Reinforcement Learning for Architecture Search by Network Transformation

Han Cai¹, Tianyao Chen¹, Weinan Zhang¹, Yong Yu¹, Jun Wang²
¹Shanghai Jiao Tong University, ²University College London
{hcai, try-skycn, wnzhang}@apex.sjtu.edu.cn

Abstract

Deep neural networks have shown effectiveness in many challenging tasks and proved their strong capability in automatically learning good feature representation from raw input. Nonetheless, designing their architectures still requires much human effort. Techniques for automatically designing neural network architectures such as reinforcement learning based approaches recently show promising results in benchmarks. However, these methods still train each network from scratch during exploring the architecture space, which results in extremely high computational cost. In this paper, we propose a novel reinforcement learning framework for automatic architecture designing, where the action is to grow the network depth or layer width based on the current network architecture with function preserved. As such, the previously validated networks can be reused for further exploration, thus saves a large amount of computational cost. The experiments on image benchmark datasets have demonstrated the efficiency and effectiveness of our proposed solution compared to existing automatic architecture designing methods.

1 Introduction

The great success of deep neural networks in various challenging applications [15, 3, 23] has led to a paradigm shift from feature designing to architecture designing, which still remains a laborious task and requires human expertise. In recent years, many techniques for automating the architecture design process have been proposed [24, 5, 4, 35, 20, 19], among which reinforcement learning (RL) based approaches [4, 35] have demonstrated promising results of designing competitive models against human-designed models on some benchmark datasets. Under the RL formulation, an agent is introduced as the meta controller to generate architectures by taking a sequence of actions, and then receives the validation performances of the generated architectures as reward signals, which are further used to update the agent. Specifically, in current RL based approaches, using convolutional neural network (CNN) architecture designing as an example, the agent sequentially chooses hyperparameters of CNN layers, e.g. number of filters, filter size and stride for a convolutional layer, in a bottom-up fashion from an empty start.

Despite the promising results as reported, learning such an RL agent largely relies on getting the accurate validation performance of each generated architecture that requires carefully and sufficiently training corresponding network on the real data, which is prohibitively long for architecture designing, especially when the network is large. Similar issue happens to other automatic architecture designing approaches that are also guided by validation performances. In [35], the authors proposed to use distributed training and asynchronous parameter updates to speedup the architecture design process, but the total computational cost was not reduced. While in [4], the authors used a quick and aggressive training scheme during exploring the architecture space, which, however, may cause models to underperform. Moreover, since the agent is asked to generate the whole architecture from scratch and the reward signal is available only at the end, the required action sequence of designing an architecture can be too long for the agent to learn when we expect it to design deeper models.
In fact, during the architecture design process, many different networks are trained for the same task. Beside their final validation performances which are used as the reward signals, the agent should also have access to their architectures, weights, training curves etc., which contain abundant knowledge and can be leveraged to accelerate the architecture design process just like human experts [7].

In this paper, we propose a novel automatic architecture designing framework based on reinforcement learning, where the agent takes network transformation actions such as widening a certain layer (more units or filters), inserting a layer, adding skip connections etc., given an existing network trained on the same task. Furthermore, to reuse weights, we consider the special family of network transformation actions named Net2Net operations [7], which allow to initialize the new network to represent the same function as the given network but use different parameterization and can be further trained to improve the performance. By allowing the agent to start from an existing network and reuse its weights instead of searching from scratch or training each generated architecture from the beginning, we are able to explore the architecture space more efficiently and accurately. The experiments on CIFAR-10 show that our agent with very limited computational resources compared to existing automatic architecture designing methods, is able to design very competitive models against both human-designed models and automatically designed models.

2 Background and Related Work

Automatic Architecture Designing: Neuro-evolution algorithms which mimic the evolution processes in the nature, are one of the earliest automatic architecture designing methods [18, 28]. Recently, authors in [20] showed that neuro-evolution algorithms are capable of constructing large networks which can match the performances of human-designed models. However, they are actually search-based methods and require enormous computational power to work well, which makes them less practical at a large scale. Besides, Bayesian optimization has also been applied for automatically selecting network architectures [6, 5, 24], but these methods only search models from a fixed-length space and are unable to generate variable-length network architectures.

