# Learning to Design Convolutional Neural Networks with Reinforcement Learning

**Aashima Arora**
Department of Computer Science
Columbia University
aa3917@columbia.edu

**Jason Krone**
Department of Computer Science
Columbia University
jpk2151@columbia.edu

## Abstract

Developing novel and effective convolutional neural network (CNN) architectures currently requires significant expertise and considerable trial and error. In this paper, we explore the use of reinforcement learning to automate the design of CNN architectures using validation accuracy as our reward function. Futhermore, we show that it is relatively simple to modify the reward function to account for computational requirements, such as the maximum number of parameters that can be used in the network. We intentionally constrain our architecture search space to decrease training time, determine the values of the reward function, and allow for a complete analysis of sample complexity and performance.

## 1 Introduction

State-of-the-art convolutional architectures have become increasingly complex over the recent years as researchers search for better performing models. This trend can be seen in the winning architectures of the ImageNet competition. The first deep neural network to win the competition, AlexNet, was a relatively simple 8-layer architecture. Soon thereafter, the networks became deeper with VGG and more complicated as skip connections were added and multiple convolutional operations were applied at each layer in the ResNet and Inception architectures respectively. In addition, the performance gains and designs of these networks continue to become more incremental and less intuitive over time. For these reasons, it is likely that future advances in neural network architectures will be driven by more complex building blocks and require a high volume of experiments to discover. Therefore, it makes sense to automate architecture design by extending the paradigm of end-to-end learning to include the optimization of network architectures.

There are a number of candidate methods for automating architecture discovery. Bayesian optimization, evolution strategies, and reinforcement learning are all capable of optimizing black box functions, such as validation accuracy. In this work, we choose the Reinforcement Learning framework because it has demonstrated success in designing state-of-the-art architectures for the ImageNet dataset. Please see our related works section for a more detailed discussion of this result and alternative methods.

To provide context we give a brief review of the reinforcement learning problem and describe how architecture search fits into this formulation. Reinforcement learning is the process of training an agent to maximize reward in an environment. More technically, the aim is to learn the optimal policy for selecting actions to take in a Markov Decision Process (MDP). A MDP is defined as $\langle S, A, P, r, \rho_0, \gamma, T \rangle$ where $S$ is a set of states, $A$ is a set of actions, $P : S \times A \times S \rightarrow \mathbb{R}_+$ is a transition function, $r : S \times A \rightarrow [R_{min}, R_{max}]$ is a reward function, $\gamma \in [0, 1]$ is a discount factor, and $T$ is a time horizon. The policy $\pi : S \times A \rightarrow \mathbb{R}_+$ is trained to maximize the expected discounted return $\eta(\pi_\theta) = \mathbb{E}_\tau \left[ \sum_{t=0}^{T} \gamma^t r(s_t, a_t) \right]$, where $\tau$ denotes a trajectory $\tau = (s_0, a_0, \dots)$ sampled according to $\pi_\theta$ with $s_0 \sim \rho_0(s_0), a_t \sim \pi_\theta(a_t \mid s_t)$, and $s_{t+1} \sim P(s_{t+1} \mid s_t, a_t)$. In the MDP corresponding to our implementation of architecture search, each action $a$ defines a neural

network architecture and $A$ is our architecture search space. The set of states is $S = \{s_{init}, s_{trained}\}$, where $s_{init}$ is an initial state in which no architecture has been trained and $s_{trained}$ is a state in which an architecture has been trained. The reward for any architecture and initial state $R : (a, s_{init}) \rightarrow R_+$ is a function of the validation accuracy of the trained neural network with architecture $a$. Given an initial state $s_{init}$ and an architecture $a$, the environment will always transition to the trained state i.e. $P(s_{trained} \mid s_{init}, a) = 1.0$. Our objective in this framework is to learn an optimal policy $\pi^*_\theta$ that selects the architecture(s) with the highest validation accuracy.

It is the case that many convolutional architectures such as VGG, ResNet, and Inception have a modular design. These networks repeat a pattern of operations, which we refer to as a convolutional "cell". For instance, the VGG cell contains three 3x3 convolutions followed by a maxpool. Taking inspiration from [2], we define our search space over cells rather than entire architectures. This approach has two advantages over predicting which operation to apply at every layer: 1. it reduces the search space and saves time 2. it reduces the likelihood of "overfitting" to the validation set. In the approach section, we describe exactly how a cell is defined and how a network is constructed given a cell architecture.

