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 speech recognition, image classification 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 automate architecture discovery. This field has grown remarkably within the last 5 years and was able to, through many optimizations, obtain competetive architectures with accuracies that rival state of the art models. 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, there are other goals other than inference accuracy such as latency, energy consumption, memory consumption, and many others to consider. Generally, the manual process to port AI algorithms on said devices include not only training a given NN, but also pruning, quantizing and in some cases designing custom hardware to implement said model. This approach typically includes not optimal solutions due to the mutual influence of the hardware and algorithm. Given the extensive work done on NAS in the last years, there has been interesting efforts that utilized existing NAS algorithms to optimize a model's search considering these many other dimensions to best fit a given hardware platform or device in an automated fashion. 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 briefly highlighting relevant applications within 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 to either improve existing architectures or create new ones for novel applications or problems.

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 and minimizing human participation. Since then, many different methods have appeared.

To better understand NAS, one can categorize it using the following characteristics [2]:

- **Search Space**: What is the set of all possible architectures that can in principle be considered by the algorithm.
- **Search Strategy**: How the *search space* is explored by the algorithm.
- **Performance Estimation Strategy**: 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. However, this limits the freedom to explore architectures that have not been conceived before, and also possibly restricts the complexity of the design.

How an architecture is constructed or described is a core concept to how the search space of an NAS algorithm is defined. Defining 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 is over-simplified and a non-realistic approach. Multiple layer connections and other modern design elements such as skip connections are needed to formulate architectures capable to deal with complex data sets. This has already been seen in the works of [1] and [3] for example. This allows one to build *multi-branch networks*, which are structures where each layer's input is a function of previous layers outputs, increasing significantly the degrees of freedom of architecture design and the search space's complexity. The predominant trends are however to search within constructs such as *cells* and *blocks*.

Cells were initially considered by works such as [4] and [5]. What is proposed there, is to break-off architectures into cells, such that the search space is then designated within a single cell instead of an entire architecture. This cell is the basic unit of the resulting architecture, which can be then combined with other identical cells in various fashions to produce a final design. This strongly reduces the search space, as there are substantially less layers within a single cell in comparison to an entire architecture. Cells drive the discussion between *micro-architectures* versus *macro-architectures* [2]. The macro architecture attempts to determine how cells should be connected and how many are needed to build a model [2]. On the other hand, the micro architecture aims to find the optimal structure within a cell [2]. Ideally, both should be optimized simultaneously, which of course leads to a complex search space. There have been efforts 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 good results.

Search within blocks, similarly to cells, consist of using blocks as basic units of the search space. The difference is that various layers of different types are combined to produce basic blocks such as ResBlock, DenseBlock, ConvBlock and so on. These blocks are fixed in their topology and require less parameters to build an architecture. Therefore, they drastically reduce the search space. Various blocks can be combined

in different manners and quantities to produce competetive architectures. Cells can be considered as a special case of block-based search, where each block is identical [8].

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 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 as 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. Furthermore, how the agent's policy is represented and it's optimization will also vary.

2) ***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. More on these different techniques will be seen in section II-B.

3) ***Gradient****-based NAS Algorithms:* EC and Reinforcement learning methods consider an approach, where neural architectures are spread across a discrete and non-differentiable search space. There are many reasons for this, as networks may differ through many different manners other than numerical parameters or weights. Gradient-based approaches however impose restrictions and reformulate the problem to obtain a continuous search space. This allows an algorithm to use gradient-based methods to pursue an optimal model, which leads many times to competetive and fast techniques.

C. Performance Estimation Strategy

Evaluating the performance of a given architecture is crucial to guide the next steps of the algorithm, regardless of the search strategy. There are many ways to estimate said performance, whereas the simplest would be complete training and validation. However, given the complexity and size of the search spaces within most NAS formulations, this may require 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. The work of [10] considered architectural hyperparameters to predict which architectures are most promising after partial learning. The work in [12] extrapolated partial learning curves to predict and eliminate sub-optimal architectures. The difficulty here lies in generating good predictions within a small amount of evaluations, in order to speed up the search.
- **Weight Inheritance**: also dubbed as *network morphisms* is a technique that passes down weights from previously trained models 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, 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, as it was one of the main reasons that the DARTS algorithm was able to cut down computational costs to a few GPU days [14]. However, significant bias is introduced as the underestimation of architectures by this approach is harsh.

