CHAPTER 7

CNN in TensorFlow

This chapter will demonstrate how to use TensorFlow to build a CNN model. A CNN model can help you build an image classifier that can predict/classify the images. In general, you create some layers in the model architecture with initial values of weight and bias. Then you tune weight and bias with the help of a training data set. There is another approach that involves using a pretrained model such as InceptionV3 to classify the images. You can use this transfer learning approach where you add some layers (whose parameter s are trained) on top of layers of pretrained models (with parameter values intact) to make very powerful classifiers.

In this chapter, I will use TensorFlow to show how to develop a convolution network for various computer vision applications. It is easier to express a CNN architecture as a graph of data flows.

Why TensorFlow for CNN Models?

In TensorFlow, images can be represented as three-dimensional arrays or tensors of shape (height, width and channels). TensorFlow provides the flexibility to quickly iterate, allows you to train models faster, and enables you to run more experiments. When taking TensorFlow models to production, you can run them on large-scale GPUs and TPUs.

N. K. Manaswi, *Deep Learning with Applications Using Python*,
https://doi.org/10.1007/978-1-4842-3516-4_7

TensorFlow Code for Building an Image Classifier for MNIST Data

In this section, I'll take you through an example to understand how to implement a CNN in TensorFlow.

The following code imports MNIST data sets with 28×28 grayscale images of digits from the TensorFlow contrib package and loads all the required libraries. Here, the aim is to build the classifier to predict the digit given in the image.

```
from tensorflow.contrib.learn.python.learn.datasets.mnist
import read_data_sets
from tensorflow.python.framework import ops
import tensorflow as tf
import numpy as np
```

You then start a graph session.

```
# Start a graph session
sess = tf.Session()
```

You load the MNIST data and create the train and test sets.

```
# Load data
from keras.datasets import mnist
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

You then normalize the train and test set features.

```
# Z- score  or Gaussian Normalization
X_train = X_train - np.mean(X_train) / X_train.std()
X_test = X_test - np.mean(X_test) / X_test.std()
```

As this is a multiclass classification problem, it is always better to use the one-hot encoding of the output class values.

```
# Convert labels into one-hot encoded vectors
num_class = 10
train_labels = tf.one_hot(y_train, num_class)
test_labels = tf.one_hot(y_test, num_class)
```

Let's set the model parameters now as these images are grayscale. Hence, the depth of image (channel) is 1.

```
# Set model parameters
batch_size = 784
samples =500
learning_rate = 0.03
img_width = X_train[0].shape[0]
img_height = X_train[0].shape[1]
target_size = max(train_labels) + 1
num_channels = 1 # greyscale = 1 channel
epoch = 200
no_channels = 1
conv1_features = 30
filt1_features = 5
conv2_features = 15
filt2_features = 3
max_pool_size1 = 2 # NxN window for 1st max pool layer
max_pool_size2 = 2 # NxN window for 2nd max pool layer
fully_connected_size1 = 150
```

Let's declare the placeholders for the model. The input data features, target variable, and batch sizes can be changed for the training and evaluation sets.

```
# Declare model placeholders
x_input_shape = (batch_size, img_width, img_height, num_channels)
x_input = tf.placeholder(tf.float32, shape=x_input_shape)
y_target = tf.placeholder(tf.int32, shape=(batch_size))
eval_input_shape = (samples, img_width, img_height, num_channels)
eval_input = tf.placeholder(tf.float32, shape=eval_input_shape)
eval_target = tf.placeholder(tf.int32, shape=(samples))
```

Let's declare the model variables' weight and bias values for input and hidden layer's neurons.

```
# Declare model variables
W1 = tf.Variable(tf.random_normal([filt1_features,
filt1_features, no_channels, conv1_features]))
b1 = tf.Variable(tf.ones([conv1_features]))
W2 = tf.Variable(tf.random_normal([filt2_features,
filt2_features, conv1_features, conv2_features]))
b2 = tf.Variable(tf.ones([conv2_features]))
```

Let's declare the model variables for fully connected layers and define the weights and bias for these last 2 layers.

```
# Declare model variables for fully connected layers
resulting_width = img_width // (max_pool_size1 * max_pool_size2)
resulting_height = img_height // (max_pool_size1 * max_pool_size2)
full1_input_size = resulting_width * resulting_height * conv2_
features
W3 = tf.Variable(tf.truncated_normal([full1_input_size,
fully_connected_size1], stddev=0.1, dtype=tf.float32))
```

```
b3 = tf.Variable(tf.truncated_normal([fully_connected_size1],
stddev=0.1, dtype=tf.float32))
W_out = tf.Variable(tf.truncated_normal([fully_connected_size1,
target_size], stddev=0.1, dtype=tf.float32))
b_out = tf.Variable(tf.truncated_normal([target_size],
stddev=0.1, dtype=tf.float32))
```

Let's create a helper function to define the convolutional and max pooling layers.

