POLITECNICO DI TORINO
Repository ISTITUZIONALE

Efficient Resource-Aware Neural Architecture Search with a Neuro-Symbolic Approach

Original
Efficient Resource-Aware Neural Architecture Search with a Neuro-Symbolic Approach / Bellodi, E., Bertozzi, D., Bizzarri, A., Favalli, M., Fraccaroli, M., Zese, R.. - STAMPA. - (2023), pp. 171-178. (2023 IEEE 16th International Symposium on Embedded Multicore/Many-core Systems-on-Chip Singapore, Singapore 18-21 December 2023) [10.1109/MCSoC60832.2023.00034].

Availability:
This version is available at: 11583/2985835 since: 2025-04-17T08:50:40Z

Publisher:
IEEE Computer society

Published
DOI:10.1109/MCSoC60832.2023.00034

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright
IEEE postprint/Author's Accepted Manuscript

©2023 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collecting works, for resale or lists, or reuse of any copyrighted component of this work in other works.

(Article begins on next page)

29 November 2026

Efficient Resource-Aware Neural Architecture Search with a Neuro-Symbolic Approach

Elena Bellodi
Department of Engineering
University of Ferrara
Ferrara, Italy
elena.bellodi@unife.it

Davide Bertozzi
Computer Science Dept.
University of Manchester
Manchester, UK
davide.bertozzi@manchester.ac.uk

Alice Bizzarri
Department of Engineering
University of Ferrara
Ferrara, Italy
alice.bizzarri@unife.it

Michele Favalli
Department of Engineering
University of Ferrara
Ferrara, Italy
michele.favalli@unife.it

Michele Fraccaroli
Unitec S.p.A.
Lugo, Italy
michele.fraccaroli@unife.it

Riccardo Zese
DOCPAS
University of Ferrara
Ferrara, Italy
riccardo.zese@unife.it

***Abstract*—Hardware-aware Neural Architectural Search (NAS) is gaining momentum to enable the deployment of deep learning on edge devices with limited computing capabilities. Incorporating device-related objectives such as affordable floating point operations, latency, power, memory usage, etc. into the optimization process makes searching for the most efficient neural architecture more complicated, since both model accuracy and hardware cost should guide the search. The main concern with most state-of-the-art hardware-aware NAS strategies is that they propose for evaluation also trivially infeasible network models for the capabilities of the hardware platform at hand. Moreover, previously generated models are frequently not exploited to intelligently generate new ones, leading to prohibitive computational costs for practical relevance. This paper aims to boost the computational efficiency of hardware-aware NAS by means of a neuro-symbolic framework revolving around a Probabilistic Inductive Logic Programming module to define and exploit a set of symbolic rules. This component learns and refines the probabilities associated with the rules, allowing the framework to adapt and improve over time, thus quickly narrowing down the search space toward the most promising neural architectures.**

I. Introduction

Deep learning is a dominant machine learning technique that is pushing the frontiers of a growing number of high-impact applications such as language translation [1], speech recognition [2], speech synthesis [3], image recognition [4] and image synthesis [5].

The current unmatched accuracy that deep learning is achieving in the above and other realms is the outcome of a historical trend where model performance optimization has been pursued as a virtually exclusive goal. This has resulted in the supervised training of complex neural network architectures using large quantities of labeled data. This approach is currently running out of steam.

On the one hand, the computational requirements of deep learning are quickly becoming economically, technically and environmentally unsustainable as limits are stretched, thus calling for less computationally-intensive methods of improvement [6]. On the other hand, as deep learning is gaining momentum for extracting meaningful information from raw sensor data from Internet-of-Things devices, matching its computational requirements with the tight resource budgets of such devices is becoming a daunting challenge. More specifically, edge computing platforms vary hugely in terms of their computation capabilities, memory capacities or power budgets, as well as performance targets. Thus, a neural network needs to be optimized for the hardware platform it will be deployed on since no-one-model fits all.

This context motivates a renewed surge of interest in Network Architecture Search (NAS) approaches. NAS has been long investigated as an effective way of overcoming manually-designed models through automatic generation [7]. However, while considerable effort has been historically devoted to accuracy optimization, hardware constraints are increasingly incorporated into multi-objective search strategies to fit the resource budgets of the inference platform at hand [8]–[11]. The most advanced approaches target holistic optimization encompassing both the hardware platform and the model architecture [12]–[16].

To date, the most daunting challenge common to NAS frameworks with different degrees of hardware awareness consists of the prohibitive computational cost for exploring a large and complex search space. Specifically, a NAS is composed of three main elements: *Search Space* (i.e., all possible architectures that can be generated during the optimization process), *Search Strategy* (the methods to explore the Search Space), and *Performance Estimation Strategy* (the possible methods for measuring the quality of the generated models) [17]. There are different search strategies that can be used to explore the search space of neural architectures: random search, Bayesian Optimization [18], reinforcement learning [19], gradient-based methods [20] and evolutionary algorithms [17], [21]. The main problem of these strategies is that they propose for evaluation also trivially infeasible network models for the hardware platform at hand. Moreover, many of them do not exploit the previously generated models

to intelligently generate new ones, leading to a more time- and resource-consuming search.

