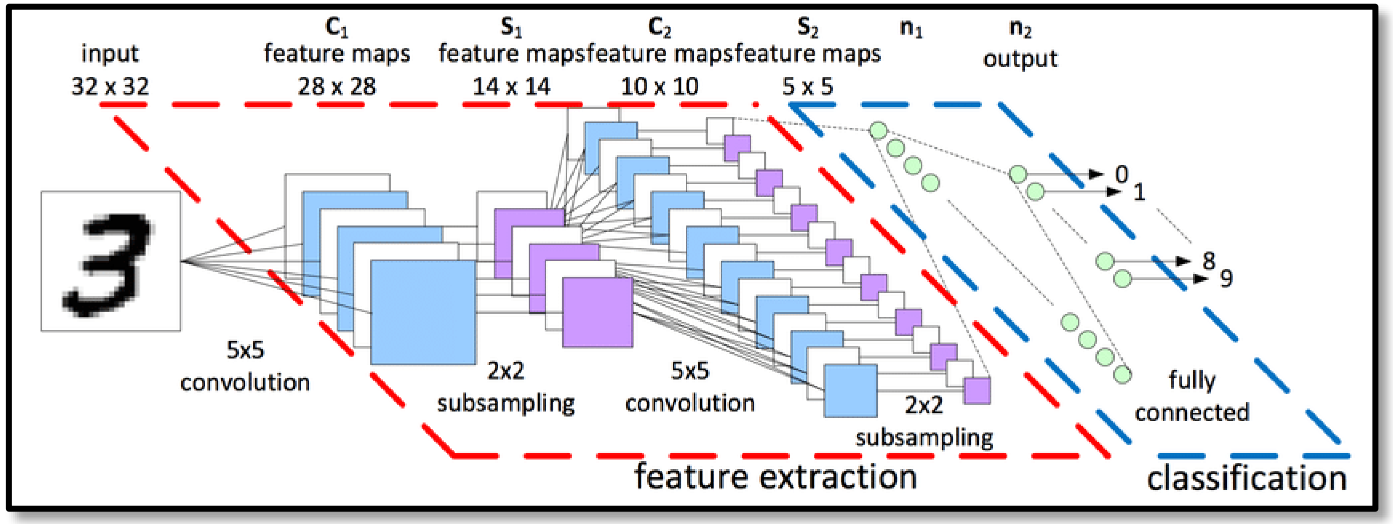
CNN Examples Using TensorFlow

CNN Examples Using TensorFlow

Build a CNN using TensorFlow

In the previous presentation, we built two simple CNN architectures using the Keras Sequential API. Utilizing Keras simplifies the process of defining and training a model. However, learning to build a CNN model using TensorFlow will provide us with a deeper understanding of what is happening under the hood.

In this presentation, we will build a very simple model using TensorFlow and train it on the MNIST dataset. The main goal here is to become familiar with building CNN models in TensorFlow, as it offers the flexibility to quickly iterate, allows us to train models faster, and enables us to conduct more experiments.

