Assessing the performance of an architecture-aware optimization tool for neural networks

Raúl Marichal
Instituto de Computación
Facultad de Ingeniería
Montevideo, Uruguay
rmarichal@fing.edu.uy

Ernesto Dufrechou
Instituto de Computación
Facultad de Ingeniería
Montevideo, Uruguay
edufrechou@fing.edu.uy

Pablo Ezzatti
Instituto de Computación
Facultad de Ingeniería
Montevideo, Uruguay
pezzatti@fing.edu.uy

***Abstract*—The important growth in the demand for Neural Network solutions has created an urgent need for efficient implementations across a wide array of environments and platforms. As industries increasingly rely on AI-driven technologies, optimizing the performance and effectiveness of these networks has become crucial. While numerous studies have achieved promising results in this field, the process of fine-tuning and identifying optimal architectures for specific problem domains remains a complex and resource-intensive task. As such, there is a pressing need to explore and evaluate techniques that can improve this optimization process, reducing costs and time-to-deployment while maximizing the overall performance of Neural Networks.**

This work focuses on evaluating the optimization process of NetAdpat for two neural networks on an Nvidia Jetson device. We observe a performance decay for the larger network when the algorithm tries to meet the latency constraint. Furthermore, we propose potential alternatives to optimize this tool. Particularly, we propose an alternative configuration search procedure that allows us to enhance the optimization process, achieving speedups of up to $\sim 7\times$.

***Index Terms*—efficient computing, neural network optimizations, edge devices, heterogeneous computing, NetAdapt**

I. Introduction

Developing and deploying efficient deep learning algorithms is of the utmost importance in today's data-driven world since they profoundly impact various domains and industries. Efficient algorithms enable faster and more accurate data analysis and reduce costs by minimizing computational resources, such as processing power and memory requirements, leading to optimized hardware utilization and energy savings. Moreover, making the deployment of machine learning models possible on resource-constrained devices, such as mobile phones or Internet of Things (IoT) devices, extends the reach of AI applications to edge computing scenarios. This is the main motivation for studying novel techniques to optimize this kind of computation.

The efforts to improve the computational performance of AI solutions are varied, ranging from hardware optimizations, developing efficient specialized platforms that suit this kind of workload (ASICs), to software or model-driven approaches that allow optimizing by, for example, understanding the specifications of the hardware and adapting the computations to a new workflow [14]. Among those efforts, Neural Architecture Search (NAS) techniques have gained importance in recent years. It consists of exploring the space of possible DNNs for a certain task and obtaining one that adjusts to a certain budget on properties such as inference latency, energy efficiency, and accuracy. The sample that better adjusts to the budget is finally trained to obtain the final network.

There is a wide range of NAS methods, such as Evolutionary NAS [15], reinforcement-learning-based approaches [16], [21], or differentiable NAS [1], [11], [17].

A common characteristic of the above approaches is their remarkable computational cost. This is often tackled by recurring to large computing infrastructures and applying High-Performance Computing (HPC) architectures and techniques. Nevertheless, this infrastructure is not always available for most developers. Therefore, it is of utmost importance to work on efficient implementations of this family of algorithms in more common or restricted environments that are equipped instead of multiple nodes with hundreds of Graphics Processing Units (GPUs) available, only a node with a couple of them.

A recent trend related to NAS is to optimize the network's architecture using performance samples taken on the target computing device [1], [2], [16], which is called platform-aware optimization. A clear example is NetAdapt [19], [20]. Following a latency budget, this software tool optimizes a pre-trained neural network model based on empirical measurements on the platform in which it will be deployed.

NetAdapt uses an iterative algorithm that simplifies the network, obtaining a set of candidate models and selecting those that show the best relation between the performance in terms of the metric to be optimized (e.g., latency) and the accuracy. Using empirical measurements of the target platform allows NetAdapt to produce optimized models without specific knowledge about that platform's specifications.

Continuing a previous work [12] where we evaluate the benefits of NetAdapt, in this opportunity, we present a detailed analysis of the optimization process done by the original implementation of the algorithm [18], in an environment with 3 GPUs. With these results, we present a strategy to speed up the optimization process of the NetAdapt, focusing on optimizing the search for optimal layer width that satisfies a given latency constraint on each algorithm step.

