NEURAL ARCHITECTURE SEARCH WITH REINFORCEMENT LEARNING

Barret Zoph,* **Quoc V. Le**
Google Brain
{barretzoph,qvl}@google.com

ABSTRACT

Neural networks are powerful and flexible models that work well for many difficult learning tasks in image, speech and natural language understanding. Despite their success, neural networks are still hard to design. In this paper, we use a recurrent network to generate the model descriptions of neural networks and train this RNN with reinforcement learning to maximize the expected accuracy of the generated architectures on a validation set. On the CIFAR-10 dataset, our method, starting from scratch, can design a novel network architecture that rivals the best human-invented architecture in terms of test set accuracy. Our CIFAR-10 model achieves a test error rate of 3.65, which is 0.09 percent better and 1.05x faster than the previous state-of-the-art model that used a similar architectural scheme. On the Penn Treebank dataset, our model can compose a novel recurrent cell that outperforms the widely-used LSTM cell, and other state-of-the-art baselines. Our cell achieves a test set perplexity of 62.4 on the Penn Treebank, which is 3.6 perplexity better than the previous state-of-the-art model. The cell can also be transferred to the character language modeling task on PTB and achieves a state-of-the-art perplexity of 1.214.

1 INTRODUCTION

The last few years have seen much success of deep neural networks in many challenging applications, such as speech recognition (**?**), image recognition (**??**) and machine translation (**???**). Along with this success is a paradigm shift from feature designing to architecture designing, i.e., from SIFT (**?**), and HOG (**?**), to AlexNet (**?**), VGGNet (**?**), GoogleNet (**?**), and ResNet (**?**). Although it has become easier, designing architectures still requires a lot of expert knowledge and takes ample time.

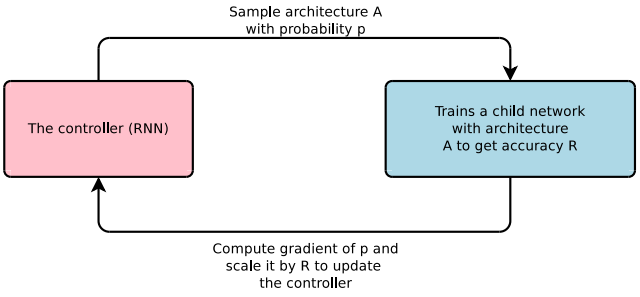

Figure 1: An overview of Neural Architecture Search.

This paper presents Neural Architecture Search, a gradient-based method for finding good architectures (see Figure 1) . Our work is based on the observation that the structure and connectivity of a neural network can be typically specified by a variable-length string. It is therefore possible to use

*Work done as a member of the Google Brain Residency program (g.co/brainresidency.)

a recurrent network – the controller – to generate such string. Training the network specified by the string – the "child network" – on the real data will result in an accuracy on a validation set. Using this accuracy as the reward signal, we can compute the policy gradient to update the controller. As a result, in the next iteration, the controller will give higher probabilities to architectures that receive high accuracies. In other words, the controller will learn to improve its search over time.

Our experiments show that Neural Architecture Search can design good models from scratch, an achievement considered not possible with other methods. On image recognition with CIFAR-10, Neural Architecture Search can find a novel ConvNet model that is better than most human-invented architectures. Our CIFAR-10 model achieves a 3.65 test set error, while being 1.05x faster than the current best model. On language modeling with Penn Treebank, Neural Architecture Search can design a novel recurrent cell that is also better than previous RNN and LSTM architectures. The cell that our model found achieves a test set perplexity of 62.4 on the Penn Treebank dataset, which is 3.6 perplexity better than the previous state-of-the-art.

2 Related Work

Hyperparameter optimization is an important research topic in machine learning, and is widely used in practice (**?????**). Despite their success, these methods are still limited in that they only search models from a fixed-length space. In other words, it is difficult to ask them to generate a variable-length configuration that specifies the structure and connectivity of a network. In practice, these methods often work better if they are supplied with a good initial model (**???**). There are Bayesian optimization methods that allow to search non fixed length architectures (**??**), but they are less general and less flexible than the method proposed in this paper.

Modern neuro-evolution algorithms, e.g., **???**, on the other hand, are much more flexible for composing novel models, yet they are usually less practical at a large scale. Their limitations lie in the fact that they are search-based methods, thus they are slow or require many heuristics to work well.

Neural Architecture Search has some parallels to program synthesis and inductive programming, the idea of searching a program from examples (**??**). In machine learning, probabilistic program induction has been used successfully in many settings, such as learning to solve simple Q&A (**???**), sort a list of numbers (**?**), and learning with very few examples (**?**).

The controller in Neural Architecture Search is auto-regressive, which means it predicts hyperparameters one a time, conditioned on previous predictions. This idea is borrowed from the decoder in end-to-end sequence to sequence learning (**?**). Unlike sequence to sequence learning, our method optimizes a non-differentiable metric, which is the accuracy of the child network. It is therefore similar to the work on BLEU optimization in Neural Machine Translation (**??**). Unlike these approaches, our method learns directly from the reward signal without any supervised bootstrapping.

Also related to our work is the idea of learning to learn or meta-learning (**?**), a general framework of using information learned in one task to improve a future task. More closely related is the idea of using a neural network to learn the gradient descent updates for another network (**?**) and the idea of using reinforcement learning to find update policies for another network (**?**).