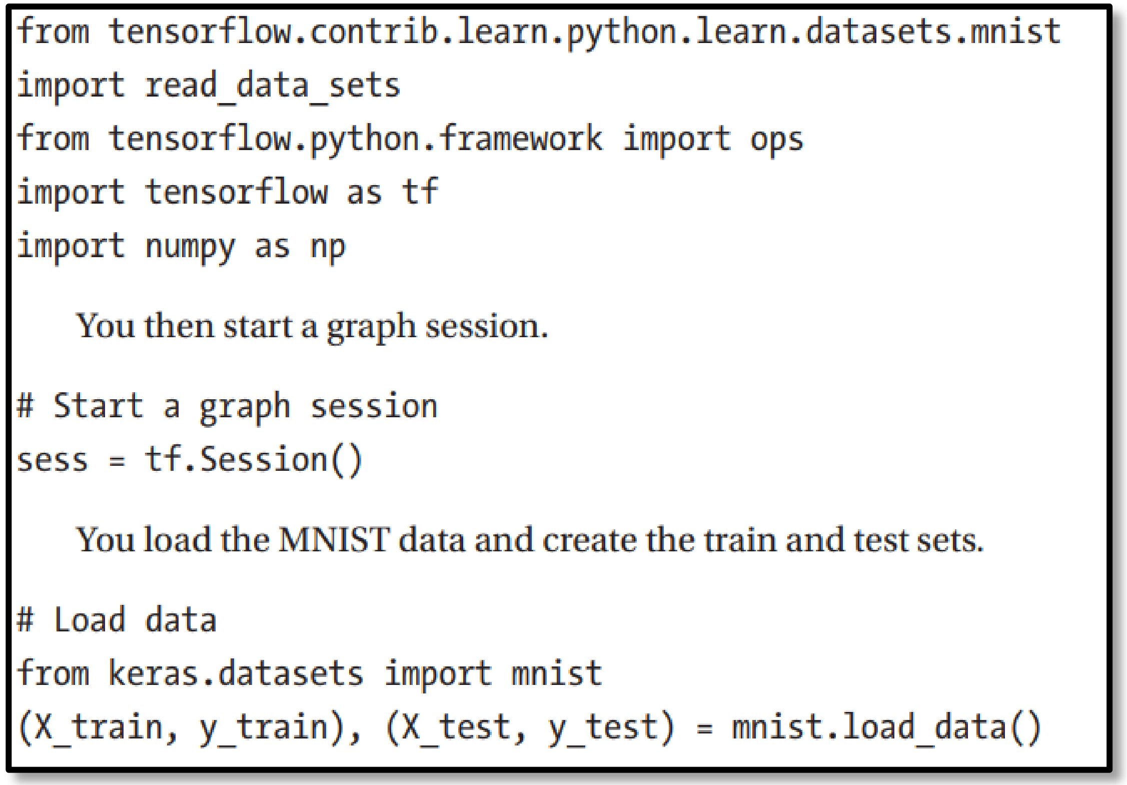


Source: www.researchgate.net

CNN Examples Using TensorFlow

Import modules and the dataset

Like every other deep learning project, we need to import the modules we are going to utilize. In this example, we will use TensorFlow to build the model. So, we start a graph session. We will also import NumPy to handle numerical operations efficiently.
Then, we load the dataset in the project. We will use the MNIST dataset, which is a simple dataset of 28 x 28 images of handwritten numbers. This dataset is readily available in TensorFlow, making it convenient for our example.
Finally, we split the dataset into training and testing sets to evaluate the performance of our model.



```
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()
```

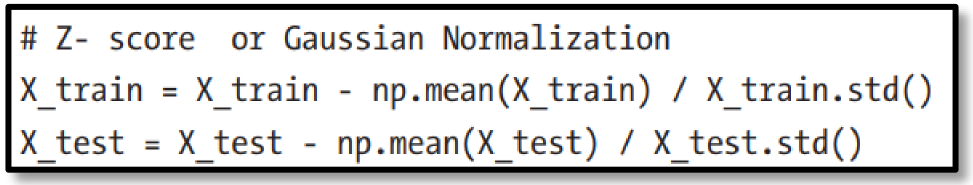
You load the MNIST data and create the train and test sets.

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

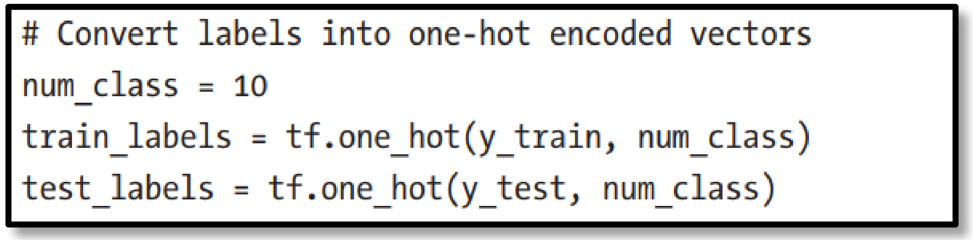
CNN Examples Using TensorFlow

Preprocess the images

In TensorFlow, images can be represented as three-dimensional tensors. We are going to perform two simple preprocessing steps. First, we normalize the inputs. To do so, we subtract the mean from the inputs and then divide them by the standard deviation. This normalization helps improve the convergence of the model during training.
Next, we will solve a multi-class classification problem, so we need to apply one-hot encoding to prepare the labels. This encoding transforms the labels into a binary matrix representation, allowing the model to understand the different classes more effectively.



```
# 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()
```