The rest of the paper is structured as follows. In Section II,

979-8-3503-8160-3/23/$31.00 ©2023 IEEE
DOI 10.1109/SBAC-PADW60351.2023.00011

we describe in detail how the NETADAPT algorithm works and present an analysis revisiting some of our previous efforts using this tool. The description of our novel proposal follows this in Section IV, and the corresponding experimental evaluation, with the environment and test cases, is in Section V. Finally, a few remarks and some lines of future work close the paper in Section VI.

II. EFFICIENT NEURAL NETWORKS

Multiple efforts are dedicated to optimizing Deep Learning computations, which can be categorized into two main levels: software and hardware optimizations.

On one hand, hardware-driven optimizations focus on efficiency and energy consumption while maximizing hardware utilization, requiring the design of specialized hardware architectures. This idea has been derived in new Application-Specific Integrated Circuits (ASICs) specifically designed for deep learning workloads, a tendency that authors like Hennessy and Patterson pointed out a few years ago [5]. Examples range from academic (implemented) architectures. One of them is Eyeriss [3], an energy-efficient accelerator that minimizes data movement. There are also industry developments such as Google's Edge Tensor Processing Unit (Edge TPU) [4], which presents a matrix of Processing Elements (PEs) that has specific limitations such as a reduced subset of operations and only supports INT8. Other companies, like META, have put effort into developing a new family of hardware platforms called MTIA [13], optimized for their deep learning recommendation models (DLRMs) workloads.

Software optimizations involve improving performance through transformations, rearrangements, pruning, and quantization techniques. Pruning removes unnecessary computations, while quantization converts the model's data format into smaller representations. For example, instead of using Floating Point (FP32, 32 bits) to represent weights and activations, in some cases, they can be stored and computed with an FP16 or INT8 representations, reducing the memory footprint associated with the model and allowing faster inferences with small and, depending on the problem, acceptable accuracy drops. Also, an important line of research is the Network Architecture Search (NAS), which helps automate the process of finding new optimal neural network architectures for specific domains and tasks. Instead of the laborious job of developing novel architectures, which is directly tied to expertise and knowledge in the field, this type of algorithm helps to alleviate this demanding task, allowing the automatic design of performing neural networks by using techniques of machine learning itself, in problems like image classification, object detection, and language translation. Even though it is an effortless and automatic way to develop new architectures, there are multiple aspects to consider when developing and deploying NAS. For example, it is necessary to define a search space of possible neural network architectures, including different types of layers, number of layers, width of layers, skipped connections, etc.. In most cases, this yields a huge spectrum of options, easily becoming a complex and costly search problem, handled by, for example, reinforcement learning and evolutionary algorithms [15], [21]. There are other aspects, such as model training and evaluation over the iterations of the algorithm, that, depending on the search spaces, can take several computing resources in the form of massively parallel hardware (e.g., hundreds of GPUs) and a lot of computing time, which is not always available.

Other authors look for better accuracy-latency trade-offs in the resulting networks by incorporating metrics such as the measured latency of a sampled network on the target inference device. This can allow obtaining DNNs with better latency and energy efficiency in resource-constrained environments than NAS methods that do not incorporate the characteristics of the inference computing platform while preserving a good level of quality in the solution [16], [19]. An excellent example of this sort of tool is NETADAPT, in which we focus on this work.

A. NETADAPT

NetAdapt [19] is an automated algorithm designed to adapt and optimize neural network models for specific hardware architectures. It sequentially optimizes a pre-trained model through layer-level simplifications, reducing the number of filters involved using empirical measures from the target platform. Note that it uses a similar approach to NAS, by automatically designing a new model, but in this case, limited to a pre-existing architecture, without taking into account more complex decisions such as defining new search spaces that can grow immensely fast. This allows, depending on the complexity of the original model, the development of efficient and high-performing neural network models without the need for an excessive computing infrastructure. Nevertheless, as we will see in the following sections, running in relatively normal setups is still costly.

As an example of an application, in [6] NETADAPT is used to complement the NAS algorithm to search a new version of Mobilenet (V3). In particular, the authors apply the algorithm to obtain a smaller version of the architecture. The algorithm is an iterative procedure where each iteration involves simplifying, training, and evaluating independent models. To gain a comprehensive understanding of the algorithm's functionality and the steps involved, we hereby present a simplified explanation as follows:

1) It starts with a pre-trained model (seed). For example, in [6], the authors start with an architecture found by NAS.
2) For each step until a maximum number of iterations or until the desired latency budget is reached (for example, $0.5 \times |L|$, where $|L|$ is the latency of the model):
    a) A set of architectures or *proposals* is generated. Each one is a modification of the current model in the step, resulting in at least a δ reduction in latency (in the NETADAPT implementation, there are as many proposals as there are layers in the network, reducing the number of filters for that layer until the constraint is reached, for example, a reduction $\delta = 0.025 \times |L|$). In [18], this step is implemented with a system of workers that

independently find a simplified network definition. This reduction in the number of filters is evaluated sequentially using a Look-Up Table (LUT) containing latency measurements for the specific platform. In some cases, depending on the configuration and the LUT, the latency value for a configuration must be obtained by interpolating the latency values in the LUT since not all the configurations are evaluated in the target platform (which would take several days or even weeks depending of the width of the layers, step reduction, and the iterations to average). For instance, we use a reduction on the channel dimension (for the input and output) of 8 for convolutional layers and 64 for fully-connected layers since the fully-connected layers have a much bigger dimensionality, taking days for smaller reductions.

b) For each of the proposals, once the constraint is met, the model is created in the worker's assigned GPU. To create the new models, the pre-trained model from the previous step is taken and pruned to match the architectures proposed in the previous step.

c) Subsequently, all models are fine-tuned by T steps to obtain a reasonable accuracy.

d) Once fine-tuned, the models are evaluated, and the accuracy is stored.

e) The best candidate given a certain metric is selected as the current model for the next iteration. This metric is calculated considering the latency and accuracy.

It can be observed that, for example, the parameters T and δ may impact the final elapsed time to compute the algorithm. T represents the number of fine-tuning iterations involving forward passes and back-propagation computations, which is well-known to be a computationally intensive task. Furthermore, δ references to the speed of latency reduction. The larger this value, the faster (at least in the number of iterations) the latency budget should be reached. As a reference, the application of NETADAPT in [6] sets the parameters $T = 10000$ (for ImageNet) and $\delta = 0.01|L|$ (i.e., 1% of latency of the seed model), and finally trains the candidates from scratch, having a 4x4 TPU Pod [8] for the training.

In our previous experimental evaluations, even though we reached good results reducing the models' latency and preserving acceptable levels of accuracy (Table I), the higher execution time when applying NETADAPT in larger networks (for example, AlexNet) stood out. This considerable computational cost can make using the tool impractical in several situations. For this reason, it is interesting to obtain a detailed performance profile of NETADAPT and evaluate the opportunities for performance improvement.

III. ANALYSIS

The benefits of applying NETADAPT, as we shown in our previous work, were clear (for example, see Table I).

TABLE I
LATENCY REDUCTION RATIO FOR ADAPTED VERSIONS OF ALEXNET EXECUTED ON DIFFERENT DEVICES: INTEL NCS2, NVIDIA GTX 970, NVIDIA JETSON TX2 AND CPU INTEL CORE I7-4790. EXTRACTED FROM [12].

Latency reduction ratio between AlexNet models					
Device	NA-NCS2	NA-GTX	NA-TX2	NA-i7	Base Model
NCS2	59%	38%	57%	45%	0.020ms
GTX	23%	82%	23%	5%	0.004ms
TX2	35%	18%	36%	17%	0.020ms
i7	62%	39%	64%	64%	0.044ms

Nevertheless, its application can be quite expensive in terms of computational cost and time since multiple instances of models need to be independently optimized (sequentially), trained, and evaluated to choose the best-performing one in each step. In this section, we present a few results of different NETADAPT configurations to show how costly the computation can be and how, depending on the parameters used, it may drastically affect the elapsed time to compute the optimization.

TABLE II
ELAPSED TIME (IN SECONDS) AND NUMBER OF ITERATIONS TAKEN TO COMPUTE NETADAPT (ORIGINAL) FOR DIFFERENT δ VALUES, USING $T = 500$.

	Mobilenet		**Alexnet**	
δ	Elapsed time	Iterations	Elapsed time	Iterations
0.01	3.434×10^4	40	1.343×10^5	68
0.025	1.662×10^4	18	1.003×10^5	24
0.05	1.010×10^4	11	7.282×10^4	12

We ran several experiments of the original implementation of NETADAPT on a computer/server equipped with 3 Nvidia GPUs over two well-known convolutional neural networks: MobileNet and AlexNet. This setup and test cases are described with more precision in the experimental evaluation (Section V). We ran NETADAPT with $T = 500$ fine-tuning iterations and varying the δ value, seeking to speed up the velocity of latency reduction. From Table II, it can be observed that for the smaller δ in each step (i.e., smaller latency constraint reduction), MobileNet takes more iterations, and the elapsed time correlates with the number of iterations in all the cases (around 15 minutes per iteration).