3 Methods

In the following section, we will first describe a simple method of using a recurrent network to generate convolutional architectures. We will show how the recurrent network can be trained with a policy gradient method to maximize the expected accuracy of the sampled architectures. We will present several improvements of our core approach such as forming skip connections to increase model complexity and using a parameter server approach to speed up training. In the last part of the section, we will focus on generating recurrent architectures, which is another key contribution of our paper.

3.1 ARCHITECTURE DESCRIPTIONS

As stated earlier, our key observation is that many common neural architectures can be summarized in variable-length strings, which we also call "architecture descriptions." Below is an example architecture description of a simple 2-layer convolutional network:

```
layer {
    name = layer_1
    type = conv
    connect_to = input
    filter_width = 2
    filter_height = 3
    stride_width = 4
    stride_height = 5
    num_filters = 6
}
layer {
    name = layer_2
    type = conv
    connect_to = layer_1
    filter_width = 7
    filter_ height = 8
    stride_width = 9
    stride_height = 10
    num_filters = 11
}
layer {
    name = classifier
    connect_to = layer_2
    num_class = 10
}
```

By focusing on key hyperparameters, the above description can be compressed into:

```
2 3 4 5 6 7 8 9 10 11
```

Notice that information regarding layer types and connectivity between layers are skipped entirely because we are only interested in feedforward convolutional networks without skip connections. The last layer is also skipped because we do not want to vary it. Although the new description does not have hyperparameter names such as "filter_width", "filter_height", they can be inferred given the order of the tokens. In the following section, we will show how we use a recurrent network to generate this type of architecture descriptions.

3.2 GENERATE MODEL DESCRIPTIONS WITH A CONTROLLER RECURRENT NEURAL NETWORK

In Neural Architecture Search, we use a controller to generate architectural hyperparameters of neural networks. To be flexible, the controller is implemented as a recurrent neural network. Let's suppose we would like to predict feedforward neural networks with only convolutional layers, we can use the controller to generate their hyperparameters as a sequence of tokens:

In our experiments, the process of generating an architecture stops if the number of layers exceeds a certain value. This value follows a schedule where we increase it as training progresses. Once the controller RNN finishes generating an architecture, a neural network with this architecture is built and trained. At convergence, the accuracy of the network on a held-out validation set is recorded. The parameters of the controller RNN, θ_c, are then optimized in order to maximize the expected validation accuracy of the proposed architectures. In the next section, we will describe a policy gradient method which we use to update parameters θ_c so that the controller RNN generates better architectures over time.

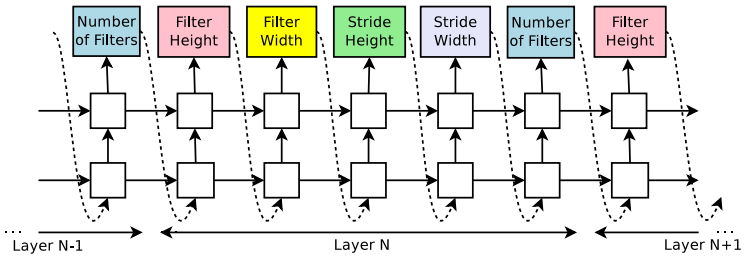

Figure 2: How our controller recurrent neural network samples a simple convolutional network. It predicts filter height, filter width, stride height, stride width, and number of filters for one layer and repeats. Every prediction is carried out by a softmax classifier and then fed into the next time step as input.

3.3 TRAINING WITH REINFORCE

The list of tokens that the controller predicts can be viewed as a list of actions $a_{1:T}$ to design an architecture for a child network. At convergence, this child network will achieve an accuracy R on a held-out dataset. We can use this accuracy R as the reward signal and use reinforcement learning to train the controller. More concretely, to find the optimal architecture, we ask our controller to maximize its expected reward, represented by $J(\theta_c)$:

$$J(\theta_c) = E_{P(a_{1:T};\theta_c)}[R]$$

Since the reward signal R is non-differentiable, we need to use a policy gradient method to iteratively update θ_c. In this work, we use the REINFORCE rule from **?**:

$$\bigtriangledown_{\theta_c} J(\theta_c) = \sum_{t=1}^{T} E_{P(a_{1:T};\theta_c)}\big[\bigtriangledown_{\theta_c} \log P(a_t|a_{(t-1):1};\theta_c)R\big]$$

An empirical approximation of the above quantity is:

$$\frac{1}{m}\sum_{k=1}^{m}\sum_{t=1}^{T} \bigtriangledown_{\theta_c} \log P(a_t|a_{(t-1):1};\theta_c)R_k$$

Where m is the number of different architectures that the controller samples in one batch and T is the number of hyperparameters our controller has to predict to design a neural network architecture. The validation accuracy that the k-th neural network architecture achieves after being trained on a training dataset is R_k.

The above update is an unbiased estimate for our gradient, but has a very high variance. In order to reduce the variance of this estimate we employ a baseline function:

$$\frac{1}{m}\sum_{k=1}^{m}\sum_{t=1}^{T} \bigtriangledown_{\theta_c} \log P(a_t|a_{(t-1):1};\theta_c)(R_k - b)$$

As long as the baseline function b does not depend on the on the current action, then this is still an unbiased gradient estimate. In this work, our baseline b is an exponential moving average of the previous architecture accuracies.

