Neural Architecture Search and tinyML: A Survey

Daniel Duclos-Cavalcanti

***Abstract*—There is no denying the increasing success that Deep Neural Networks (DNNs) have displayed across various tasks such as image classification, speech recognition, machine translation and many others. This progress can be attributed largely due to *architecture engineering*, which is a process that requires immense domain expertise, intuition and some amount of trial and error. This is not ideal as it is both time consuming and error-prone. Neural Architecture Search (NAS) rises as the next logical step aiming to both automate architecture discovery and further the understanding of the inner workings of DNNs. This field has grown remarkably within the last 5 years and among the many challenges of NAS, there are concerns regarding computational costs and time feasability. This however changes within the context of *tinyML*, which is an expanding field at the intersection of machine learning and embedded systems. Due to the resource constrained conditions involved in most embedded devices, various techniques have been developed to reduce size and memory consumption of deep learning models by the tinyML community. Due to that, the stages concerning training and performance evaluation become substantially faster in comparison to the models used in NAS research. Thus, presenting an interesting opportunity to explore NAS applications within a different paradigm. Throughout this work, an overview of existing research in NAS, specifically concerned with the use of evolutionary algorithms methods will be presented, as well as highlighting relevant applicabilities to tinyML.**

***Index Terms*—Neural Architecture Search (NAS), evolutionary computation (EC), Deep Learning, tinyML.**

I. Introduction

The advancement of deep learning, although extremely beneficial, has also caused a continuous demand for architecture design. This coupled with a growing model complexity, demands ample time and expert knowledge for any individual to not only benefit from it's application, but also be able to improve any given architecture.

After the work in [1] proposed by Google, it was shown for the first time that NAS algorithms have the potential to find models that rival the current state of the art. However, done so in an automated fashion, minimizing human participation. Since then, many different methods have appeared.

To better visualize NAS and understand the difference between its different methods, one can categorize it within three dimensions [2]:

- **Search Space**: defines all the possible architectures that can in principle be considered.
- **Search Strategy**: defines how the *search space* is explored by the algorithm.
- **Performance Estimation Strategy**: defines how performance is evaluated at every architecture iteration.

A. Search Space

NAS is an optimization problem, whose search space is the defining factor to its complexity. The smaller the search space is, the faster the search may converge, as well as requiring less computational resources. This comes at the cost of less freedom to explore unseen architectures and also possibly limiting the complexity of the design.

The simplest approach is to define an architecture as a *chain-structured neural network*, which essentially consists of a sequence of layers whose inputs are the output of their preceding layer. In this case the space is parametrized by maximum number of layers, type of operations per layer, and hyperparameters conditioned by the chosen operation. An illustration to this can be seen in Fig. 1.

The next step consists then of including more modern design elements such as skip connections, which has already been seen in [1] and [3]. This allows to build more complex *multi-branch networks*, which cannot be described as simple sequential layer chaining, but as a structure where each layer's input is a function of previous layers outputs. This increases significantly the degrees of freedom of architecture design, which leads to a much larger search space. An illustration to this can be seen in Fig. 1. A chain-structured neural network is then a special case of the multi-branch network.

Another predominant trend includes the search within *cells* or *blocks*, initially considered by works such as [4] and [5]. What is proposed, is to break-off architectures in cells, such that the search space is then designated within a single cell per time. This drastically reduces the search space, as there are substantially less layers within a cell in comparison to an entire architecture. Additionally, subdivision of architectures into units is considered a good design practice, which also enables easy transferability to other data sets. The same modelling used on multi-branch networks can be used with cells, simply replacing layers with cell architectures.

This drives then the discussion between *micro-architectures* versus *macro-architectures*. The macro architecture attempts to determine how cells should be connected and how many are needed to build a model. On the other hand, the micro architecture aims to find the optimal structure for each cell. Ideally, both viewpoints should be optimized jointly, which of course leads to a complex search space. There have been efforts that aimed to minimize this endeavor by fixing macro-architectures with known working topographies such as in [6] with DenseNet [7]. This practice, dubbed as *human knowledge injection* attempts to reduce the search space through applying domain expertise known to obtain effective results. This includes human bias in the model.