The main goal of this paper is to boost the computational efficiency of hardware-aware NAS through an adaptive search strategy that accounts for previous experience during the generation of new model architectures. The new approach revolves around Symbolic DNN-Tuner [22], [23], a neuro-symbolic framework [24] driving the training of a DNN, using the symbolic part to analyse the performance of each training experiment and to guide the choice of Hyper-Parameters (HPs) to obtain a network with better performance. In particular, the symbolic part uses a Probabilistic Inductive Logic Programming [25] module to define and exploit a set of *symbolic rules*. This component learns and refines the probabilities associated with the rules, allowing the system to adapt and improve over time. This adaptive learning capability increases the system's effectiveness in exploring the search space over time, and finding neural architectures that satisfy hardware constraints.

Without lack of generality, hardware awareness is gained through a commonly-used constraint on the number of floating point operations (FLOPS) that is affordable for the computing platform at hand [9]. More accurate models of the underlying hardware (e.g., encompassing latency and power) are left for future work, while the emphasis of this paper is on demonstrating the potential of the new NAS approach. To the best of our knowledge, this is the first time the problem of hardware-aware NAS is solved with a rule-based learning component based on probabilistic logic languages.

Last but not least, the use of logical rules makes the system strongly declarative as their meaning is clear to the user. In this way, operations like additions or updates of rules can be easily done, making the system easily extensible. As a result, this work is the the first step towards an extensible, declarative, and intelligent hardware-aware NAS approach, which can possibly be used, thank to its declarativeness, by users not expert in the deep learning domain.

The proposed NAS framework is validated through a comparison with the Efficient Neural Architecture Search (ENAS) [19] implemented in Autokeras[1] [18], one of the most widely used open source NAS frameworks, which was extended in order to incorporate the FLOPS constraint. Autokeras efficiently explores the search space by developing a neural network kernel and a tree-structured acquisition function optimization algorithm. It integrates network morphism [26] and utilizes Bayesian Optimization (BO) [27] to move efficiently through the search space by selecting the most promising operations each time.

When comparing the proposed NAS framework with Autokeras to synthesize neural architectures for a set of hardware platforms with increasing compute capabilities, the former takes on average 78% less time while achieving an inference accuracy that is 31% higher.

The paper is organized as follows: Section II discusses the related works. Section III presents the Neural Architecture Search and the Autokeras framework, while Section IV presents the proposed approach to NAS. Finally, Section V illustrates the experiments while Section VI concludes the paper.

[1]https://autokeras.com/

II. Related works

In this section, we review previous work that addresses the NAS problem with physical constraints, i.e., the search for optimal neural architectures while satisfying time, resource, or accuracy constraints.

Constrained NAS arises as a crucial challenge in the optimization of neural architectures. Several studies have proposed innovative approaches to achieve a balance between the search for high-performance architectures and the imposition of computational constraints.

In [28], the authors present a NAS system for micro-controllers where the search space is being reduced from respecting physical constraints due to limited resources, and is explored by a multi-objective function. This function takes into account constraints such as the maximum memory used, storage space required, and time efficiency. Two search algorithms are discussed in [28]: Aging Evolution (AE) and Bayesian Optimization (BO). The former performs a local search by applying a random morphism at each iteration, i.e. by introducing random changes or modifications to the architecture, such as adding or removing layers, changing layer sizes, or altering the connectivity patterns between layers. The latter approximates the target function using a surrogate model and guides the search through unexplored regions of the search space.

In [29] the authors propose a progressive search strategy to identify Pareto-optimal neural architectures while considering constraints of the target device. The approach relies on an initial pool of candidate architectures and iteratively selects promising solutions to generate new architectures, optimizing the trade-off between accuracy and computational resources. Other approaches popular in the literature are NAS using Reinforcement Learning (RL) with various physical constraints, i.e. computational resources, memory and latency constraints [8], [9], [30], and also energy efficiency constraints [9].

Monas [30] combines RL with a multi-objective formulation to explore the trade-off between multiple objectives, resulting in Pareto-optimal architectures. Mnasnet [8] incorporates RL for mobile-specific architecture search, considering platform awareness and resource constraints. RENA [9] uses RL techniques to optimize neural architectures based on resource efficiency, enabling the discovery of efficient models.

These RL-based NAS approaches demonstrate the effectiveness of reinforcement learning in guiding the search process, considering various goals and constraints specific to different application scenarios.

Multi-target genetic algorithms are also utilized in the literature. In [31], the power of genetic algorithms is harnessed to explore the search space and optimize multiple objectives simultaneously. Specifically, it aims to find solutions that satisfy a combination of goals, generating a set of high-quality

neural architectures that represent the Pareto front of optimal solutions. NSGA-Net [31] provides an effective alternative for optimizing neural architectures by considering multiple objectives concurrently, typically encompassing model accuracy, required computational resources (such as memory or runtime), and other desired performance metrics. The same multi-target optimization evolutionary engine is used in [32] that makes a neural network population to evolve from a pre-trained starting super-net to smaller networks satisfying optimization targets for specific data-sets.

In the context described above, our approach distinguishes itself by using a neural-symbolic framework based on symbolic tuning rules that guide the search in the search space. The set of rules allows an easier addition and modification of the search strategy and the management of constraints for the search space.