Accelerate Training with Parallelism and Asynchronous Updates: In Neural Architecture Search, each gradient update to the controller parameters θ_c corresponds to training one child network to convergence. As training a child network can take hours, we use distributed training and asynchronous parameter updates in order to speed up the learning process of the controller (**?**). We

use a parameter-server scheme where we have a parameter server of S shards, that store the shared parameters for K controller replicas. Each controller replica samples m different child architectures that are trained in parallel. The controller then collects gradients according to the results of that minibatch of m architectures at convergence and sends them to the parameter server in order to update the weights across all controller replicas. In our implementation, convergence of each child network is reached when its training exceeds a certain number of epochs. This scheme of parallelism is summarized in Figure 3.

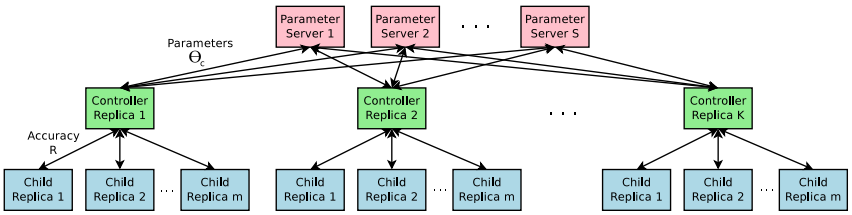

Figure 3: Distributed training for Neural Architecture Search. We use a set of S parameter servers to store and send parameters to K controller replicas. Each controller replica then samples m architectures and run the multiple child models in parallel. The accuracy of each child model is recorded to compute the gradients with respect to θ_c, which are then sent back to the parameter servers.

3.4 Increase Architecture Complexity with Skip Connections and Other Layer Types

In Section 3.2, the search space does not have skip connections, or branching layers used in modern architectures such as GoogleNet (**?**), and Residual Net (**?**). In this section we introduce a method that allows our controller to propose skip connections or branching layers, thereby widening the search space.

To enable the controller to predict such connections, we use a set-selection type attention (**?**) which was built upon the attention mechanism (**??**). At layer N, we add an anchor point which has $N-1$ content-based sigmoids to indicate the previous layers that need to be connected. Each sigmoid is a function of the current hiddenstate of the controller and the previous hiddenstates of the previous $N-1$ anchor points:

$$\text{P(Layer j is an input to layer i)} = \text{sigmoid}(v^{\text{T}}\tanh(W_{prev} * h_j + W_{curr} * h_i)),$$

where h_j represents the hiddenstate of the controller at anchor point for the j-th layer, where j ranges from 0 to $N-1$. We then sample from these sigmoids to decide what previous layers to be used as inputs to the current layer. The matrices W_{prev}, W_{curr} and v are trainable parameters. As these connections are also defined by probability distributions, the REINFORCE method still applies without any significant modifications. Figure 4 shows how the controller uses skip connections to decide what layers it wants as inputs to the current layer.

In our framework, if one layer has many input layers then all input layers are concatenated in the depth dimension. Skip connections can cause "compilation failures" where one layer is not compatible with another layer, or one layer may not have any input or output. To circumvent these issues, we employ three simple techniques. First, if a layer is not connected to any input layer then the image is used as the input layer. Second, at the final layer we take all layer outputs that have not been connected and concatenate them before sending this final hiddenstate to the classifier. Lastly, if input layers to be concatenated have different sizes, we pad the small layers with zeros so that the concatenated layers have the same sizes.

Finally, in Section 3.2, we do not predict the learning rate and we also assume that the architectures consist of only convolutional layers, which is also quite restrictive. It is possible to add the learning rate as one of the predictions. Additionally, it is also possible to predict pooling, local contrast normalization (**??**), and batchnorm (**?**) in the architectures. To be able to add more types of layers, we need to add an additional step in the controller RNN to predict the layer type, then other hyperparameters associated with it.

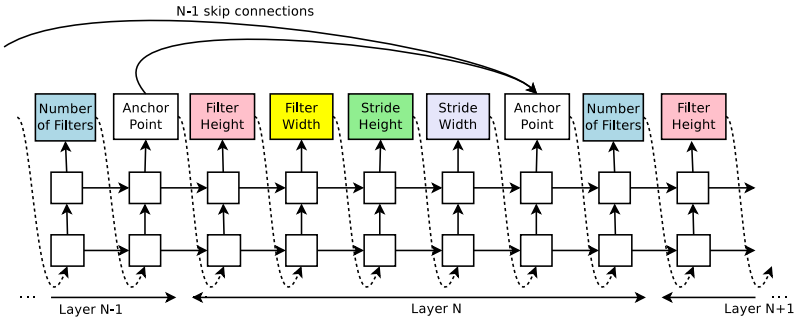

Figure 4: The controller uses anchor points, and set-selection attention to form skip connections.

3.5 Generate Recurrent Cell Architectures

In this section, we will modify the above method to generate recurrent cells. At every time step t, the controller needs to find a functional form for h_t that takes x_t and h_{t-1} as inputs. The simplest way is to have $h_t = \tanh(W_1 * x_t + W_2 * h_{t-1})$, which is the formulation of a basic recurrent cell. A more complicated formulation is the widely-used LSTM recurrent cell (**?**).