Reinforcement learning has recently been introduced in automatic architecture designing and have shown strong empirical results [4, 35]. The authors in [4] presented a Q-learning agent to sequentially pick CNN layers; the authors in [35] used an auto-regressive recurrent network to generate a variable-length string that specifies the architecture of a neural network and trained the recurrent network with policy gradient. The key distinction of our work from previous work is that we ask the agent to take network transformation actions and allow to reuse the existing networks trained on the same task. Further discussions are provided in Section 3.4.

Network Transformation and Knowledge Transfer: Generally, any modification to a given network can be viewed as a network transformation operation. In this paper, since we aim to utilize knowledge stored in previously trained networks, we focus on the kind of network transformation operations, which are able to reuse pre-existing models. The idea of reusing pre-existing models or knowledge transfer between neural networks has been studied before. Net2Net techniques introduced in [7] describe two specific function-preserving transformations, namely Net2WiderNet and Net2DeeperNet, which initialize a wider (Net2WiderNet) or deeper (Net2DeeperNet) student network to represent the same functionality of the given teacher network and have proved to significantly accelerate the training of the student network especially for large networks. Similar function-preserving network transforming schemes have also been proposed in ResNet particularly for training very deep architectures [10]. Besides, the network compression technique presented in [9] prunes less important connections (low-weight connections) to reduce the size of neural networks without affecting their accuracy.

In this paper, on the other hand, we focus on training an agent to automatically learn to take network transformation actions to efficiently and accurately explore the architecture space to find high performance architectures. Specifically we mainly consider Net2Net operations while other kind of network transformation operations can be easily incorporated under the same framework. To the best of our knowledge, it is the first work trying to learn an intelligent agent to perform network transformation operations.

Reinforcement Learning and Meta-Learning: Our work is based on RL, techniques for training the agent to maximize the cumulative reward in a sequential agent-environment interaction process.
3 Learning to Perform Network Transformation

In this section, we first introduce a general framework for learning an intelligent agent to automatically take network transformation actions. We will further show how the agent can be trained via policy gradient methods to maximize the expected validation performances of the result networks. Finally, we will discuss the connections of our work to previous automatic architecture designing approaches.

We consider to learn an agent to generate network transformation actions given the network architecture which can be specified with a variable-length string [35]. To be able to generate various types of network transformation actions while keeping the agent simple, we use a shared encoder network to learn a low-dimensional representation of the given architecture, which is then fed into each separate actor network to generate a certain type of network transformation actions. Furthermore, to handle variable-length network architecture as input and take the whole input architecture into consideration when making decisions, the encoder network is implemented with a bidirectional recurrent network [22] in this work. The overall framework is illustrated in Figure 1, which is an analogue of encoder-decoder framework in end-to-end sequence to sequence learning [30, 3].

Given the low dimensional representation of the input network architecture, each actor network makes necessary decisions for taking a certain type of network transformation actions. In this work, we introduce two specific actor networks, namely Net2Wider actor and Net2Deeper actor which correspond to Net2WiderNet and Net2DeeperNet (Section 2) respectively.

3.1 Net2Wider Actor

Net2WiderNet operation allows to replace a layer with a wider layer, meaning more units for fully-connected layer, or more filters for convolutional layers, while preserving the functionality. In our work, to be flexible and efficient, the Net2Wider actor simultaneously determines whether each layer should be extended. Specifically, for each layer, this decision is carried out by a shared sigmoid classifier given the hidden state of the layer learned by the bidirectional encoder network. Moreover, we follow previous work [4, 35] and search the number of filters for convolutional layers and units for fully-connected layers in a discrete space. Therefore, if the Net2Wider actor decides to widen a layer, the number of filters or units of the layer increases to the next discrete level, e.g. from 32 to 48, and the weights are reparameterized to preserve the functionality of the given network. The structure of Net2Wider actor is shown in Figure 2(a). For detailed reparameterization scheme for preserving the functionality we refer to the original Net2Net work [7].
Furthermore, while Net2WiderNet operation in [7] only modifies the number of filters or units of a layer, we find a simple way to increase the filter size of a convolutional layer with the functionality preserved, which is more flexible for automatic architecture designing. Consider the most commonly used setting, i.e. the width and the height of the filter are odd and the zero-padding parameter is set to preserve the spatial size of the input volume, for any kernel $K$ with width $k_x$, height $k_y$, input channel $f_i$ and output channel $f_o$, expressed as a tensor of shape $(k_x, k_y, f_i, f_o)$, it can be widen to another kernel $K'$ of shape $(k'_x, k'_y, f_i, f_o)$ where $k_x, k'_x, k_y$ and $k'_y$ are odd and $k'_y > k_y, k'_x > k_x$, with functionality preserved as long as:

$$K'[i, j, p, l] = \begin{cases} K[i - \delta_x, j - \delta_y, p, l] & \delta_x \leq i < k'_x - \delta_x, \delta_y \leq j < k'_y - \delta_y \\ 0 & \text{otherwise} \end{cases}$$

where $\delta_x = \frac{k'_x - k_x}{2}$ and $\delta_y = \frac{k'_y - k_y}{2}$. As such, we can modify both the filter number of a convolutional layer and its filter size while preserving the functionality, thereby widen the search space.

### 3.2 Net2Deeper Actor

Net2DeeperNet operation allows to insert a new layer that is initialized as adding an identity mapping between two layers so as to preserve the functionality. For a new convolutional layer, the kernel is set to be identity filters while for a new fully-connected layer, the weight matrix is set to be identity matrix. So the new layer will have the same number of filters or units as the layer below at first, and can further get wider when Net2WiderNet operation is performed on it. To fully preserve the functionality, Net2DeeperNet operation has a constraint on the activation function $\phi$, i.e. $\phi$ must satisfy $\phi(I(\phi(v))) = \phi(v)$ for all vectors $v$. This property holds for rectified linear activation (ReLU) but fails for sigmoid and tanh activation. Additionally, when using batch normalization [12], we need to set output scale and output bias of the batch normalization layer to undo the normalization, not to be just initialized as ones and zeros. Further details about the Net2DeeperNet operation is provided in the original paper [7].

The structure of the Net2Deeper actor is shown in Figure 2(b). Similar to previous RL based approaches which pick the next layer at each step, in our work, we allow the Net2Deeper actor to insert one new layer at each step. Specifically the Net2Deeper actor first determines the index of the layer where the new layer is inserted in and then determines parameters of the new layer, which are carried out by separate softmax classifiers. For a new convolutional layer, the agent needs to determine the filter size and the stride while for a new fully-connected layer, no parameter prediction is needed. In CNN architectures, any fully-connected layer should be on the top of all convolutional and pooling layers. To avoid resulting in unreasonable architectures, if the Net2Deeper actor decides to insert a new layer after a convolutional layer or pooling layer, the new layer is restricted to be a convolutional layer, and if the agent decides to insert a new layer after a fully-connected layer, the new layer must be a fully-connected layer.

### 3.3 Training via Policy Gradient

The shared encoder network and separate actor networks, as described above, form a stochastic policy (with parameters $\theta$) of the agent which, at each step, takes a network architecture (denoted as $s_t$) as input and samples network transformation actions (denoted as $a_t$) to be performed on the input
network architecture and leads to a new network architecture \(s_{t+1}\). After \(T\) steps of transformations, we get the final network architecture \(s_{T+1}\) along with its weights transferred from the initial input network \(s_1\), which is then trained in the real data and gets a performance score (typically accuracy) \(R\) on the held-out validation set. With the validation performance \(R\) as the reward signal, we can use policy gradient methods to train the agent to maximize the expected validation performances of the result networks, i.e. \(J(\theta) = \mathbb{E}_{\theta}[R]\). We use the REINFORCE algorithm [33] in this work as done in [35], while other advanced policy gradient methods [13, 21] can also be applied. The gradient of \(J(\theta)\) w.r.t the parameters \(\theta\) is given as:

\[
\nabla_{\theta} J(\theta) = \sum_{t=1}^{T} \mathbb{E}_{\theta} \left[ \nabla_{\theta} \log P(a_t|s_t; \theta) R \right],
\]

(2)

which is usually empirically estimated by unbiased sampling:

\[
\nabla_{\theta} J(\theta) \approx \frac{1}{m} \sum_{k=1}^{m} \sum_{t=1}^{T} \nabla_{\theta} \log P(a_t|s_t; \theta) R_k.
\]

(3)

Additionally, to reduce the variance of the above gradient estimation, a baseline function is incorporated:

\[
\nabla_{\theta} J(\theta) \approx \frac{1}{m} \sum_{k=1}^{m} \sum_{t=1}^{T} \nabla_{\theta} \log P(a_t|s_t; \theta) (R_k - b).
\]

(4)

Since the baseline function \(b\) does not depend on the parameters \(\theta\), it is still an unbiased estimation. In this work, we follow [35] and use an exponential moving average of the validation performances of previous sampled architectures as the baseline function.