On the other hand, for the case of AlexNet, we can see that for a higher δ value, the budget is, as expected, also reached in fewer iterations. Nevertheless, the elapsed time does not directly relate to these results, having very similar elapsed times or, at least, not a homogeneous time per iteration. Given that all tests have the same number of fine-tuning iterations (T), the discrepancy in elapsed time cannot be attributed to either the fine-tuning stage or the evaluation process. In fact, there are fewer iterations for increasing δ values, resulting in a reduced number of fine-tuning and evaluation stages. The reason for NETADAPT's behavior in the AlexNet case is that most of the time was spent finding the new architecture that satisfies the constraint in each step, computing the latency from the look-up table, and interpolating the missing values.

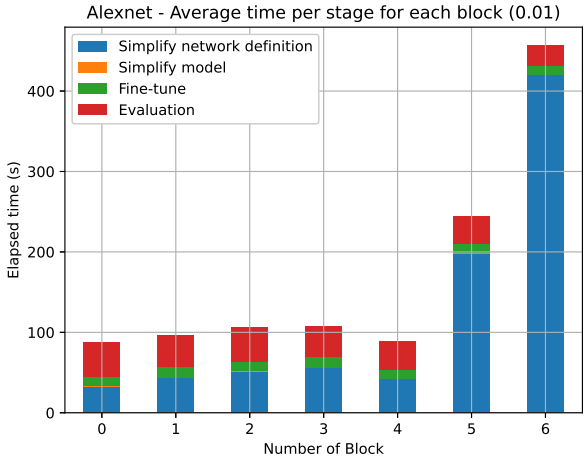

Fig. 1. Average elapsed time per stage in the worker's block optimization process. $T = 100$ and $\delta = 0.01$.

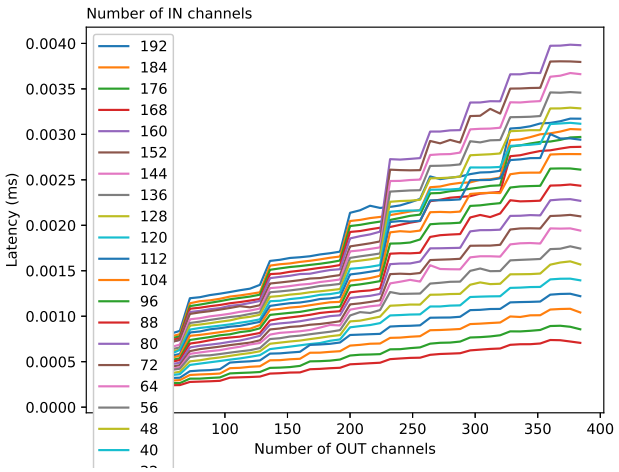

Fig. 3. Latency measurements of a Convolutional layer latency measures varying IN and OUT number of channels on an Nvidia Jetson Xavier.

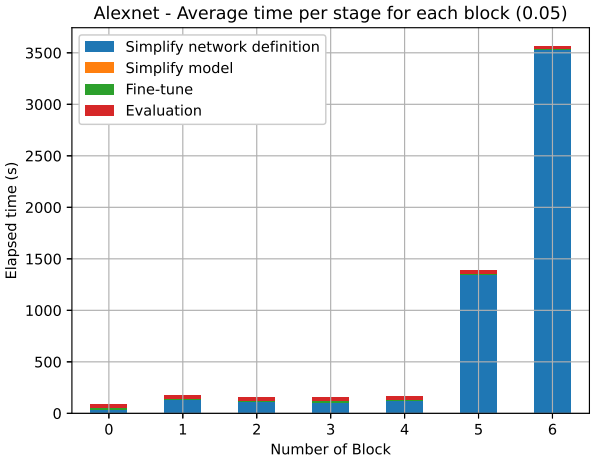

Fig. 2. Average elapsed time per stage in the worker's block optimization process. $T = 100$ and $\delta = 0.05$.

