TENSORFLOW TUTORIAL

Sanket Shanbhag [1]

ABSTRACT

This document describes some basic concepts on how to use tensorflow. Familiarity with machine learning concepts is assumed. We will be using Python3 along with tensorflow version 1.2.0.

INTRODUCTION

To do efficient numerical computing in Python, we typically use libraries like NumPy that do expensive operations such as matrix multiplication outside Python, using highly efficient code implemented in another language. Unfortunately, there can still be a lot of overhead from switching back to Python every operation. This overhead is especially bad if you want to run computations on GPUs or in a distributed manner, where there can be a high cost to transferring data.

TensorFlow also does its heavy lifting outside Python, but it takes things a step further to avoid this overhead. It does this by building a so called computational graph before-hand and performing all computations afterwards on this graph using highly efficient hardware specific instructions. Since GPU's are specifically designed for optimizing matrix multiplications, using TensorFlow allows us to harness the entire power of the GPU to speed up computations.

SETTING UP

Before we use Tensorflow, we must import it:

```
import tensorflow as tf
```

The best place to find more information on all the functions we will use is the API docs.

The central unit of data in TensorFlow is the tensor. A tensor can simply be thought of as a multidimensional array. A tensor's rank is its number of dimensions. Here are some examples of tensors:

```
3 # a rank 0 tensor; this is a scalar with shape []
[1, 2, 3] # a rank 1 tensor; this is a vector with shape [3]
[[1, 2, 3], [4, 5, 6]] # a rank 2 tensor; a matrix with shape [2, 3]
[[[1, 2, 3]], [[7, 8, 9]]] # a rank 3 tensor with shape [2, 1, 3]
```

1

To create a computational graph, we create nodes and then run a session on these nodes to generate the output. Each node takes zero or more tensors as inputs and produces a tensor as an output.

INPUTS AND SESSIONS

Constants

One type of node is a constant. Like all TensorFlow constants, it takes no inputs, and it outputs a value it stores internally.

```
# Use dtype to optionally specify a type
node = tf.constant(42.0, dtype=tf.float32)
```

At this stage node is a tensor object that, when evaluated will hold the value 3.0. To actually evaluate this node, we have to run the computational graph in a session.

```
# Start a new Tensorflow session
sess = tf.Session()
print(sess.run([node1])) # Prints 42
```

Placeholders

Constants are not that interesting, as they cannot be changed. To accept external inputs at run-time, we use placeholders.

```
a = tf.placeholder(tf.float32)
b = tf.placeholder(tf.float32)
adder_node = tf.add(a,b)
```

To give a value to the placeholder in the computational graph, use the `feed_dict` argument in the `run` function of the Session object to pass a python dictionary specifying the placeholders as key-value pairs.

```
print(sess.run(adder_node, feed_dict={a:18, b:24})) # Prints 42!
```

To assign placeholders of a higher rank, use the `shape` argument to specify the shape of the tensor. If any of the dimensions can be arbitary, for example when using arbitary number of training samples, you can use `None` instead.

```
# Takes a tensor of dimensions [None, 500]
a = tf.placeholder(tf.float32, shape=[None, 500])
# Tensor of dimensions [40, 50, 100]
b = tf.placeholder(tf.float32, shape=[40, 50, 100])
```

Variables

Variables allow us to add trainable parameters to a graph. A variable maintains state in the graph across calls to `run()`. You add a variable to the graph by constructing an instance of the class Variable. They are constructed with a type and initial value:

```
# Create some variables.
W = tf.Variable([.3], dtype=tf.float32)
# tf.random_normal(): Outputs random values from a normal
# distribution.
weights = tf.Variable(tf.random_normal([784, 200], stddev=0.35),
                      name="weights")
```

```
# tf.zeros(): Creates a tensor with all elements set to zero
biases = tf.Variable(tf.zeros([200]), name="biases")
```

Like `tf.random_normal()` and `tf.zeros()` shown above, TensorFlow provides a collection of ops that produce tensors often used for initialization from constants or random values.

Variables are not initialized when you call `tf.Variable`. To initialize all the variables in a TensorFlow program, you must explicitly call a special operation as follows:

```
sess.run(tf.global_variables_initializer())
```

SIMPLE LINEAR MODEL

In this section, we will create a simple model for classification on the easily available MNIST data-set and train it. Some familiarity with neural networks, activation functions and backpropagation are prerequisites for this section.

Downloading and Formatting Data

We will be using a single hidden layer of 500 nodes and an output layer of 10 classes.

```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("./data/", one_hot = True)
```

This will download the MNIST data-set into a `data` folder in the current working directory. The data-set has been loaded as so-called One-Hot encoding. This means the labels have been converted from a single number to a vector whose length equals the number of possible classes. All elements of the vector are zero except for the `i`'th element which is one and means the class is `i`. For example, if the class value is 4, then it's one-hot encoded vector will be:

```
# 4th value is 1, everything else is 0
[0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
```

Using your own dataset

Using the pandas library we can import our own dataset as follows:

```
import pandas as pd

# Load in the data with `read_csv()`
mnist = pd.read_csv("http://archive.ics.uci.edu/ml/machine-learning-
    databases/optdigits/optdigits.tra", header=None)
```

Setting up Hyperparamaters

```
# Hyperparameters
# 10 digits to identify
n_classes = 10
# Train 100 images at a time to avoid using up too much RAM
batch_size = 100
# We will be using a single hidden layer of 500 neurons
```

```
n_nodes_hl1 = 500
# The images are 28x28 pixels each
img_size_flat = 28 * 28
```

Setting up a saver object

To save your trained model in case of a crash, we can use tensorflow Saver objects as shown:

```
# Declare a saver object
saver = tf.train.Saver()
# Directory to save checkpoints to
save_dir = 'checkpoints/'
# Create the directory if it doesn't exist
if not os.path.exists(save_dir):
    os.makedirs(save_dir)
# Prefix to attach with which checkpoints will be saved
save_path = os.path.join(save_dir, 'saved')
```

To restore the saved checkpoints, after declaring your model, simply call the following function:

```
# Restore saved model from save_path
saver.restore(sess=session,save_path=save_path)
```

This will allow you to predict on a saved model.