### 3.4 Connections to Previous Work

In this section, we compare our proposed agent which learns to take network transformation actions with existing methods for automatically designing network architectures. The process of automatic architecture designing can be viewed as an agent navigating in the large architecture space, searching for high performance architectures. Previous two RL based approaches [4, 35] pose schemes with sequentially picking the next layer, which have two very restrictive constraints on how the agent navigates in the architecture space: (i) the agent is restricted to start from scratch at the beginning of each episode without reusing any weights of previously trained architecture, which makes the RL training very time consuming; (ii) the agent is restricted to add a layer at the top given existing layers below at each step.

In this work, we relax the first constraint by allowing the agent to start from any existing network architectures rather than empty, and relax the second constraint by exploring a different action space, network transformation actions which can reuse weights. As a first trial, we try two simple operations, i.e. Net2WiderNet and Net2DeeperNet, in this paper. It is of great interest to explore other actions, including network compression [9], to efficiently find high performance network architectures.

Furthermore, since the agent is not restricted to start from scratch now, choosing which network architecture to start from is also very interesting. For example, it is possible to maintain a population of high performance network architectures as is done in neuro-evolution algorithms (Section 2) and sample architectures from the population as the start points to explore new architectures while updating the agent and evolving the population at the same time, which is a combination of neuro-evolution algorithms and RL based approaches. It is also possible to design other complex selection schemes or even learn another agent to select the starting points. We think it is an interesting future direction that is worth exploring.

### 4 Experiments and Results

In line with previous work [4, 35, 20], we apply our agent to an image classification task with CIFAR-10, a benchmark dataset which requires well-designed architectures to reach high performance, to efficiently find high performance CNN architectures. For experiment reproducibility, we publicize our code including discovered top architectures on CIFAR-10 along with their weights.

---

1. Experiment code and discovered top architectures along with weights: [https://github.com/han-cai/RL4AS_NetTrans](https://github.com/han-cai/RL4AS_NetTrans)
Table 1: Simple start point networks. $C(n, f, l)$ denotes a convolutional layer with $n$ filters, filter size $f$ and stride $l$; $P(f, l)$ denotes a pooling layer with filter size $f$ and stride $l$; $FC(n)$ denotes fully-connected layer with $n$ units; $SM(n)$ denotes a softmax layer with $n$ output units.

| Model Architecture                                      | Validation Accuracy (%) |
|--------------------------------------------------------|-------------------------|
| Start net 1 $C(32, 3, 1), P(2, 2), C(32, 3, 1), P(2, 2)$, $FC(64), SM(10)$ | 73.10                   |
| Start net 2 $C(6, 3, 1), P(2, 2), C(6, 3, 1), P(2, 2)$, $FC(16), SM(10)$       | 36.44                   |

4.1 Experiment Details

Dataset: The CIFAR-10 dataset [14] consists of 50,000 training images and 10,000 test images. Following previous work, we randomly sample 5,000 images from the training set to form a validation set for computing the reward signals while using the remaining 45,000 images for training. We use a standard data augmentation scheme that is widely used for CIFAR-10 [11, 35]. Specifically, we first zero pad the images with 4 pixels on each side, which are then randomly cropped to $32 \times 32$, and further perform random horizontal flips on the cropped images. Finally, we normalized the data using the channel means and standard deviations.

Architecture Space: Our agent explores the architecture space consisting of convolutional, pooling and fully-connected layers, by taking Net2Wider or Net2Deeper actions given a simple network as a start point. Due to resource and time constraints, our agent searches layer parameters in a discrete and limited space as is done in [4, 35]. For every convolutional layer, the filter size is chosen from {1, 3, 5} and the number of filters is chosen from {32, 48, 64, 80, 96, 128, 192, 256}, while the stride is fixed to be 1. For every fully-connected layer, the number of units is chosen from {64, 96, 128, 192, 256, 320, 416, 512}. Additionally, we use ReLU as non-linearities and batch normalization for each convolutional or fully-connected layer.