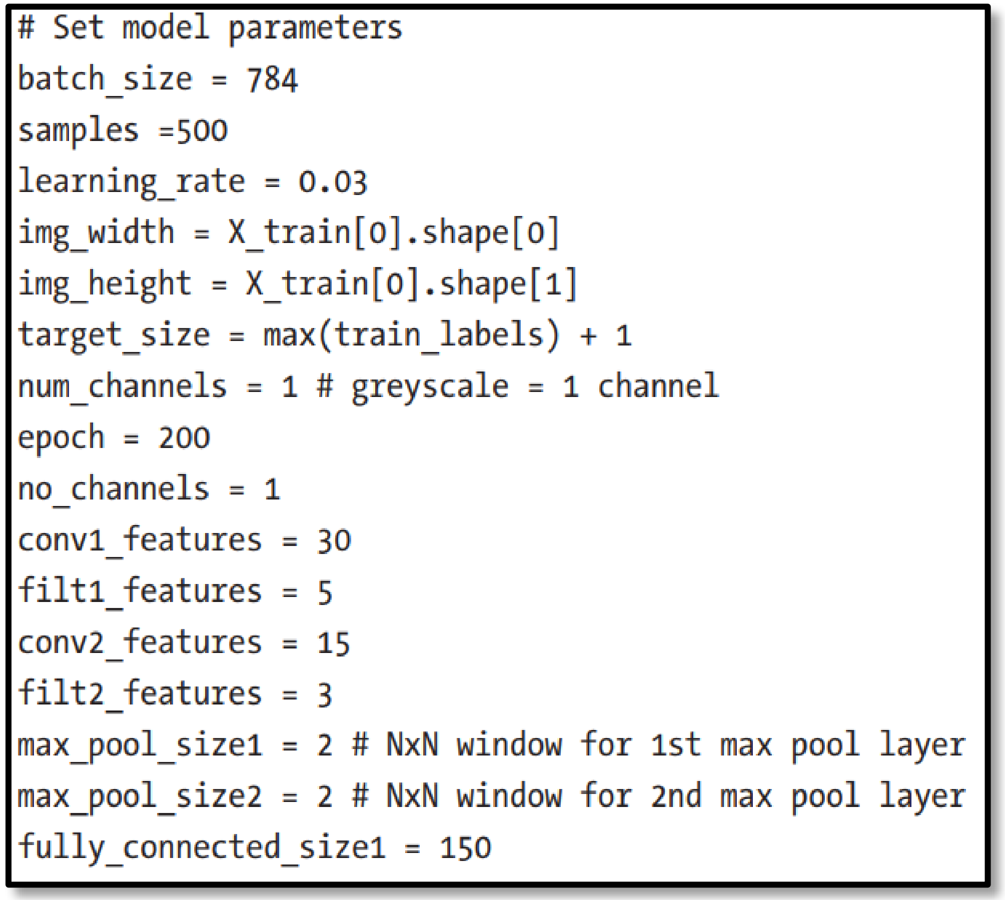
```
# Convert labels into one-hot encoded vectors
num_class = 10
train_labels = tf.one_hot(y_train, num_class)
test_labels = tf.one_hot(y_test, num_class)
```

CNN Examples Using TensorFlow

Initialize the hyperparameters

In this section, we set the batch size, number of samples, learning rate, image width and height, number of classes, and the number of epochs for training our model. Then, we will initialize the hyperparameters of our model. These hyperparameters will be configured with respect to the architecture we have in mind.

In this example, we are going to use a model with two convolutional layers, each followed by a max pooling layer. The first convolutional layer will have 30 filters of size 5 x 5, and the second convolutional layer will have 15 filters of size 3 x 3. Finally, we will add a fully connected layer with 150 neurons.

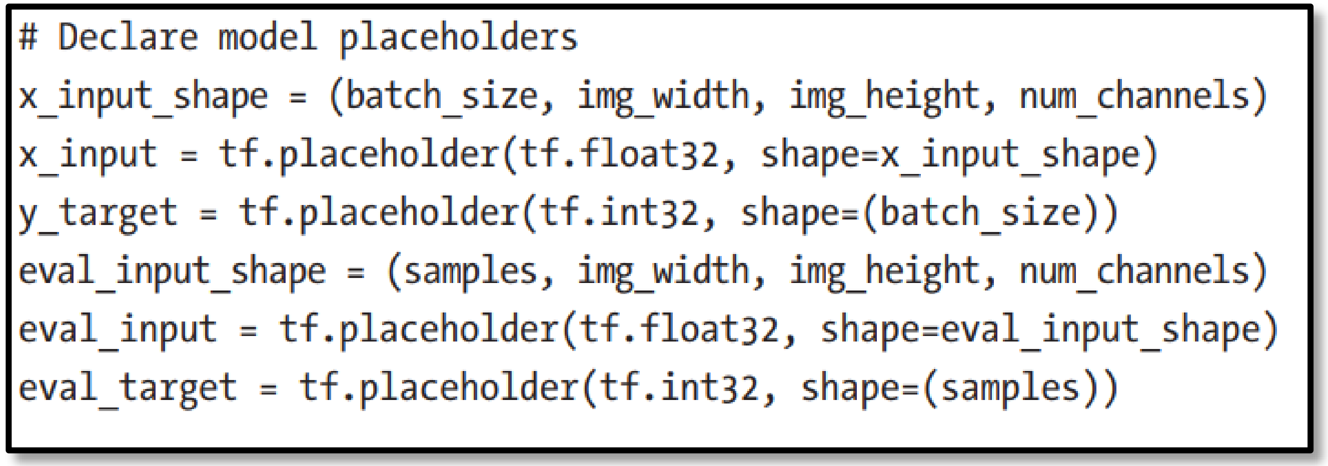


```
# Set model parameters
batch_size = 784
samples =500
learning_rate = 0.03
img_width = X_train[0].shape[0]
img_height = X_train[0].shape[1]
target_size = max(train_labels) + 1
num_channels = 1 # greyscale = 1 channel
epoch = 200
no_channels = 1
conv1_features = 30
filt1_features = 5
conv2_features = 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
```