```
# Define helper functions for the convolution and maxpool layers:
def conv_layer(x, W, b):
    conv = tf.nn.conv2d(x, W, strides=[1, 1, 1, 1],
    padding='SAME')
    conv_with_b = tf.nn.bias_add(conv, b)
    conv_out = tf.nn.relu(conv_with_b)
    return conv_out
def maxpool_layer(conv, k=2):
    return tf.nn.max_pool(conv, ksize=[1, k, k, 1],
    strides=[1, k, k, 1], padding='SAME')
```

A neural network model is defined with two hidden layers and two fully connected layers. A rectified linear unit is used as the activation function for the hidden layers and the final output layers.

```
# Initialize Model Operations
def my_conv_net(input_data):
    # First Conv-ReLU-MaxPool Layer
    conv_out1 = conv_layer(input_data, W1, b1)
    maxpool_out1 = maxpool_layer(conv_out1)
```

```
    # Second Conv-ReLU-MaxPool Layer
    conv_out2 = conv_layer(maxpool_out1, W2, b2)
    maxpool_out2 = maxpool_layer(conv_out2)

    # Transform Output into a 1xN layer for next fully
    connected layer
    final_conv_shape = maxpool_out2.get_shape().as_list()
    final_shape = final_conv_shape[1] * final_conv_shape[2] *
    final_conv_shape[3]
    flat_output = tf.reshape(maxpool_out2, [final_conv_shape[0],
    final_shape])

    # First Fully Connected Layer
    fully_connected1 = tf.nn.relu(tf.add(tf.matmul(flat_output,
    W3), b3))
    # Second Fully Connected Layer
    final_model_output = tf.add(tf.matmul(fully_connected1,
    W_out), b_out)

    return(final_model_output)

model_output = my_conv_net(x_input)
test_model_output = my_conv_net(eval_input)
```

You will use a softmax cross entropy function (tends to work better for multiclass classification) to define the loss that operates on logits.

```
# Declare Loss Function (softmax cross entropy)
loss = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_
logits(logits=model_output, labels=y_target))
```

Let's define the train and test sets' prediction function.

```
# Create a prediction function
prediction = tf.nn.softmax(model_output)
test_prediction = tf.nn.softmax(test_model_output)
```

To determine the model accuracy on each batch, let's define the accuracy function.

```
# Create accuracy function
def get_accuracy(logits, targets):
    batch_predictions = np.argmax(logits, axis=1)
    num_correct = np.sum(np.equal(batch_predictions, targets))
    return(100. * num_correct/batch_predictions.shape[0])
```

Let's declare the training step and define the optimizer function.

```
# Create an optimizer
my_optimizer = tf.train.AdamOptimizer(learning_rate, 0.9)
train_step = my_optimizer.minimize(loss)
```

Let's initialize all the model variables declared earlier.

```
# Initialize Variables
varInit = tf.global_variables_initializer()
sess.run(varInit)
```

Let's start training the model and loop randomly through the batches of data. You want to evaluate the model on the train and test set batches and record the loss and accuracy.

```
# Start training loop
train_loss = []
train_acc = []
test_acc = []
for i in range(epoch):
    random_index = np.random.choice(len(X_train), size=batch_size)
    random_x = X_train[random_index]
    random_x = np.expand_dims(random_x, 3)
    random_y = train_labels[random_index]

    train_dict = {x_input: random_x, y_target: random_y}
```

```
    sess.run(train_step, feed_dict=train_dict)
    temp_train_loss, temp_train_preds = sess.run([loss,
    prediction], feed_dict=train_dict)
    temp_train_acc = get_accuracy(temp_train_preds, random_y)

    eval_index = np.random.choice(len(X_test),
    size=evaluation_size)
    eval_x = X_test[eval_index]
    eval_x = np.expand_dims(eval_x, 3)
    eval_y = test_labels[eval_index]
    test_dict = {eval_input: eval_x, eval_target: eval_y}
    test_preds = sess.run(test_prediction, feed_dict=test_dict)
    temp_test_acc = get_accuracy(test_preds, eval_y)
```

The results of the model are recorded in the following format and printed in the output:

```
# Record and print results
train_loss.append(temp_train_loss)
train_acc.append(temp_train_acc)
test_acc.append(temp_test_acc)
print('Epoch # {}. Train Loss: {:.2f}. Train Acc : {:.2f} .
temp_test_acc : {:.2f}'.format(i+1,temp_train_loss,
temp_train_acc,temp_test_acc))
```

Using a High-Level API for Building CNN Models

TFLearn, TensorLayer, tflayers, TF-Slim, tf.contrib.learn, Pretty Tensor, keras, and Sonnet are high-level TensorFlow APIs. If you use any of these high-level APIs, you can build CNN models in a few lines of code. So, you can explore any of these APIs for working smartly.