Training: We use a two-layer bidirectional LSTM with 100 hidden units on each layer as the encoder network (Figure 1). Similar to [35], all weights are initialized by a truncated normal initializer with 0 mean and 0.08 standard deviation, and the agent is trained with the ADAM optimizer with a learning rate of 0.0006.

At each step, the agent samples 10 networks by taking network transformation actions. Since the sampled networks are not trained from scratch but we reuse weights of the given network in our scenario, they are then trained for 20 epochs, a relative small number compared to 50 epochs in [35]. Besides, we use a smaller initial learning rate for this reason. Other settings are similar for training the networks in CIFAR-10 as used in [11, 35]. Specifically, we use the SGD with a Nesterov momentum [29] of 0.9, a weight decay of 0.0001, a batch size 64. The initial learning rate is 0.025 and is reduced to 0.005 after 15 epochs. The accuracy in the held-out validation set is used as the reward signal for each sampled network. Our experiments are conducted using Tensorflow [1] with 5 GeForce GTX 1080 GPUs.

4.2 Results on CIFAR-10

Search High Performance Architectures with A Simple Start Network: We conduct the first experiment on CIFAR-10 to search high performance architectures, using a simple network architecture, i.e. start net 1 in Table 1, which has a poor performance (73.10% accuracy) in the held-out validation set. Since our agent is not restricted to start from empty, it is possible to divide the whole architecture search process into several stages where the best architecture from previous stages is used as the start point for the current stage, which makes each stage easier for the agent to learn and search, since the agent do not have to take a very long sequence of actions to get a reward signal. And therefore save a lot resources and time.

In this experiment, we divide the whole process into three stages. At the first stage, we allow our agent to take 5 steps of Net2Deeper action, starting from the start net 1. After 160 networks are sampled, we take the network which achieves the best validation accuracy currently and train it with a longer period of time (100 epochs) to be used as the start point for the second stage. Similarly, in the second stage, we allow the agent to take 4 steps of Net2Wider action. And finally, in the third stage, we allow the agent to take 2 steps of Net2Deeper action and 3 steps of Net2Wider action.
Figure 3: Progress of three stages architecture search on CIFAR-10.

Table 2: Test error rate comparison on CIFAR-10 with human-designed architectures and automatically discovered architectures.

| Model                        | Depth | Params     | Test Error Rate (%) |
|------------------------------|-------|------------|---------------------|
| Maxout [8]                   | -     | -          | 9.38                |
| Network in Network [17]      | -     | -          | 8.81                |
| Highway Network [27]         | -     | -          | 7.72                |
| All-CNN [26]                 | -     | -          | 7.25                |
| ResNet [10]                  | 110   | 1.7M       | 6.61                |
| Trees+Max-Avg [16]           | -     | -          | 6.05                |
| Wide ResNet [34]             | 16    | 11.0M      | 4.81                |
| DenseNet [11]                | 190   | 25.6M      | 3.46                |
| MetaQNN [4]                  | 8     | 11.18M     | 6.92                |
| Scalable Bayesian Optimization [25] | -     | -          | 6.37                |
| NAS with strides [35]        | 20    | 2.5M       | 6.01                |
| Large-Scale Evolution [20]   | -     | 5.4M       | 5.40                |
| NAS with max pooling + more filters [35] | 39    | 37.4M      | 3.65                |
| RL with Network Transformation (ours) | 12    | 5.12M   | 6.74                |
|                              | 17    | 19.69M     | 5.70                |

The progress of the three stages architecture search is shown in Figure 3, where we can find that our agent gradually learns to pick high performance architectures at each stage. As our agent takes function preserving transformation to explore the architecture space, we can also find that the sampled architectures consistently perform better than the start point at each stage. Thus it is usually “safe” to explore the architecture space with function preserving transformations. Furthermore, we take the top network discovered during the third stage and further train the network with 300 epochs using the full training set. For this process, the initial learning rate is 0.025 and is divided by 5 at 50% and 75% of the total number of epochs. All other settings are the same with those introduced in Section 4.1. This network finally achieves 93.26% test accuracy (i.e. 6.74% test error rate), whose detailed architecture is provided in Table 3. We would like to emphasize that the required computational resources to achieve this result is much smaller than those required in [4, 35, 20]. Specifically, it takes less than 2 days on 5 GeForce GTX 1080 GPUs with totally 480 networks trained to achieve 6.74% test error rate starting from a poor network.