B. Search Strategy

As any space search problem, there is a *exploration-exploitation* trade-off to be considered [2]. Obtaining a high-performing architecture within a feasible amount of time is

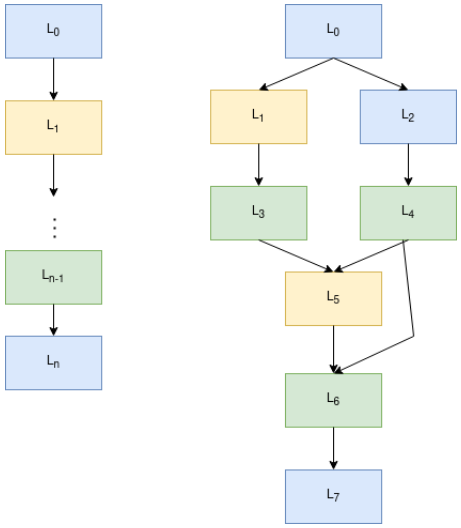

Fig. 1: Illustration of different ways to model neural architectures. Left: an example of a chain-structured NN. Right: an example of a multi-branch NN.

desired. However, converging too early to a suboptimal result is also not the goal.

Based on the current state of the art, NAS search algorithms can be classified mainly intro three different categories [8]:

1) ***Reinforcement Learning*** *(RL)-based NAS Algorithms:* Here, one considers the development of a neural architecture the agent's action, where the action space is the same as the search space. Therefore, it is then possible to frame NAS as a reinforcement learning problem [2]. After evaluating the performance of the given trained architecture on fitness data, it is possible to determine the agent's reward. How this estimation is performed will depend on the given method, more to these approaches will be seen at I-C. Furthermore, how the agent's policy is represented and it's optimization will also vary. More on trade-offs and details to these approaches will be seen in II.

2) ***Gradient****-based NAS Algorithms:* Consists of transforming the search space from discrete to continuous and performing gradient descent with respect to the fitness data set. This transformation requires a set of conditions and has still not been mathematically proven [8].

3) ***Evolutionary Computation*** *(EC)-based NAS Algorithms:* By the application of well established EC methods, which are various techniques based on the evolution of species within nature, many different efforts were implemented to navigate their respective search space. Among others, genetic algorithms (GAs), genetic programming (GP) and particle swarm optimization (PSO) techniques have already been successfully applied. More on these different techniques will be seen in II-B.

C. Performance Estimation Strategy

Independent of the search strategy, it is necessary to know how any given architecture performed in order to guide the next steps of the algorithm. There are many ways to estimate said performance, whereas the simplest would be complete training and validation. Given the complexity and size of the search spaces within NAS, this requires GPU days in the order of thousands [2]. This is why extensive research has been employed to reduce time on performance estimation, since it is a significant time bottleneck. Some of these approaches include [2]:

- **Lower Fidelity Techniques**: Shorter training time [9], [5], training solely on a subset of the data [10], training on downscaled data [11] or with downscaled models [5], [3]. These methods do introduce bias as performances will normally be underestimated.
- **Learning Curve Extrapolation**: Performance is extrapolated after just a small number of epochs and then decided upon directly. Klein et al. [10] considered architectural hyperparameters to predict which architectures are most promising after partial learning. Domhan et al [12] extrapolated partial learning curves to predict and eliminate sub-optimal architectures.
- **Weight Inheritance**: also dubbed as *network morphisms* is a technique that passes down weights from previously trained models forward to new ones. This approach can cut down computational costs to just a few GPU days [13].
- **One-Shot Models**: also called *weight sharing* is a technique that treats all architectures as subgraphs of a supergraph, which is named the one-shot model. Weights are shared between architectures that meet the condition of having edges in common. Finally, only the weights of a single one-shot model has to be trained and the sub-graph architectures can be evaluated directly as they inherit weights from the one-shot model. This demonstrates great success by cutting-down the entire process to a few GPU days. However, significant bias is introduced as the underestimation of architectures by this approach is harsh.

