Regularized Evolution for Macro Neural Architecture Search

George Kyriakides(✉) and Konstantinos Margaritis

University of Macedonia, 55236 Thessaloniki, Greece
{ge.kyriakides,kmarg}@uom.edu.gr

Abstract. Neural Architecture Search is becoming an increasingly popular research field and method to design deep learning architectures. Most research focuses on searching for small blocks of deep learning operations, or micro-search. This method yields satisfactory results but demands prior knowledge of the macro architecture's structure. Generally, methods that do not utilize macro structure knowledge perform worse but are able to be applied to datasets of completely new domains. In this paper, we propose a macro NAS methodology which utilizes concepts of Regularized Evolution and Macro Neural Architecture Search (DeepNEAT), and apply it to the Fashion-MNIST dataset. By utilizing our method, we are able to produce networks that outperform other macro NAS methods on the dataset, when the same post-search inference methods are used. Furthermore, we are able to achieve 94.46% test accuracy, while requiring considerably less epochs to fully train our network.

Keywords: NAS · Neural Architecture Search · Fashion MNIST

1 Introduction

As the amount of available raw data has increased over the years, in conjunction with the steady increase in available processing power, machine learning has also increased in popularity as a modelling and predictive tool. This has spurred new research in the field, which in turn has enabled the implementation and automation of tasks previously considered exceptionally hard for a computer. Especially in the past decade, neural networks in the form of deep learning have been an increasingly popular tool for such industrial and research applications [2,4,6,20].

Deep learning demands specific neural architectures in order to be successfully and effectively implemented. Traditionally, these architectures were hand-crafted by human experts, a laborious and time-consuming process. More recently, there have been significant efforts to automate the process of designing them. Thus, a new research field has emerged from these efforts, known as Neural Architecture Search (NAS). Many state-of-the-art architectures have been produced as the research in the field has progressed [11,13,16,17,22]. The process of designing the architectures requires significantly more computational resources than training a single architecture, but can be more resource-efficient in the long run, especially when low-power hardware is targeted [19].

I. Maglogiannis et al. (Eds.): AIAI 2020, IFIP AICT 584, pp. 111–122, 2020.
https://doi.org/10.1007/978-3-030-49186-4_10

Although most methodologies are able to produce state-of-the-art architectures, they rely on a pre-determined architectural macro structure (outer skeleton). Instead of designing the entire network, a small network block (called cell) is designed, and several cells are stacked in order to produce the final network, as dictated by the skeleton. This can essentially be called a micro-search, as the micro-structure of the network's cells is optimized. Although this philosophy is both effective and efficient, it requires prior knowledge of an appropriate skeleton. Thus, it is only applicable in areas where such prior knowledge has been obtained, usually through human experimentation. This greatly reduces the applicability of such methods to domains that have already been studied to a certain degree.

In this paper, we explore an algorithm created by combination of Regularized Evolution from [17] and genetic encoding from DeepNEAT [13], in order to generate architectures for the Fashion-MNIST dataset [21], by conducting a macro-architecture search. We aim to evaluate the proposed method's ability to generate good architectures without any prior knowledge about the application's domain, as well as compare it to other macro-architecture search methodologies. We first briefly present the two algorithms from which we borrow elements from, as well as the dataset and various NAS algorithms tested on it. Following, we explain our methodology and our experimental results. Finally, we present our research's limitations and discuss our findings.

2 Related Work

As previously mentioned, for our method we utilize a combination of Regularized Evolution and DeepNEAT, in order to formulate a macro NAS methodology. As such, in this section we present the basics of the two algorithms, as well as the Fashion-MNIST dataset.

2.1 Regularized Evolution

In the corresponding paper [17], the authors propose the usage of an evolutionary algorithm in order to generate architectures of increasingly better performance. Instead of utilizing tournament selection when updating the populations, the oldest solution in the population is discarded. Following, by randomly sampling N individuals, the best is selected to generate an offspring. The offspring is added to the population, while the oldest individual is discarded. This forces the algorithm to retain architectures that consistently perform well. An architecture which was trained to a high test accuracy by luck will not be able to survive for very long. The authors were able to produce better architectures than the state-of-the-art for the CIFAR10 dataset in the NASNet Search Space [23]. The search space restricts the algorithm's ability to produce arbitrary architectures. Instead, a pre-defined skeleton of repeated cells is used, while the algorithm designs the individual cells.

