in unreliable estimations. It is essential to consider platform-specific non-linearities and employ a comprehensive approach that extends beyond proxy measurements to achieve accurate estimation of execution time [7].

The "ANNETTE" research paper highlights that the current research emphasis is primarily on estimating latency for CPUs and GPUs, while overlooking the significance of FPGAs, which are extensively used in AI tasks. That is why it focuses on estimating latencies of CNNs targeted for FPGAs. It discusses various latency estimation methods targeting CPUs, predominantly using linear regression models. These models demonstrate high average accuracies ranging from 84% to 96%. This finding suggests that latency estimation on

optimized hardware accelerators poses a more significant challenge compared to CPUs [7][1].

To address this challenge, the authors propose ANNETTE, a combination of stacked models for accurate latency estimation. The process involves abstract model generation, benchmarking, and estimation using optimized graphs, enabling latency estimation without executing the network itself.

The process begins with the Benchmark tool, which generates abstract models on the target hardware. It executes the models and extracts layer-wise execution times using a hardware-specific benchmark application (though the specific tool is not mentioned). Micro-kernel and multi-layer benchmarks are utilized to assess the performance of individual layers and mixed ("fused") layers, respectively. The micro-kernel benchmarks cover 2D convolution, 2D depth-wise separable convolution, and fully connected layers, considering parameters such as input channels, filters, number of input/output neurons, and kernel sizes.

Following this, the Model Generator creates optimized graphs representing abstract models. Finally, the Estimator tool estimates the latency of a network without the need for compilation and execution. It utilizes the benchmarks obtained from the Benchmark tool as a reference.

Regarding the obtained results, as the researchers state in their work, they "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" [7, p. 1].

B.1 Already existing latency estimation models used in ANNETTE

Analytical & Statistical Models: For simple models, the estimator uses analytical approach which for each layer n sets the smallest achievable execution time to be between the peak computational performance P_{peak} and the maximum bandwidth B_{peak} in layer n having D_n data to be transferred and number of operations f_n to be performed the estimated execution time with the optimal computational performance equals to:

$$T(f_n, D_n) \approx \max\left(\frac{f_n}{D_n}, \frac{D_n}{B_{peak}}\right) \quad (1)$$

Additionally, the paper discusses the inclusion of more complex computations that consider the intricate configurations of each layer and hardware-specific procedures. Apart from the analytical estimation model, statistical regression models, like random forest regression, are employed to estimate the performance of various benchmarked layer types [7].

In conclusion, the final stage of estimation involves the use of mapping models. These models reconstruct an optimized version of the given neural network architecture using a platform mapping toolchain specific to the hardware. By leveraging both statistical and analytical models, the framework estimates the latency of each layer and aggregates them to obtain the total network execution time. The authors assert that ANNETTE can make valuable contributions to the field of hardware-aware neural architecture search (HW-NAS).

Key Takeaways: When discussing related methods, the research conducted on ANNETTE provided valuable insights into alternative approaches, such as the effectiveness of linear regression models for estimating latency on CPUs. Inspired by this, our study aims to adopt a similar approach for latency estimation. However, ANNETTE itself with its complexity, incorporated stacked models and

[1] We will see the truth of this statement in the Results section; our simple estimator model performed well on CPU-based platform

analytical calculations, and its focus on FPGAs, falls outside the scope of our research interest. Therefore, we have chosen a simpler approach for comparison purposes.

C. Other approaches of latency estimation

Look-Up Tables (LUTs) / Real-time measurements: The aforementioned paper on ANNETTE discusses additional approaches for estimating time, such as the utilization of Look-Up Tables (LUTs). These tables store latency information for specific building blocks of neural architectures within the search space of HW-NAS problems, tailored to specific hardware platforms. The tables capture latency simulations or measurements considering factors such as operation types and counts, layer configurations, layer dimensions, memory access information, and more. When estimating latency for a new neural network, the closest entry in terms of parameters within the look-up table is located, and the corresponding recorded measurement is outputted [7].

However, there are certain drawbacks to this approach. Firstly, in order to maintain the reliability and comprehensiveness of the LUTs, it is necessary to keep pace with rapid advancements in hardware and new variations of neural network architectures. This requires continuous measurement, validation, and recording of latencies for all possible combinations, necessitating regular updates to the tables. Secondly, the estimated measurements obtained directly from the table may not be suitable for direct output. Adjustments need to be made based on the parameter variations between the network being estimated and its closest model recorded in the LUT. The challenge lies in determining how and to what extent these adjustments should be applied.