II. State of the Art

A. Overview

Currently RL-based algorithms are extremely costly in terms of computation, requiring thousands of graphics processing cards (GPUs) for days. This is already the case for median-scale data sets, as as data grows more complex, so does the processing time needed to search for a suitable architecture.

Gradient-based algorithms are faster, there are examples such as the DARTS algorithm, where processing power is cut down to single digit GPUs [14]. However, due to the not completely compatible relationship of NAS and a gradient-based optimization, frequently sub-optimal architectures are found [8].

On the other hand, EC methods, while not perfect, have been around for decades and are easily applicable to solve complex non-convex optimization problems, as they are insensitive to local minima and do not require gradient information [8].

B. Evolutionary Neural Architecture Search (ENAS)

ENAS algorithms are NAS algorithms that leverage existing evolutionary computational methods to search for an optimal architecture within a defined search space.

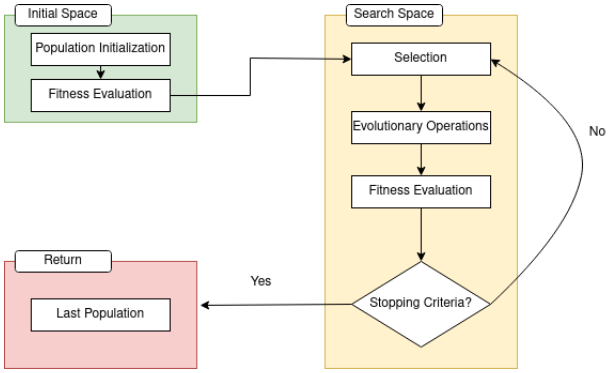

Fig. 2: Generic flowchart of a typical ENAS algorithm

Any ENAS algorithm can be broken down into a series of steps exemplified by Fig. 2. The term *population* seen in Fig. 2 is defined as a a finite number of *individuals*, where each individual is any given neural architecture or model within the defined search space.

At the beginning of the algorithm an initial population is created within a pre-defined initial space. Then, their fitness is evaluated, which essentially is the process of performance estimation already covered in I-C for each individual in this generation's population. Finally, this population undergoes a repetitive process that includes *selection*, *evolutionary operations* and once again a fitness evaluation. If the resulting population meet the defined goal, the final architecture has been found. Otherwise repeat the last process until this goal is reached.

Having understood the main steps of an ENAS implementation, it becomes clear that ENAS methods can be differentiated by their [8]:

- **Encoding Space**: contains all valid individuals (architectures) within a given population.
- **Encoding Strategy**: How architectures are represented within the encoding space.
- **Population Update Method**: These are the selection strategies, how the evolutionary operators are applied and how populations are ultimately updated.
- **Fitness Evaluation**: The same concept of performance estimation strategy already touched on in I-C.

In the following subsections, state of the art ENAS methods will be discussed and illustrated through the lense of these different elements that compose an ENAS algorithm. Although, ENAS encompasses many different techniques such as evolutionary algorithms (EA), Swarm Intelligence (SI) methods and many others; this work will focus mostly on EA, since the vast majority of relevant ENAS research have utilized it.

1) Encoding Space: The encoding space can be further divided into the initial space and the search space. These two may be the same in some cases, but often are not. The initial space is the set of all possible architectures that any individual in the initial population may become. There are three types of architecture initialization approaches [8]: starting from trivial conditions, rich initialization and random initialization in the encoding space.

The trivial space approach has been done by [3] and had the benefit of giving much more freedom to the algorithm to explore unseen architectures and also justifies well the used of EC-methods instead of any other. However, this comes at the cost of high computational resources.

The rich initialization method, also named *well-designed space*, consists of initializing the population within a set of state-of-the-art architectures. This way, a good architecture can be found early on and potentially decrease search time. On the other hand, finding novel architectures is very improbable.