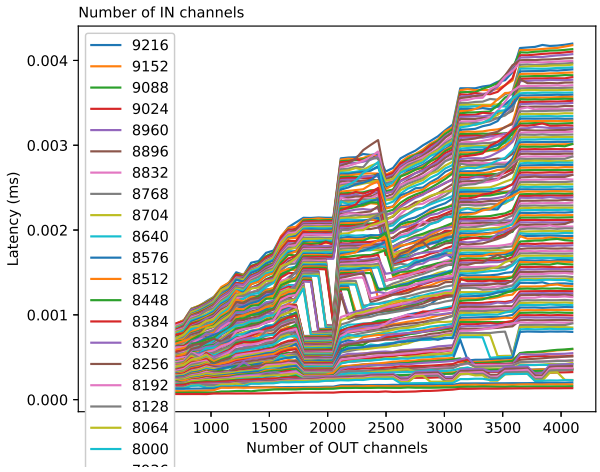

Fig. 4. Latency measurements of a Fully connected layer latency measures varying input and output size on an Nvidia Jetson Xavier.

The higher the δ, the harder it is (in terms of the number of latency evaluations) to find a good configuration that fulfills the latency restriction. This exact behavior can be observed in Figures 1 and 2, where in both cases, the time spent to simplify the network definition of AlexNet is larger than the other steps involved, the fine-tuning, model generation and evaluation (which have quite regular times across the simplified blocks). It must be noted that there is a certain regularity between the first blocks, but the last two stand out, associated with the fully connected layers with higher dimensionality. This gap is even more remarkable in the case of $\delta = 0.05$.

In Figures 3 and 4, we show how the latency (empirical measures) varies given the input and output dimensions for a specific device (Xavier). In particular, we can observe mostly linear growth for both types of layers for larger input and output sizes. Even though there are only two examples, we observe this behavior in most of the layers for other devices.

Considering these results, it would be interesting to address the problem of finding the configuration that allows satisfying the constraint. Particularly, the quasi-linear growth of the latency regarding the number of channels suggests using a faster method than linear evaluation over the number of channels dimension.

IV. Proposal

As pointed out, since the number of iterations of fine-tuning and evaluation remained fixed in the tests, and we vary the ratio reduction per step (δ), comparing the Figures 1 and 2 evidence the important and crescent cost is associated with the process of simplifying the network definition by searching the configuration that satisfies the step constraint. Following, we present a strategy to reduce the number of evaluations of the latency LUT and interpolations done to estimate the latency.

A. Modified search procedure (Binary search)

As discussed above, for most cases, latency measures present a "linear" growth in the function of the output and input channel dimensions. Therefore, to speed up the configuration search to meet the latency constraint for each layer, we propose to perform such a search in a different order. Instead of using a purely sequential evaluation, reducing the dimension in a magnitude of `min_feature_size` to compute the total latency of the architecture, we propose combining it with a simple binary search algorithm that modifies the size of the layer in a variable step until a certain threshold is met, to evaluate linearly later until the latency constraint is met. Note that the order of complexity of the original search is $O(n)$. In the worst case, all the possible configurations are evaluated. This is especially important since the latency LUT elements, the pairs of *((in_channels,out_channels): latency)* scale with a factor of $(in\_channels/step) \times (out\_channels/step)$.

For the case of AlexNet, we have a fully connected layer with 9126 in channels and 4096 out channels. Measuring the latency with a step of 64, e.g., the configurations (9126,4096),(9062,4096),... (64,4096) are evaluated until the latency constraint is met resulting in a total of (4096/64 * 9126/64) = 9126 configurations. Moreover, a modification in a layer dimension may affect the following layers. In some cases or configurations where the latency can not be computed directly from the LUT, the values are interpolated, adding an extra cost.

Therefore, assuming a linear growth for increasing channel dimensions, we propose a fairly simple variation of the classical binary search. The new search procedure is summarized in Figure 5. Although a pure binary search is not viable because the latency growth is not entirely homogeneous in some cases, only a few iterations of the binary search allow performing the linear search in a significantly more restricted neighborhood. This is especially useful in situations where the latency constraint is not reached, and, therefore, a long search is performed, which implies traversing a significant part of the dictionary associated with the latency LUT. Figure 6 shows an example of the new search procedure with $threshold = 1$ (pure binary search), trying to meet the 0.5 latency constraint. It has to be noted that this simple idea, assuming the quasi-linear growth, can lead to an important reduction in the number of latency computations, avoiding unnecessary latency LUT reads and interpolations, allowing to meet a latency constraint with only five evaluations instead of 10. The complexity of the new search algorithm reduces to $O(log(n) + threshold)$ instead of $O(n)$, being n the possible layer configurations.

```
threshold = 16
left = 0
right = len(num_out_channels_try) - 1
while right - left > 1:
    if (right - left) <= threshold:
        break
    mid = (right + left)//2
    current_num_out_channels =
        num_out_channels_try[mid]
    # Get the current resource consumption
    simplified_resource = compute_resource(
                            simplified_network_def,
                            resource_type,
                            lookup_table)
    if simplified_resource < constraint:
        right = mid
    else:
        left = mid
num_out_channels_try =
    num_out_channels_try[left:right+1]
```

Fig. 5. Pseudocode of the new search procedure.

