Neural Architecture Search for Reinforcement Learning

Barret Zoph , Quoc V. Le [1]

[1]Google Brain

ICLR 2017/ Presenter: Ji Gao

Outline

1. Motivation
2. Method
   - Overview
   - More complicated model structure
3. Experiment

Motivation

- Choose neural network architecture is hard: Need expert knowledge and ample time
- Motivation of this paper: Automatically design neural network architecture.

Related Work

- Hyperparameter optimization: Limited in a fixed-length space.
- Neuro-evolution algorithms: Flexible but not applicable in large scale.
- Other related topics:
  - Program synthesis
  - End-to-end sequence to sequence learning
  - Neural Turing Machine
  - Learning to learn, meta-learning

Method overview

- Based on the observation that the structure and connectivity of a neural network can be specified as a variable-length string.
- A controller generate the hyperparameters.

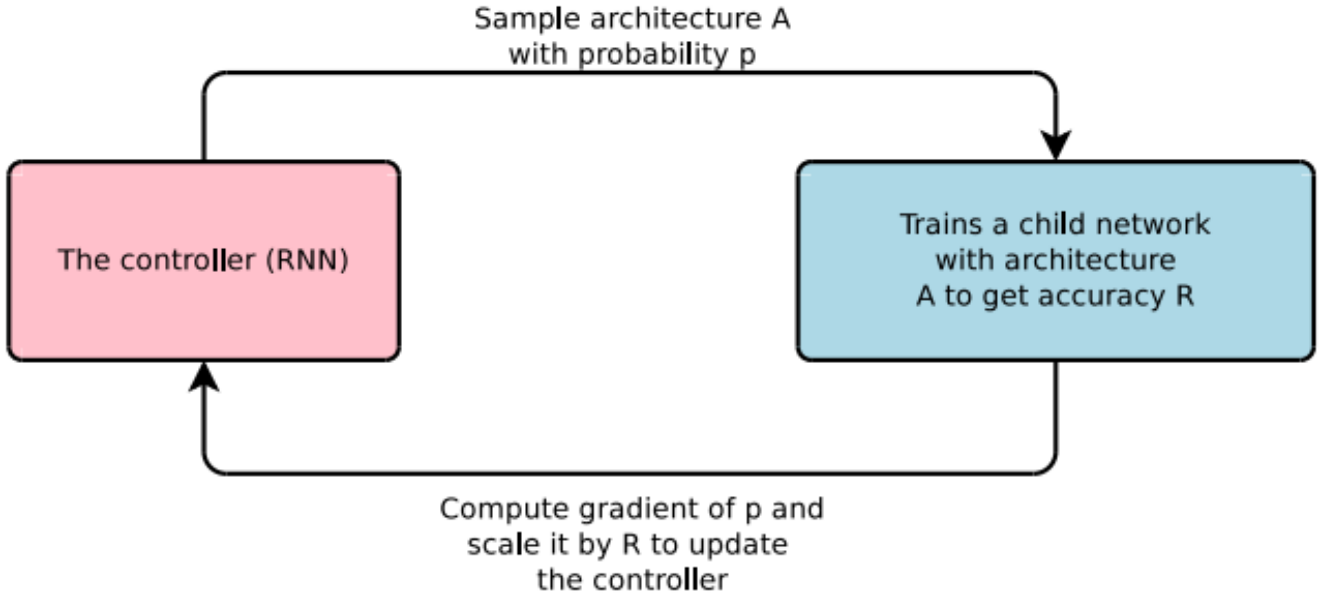


Figure 1: An overview of Neural Architecture Search.

Controller

- The controller works like this: A RNN generator.