2.2 DeepNEAT

Even before the advent of deep learning, efforts where made in order to optimize both the design, as well as the weights of neural networks. One such example is the NEAT algorithm [18], which utilized generational evolution in order to evolve simple neural networks. The method was more recently extended in order to design deep neural architectures, in the forms of DeepNEAT and CoDeepNEAT [13]. DeepNEAT is able to design macro-architectures, by encoding each layer's parameters, as well as connections between layers to different genes. In order to achieve crossover, each gene is marked with an indicator, which lets homologous genes align. In the paper, the researchers are able to utilize CoDeepNEAT which employs co-evolution of cells and skeletons, in order to generate image recognition and captioning architectures with success.

2.3 Fashion-MNIST

Fashion-MNIST [21] is a benchmarking dataset for image recognition tasks. The dataset contains a number of labeled fashion items. Each image is a 28 × 28 pixel grayscale image of various clothing items. In total there are 60,000 items in the training set and 10,000 in the test set. There are a total of 10 different classes in the dataset. An example of the dataset can be seen in Fig. 1. It has been proposed as a direct drop-in replacement for the popular MNIST dataset [10].

A number of NAS methodologies have been able to produce state-of-the-art networks for this specific dataset. The latest NAS paper which was able to produce a new state-of-the-art is [14]. By using the experts advice framework, it is able to achieve a 96.36% test set accuracy, higher than any other NAS paper. Other methodologies are presented and compared in [15], which are able to produce very high-performing architectures. Nonetheless, these are all micro-search methods. Macro-search, although not as successful, is able to produce sufficient architectures, taking into account that the search space usually becomes unbound. One such methodology is described in [1], which combines Genetic Algorithms and Dynamic Structured Grammatical Evolution, in order to evolve the architectures. The algorithm is able to produce high-performing architectures, although some post-search training and inference techniques are utilized, in order to boost the network's performance. Another study employs the Ant Colony Optimization algorithm in order to design neural architectures [3] with marginally less success, although the concept of weight re-usability that the algorithm enables is very interesting. Finally, in [7], the authors propose Bayesian Optimization as a means to guide through the search space, while using network morphism first proposed in [5].

3 Methodology

In this paper, we utilize concepts from DeepNEAT and Regularized Evolution in order to generate convolutional architectures for the Fashion-MNIST dataset.

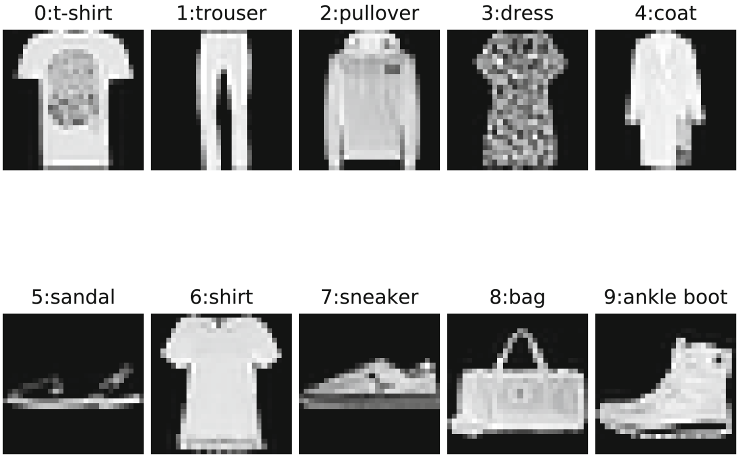

Fig. 1. Sampled Fashion-MNIST images, one for each class.

In order to do so, we first implement the genes that will encode the information regarding the connections, as well as the nodes of each architecture. This is inspired by the work done in [13]. Following, we define the rules of evolution, taken from [17]. Finally, we explain how a genome (a collection of genes) is translated to a functioning neural network and evaluated.