III. Background

A. Neural Architecture Search

NAS is a technique for automating the design of DNN architectures. It is strictly correlated to AutoML [33]. The three main elements that compose a NAS are *Search Space, Search Strategy* and *Performance Estimation Strategy*. *Search Space* refers to the space of all possible architectures that can be generated by the NAS and strictly depends on the hyper-parameters (HP) set for the model search. *Search Strategy* refers to the methods to explore the search space with the canonical exploration-exploitation trade-off. *Performance Estimation Strategy* refers to the methods to measure the performance of the built neural network [17]. Given a *Search Space*, there are many *Search Strategies* that can be used to explore this space. These strategies include: random search, Bayesian Optimization (BO) [18], reinforcement learning [19], gradient-based methods [20] and evolutionary algorithms like genetic algorithms [17], [21]. We can group NAS in two branches: the standard and the one-shot NAS [34]. Standard NAS follows the traditional search approach also used by Grid Search, where each generated DNN runs as an independent train. One-shot NAS uses weight sharing among models in neural architecture search space to train a super-net and uses this to select better models. This type of algorithm reduces computational resources compared to the classical NAS algorithm. The state-of-the-art of one-shot NAS are: Efficient Neural Architecture Search (ENAS) [19], Differentiable Architecture Search (DARTS) [20], Single Path One-Shot (SPOS) [35] and ProxylessNAS [36].

B. The Benchmark: Autokeras

Many of the mentioned NAS approaches require a huge amount of time and computational power to reach good results. Reducing the computational complexity of NAS approaches is thus becoming a critical research area. Network morphism, which keeps the functionality of a neural network while changing the model architecture [26], is one of the most promising approaches and is thus considered in this paper as a benchmark. Autokeras is a state-of-the-art open-source software that integrates network morphism: the key idea is to explore the search space via morphing the neural architectures (e.g., inserting a layer or adding a skip-connection) guided by the BO algorithm. This algorithm iteratively runs three steps: trains the underlying Gaussian process model with the existing architectures and their performance, generates the next architecture to be observed by optimizing a delicately defined acquisition function and obtains the actual performance by training the generated neural architecture.

For the sake of homogeneous comparison with the proposed NAS framework, Autokeras' flexibility has been exploited to incorporate the FLOPS constraint into the architecture search process. More specifically, the modification made to Autokeras consists of calculating the number of FLOPS of any new generated DNN and discarding it in case it exceeds the FLOPS limit, forcing the algorithm to regenerate another one. This process is iterated until a network is generated that meets the imposed limits.

The choice of Autokeras as a validation benchmark is due to two main factors. First, Autokeras is one of the most widely used open source NAS. Second, it is strongly connected with Tensorflow, the framework used by Symbolic DNN-Tuner to manage neural networks. Many other related systems are based on different neural network frameworks or are not implemented with the aim of making them easily extensible as Autokeras. We believe that in order to compare two systems fairly, it is necessary to put them both under the same conditions and be sure that these conditions are optimal for both. With this in mind, considering two systems having the same underlying framework for neural networks makes it possible to obtain directly comparable results.

IV. Proposed Neuro-Symbolic NAS

A. Symbolic DNN-Tuner

The approach proposed by this paper leverages "Neuro-symbolic Artificial Intelligence" technology. Neuro-symbolic AI refers to a field of research and applications that combines machine learning methods based on artificial neural networks, such as deep learning, with symbolic approaches to computing and AI, as can be found for example in the AI subfield of knowledge representation and reasoning. Central to our approach is the extension of a state-of-the-art neuro-symbolic framework named Symbolic DNN-Tuner [22], [23] to incorporate hardware-level objectives in the search strategy.

In short, Symbolic DNN-Tuner drives the training of a DNN by analysing the performance of each training experiment and automating the choice of Hyper-Parameters (HPs) to obtain a network with better performance. This system exploits Probabilistic Logic Programming (PLP) to analyse the network's performance and exploits probabilistic symbolic rules to drive the choice of both HPs and the network architecture.

More specifically, Symbolic DNN-Tuner can be seen as a symbolic Hyper-Parameters Optimization (HPO) algorithm or a symbolic NAS with HPs optimization based on BO. This framework combines an automatic tuning approach with some tricks usually used in manual approaches [37]. For the

automatic approach, we use BO [27]. This choice is motivated by the fact that tuning DNNs is computationally expensive and the BO algorithm limits the evaluations of the objective function (in this case, DNNs training and validation) by more intelligently choosing the next set of HPs values.

The techniques used in the manual approach are mapped to non-deterministic and probabilistic Symbolic Tuning Rules (STRs). These tuning rules are designed to edit the HP search space, add new HPs, or update the network structure. Each STR encapsulates a Tuning Action (TA) associated with a problem like overfitting, underfitting, or wrong learning rate values. The activation of these rules allows one to avoid such network problems and to guide the entire learning process to better results.

Symbolic DNN-Tuner is composed of two parts: a Neural Block that contains the neural network, the HPs search space and the application of the TAs, and a Symbolic Block that diagnoses problems and identifies the (most probable) TA to be applied on the network architecture based on the network performance and computed metrics after each training. The whole execution pipeline of Symbolic DNN-Tuner and its blocks is shown in Figure 1.

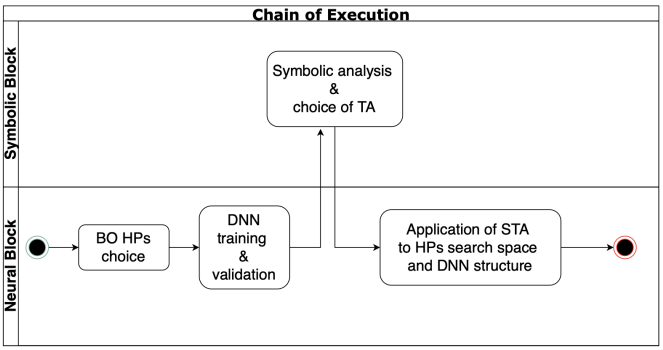

Fig. 1. Symbolic DNN-Tuner execution pipeline with the *Neural Block* and the *Symbolic Block*.