On the other hand, LUT is often preferred over real-time measurements, which directly measure the exact latency of an NN architecture using Entry/Exit Systems (EES), making obtaining accurate latency measurements a slow and computationally expensive process. This is especially challenging when considering the vast number of devices on which an architecture needs to be executed and latency measured. To address this, LUTs can be employed to expedite the process.

C.1 Utilization of LUTs in Chameleon (ChamNet: Efficient NN design Framework)

In another article [6], the researchers introduce a framework called Chameleon, which facilitates optimized NN architecture design through platform-aware adaptations of models. One of the key components of this framework is the construction and utilization of an operator latency LUT specific to the target device. This involves benchmarking the latency of individual operators on various types of devices with different input sizes. By summing up the latencies of all operations involved in a specific NN execution, the total latency can be obtained [6].

While constructing the LUT may initially seem extensive and time-consuming, it is considered a "one-time cost" [4, p. 5] and can be reused for different NN models. Also, LUT approach is much faster than real-time measurements since it can be completed in “less than one CPU second” [6, p. 5], whereas real measurements can take minutes. The paper highlights that by keeping the LUT up to date and recording relevant information, the LUT can provide a reliable approximation of the real latency measure [4] (as depicted in Figure 2).

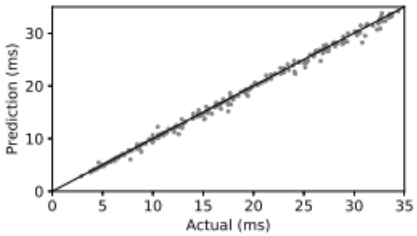

Figure 2: Latency predictor evaluation on Snapdragon 835 CPU.

D. Prediction-based NAS using GCNs (Graph Convolutional Network)

An alternative approach to estimating latencies of models from the DARTS search space is through the use of an end-to-end NAS latency predictor model based on Graph Convolutional Networks (GCN). This approach

is implemented in a HW- NAS tool called BRP-NAS designed for optimized neural network design search. According to the authors, the integration of the GCN latency predictor enhances the performance of BRP-NAS, surpassing previous estimator models on NAS-Bench-101 and NASBench-201 datasets [5, p. 1]. GCN estimator is good for estimating latencies of any type of NN on desktop GPUs (see in Fig. 3), as recognized in another research study on prediction on mobile GPUs(which are types of embedded GPUs). There also they state that LUTs exhibit high errors when estimating the latency in a layer-wise way on mobile GPUs while yield better latency approximations for CPUs [4]. The results, as shown in Figure 3, demonstrate the superior performance of the GCN latency estimator compared to proxy-based methods (e.g., FLOPs) or layer-wise approaches (e.g., ANNETTE), as discussed in the article.

Architecture: It is crucial to ensure a comprehensive understanding of the proposed estimator model and the training and testing processes, delving into the details of the model architecture and the methodologies employed in training and evaluation. This will provide valuable insights and guidance for conducting our own work.

The GCN latency predictor has 4 layers of GCNs, each having 600 hidden units. After GCNs comes a fully connected layer that outputs the continuous value of the latency [5].

The input to the estimator is an encoded graph representation (adjacency matrix) of the NN model which represents the connectivity of the architecture. The graph is defined as $G = (V, E)$, where V is a set of N nodes, and E is a set of edges. Each node in the graph corresponds to a component or layer of the neural network architecture. To train the GCN, it requires two main inputs: a feature description X and a description of the graph structure as an adjacency matrix A. The feature description X is a matrix of size $N \times D$, where N is the number of nodes and D represents the number of features associated with each node. These features can capture information about the nodes or components of the neural network, such as their type, size, or other relevant characteristics. The adjacency matrix A is a matrix of size $N \times N$ that represents the connectivity between the nodes in the graph. Each entry in the adjacency matrix indicates whether there is a connection (edge) between two nodes. If there is a connection between nodes i and j, the corresponding entry $A[i][j]$ will have a non-zero value, indicating the presence of an edge. The GCN operates in multiple layers and the layer-wise propagation rule is applied to update the node representations in the matrix A [4, p. 11].

The proposed GCN predictor was trained 100 times on random 900 samples from NAS-Bench-201 dataset, validated on 100 random NN architectures, tested on 14000 architectures [5, p. 5].

