R10/Sarthak Mittal/200050129

March 13, 2023

This paper uses a **recurrent neural network (RNN)** as a controller to generate a variable-length string that specifies a neural network (NN). Using **reinforcement learning (RL)** with accuracy as the reward, they train the RNN with policy gradient updates. This method was even able to design good models **from scratch** and achieved novel results in image recognition and language modeling tasks compared to **previous state-of-the-art** architectures.

Past methods of hyper-parameter optimization were limited to fixed-length space, while Bayesian optimization methods had lesser flexibility. The neuro-evolution algorithms are more flexible but not practical for large scale because of being search-based. There are some connections to inductive programming using search and machine learning (ML). The controller RNN is **auto-regressive** (prediction one at a time conditioned on past), but unlike sequence-to-sequence, uses a **non-differentiable metric** (accuracy of child network) and learns directly from the reward (**no supervised bootstrapping**), thus incorporating **meta-learning** and the idea of using NN and RL to update another NN.

The RNN controller generates the hyper-parameters as a **token sequence** (for example, the number of filters and filter/stride dimensions for each layer in a CNN). By **bounding** the number of layers, they complete generation and then record **validation accuracy (reward)**. The parameters θ_c are updated according to the maximization of the expected reward.

The list of tokens predicted by the controller is considered a **list of actions** to design an architecture. The expected validation accuracy, represented by $J(\theta_c)$, is maximized using an approximate version of the **REINFORCE** algorithm. To control the variance, they use a **baseline** function that is an exponential moving average of past accuracies. While training, **parallelism** was achieved using distributed training, asynchronous updates, shared parameters among controller replicas, and parameter update servers.

The authors used a set-selection type attention to incorporate **"skip connections"**. To tackle the "compilation failures" (resulting due to incompatible layers or absence of input/output), they (1) used the **image** if there was no input layer (2) concatenated **unconnected** layer outputs (3) **padded** smaller input layers with zeros before concatenation. To account for more **types of layers** (pooling, normalization, batch-norm, etc.), they add another step to predict the layer type.

The computation involved in the generation of recurrent cells is modeled as a **tree of steps** where each indexed node is labeled with an **aggregator** (addition, multiplication, etc.) and an **activation** (tanh, sigmoid, etc.). They added additional variables to represent memory states in LSTM cells.

They ran experiments for convolutional architectures for **CIFAR-10** and recurrent cell architecture for **Penn Treebank** and achieved **promising results** when compared with the best human-invented architectures. After conducting a few "control experiments" they concluded that Neural Architecture Search (NAS) was relatively **robust** to search space size (tested using max aggregator and sin activation), and policy gradient proved to be a better method than random search. The RNN cell found using their approach has been incorporated as **NASCell** in TensorFlow.