CNN Examples Using TensorFlow

Declare the placeholders

we declare placeholders for the input and target data of our CNN model using TensorFlow.
Placeholders are special variables that allow us to feed data into the TensorFlow graph during execution. The x_input placeholder is defined with a shape which corresponds to a batch of images where each image has a specified width, height, and number of color channels (In this example 1 for grayscale images in the MNIST dataset).
The y_target placeholder, on the other hand, is designed to hold the labels for these images which indicates that each batch contains a corresponding label for each image.



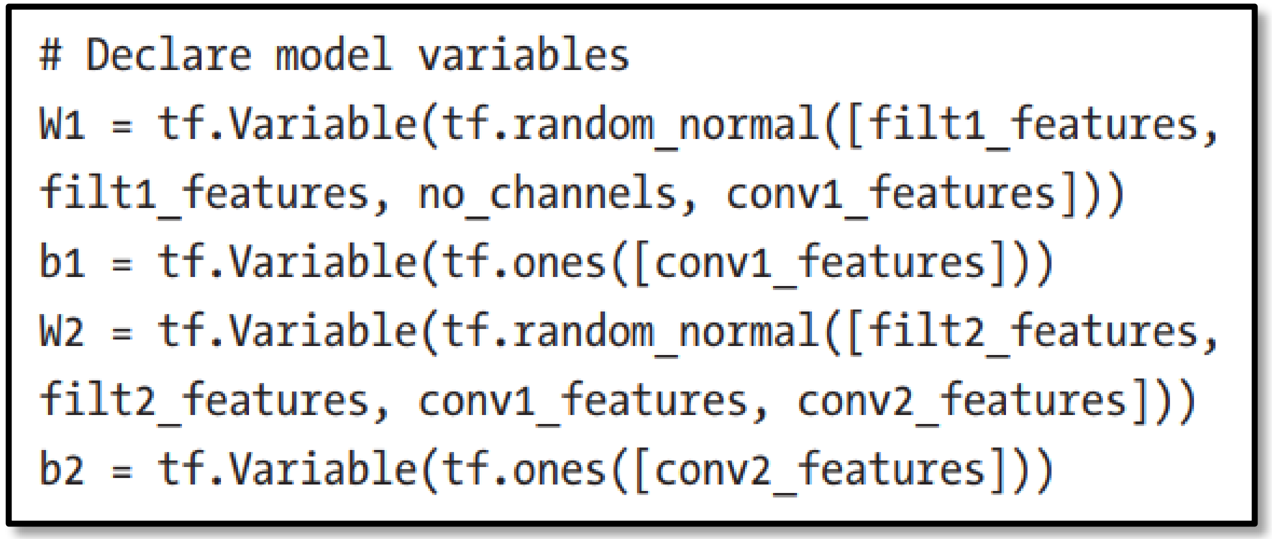
```
# Declare model placeholders
x_input_shape = (batch_size, img_width, img_height, num_channels)
x_input = tf.placeholder(tf.float32, shape=x_input_shape)
y_target = tf.placeholder(tf.int32, shape=(batch_size))
eval_input_shape = (samples, img_width, img_height, num_channels)
eval_input = tf.placeholder(tf.float32,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

CNN Examples Using TensorFlow

Initialize model parameters

Now, we will initialize the model parameters with random numbers. We will start with the weights and biases of the convolutional layers. To initialize the weights of each layer, we will use randomly generated numbers from a normal distribution, which helps in achieving better convergence during training.

We will set all the biases to one at the beginning. This approach can help prevent issues related to symmetry during the training process. It is important to remember that these parameters are going to be optimized when we train the model, allowing it to learn and adapt to the data effectively.