II. State of the Art

In the following sections, the state-of-the-art of NAS and tinyML-NAS crossovers will be discussed. In section II-A, a quick overview of the main NAS approaches will be presented, as well as their contrasting qualities. Then, in section II-B a deeper dive in *Evolutionary Neural Architecture Search* (ENAS) algorithms is presented. Finally, in section II-C, relevant work that brings NAS to tinyML applications will be illustrated.

A. Overview

Currently RL-based algorithms are extremely costly in terms of computation, requiring hundreds of GPUs for days. This is already the case for median-scale data sets, so 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 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 [15], as they are insensitive to local minima and do not require gradient information [8]. In some implementations, EC techniques within NAS do require significant computational resources, but have shown to find competetive models from trivial initial conditions such as in [3]. Efforts like these show promising implementations that require no human interaction as well as little to no bias. In

addition to that, the fact that simple initial conditions are applied, also possibly prevents the finding of overly complicated architectures, which generally occur when starting from well-established macro-architectures.

B. Evolutionary Neural Architecture Search (ENAS)

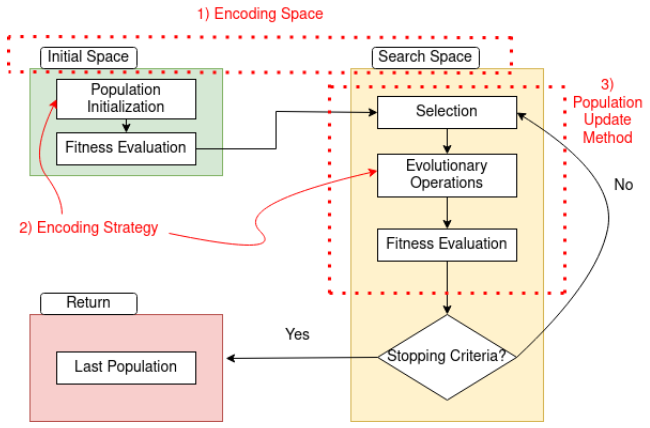

Fig. 1: Generic flowchart of a typical ENAS algorithm. The white boxes show concrete steps in the algorithm's implementation. The enumerated red keywords are the terms used to define ENAS algorithms, and they are used in this figure to illustrate which steps in the algorithm belong to which terms, since they are explored later in this work.

ENAS algorithms are NAS algorithms that leverage existing evolutionary computational methods to search for an optimal architecture within a defined search space. It can be broken down into a series of steps exemplified by Fig. 1. The term *population* seen in Fig. 1 is defined as a finite number of *individuals*, where each individual is one neural architecture within the defined search space.

At the beginning of the algorithm an initial population is created within a pre-defined initial space. Then, their fitnesses are evaluated, which essentially is the process of performance estimation already covered in section I-C.

Now, this population undergoes a process that includes in sequential order: A) *selection*, B) *evolutionary operations* and C) *fitness evaluation*. These steps can be broadly grouped into the *population update method*, which will later be explored in section II-B3. The general concept of this phase is the core of evolutionary methods. Inspired by the evolutionary processes in nature, a selection criteria is imposed on a given population, determining which individuals advance to the next generation. Following that, evolutionary operations are applied to these survivors. These mimic different concepts such as mutation, crossovers and many others to change the current architectures and continue the search efficiently. Finally, the new changed individuals have their fitness evaluated. If the resulting population meet the defined goal, the final architecture has been found. Otherwise this process will be repeated using the last generation until the criteria is met.

It is possible to differentiate ENAS algorithms through four dimensions [8]: Encoding Space, Encoding Strategy, Population Update Method and Fitness Evaluation. Fitness evaluation has however been covered already in section I-C, therefore the focus will remain on the first three categories.

