The First Step-by-Step Guide for Implementing Neural Architecture Search with Reinforcement Learning Using TensorFlow

Wallarm

Dec 12, 2017 · 7 min read

Our team is no stranger to various flavors of AI including deep learning (DL). That's why we've immediately noticed when Google came out with AutoML project, designed to make AI build other AIs.

Neural networks have recently gained popularity and wide practical applications. However, to get good results with neural networks, it is critical to pick the right network topology, which has always been a difficult manual task.

Google's recent project promises to help solve this task automatically with a meta-AI which will design the topology for neural network architecture. Google, however, did not offer documentation or examples of how to use this new wonderful technology. We liked the idea and, among the first, came up with a practical implementation that other people can follow, using it as an example. This is similar in concept to AlphaGo, for instance.

Google's approach is based on the AI concept called Reinforcement Learning, meaning that the parent AI reviews the efficiency of the child AI and makes adjustments to the neural network architecture, such as adjusting the number of layers, weights, regularization methods, etc. to improve efficiency.

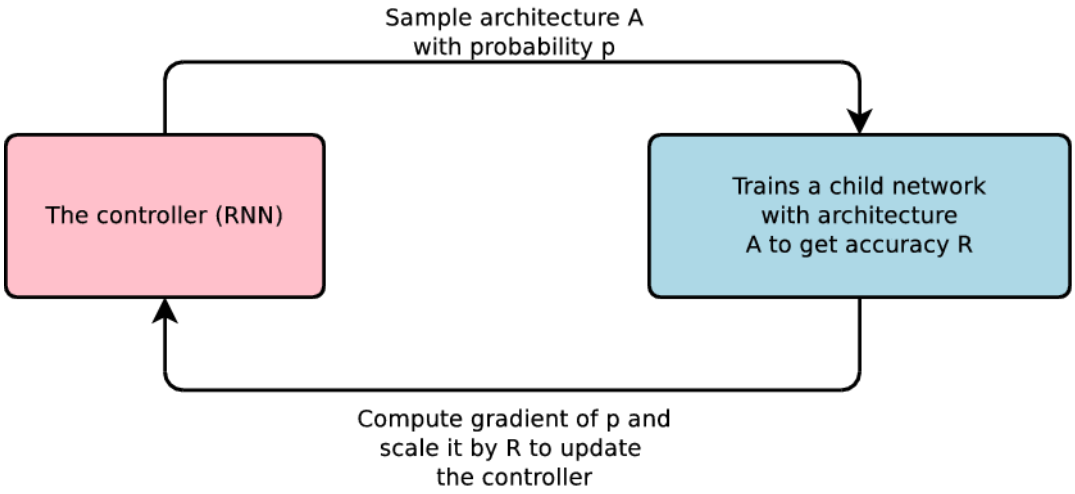


Image from Google Blog

The advantage of automation is the ability to eliminate guesswork from the manual neural network model design as well as significantly reducing the time required for each problem, since designing the neural network model is the most labor-intensive part of the task.

Although Google has recently open sourced an example of NASnet, how they found the architecture of NASnet is still unclear to most folks.

In addition, in our opinion, the name itself adds to the confusion with these technologies.

In this post, we will take a detailed look (with a step by step explanation) at implementing a simple model for neural architecture search with AutoML and reinforcement learning.

Note: To understand this post, you will need to have sufficient background understanding of the convolutional neural networks, recurrent neural networks, and reinforcement learning.

Links below will provide you with good background information:

- https://en.wikipedia.org/wiki/Reinforcement_learning
- https://www.cs.cmu.edu/afs/cs/project/jair/pub/volume4/kaelbling96a-html/node37.html
- http://karpathy.github.io/2015/05/21/rnn-effectiveness/
- https://en.wikipedia.org/wiki/Convolutional_neural_network

Neural Architecture Search (NAS) with Reinforcement Learning is a method for finding good neural networks architecture. For this post, we will try to find optimal architecture for Convolutional Neural Network (CNN) which recognizes handwritten digits.

For this implementation, we use TensorFlow 1.4, but if you want to try this at home, you can use any version after 1.1, since NASCell first became available in TensorFlow 1.1. It is important not to confuse AutoML and NAS.

Data and preprocessing.

To train the model we will use the MNIST database of handwritten digits, which has a training set of 55,000 examples and a test set of 10,000 examples.

The Model.