The computations for basic RNN and LSTM cells can be generalized as a tree of steps that take x_t and h_{t-1} as inputs and produce h_t as final output. The controller RNN needs to label each node in the tree with a combination method (addition, elementwise multiplication, etc.) and an activation function ($\tanh$, $\mathrm{sigmoid}$, etc.) to merge two inputs and produce one output. Two outputs are then fed as inputs to the next node in the tree. To allow the controller RNN to select these methods and functions, we index the nodes in the tree in an order so that the controller RNN can visit each node one by one and label the needed hyperparameters.

Inspired by the construction of the LSTM cell (**?**), we also need cell variables c_{t-1} and c_t to represent the memory states. To incorporate these variables, we need the controller RNN to predict what nodes in the tree to connect these two variables to. These predictions can be done in the last two blocks of the controller RNN.

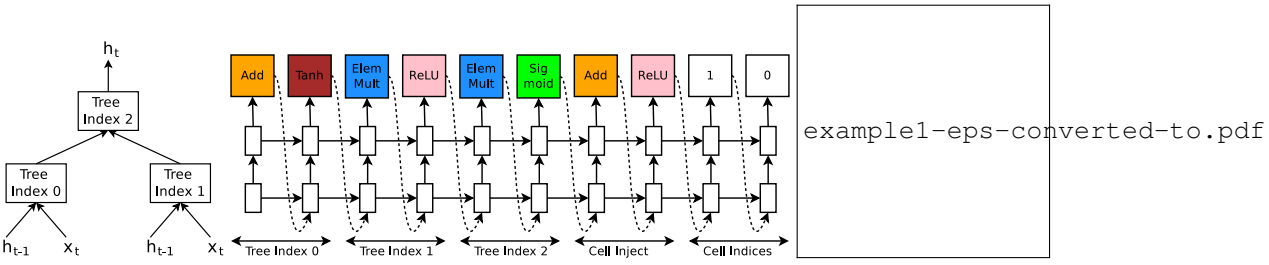

Figure 5: An example of a recurrent cell constructed from a tree that has two leaf nodes (base 2) and one internal node. Left: the tree that defines the computation steps to be predicted by controller. Center: an example set of predictions made by the controller for each computation step in the tree. Right: the computation graph of the recurrent cell constructed from example predictions of the controller.

To make this process more clear, we show an example in Figure 5, for a tree structure that has two leaf nodes and one internal node. The leaf nodes are indexed by 0 and 1, and the internal node is indexed by 2. The controller RNN needs to first predict 3 blocks, each block specifying a combination method and an activation function for each tree index. After that it needs to predict the last 2 blocks that specify how to connect c_t and c_{t-1} to temporary variables inside the tree. Specifically, according to the predictions of the controller RNN in this example, the following computation steps will occur:

- The controller predicts Add and $Tanh$ for tree index 0, this means we need to compute $a_0 = \tanh(W_1 * x_t + W_2 * h_{t-1})$.
- The controller predicts $ElemMult$ and $ReLU$ for tree index 1, this means we need to compute $a_1 = \mathrm{ReLU}\big((W_3 * x_t) \odot (W_4 * h_{t-1})\big)$.
- The controller predicts 0 for the second element of the "Cell Index", Add and $ReLU$ for elements in "Cell Inject", which means we need to compute $a_0^{new} = \mathrm{ReLU}(a_0 + c_{t-1})$. Notice that we don't have any learnable parameters for the internal nodes of the tree.
- The controller predicts $ElemMult$ and $Sigmoid$ for tree index 2, this means we need to compute $a_2 = \mathrm{sigmoid}(a_0^{new} \odot a_1)$. Since the maximum index in the tree is 2, h_t is set to a_2.
- The controller RNN predicts 1 for the first element of the "Cell Index", this means that we should set c_t to the output of the tree at index 1 before the activation, i.e., $c_t = (W_3 * x_t) \odot (W_4 * h_{t-1})$.

In the above example, the tree has two leaf nodes, thus it is called a "base 2" architecture. In our experiments, we use a base number of 8 to make sure that the cell is expressive.

4 Experiments and Results

We apply our method to an image classification task with CIFAR-10 and a language modeling task with Penn Treebank, two of the most benchmarked datasets in deep learning. On CIFAR-10, our goal is to find a good convolutional architecture whereas on Penn Treebank our goal is to find a good recurrent cell. On each dataset, we have a separate held-out validation dataset to compute the reward signal. The reported performance on the test set is computed only once for the network that achieves the best result on the held-out validation dataset. More details about our experimental procedures and results are as follows.

4.1 Learning Convolutional Architectures for CIFAR-10

Dataset: In these experiments we use the CIFAR-10 dataset with data preprocessing and augmentation procedures that are in line with other previous results. We first preprocess the data by whitening all the images. Additionally, we upsample each image then choose a random 32x32 crop of this upsampled image. Finally, we use random horizontal flips on this 32x32 cropped image.

Search space: Our search space consists of convolutional architectures, with rectified linear units as non-linearities (**?**), batch normalization (**?**) and skip connections between layers (Section 3.4). For every convolutional layer, the controller RNN has to select a filter height in [1, 3, 5, 7], a filter width in [1, 3, 5, 7], and a number of filters in [24, 36, 48, 64]. For strides, we perform two sets of experiments, one where we fix the strides to be 1, and one where we allow the controller to predict the strides in [1, 2, 3].