An example of STRs is shown in Figure 2. Each STR is triggered depending on the problem detected on the network and defines a possible TA to do to solve the detected problem. A probability value associated with each STR determines the probability of application of its TA, in case the associated problem is diagnosed. For example, the first rule proposes to perform data augmentation in case of overfitting, while the second to decrease the learning rate in case of underfitting.

The probabilistic weights are learned from the experience (called *evidence*) gained from previous iterations. This *experience* becomes the set of *training examples* for a "Learning from Interpretation (LFI)" program [25], a kind of Inductive Logic Programming. Then, the LFI program is composed of two parts: the program and the evidence, as shown in Figure 2. The program contains the actions to apply with the corresponding probability value learned during the tuning. The evidence tells whether the application of a specific action improved the model performance. Accumulating evidence forms the LFI training set that is used to tune the probabilities in the program.

```
% Rules
0.7::action(data_augment):-
        problem(overfitting).
0.3::action(decr_lr):-
        problem(underfitting).
...
% ============ LFI Program ============ %
% Program
t(0.5)::action(data_augment).
t(0.2)::action(decr_lr).
...
% - - - - - - - - - - - - - - - - - - - - %
% Evidence
evidence(action(data_augment), True).
evidence(action(decr_lr), False).
```

Fig. 2. Symbolic Tuning Rules in the Symbolic Block of Symbolic DNN-Tuner.

B. Incorporating Hardware Objectives

The optimization process of Symbolic DNN-Tuner is based on the optimization performed by BO. Differently from the original tool, in this work these optimization methods will be used to optimize the neural architecture with respect to both network performance *and* device-related objectives.

More specifically, we considered the number of FLOPS (FLOating Point OperationS) as a constraint to fulfil during the tuning of the model. It is an approximate metric that quantifies the available computing power in the underlying execution platform.

Similar to other NAS approaches [9], in this early stage of tool development the goal is not to model hardware cost in the most precise way, but rather to show that when approximate metrics are considered the proposed NAS can efficiently optimize them. This leads to a precise cost quantification in terms of the fundamental operations and to easily interpretable results without an excessive specialization of the NAS framework for the hardware platform at hand.

Overall, the aim is to maximize model accuracy and obtain a network architecture to be deployed in resource-constrained platforms with a number of FLOPS as close as possible to a predetermined threshold. Hence, the focus of BO is on solving the problem:

$$\min_{x \in D} f_{flops} \tag{1}$$

where f_{flops} is the objective function, the input x is in $\mathbb{R}^d$, d is the number of HPs and D is the search space which can be seen as a hyper-cube where each dimension is a hyper-parameter. In detail, f_{flops} is

$$f_{flops} = -|Acc_i - |FLOPS_{th} - FLOPS_i| \times \epsilon| \tag{2}$$

with Acc_i and $FLOPS_i$ the accuracy and the number of FLOPS of the ith model respectively, and $FLOPS_{th}$ the

```
0.4::action(dec_neurons,thr_exceeded):-
        problem(thr_exceeded).
0.7::action(dec_layers,thr_exceeded):-
        problem(thr_exceeded).
```

Fig. 3. STRs for imposing the constraint on the number of FLOPS.

threshold of the FLOPS that the hardware can manage. ϵ is an HP used to balance and adjust the contribution of the calculated gap between the FLOPS on the objective function. ϵ has been empirically set to 0.33 performing parameter sweep. $FLOPS_i$ and $FLOPS_{th}$ are normalized between 0 and 1.

In order for Symbolic DNN-Tuner to optimize DNNs taking into account the FLOPS constraint, it is necessary to create a new Symbolic Tuning Rule that activates when the created DNN exceeds the FLOPS limit. We defined a new `problem(thr_exceeded)`, occurring when the number of FLOPS exceeds the threshold, and two new Tuning Actions. The first TA decreases the number of neurons in the various layers and the second one removes the last convolutional layer and any other layer related to it (for example, pooling layer, dropout etc.). This is to maintain consistency in the neural network architecture after removing the convolutional layer. Figure 3 shows the two new STRs with their respective probabilistic weights. These weights are randomly initialized and adaptively tuned during the execution of the system.

In the extended Symbolic DNN-Tuner, the designer knowledge is embedded in the rules used at each iteration to mutate a HP that is likely to be a problem in the current network model. This is a very different approach with respect to Autokeras, where the network is mutated in a way that is unaware of its possible problems. As it is will be shown in the experimental results, this approach improves the growth rate of the quality of produced solutions.

V. Experimental Results

We evaluated the extended Symbolic DNN-Tuner and the modified version of AutoKeras with matched hardware awareness on image classification problems using the CIFAR10 dataset [38]. Both frameworks were put at work in 8 different experiments, distinguished by various and progressively relaxed FLOPS thresholds to be met: 10M, 30M, 40M, 80M, 90M, 100M, 110M, and 120M FLOPS. This experimental campaign corresponds to the design space exploration that designers of edge devices may perform to characterize the accuracy-cost trade-off: high-end neural network models are likely to result in violations of the resource budget, while small models would not provide sufficient levels of accuracy.