Fig. 6. Example of binary search with threshold = 1, trying to met the constraint 0.5. Instead of 10 evaluations, assuming ordered latency measures, the searching problem can be solved in 5 evaluations.

V. Experimental evaluation

This section describes the experiments performed to assess the performance of our proposal and summarizes the main results. First, we present the experimental platform used for the tests, also used for the analysis presented in Section II.

A. Experimental Setup

All the experiments, including the ones reported in Section II, were performed in a server equipped with 3 GPUs, detailed described in Table III and Table IV, including hardware and software aspects such as *pytorch* version. The models were optimized for an NVIDIA Jetson Xavier using PyTorch CUDA.

B. Test cases

For the test cases, we evaluate the performance of NetAdapt applied to classical architectures: AlexNet [10] and MobileNet [7]. Both AlexNet and MobileNet are convolutional neural networks. AlexNet was specifically designed to address the challenge of large-scale image classification. It consists of multiple convolutional and high dimensional fully connected layers. We choose AlexNet (218MiB) to evaluate the possibility of deploying high-performing and larger-scale models on resource-constrained devices. Even if it is not a novel architecture, the dimensions and the number of parameters

TABLE III
DETAILED DESCRIPTION OF EXPERIMENTAL SETUP COMPONENTS.

Component	Specifications
CPU	Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz (4 cores, 64kB L1 cache)
RAM	64 GB
GPU 1	NVIDIA GeForce GTX 980 Ti (6GB)
GPU 2	NVIDIA GeForce GTX 980 Ti (6GB)
GPU 3	NVIDIA GeForce GTX 1060 (3GB)
Operative System	CentOS Linux 7 (Core)
Python version	3.6.8
PyTorch	1.10.1

TABLE IV
CAPABILITIES AND THEORETICAL PERFORMANCE OF NVIDIA GTX 980 TI AND GTX 1060 GRAPHICS CARDS.

Capability	Nvidia GTX 980 Ti	Nvidia GTX 1060
Architecture	Maxwell	Pascal
CUDA Cores	2816	1280
Base Clock (MHz)	1000	1506
Boost Clock (MHz)	1075	1708
Memory	6 GB GDDR5	3 GB GDDR5
Mem. Bus Width	384-bit	192-bit
Mem. Speed	7 Gbps	8 Gbps
Mem. Bandwidth (GB/s)	336.5	192.2
TDP (W)	250	80
FP32 Theoretical perf.	5.63 TFLOPS	3.85 TFLOPS
FP64 Theoretical perf.	187 GFLOPS	120 GFLOPS

employed in the layers still represent highly parameterized new models.

On the other hand, MobileNet is a way smaller model designed for efficient and lightweight deep learning on mobile and embedded devices. It has achieved high accuracy on various computer vision tasks while minimizing the computational resources required for inference. It is interesting to evaluate these two different scale models, the case for MobileNet (in its PyTorch implementation weights around 13MiB), seeking opportunities to develop even more lightweight models.

Both architectures are more precisely described in Table V, focusing on the dimensions and number of layers.

TABLE V
COMPARISON AND DESCRIPTION OF ALEXNET AND MOBILENET ARCHITECTURES.

Parameter	AlexNet	MobileNetV1
Depth	8 layers	28 layers
Number of Parameters	∼61 million	∼4.2 million
Size (MiB)	∼220 MiB	∼13 MB
Max FC Input Dims.	9216	1024
Max FC Output Dims.	4096	1000
Number of Conv. Layers	5	13
Number of FC Layers	3	1
Number of Pooling Layers	3	1
Number of Activation Layers	5	13

For the dataset, we use the CIFAR10 [9], composed of 60000 color images divided into ten classes. We set NETADAPT parameters: $T = 500$ to avoid the huge execution cost of the fine-tuning stage (since it is a simple dataset and we do not have enough parallelism to train all the variations at the same time), and the δ values, which controls the latency constraint reduction for the candidates' models in each iteration, we use $\{0.01, 0.025, 0.05\}$. We want to achieve a budget ratio of $0.5 \times L$ (half of the original latency).