In addition to searching for the convolution cell that achieves maximum validation accuracy on MNIST, we show it is simple to modify the reward function such that architecture search finds the best architecture with fewer than $n$ parameters. This extension could be particularly useful when searching for architectures to be run on embedded devices, which often have specific memory constraints. Moreover, our approach is flexible and can accommodate using other metrics, such as maximum inference time, in place of parameter count. To the best of our knowledge this is a new extension of architecture search that has not been explored in other publications.

## 2 Related Work

The process of manually designing machine learning models is difficult because the search space of all possible models can be combinatorially large. Hence, the process of designing networks often takes a significant amount of time and experimentation by those with significant machine learning expertise. Recently, the idea of using certain types of neural networks (LSTM RNN's) to automatically generate neural network architectures consisting of convolutional cells has been attracting a lot of attention. Evolutionary algorithms and reinforcement learning have shown great promise among many algorithms that have been studied in this aspect. Recent works such as Zoph & Le(2016)[1] focus on learning architectures for large academic datasets like ImageNet and COCO datasets that vastly outperform state-of-the-art-models.

The design of our search-action space is inspired from LSTMs, and Neural Architecture Search Cell[1]. The modular structure of the convolutional cell is also related to previous methods on ImageNet such as VGG, Inception, and ResNet. We constrain the search to finding a good convolutional cell design, and simply stack it to handle inputs of arbitrary spatial dimensions and filter depth. The controller in NASnet is auto-regressive, which means it predicts hyperparameters one a time, conditioned on previous predictions. Eventually the controller learns to assign high probability to areas of architecture space that achieve better accuracy on the validation dataset, and low probability to areas of architecture space that score poorly. Thus, the controller learns directly from the reward signal.

Our work is most inspired by two consecutive works of Zoph & Le. Zoph & Le(2016)[1] used reinforcement learning to train a recurrent network that generates descriptions of neural networks to minimize validation error, and found convolutional and LTSM architectures that performed competitively in CIFAR-10 and Penn Treebank datasets, respectively. Using this approach, Zoph et al. (2017)[2] learn a convolutional cell on the CIFAR-10 dataset that can be transferred to the ImageNet dataset. The architecture obtained by stacking these convolutional cells is called NasNet. A key element of NasNet is to design the search space $S$ to generalize across problems of varying complexity and spatial scales. Applying NasNet directly on the ImageNet dataset would be very expensive and require months to complete an experiment. However, if the search space is properly constructed, architectural elements can transfer across datasets. By stacking together more of this cell, they achieve better top-1 accuracy than the best human-invented architectures with less computation.

Q-Learning has also been used to automate the network design process. Baker et al.(2017)[3] train a learning agent to sequentially choose CNN layers using Q-learning with $\epsilon$-greedy exploration

strategy and experience replay. They beat existing networks designed with only standard convolution, pooling and fully-connected layers. But reinforcement learning is more popular in comparison to other approaches. Cai et al. (2017)[4] train a reinforcement learning agent to grow the depth or layer width of a neural network, allowing previously learned weights to be reused. Li & Malik, 2016[5] apply the idea of using reinforcement learning to find update policies for another network. Another related work is the idea of learning to learn or meta-learning (Thrun & Pratt, 2012)[6], a general framework of using information learned in one task to improve a future task.

In addition, there has been some related work in hyperparameter optimization (Bergstra et al., 2011; Bergstra & Bengio, 2012; Snoek et al., 2012; 2015; Saxena & Verbeek, 2016). It is difficult to use them generate variable-length outputs that specify the network configuration and have been observed to work provided a good initial model (Bergstra & Bengio, 2012; Snoek et al., 2012; 2015). Then, there are Bayesian optimization methods that allow to search non fixed length architectures (Bergstra et al., 2013; Mendoza et al., 2016), but they are less general and less flexible.

## 3 Approach

Our work extends the Neural Architecture Search (NAS) framework put forward in [1]. In this framework, a policy network $\pi_\theta$ predicts a mini-batch of cell architectures $A_{mini}$, which in turn define a min-batch of child networks $C_{mini}$. The child networks are then trained until convergence on the MNIST dataset and their validation accuracies are used as the reward to update $\pi_\theta$. Pseudo code for this training loop is given below.