In the following subsections, state-of-the-art ENAS methods will be discussed through these different elements. Although, ENAS encompasses many different techniques, this work will focus mostly on EA, since the vast majority of relevant ENAS research has utilized it.

1) Encoding Space: The encoding space contains all the valid individuals or architectures encoded in the population [8]. It can be further divided into the initial space and the search space, as seen in red illustrated in Fig. 1. 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 be initialized as. There are three types of architecture initialization approaches [8]:

- **Starting from trivial conditions**: This method aims to impose little to no human bias in the initial population. The groundbreaking work in [3] had the benefit of giving much more freedom to the algorithm to explore unseen architectures and also justified well the used of EC-methods instead of other approaches. However, this comes at the cost of high computational resources. The work in [16] is also another example of competetive architecture achievements that used a trivial initial space.
- **Rich Initialization**: This technique, 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. There has been relevant work that used this method as initialization technique, such as from [17].
- **Random Initialization**: This form of initialization also aims to reduce human bias or intervention in the process of architecture evolution. Many efforts have utilized this method such as [18] and [19]. Here, the search space and the initial space are the same and initialization is random.

The search space and their constraints have been touched in section I-A, where the concept of block-based and cell-based search spaces have been introduced. Notable works like [19] use traditional blocks such as ResBlock, DenseBlock and ConvBlock to further constraint the search space and facilitate architecture discovery. Other efforts such as [20] and [21] used different basic blocks to adapt to their respective desired architecture use-cases.

Most implementations of cell-based searches have the macro-architecture be based on human expertise and concentrated mostly on optimizing their micro-architectures. This was done for example in [22].

Nonetheless, there are more EA specific parameters to consider when evaluating the search space of an ENAS algorithm. Evolutionary operators play a role in constraining the algorithm's search space and will be touched on in more depth in section II-B3. An example would be the work of in [23], which did not specify explicitly the maximum depth of the resulting architectures, but then used the algorithm's evolutionary operators to extend the individual's architectures indefinitely if needed.

2) Encoding Strategy: The encoding strategy defines how network architectures are encoded into individuals. They can be typically split into *fixed-length* and *variable-length* encoding strategies. These approaches differ in whether the length of an individual varies during the evolutionary process, which is essentially the layer depth of resulting architectures.

In the fixed-length variant, all individuals have the same length during the search process. This facilitates the process, since standard evolutionary operations are by design to be used with individuals of equal lengths. For example, in the work of [16], this enabled a much easier implementation of its ENAS algorithm, especially with the utilization of crossover. However, for fixed-length implementations, a maximum length has to be established, which then also creates a dependency on human expertise and introduces bias.

On the other hand, the variable-length methods do not require human knowledge regarding layer depth, which creates a potential to have a fully-automated implementation. Moreover, the greatest advantage to this approach is that the algorithm can define more details of the architecture with greater freedom. This comes at the cost of a harder and more complex implementation, since evolutionary operators may not be directly compatible with this encoding strategy, requiring a redesign. In addition to that, the extra flexibility provided by this technique may also cause overly complex architectures with costly performance evaluations. The work of [18] is a prime example of a variable-length encoding strategy that needed to redesign its operators.

3) Population Update Method: The population update method includes selection strategies and how evolutionary operators are applied. This will then define how populations ultimately evolve. This is also illustrated in Fig. 1.

Genetic Algorithm (GA)-based methods are the most popular approach among the EA techniques used in ENAS, since architecture representation is very convenient in GA [8]. 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 [24] 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 [25] 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 [26], where both discarding the worst and the oldest were used. Roulette selection, as the name implies, applies a probability to each invidividual according to its fitness. So, there is a chance associated to each individual's survival. This means that any individual can theoretically be discarded independent of performance or not. Finally, Tournament selection will chose from en equally likely sampling of individuals the single best one.

Finally, the most common operations in ENAS are crossover and mutation. Crossover needs two individuals to generate an offspring, while mutation is applied solely to a single individual. Mutation wishes to reach the global optimum around the individual which it is applied upon and generally introduces exploration [8]. Crossover on the other hand contributes mainly to exploitation as it reunites different characteristics from ancestors into a single offspring [8].