3.1 Architecture Representation

In order to represent a neural architecture, we utilize layer and connection genes. These define the network's topology, as well as each layer's functionality and parameters.

Layer Genes. Each layer gene dictates the layer implemented by each node. Originally, DeepNEAT proposes the use of a Convolution, followed by a Dropout and possibly a MaxPool layer. In this configuration, the gene contains information about the kernel size, number of filters, dropout probability, as well as the existence of the MaxPool layer. As this can greatly increase the search space size (in the original paper, there were 896 possible combinations of kernel size, number of filters, and MaxPooling), we utilize only 12 discrete layer choices. First, we define a fixed number of channels for all architectures, as in [17]. Each node can be either a convolution or a pooling layer, with kernels of sizes 2×2, 3×3, and 5×5. Furthermore, each pooling layer can be either a MaxPooling or an AveragePooling layer. Finally, each convolution layer can either have filters equal to the initially defined number of channels, or half of that. The available choices are depicted in Table 1. Thus, we are able to reduce the number of choices to 12 for each layer gene. As we only employ the mutation operator, we mutate a gene by selecting a different layer setup from the 12 available.

Table 1. Available layer choices.

Layer selection	Layer type	Kernel size	Operation
CONV_2.H	Convolution	2×2	Filters Number = Half Channels
CONV_3.H		3×3	
CONV_5.H		5×5	
CONV_2.N	Convolution	2×2	Filters Number = All Channels
CONV_3.N		3×3	
CONV_5.N		5×5	
POOL_2.A	Pooling	2×2	Average
POOL_3.A		3×3	
POOL_5.A		5×5	
POOL_2.M	Pooling	2×2	Max
POOL_3.M		3×3	
POOL_5.M		5×5	

Connection Genes. Connection genes dictate the network's topology. Each gene contains three values; the origin node, the destination node, as well as a boolean flag, indicating whether the connection is active. The mutation operator reverses the connection's status. Namely, it disables the connection if it is active and vice versa. Although in [17] the authors allow a specific connection to change either its origin or destination, we choose not to. Our choice does not hinder the development of non-sequential architectures i.e. having two or more layers with the same inputs, or a layer with multiple inputs. This will be further analyzed in the following subsection.

3.2 Evolving Architectures

Following the basics of [17], we first initialize a population of P individuals. Each individual consists of 4 random layers. Note that this does not pose a lower limit to the number of layers in the population, as connections can be disabled and thus layers can be deactivated, by disabling its incoming connections. In order to evolve our architectures, we utilize the mutation operator, with a probability of p_{mutate}. Furthermore, we define a probability p_{add} to add a new node to the network, as well as a probability $p_{identity}$ to leave the network intact. At each evolution cycle (offspring creation and replacement of the oldest individual), N individuals are sampled from the population. The best out of N is selected and its offspring is generated through mutation. After evaluating its fitness, the offspring is inserted into the population and the oldest individual is discarded. We utilize the generated network's accuracy on the test set as an individual's fitness.

Adding Nodes. In order to insert a new node into the network, a random connection is selected. The new node is placed between the connection's origin and destination nodes, while the connection is disabled. As a layer gene's mutation allows the re-activation of the connection, the original destination node can have multiple inputs. Likewise, the original origin node can have multiple outputs. This can be seen in Fig. 2. Each node is assigned a unique number which indicates when the node was added. Node −2 is the network's input, while 99 is the network's output. Here it is evident how a single node may have multiple inputs and outputs. Node −2 has developed multiple outputs, while node 99 has developed multiple inputs.