Error bound	Accuracy of GCN predictor [%]			Accuracy of Layer-wise predictor [%]		
	Desktop CPU	Desktop GPU	Embedded GPU	Desktop CPU	Desktop GPU	Embedded GPU
±1%	36.0±3.5	36.7±4.0	24.3±1.4	3.5±0.2	4.2±0.2	6.1±0.3
±5%	85.2±1.8	85.9±1.9	82.5±1.5	18.2±0.4	17.1±0.3	29.7±0.8
±10%	96.4±0.7	96.9±0.8	96.3±0.5	29.6±1.1	32.6±1.2	54.0±0.8

Figure 3: Performance of latency predictors on NAS-Bench-201 (GCN vs state-of-the-art NAS layer-wise predictor): The GCN predictor demonstrates significant improvement over the layer-wise predictor across devices. The values represent the corresponding percentages of models from NAS-Bench-201 search space for which the latency predictions were within respectively the 1%, 5% and 10% error bounds relative to the measured latency *[5, p. 5].*

Key Takeaways: In conclusion, the GCN estimator approach shows promise as it leverages the graph representation of the architecture, allowing for a more comprehensive consideration of the architecture design characteristics. However, it is important to note that the GCN estimator demonstrates better performance on GPUs rather than CPUs, making it less directly comparable to our work, which focuses on latency estimation specifically for laptop CPUs. Nonetheless, by understanding the strengths and limitations of the GCN estimator approach, we can gain insights into the potential benefits and considerations for our own work.

E. Latency estimation of NN inference on Mobile GPU

As described in this work on latency prediction of different NN architectures specifically on mobile GPUs, there were many trials targeted on mobile CPUs so far. Examples are applying NAS for searching optimal model and having measurements for any suggested NN model on mobile CPU. Another approach was to simulate the target device (again mobile CPU) to parallelize NN latency estimation for making the estimator performance fast. However that approach seems "quite complex engineering task" for the authors [4, p. 2]. As stated in the paper, although for NN inference platforms based on GPUs and NPUs (Neural Processing Unit) are more preferred, there are very few studies on modeling the latency estimation problem for these, especially for mobile GPU [4].

The paper aggregates different approaches (Prediction models and LUTs) and evaluates their performance on a collection of NNs. Other than the GCN estimator discussed above [5], they talk about Gradient Boosting method in which the input data is the combination of number of FLOPs in the considered NN model, "flatten adjustment matrix", list of layers and peak memory. The algorithm tries to optimize the latency approximation function using gradient descent method. Similar ML algorithms belonging to the family of Gradient boosting, such as XGBoost, LightGBM and CatBoost are discussed [4].

As in other Prediction model approaches, here as well, target NN architectures need to be encoded before being fed to the predictor model. Here generated models are represented as adjacency matrices of graphs with nodes standing for the layers and edges for the connections between them, like in the case of

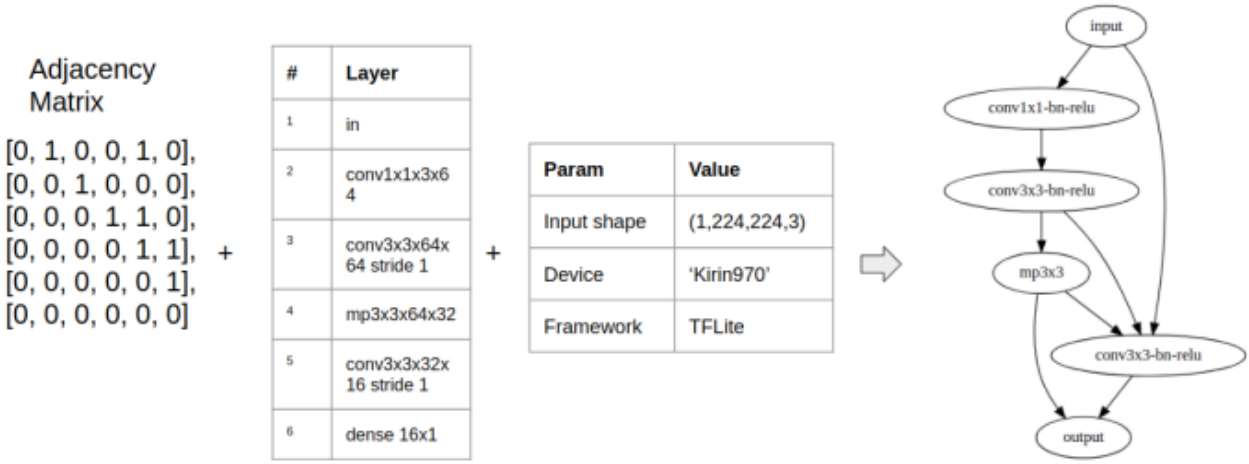

Figure 4: Parametrization of neural network implementation as a set of adjacency matrix, layers list, and config dictionary *[4, p. 3]*.