Training details: The controller RNN is a two-layer LSTM with 35 hidden units on each layer. It is trained with the ADAM optimizer (**?**) with a learning rate of 0.0006. The weights of the controller are initialized uniformly between -0.08 and 0.08. For the distributed training, we set the number of parameter server shards S to 20, the number of controller replicas K to 100 and the number of child replicas m to 8, which means there are 800 networks being trained on 800 GPUs concurrently at any time.

Once the controller RNN samples an architecture, a child model is constructed and trained for 50 epochs. The reward used for updating the controller is the maximum validation accuracy of the last 5 epochs cubed. The validation set has 5,000 examples randomly sampled from the training set, the remaining 45,000 examples are used for training. The settings for training the CIFAR-10 child models are the same with those used in **?**. We use the Momentum Optimizer with a learning rate of 0.1, weight decay of 1e-4, momentum of 0.9 and used Nesterov Momentum (**?**).

During the training of the controller, we use a schedule of increasing number of layers in the child networks as training progresses. On CIFAR-10, we ask the controller to increase the depth by 2 for the child models every 1,600 samples, starting at 6 layers.

Results: After the controller trains 12,800 architectures, we find the architecture that achieves the best validation accuracy. We then run a small grid search over learning rate, weight decay, batchnorm epsilon and what epoch to decay the learning rate. The best model from this grid search is then run until convergence and we then compute the test accuracy of such model and summarize the results in Table 1. As can be seen from the table, Neural Architecture Search can design several promising architectures that perform as well as some of the best models on this dataset.

Model	Depth	Parameters	Error rate (%)
Network in Network (**?**)	-	-	8.81
All-CNN (**?**)	-	-	7.25
Deeply Supervised Net (**?**)	-	-	7.97
Highway Network (**?**)	-	-	7.72
Scalable Bayesian Optimization (**?**)	-	-	6.37
FractalNet (**?**)	21	38.6M	5.22
with Dropout/Drop-path	21	38.6M	4.60
ResNet (**?**)	110	1.7M	6.61
ResNet (reported by **?**)	110	1.7M	6.41
ResNet with Stochastic Depth (**?**)	110	1.7M	5.23
	1202	10.2M	4.91
Wide ResNet (**?**)	16	11.0M	4.81
	28	36.5M	4.17
ResNet (pre-activation) (**?**)	164	1.7M	5.46
	1001	10.2M	4.62
DenseNet ($L = 40, k = 12$) **?**	40	1.0M	5.24
DenseNet($L = 100, k = 12$) **?**	100	7.0M	4.10
DenseNet ($L = 100, k = 24$) **?**	100	27.2M	3.74
DenseNet-BC ($L = 100, k = 40$) **?**	190	25.6M	3.46
Neural Architecture Search v1 no stride or pooling	15	4.2M	5.50
Neural Architecture Search v2 predicting strides	20	2.5M	6.01
Neural Architecture Search v3 max pooling	39	7.1M	4.47
Neural Architecture Search v3 max pooling + more filters	39	37.4M	3.65

Table 1: Performance of Neural Architecture Search and other state-of-the-art models on CIFAR-10.

First, if we ask the controller to not predict stride or pooling, it can design a 15-layer architecture that achieves 5.50% error rate on the test set. This architecture has a good balance between accuracy and depth. In fact, it is the shallowest and perhaps the most inexpensive architecture among the top performing networks in this table. This architecture is shown in Appendix A, Figure 7. A notable feature of this architecture is that it has many rectangular filters and it prefers larger filters at the top layers. Like residual networks (**?**), the architecture also has many one-step skip connections. This architecture is a local optimum in the sense that if we perturb it, its performance becomes worse. For example, if we densely connect all layers with skip connections, its performance becomes slightly worse: 5.56%. If we remove all skip connections, its performance drops to 7.97%.

In the second set of experiments, we ask the controller to predict strides in addition to other hyper-parameters. As stated earlier, this is more challenging because the search space is larger. In this case, it finds a 20-layer architecture that achieves 6.01% error rate on the test set, which is not much worse than the first set of experiments.

Finally, if we allow the controller to include 2 pooling layers at layer 13 and layer 24 of the architectures, the controller can design a 39-layer network that achieves 4.47% which is very close to the best human-invented architecture that achieves 3.74%. To limit the search space complexity we have our model predict 13 layers where each layer prediction is a fully connected block of 3 layers. Additionally, we change the number of filters our model can predict from [24, 36, 48, 64] to [6, 12, 24, 36]. Our result can be improved to 3.65% by adding 40 more filters to each layer of our architecture. Additionally this model with 40 filters added is 1.05x as fast as the DenseNet model

that achieves 3.74%, while having better performance. The DenseNet model that achieves 3.46% error rate (**?**) uses 1x1 convolutions to reduce its total number of parameters, which we did not do, so it is not an exact comparison.

4.2 LEARNING RECURRENT CELLS FOR PENN TREEBANK

Dataset: We apply Neural Architecture Search to the Penn Treebank dataset, a well-known benchmark for language modeling. On this task, LSTM architectures tend to excel (**??**), and improving them is difficult (**?**). As PTB is a small dataset, regularization methods are needed to avoid overfitting. First, we make use of the embedding dropout and recurrent dropout techniques proposed in **?** and (**?**). We also try to combine them with the method of sharing Input and Output embeddings, e.g., **??**, especially **?** and **?**. Results with this method are marked with "shared embeddings."