```
# 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]))
```

CNN Examples Using TensorFlow

Initialize model parameters

Next, we will focus on the weights and biases of the fully connected layers. First, we calculate the feature size after the two max pooling layers. Then, we will determine the number of features after the flatten layer by multiplying the height, width, and the number of filters in the second convolutional layer.

Now that we know the required number of weights and biases, we will initialize them with random values. This initialization is essential for effective training, as it helps the model learn complex patterns in the data.

We will use one fully connected layer followed by the output layer. Therefore, we need two sets of weights and biases: one for the fully connected layer and another for the output layer.

```
# 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))
```

CNN Examples Using TensorFlow

Build the model

Now that we have our parameters and hyperparameters initialized, we can start building the network. To facilitate defining our model in TensorFlow, we will use two auxiliary functions.

The first function will take the input, a set of weights and biases, and apply a convolutional layer on the input, returning the output. It places a convolutional layer between the input and the output.

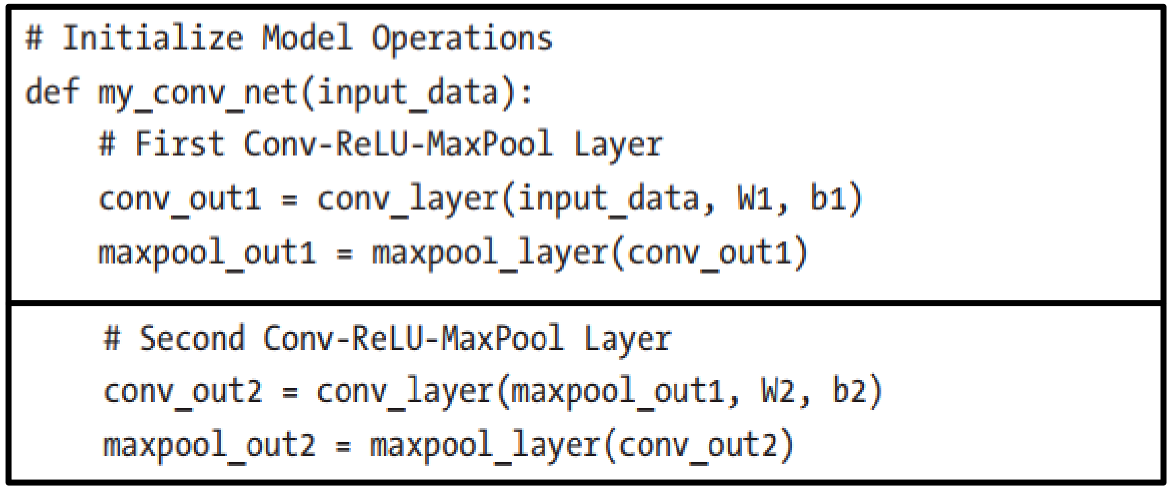
The second function is responsible for adding a max pooling layer between the input and the output. The default size of the max pooling kernel is set to 2.