A. Results

The experiments with both Autokeras and Symbolic DNN-Tuner demonstrate two very different behaviours. In the former, the exploration phase produces many neural networks achieving very low accuracy. Also, the computation of the accuracy tipically requires a long time as the network must be trained. On the other hand the latter, thanks to an intelligent exploration, can find a good model in the very beginning that is then improved by fine-tuning the hyper-parameters. This behaviour is shown for both systems in Figure 4. Each $\times$ represents the accuracy (y-axis) for a given value of FLOPS (x-axis) for every model trained by Autokeras, while each circle represents the same for Symbolic DNN-Tuner. The colors of $\times$ and circles correspond to the FLOPS thresholds, represented by vertical lines. As can be seen, $\times$ are more sparse, showing that very different models are tested by Autokeras. A large number of $\times$ are placed in the lower portion of the graph, corresponding to models with low accuracy. Symbolic DNN-Tuner finds a good model and gradually improves that model, instead of trying very different architectures. The accuracy trend of the two frameworks for each FLOPS threshold is highlighted in Figure 5: Symbolic DNN-Tuner always exceeds Autokeras, with an average improvement of 31%.

Execution times are displayed in Figure 6: for every threshold Symbolic DNN-Tuner is the fastest system, with an average improvement that is as large as 78%. This is because of the previously mentioned Autokeras behaviour: its execution time is unnecessarily increased by the construction and training of networks that have a high probability of never reaching the optimum. This behaviour demonstrates one of the main capabilities of the Symbolic DNN-Tuner: the symbolic block enables one to adaptively change the search space, saving time by dynamically excluding HPs values that would move the optimization away from a potential optimum.

In the context of this work, it is interesting to see the symbolic analysis made by Symbolic DNN-Tuner and which Tuning Rules were activated. Tables I, II, III, IV show the diagnosis and TAs used by Symbolic DNN-Tuner for the eight experiments. In bold one can see every time that the symbolic analysis detected when the threshold of the maximum FLOPS had been exceeded. As also highlighted in Figure 7, the FLOPS threshold is exceeded in the range 10-90M FLOPS. From 100M FLOPS onwards this phenomenon was not detected anymore. This behaviour is also visible in Figure 4, where there are dots for Symbolic DNN-Tuner presenting a number of FLOPS higher than the fixed threshold (e.g., two red dots for the 800M threshold are placed around 100M and 130M FLOPS). We could interpret this result as an indication of the basic size (or capacity) of the DNN: networks allowing less than 10M FLOPS are not usable on this dataset or for this specific experiment. DNNs with more than 90M may work best for this task. Thus, we can tackle the problem of how to find the best dimensioning of a DNN given a dataset.

VI. Conclusions

This paper leverages neuro-symbolic artificial intelligence technology for efficient hardware-aware NAS, which quickly guides the learning process toward optimized neural architectures in terms of both network performance and device-related objectives. The framework consistently and significantly outperforms a popular open-source Automated Machine Learning system based on Keras with matched hardware awareness

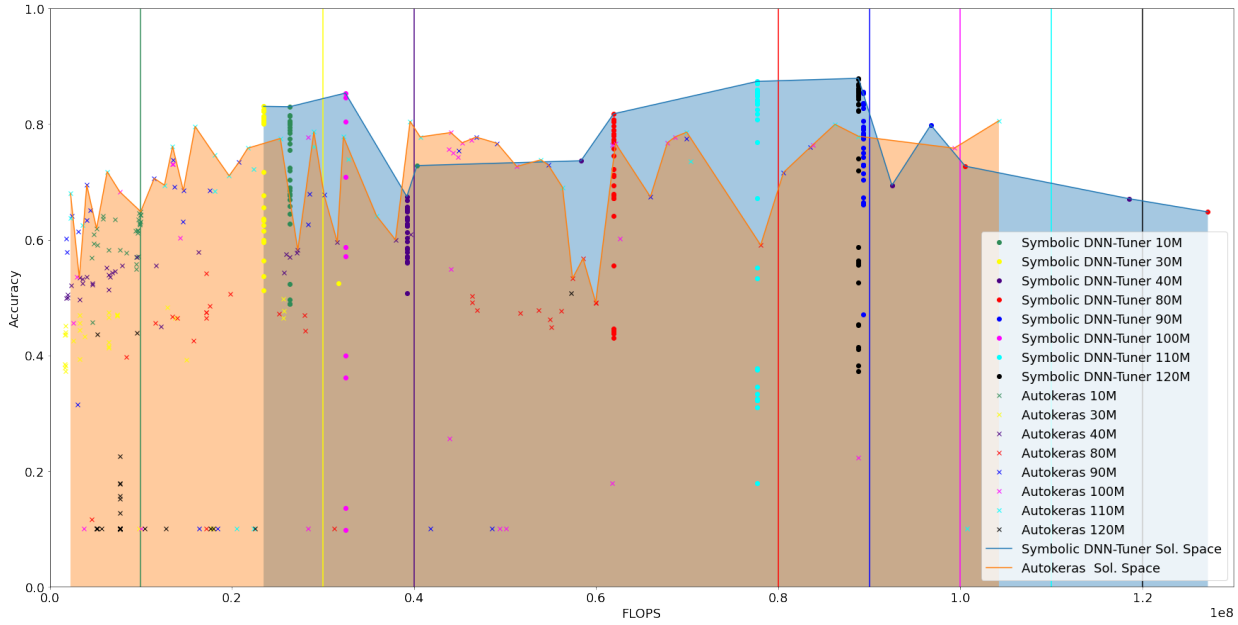

Fig. 4. Comparison between Symbolic DNN-Tuner (blue area) and Autokeras (orange area) accuracy at various FLOPS thresholds. MFLOPS are represented on the x-axis, and the accuracy obtained by the various models is shown on the y-axis. The FLOPS set for each experiment are represented by vertical lines and reported in the legend. Models generated by Symbolic DNN-Tuner and Autokeras are indicated with a $\times$ and a circle, respectively.