The network we are building in this exercise consists of a controller and the actual neural network that we are trying to optimize. The Controller is an rnn tensorflow with NAS cells and special reinforcement learning methods for training and getting rewards. We will define “rewards” as maximizing the accuracy of the desired neural network and train the Controller to improve this outcome. The controller should generate Actions to modify the architecture of CNN. **Specifically, Actions can modify filters: the dimensionality of the output space, kernel_size (integer, specifying the length of the 1D convolution window), pool_size (integer, representing the size of the pooling window) and dropout_rate per layer.**

Details of architecture search space

All convolutions employ Rectified Linear Units (ReLU) nonlinearity. Weights were initialized by the Xavier initialization algorithm.

Implementation.

For the Controller, we built a method for policy network based on NASCell. This network takes, as inputs, the current state (in this task, state and action are the same things) and maximum number of searching layers and outputs new Action to update the desired neural network. If for some reason, NASCell is not available, you can use any RNNCell.

https://gist.github.com/d0znpp/db1caaefd6c9ed5c72e65721fbd859bb

To allow hyperparameter tuning we put our code into a Reinforce class.

```
1  class Reinforce():
2      def __init__(self, sess, optimizer, policy_network, max_layers,
   global_step,
3                   division_rate=100.0, reg_param=0.001,
   discount_factor=0.99, exploration=0.3):
4          self.sess = sess
5          self.optimizer = optimizer
6          self.policy_network = policy_network
7          self.division_rate = division_rate
```

```
        self.reg_param = reg_param
        self.discount_factor=discount_factor
        self.max_layers = max_layers
        self.global_step = global_step

        self.reward_buffer = []
        self.state_buffer = []
```

sess and optimizer — TensorFlow session and optimizer, will be initialized separately.

- policy_network — Method initialized above.
- max_layers — The maximum number of layers
- division_rate — Normal distribution values of each neuron from -1.0 to 1.0.
- reg_param — Parameter for regularization.
- exploration — The probability of generating random action.

Of course, we also must create variables and placeholders, consisting of logits and gradients. To do this, let's write a method create_variables:

```
def create_variables(self):
    with tf.name_scope("model_inputs"):
        # raw state representation
        self.states = tf.placeholder(tf.float32, [None, self.max_layers*4], name="states")

        with tf.name_scope("predict_actions"):
            # initialize policy network
            with tf.variable_scope("policy_network"):
                self.policy_outputs = self.policy_network(self.states, self.max_layers)
                self.action_scores = tf.identity(self.policy_outputs, name="action_scores")
                self.predicted_action = tf.cast(tf.scalar_mul(
                    self.division_rate, self.action_scores), tf.int32, name="predicted_action")

                # regularization loss
                policy_network_variables = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES,

scope="policy_network")

                # compute loss and gradients
                with tf.name_scope("compute_gradients"):
```

```
                    # gradients for selecting action from policy
network
                    self.discounted_rewards =
tf.placeholder(tf.float32, (None,),

name="discounted_rewards")

                    with tf.variable_scope("policy_network",
reuse=True):
                        self.logprobs =
self.policy_network(self.states, self.max_layers)
                        # compute policy loss and regularization
loss
                        self.cross_entropy_loss =
tf.nn.softmax_cross_entropy_with_logits(
                            logits=self.logprobs,
labels=self.states)
                        self.pg_loss =
tf.reduce_mean(self.cross_entropy_loss)
                        self.reg_loss =
tf.reduce_sum([tf.reduce_sum(tf.square(x))
                                                        for x in
policy_network_variables])
                        self.loss = self.pg_loss + self.reg_param *
self.reg_loss

                        #compute gradients
                        self.gradients =
self.optimizer.compute_gradients(self.loss)

                        # compute policy gradients
                        for i, (grad, var) in
enumerate(self.gradients):
                            if grad is not None:
                                self.gradients[i] = (grad *
self.discounted_rewards, var)
                                # training update
                                with
tf.name_scope("train_policy_network"):
                                    # apply gradients to update
policy network
                                    self.train_op =
self.optimizer.apply_gradients(
                                        self.gradients)
```

After computing the initial gradients, we launch the gradient descent method. Now let's take a look at how reinforcement learning is implemented.

First, we can multiply gradient value to the discounted reward.

After defining the variables, we should initialize it in a TensorFlow graph in end of __init__:

```
self.create_variables()
var_lists = tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES)
self.sess.run(tf.variables_initializer(var_lists))
```

Every Action depends on the previous state, but sometimes, for more effective training, we can generate random actions to avoid local minimums.

In each cycle, our network will generate an Action, get rewards and after that, take a training step.

The implementation of the training step includes store_rollout and train_step methods below:

```
def store_rollout(self, state, reward):
        self.reward_buffer.append(reward)
        self.state_buffer.append(state[0])

def train_step(self, steps_count):
    states = np.array(self.state_buffer[-steps_count:])/self.division_rate
    rewars = self.reward_buffer[-steps_count:]
    _, ls = self.sess.run([self.train_op, self.loss],
                          {self.states: states, self.discounted_rewards: rewars})
    return ls
```

As mentioned above, we need to define rewards for each Action\State.