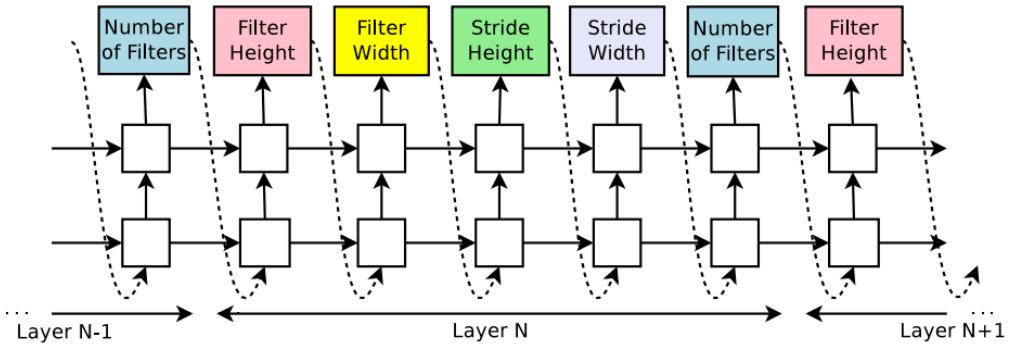


Figure 2: How our controller recurrent neural network samples a simple convolutional network. It predicts filter height, filter width, stride height, stride width, and number of filters for one layer and repeats. Every prediction is carried out by a softmax classifier and then fed into the next time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

Optimize the controller

- Parameters of the controller: θ. How should θ be optimized?
- Use the accuracy of the sampled, trained network.
- Reinforcement learning: Treat the accuracy as the **reward** R
- Suppose each time the controller generates T hyperparameters $a_1..a_T$ (actions in this setting), we optimize the expectation of Reward:

$$J(\theta) = E_{P(a_1..a_T;\theta)}[R(a_1..a_T)]$$

REINFORCE algorithm

- Problem: R is non-differentiable. How can we get $\nabla_\theta J(\theta)$?
- Use policy gradient methods. REINFORCE algorithm is one of them.
- We know that

$$\begin{aligned}\nabla_\theta J(\theta) &= \sum_A \nabla_\theta P(A;\theta) R(A) \\ &= \sum_A P(A;\theta) \nabla_\theta \log P(A;\theta) R(A) \\ &= E_{P(A;\theta)}[\nabla_\theta \log P(A) R(A)]\end{aligned}$$

- This is called the REINFORCE trick
- And the expectation can be approximate by sampling.

Optimize the controller network

- Equation is:

$$\nabla_\theta J(\theta) = \sum_{t=1}^{T} E[\nabla_\theta \log P(a_t|a_1..a_{t-1};\theta)R]$$

$$\approx \frac{1}{m}\sum_{k=1}^{m}\sum_{t=1}^{T} \nabla_\theta \log P(a_t|a_1..a_{t-1};\theta)R_k$$

- Each time the controller generate a batch of m group of hyperparameters.
- m networks are trained to get the accuracy(reward).
- θ is then updated by the gradient.
- In real practice, the equation is actually

$$\approx \frac{1}{m}\sum_{k=1}^{m}\sum_{t=1}^{T} \nabla_\theta \log P(a_t|a_1..a_{t-1};\theta)(R_k - b)$$

to reduce bias. In their experiment, b is set to be the exponential moving average of the previous architecture accuracies

Distributed training to accelerate

- Training a child network take hours
- Train distributedly to accelerate
- Multiple controller run multiple child models.