Search space: Following Section 3.5, our controller sequentially predicts a combination method then an activation function for each node in the tree. For each node in the tree, the controller RNN needs to select a combination method in $[add, elem\_mult]$ and an activation method in $[identity, tanh, sigmoid, relu]$. The number of input pairs to the RNN cell is called the "base number" and set to 8 in our experiments. When the base number is 8, the search space is has approximately 6×10^{16} architectures, which is much larger than 15,000, the number of architectures that we allow our controller to evaluate.

Training details: The controller and its training are almost identical to the CIFAR-10 experiments except for a few modifications: 1) the learning rate for the controller RNN is 0.0005, slightly smaller than that of the controller RNN in CIFAR-10, 2) in the distributed training, we set S to 20, K to 400 and m to 1, which means there are 400 networks being trained on 400 CPUs concurrently at any time, 3) during asynchronous training we only do parameter updates to the parameter-server once 10 gradients from replicas have been accumulated.

In our experiments, every child model is constructed and trained for 35 epochs. Every child model has two layers, with the number of hidden units adjusted so that total number of learnable parameters approximately match the "medium" baselines (**??**). In these experiments we only have the controller predict the RNN cell structure and fix all other hyperparameters. The reward function is $\frac{c}{(\text{validation perplexity})^2}$ where c is a constant, usually set at 80.

After the controller RNN is done training, we take the best RNN cell according to the lowest validation perplexity and then run a grid search over learning rate, weight initialization, dropout rates and decay epoch. The best cell found was then run with three different configurations and sizes to increase its capacity.

Results: In Table 2, we provide a comprehensive list of architectures and their performance on the PTB dataset. As can be seen from the table, the models found by Neural Architecture Search outperform other state-of-the-art models on this dataset, and one of our best models achieves a gain of almost 3.6 perplexity. Not only is our cell is better, the model that achieves 64 perplexity is also more than two times faster because the previous best network requires running a cell 10 times per time step (**?**).

The newly discovered cell is visualized in Figure 8 in Appendix A. The visualization reveals that the new cell has many similarities to the LSTM cell in the first few steps, such as it likes to compute $W_1 * h_{t-1} + W_2 * x_t$ several times and send them to different components in the cell.

Transfer Learning Results: To understand whether the cell can generalize to a different task, we apply it to the character language modeling task on the same dataset. We use an experimental setup that is similar to **?**, but use variational dropout by **?**. We also train our own LSTM with our setup to get a fair LSTM baseline. Models are trained for 80K steps and the best test set perplexity is taken according to the step where validation set perplexity is the best. The results on the test set of our method and state-of-art methods are reported in Table 3. The results on small settings with 5-6M parameters confirm that the new cell does indeed generalize, and is better than the LSTM cell.

Additionally, we carry out a larger experiment where the model has 16.28M parameters. This model has a weight decay rate of $1e-4$, was trained for 600K steps (longer than the above models) and

Model	Parameters	Test Perplexity
? - KN-5	2M‡	141.2
? - KN5 + cache	2M‡	125.7
? - RNN	6M‡	124.7
? - RNN-LDA	7M‡	113.7
? - RNN-LDA + KN-5 + cache	9M‡	92.0
? - Deep RNN	6M	107.5
? - Sum-Prod Net	5M‡	100.0
? - LSTM (medium)	20M	82.7
? - LSTM (large)	66M	78.4
? - Variational LSTM (medium, untied)	20M	79.7
? - Variational LSTM (medium, untied, MC)	20M	78.6
? - Variational LSTM (large, untied)	66M	75.2
? - Variational LSTM (large, untied, MC)	66M	73.4
? - CharCNN	19M	78.9
? - Variational LSTM, shared embeddings	51M	73.2
? - Zoneout + Variational LSTM (medium)	20M	80.6
? - Pointer Sentinel-LSTM (medium)	21M	70.9
? - VD-LSTM + REAL (large)	51M	68.5
? - Variational RHN, shared embeddings	24M	66.0
Neural Architecture Search with base 8	32M	67.9
Neural Architecture Search with base 8 and shared embeddings	25M	64.0
Neural Architecture Search with base 8 and shared embeddings	54M	62.4

Table 2: Single model perplexity on the test set of the Penn Treebank language modeling task. Parameter numbers with ‡ are estimates with reference to **?**.

RNN Cell Type	Parameters	Test Bits Per Character
? - Layer Norm HyperLSTM	4.92M	1.250
? - Layer Norm HyperLSTM Large Embeddings	5.06M	1.233
? - 2-Layer Norm HyperLSTM	14.41M	1.219
Two layer LSTM	6.57M	1.243
Two Layer with New Cell	6.57M	1.228
Two Layer with New Cell	16.28M	1.214

Table 3: Comparison between our cell and state-of-art methods on PTB character modeling. The new cell was found on word level language modeling.

the test perplexity is taken where the validation set perplexity is highest. We use dropout rates of 0.2 and 0.5 as described in **?**, but do not use embedding dropout. We use the ADAM optimizer with a learning rate of 0.001 and an input embedding size of 128. Our model had two layers with 800 hidden units. We used a minibatch size of 32 and BPTT length of 100. With this setting, our model achieves 1.214 perplexity, which is the new state-of-the-art result on this task.