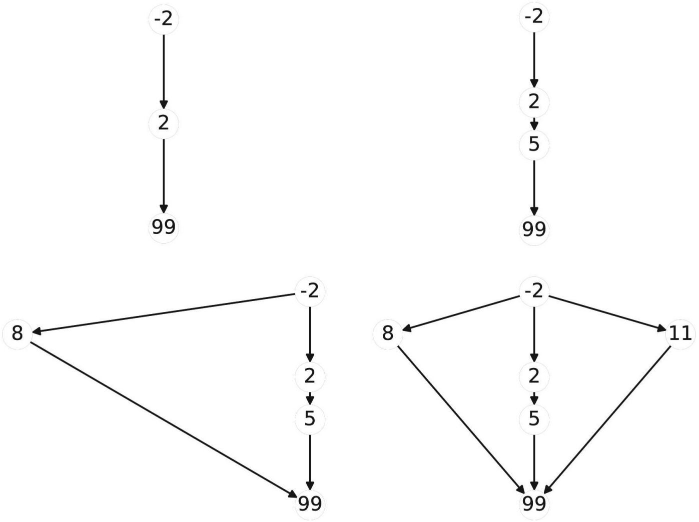

Fig. 2. Three node additions to the same genome.

3.3 Implementing the Networks

As our genomes describe the convolutional section of the network, we define a small outer skeleton, in order to be able to utilize the architecture for classification. We add three fully connected layers (FC) after the output node, with decreasing number of filters. The first FC layer (FC1) has number of filters equal to a quarter of the input features. Layer FC2 has half of the filters FC1 has. Finally, FC3 has 10 filters, equal to the number of classes in the dataset.

When a gene represents a convolution layer, the ReLU activation function is used after the layer, clipped to a maximum value of 6 (ReLU6). Following, a Batch Normalization layer is employed, along with a Dropout2d layer, with 0.1 dropout probability. In case a node has multiple inputs, all of them are scaled to the same size and summed pixel-wise. In case there are inputs with different numbers of channels, a 1×1 convolution is utilized in order to scale the number of channels, before summing the inputs.

4 Experiments

4.1 Experimental Setup

For our experiments, we employed a population size of $P = 100$ genomes. A total of 400 cycles were utilized, in order to evolve the networks. At each cycle, $N = 25$ individuals were selected as parents. Each network was trained for 10 epochs on the train set and then evaluated on the test set. In order to train the networks we utilized the Adam optimizer [8]. No data augmentation was performed on the dataset during the search phase. We chose 64 as the standard number of channels, meaning that CONV_2.H, CONV_3.H, and CONV_5.H will each contain 32 filters. The probability to add a new node to a network was set to $p_{add} = 0.25$, while the probability to leave a network intact was set to $p_{identity} = 0.05$. Thus, the probability to mutate either a node or a connection was $p_{mutate} = 0.7$.

In order to establish a baseline for our method, we also conducted a random search. For this purpose, we repeated the experiment, but selected a single random individual from the population to generate the offspring each time, instead of 25. Thus, the algorithm reduced to a random search. Both experiments were conducted by utilizing 4 NVIDIA Tesla V100 GPU cards. The experiments were implemented using the NORD framework [9]. All code is available on github[1], along with the PyTorch state dictionaries of the fully trained networks.

4.2 Results

Evolutionary search (ES) seems to outperform random search (RS) significantly. It is able to find better models, as well as do so more consistently. This can be seen in Fig. 3, where the best-performing architecture found so far is depicted for both ES and RS. ES is able to continuously improve its best-performing architecture, while RS struggles to find better architectures (left). Furthermore, ES's empirical cumulative distribution function (right) exerts first-order stochastic dominance over ES. This means that it is able to find solutions as good as random search or better, with higher probability. ES was able to find architectures with up to 93.62% test accuracy, which consisted of two CONV_2.N layers, followed by a CONV_3.N, a CONV_5.N, a POOL_5.A, and a CONV_3.H layers, all of them sequentially connected. Random Search was able to find architectures with up to 91.91% test accuracy, although these architectures consisted of a CONV_3.H, CONV_5.H, a CONV_3.H, and a POOL_5.M layer, again connected sequentially.

Most of the best architectures found by ES consisted of sequentially connected convolutions with full channel count, followed by a pooling layer and a reduced-channel convolution. Some architectures developed fanning inputs and outputs, with satisfactory performance. The most prominent features were two sequential CONV_2.N layers, followed by a CONV_3.N, a CONV_5.N, and a

[1] https://github.com/GeorgeKyriakides/nord/tree/AIAI2020/nord.