C. Experimental results

TABLE VI
ELAPSED TIME TO COMPUTE NETADAPT IN THE DIFFERENT VERSIONS: ORIGINAL AND USING BINARY SEARCH, FOR DIFFERENT δ VALUES, WITH $T = 500$ OVER MOBILENET.

δ	Original	Binary search	SpeedUp
0.01	572	622	0.92×
0.025	270	268	1.01×
0.05	168	167	1.01×

TABLE VII
ELAPSED TIME TO COMPUTE NETADAPT IN THE DIFFERENT VERSIONS: ORIGINAL AND USING BINARY SEARCH, FOR DIFFERENT δ VALUES, WITH $T = 500$ OVER ALEXNET.

δ	Original	Binary search	SpeedUp
0.01	2239	798	2.81×
0.025	1672	319	5.24×
0.05	1214	174	6.98×

Table VII shows the important speedup reached for AlexNet while implementing the variation in the search for an architecture that satisfies the latency constraint in the iteration step. With a speedup of $\sim 7\times$ for the $\delta = 0.05$. Note that this behavior is not present for MobileNet (Table VI), which is consistent with the analysis done in Section II, where the average time per iteration remain consistent through the variations in the δ values, achieving an almost imperceptible speedup for $\delta = \{0.025, 0.05\}$. This result can also be observed in Figures 7 and 8, which shows the average time in minutes per iteration for the two different architectures. From Figure 8 can be observed how the time per iteration using the new search algorithm is way more regular for crescent δ values, as happened for MobileNet even before applying the optimized algorithm (Figure 7).

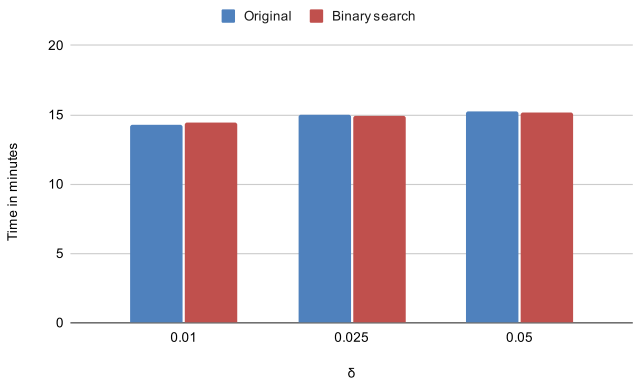

Fig. 7. Average time (in minutes) per iteration for the different implementations over MobileNet, with different δ values.

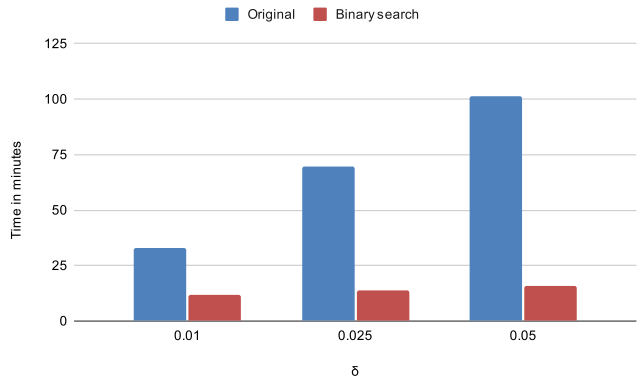

Fig. 8. Average time (in minutes) per iteration for the different implementations over AlexNet, with different δ values.

This can be explained due to the notorious difference in the channel dimensions on both architectures. An important question that arises from the proposal and these results is how the new search technique affects the final solution. The results show that it does affect the architecture found by the algorithm but with no big differences. For example, we did not perceive an important variability in terms of accuracy. This may be due to the configuration used since we evaluate the tool with a fairly simple dataset. Further tests with more complex data must be done.

