Neural Architecture Search with Reinforcement Learning

BARET Zoph, Quoc V. Le

Google Brain

ICLR 2017

2021.04.07

최영제

Index

1. Introduction
2. Related Works
3. Methods
4. Experiments and Results

Introduction

Background

- Neural Architecture Search (NAS) is a gradient-based method for finding good architectures
- NAS use a recurrent network – the controller – to generate string which denoted hyperparameter of architectures
- Training the network specified by the string – the "child network" – on the real data will result in an accuracy on a validation set

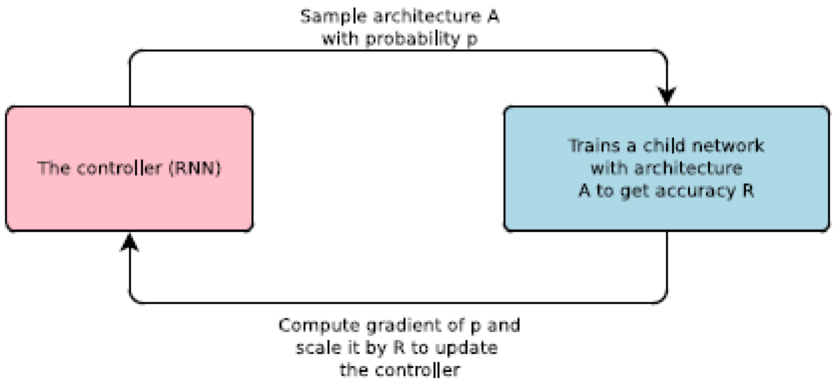


Figure 1: An overview of Neural Architecture Search.

Related works

Various kinds of auto machine learning

- Hyperparameter optimization

  → Lots of these methods are limited in that they only search models from a fixed-length space

  → Bayesian optimization methods that allow to search non fixed length architectures they are less general and less flexible

- Neuro-evolution argorithms

  → These are search-based methods, so they are slow and require many heuristics to work well

- Neural architecture search

  → End-to-end sequence learning

  → NAS is learned directly from the reward signal without any heuristic informations

  → More general and flexible compare to above methods

Methods

Controller descriptions

- RNN structure
- Every prediction is carried out by a softmax classifier
- Every output is fed into the next time step as input
- If the number of layers exceeds a certain value, the process of generating an architecture stops

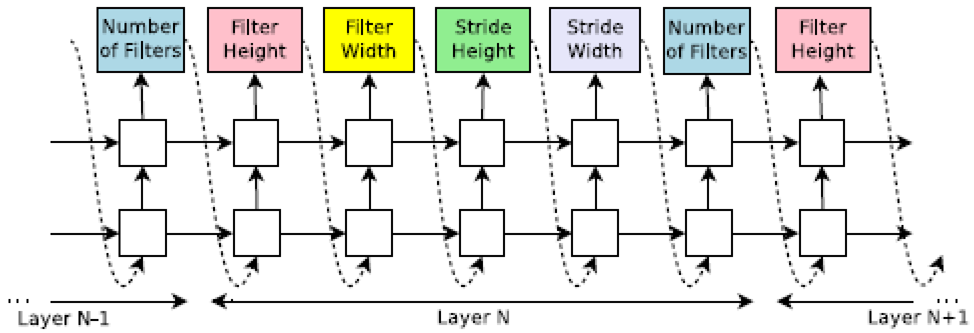


Figure 2: How our controller recurrent neural network samples a simple convolutional network. It predicts filter height, filter width, stride height, stride width, and number of filters for one layer and repeats. Every prediction is carried out by a softmax classifier and then fed into the next time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

Methods

Training with REINFORCE

- The parameters of the controller recurrent neural network (RNN) is noted θ_c

  → Have to optimized in order to maximize the validation accuracy of the proposed architectures

- For optimized the parameters, policy gradient methods is used

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

- An empirical approximation of the above quantity is follow

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

m is number of different architectures (episode)
T is number of hyperparameters our controller has to predict to design a neural network (length of RNN per episode)
R is the accuracy which achieved on a held-out dataset using child network (R_k : validation score of k-th neural network architecture)

Methods

Training with REINFORCE-baseline & Accelerate training

- Above gradient has high variance, thus using baseline for reduce variance

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

- Using exponential moving average of the previous architecture accuracies for baseline function b
- 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