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 the cheapest off-the-shelf microcontrollers. 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) **MnasNET**:* The work in [27] used a reinforcement learning based NAS algorithm to obtain architectures that consider trade-offs between latency and accuracy. The search process included real latency information instead of proxies through metrics such as FLOPs. Also, MnasNet developed what was called a *factorized hierarchical search space*, a block-based search space that differred from previous related-work that focused on cells. This allowed different architectures per block, which effectively resulted better trade-offs between accuracy and latency. This work was able to achieve state-of-the-art performance at the time on both ImageNet and COCO object detection within mobile benchmarks [27].

*2) **MCUNet**:* MCUNet is a framework that shows promising results, composed of two major components: the efficient neural architecture (TinyNAS) and the lightweight inference engine (TinyEngine) [28]. 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 [29], 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 usage.

MCUNet used weight sharing [28], such that the time needed to train all possible individuals is drastically reduced. 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 [28]. 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 [28]. MCUnet achieved state of the art performance, taking 12.5 GPU days to design a model [28]. This is a great improvement compared to MnasNET, which took 40,000 GPU hours for the same data set [27] and is also faster than most NAS methods performed on regular machines.

*3) **MicroNets**:* The work [30] implemented a tailored differentiable NAS algorithm based on the work in DARTS [14], that discovered highly accurate models, while also satisfying SRAM, eFlash and latency constraints within a given hardware architecture. Regularization terms are applied to the used NAS algorithm such that the resulting models consider eFLash memory and produce activations that fit in the available SRAM [30], whilst aiming to achieve high accuracy results.

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. Therefore, displaying an interesting challenge, which brings however great promise to the field.

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] Y. Sun, G. G. Yen, and Z. Yi, "Igd indicator-based evolutionary algorithm for many-objective optimization problems," *IEEE Transactions on Evolutionary Computation*, vol. 23, no. 2, pp. 173–187, 2018.

[16] L. Xie and A. Yuille, "Genetic cnn," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 1379–1388.

[17] S. Fujino, N. Mori, and K. Matsumoto, "Deep convolutional networks for human sketches by means of the evolutionary deep learning," in *2017 Joint 17th World Congress of International Fuzzy Systems Association and 9th International Conference on Soft Computing and Intelligent Systems (IFSA-SCIS)*. IEEE, 2017, pp. 1–5.

[18] Y. Sun, B. Xue, M. Zhang, and G. G. Yen, "Evolving deep convolutional neural networks for image classification," *IEEE Transactions on Evolutionary Computation*, vol. 24, no. 2, pp. 394–407, 2019.

[19] ——, "Completely automated cnn architecture design based on blocks," *IEEE transactions on neural networks and learning systems*, vol. 31, no. 4, pp. 1242–1254, 2019.

[20] Z. Chen, Y. Zhou, and Z. Huang, "Auto-creation of effective neural network architecture by evolutionary algorithm and resnet for image classification," in *2019 IEEE international conference on systems, man and cybernetics (SMC)*. IEEE, 2019, pp. 3895–3900.

[21] D. Song, C. Xu, X. Jia, Y. Chen, C. Xu, and Y. Wang, "Efficient residual dense block search for image super-resolution," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 07, 2020, pp. 12 007–12 014.

[22] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized evolution for image classifier architecture search," in *Proceedings of the aaai conference on artificial intelligence*, vol. 33, no. 01, 2019, pp. 4780–4789.

[23] W. Irwin-Harris, Y. Sun, B. Xue, and M. Zhang, "A graph-based encoding for evolutionary convolutional neural network architecture design," in *2019 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2019, pp. 546–553.

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

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] C. Banbury, C. Zhou, I. Fedorov, R. Matas, U. Thakker, D. Gope, V. Janapa Reddi, M. Mattina, and P. Whatmough, "Micronets: Neural network architectures for deploying tinyml applications on commodity

microcontrollers," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 517–532, 2021.