Neural Architecture Search with Reinforcement Learning: A Review

Ashutosh Jha (ME14B148)

February 2017

1 Introduction

Deep Learning has shown lot of promise for different tasks in varied fields. It is however noted that designing the architectures of the deep networks is a time-consuming task which also requires expert domain knowledge.
As a result, deep learning is not *end-to-end learnt* as it is still humans who design these architectures. This paper presents an approach to *navigate* through the architecture space for obtaining good architectures.

2 The Million Dollar Idea (Or the TL;DR)

The idea the authors propose is simple. The structure and connectivity of a neural network can be specified by a variable-length string. So, it is possible for a recurrent neural network to generate such a string. The RNN is now trained using a reinforcement learning algorithm (REINFORCE in this paper) where the reward of choosing an architecture(the "action") is essentially the validation accuracy obtained by the chosen architecture on the dataset. Since the RNN gets higher rewards on choosing better architectures, if trained for long enough, it would essentially learn to choose a good architecture.

3 The (Gory) Details

3.1 Generating Model Descriptions using a RNN Controller

For simplicity, at this point the paper assumes that we want to predict a network with only convolutional layers. So the NAS architecture is essentially supposed to return to us a set of hyper-parameters of each of these conv layers(i.e. filter height,width, strides, number of filters,etc.) as a sequence of tokens.

In the paper, the generation of architecture stops once the number of layers exceeds a certain threshold value. This threshold is increased as the training

progresses (which makes sense as at a point you are supposed to be near the best architecture for threshold or less number of layers, provided you aren't increasing the threshold too fast).
Once an architecture is obtained, accuracy of the architecture on a fixed validation set is recorded which is then used as a reward as explained in the next section.

3.2 Training with REINFORCE

The list of hyper-parameters predicted by the network can be viewed as a set of actions $\{a_1, a_2, \cdots, a_T\}$ and the state at a given time-step is essentially the history of actions taken in the past i.e. $s_t = \{a_1, a_2, \cdots, a_{t-1}\}$. The reward of choosing a set of actions say R is essentially the validation accuracy of the architecture defined by the set of actions. With these definitions of state action and rewards, we can now use a Reinforcement Learning algorithm to find optimal actions to maximize the expected return (in this case just the reward because you get a single reward at the end).

The paper uses REINFORCE as its reinforcement learning algorithm. The gradient of the performance is thus given by:

$$\nabla_{\theta_c} J(\theta_c) = \sum_{t=1}^{T} \mathbb{E}_{P(a_{1:T};\theta_c)}[\nabla_{\theta_c} P(a_t|s_t;\theta_c) R] \tag{1}$$

which can now be empirically estimated as:

$$\frac{1}{m} \sum_{k=1}^{M} \sum_{t=1}^{T} [\nabla_{\theta_c} P(a_t|s_t;\theta_c) R_k] \tag{2}$$

where m is the number of different architectures sampled in one batch and T is the number of hyperparameters predicted.

The above update is unbiased but has very high variance and can thus suffer from slow learning. The variance due to REINFORCE can be reduced by introducing a baseline as follows:

$$\frac{1}{m} \sum_{k=1}^{M} \sum_{t=1}^{T} [\nabla_{\theta_c} P(a_t|s_t;\theta_c)](R_k - b) \tag{3}$$

This baseline can be set to anything as long as it doesn't depend on the current action taken. The paper uses an exponential moving average of the previous architecture accuracies as the baseline b.

Side Note to the Reader: I tried figuring out why the authors used REINFORCE as their training algorithm rather than something n-step TD like

Actor-Critic and haven't found any satisfactory answer yet. I mailed the authors and they haven't replied yet so it's either something very trivial or it was just chosen based on their choice. From what I see, using A3C instead of REINFORCE should clearly speed up the process since its variance is low.

3.3 Accelerating Training with Parallelism and Asynchronous Updates

As seen above, the update step can be taken only when the R_k's are known. Thus, we would essentially have to wait till each child network gets trained to make one update to the RNN controller which is clearly tedious. The paper thus suggests using parallelism where you would have a copy of the RNN controller in different threads and each thread would have children being trained and would thus be accumulating gradients. The updates from the threads are then applied asynchronously to the main controller after a certain number of steps and this process is repeated. This greatly speeds up the training and allows parallel training on different GPU's.