Finally, we also drop our cell into the GNMT framework (**?**), which was previously tuned for LSTM cells, and train an WMT14 English $\rightarrow$ German translation model. The GNMT network has 8 layers in the encoder, 8 layers in the decoder. The first layer of the encoder has bidirectional connections. The attention module is a neural network with 1 hidden layer. When a LSTM cell is used, the number of hidden units in each layer is 1024. The model is trained in a distributed setting with a parameter sever and 12 workers. Additionally, each worker uses 8 GPUs and a minibatch of 128. We use Adam with a learning rate of 0.0002 in the first 60K training steps, and SGD with a learning rate of 0.5 until 400K steps. After that the learning rate is annealed by dividing by 2 after every 100K steps until it reaches 0.1. Training is stopped at 800K steps. More details can be found in **?**.

In our experiment with the new cell, we make no change to the above settings except for dropping in the new cell and adjusting the hyperparameters so that the new model should have the same computational complexity with the base model. The result shows that our cell, with the same computational complexity, achieves an improvement of 0.5 test set BLEU than the default LSTM cell. Though this

improvement is not huge, the fact that the new cell can be used without any tuning on the existing GNMT framework is encouraging. We expect further tuning can help our cell perform better.

Control Experiment 1 – Adding more functions in the search space: To test the robustness of Neural Architecture Search, we add max to the list of combination functions and sin to the list of activation functions and rerun our experiments. The results show that even with a bigger search space, the model can achieve somewhat comparable performance. The best architecture with max and sin is shown in Figure 8 in Appendix A.

Control Experiment 2 – Comparison against Random Search: Instead of policy gradient, one can use random search to find the best network. Although this baseline seems simple, it is often very hard to surpass (**?**). We report the perplexity improvements using policy gradient against random search as training progresses in Figure 6. The results show that not only the best model using policy gradient is better than the best model using random search, but also the average of top models is also much better.

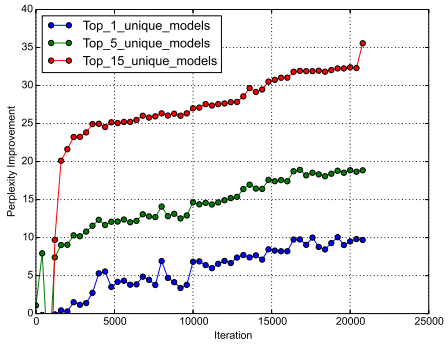

Figure 6: Improvement of Neural Architecture Search over random search over time. We plot the difference between the average of the top k models our controller finds vs. random search every 400 models run.

5 Conclusion

In this paper we introduce Neural Architecture Search, an idea of using a recurrent neural network to compose neural network architectures. By using recurrent network as the controller, our method is flexible so that it can search variable-length architecture space. Our method has strong empirical performance on very challenging benchmarks and presents a new research direction for automatically finding good neural network architectures. The code for running the models found by the controller on CIFAR-10 and PTB will be released at https://github.com/tensorflow/models . Additionally, we have added the RNN cell found using our method under the name NASCell into TensorFlow, so others can easily use it.

Acknowledgments

We thank Greg Corrado, Jeff Dean, David Ha, Lukasz Kaiser and the Google Brain team for their help with the project.