Explore Larger Architecture Space with High Performance Start Network: In the second experiment, we use the top network architecture discovered in the first experiment, which achieves 6.74% test error rate as the start point to search for high performance networks in a larger architecture space. This experiment takes around 1 day on 5 GPUs. We report the detailed architecture of top model discovered during this experiment in Table 3 and compare it to both human-designed models and automatically designed models on CIFAR-10.

The summarized results are reported in Table 2, where the first block of models, from Maxout to All-CNN, are top human-designed models on CIFAR-10 that use convolutional, pooling and fully-connected layers alone, which is the same as our agent; the second block of models contain complex layer types and design ideas such as skip connections, generalized pooling function and etc; the third block of models use automatically discovered architectures. From the result, we find that: (i) the top models discovered in our experiments outperform all human-designed models that
Figure 4: Comparison over random search on CIFAR-10.

use the same design scheme; (ii) Our top models show competitive performance compared to models with complex layer types and design ideas. Specifically, the 17-layer model which has a test error rate 5.70% performs better than ResNet (110) and Tree+Max-Avg; (iii) Direct comparison with other automatic architecture designing approaches can be difficult, since they use different computational resources and explore different search space. But we note that even using a much smaller amount of computational resources, the top models discovered by our agent are still very competitive and outperform models discovered by MetaQNN, Scalable Bayesian Optimization, and NAS with strides.

**Comparison against Random Search:** Beside policy gradient, one can also take network transformation actions to explore the architecture space by random search, which although seems to be simple but is usually hard to surpass [5]. Due to resource constraints, we conduct experiment versus random search in a smaller architecture space. Specifically, the number of filters is chosen from \([6, 8, \cdots, 32]\) and the number of units is chosen from \([16, 18, \cdots, 42]\). We use the start net 2 in Table 1 with validation accuracy 36.44% as the start point and report performance comparison between policy gradient and random search in Figure 4. The results show that, the policy gradient can efficiently focus on the right search direction, while random search cannot (left plot), and therefore find high performance architectures much more quickly than random search.

**Table 3: Top architectures discovered during our experiments on CIFAR-10.**

| Model Architecture                                                                 | Params | Test Error Rate (%) |
|-----------------------------------------------------------------------------------|--------|---------------------|
| C(192, 3, 1), C(128, 5, 1), C(96, 3, 1), P(2, 2), C(192, 3, 1), C(256, 5, 1), C(128, 5, 1), P(2, 2), C(192, 5, 1), C(128, 3, 1), C(96, 5, 1), P(2, 2), C(192, 3, 1), FC(192), SM(10) | 5.12M  | 6.74                |
| C(192, 3, 1), C(128, 5, 1), C(256, 3, 1), C(128, 5, 1), C(256, 3, 1), C(128, 3, 1), C(128, 3, 1), P(2, 2), C(256, 3, 1), C(256, 3, 1), C(256, 5, 1), P(2, 2), C(256, 5, 1), C(256, 3, 1), P(2, 2), C(256, 3, 1), C(256, 3, 1), C(256, 3, 1), P(2, 2), C(256, 3, 1), FC(1024), FC(256), SM(10) | 19.69M | 5.70                |

**5 Conclusion**

In this paper, we present an RL agent, learning to take network transformation actions to explore the architecture space for automatic architecture designing. By starting from an existing network and reusing its weights via the class of function preserving transformation operations, the agent is able to utilize knowledge stored in previously trained networks to search the architecture space efficiently and accurately. Experiments on designing CNN architectures for an image classification task with the benchmark dataset, CIFAR-10, demonstrated the flexibility, efficiency and effectiveness of our proposed methods compared to existing automatic architecture designing approaches which train each sampled network from scratch. For future work, we would like to further explore various network transformation actions and learn RL agents for different purposes, e.g. searching networks that not only have high accuracy but also keep a balance between the size and the performance. Also, since our agent can pick any existing network as the start point, we would also explore more complex schemes for selection the start point, such as a combination of RL based approaches and neuro-evolution approaches as discussed in Section 3.4.
References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. arXiv preprint arXiv:1603.04467, 2016.