the GCN estimator (this process is called parametrization, see in Fig 4). This is how the dataset of the predictor is created. Based on this graph parametrizations TensorFlow.keras models are generated and converted to TensorFlow Lite, TFLite, (since models for mobile devices need to be small and light) and the latencies are measured by directly running them on desktop/server GPUs, also CPUs or NPUs of Android devices. Together with the tested NN configurations the observed latencies are recorded as one dataset. According to the work, having a dataset with actual latencies recorded gives an advantage since they show the non-trivial relations with computational operations, peak memory size,etc [4][2].

Key Takeaways: As an evaluation of this work, we will incorporate several key ideas into our approach, despite its primary focus on mobile GPUs. Similar to the GCN approach, we will explore the concept of encoding the input NN architectures to capture their design characteristics, which will enable more realistic

[2] We took this approach of creating the dataset in our work

estimations of the inference latency. Additionally, we will generate the dataset using the tf.keras framework, following a similar methodology. By including the ground truths (true latencies) alongside the architecture encodings, we can address the challenge of lacking explicit knowledge and representation of the inherent relationships between latency and architecture design.

However, it is worth noting that some methods mentioned in the paper, such as utilizing FLOPs or Gradient Boosting Methods, will not be employed in our project. This decision is driven by the fact that in case of those methods, constructing the objective cost function requires a deep understanding of the laptop's architecture and how models will be executed on it, which itself requires an extensive study. Hence, for the sake of simplicity and feasibility, we will focus on the aforementioned approaches that align more closely with our project goals.

F. Conclusion

In summary, the estimation of latency for neural network architectures involves the use of various approaches across different hardware specifications. Figure 5 provides a representative overview of these approaches. These methods encompass techniques such as real-time measurements, operator-level latency lookup tables and prediction models (which are NNs themselves). Each approach has its own advantages and limitations, and the choice of method depends on factors such as available computational resources, estimable model's complexity and architecture specifics, and target hardware platform. By exploring and combining these different approaches, researchers strive to improve the accuracy and efficiency of latency estimation for neural network architectures.

Method	How the method is achieved ?	Hardware Cost Metric
Real-time measurements	The sampled model is executed on the hardware target while searching.	Latency
		Energy
Lookup Table Models	A lookup table is created beforehand and filled with each operator latency on the targeted hardware. Once the search starts, the system will calculate the overall cost from the lookup table.	Latency
Analytical Estimation	Compute a rough estimate using the processing time, the stall time, and the starting time.	Latency
		Energy
		Memory footprint
		Area
Prediction Model	Build a ML model to predict the cost using architecture and dataset features.	Latency

Figure 5: Different methods of estimating different HW cost target variables (see latencies only) *[2, p. 16]*.

Pros & Cons:

A succinct evaluation of the latency estimation approaches:

- **Analytical models** – Use theoretical analyses and mathematical expressions to estimate the latency of neural networks depending on features, such as layer configurations, input and output dimensions, architecture, etc.

  Advantage- **1.** fast estimations
  2. no need to execute computationally heavy NNs or perform heavy experiments on HW.

  Disadvantage- (according to OpenAI's ChatGPT (OpenAI, April, 2023))[3]- **1.** analytical models are based on theory, simple and thus incapable of capturing the real-world sophisticated scenarios, HW intricacies, intralayer dependencies and low-level operations **2.** Analytical models are based on too general assumptions and rules.

- **Look-Up Tables** – Represent precomputed and recorded runtime performance data about the NN architectures. LUTs record the observed latencies of different building blocks of existing neural networks on various hardware platforms, providing valuable insights into the latency characteristics of specific network components.

  Advantage- **1.** fast and efficient
  2. requires only as many experiments as many NN building blocks exist in modern architectures [4]

  Disadvantage – **1.**Does not incorporate the latency of data flow between blocks model loading on the target HW which can have a considerable impact on the overall latency [4].
  2. The absence of relevant data points for similar NN architecture configurations can limit the accuracy and generalizability of the latency predictions.

- **Real-time measurements** – Directly measure the execution or inference time of a NN architecture on a target HW platform by running the NN model multiple times and providing different input data to capture latency fluctuations.