2) Encoding Strategy: (TBD)

3) Population Update: Genetic Algorithm (GA)-based ENAS is the most popular approach among ENAS methods, since architecture representation is very convenient in GA [8]. The selection is the first stage of updating the current population and can be divided into several strategies, four of whom are widely used: Elitism, Discard worst or oldest, Roulette and Tournament Selection.

Works such as [15] take use of elitism, where essentially only the fittest of individuals are selected to compose the next population. This can cause a loss of diversity, as there is a chance that similar architectures perform similarly well and thus may cause generations to breed only akin individuals, since only the fittest are kept. This in turn, can cause the population to fall within a local optima and not being able to explore efficiently.

Other approaches such as [3] and [16] discarded the oldest individual from the population, which is also known as aging evolution. This ensures that the search does not focus on good models too early and therefore performs a more broad search of the encoding space in comparison to non-aging evolution.

There is also the possibility to combine different strategies such as in [17], where both discarding the worst and the oldest were used.

C. NAS and tinyML

Machine learning on small microcontrollers is a great ambition. TinyML is the field that answers to that challenge and aims to provide intelligent features to even a $5 device, such as an off-the-shelf microcontroller. This is not a small effort as microcontrollers possess very limited resources, especially concerning memory and storage. In the following subsections, relevant work that combined NAS algorithms within the tinyML universe will be shown.

*1) **MCUNet**:* MCUNet, a framework that shows promising results, is composed of two major components: the efficient neural architecture (TinyNAS) and the lightweight inference engine (TinyEngine) [18]. The TinyNAS algorithm is a two-stage neural architecture search method, where firstly it optimizes the search space according to the given resource constraints and then performs an ENAS algorithm to find the best architecture within this new search space. Since the performance of NAS methods depend strongly on the search space [19], this technique through extra constaints forces the search to consider a smaller set, where only architectures that fit the desired requirements can be found. These requirements may be limited memory consumption, storage limits, latency and even energy.

MCUNet used weight sharing [18], where a single super network is created, which contains all the sub-networks in the search space. This method leverages the graph attribute of these networks to share weights among subnetworks and drastically reduce the time needed to train all possible individuals.

The ENAS algorithm used in MCUNet uses elitism as its selection strategy, always choosing the top-20 individuals in terms of accuracy within a given generation. Crossover is applied to generate 50 new candidates and then mutation with a probablity of 10% is used to generate the remaining 50. All generations are of size 100. Finally, after 30 iterations, the fittest architecture is chosen.

In addition to that, TinyEngine implements a code generator-based compilation method that not only eliminates memory overhead, but also improves the speed of inference as well [18]. This addresses the shortcomings of similar libraries such as TF-Lite Micro [20] and CMSIS-NN [21] that opt for runtime interpretation of code instead.

Finally, this work has shown to be able to reduce memory usage by 2.7x and improve inference speed by 1.7-2.2 compared to TF-Lite Micro and CMSIS-NN, also decreased code size by up to 4.5X and 5.0x for TF-Lite Micro and CMSIS-NN respectively [18]. MCUnet achieved state of the art performance, taking 12.5 GPU days to design a model [18]. This is a great improvement compared to MnasNET, which took 40,000 GPU hours for the same data set [22] and is also faster than most NAS methods performed on regular machines.

III. Conclusion

Throughout this work a brief overview of the current research on NAS has been displayed. NAS is not only a complex optimization problem, but also one that requires significant computational resources in most applications. The trade-offs of the mainstream approaches to NAS have also been demonstrated, whilst also highlighting efforts done with the use of EC-based approaches.

In addition to that, different applications of NAS methods within the tinyML paradigm have been illustrated. For IoT devices and microcontrollers it is not only needed to train models considering performance, but also considering other parameters such as latency, memory usage and even energy preservation.

References

[1] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," *arXiv preprint arXiv:1611.01578*, 2016.