This is accomplished by generating a new CNN network with new architecture per Action, training it and assessing its accuracy. Since this process generates a lot of CNN networks, let's write a manager for it:

```
class NetManager():
    def __init__(self, num_input, num_classes, learning_rate, mnist,
                 max_step_per_action=5500, bathc_size=100,
dropout_rate=0.85):

        self.num_input = num_input
        self.num_classes = num_classes
        self.learning_rate = learning_rate
        self.mnist = mnist

        self.max_step_per_action = max_step_per_action
        self.bathc_size = bathc_size
        self.dropout_rate = dropout_rate #Dropout after dense layer
in CNN
```

Then we formed bathc with hyperparameters for every layer in "action" and we created cnn_drop_rate — list of dropout rates for every layer.

```
def get_reward(self, action, step, pre_acc):
    action = [action[0][0][x:x+4] for x in range(0, len(action[0]
[0]), 4)]
    cnn_drop_rate = [c[3] for c in action]

    with tf.Graph().as_default() as g:
        with g.container('experiment'+str(step)):
            model = CNN(self.num_input, self.num_classes, action)
            loss_op = tf.reduce_mean(model.loss)
            optimizer =
tf.train.AdamOptimizer(learning_rate=self.learning_rate)
            train_op = optimizer.minimize(loss_op)
```

Here we define a convolution neural model with CNN class. It can be any class that is able to generate the neural model by some action.

We created a separate container to avoid confusion in TF graph.

After creating a new CNN model, we can train it and get a reward.

```
with tf.Session() as train_sess:
    init = tf.global_variables_initializer()
    train_sess.run(init)

    for step in range(self.max_step_per_action):
```

```
        batch_x, batch_y =
self.mnist.train.next_batch(self.bathc_size)
        feed = {model.X: batch_x,
                model.Y: batch_y,
                model.dropout_keep_prob: self.dropout_rate,
                model.cnn_dropout_rates: cnn_drop_rate}
        _ = train_sess.run(train_op, feed_dict=feed)


        batch_x, batch_y = self.mnist.test.next_batch(10000)
        loss, acc = train_sess.run(
            [loss_op, model.accuracy],
            feed_dict={model.X: batch_x,
                       model.Y: batch_y,
                       model.dropout_keep_prob: 1.0,
                       model.cnn_dropout_rates:
[1.0]*len(cnn_drop_rate)})
        if acc - pre_acc <= 0.01:
            return acc, acc
        else:
            return 0.1, acc
```

As defined, the reward improves accuracy on all test datasets; for MNIST it is 10000 examples.

Now that we have everything in place, let's find the best architecture for MNIST. First, we will optimize the architecture for the number of layers. Let's set the maximum number of layers to 2. Of course, you can set this value to be higher, but every layer needs a lot of computing power.

```
def train(mnist, max_layers):
    sess = tf.Session()
    global_step = tf.Variable(0, trainable=False)
    starter_learning_rate = 0.1
    learning_rate = tf.train.exponential_decay(0.99, global_step,
500, 0.96, staircase=True)

    optimizer =
tf.train.RMSPropOptimizer(learning_rate=learning_rate)
    reinforce = Reinforce(sess, optimizer, policy_network,
args.max_layers, global_step)
    net_manager = NetManager(num_input=784, num_classes=10,
learning_rate=0.001, mnist=mnist)

    MAX_EPISODES = 250
```

```
    step = 0
    state = np.array([[10.0, 128.0, 1.0, 1.0]*max_layers],
dtype=np.float32)
    pre_acc = 0.0
    for i_episode in range(MAX_EPISODES):
        action = reinforce.get_action(state)
        print("current action:", action)
        if all(ai > 0 for ai in action[0][0]):
            reward, pre_acc = net_manager.get_reward(action, step,
pre_acc)
        else:
            reward = -1.0
        # In our sample action is equal state
        state = action[0]
        reinforce.store_rollout(state, reward)

        step += 1
    ls = reinforce.train_step(MAX_STEPS)
    log_str = "current time:  " +
str(datetime.datetime.now().time()) +
        " episode:  " + str(i_episode) + " loss:  " + str(ls) + "
last_state:  " +
        str(state) + " last_reward:  " + str(reward)
    print(log_str)

def main():
    max_layers = 3
    mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
    train(mnist, max_layers)

if __name__ == '__main__':
  main()
```

We couldn't be sure about what we should feed to our policy network. First, we tried to always feed in the array of 1.0 to our RNN per episode, but it yielded no results. Then we tried feeding every new state per episode and it resulted in a good architecture. We concluded that the first state can be any non-zero array, to expedite finding a suitable architecture we set the first state: [[10.0, 128.0, 1.0, 1.0]*args.max_layers]

We have updated the weights after every episode. Otherwise, our calculations would have been useless. That's why our "batch size" for reinforce = 1.