---

**Pseudo Code:** NAS Training Loop

---

**for** $k = 0, 1, 2, \ldots$ **do**

1. Sample a mini-batch of architectures $A_{mini}$ from $\pi_\theta$
2. Train child networks $C_{mini}$ on MNIST and report reward $R_{mini}$
3. Form a training dataset $D = (A_{mini}, R_{mini})$
4. Compute policy update $\theta_{k+1} = \text{argmax}_\theta L^{CLIP}_{\theta_k}(\theta)$ by taking $K$ steps of minibatch SGD (with Adam) using proximal policy optimization (PPO)

---

### 3.1 Search Space

We experiment with two search spaces $A_{small}$ and $A_{large}$. $A_{small}$ is a relatively small search space where a convolutional cell is defined by two operations $\langle op_1, op_2 \rangle$ applied consecutively, and the list of operations is as follows:

- 1x3 then 3x1 conv
- 3x3 conv
- 3x3 depthwise-seperable conv
- 5x5 conv
- 7x7 conv
- 2x2 max pooling
- 2x2 average pooling
- identity

The advantage of using $A_{small}$ is that there are only $8 \times 8 = 64$ possible child networks. As a result, we can pre-compute the exact values of the reward function (child network validation accuracies) prior to training $\pi_\theta$. Then during the training procedure, we can use these pre-computed rewards to train $\pi_\theta$ more quickly. As we show in the experiments section, even on this small search space, NAS takes hundreds of samples to converge. With our limited compute (1 GPU), it was infeasible for us to train hundreds of child network architectures. Pre-computing the reward function allows us to both run experiments in a shorter amount of time and obtain accurate results. We provide more information on the pre-computed reward function in the experiments section.

While $A_{small}$ is useful for analyzing the sample complexity and performance of NAS, it does not demonstrate the ability NAS to generate complex architectures and learn without exploring the entire search space. For this reason, we also experiment with a much larger search space $A_{large}$, which is a subset of the search spaced used in [2]. In $A_{large}$, a convolutional cell is defined as $B$ convolutional

blocks, where a block takes the form given in figure 1. Each convolutional block requires selecting four parameters $\langle h_a, h_b, op_1, op_2 \rangle$, where

- $h_a$ is the hidden layer input to $op_1$
- $h_b$ is the hidden layer input to $op_2$
- $op_1$ is an operation from the list given above
- $op_2$ is an operation from the list given above

As illustrated in figure 1, the representations produced by $op_1(h_a)$ and $op_2(h_b)$ in block $B_i$ are combined using filter-wise concatenation to form a new hidden layer $h_i$. The hidden layer $h_i$ can be selected as the value for the parameters $h_a$, $h_b$ in the following blocks. This structure allows for cells in this search space to form skip connections between layers within a cell.

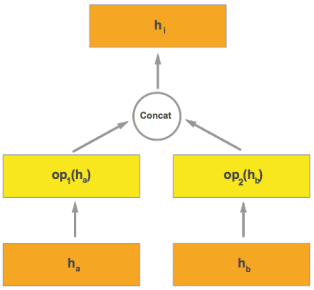


Figure 1: $A_{large}$ cell-block structure

### 3.2 Controller Network

We use a LSTM with one hundred hidden units to approximate our policy $\pi_\theta$. The outputs of the LSTM are softmax probability distributions over the operations, and or hidden states, that define the convolutional cell in our search space. The softmax distribution produced at time-step $t$ is fed as the input to the LSTM at time-step $t + 1$ to condition future parameters on the parameters selected so far. To sample a child network, we sample the values of the parameters that define a cell according to their respective softmax probabilities. The controller is trained to maximize the validation accuracy of the sampled child networks using Proximal Policy Optimization (PPO). Becuase cells architectures in $A_{small}$ and $A_{large}$ differ, our controller architectures for $A_{small}$ and $A_{large}$ differ as well. The controllers for $A_{small}$ and $A_{large}$ are detailed below:

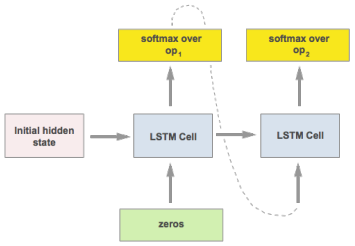


Figure 2: Controller architecture for $A_{small}$

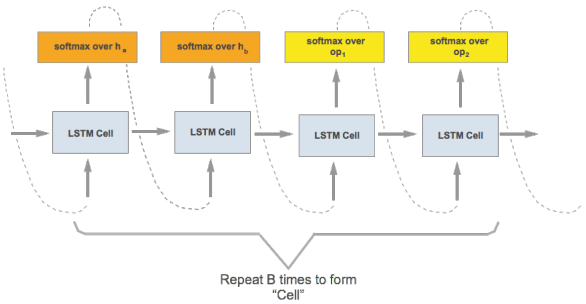


Figure 3: Controller architecture for $A_{large}$

### 3.3 Child Network

Given a cell architecture $a$, the corresponding child network $C$ is defined by the sequence: $a$-maxpool-$a$-maxpool-fc-softmax where maxpool is a 2x2 max pooling layer and fc is a 4096 hidden unit fully

connected layer with dropout. We follow the convention set by VGG and double the number of filters used in the convolutional cell whenever the spatial dimension is reduced by a factor of two. Specifically, we use $64$ filters in the first convolutional cell and $128$ filters in the second convolutional cell. This child network structure is illustrated in figure 4 below. Child networks are trained on the MNIST dataset and evaluated on a held out validation set. Although it is beyond the scope of this work, it is likely that stacking a greater number of convolutional cells could increase performance and allow the cell architecture to "scale" to larger datasets such as ImageNet.

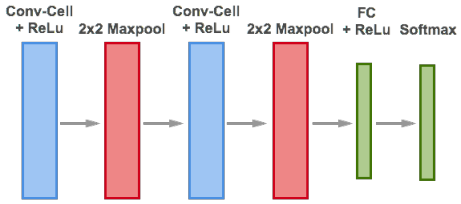


Figure 4: Child network architecture

# 4 Experiments

In this section, we discuss the three NAS experiments we ran on the MNIST dataset. There are two factors that differentiate these experiments. Namely, the search space used in the experiment ($A_{small}$ or $A_{large}$) and the reward function used in the experiment ($r_{val}$ or $r_{param}$). We explicitly define both of these reward functions in the subsection below. Our experiments are as follows:

1. NAS applied to $A_{small}$ using the $r_{val}$ reward function
2. NAS applied to $A_{small}$ using the $r_{param}$ reward function
3. NAS applied to $A_{large}$ using the $r_{val}$ reward function

To analyze the performance of NAS, we plot of the mean, min, max, and standard deviation of the reward over training episodes. Here an episode corresponds to one iteration of the NAS training loop wherein we sample a mini-batch of $20$ child network architectures. For both of our experiments on the $A_{small}$ search space, we pre-compute the values of our reward function prior to training the NAS controller to dramatically speed up training time.

## 4.1 Reward functions

The validation accuracy reward function, $r_{val}$, is defined as $r_{val} = Val(a)$, where $a \in A_{small}$ and $Val(a)$ denotes the validation accuracy of the child network with cell architecture $a$ on the MNIST validation set.

The maximum parameter reward function, $r_{param}$, is meant to constrain the search space such that NAS finds the best architecture with fewer than $n$ parameters. Let $P_a$ be the total number of trainable parameters in the child network with cell architecture $a \in A_{small}$. Then $r_{param}$ is defined as follows:

$$r_{param} = \begin{cases} -100.0 & P_a \geq n \\ r_{val} & P_a < n \end{cases}$$

## 4.2 Dataset

All of the child networks in our experiments are trained and evaluated on the MNIST dataset. We choose the MNIST dataset because it is both a standard computer vision benchmark and a relatively small dataset, which allows us to minimize training time. The MNIST dataset contains $70,000$ black and white images of hand written digits. We use $54,000$ images for the training set, $6,000$ images for the validation set, and $10,000$ images for the test set. We train each child network on the training split for $10$ epochs using a negative log likelihood loss and the ADAM optimizer with a learning rate

of $0.0001$. We found empirically that after $10$ epochs the validation accuracy of the child network converged.

