DeepSwarm: Optimising Convolutional Neural Networks using Swarm Intelligence

Edvinas Byla and Wei Pang

Department of Computing Science, University
of Aberdeen, Aberdeen AB24 3UE, UK
e.byla.15@aberdeen.ac.uk,
pang.wei@abdn.ac.uk

Abstract. In this paper we propose DeepSwarm, a novel neural architecture search (NAS) method based on Swarm Intelligence principles. At its core DeepSwarm uses Ant Colony Optimization (ACO) to generate ant population which uses the pheromone information to collectively search for the best neural architecture. Furthermore, by using local and global pheromone update rules our method ensures the balance between exploitation and exploration. On top of this, to make our method more efficient we combine progressive neural architecture search with weight reusability. Furthermore, due to the nature of ACO our method can incorporate heuristic information which can further speed up the search process. After systematic and extensive evaluation, we discover that on three different datasets (MNIST, Fashion-MNIST, and CIFAR-10) when compared to existing systems our proposed method demonstrates competitive performance. Finally, we open source DeepSwarm[1] as a NAS library and hope it can be used by more deep learning researchers and practitioners.

Keywords: Ant Colony Optimization, Neural Architecture Search

1 Introduction

In recent years it has become increasingly challenging for human engineers to manually design deep neural architectures for specific tasks. This is mainly due to the following two facts: (1) modern deep neural architectures tend to be very complex with a lot of layers and hyperparameters; (2) one architecture might perform well on one dataset or on one type of problems but poorly on others. These two factors have resulted in a boom of research that tries to develop methods that can automate the design of neural architectures, the so-called neural architecture search [22].

In this paper we propose a novel neural architecture search method based on Swarm Intelligence (SI). To start with, we focus on Convolutional Neural Networks (CNN) [13], one of the most commonly used deep neural architectures. To discover new CNN architectures our method uses Ant Colony Optimization

[1] https://github.com/Pattio/DeepSwarm

(ACO) [5]. The motivation for using SI for NAS is due to the fact that SI possesses many appealing properties that could be helpful when dealing with NAS problems. This includes fault tolerance, decentralisation, scalability and ability to share and combine the knowledge, just to name a few. In particular, ACO has few distinct characteristics that make it naturally fit into the NAS domain: ACO is good at solving discrete problems which can be represented as graphs and it can easily adapt to dynamic environment (changing graph). Another significant motivating factor to use SI is the fact that the majority of its methods have not been explored in the context of NAS.

The novel contributions of this research are summarised as follows:

- We show that ACO can be used to effectively optimise CNNs.
- We use heuristic information when performing NAS based on ACO.
- We dynamically change the graph size and progressively search for the architectures when performing NAS based on ACO.

The rest of the paper is organised as follows: Section 2 presents related work; Section 3 introduces our proposed method; Section 4 presents the evaluation of our method; and Section 5 concludes the paper and explores possible future directions.

2 Related Work

Neural Architecture Search (NAS) is an automated process that aims to discover the best performing neural network architectures for a specific problem. Even though NAS research goes back as far as three decades [16], it has attracted new attention in recent years with the rapid development of deep learning, significant improvements in hardware, and growing interest of the machine learning community. Furthermore, even with this renewed interest from many deep learning researchers and practitioners it still seems that most of the existing NAS research predominantly focuses on using Evolutionary Algorithms [19, 21, 15], Bayesian Optimisation [4, 10], and Reinforcement Learning [24, 1, 25]. However, considering most of these approaches require huge amounts of computational resources, some new work which tries to reduce the computational costs have emerged [17, 8, 18]. For example, in [18] the authors proposed to use large computational graph which stores all the weights, and they reported that sharing these weights among child models could be 1000 times less computationally expensive than standard NAS approaches.

To the best of our knowledge, ACO was first applied to NAS problem in 2014 [20], and in their work ACO was used to optimise feed-forward neural networks. Furthermore, in their work the authors discovered that reusing the weights of the best solution can further improve the performance of their method. In 2015 ACO was used to optimise the structure of deep recurrent neural networks [3], where the authors try to address the problem of predicting general aviation flight data. The authors reported that using ACO they could achieve better prediction performance for airspeed, altitude, and pitch compared with the previous best

published results. Finally, in more recent work [7], ACO was used to optimise long short-term memory recurrent neural networks, and they achieved an increase in prediction accuracy, while also reducing the number of trainable weights by 55%.