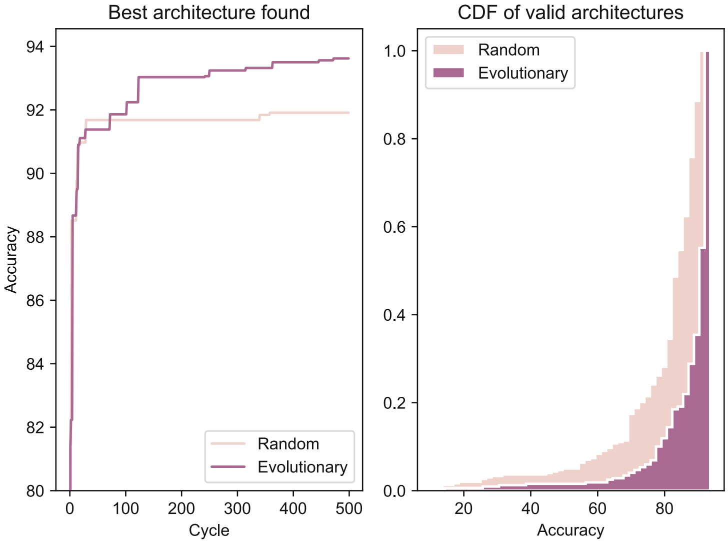

Fig. 3. Best architecture's performance (left) and performance empirical cumulative distributions (right).

POOL_5.A layer. In Fig. 4 each evolved genome is plotted, with alpha (opacity factor) equal to $\frac{1}{400}$ (as we have a total of 400 evolved architectures). Thus, more dominant patterns can be visually distinguished. Here, OUTPUT layers denote the output of the convolutional part of the network, i.e. the input to the fully connected layers. On average, valid evolved architectures contained 5.5 nodes, with a standard deviation of 1.1. Invalid architectures include those that did not contain at least a single path from the input to the output layer, or the intermediate operations were not able to be computed (e.x. a kernel size greater than the input tensor size). Here, it is easy to distinguish the pattern of 4 convolutions of increasing kernel size, as well as the existence of branching layer outputs. Another feature that seems to appear frequently is a convolution layer with kernel size 3, which branches out of the main network's structure and is connected directly to the output layer. Finally, it is interesting to note that the dominant pattern seen in Fig. 4 evolved from many different genomes by mutation, and was then established due to its performance.

We chose the architecture with the best average performance, out of the architectures sampled at least 5 times, in order to further train and evaluate it. We select the average best, as we would like to avoid choosing a network trained to a high accuracy due to luck (initialization weights, sequence of non-deterministic operations). This is also in line with the work done in [12], as similar levels of variance are recorded (0.2% on average). The chosen network closely follows the dominant architecture (first 5 sequential layers in 4), followed by two CONV_3.H layers branching out after the pooling operation. By utilizing data augmentation (horizontal flipping and random erasing) on the train set,

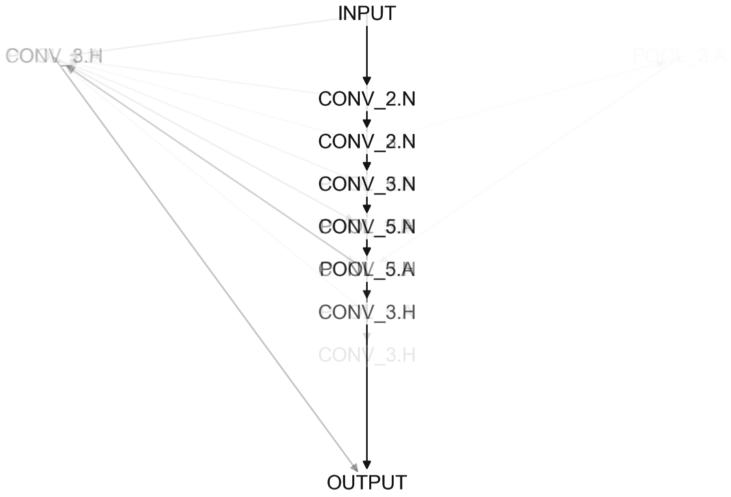

Fig. 4. Overlay of all genomes' architectures.