  Advantage – **1.** precise since it is the real latency
  2. reliable for the specific target HW since is impacted by the platform-specific characteristics (number of computational operations, clock speed, etc.)

  3.may be useful in measuring and optimizing (NAS) latency for NN models employed in real-time applications especially on mobile GPUs [4] or real-time server-client interactions (time-sensitive networks) [8]

  4. Since this approach yields the true measurement of the NN latency, it is considered to be "the best match to real user experience" [4].

  Disadvantage – **1**.may be a computationally heavy and time-consuming process in case the size of the target NN and input data is huge

  2.The inference time of a neural network can vary from one run to another, resulting in fluctuations. To accurately estimate the latency, it is necessary to conduct a significant number of experiments. However, this can be challenging as it often requires multiple devices of exactly the same type to simultaneously run the experiments in parallel [4].

- **Proxy measurements: FLOPs** – Count the low-level operations to be executed during the target architecture's execution or inference and output that number as a proxy estimation for the latency.

[3] Text generated by ChatGPT: https://chat.openai.com/

** Note that answers provided by ChatGPT are generated based on the information it has been trained on and may not always be accurate or reliable.

Advantage – **1.**Straightforward to compute.
2.can be calculated without having deep knowledge of the HW platform
3.When the size of the NN architecture grows, the proxy measures usually grow as well, enabling rough estimations for large networks (scalability)[4]

Disadvantage – **1.** If the proxy measure for NN latency is chosen to be the number of FLOPs, it does not consider that different FLOPs take different amounts of time to be executed on the HW. FLOPs also are influenced by the type of NN architecture and device being used which can hinder the cross-platform interpretability of the obtained results.

2.This approach usually does not include the time spent on data movement, loading and transfer operations since these are not considered to be computational operations, hence are not also FLOPs. But loading and movement take considerable time and not adding that up can bring to misleading results [4]

3. In many cases, the number of FLOPs is not directly correlated with the latencies of DNN models. This is because modern DNN frameworks often employ optimization techniques, such as layer fusion, where multiple layers are combined into a single operation. According to an academic work on enhancing the latency predition of DNNs, DNN frameworks "often fuse multiple layers into a single operation to accelerate the inference on real hardware devices" [9, p. 2]. These optimizations reduce the actual latency of the network and, therefore, merely counting the FLOPs may not accurately reflect the actual latencies of DNN models [9].

- **Prediction models** – NN models that are trained on some data representative of the NN architectures for which the latency is to be measured; try to learn patterns and dependencies from the features of that data and output a prediction for latency.

  Advantage – considers the connectivity of the input NN architecture which makes the latency estimation close to the actual [9]

  Disadvantage – **1.**Need to be trained on high-quality data that accurately represents all the aspects of the considerable NN architecture
  2.It is crucial to understand what parameters impact the latency and pick the most relevant ones (for example, layer parameters like kernel size for CNNs are often incorporated in the input data) [9]. However, picking the input data features and format is not an easy process and requires domain knowledge and deep understanding of the underlying mechanisms of the network inference.

IV. Our Approach

Our proposed approach builds upon existing methodologies for estimating the latencies of different neural network architectures on various target platforms. We acknowledge the limitations of the alternative approaches, such as the proxy measurements, and aim to address them by developing a prediction model that considers the dataset features and architecture encodings. The proposed approach is an example of a supervised learning.

Drawing from previous studies, we recognize the importance of using a comprehensive dataset. In our approach, we gather encoded vectors that represent the number of neurons on each layer of deep FCNN architectures. Additionally, we incorporate the use of the non-linear ReLU (Rectified Linear Unit) activation function to capture the non-linear relationships between latency and layer configurations.

The problem we seek to address is the estimation of latency for deep FCNNs with up to five-layers and a random number of neurons in each (up to 50). These FCNNs take as input

[4] Text generated by ChatGPT: https://chat.openai.com/

1D data, which serves as a proxy for sequential or time series data. This enables us to simulate signal processing tasks, where the 1D data represents sequential inputs corresponding to different time stamps.

The input architectures of the estimator model can be conceptualized as sequence-to-sequence (Seq2Seq) models. Seq2Seq models are commonly employed in tasks that involve processing sequential data, such as speech recognition, text summarization, and machine translation. In our case, the input architecture takes sequential data as input and generates a random continuous value as output.

By leveraging the Seq2Seq model setup, our latency estimation model serves as a preliminary step towards addressing latency in signal processing applications. Signal processing tasks often require low-latency processing for real-time analysis and timely responses. By applying our latency estimation model, we can measure and optimize the latency of the system, ensuring that it meets the strongest requirements of low-latency real-time processing in signal processing applications.