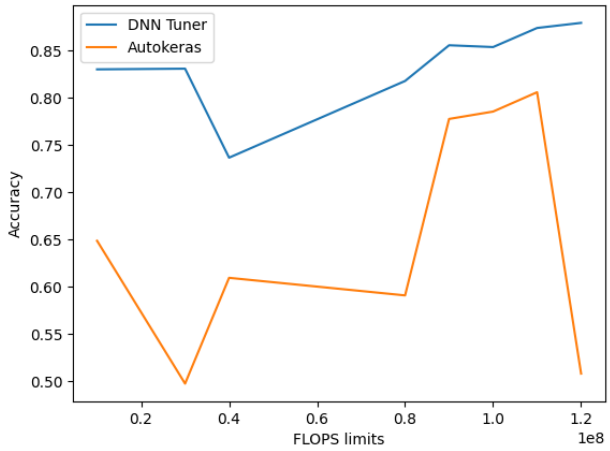

Fig. 5. Comparison of the accuracy of Autokeras (orange line) and Symbolic DNN-Tuner (blue line), both of which were tested at various FLOPS thresholds. The thresholds are represented on the x-axis, and the accuracy attained by the best models is shown on the y-axis.

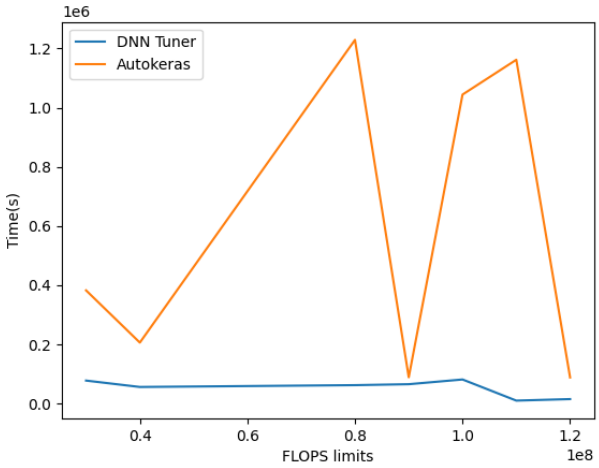

Fig. 6. Autokeras (orange line) and Symbolic DNN-Tuner (blue line) time comparisons at various FLOPS thresholds. The thresholds are shown on the x-axis, and the time taken by each framework to complete 30 iterations with a given FLOPS constraint is shown on the y-axis.

when exploring neural architectures for a range of hardware platforms with increasing compute capabilities.

The proposed approach to NAS comes with two distinctive benefits. On the one hand, the probabilistic weights of tuning actions are learned from the experience gained in previous iterations. This enables to quickly narrow down the search space toward the most promising solutions, thus cutting down on the computation cost associated with the evaluation of trivially-infeasible solutions for the capability of underlying hardware. On the other hand, the declarative nature of the tool makes it easily extendible, with the specification of logic rules and tuning actions that are potentially within reach of non-experts in the deep learning domain as well. In future work, the tool will be further extended to gain deeper awareness of the underlying hardware through refined cost models of heterogeneous computing platforms.

TABLE I
DIAGNOSIS & TUNING WITH 10M AND 30M FLOPS RESPECTIVELY (ACRONYMS OF THE TUNING RULES ARE AVAILABLE IN TABLE 2 OF [22] AND FIGURE 3)

Step	10M FLOPS		30M FLOPS	
	Diagnosis	Tuning rule	Diagnosis	Tuning rule
1	reg_l2 data_augmentation inc_lr dec_layers	overfitting overfitting low_lr **thr_exceeded**	inc_lr dec_layers	low_lr **thr_exceeded**
2	inc_lr inc_batch_size dec_neurons	low_lr floating_loss **thr_exceeded**	reg_l2 data_augmentation inc_lr	overfitting overfitting low_lr
3	inc_lr inc_batch_size dec_neurons	underfitting floating_loss **thr_exceeded**	reg_l2 data_augmentation decr_lr inc_batch_size	overfitting overfitting underfitting floating_loss
4	inc_lr inc_batch_size dec_neurons	underfitting floating_loss **thr_exceeded**	decr_lr	underfitting
5	inc_lr inc_batch_size dec_neurons	low_lr floating_loss **thr_exceeded**	decr_lr	underfitting

TABLE II
DIAGNOSIS & TUNING WITH 40M AND 80M FLOPS RESPECTIVELY (ACRONYMS OF THE TUNING RULES ARE AVAILABLE IN TABLE 2 OF [22] AND FIGURE 3)

Step	40M FLOPS		80M FLOPS	
	Diagnosis	Tuning rule	Diagnosis	Tuning rule
1	reg_l2 data_augmentation inc_lr dec_layers	overfitting overfitting low_lr **thr_exceeded**	decr_lr dec_layers	underfitting **thr_exceeded**
2	decr_lr inc_batch_size dec_layers	underfitting floating_loss **thr_exceeded**	reg_l2 data_augmentation decr_lr dec_layers	overfitting overfitting underfitting **thr_exceeded**
3	decr_lr dec_layers	underfitting **thr_exceeded**	decr_lr inc_batch_size dec_layers	underfitting floating_loss **thr_exceeded**
4	decr_lr inc_batch_size	underfitting floating_loss	decr_lr	underfitting
5	decr_lr inc_batch_size	underfitting floating_loss	decr_lr inc_neurons	underfitting underfitting