A APPENDIX

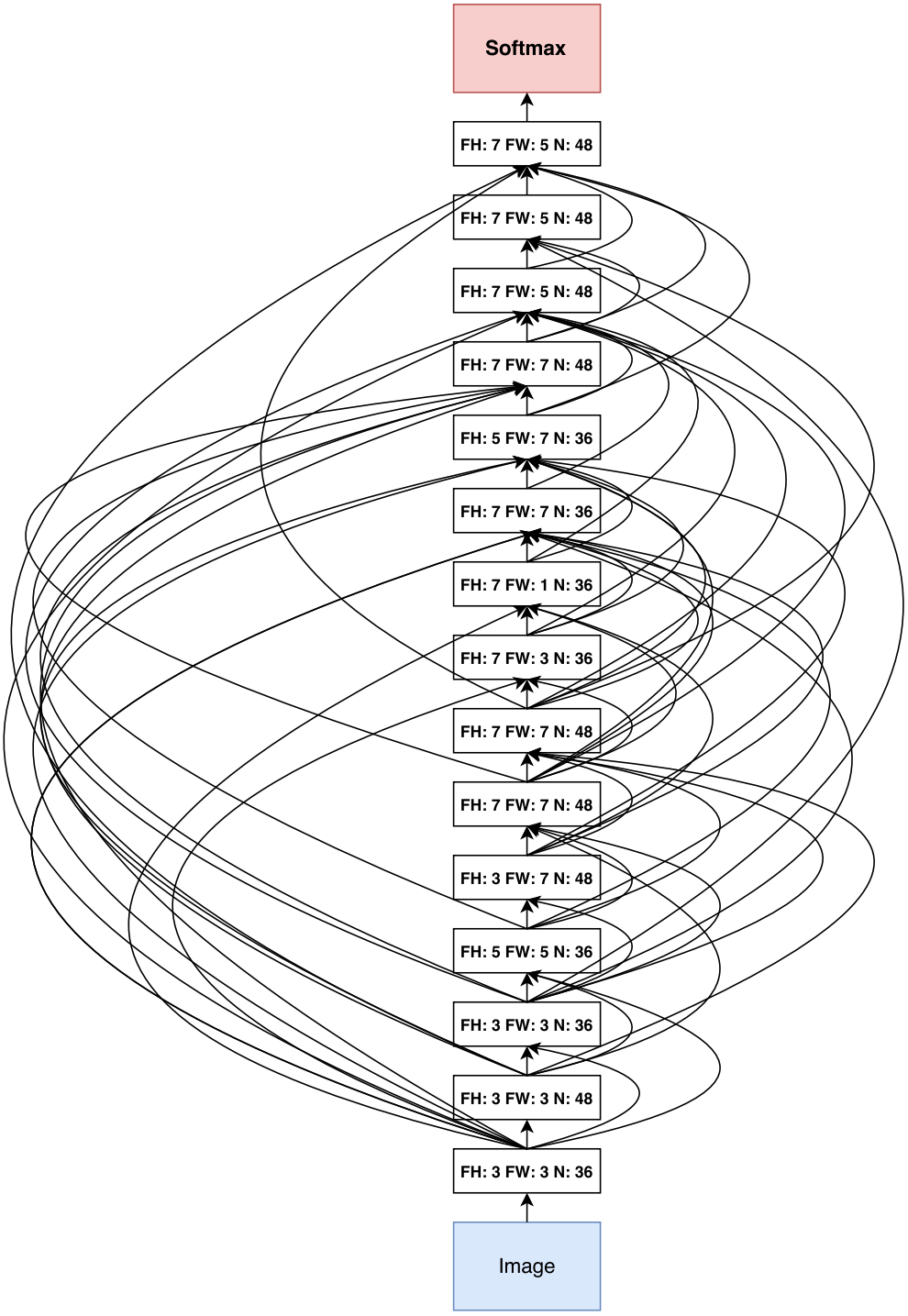

Figure 7: Convolutional architecture discovered by our method, when the search space does not have strides or pooling layers. FH is filter height, FW is filter width and N is number of filters. Note that the skip connections are not residual connections. If one layer has many input layers then all input layers are concatenated in the depth dimension.

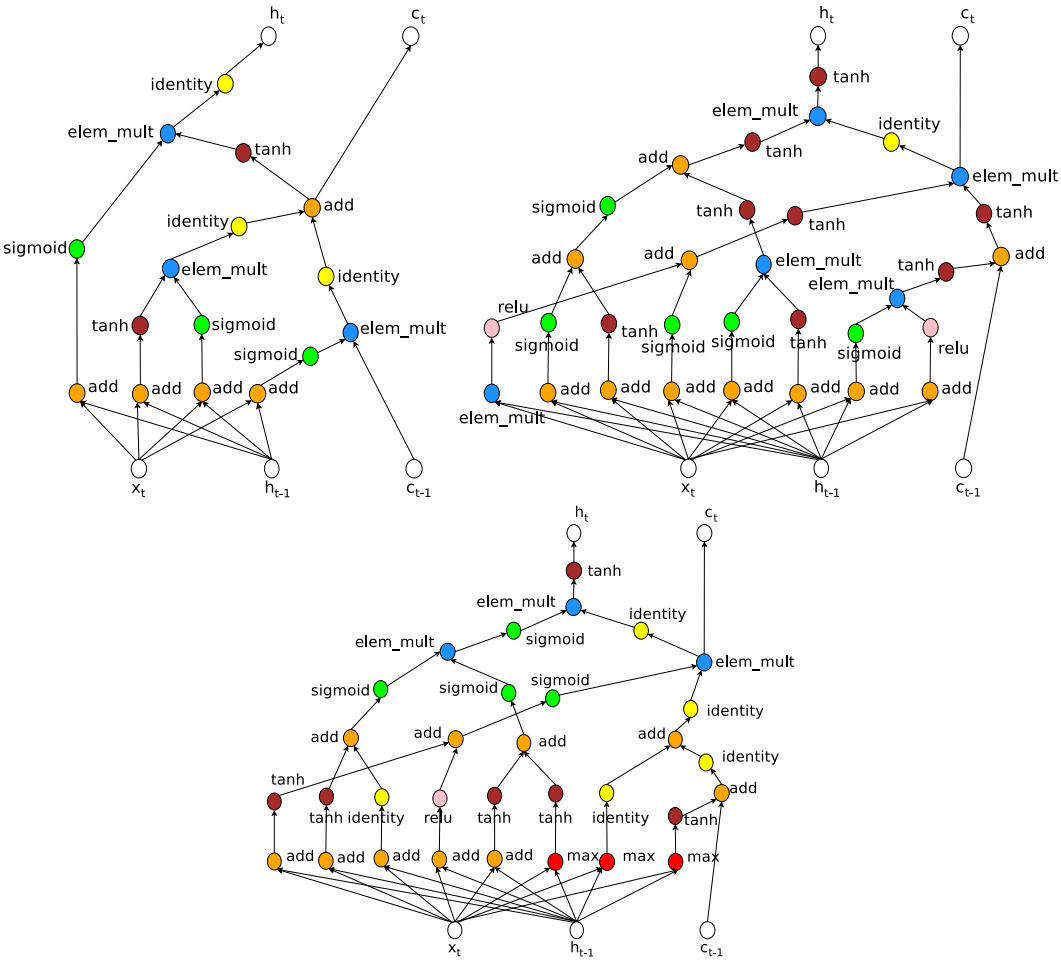

Figure 8: A comparison of the original LSTM cell vs. two good cells our model found. Top left: LSTM cell. Top right: Cell found by our model when the search space does not include max and sin. Bottom: Cell found by our model when the search space includes max and sin (the controller did not choose to use the sin function).