```
# 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')
```

CNN Examples Using TensorFlow

Build the model

First, we will instantiate a convolutional layer using the set of weights and biases that we have previously defined. Next, we will add our max pooling layer using the auxiliary functions we have created.
After adding the first pair of convolutional and max pooling layers, we will repeat this process with another set of weights and biases to apply our second pair of convolutional and max pooling layers.
Now that our convolutional layers are fully defined and ready, we will proceed to add the fully connected layers on top of them.



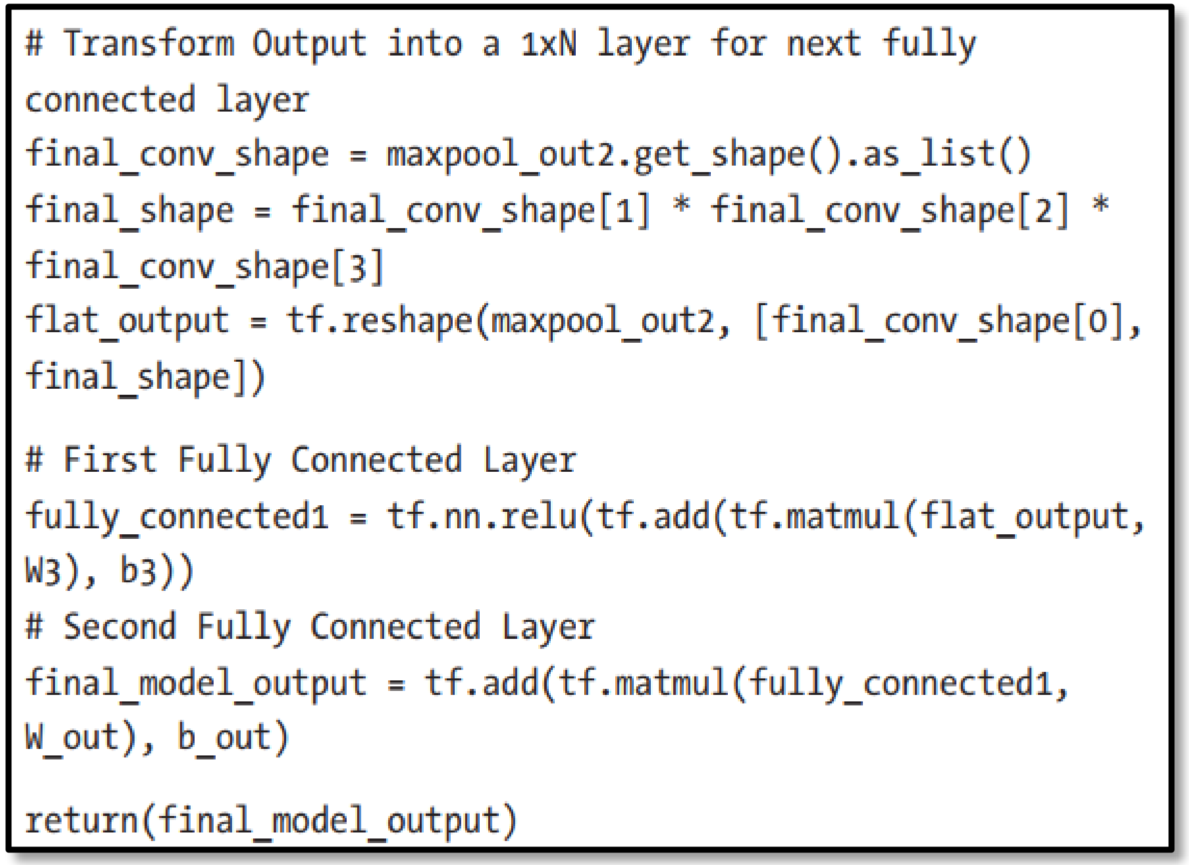
```
# Initialize Model Operations
def my_conv_net(input_data):
    # First Conv-ReLU-MaxPool Layer
    conv_out1 = conv_layer(input_data, W1, b1)
    maxpool_out1 = maxpool_layer(conv_out1)

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

CNN Examples Using TensorFlow

Build the model

In order to add the fully connected layers, we need to flatten the input. First, we will obtain the output shape of the convolutional part of the network. Then, we will multiply the dimensions of the output to calculate the total number of elements in the output. Finally, we will convert the initial output of the convolutional layers to a flattened layer with the corresponding size.

Now that we have flattened the output, we will add the fully connected layers on top. We will use ReLU as the activation function for these layers.

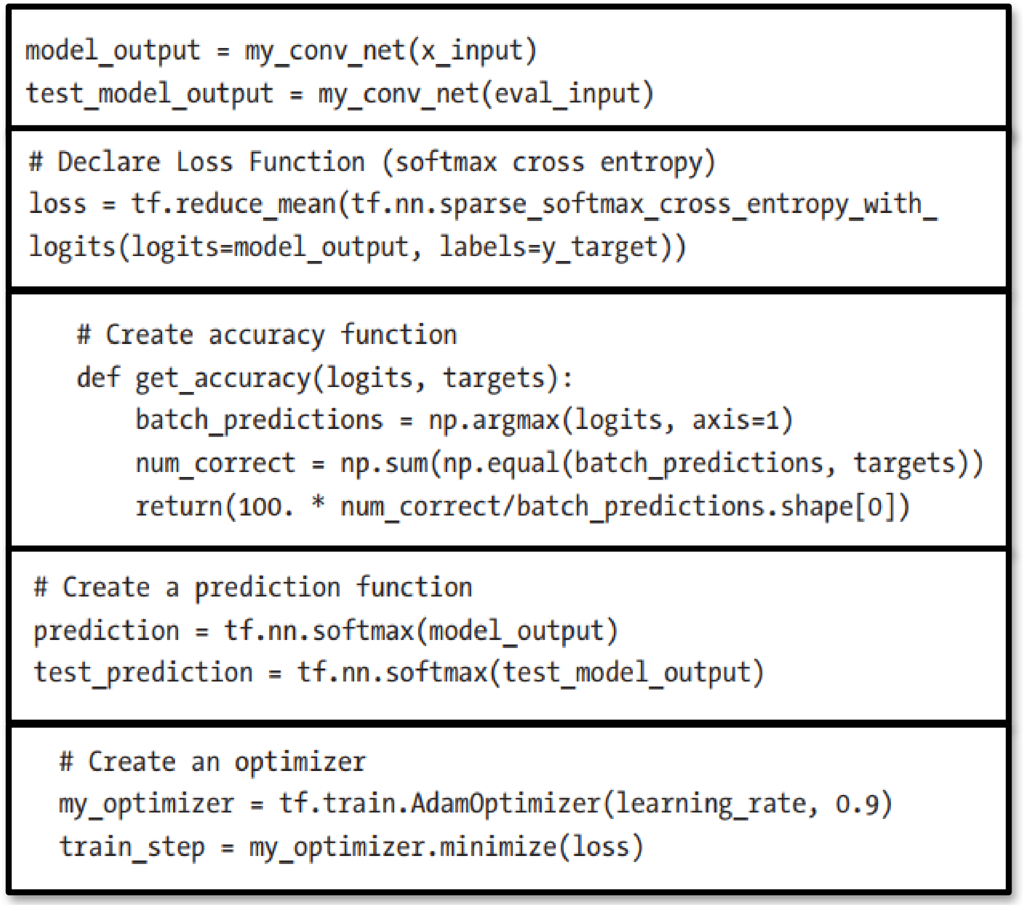


```
# Transform Output into a 1xN layer for next fully
connected layer
final_conv_shape = maxpool_out2.get_shape().as_list()
final_shape = final_conv_shape[1] * final_conv_shape[2] *
final_conv_shape[3]
flat_output = tf.reshape(maxpool_out2, [final_conv_shape[0],
final_shape])

# 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)
```