Setting up the model

To input images to our model, we will first flatten them into a single dimensional vector of length `img_size_flat`, and then feed this vector into our model. To do this, we will define a few placeholders:

```
# Input placeholder for flattened images
X = tf.placeholder('float', shape=[None, img_size_flat])
# Input vector for true class labels.
y = tf.placeholder('float')
```

Now we define the weights and biases for the hidden layer and output layer:

```
# Hidden layer
hidden_layer_weights = tf.Variable(tf.random_normal([img_size_flat,
                                   n_nodes_hl1]))
hidden_layer_biases = tf.Variable(tf.random_normal([n_nodes_hl1]))
# Output layer
output_layer_weights = tf.Variable(tf.random_normal([n_nodes_hl1,
                                   n_classes]))
output_layer_biases = tf.Variable(tf.random_normal([n_classes]))
```

Connecting the Layers

We now define the relationsip between our layers by matrix multiplying the data with the weights and adding the biases. We use the ReLu activation function. Other activation functions are also available in tensorflow and can be found here.

```
l1 = tf.add(tf.matmul(X,hidden_layer_weights),
                      hidden_layer_biases)
# Using ReLu activation
```

```
l1 = tf.nn.relu(l1)
# You can use the + operator instead of using the tf.add() function
output = tf.matmul(l1,output_layer_weights) + output_layer_biases
```

Training

We now define a function to train this network. We will use the GradientDescent Optimizer to reduce the mean squared error (MSE) loss function. Other optimizers in tensorflow can be found here.

```
def train_neural_network(x):
    prediction = x
    # Softmax function
    smx = tf.nn.softmax_cross_entropy_with_logits(logits=prediction,
                                                   labels=y)
    # Define the cost function
    cost = tf.reduce_mean(smx)
    # Use the GradientDescentOptimizer with a learning rate of 0.5
    # to minimize the cost
    optimizer = tf.train.GradientDescentOptimizer(0.5).\
                minimize(cost)
    # Number of epochs to train for
    hm_epochs = 20

    # Start a new tensorflow session
    with tf.Session() as sess:
        # Initialize the global variables
        sess.run(tf.global_variables_initializer())
        # Run a loop for the total number of epochs
        for epoch in range(hm_epochs):
            epoch_loss = 0
            # Divide the data set into batches of size batch_size
            batchquot = int(mnist.train.num_examples/batch_size)

            for _ in range(batchquot):
                # Get a batch of images and labels
                xt, yt = mnist.train.next_batch(batch_size)

                # Run the optimizer to minimize the
                # cost on the batch
                _, c = sess.run([optimizer, cost],
                                feed_dict={X:xt, y:yt})
                # Add the cost to our epoch loss
                epoch_loss += c

                # Save this epoch so we can continue
                # in case of crash
                saver.save(sess=session,
                           save_path=save_path)

            print('Epoch', epoch+1, 'completed out of',hm_epochs,
                  'loss:',epoch_loss)

        # Caculate and print accuracy
```

```
        correct = tf.equal(tf.argmax(prediction, 1),
                           tf.argmax(y, 1))
        accuracy = tf.reduce_mean(tf.cast(correct, 'float'))
        print('Accuracy:',accuracy.eval({X:mnist.test.images,
                                         y:mnist.test.labels}))
```

To begin training, we simply pass our model to the function:

```
train_neural_network(output)
```

Result

Running the above network gives us the following results:

```
Epoch 1 completed out of 20 loss: 4112.72234179
Epoch 2 completed out of 20 loss: 292.313294291
Epoch 3 completed out of 20 loss: 179.839554987
Epoch 4 completed out of 20 loss: 126.560542699
Epoch 5 completed out of 20 loss: 97.231446553
Epoch 6 completed out of 20 loss: 76.5352744549
Epoch 7 completed out of 20 loss: 64.625726237
Epoch 8 completed out of 20 loss: 54.0737959952
Epoch 9 completed out of 20 loss: 47.0280316649
Epoch 10 completed out of 20 loss: 41.1886193645
Epoch 11 completed out of 20 loss: 36.654014512
Epoch 12 completed out of 20 loss: 32.950176964
Epoch 13 completed out of 20 loss: 29.3913420243
Epoch 14 completed out of 20 loss: 26.9070334777
Epoch 15 completed out of 20 loss: 24.5256814114
Epoch 16 completed out of 20 loss: 21.9067392419
Epoch 17 completed out of 20 loss: 20.2392465728
Epoch 18 completed out of 20 loss: 18.6161083714
Epoch 19 completed out of 20 loss: 17.0158358993
Epoch 20 completed out of 20 loss: 15.9119512606
Accuracy: 0.943
```

CONCLUSION

We learnt the basics of how to use tensorflow. It is important to get hands-on experience with TensorFlow in order to learn how to use it properly. Try changing the learning rate, batch size and the optimizers used and see how they impact speed and performance. Also try to increase the number of layers to see how it affects the accuracy.