TABLE III
DIAGNOSIS & TUNING WITH 90M AND 100M FLOPS RESPECTIVELY (ACRONYMS OF THE TUNING RULES ARE AVAILABLE IN TABLE 2 OF [22] AND FIGURE 3)

Step	90M FLOPS		100M FLOPS	
	Diagnosis	Tuning rule	Diagnosis	Tuning rule
1	reg_l2 data_augmentation inc_lr dec_layers	overfitting overfitting low_lr **thr_exceeded**	decr_lr	underfitting
2	reg_l2 data_augmentation decr_lr inc_batch_size	overfitting overfitting underfitting floating_loss	decr_lr inc_batch_size	underfitting floating_loss
3	decr_lr decr_lr	underfitting floating_loss	reg_l2 data_augmentation decr_lr inc_batch_size	overfitting overfitting underfitting floating_loss
4	reg_l2 data_augmentation decr_lr	overfitting overfitting floating_loss	decr_lr inc_batch_size	underfitting floating_loss
5	decr_lr decr_lr	underfitting floating_loss	reg_l2 data_augmentation decr_lr inc_batch_size	overfitting overfitting underfitting floating_loss

TABLE IV
DIAGNOSIS & TUNING WITH 110M AND 120M FLOPS RESPECTIVELY (ACRONYMS OF THE TUNING RULES ARE AVAILABLE IN TABLE 2 OF [22] AND FIGURE 3)

Step	110M FLOPS		120M FLOPS	
	Diagnosis	Tuning rule	Diagnosis	Tuning rule
1	reg_l2 data_augmentation decr_lr	overfitting overfitting underfitting	reg_l2 data_augmentation dec_lr inc_batch_size	overfitting overfitting underfitting floating_loss
2	reg_l2 inc_dropout inc_lr inc_neurons inc_batch_size	overfitting overfitting low_lr underfitting floating_loss	reg_l2 inc_dropout inc_neurons decr_lr	overfitting overfitting underfitting floating_loss
3	reg_l2 inc_dropout inc_neurons	overfitting overfitting underfitting	reg_l2 inc_dropout inc_neurons	overfitting overfitting underfitting
4	reg_l2 inc_dropout inc_neurons inc_batch_size	overfitting overfitting underfitting floating_loss	reg_l2 inc_dropout decr_lr inc_neurons	overfitting overfitting floating_loss underfitting
5	inc_lr inc_neurons	low_lr underfitting	inc_lr inc_neurons	low_lr underfitting

REFERENCES

[1] Yonghui Wu et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016.

[2] Dario Amodei et al. Deep speech 2: End-to-end speech recognition in english and mandarin. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, page 173–182. JMLR.org, 2016.

[3] Aäron van den Oord et al. Parallel wavenet: Fast high-fidelity speech synthesis. *ArXiv*, abs/1711.10433, 2018.

[4] Hanxiao Liu, Karen Simonyan, Oriol Vinyals, Chrisantha Fernando, and Koray Kavukcuoglu. Hierarchical representations for efficient architecture search. *CoRR*, abs/1711.00436, 2017.

[5] Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. Progressive growing of gans for improved quality, stability, and variation. *CoRR*, abs/1710.10196, 2017.

[6] Neil C. Thompson, Kristjan H. Greenewald, Keeheon Lee, and Gabriel F. Manso. The computational limits of deep learning. *CoRR*, abs/2007.05558, 2020.

[7] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *The Journal of Machine Learning Research*, 20(1):1997–2017, 2019.

[8] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 2820–2828, 2019.

[9] Yanqi Zhou, Siavash Ebrahimi, Sercan Ö Arık, Haonan Yu, Hairong Liu, and Greg Diamos. Resource-efficient neural architect. *arXiv preprint arXiv:1806.07912*, 2018.

[10] Guihong Li, Sumit K Mandal, Umit Y Ogras, and Radu Marculescu. Flash: F ast neura l a rchitecture s earch with h ardware optimization.

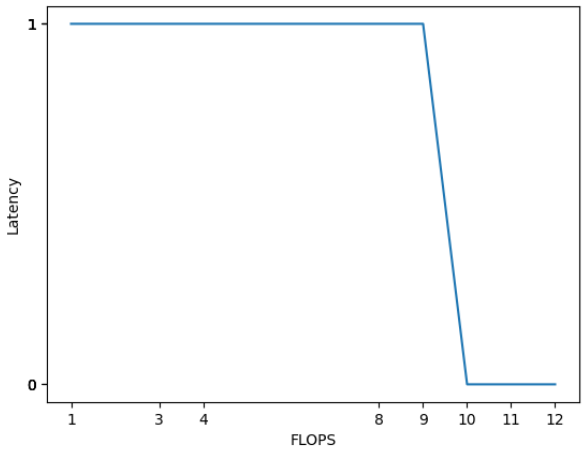

Fig. 7. Cases where FLOPS threshold is exceeded in the various experiments. The x-axis reports the FLOPS thresholds and the y-axis if the threshold is exceeded (1) or not (0).

ACM Transactions on Embedded Computing Systems (TECS), 20(5s):1–26, 2021.

[11] 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. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*, pages 10734–10742. Computer Vision Foundation / IEEE, 2019.

[12] Kanghyun Choi, Deokki Hong, Hojae Yoon, Joonsang Yu, Youngsok Kim, and Jinho Lee. Dance: Differentiable accelerator/network co-exploration. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 337–342. IEEE, 2021.