3.4 Increasing Architecture Complexity

3.4.1 Skip-Connections

The architecture of the RNN controller defined so far doesn't allow us to introduce skip connections which are used in many modern architectures like GoogleNet and ResNets. To allow for such skip connections, a simple anchor trick was applied.
At layer N , we add an anchor point which has N - 1 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 hiddenstates of the previous N - 1 anchor points which is given by:

$$\text{P(layer j is input to layer i)} = \sigma(v^T \tanh(W_{prev} h_j + W_{curr} h_i)) \tag{4}$$

v, W_{prev}, W_{curr} are learnable parameters and h_i, h_j are the hiddenstates at layer i and j.

Skip connections can cause "compilation failures" where one layer is not compatible with another layer, or a layer may not have any input or output. To handle these, the paper employs three simple techniques:

1. If a layer is not connected to any input layer then the image is used as the input layer.
2. At the final layer take all layer outputs that have not been connected and concatenate them before sending this final hiddenstate to the classifier.
3. If input layers to be concatenated have different sizes, pad the small layers with zeros so that the concatenated layers have the same sizes.

3.4.2 Other Layer Types

In section 3.1, we assumed that the network only consists of conv layers. This assumption can be gotten around by adding a step after every layer of required network to pick the next layer's type (pool or conv or batch normalize) and then have layers in the controller for their appropriate hyperparameters.

3.5 Generating Recurrent Cell Architectures

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 idea to getting a Recurrent Cell Architecture is pretty simple if you break down the method explained in the paper. Since you have a tree of steps, you can further expand this tree of steps into individual computations (and thus a computation graph) and now, you actually treat each step of the tree as you treated a layer earlier and each computation in a step as you did for a hyperparameter earlier. At the end of the controller, you have two more layers to represent the memory states of the LSTM. Based on the number of leaf nodes, you define the base of recurrent cell architecture. The paper makes use of base 8 architecture to make the cell more expressive.

Side note to the reader: I still don't like the idea of choosing a base 8 architecture. This is because, an LSTM is supposed to have a base of 4 so are their comparisons with other models even fair?

3.6 Experiments and Results

The authors show results of architectures chosen by NAS on CIFAR-10 and Penn Treebank datasets. The best architectures were chosen after the a certain threshold number of networks were evaluated(12800 for CIFAR-10 and 15000 for PTB). The architectures chosen by NAS beat the present state of the art networks for the Penn Treebank and came extremely close in case of CIFAR-10.

The rewards for the 2 tasks were chosen heuristically (so as to set the rewards between 0 and 1) as:

1. CIFAR-10 : (max. validation accuracy of last 5 epochs)^3
2. PTB : $\frac{c}{(\text{validation perplexity})^3}$ where c=80 usually

The experiments for CIFAR-10 was done was done concurrently on 800 GPU's running 800 children networks and the experiments for Penn Treebank were done on 400 CPU's running 400 children networks. As to why CPU's were used for Penn Treebank, the authors clarified on OpenReview that it was only because they had many of them at their disposal.

The authors also performed 2 very interesting control experiments to show that their work is actually important:

1. The authors test the robustness of NAS and show that even with a bigger search space, NAS gives comparable performance.

2. This expt. was more interesting. Earlier work by Bestraga & Bengio showed that random search performs really well in hyperparameter search and is a baseline which is hard to beat. The authors show that that not only is the best model using NAS better than the best model using random search, but the average of top models (5 and 15 top models) is also much better.

4 Drawbacks of the paper

1. The amount of computational power required to do perform NAS is enormous. It makes reproducing the results hard and also makes it harder to use in practice.

2. The base number of the recurrent cell used was 8 and an LSTM has a base number of 4. Making comparisons thus doesn't seem fair anymore. The authors did try to address the issue on OpenReview but there doesn't seem to be any comparisons for a base number of 4 to verify their claims.

5 Further Possible Directions of Research:

1. **Blackhole of Computational Power:** As mentioned in the previous section, the amount of computational power used to run NAS is very very large. Ways to make it faster and efficient in terms of the number of steps to convergence for the controller are clearly a very important area for further research.
   One suggestion which I feel would clearly help is using Actor-Critic (A3C in particular) as the RL algorithm to train the controller as it has much lower variance and is bound to converge faster.

2. **A Bootstrapping AI:** NAS could essentially be used to bootstrap a given neural architecture to improve its own architecture and then continually bootstrap to reach the optimal architecture (Rise of Machines is here!). We could thus in principle give it a human designed architecture to start with and then let it improve from that point to get better architectures.