setting the number of channels to 256 (3.1 million parameters), as well as training the architecture for 20 epochs, we were able to achieve 94.46% test accuracy with this architecture. With a more modest number of 128 channels (791,338 parameters), we were able to achieve a performance of 94.26% accuracy. We call the architecture REMNet (Regularized Evolution for Macro-NAS Net) for ease of reference.

4.3 Discussion

When compared to networks found by similar macro NAS methodologies, applied to the same dataset, the network found by our method seems to perform marginally better. There are post-search training and inference techniques that produce even better-performing networks, such as test-set data augmentation and ensembles of neural networks, used in [1]. By utilizing those techniques, the authors are able to boost their network's performance to 94.7%, with test set augmentation and to 95.26% with ensembles. As we do not utilize any of these, we do not think it is fair to compare our results to those obtained by using them.

A summary of results obtained by similar methods is depicted in Table 2. One interesting detail is that our methodology needs less epochs to fully train the final architecture, compared to other methods. This can be attributed to two facts. First, we restrict the search phase training epochs to only 10. This seems to pose a regulatory effect on the network's structure, as networks that need a large number of epochs to train have poor performance when evaluated. This has also been observed in [13]. Furthermore, we utilize optimizers with adaptive learning rate. Thus, we do not need to employ complex learning rate policies, as in [1].

Table 2. Performance of similar algorithms

Method	Final accuracy	Search epochs	Training epochs
DENSER [1]	*94.23%* (**94.70%**, test set augment.)	**10**	400
Auto-Keras [7]	93.28%	200	200
DeepSwarm [3]	93.56%	50	100
NASH [5][a]	91.95%	20–105	205
REMNet-256	**94.46%**	**10**	**20**
REMNet-128	*94.26%*	**10**	**20**

[a] As implemented in [7]

4.4 Limitations

Although our method is able to produce better results when compared to similar methodologies, it is not able to produce better architectures than the state-of-the-art. This is partly due to the fact that we conduct a macro-search, and as such explore a greater search space. Furthermore, we do not implement any other methodology to directly compare ours, and as such comparisons may be unfair either to our advantage or disadvantage.

5 Conclusions and Future Work

As Deep Learning is becoming more popular and widely adopted, Neural Architecture Search is also increasingly studied. Many methods are able to produce state-of-the-art networks, by utilizing previous knowledge about the network's macro-structure, and thus conducting NAS on the micro (cell) level.

In this paper we have proposed a methodology for macro NAS, by combining elements of regularized evolution utilized for micro NAS [17], and DeepNEAT [13], utilized for macro-search. Compared to [17] we were able to conduct search not only on a cell-level, but also on a macro-architecture level. Compared to [13], we had less parameters to optimize and less operations to define, as we did not utilize crossover and we restricted the parameter space to 12 discrete values.

By applying our method on the Fashion-MNIST dataset, we were able to out-perform other macro NAS methodologies, when the post-search evaluation methods were similar. We designed networks capable of up to 94.46% accuracy on the test set, while requiring relatively few epochs to train. As such, we hope to continue our experimentation, in order to expand it to other datasets and network types, as well as study the generated architectures further.

Acknowledgements. This work was supported by computational time granted from the Greek Research & Technology Network (GRNET) in the National HPC facility - ARIS - under project ID DNAD. Furthermore, this research is funded by the University of Macedonia Research Committee as part of the "Principal Research 2019" funding program.

This paper's first author was supported by the Hellenic Foundation for Research and Innovation (HFRI) under the HFRI PhD Fellowship grant (Award Number: 20540).