It is noted that another relevant work to our research is the Progressive Neural Architecture Search (PNAS) approach [14]: similar to PNAS, the system proposed in this paper explores enormous CNN search space by using small incremental steps. In [14] the authors concluded that PNAS can achieve the same level of performance as the previous NAS approach [25] while being 8 times faster in terms of the required total computational time.

3 DeepSwarm

In this section we first present the details of the proposed DeepSwarm, and then we give the overall workflow.

As mentioned before, DeepSwarm search for new architectures in the order of increasing complexity similar to PNAS. At the beginning of a NAS task, DeepSwarm creates an internal graph which contains only the input node. Then a specified number of ants are generated. Next, one by one each ant is placed on the input node. After being placed on the input node each ant uses the Ant Colony System (ACS) [6] selection rule to select one of the available nodes in the next layer of CNN, and the ACS selection rule is as follows:

$$s = \begin{cases} \underset{u \in J_k(r)}{\arg\max}\{[\tau(r,u)] \cdot [\eta(r,u)]^{\beta}\}, & \text{if } q \leq q_0 \quad \text{(exploitation)}. \\ S, & \text{otherwise} \quad \text{(biased exploration)}, \end{cases} \tag{1}$$

In the above $\tau(r,u)$ denotes the pheromone amount on the edge that goes from node r to node u and $\eta(r,u)$ denotes the heuristic value associated with the edge going from node r to node u. Furthermore, $J_k(r)$ denotes a set of nodes that are available to visit from node r. The value of q is a random number uniformly distributed over $[0 \dots 1]$. Parameters $q_0 \in (0,1]$ and $\beta \in (0, \text{inf})$ control the algorithm's greediness and the relative importance of heuristic information. Finally, S is a random variable selected according to the probabilistic distribution defined by Equation (2):

$$p_k(r,s) = \begin{cases} \frac{[\tau(r,s)] \cdot [\eta(r,s)]^{\beta}}{\sum_{u \in J_k(r)} [\tau(r,u)] \cdot [\eta(r,u)]^{\beta}}, & \text{if } s \in J_k(r). \\ 0, & \text{otherwise.} \end{cases} \tag{2}$$

Once a node is selected the system checks if this node already exists in the graph at the depth of the selection. If this node is a new one which does not exist in the graph, it is added to the graph as a neighbour node to the previous node (i.e., the node where the ant was before the selection) so the subsequent ants can exploit the pheromone information. After an ant selects a particular node it also performs the same selection rule as defined by Equations (1) and (2) to select the attributes of that node (i.e. filter size, kernel size). When the selection

is completed the node is added to the ant's path. Once an ant reaches the current maximum allowed depth, its path is transformed into a neural network architecture which then gets evaluated. Furthermore, after an ant finishes a walk it performs ACS local pheromone update as defined by Equation (3) for each edge it has used:

$$\tau(r,s) \longleftarrow (1-\rho)\cdot\tau(r,s) + \rho\cdot\tau_0 \tag{3}$$

In the above, parameter ρ denotes the pheromone decay factor and parameter τ_0 is the initial pheromone value. This local update rule decays pheromone values so the other ants can be encouraged to explore other paths. After all ants are evaluated the best ant is found (the ant which found the architecture with the highest accuracy). This best ant then performs the ACS global pheromone update as defined by Equation (4), which increases the pheromone values for the edges found in the best path.

$$\tau(r,s) \longleftarrow (1-\alpha)\cdot\tau(r,s) + \alpha\cdot\Delta\tau(r,s), \tag{4}$$

where

$$\Delta\tau(r,s) = \begin{cases} C_{gb}, & \text{if } (r,s) \in \text{global-best-tour.} \\ 0, & \text{otherwise.} \end{cases} \tag{5}$$

Here parameter α controls pheromone evaporation and its range is $(0,1)$. C_{gb} is the cost of the global best tour (the best model accuracy). After the graph's current maximum allowed depth is increased, a new population of ants is generated. This cycle is repeated until the maximum depth (specified by the user) is reached. An illustrative example of NAS performed by DeepSwarm can be seen in Fig. 1, and the pseudocode is given in Algorithm 1.