CNN Examples Using TensorFlow

Compile the model

Now that we have defined our model, we need to prepare it for training. In Keras, we specify the optimizer and loss function when compiling the model.
In this section, we will do the same. First, we will add a loss function. For our multi-class classification problem, we will use the Softmax function to convert output logits into probabilities.
Next, we will define our optimizer. We will use the Adam optimizer with a learning rate of 0.03. The goal of the optimizer is to minimize the loss function we declared earlier.
Finally, we will create an auxiliary function to calculate the model's accuracy during training, providing feedback on its performance.



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

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

```
# 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])
```

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

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

CNN Examples Using TensorFlow

Train the model

Now, we will begin training the model. First, we initialize the variables and run the session. Next, we declare lists to store the loss, training accuracy, and testing accuracy of the model during training.
After that, we define our main for loop, which is responsible for the training process. The first step is to select a random batch from our dataset. We will generate a list of random indexes, having already defined the size of our batch. Then, we will retrieve the corresponding inputs and outputs to create our batch. TensorFlow expects the input and labels in a dictionary format, so we will create this dictionary as the final step.

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

```
# 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}
```

CNN Examples Using TensorFlow

Train the model

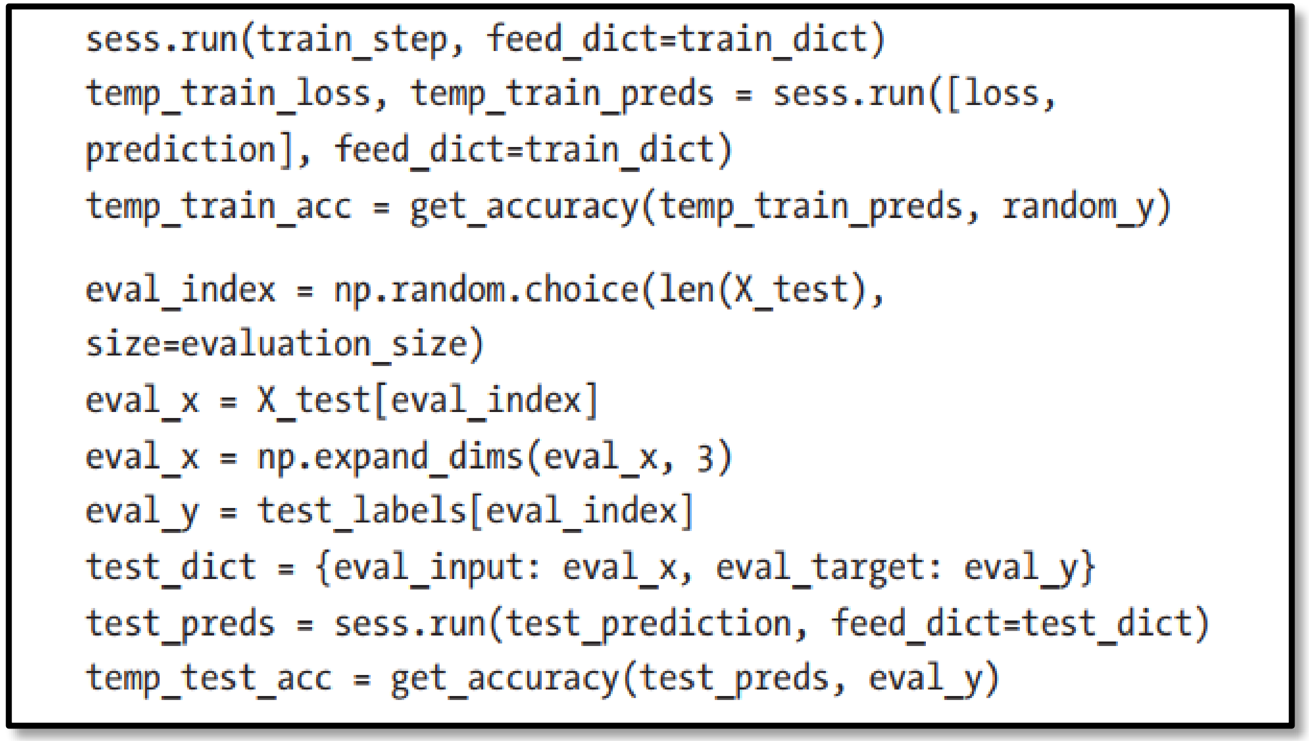
In this section of the code, we perform training and evaluation steps.