[2] M. Andrychowicz, M. Denil, S. Gomez, M. W. Hoffman, D. Pfau, T. Schaul, and N. de Freitas. Learning to learn by gradient descent by gradient descent. In NIPS, 2016.

[3] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. arXiv preprint arXiv:1409.0473, 2014.

[4] B. Baker, O. Gupta, N. Naik, and R. Raskar. Designing neural network architectures using reinforcement learning. arXiv preprint arXiv:1611.02167, 2016.

[5] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. JMLR, 2012.

[6] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. Algorithms for hyper-parameter optimization. In NIPS, 2011.

[7] T. Chen, I. Goodfellow, and J. Shlens. Net2net: Accelerating learning via knowledge transfer. arXiv preprint arXiv:1511.05641, 2015.

[8] I. J. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio. Maxout networks. ICML, 2013.

[9] S. Han, J. Pool, J. Tran, and W. Dally. Learning both weights and connections for efficient neural network. In NIPS, 2015.

[10] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In CVPR, 2016.

[11] G. Huang, Z. Liu, K. Q. Weinberger, and L. van der Maaten. Densely connected convolutional networks. arXiv preprint arXiv:1608.06993, 2016.

[12] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. arXiv preprint arXiv:1502.03167, 2015.

[13] S. Kakade. A natural policy gradient. NIPS, 2002.

[14] A. Krizhevsky and G. Hinton. Learning multiple layers of features from tiny images. 2009.

[15] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In NIPS, 2012.

[16] C.-Y. Lee, P. W. Gallagher, and Z. Tu. Generalizing pooling functions in convolutional neural networks: Mixed, gated, and tree. In International conference on artificial intelligence and statistics, 2016.

[17] M. Lin, Q. Chen, and S. Yan. Network in network. arXiv preprint arXiv:1312.4400, 2013.

[18] G. F. Miller, P. M. Todd, and S. U. Hegde. Designing neural networks using genetic algorithms. In ICGA. Morgan Kaufmann Publishers Inc., 1989.

[19] R. Negrinho and G. Gordon. Deeparchitect: Automatically designing and training deep architectures. arXiv preprint arXiv:1704.08792, 2017.

[20] E. Real, S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, Q. Le, and A. Kurakin. Large-scale evolution of image classifiers. arXiv preprint arXiv:1703.01041, 2017.

[21] J. Schulman, S. Levine, P. Abbeel, M. I. Jordan, and P. Moritz. Trust region policy optimization. In ICML, 2015.

[22] M. Schuster and K. K. Paliwal. Bidirectional recurrent neural networks. IEEE Transactions on Signal Processing, 1997.

[23] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of go with deep neural networks and tree search. Nature, 2016.

[24] J. Snoek, H. Larochelle, and R. P. Adams. Practical bayesian optimization of machine learning algorithms. In NIPS, 2012.
[25] J. Snoek, O. Rippel, K. Swersky, R. Kiros, N. Satish, N. Sundaram, M. M. A. Patwary, M. Prabhat, and R. P. Adams. Scalable bayesian optimization using deep neural networks. In ICML, 2015.

[26] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller. Striving for simplicity: The all convolutional net. arXiv preprint arXiv:1412.6806, 2014.

[27] R. K. Srivastava, K. Greff, and J. Schmidhuber. Training very deep networks. In NIPS, 2015.

[28] K. O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. Evolutionary computation, 2002.

[29] I. Sutskever, J. Martens, G. Dahl, and G. Hinton. On the importance of initialization and momentum in deep learning. In International conference on machine learning, pages 1139–1147, 2013.

[30] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In NIPS, 2014.

[31] R. Vilalta and Y. Drissi. A perspective view and survey of meta-learning. Artificial Intelligence Review, 2002.

[32] J. X. Wang, Z. Kurth-Nelson, D. Tirumala, H. Soyer, J. Z. Leibo, R. Munos, C. Blundell, D. Kumaran, and M. Botvinick. Learning to reinforcement learn. arXiv preprint arXiv:1611.05763, 2016.

[33] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. Machine learning, 1992.

[34] S. Zagoruyko and N. Komodakis. Wide residual networks. arXiv preprint arXiv:1605.07146, 2016.

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