After 100 cycles, we get the following architecture:

- input layer : 784 nodes (MNIST images size)

- first convolution layer : 61x24
- first max-pooling layer: 60
- second convolution layer : 57x55
- second max-pooling layer: 59
- output layer : 10 nodes (number of class for MNIST)

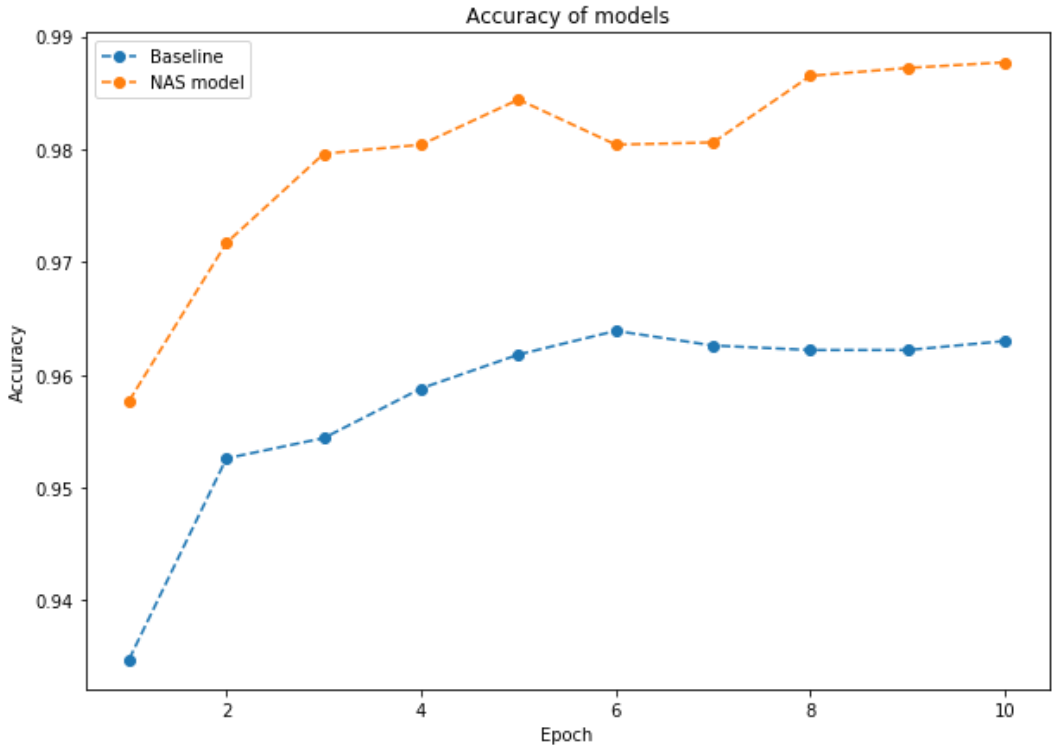
Measuring Results

Now that we've trained our "NAS model" on MNIST dataset, we should be able to compare the architecture our AI has created with the other architectures created manually. For comparable results we will use popular Convolutional Neural Network (CNN) architecture for MNIST [It's not the state-of-the-art architecture, but it's good for comparing]:

- input layer : 784 nodes (MNIST images size)
- first convolution layer : 5x32
- first max-pooling layer: 2
- second convolution layer : 5x64
- second max-pooling layer: 2
- output layer : 10 nodes (number of class for MNIST)

All weights were initialized by the Xavier algorithm.

We trained our models on 10 epochs and got of the accuracy of 0.9987 for our "NAS model", compared to 0.963 for the popular manually defined neural network architecture.



Conclusion

We have presented a code example of a simple implementation that automates the design of machine learning models and:

- doesn't require any human time to design
- actually delivered a better performance than a manually designed network

Going forward, we will continue working on careful analysis and testing of these machine-generated architectures to help refine our understanding of them. Naturally, if we search for more parameters using our model, we'll achieve better results for MNIST, but more importantly, this simple example illustrates how this approach can be applied to the problems that are much more complicated. We built this model using some assumptions which are quite difficult to justify if you notice any mistakes, please write in issues on GitHub.

The full code of the project is available on Github