[13] Yanqi Zhou, Xuanyi Dong, Berkin Akin, Mingxing Tan, Daiyi Peng, Tianjian Meng, Amir Yazdanbakhsh, Da Huang, Ravi Narayanaswami, and James Laudon. Rethinking co-design of neural architectures and hardware accelerators. *arXiv preprint arXiv:2102.08619*, 2021.

[14] Cong Hao, Xiaofan Zhang, Yuhong Li, Sitao Huang, Jinjun Xiong, Kyle Rupnow, Wen-mei Hwu, and Deming Chen. Fpga/dnn co-design: An efficient design methodology for iot intelligence on the edge. In *Proceedings of the 56th Annual Design Automation Conference 2019*, DAC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[15] Weiwen Jiang, Lei Yang, Edwin Hsing-Mean Sha, Qingfeng Zhuge, Shouzhen Gu, Yiyu Shi, and Jingtong Hu. Hardware/software co-exploration of neural architectures. *CoRR*, abs/1907.04650, 2019.

[16] Weiwen Jiang, Lei Yang, Sakyasingha Dasgupta, Jingtong Hu, and Yiyu Shi. Standing on the shoulders of giants: Hardware and neural architecture co-search with hot start. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11), 2020.

[17] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *arXiv preprint arXiv:1808.05377*, 2018.

[18] Haifeng Jin, Qingquan Song, and Xia Hu. Auto-keras: An efficient neural architecture search system. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 1946–1956, 2019.

[19] Hieu Pham, Melody Y Guan, Barret Zoph, Quoc V Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. *arXiv preprint arXiv:1802.03268*, 2018.

[20] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018.

[21] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search. In *Proceedings of the aaai conference on artificial intelligence*, volume 33, pages 4780–4789, 2019.

[22] Michele Fraccaroli, Evelina Lamma, and Fabrizio Riguzzi. Symbolic dnn-tuner. *Machine Learning*, 111(2):625–650, 2022.

[23] Michele Fraccaroli, Evelina Lamma, and Fabrizio Riguzzi. Symbolic dnn-tuner: a python and problog-based system for optimizing deep neural networks hyperparameters. *SoftwareX*, 17:100957, 2022.

[24] Artur S. d'Avila Garcez, Tarek R. Besold, Luc De Raedt, Peter Földiák, Pascal Hitzler, Thomas F. Icard, Kai-Uwe Kühnberger, L. Lamb, Risto Miikkulainen, and Daniel L. Silver. Neural-symbolic learning and reasoning: Contributions and challenges. In *AAAI Spring Symposia*, 2015.

[25] Bernd Gutmann, Ingo Thon, and Luc De Raedt. Learning the parameters of probabilistic logic programs from interpretations. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 581–596. Springer, 2011.

[26] Tao Wei, Changhu Wang, Yong Rui, and Chang Wen Chen. Network morphism. In *International conference on machine learning*, pages 564–572. PMLR, 2016.

[27] Ian Dewancker, Michael McCourt, and Scott Clark. Bayesian optimization primer, 2015.

[28] Edgar Liberis, Łukasz Dudziak, and Nicholas D Lane. μnas: Constrained neural architecture search for microcontrollers. In *Proceedings of the 1st Workshop on Machine Learning and Systems*, pages 70–79, 2021.

[29] Jin-Dong Dong, An-Chieh Cheng, Da-Cheng Juan, Wei Wei, and Min Sun. Dpp-net: Device-aware progressive search for pareto-optimal neural architectures. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 517–531, 2018.

[30] Chi-Hung Hsu, Shu-Huan Chang, Jhao-Hong Liang, Hsin-Ping Chou, Chun-Hao Liu, Shih-Chieh Chang, Jia-Yu Pan, Yu-Ting Chen, Wei Wei, and Da-Cheng Juan. Monas: Multi-objective neural architecture search using reinforcement learning. *arXiv preprint arXiv:1806.10332*, 2018.

[31] Zhichao Lu, Ian Whalen, Vishnu Boddeti, Yashesh Dhebar, Kalyanmoy Deb, Erik Goodman, and Wolfgang Banzhaf. Nsga-net: neural architecture search using multi-objective genetic algorithm. In *Proceedings of the genetic and evolutionary computation conference*, pages 419–427, 2019.

[32] Zhichao Lu, Gautam Sreekumar, Erik Goodman, Wolfgang Banzhaf, Kalyanmoy Deb, and Vishnu Naresh Boddeti. Neural architecture transfer. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 43(9):2971–2989, 2021.

[33] Matthias Feurer and Frank Hutter. Towards further automation in automl. In *ICML AutoML workshop*, page 13, 2018.

[34] Gabriel Bender, Pieter-Jan Kindermans, Barret Zoph, Vijay Vasudevan, and Quoc Le. Understanding and simplifying one-shot architecture search. In *International Conference on Machine Learning*, pages 550–559, 2018.

[35] Zichao Guo, Xiangyu Zhang, Haoyuan Mu, Wen Heng, Zechun Liu, Yichen Wei, and Jian Sun. Single path one-shot neural architecture search with uniform sampling. *arXiv preprint arXiv:1904.00420*, 2019.

[36] Han Cai, Ligeng Zhu, and Song Han. Proxylessnas: Direct neural architecture search on target task and hardware. *arXiv preprint arXiv:1812.00332*, 2018.

[37] Grégoire Montavon, Geneviève Orr, and Klaus-Robert Müller. *Neural networks: tricks of the trade*, volume 7700. springer, 2012.

[38] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.