[2] T. Elsken, J. H. Metzen, and F. Hutter, "Neural architecture search: A survey," *The Journal of Machine Learning Research*, vol. 20, no. 1, pp. 1997–2017, 2019.

[3] E. Real, S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, J. Tan, Q. V. Le, and A. Kurakin, "Large-scale evolution of image classifiers," in *Proceedings of the 34th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, D. Precup and Y. W. Teh, Eds., vol. 70. PMLR, 06–11 Aug 2017, pp. 2902–2911. [Online]. Available: https://proceedings.mlr.press/v70/real17a.html

[4] Z. Zhong, J. Yan, W. Wu, J. Shao, and C.-L. Liu, "Practical block-wise neural network architecture generation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 2423–2432.

[5] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 8697–8710.

[6] H. Cai, J. Yang, W. Zhang, S. Han, and Y. Yu, "Path-level network transformation for efficient architecture search," in *Proceedings of the 35th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, J. Dy and A. Krause, Eds., vol. 80. PMLR, 10–15 Jul 2018, pp. 678–687. [Online]. Available: https://proceedings.mlr.press/v80/cai18a.html

[7] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[8] Y. Liu, Y. Sun, B. Xue, M. Zhang, G. G. Yen, and K. C. Tan, "A survey on evolutionary neural architecture search," *IEEE transactions on neural networks and learning systems*, 2021.

[9] A. Zela, A. Klein, S. Falkner, and F. Hutter, "Towards automated deep learning: Efficient joint neural architecture and hyperparameter search," *arXiv preprint arXiv:1807.06906*, 2018.

[10] A. Klein, E. Christiansen, K. Murphy, and F. Hutter, "Towards reproducible neural architecture and hyperparameter search," 2018.

[11] P. Chrabaszcz, I. Loshchilov, and F. Hutter, "A downsampled variant of imagenet as an alternative to the cifar datasets," *arXiv preprint arXiv:1707.08819*, 2017.

[12] T. Domhan, J. T. Springenberg, and F. Hutter, "Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves," in *Twenty-fourth international joint conference on artificial intelligence*, 2015.

[13] H. Cai, T. Chen, W. Zhang, Y. Yu, and J. Wang, "Efficient architecture search by network transformation," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, no. 1, 2018.

[14] H. Liu, K. Simonyan, and Y. Yang, "Darts: Differentiable architecture search," *arXiv preprint arXiv:1806.09055*, 2018.

[15] T. Elsken, J.-H. Metzen, and F. Hutter, "Simple and efficient architecture search for convolutional neural networks," *arXiv preprint arXiv:1711.04528*, 2017.

[16] W. Zhang, L. Zhao, Q. Li, S. Zhao, Q. Dong, X. Jiang, T. Zhang, and T. Liu, "Identify hierarchical structures from task-based fmri data via hybrid spatiotemporal neural architecture search net," in *International Conference on Medical Image Computing and Computer-Assisted Intervention*. Springer, 2019, pp. 745–753.

[17] H. Zhu, Z. An, C. Yang, K. Xu, E. Zhao, and Y. Xu, "Eena: Efficient evolution of neural architecture," in *Proceedings of the IEEE/CVF International Conference on Computer Vision Workshops*, 2019, pp. 0–0.

[18] J. Lin, W.-M. Chen, Y. Lin, C. Gan, S. Han *et al.*, "Mcunet: Tiny deep learning on iot devices," *Advances in Neural Information Processing Systems*, vol. 33, pp. 11 711–11 722, 2020.

[19] I. Radosavovic, R. P. Kosaraju, R. Girshick, K. He, and P. Dollár, "Designing network design spaces," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 10 428–10 436.

[20] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "{TensorFlow}: A system for {Large-Scale} machine learning," in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016, pp. 265–283.

[21] L. Lai, N. Suda, and V. C.-N. Chandra, "Efficient neural network kernels for arm cortex-m cpus. arxiv 2018," *arXiv preprint arXiv:1801.06601*.

[22] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le, "Mnasnet: Platform-aware neural architecture search for mobile," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 2820–2828.