A. Data preparation

1) Firstly, we create a search space for fully connected deep neural net architectures in a popular open-source ML framework, called TensorFlow, using its high-level NN API Keras which is used for building deep learning models abstracting the low-level complexities. The created nets have varying depths (number of layers). For each of them the number of nodes (neurons) in the input and output layers is the same and all hidden layers can have random number of nodes.

2) Then generate a dataset by sampling 7000 network architectures from that search space. For each architecture, we retain the corresponding encodings that capture the specific characteristics of the architecture. An encoding is represented as a 1D vector with a size equal to the number of layers in the architecture. Each cell in the vector indicates the number of neurons present in the corresponding layer of the architecture. By collecting these encodings, we create a dataset that represents a diverse range of network architectures with varying layer configurations.

3) Subsequently, we create TensorFlow.keras Sequential models based on those configurations. Then measure the inference latency of the sampled network architectures 5 times, take the average, and incorporate that value into the dataset alongside the input model architecture encodings. The measurements are conducted on the CPUs of a local PC and serve as ground truth.

The generated dataset, consisting of the architecture encodings and corresponding latency measurements, forms the foundation for training and evaluating our latency estimation model. As suggested in [4], this approach of dataset creation allows us to gather information about both the neural network architecture and the underlying hardware platform. By including the ground truth latency values, we ensure that the model learns the true latency patterns and can make accurate predictions for unseen network architectures.

B. Building, training, testing the latency estimator

1) Build a deep FCNN which serves as our latency estimator model for predictions of inference latency on the input deep FCNN architectures.

2) Train the estimator to learn the relationship between the architectures (encoded as config vectors) and the latency.

3) Refine the model through hyperparameter tuning, which involves adjusting the architecture of the estimator model, such as adding or reducing the number of layers and changing the number of nodes in each layer, adding normalization or regularization, etc. The training and validation processes can be tracked

and analyzed using the TensorBoard extension in the Jupyter Notebook environment. TensorBoard provides a user-friendly interface to monitor and visualize various aspects of the ML experiments, including loss curves, metrics, and plots. This allows for better insights into the model's performance and aids in making informed decisions during the refinement process.

4) Evaluate the estimator performance using the MSE (Mean Squared Error), RMSE (Root Mean Square Error), MAE (Mean Absolute Error) and the average absolute latency deviation as the evaluation metrics, having the latency predictions and the ground truths.

5) Report the obtained performance metrics. Alongside create visualizations of the training and validation curves, using python's standard libraries(matplotlib, seaborn) and TensorBoard extension.

B. Description of the proposed estimator model

The estimator is a linear regression model with the non-linear ReLU activation function in each layer to capture the non-linear dependencies between the NN architectures and their inference latencies. This choice is motivated by the findings presented in the ANNETTE paper [7], which indicate that linear regression models achieve high accuracies when CPUs are the target platform.

Given the limited scope of our project, our estimator model is designed to be simpler in structure compared to the more complex Prediction model GCN discussed earlier. This decision was made to ensure a manageable and focused approach that aligns with the specific goals and constraints of our project.

After multiple trials and fine-tuning, the initial estimator model was refined. The final model architecture consists of three hidden layers with regularization and normalization techniques and input and output layers. The output layer still utilizes a "linear" activation function. These modifications were made to improve the model's performance and enhance its ability to generalize to new data.

The model was compiled with the Adam optimizer, MSE as the loss function and MSE, MAE, RMSE as the evaluation metrics.

The estimator model was trained on a dataset of 4,480 architecture encodings. The model's performance was validated using a separate set of 1,120 encodings, and then tested on another independent set of 1,400 encodings. The limited size of the training and test datasets was due to memory constraints, which prevented the creation of a larger dataset containing approximately 15,000 architecture encodings. Despite the limited data size (overall 7000 architectures), the model's performance was evaluated and refined to the best extent possible within the available resources.

Performance metrics: During the analysis of the estimator's performance, various metrics were utilized to assess its effectiveness from different perspectives. The following metrics were employed to evaluate the predictor's performance:

- $MSE = \frac{1}{n}\sum(Y - \acute{Y})^2$

Here Y is the actual latency of the input model and Ý is the estimated one.The choice to use MSE (Mean Squared Error) is justified since MSE is a convex function which makes the gradient descent optimization algorithm converge to the global minimum. Also, MSE, being a differentiable loss function, was used when training our estimator to update and optimize the parameters.