### 4.3 Results for $A_{small}$ using $r_{val}$

The reward plot for $A_{small}$ using $r_{val}$ is given in the figure below. There are two takeaways from this figure. Firstly, we see that NAS is successful in finding the best cell architecture in the search space; this is illustrated in the graph by the min and mean validation accuracy (reward) converging to the maximum validation accuracy. Secondly, the plot demonstrates that NAS has poor sample complexity (requires many samples to converge). As shown in the graph, convergence occurs after approximately 200 episodes, which would require training of $4,000$ child networks if the values of the reward function were not pre-computed. The two cells with the highest validation accuracy found by NAS on this search space are $\langle$5x5 conv, 1x3 then 3x1 conv$\rangle$ and $\langle$3x3 seperable conv, 7x7 conv$\rangle$. We give the validation and test accuracies for the child networks corresponding to these cells in table 1.

Table 1: Performance of best cells for $A_{small}$ and $r_{val}$

| | **Validation Accuracy** | **Test Accuracy** |
|---|---|---|
| $\langle$5x5 conv, 1x3 then 3x1 conv$\rangle$ | 98.85 | 98.88 |
| $\langle$3x3 seperable conv, 7x7 conv$\rangle$ | 98.85 | 98.2 |

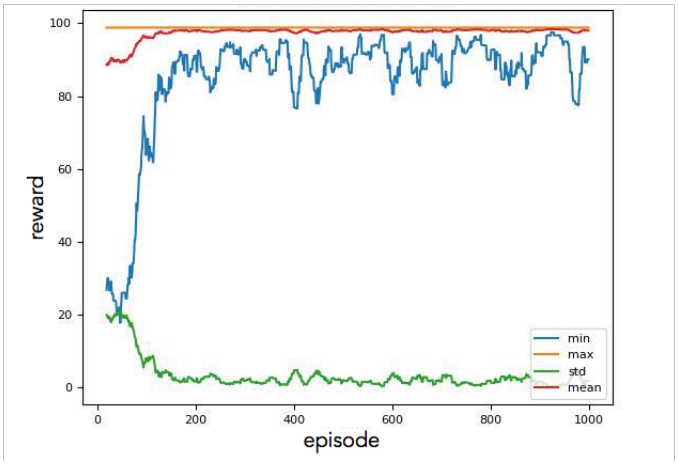


Figure 5: Small search space validation reward plot

### 4.4 Results for $A_{small}$ using $r_{param}$

As discussed above, the maximum parameter reward function associates all architectures that have more than $n$ parameters with a negative reward. This negative reward discourages the controller from selecting architectures with more than $n$ parameters. We choose $n = 150,000$ as our parameter limit because it partitions the search space such that only cells with at most one convolutional operation from the set $\{$3x3 seperable conv, 3x3 conv$\}$ are associated with positive reward.

The reward plot for this experiment is given in figure 6. It is clear this graph that finding the best architecture with fewer than $150,000$ parameters is more difficult than simply finding the best architecture with no parameter limit. While the reward plot for $r_{val}$ converges after 200 episodes, the reward plot for $r_{param}$ does not fully converge within $1000$ episodes. This is evidenced by the fact that, even after $1000$ episodes, the standard deviation of $r_{param}$ is high and the minimum reward appears to still be increasing. That said, the controller does quickly learn **not** to predict architectures with negative reward as you can see by the rapid increase in reward from $-25$ to

50 within the first 100 episodes. In this experiment, NAS was unable to find the best architecture with fewer than $150,000$ paramters, $\langle$3x3 conv, max$\rangle$. However, it was able to find the second best architecture, $\langle$3x3 conv, avg$\rangle$. We give the parameter counts and validation accuracies for both of these architectures in the following table.

Table 2: Performance of best and second best cells for $A_{small}$ and $r_{param}$

| | **Parameter Count** | **Validation Accuracy** |
|---|---|---|
| $\langle$3x3 conv, max$\rangle$ | $117,972$ | 98.61 |
| $\langle$3x3 conv, avg$\rangle$ | $117,972$ | 98.01 |