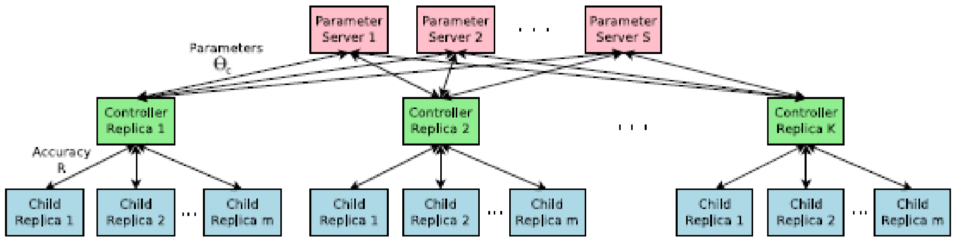


Figure 3: Distributed training for Neural Architecture Search. We use a set of S parameter servers to store and send parameters to K controller replicas. Each controller replica then samples m architectures and run the multiple child models in parallel. The accuracy of each child model is recorded to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

Methods

Expand search space include skip connections, branching layers

- To enable the controller to predict skip connections, they add an anchor point
- Anchor point has N-1 outputs with sigmoid activation function
- If j-th anchor point predict over threshold, then j-th layer connect to i-th layer
- Concatenated in the depth dimension, pad the small layers with zeros

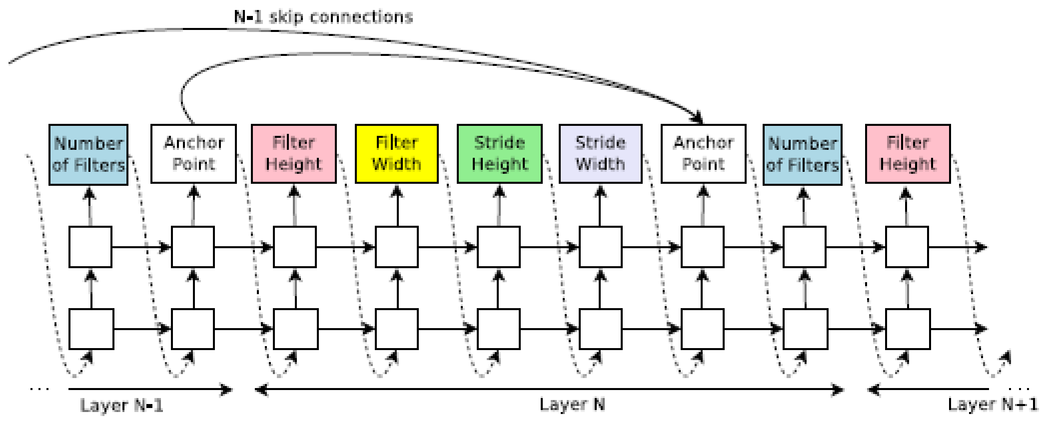
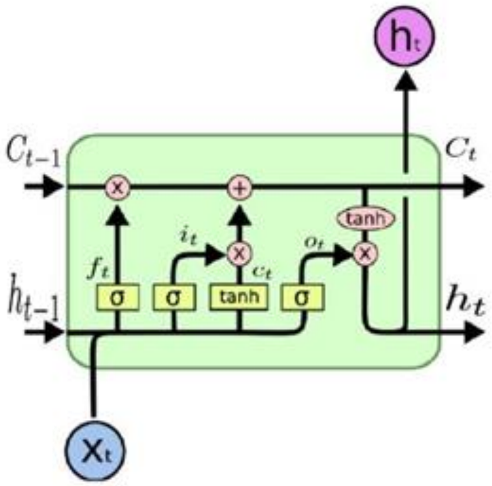


Figure 4: The controller uses anchor points, and set-selection attention to form skip connections.

Methods

Generate Recurrent cell architectures

- Long short term memory (LSTM) cells has three input (x_t, h_{t-1}, c_{t-1}) and two output (h_t, c_t)
- To make h_t, we need to calculate follow