- $RMSE = \sqrt{\frac{\sum_{i=1}^{N}(Y_i - \acute{Y}_i)^2}{N}}$

RMSE (Root Mean Squared Error) is a metric commonly used in supervised learning tasks. It represents the square root of the average of the squared differences between

the actual and predicted values of the target variable (latency). The advantage of RMSE over MSE is that its units are in the same scale as the target variable, which in this case is milliseconds. In contrast, MSE is measured in squared milliseconds. Also, RMSE penalizes larger errors in latency prediction than the MSE since by taking the square root, RMSE magnifies the impact of larger deviations, as the squared differences grow non-linearly with increasing deviation. Thus, RMSE is preferred for its better interpretability in terms of the original units of the target variable.

- $MAE = \frac{\sum_{i=1}^{N} |Y_i - \hat{Y}_i|}{N}$

MAE (Mean Absolute Error) is another commonly used metric for evaluating the performance of a predictor. It measures the average absolute difference between the actual and predicted values of the target variable. Unlike RMSE, MAE does not square the differences, making it more robust to outliers. The units of MAE are the same as the target variable, in this case, milliseconds, allowing for easy interpretation. MAE provides a straightforward measure of the average magnitude of the errors in the predictions and is often used together with RMSE.

V. RESULTS

Before discussing the results, it is important to mention the target platform specifications on which the average latencies of the input architectures were measured on. It is based on laptop CPUs (Intel(R) Core(TM) i7-8750H CPU 2.20GHz), having 16.0 GB of RAM with 2400MHz of speed, SSD with disk transfer rate of 10MB/s, 239GB of capacity TOSHIBA KSG60ZMV256G M.2 2280 256GB.

This platform was utilized to generate the dataset, providing valuable insights into latency measurements on CPU-based platforms. This was achieved by changing the connection to a local runtime on Google Colaboratory during the dataset creation (the whole project was conducted on Google Colaboratory environment). However, to expedite the training process of the estimator model on the already created dataset, the available remote resources of Google Colaboratory were leveraged by changing the connection back to the remote runtime. This allowed for efficient utilization of computational power and reduced the training time required for the model.

The results observed were (MSE=0.0007, MAE=0.0211, RMSE=0.0261, average absolute latency deviation=0.02).The achieved performance metrics are highly satisfactory and align with the expectations based on prior research, which suggests that simple regression models perform well on CPU-based platforms [7]. This observation supports the notion that the estimator model effectively captures the underlying patterns and relationships in the data, leading to accurate predictions of latency on CPU architectures.

Below you can find the training and validation loss curves and the distribution plots of the average absolute deviation values for the latency (See Fig. 6 and Fig. 7).

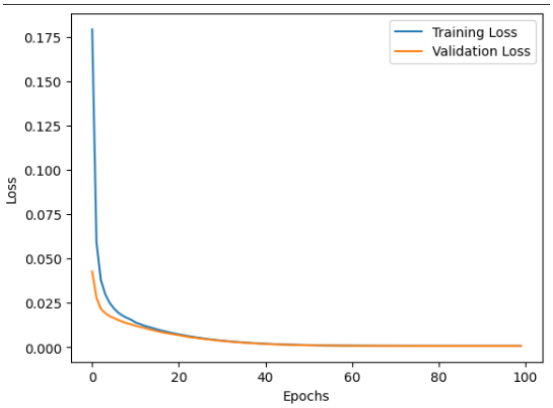

Figure 6: Training vs Validation loss curves (both decrease, i.e., no overfitting)

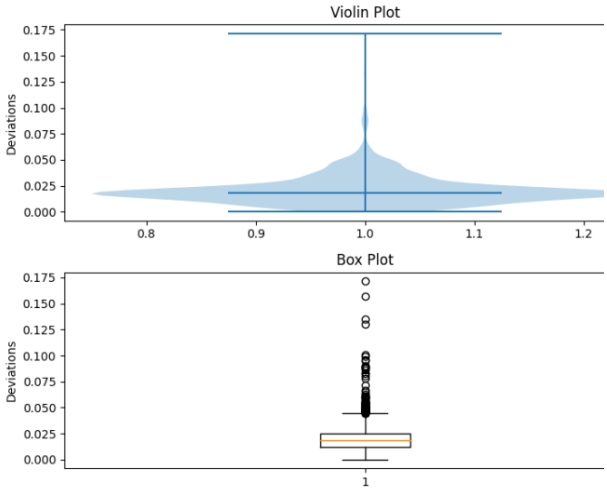

Figure 7: Violin plot and Boxplot of the absolute deviations between the predicted and actual latencies. The median (Q2) is a small number (0.025), i.e., the 50% of latency deviations lies below 0.025. This indicates that the estimator's predictions are generally accurate and have minimal error or discrepancy compared to the ground truth.