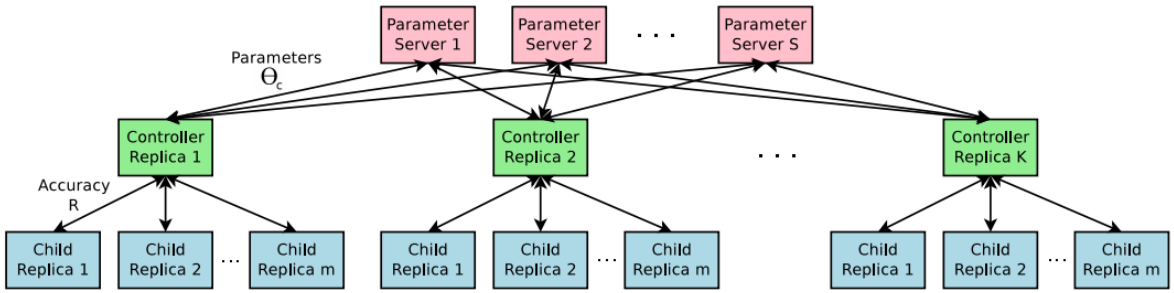
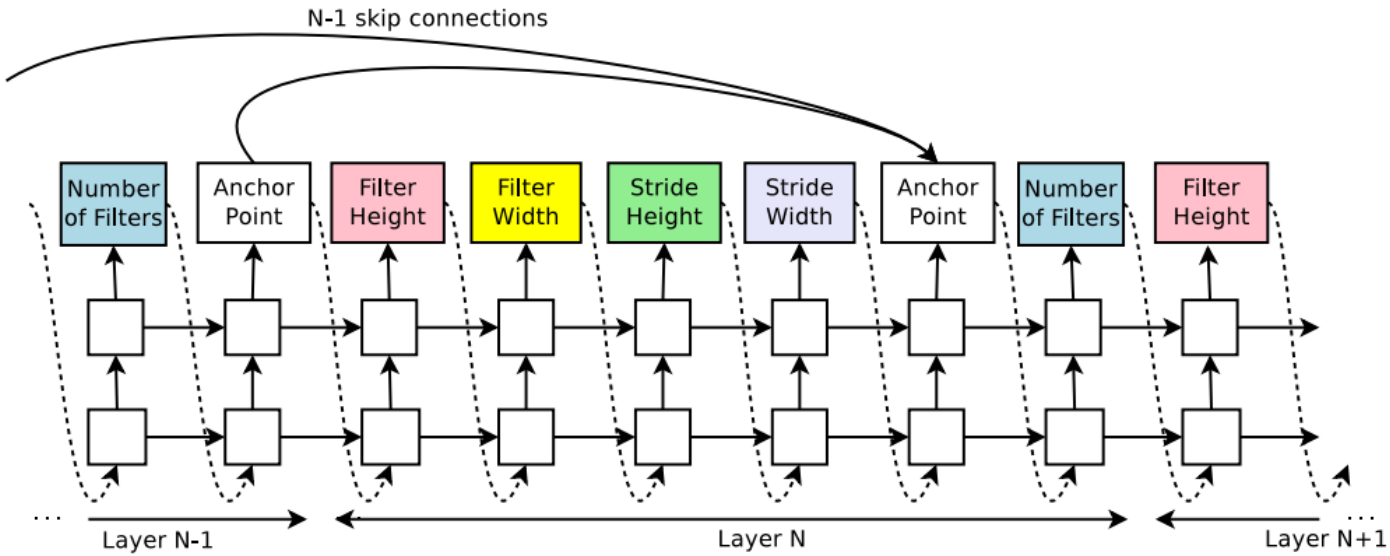


Figure 3: Distributed training for Neural Architecture Search. We use a set of S parameter servers to store and send parameters to K controller replicas. Each controller replica then samples m architectures and run the multiple child models in parallel. The accuracy of each child model is recorded to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

Skip connections

- Skip connections exist in Googlenet and ResNet, which are crucial.
- Add an anchor node to handle such skip connections

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



Recurrent layers

- Model recurrent structure as linking the inputs: x_t and h_{t-1}
- Controller RNN produce combination methods (addition, elementwise multiplication, etc.) and activation functions (tanh, sigmoid, etc.) to merge inputs and produce one output.