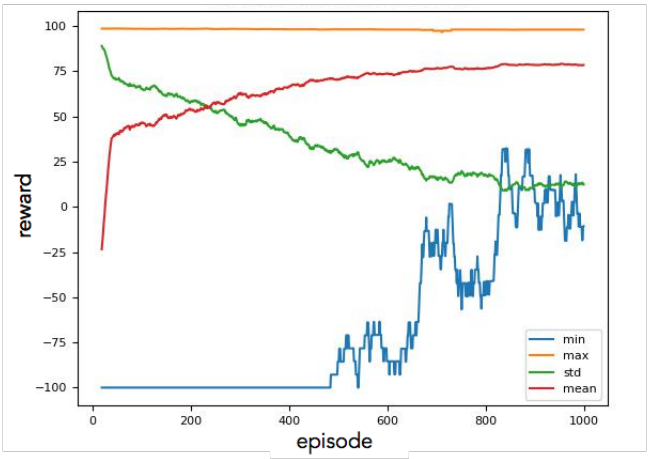


Figure 6: Small search space maximum parameter reward plot

### 4.5 Results for $A_{large}$ and $r_{val}$

While the experiments on the search space $A_{small}$ allow us to analyze the performance of NAS in detail, they do not illustrate how NAS can be used to generate novel and complex cell architectures. For this reason, we also experiment with $A_{large}$. Unfortunately, we did not have the compute necessary to train NAS on enough samples for the reward plot to show any signs of improvement on $A_{large}$. For this reason we do not give a reward plot for this experiment. Instead, we report the performance of the best cell NAS has found to-date and provide a digram of this cell architecture in figure 7.

Table 3: Performance of best cell for $A_{large}$ and $r_{val}$

| | **Validation Accuracy** | **Test Accuracy** |
|---|---|---|
| Complex convolutional cell | 41.41 | 42.97 |

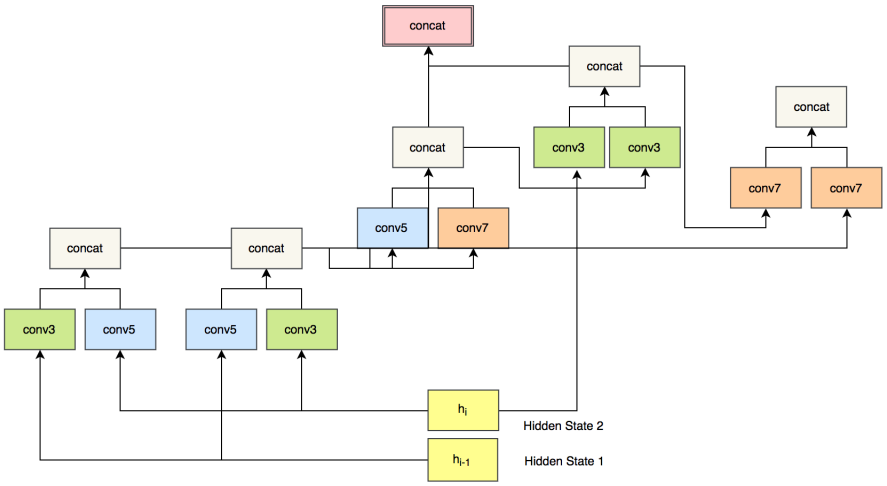


Figure 7: Complex convolutional cell architecture

## 5 Conclusion

In this work, we demonstrate that it is possible to use NAS to find high performing convolutional cells on the MNIST dataset. In addition, we show that it is relatively simple to modify the reward function to account for computational requirements. Although, NAS succeeds at finding architectures that perform well on MNIST, the number of samples necessary to find these architectures is very large. We see the poor sample complexity of NAS as the main limitation associated with this method. Even on the relatively small MNIST dataset, NAS required thousands of samples to converge, and we believe that tens of thousands of samples would be required for convergence on larger dataset such as CIFAR and ImageNet. Therefore, reducing the sample complexity of NAS by developing more data efficient reinforcement learning algorithms is an important area for future work. In addition, we think it would be interesting to apply the NAS framework to design other neural network components such as attention or memory mechanisms.

## References


[1] Zoph and Le. “Neural Network architecture search with reinforcement learning”. In: *ICLR*. 2016.

[2] Zoph et Al. “Learning Transferable Architectures for Scalable Image Recognition”. In: *arXiv:1707.07012*. 2017.

[3] Baker et Al. “Designing neural network architectures using reinforcement learning”. In: *ICLR*. 2017.

[4] Cai et Al. “Efficient Architecture Search by Network Transformation”. In: *arXiv:1707.04873*. 2017.

[5] Li and Malik. “Learning to Optimize”. In: *arXiv:1606.01885*. 2016.

[6] Sebastian Thrun and Lorien Pratt. *Learning to Learn*. 1998.

[7] Liu Et Al. “Progressive Neural Architecture Search”. In: *arXiv:1712.00559*. 2017.