You can find the source code in this github repository: **https://github.com/lilitbeglaryan/CAPSTONE_AUA_2023_SPRING.git**[5]

VI. Future Improvements

In terms of future enhancements, there are several possibilities to consider for the estimator model:

- ✓ **Generalizing input architectures:** Currently, the estimator focuses on encoding deep FCNNs. An improvement could involve extending the encoding to accommodate other types of neural networks, such as CNNs or RNNs. This expansion would allow for a more comprehensive representation of different neural network architectures. The capacity of the estimator can also be expanded (often via improving the number of weights and parameters of the estimator network, its architecture and depth) to estimate latencies for sophisticated network structures.

- ✓ **Multidimensional tensor encoding:** Instead of limiting the configuration vector of the architecture design representation to 1D, a potential enhancement could involve using a multidimensional tensor to capture the intricate details of the input neural network architecture, not only the number of nodes in each layer. This approach would offer a more expressive representation.

- ✓ **Targeting different hardware platforms:** Currently focused on estimating latencies on CPUs, an additional extension could involve expanding the estimator's capabilities to target other hardware platforms, such as GPUs, FPGAs, NPUs (neural processing units), and others. This enhancement would allow for a broader range of latency estimations across various computing devices.

These future enhancements would broaden the estimator's applicability, allowing for a more comprehensive analysis of different neural network architectures and hardware platforms.

Acknowledgment

The author would like to express their sincere gratitude to Professor Dr. Gregor Schiele from the University of Duisburg-Essen for graciously agreeing to assign one of his exceptional PhD students, Mr. Christopher Ringhofer, to collaborate on this capstone project. We sincerely appreciate his willingness to support the project and provide us with a talented and dedicated team member.

Furthermore, we would like to extend special thanks to Mr. Christopher Ringhofer for his invaluable guidance, support, and technical contributions throughout the research process.

5 *https://github.com/lilitbeglaryan/CAPSTONE_AUA_2023_SPRING.git*

We are deeply grateful for his feedback, cooperation, and unwavering dedication. The author recognizes the immense value that Mr. Ringhofer's involvement brings to this capstone project. His knowledge and expertise in the field have been instrumental in shaping the research approach and enhancing the overall outcomes.

VII. BIBLIOGRAPHY

[1] C. F. R. G. R. e. a. Eva García-Martín, "Estimation of energy consumption in machine learning," *Journal of Parallel and Distributed Computing,* vol. Volume 134, no. ISSN 0743-7315, pp. Pages 75-88, 21 August 2019.

[2] K. E. M. H. O. e. a. Benmeziane, "Comprehensive Survey on Hardware-Aware Neural Architecture Search," *arxiv.org,* 22 January 2021.

[3] OpenGenus IQ, "What is Latency in Machine Learning (ML)?," OpenGenus IQ, [Online]. Available: https://iq.opengenus.org/latency-ml/. [Accessed 20 April 2023].

[4] S. M. I. O. a. Evgeny Ponomarev, "Latency Estimation Tool and Investigation of Neural Networks Inference on Mobile GPU," *MDPI,* 23 August 2021.

[5] T. C. M. S. A. e. a. Łukasz Dudziak, "BRP-NAS: Prediction-based NAS using GCNs," in *34th Conference on Neural Information Processing Systems (NeurIPS 2020),* Vancouver, Canada.

[6] P. Z. ,. B. W. e. a. Xiaoliang Dai, "ChamNet: Towards Efficient Network Design through Platform-Aware Model," in *CVPR(Conference on Computer Vision and Pattern Recognition),* 2019.

[7] A. N. A. W. e. a. Matthias Wess, "ANNETTE: Accurate Neural Network Execution Time Estimation with Stacked Models," *arxiv.org,* 10 May 2021.

[8] C.-C. Wu, "A SURVEY FOR REAL-TIME NETWORK PERFORMANCE," Cornell University (arXiv), 2020.

[9] L. M. F. C. Robin Zbinden, "COBRA: ENHANCING DNN LATENCY PREDICTION," in *Deep Learning for Code workshop at ICLR 2022*, 2022.