We list several interesting outcomes of using ACO as a search strategy as follows: (1) weight reusability is straightforward to implement: we find the longest common sub-path in the graph and reuse the best weights from that sub-path, (2) the search space can be explored progressively as ants can adapt to the dynamic environment (when we expand the graph from depth n to $n+1$ we do not lose the information which was gathered up to depth $n+1$), and (3) because ACO uses domain-specific heuristics (Equations (1) and (2)) domain experts can easily provide their own knowledge to speed up the search further.

We finally point out the differences of DeepSwarm compared with previous work on ACO for optimising neural networks as well as PNAS [14] as mentioned in Section 2. First, DeepSwarm uses dynamic graphs and performs the search layer by layer progressively, and this is similar to PNAS [14]; while previous work [20, 3, 7] used static graphs and tried to search complete paths, for instance, in [20], each ant constructs a complete neural network at each iteration. Second, DeepSwarm differs from PNAS [14] as it uses heuristic information and individual searchers (ants) share information with each other; while PNAS employed a deterministic search strategy. Third, DeepSwarm reused the weights of the partial neural networks trained previously in order to improve the training speed,

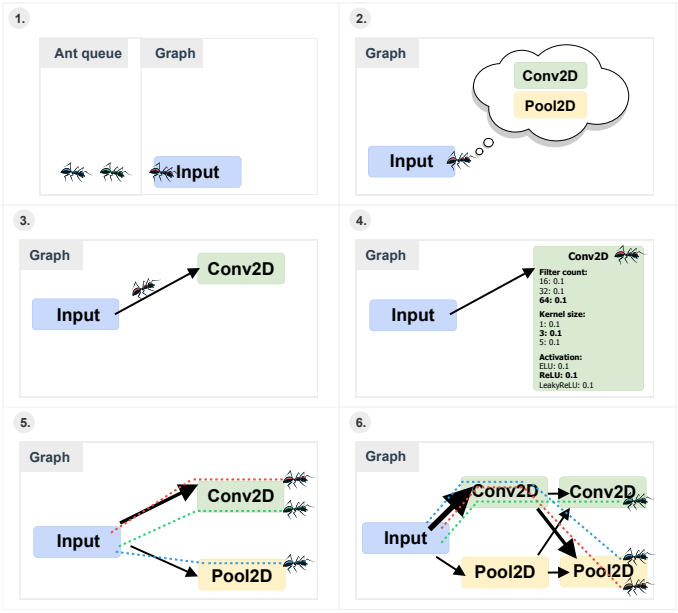

Fig. 1. An overview of the NAS process of DeepSwarm. (1) The ant is placed on the input node. (2) The ant checks what transitions are available. (3) The ant uses the ACS selection rule to choose the next node. (4) After choosing the next node the ant selects the node's attributes. (5) After all ants finished their tour the pheromone is updated. (6) The maximum allowed depth is increased and the new ant population is generated. **Note**: Arrow thickness indicates the pheromone amount, meaning that thicker arrows have more pheromone.

while most of the previous work did not reuse the previously trained weights except for [20].

4 Experiments

For the experimental design, three different datasets were chosen: (1) MNIST [12], (2) Fashion-MNIST [23], and (3) CIFAR-10 [11]. Each of these three datasets is quite different from the others and requires different CNN architectures to achieve the best results. As a result the combination of them is a good way to test the algorithm's robustness and performance. In order to evaluate our proposed method the baselines taken from [10] were used. All of our tests were carried out in the Google Colab environment (1x Tesla K80 GPU) [9] using a MacBook Pro (Early 2015 model) to interact with this environment. Note that even though in [10] they ran each method only for 12 hours, they used NVIDIA GeForce GTX 1080 Ti GPU, which according to a few benchmarks is approx-