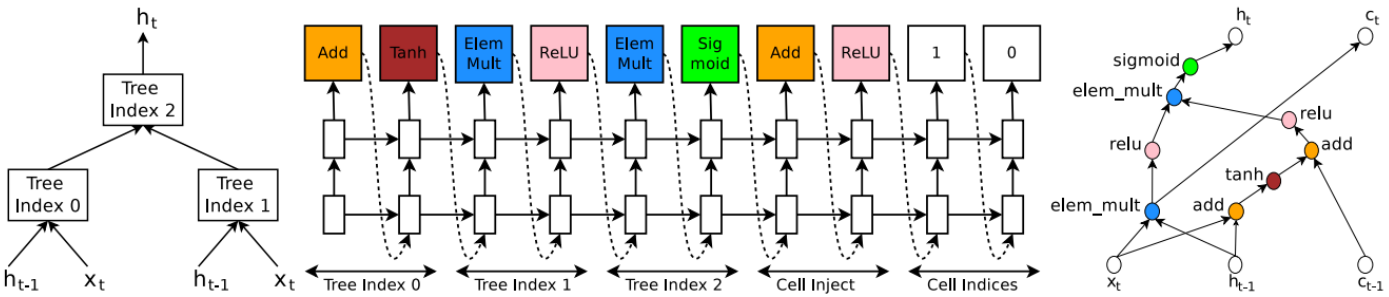


Figure 5: An example of a recurrent cell constructed from a tree that has two leaf nodes (base 2) and one internal node. Left: the tree that defines the computation steps to be predicted by controller. Center: an example set of predictions made by the controller for each computation step in the tree.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

Experiment design

Data:

- CIFAR-10
- Penn Treebank

The child network are tested using a validation dataset. Reported performance on the test set is computed only once.

CIFAR10

CNN for CIFAR10 task.
Details:

- Conv layers with Relu: filter height and width in [1, 3, 5, 7] and a number of filters in [24, 36, 48, 64].
- For strides, test both 1 and [1, 2, 3].
- Also batch normalization and skip connections between layers

For the controller:

- Two-layer LSTM with 35 hidden units on each layer.
- Optimized with the ADAM with a learning rate of 0.0006.
- Initialization: Between -0.08 and 0.08.
- Number of parameter server shards S= 20, Number of controller replicas K=100 and the number of child replicas m=8, which means there are 800 networks being trained on 800 GPUs concurrently at any time.

CIFAR10 result

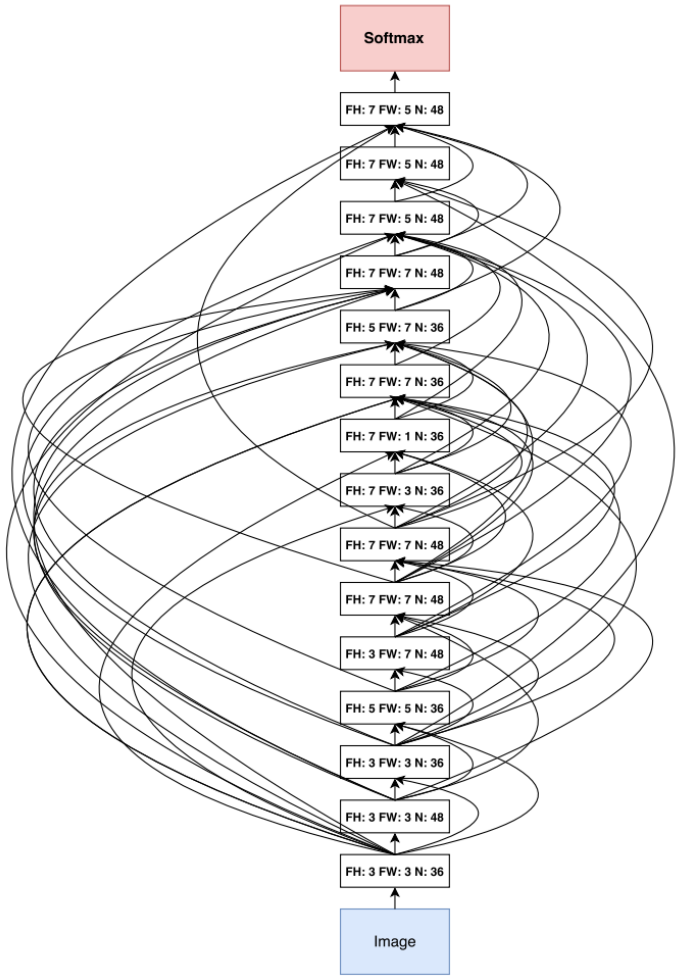


Figure 7: Convolutional architecture discovered by our method, when the search space does not have strides or pooling layers. FH is filter height, FW is filter width and N is number of filters. Note that the skip connections are not residual connections. If one layer has many input layers then all input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