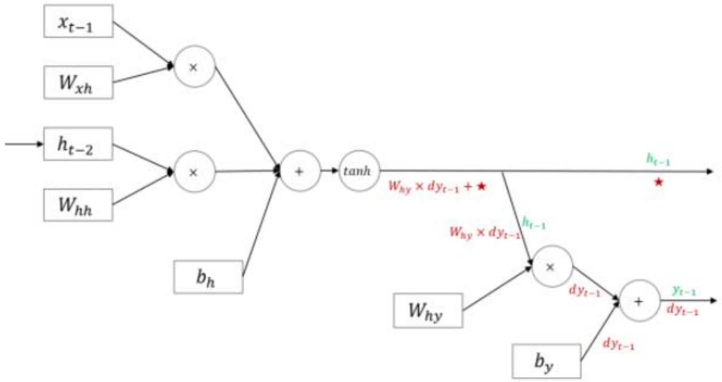
$$a_0 = sigmoid\,(W_{xh\_f} * x_t + W_{hh\_f} * h_{t-1})$$

$$a_0^{new} = tanh(a_0 \odot c_{t-1} + a_1)$$

$$a_1 = sigmoid\,(W_{xh\_i} * x_t + W_{hh\_i} * h_{t-1}) \odot tanh\,(W_{xh\_g} * x_t + W_{hh\_g} * h_{t-1})$$

$$a_2 = a_0^{new} \odot a_0$$

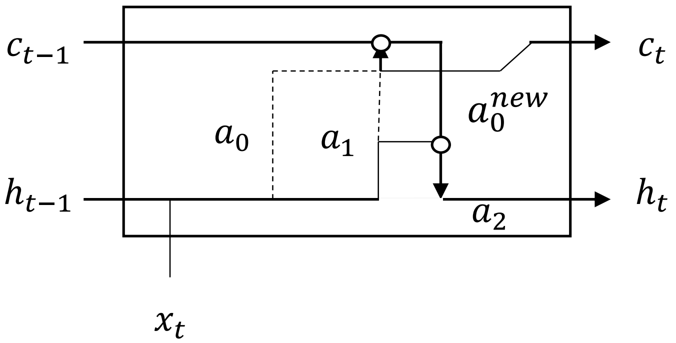
- RNN structure including LSTM, can be interpreted by tree structure as follow



Methods

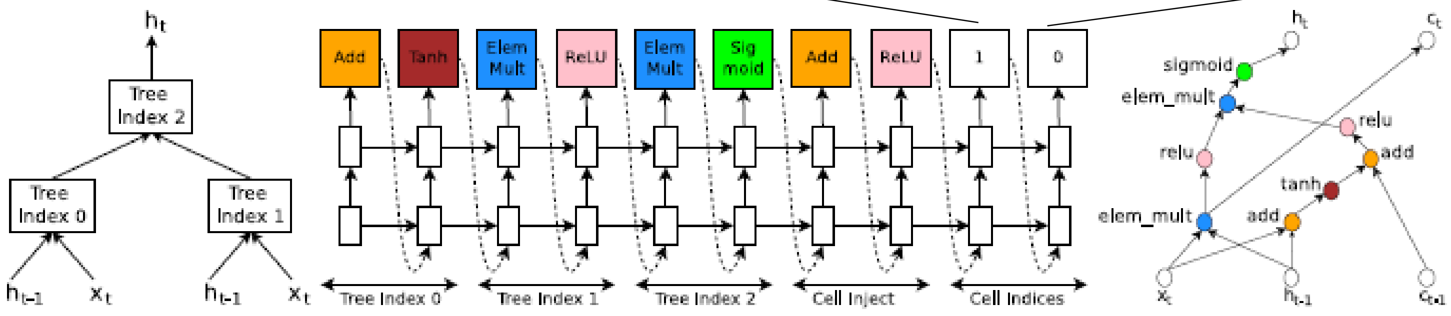
Generate Recurrent cell architectures

- Example of LSTM controller



Tree index which to make c_t

Tree index to be combined with c_{t-1}



- 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 $a_1 = \text{ReLU}\big((W_3 * x_t) \odot (W_4 * h_{t-1})\big)$.
- 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 $a_0^{new} = \text{ReLU}(a_0 + c_{t-1})$. 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 $a_2 = \text{sigmoid}(a_0^{new} \odot a_1)$. 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., $c_t = (W_3 * x_t) \odot (W_4 * h_{t-1})$.

Experiments and Results

Model descriptions

- Results on cifar-10

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.

Experiments and Results

Model descriptions

- Results on penn treebank (language modeling)

* Perplexity : 특정 시점에서 평균적으로 몇 개의 선택지를 가지고 고민하고 있는지

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

Table 2: Single model perplexity on the test set of the Penn Treebank language modeling task. Parameter numbers with ‡ are estimates with reference to Merity et al. (2016).

Q&A