Algorithm 1: DeepSwarm

```
Function search():
    graph = Graph() // build graph containing only the input node
    while graph.current_depth < max_depth do
        ants = generate_ants()
        best_ant = find_best(ants)
        graph.global_pheromone_update(best_ant)
        graph.increase_depth()
    return best_ant

Function generate_ants():
    ants = []
    for i = 0 to ant_count do
        ant = Ant()
        ant.path = generate_path()
        ant.evaluate()
        ants.append(ant)
        graph.local_pheromone_update(ant)
    return ants

Function generate_path():
    current_node = graph.input_node
    path = [current_node]
    for i = 0 to current_max_depth do
        if current_node.neighbours ⟵ ∅ then
            break
        current_node = aco_select_rule(current_node.neighbours)
        path.append(current_node)
    completed_path = complete_path(path) // completes the path if needed
    return path

Function aco_select(neighbours):
    foreach neighbour ∈ neighbours do
        probability = neighbour.pheromone × neighbour.heuristic
        probabilities ← probability
        denominator += probability
    if random.uniform(0, 1) ≤ greediness then
        max_index = probabilities.index(max(probabilities))
        return neighbours[max_index]
    probabilities = probabilities / denominator
    neighbour_index = wheel_selection(probabilities)
    return neighbours[neighbour_index]
```

imately 2-3 times faster than our selected Tesla K80 GPU. This is the reason why we are not going to constrain our runs to 12 hours.

4.1 Evaluation Procedure

When evaluating the system the following procedure was followed: (1) create a new Google Colab instance, (2) import the source code of the library, (3) split the training set 90-10 to training and validation sets, (4) run the algorithm until the max depth is reached, (5) take the best found network, (6) for CIFAR-10 dataset apply standard data augmentation (random horizontal flips, rotation and scaling) to the training data, (7) train the best found network for additional 50 epochs on the augmented data, (8) load the weights which showed the best performance on the validation set during those 50 epochs, and (9) evaluate the network with these best weights on the testing data.

4.2 Ant Count

The ant count (the number of ants used during search) is one of the most important hyperparameters in DeepSwarm. This is because it is a trade-off between the performance of the final model and the run-time of the algorithm. In order to find a good trade-off, we ran multiple tests by exponentially increasing the ant count. Furthermore, we split the results into two parts: before and after the final training. Before the final training is a part where DeepSwarm finds potentially the best model and after the final training is the part where the best found model is trained for an additional 50 epochs on augmented data. The reason for this choice is that the results before the final training can reflect the real implications that the ant count has on the error rate, whereas the results after the final training can show how the ant count can affect the generalisation. This follows from the fact that before the final training the models are trained on the same data, whereas during the final training the models are trained on the augmented data which can show how well they can learn. The results before the final training are presented in Fig. 2, the results after the final training can be seen in Fig. 3, and the run time is shown in Fig. 4.

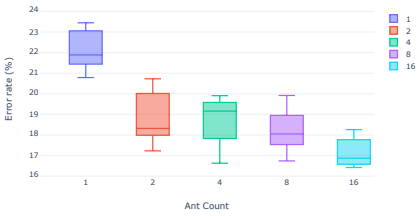

Fig. 2. The error rate on the CIFAR-10 dataset before the final training across five separate trials.

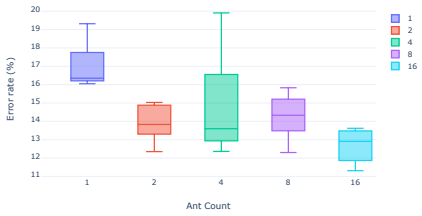

Fig. 3. The error rate on the CIFAR-10 dataset after the final training across five separate trials.