First, we execute the training operation which updates the model parameters based on the training data.

Next, we compute the training loss and predictions. We then calculate the training accuracy utilizing the auxiliary function we defined earlier.

For validation, we randomly select a subset of the test data. We prepare the validation inputs and labels, ensuring the input shape is correct.

Finally, we create a dictionary for the validation data and run the prediction operation followed by calculating the validation accuracy.



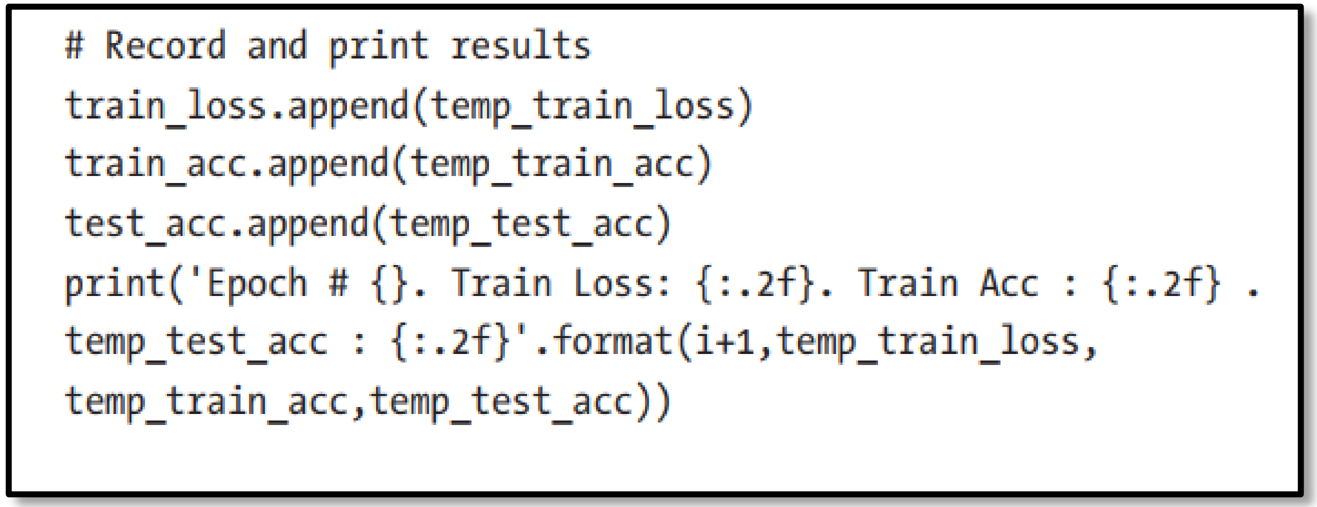
```
sess.run(train_step, feed_dict=train_dict)
temp_train_loss, temp_train_preds = sess.run([loss,
prediction], feed_dict=train_dict)
temp_train_acc = get_accuracy(temp_train_preds, random_y)

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)
```

CNN Examples Using TensorFlow

Train the model

We complete the for loop of training the model by appending the temporary loss and accuracies calculated during each iteration to the lists we previously declared. This allows us to track the model's performance over time.

Additionally, we print the temporary loss and accuracies to monitor the training process closely. This feedback is crucial for identifying potential issues, such as overshooting or overfitting, allowing us to make timely adjustments to the training strategy. By keeping a close eye on these metrics, we can ensure that the model converges effectively and achieves optimal performance.



```
# 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))
```