CIFAR10

Model	Depth	Parameters	Error rate (%)
Network in Network (Lin et al., 2013)	-	-	8.81
All-CNN (Springenberg et al., 2014)	-	-	7.25
Deeply Supervised Net (Lee et al., 2015)	-	-	7.97
Highway Network (Srivastava et al., 2015)	-	-	7.72
Scalable Bayesian Optimization (Snoek et al., 2015)	-	-	6.37
FractalNet (Larsson et al., 2016)	21	38.6M	5.22
with Dropout/Drop-path	21	38.6M	4.60
ResNet (He et al., 2016a)	110	1.7M	6.61
ResNet (reported by Huang et al. (2016c))	110	1.7M	6.41
ResNet with Stochastic Depth (Huang et al., 2016c)	110	1.7M	5.23
	1202	10.2M	4.91
Wide ResNet (Zagoruyko & Komodakis, 2016)	16	11.0M	4.81
	28	36.5M	4.17
ResNet (pre-activation) (He et al., 2016b)	164	1.7M	5.46
	1001	10.2M	4.62
DenseNet ($L = 40, k = 12$) Huang et al. (2016a)	40	1.0M	5.24
DenseNet($L = 100, k = 12$) Huang et al. (2016a)	100	7.0M	4.10
DenseNet ($L = 100, k = 24$) Huang et al. (2016a)	100	27.2M	3.74
DenseNet-BC ($L = 100, k = 40$) Huang et al. (2016b)	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.

Penn Treebank

Task: Build a good language model
Details:

- For each node in the tree, 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=8
- Use perplexity($e^{-\frac{1}{N}\sum_i \ln p(x_i)}$) instead of accuracy in this case

Controller:

- Similar to the CIFAR-10 experiments
- learning rate for the controller RNN is 0.0005
- Distributed setting: S=20, K=400 and m to 1, which means there are 400 networks being trained on 400 CPUs