Looking at the results one can see that changing the ant count from 1 to 2 had a significant impact on the error rate. This finding was to be expected

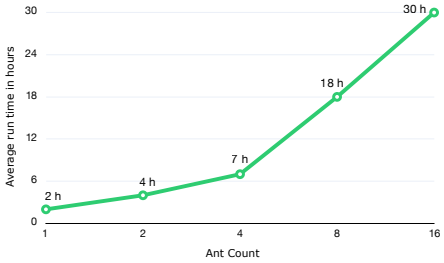

Fig. 4. The average run time (across five different trials) in hours for different ant counts on the CIFAR-10 dataset.

because when only one ant exists both exploration and exploitation must suffer. The exploration suffering is associated with the fact that the ant can only explore one architecture per depth, meaning that only a small subset of available architectures will be explored. The exploitation degradation occurs because at each depth acquired knowledge scales only linearly, for example, at depth 3 the ant will only know about 2 other architectures. Furthermore, having only one ant will result in rather greedy behaviour where the same ant will explore the same sub-tree in the graph and will only rarely explore the parallel sub-trees. We further noticed that even though doubling the ant count almost doubles the run time, it will not always result in drastically improved performance. For example, when we increased the ant count from 4 to 8 ants the run time increased from 7 hours to 18 hours, while the average error rate decreased only by 0.13%. The most drastic changes in the error rate happened when the ant count was changed from 1 to 2 (3.11% decrease) and from 8 to 16 (2.1% decrease). However, due to the computational restrictions we did not test ant counts beyond 16 which means that there might be even bigger performance improvements when going beyond 16 ants.

4.3 Greediness

Another important hyperparameter of DeepSwarm is greediness. As mentioned in Section 3, the greediness is used in Equation (1) to decide how greedy each ant should be. As greediness can be defined in the range from 0.0 to 1.0, we test the greediness with its value increases from 0 to 1 at a step size of 0.25. Furthermore, similarly to the ant count, the results were divided into before and after the final training. The results before the final training are shown in Fig. 5 and the results after the final training are shown in Fig. 6.

Looking at the results it seems that when selecting the greediness for the algorithm one should never go to extremes as this will most likely result in poor

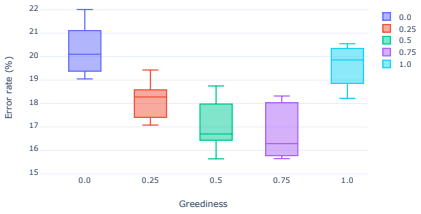

Fig. 5. The error rate on the CIFAR-10 dataset before the final training across five separate trials.

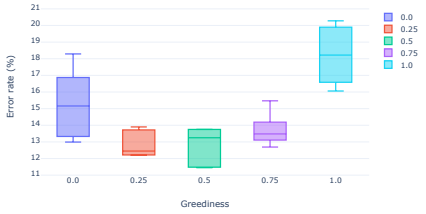

Fig. 6. The error rate on the CIFAR-10 dataset after the final training across five separate trials.

performance. The more general insight we gathered from the results was that selecting the greediness values which were close towards the middle (0.5) resulted in the best performance. The reason why the extremely greedy ants perform poorly is as follows: at the beginning of the search they base their search purely on the heuristic information and then, once the pheromone is laid on the graph, all of them will reuse the same path, therefore generating the same architecture. Furthermore, the local pheromone update rule will not help here because once the pheromone evaporates these greedy ants will use the same heuristics which will result in the same paths being chosen again. In contrast, the ants with no greediness will always base each of their decisions only on the wheel selection without exploiting the gathered information (as the first part in Equation 1 is always skipped) and because during the path generation an ant needs to make a lot of these decisions (choosing the next node and each attribute), a substantial part of them will be random, which will result in a poor performance. Another interesting observation was that the greedy models tend to generalise worse than the less greedy ones. For example, the average error rate difference before the final training between 0.25 greediness and 0.75 greediness was 1.32% (18.12% and 16.80% respectively), but after the final training, the difference was -0.83% (12.89% and 13.72% respectively). Furthermore, we noticed that the greediness had some impact on the average network depth, for example, the best architectures which were found using no greediness, were on average five layers deeper than the ones which were found using 1.0 greediness. As a result of that, these less greedy architectures had more regularisation and feature extraction. We believe that this could be the reason why these less greedy architectures were generalising better during the final training.