References

1. Assunçao, F., Lourenço, N., Machado, P., Ribeiro, B.: Denser: deep evolutionary network structured representation. Genet. Program Evolvable Mach. **20**(1), 5–35 (2019)
2. Baccouche, M., Mamalet, F., Wolf, C., Garcia, C., Baskurt, A.: Sequential deep learning for human action recognition. In: Salah, A.A., Lepri, B. (eds.) HBU 2011. LNCS, vol. 7065, pp. 29–39. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25446-8_4
3. Byla, E., Pang, W.: DeepSwarm: optimising convolutional neural networks using swarm intelligence. In: Ju, Z., Yang, L., Yang, C., Gegov, A., Zhou, D. (eds.) UKCI 2019. AISC, vol. 1043, pp. 119–130. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-29933-0_10
4. Cheng, Z., Yang, Q., Sheng, B.: Deep colorization. In: Proceedings of the IEEE International Conference on Computer Vision, pp. 415–423 (2015)
5. Elsken, T., Metzen, J.H., Hutter, F.: Simple and efficient architecture search for convolutional neural networks. arXiv preprint arXiv:1711.04528 (2017)
6. He, K., Wang, Y., Hopcroft, J.: A powerful generative model using random weights for the deep image representation. In: Advances in Neural Information Processing Systems. pp. 631–639 (2016)
7. Jin, H., Song, Q., Hu, X.: Auto-keras: an efficient neural architecture search system. In: Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, pp. 1946–1956 (2019)
8. Kingma, D.P., Ba, J.: Adam: a method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014)
9. Kyriakides, G., Margaritis, K.G.: Towards automated neural design: an open source, distributed neural architecture research framework. In: Proceedings of the 22nd Pan-Hellenic Conference on Informatics, pp. 113–116 (2018)
10. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proc. IEEE **86**(11), 2278–2324 (1998)
11. Liu, C., et al.: Progressive neural architecture search. In: Proceedings of the European Conference on Computer Vision, ECCV, pp. 19–34 (2018)
12. Liu, H., Simonyan, K., Vinyals, O., Fernando, C., Kavukcuoglu, K.: Hierarchical representations for efficient architecture search. arXiv preprint arXiv:1711.00436 (2017)
13. Miikkulainen, R., et al.: Evolving deep neural networks. In: Artificial Intelligence in the Age of Neural Networks and Brain Computing, pp. 293–312. Elsevier (2019)
14. Nayman, N., Noy, A., Ridnik, T., Friedman, I., Jin, R., Zelnik, L.: Xnas: neural architecture search with expert advice. In: Advances in Neural Information Processing Systems, pp. 1975–1985 (2019)
15. Noy, A., et al.: Asap: Architecture search, anneal and prune. arXiv preprint arXiv:1904.04123 (2019)
16. Pham, H., Guan, M.Y., Zoph, B., Le, Q.V., Dean, J.: Efficient neural architecture search via parameter sharing. arXiv preprint arXiv:1802.03268 (2018)
17. Real, E., Aggarwal, A., Huang, Y., Le, Q.V.: Regularized evolution for image classifier architecture search. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 33, pp. 4780–4789 (2019)
18. Stanley, K.O., Miikkulainen, R.: Evolving neural networks through augmenting topologies. Evol. Comput. **10**(2), 99–127 (2002)

19. Wu, B., et al.: Fbnet: hardware-aware efficient convnet design via differentiable neural architecture search. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 10734–10742 (2019)
20. Wu, B., Iandola, F., Jin, P.H., Keutzer, K.: Squeezedet: unified, small, low power fully convolutional neural networks for real-time object detection for autonomous driving. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops, pp. 129–137 (2017)
21. Xiao, H., Rasul, K., Vollgraf, R.: Fashion-MNIST: a novel image dataset for benchmarking machine learning algorithms. arXiv preprint arXiv:1708.07747 (2017)
22. Zoph, B., Le, Q.V.: Neural architecture search with reinforcement learning. arXiv preprint arXiv:1611.01578 (2016)
23. Zoph, B., Vasudevan, V., Shlens, J., Le, Q.V.: Learning transferable architectures for scalable image recognition. In: Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 8697–8710 (2018)