Try to combine with the method of sharing Input and Output embeddings

Penn treebank result

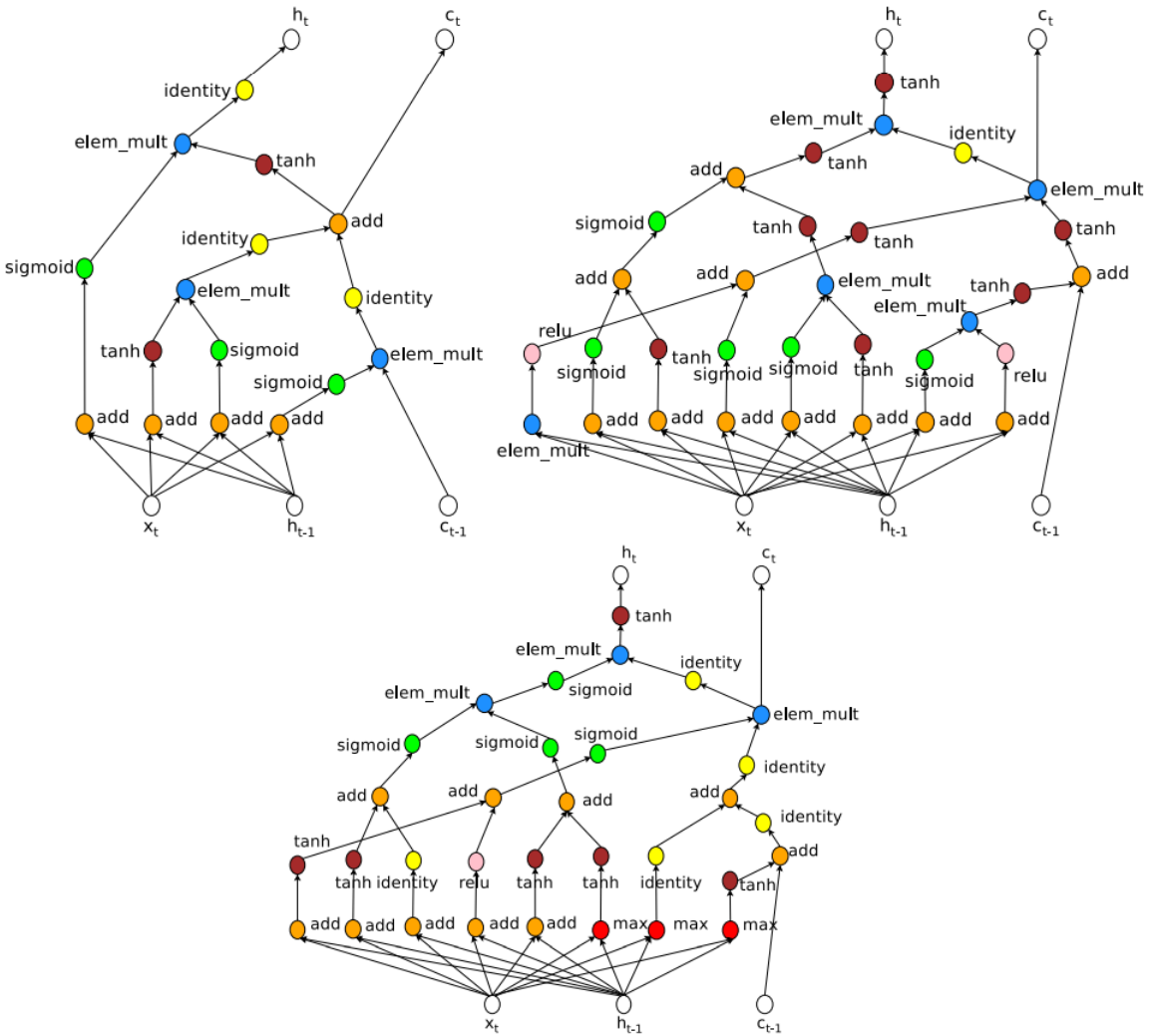


Figure 8: A comparison of the original LSTM cell vs. two good cells our model found. Top left: LSTM cell. Top right: Cell found by our model when the search space does not include max and sin. Bottom: Cell found by our model when the search space includes max and sin (the controller did not choose to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

Penn Treebank

Model	Parameters	Test Perplexity
Mikolov & Zweig (2012) - KN-5	2M‡	141.2
Mikolov & Zweig (2012) - KN5 + cache	2M‡	125.7
Mikolov & Zweig (2012) - RNN	6M‡	124.7
Mikolov & Zweig (2012) - RNN-LDA	7M‡	113.7
Mikolov & Zweig (2012) - RNN-LDA + KN-5 + cache	9M‡	92.0
Pascanu et al. (2013) - Deep RNN	6M	107.5
Cheng et al. (2014) - Sum-Prod Net	5M‡	100.0
Zaremba et al. (2014) - LSTM (medium)	20M	82.7
Zaremba et al. (2014) - LSTM (large)	66M	78.4
Gal (2015) - Variational LSTM (medium, untied)	20M	79.7
Gal (2015) - Variational LSTM (medium, untied, MC)	20M	78.6
Gal (2015) - Variational LSTM (large, untied)	66M	75.2
Gal (2015) - Variational LSTM (large, untied, MC)	66M	73.4
Kim et al. (2015) - CharCNN	19M	78.9
Press & Wolf (2016) - Variational LSTM, shared embeddings	51M	73.2
Merity et al. (2016) - Zoneout + Variational LSTM (medium)	20M	80.6
Merity et al. (2016) - Pointer Sentinel-LSTM (medium)	21M	70.9
Inan et al. (2016) - VD-LSTM + REAL (large)	51M	68.5
Zilly et al. (2016) - 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