4.4 Accuracy

In order to compare the performance of DeepSwarm with that of other methods, we report the average and best performance achieved during the five separate runs on three different datasets. These results are shown in Table 1. From these results we can see that on the MNIST dataset from all of the methods DeepSwarm showed the best performance. When compared with the straightforward

methods (random and grid search [2]) DeepSwarm showed a significantly lower error rate (1.79%, 1.68% versus 0.46%). On the Fashion-MNIST dataset, DeepSwarm achieved the lowest error rate and once again proved to be superior to the straightforward methods which had almost a two times bigger error rate (11.36%, 10.28% versus 6.75%). Finally, on the CIFAR-10 dataset, even though DeepSwarm managed to find the architecture with the lowest error rate (11.31%), on average its performance was not as good as some other methods. Overall on all of the three datasets, DeepSwarm still produced very competitive and promising results. To see the best architectures discovered by DeepSwarm please visit following external resource[2].

Method	MNIST	Fashion-MNIST	CIFAR-10
RANDOM	1.79%	11.36%	16.86%
GRID	1.68%	10.28%	17.17%
SPMT	1.36%	9.62%	14.68%
SMAC	1.43%	10.87%	15.04%
SEAS	1.07%	8.05%	12.43%
NASBOT	N/A	N/A	12.30%
AutoKeras BFS	1.56%	9.13%	13.84%
AutoKeras BO	1.83%	7.99%	12.90%
AutoKeras BFS	0.55%	7.42%	**11.44**%
DeepSwarm Average	**0.46%**	**6.75%**	12.70%
DeepSwarm Best	0.39%	6.44%	11.31%

Table 1. The error rates on the CIFAR-10 dataset.

4.5 Discussion

Even though there exists a NAS approach developed by Google Brain [25] which can achieve better results than DeepSwarm on the CIFAR-10 dataset, we think that it would be not fair to compare our work with theirs for the following reasons: (1) they used 400 GPUs (also their GPUs were much more powerful than the one used in our experiments) for 4 days, (2) they used skip and add connections which are not implemented into DeepSwarm yet. We also point out that as they did not open source their code, it is not easy for us to test their approach in our environment to compare the performance difference. Nevertheless, based on the results seen in Section 4.4 DeepSwarm proved to be a competitive approach against already existing NAS methods. However, there is still some work that needs to be done in order to further improve DeepSwarm. We think that the two main components that can be added in the future are skip and add nodes. Adding these two components would allow DeepSwarm to search for more complex architectures which in turn could substantially improve the overall learning

[2] https://edvinasbyla.com/assets/images/best-architectures.pdf

performance. Finally, we list the main advantages of DeepSwarm compared with other existing NAS systems as follows:

- DeepSwarm offers competitive performance. As shown in Section 4.4, on all 3 datasets DeepSwarm can achieve comparable or better results than the other NAS systems.
- DeepSwarm can look for diverse structures. DeepSwarm does not enforce a specific structure, which allows it to find novel and interesting architectures.
- DeepSwarm can offer fast search. As mentioned earlier, DeepSwarm is built to search for architectures progressively and has a mechanism to reuse the old weights which boosts its performance.
- DeepSwarm allows the users to provide heuristic information which can further speed up the search process.
- DeepSwarm is easy to use. To start the neural architecture search a user just needs to write a few lines of code (see detailed instructions on DeepSwarm's GitHub page).
- DeepSwarm is easy to be further developed and extended. As we open source DeepSwarm and share it with the wider machine learning community, other researchers can further develop and extend DeepSwarm.

5 Conclusion and Future Work

In this paper we presented DeepSwarm and demonstrated that Swarm Intelligence can be used to effectively tackle NAS problems. After evaluating DeepSwarm we discovered that when compared to other similar methods it can show competitive performance. Furthermore, we open source DeepSwarm[3] and share it with the community, and we hope more people will benefit from it and further develop it.

The main contribution of this work is to show that ACO can be used to effectively search for optimal CNN architectures. Our second contribution is to demonstrate that domain expert knowledge can be successfully incorporated into ACO based NAS. The final contribution of this work is to show that progressive architecture search approach can be applied to ACO based NAS methods.

For future work we propose to explore the following directions: (1) implement skip and add connections which would allow ants to look for more complex architectures, (2) try to use ACO to perform cell based search (similar to [25]) rather than the full architecture search, (3) compare conventional search method with the progressive search when ACO is applied to NAS problem, and (4) explore ACO in other deep learning contexts i.e. find which neurons to drop in the dropout layer.