VI. Concluding Remarks and Future Work

In this work, we evaluate the NetAdapt, focusing on the cost of computing it for networks with many filters per layer. In this case, AlexNet, with a fully connected layer with input size 9216 and 4096 as output size, shows that the sequential evaluation of the configurations that reduce the size of the layers, in some cases, takes an extraordinary amount of time. This is because the latency constraint was not reached during the reduction, thus hitting the minimum size of the filters and maximizing the number of LUT evaluations and interpolations. In addition, this result does not seem optimal since a layer that could be important is almost completely pruned. Given this cost, we propose an alternative implementation. Assuming some conditions based on the empirical latency measures, we replace the sequential evaluation with a faster algorithm to find the architecture that satisfies the latency constraint. In this case, we implement a binary search with a threshold and perform a sequential evaluation over a smaller subset of options. This change allowed us to achieve important speedups for the large model AlexNet, and a regular cost over the different δ values, from $\sim 3\times$ to $\sim 7\times$.

It must be mentioned that NetAdapt does have other important execution costs, for example, given by the fine-tuning iterations (T), the evaluation itself to get the candidates' accuracy, or even the interpolations done over the LUT to generate latency values for not measured configurations. With enough computing resources and parallelism (for example, in the form of more GPUs or TPUs), this cost can be reduced to a theoretical cost of simplifying, training, and evaluating only one model.

We can see that this implementation of NetAdapt has opportunities for improvement. In future work, we seek to tackle these problems by, for example, optimizing the interpolation that takes place when the configurations are missing in the LUT, or incorporating other NAS techniques to the search algorithm of NetAdapt.

References

[1] Han Cai, Ligeng Zhu, and Song Han. Proxylessnas: Direct neural architecture search on target task and hardware, 2018.

[2] Bo Chen, Golnaz Ghiasi, Hanxiao Liu, Tsung-Yi Lin, Dmitry Kalenichenko, Hartwig Adam, and Quoc V. Le. MnasFPN: Learning Latency-Aware Pyramid Architecture for Object Detection on Mobile Devices. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 13604–13613, 2020.

[3] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, 2017.

[4] Google. Edge TPU - Run Inference at the Edge — Google Cloud.

[5] John L. Hennessy and David A. Patterson. A New Golden Age for Computer Architecture. *Commun. ACM*, 62(2):48–60, 1 2019.

[6] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, Quoc V. Le, and Hartwig Adam. Searching for MobileNetV3. *CoRR*, abs/1905.02244, 2019.

[7] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *CoRR*, abs/1704.04861, 2017.

[8] Norman P. Jouppi, Cliff Young, Nishant Patil, and David Patterson et al. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, June 2017.

[9] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. CIFAR-10 and CIFAR-100 datasets (Canadian Institute for Advanced Research).

[10] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25*. Curran Associates, Inc., 2012.

[11] Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: differentiable architecture search. *CoRR*, abs/1806.09055, 2018.

[12] Raúl Marichal, Guillermo Toyos, Ernesto Dufrechou, and Pablo Ezzatti. Evaluation of architecture-aware optimization techniques for Convolutional Neural Networks. In Raffaele Montella, Javier García Blas, and Daniele D'Agostino, editors, *31st Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2023, Naples, Italy, March 1-3, 2023*, pages 177–184. IEEE, 2023.

[13] META. MTIA v1: Meta's first-generation AI inference accelerator.

[14] NVIDIA. NVIDIA TensorRT. https://developer.nvidia.com/tensorrt.

[15] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc V. Le, and Alexey Kurakin. Large-Scale Evolution of Image Classifiers. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, page 2902–2911. JMLR.org, 2017.

[16] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V. Le. MnasNet: Platform-Aware Neural Architecture Search for Mobile. *CVPR 2019*, 2018.

[17] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. FBNet: Hardware-Aware Efficient ConvNet Design via Differentiable Neural Architecture Search, 2018.

[18] Tien-Ju Yang. NetAdapt: Platform-Aware Neural Network Adaptation for Mobile Applications - Official Pytorch implementation. https://github.com/denru01/netadapt.

[19] Tien-Ju Yang, Andrew G. Howard, Bo Chen, Xiao Zhang, Alec Go, Mark Sandler, Vivienne Sze, and Hartwig Adam. NetAdapt: Platform-Aware Neural Network Adaptation for Mobile Applications. In Vittorio Ferrari, Martial Hebert, Cristian Sminchisescu, and Yair Weiss, editors,

Computer Vision - ECCV 2018 - 15th European Conference, Munich, Germany, September 8-14, 2018, Proceedings, Part X, volume 11214 of *Lecture Notes in Computer Science*, pages 289–304. Springer, 2018.

[20] Tien-Ju Yang, Yi-Lun Liao, and Vivienne Sze. Netadaptv2: Efficient neural architecture search with fast super-network training and architecture optimization. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2021.

[21] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.