References

1. Baker, B., Gupta, O., Naik, N., Raskar, R.: Designing neural network architectures using reinforcement learning. arXiv preprint arXiv:1611.02167 (2016)

[3] https://github.com/Pattio/DeepSwarm

2. Bergstra, J., Bengio, Y.: Random search for hyper-parameter optimization. Journal of Machine Learning Research **13**(Feb), 281–305 (2012)
3. Desell, T., Clachar, S., Higgins, J., Wild, B.: Evolving deep recurrent neural networks using ant colony optimization. In: European Conference on Evolutionary Computation in Combinatorial Optimization, pp. 86–98. Springer (2015)
4. Domhan, T., Springenberg, J.T., Hutter, F.: Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In: IJCAI, vol. 15, pp. 3460–8 (2015)
5. Dorigo, M.: Optimization, learning and natural algorithms. PhD Thesis, Politecnico di Milano (1992). URL https://ci.nii.ac.jp/naid/10000136323/en/
6. Dorigo, M., Gambardella, L.M.: Ant colony system: a cooperative learning approach to the traveling salesman problem. IEEE Transactions on evolutionary computation **1**(1), 53–66 (1997)
7. ElSaid, A., Jamiy, F.E., Higgins, J., Wild, B., Desell, T.: Using ant colony optimization to optimize long short-term memory recurrent neural networks. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 13–20. ACM (2018)
8. Elsken, T., Metzen, J.H., Hutter, F.: Simple and efficient architecture search for convolutional neural networks. arXiv preprint arXiv:1711.04528 (2017)
9. Google: https://colab.research.google.com
10. Jin, H., Song, Q., Hu, X.: Efficient neural architecture search with network morphism. arXiv preprint arXiv:1806.10282 (2018)
11. Krizhevsky, A., Nair, V., Hinton, G.: The cifar-10 dataset. online: http://www. cs. toronto. edu/kriz/cifar. html p. 4 (2014)
12. LeCun, Y.: The mnist database of handwritten digits. http://yann. lecun. com/exdb/mnist/ (1998)
13. LeCun, Y., Bengio, Y., Hinton, G.: Deep learning. nature **521**(7553), 436 (2015)
14. Liu, C., Zoph, B., Neumann, M., Shlens, J., Hua, W., Li, L.J., Fei-Fei, L., Yuille, A., Huang, J., Murphy, K.: Progressive neural architecture search. In: Proceedings of the European Conference on Computer Vision (ECCV), pp. 19–34 (2018)
15. Miikkulainen, R., Liang, J., Meyerson, E., Rawal, A., Fink, D., Francon, O., Raju, B., Shahrzad, H., Navruzyan, A., Duffy, N., et al.: Evolving deep neural networks. In: Artificial Intelligence in the Age of Neural Networks and Brain Computing, pp. 293–312. Elsevier (2019)
16. Miller, G.F., Todd, P.M., Hegde, S.U.: Designing neural networks using genetic algorithms. In: ICGA, vol. 89, pp. 379–384 (1989)
17. Negrinho, R., Gordon, G.: Deeparchitect: Automatically designing and training deep architectures. arXiv preprint arXiv:1704.08792 (2017)
18. 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)
19. Real, E., Moore, S., Selle, A., Saxena, S., Suematsu, Y.L., Tan, J., Le, Q., Kurakin, A.: Large-scale evolution of image classifiers. arXiv preprint arXiv:1703.01041 (2017)
20. Salama, K., Abdelbar, A.M.: A novel ant colony algorithm for building neural network topologies. In: International Conference on Swarm Intelligence, pp. 1–12. Springer (2014)
21. Suganuma, M., Shirakawa, S., Nagao, T.: A genetic programming approach to designing convolutional neural network architectures. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 497–504. ACM (2017)
22. Wistuba, M., Rawat, A., Pedapati, T.: A survey on neural architecture search (2019)

23. Xiao, H., Rasul, K., Vollgraf, R.: Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. arXiv preprint arXiv:1708.07747 (2017)
24. Zoph, B., Le, Q.V.: Neural architecture search with reinforcement learning. arXiv preprint arXiv:1611.01578 (2016)
25. 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)