Building Models In TensorFlow

Building models

So far, we have become familiar with the basics of TensorFlow and how to construct simple tensors. Now, we are ready to advance our understanding by exploring how to build a model in TensorFlow.

We will begin with the most fundamental type of model, a linear model. This will provide us with a solid foundation for understanding more complex architectures. Following this, we will explore how to create a-

multi-layer perceptron (MLP), a more sophisticated model that allows us to tackle a wider range of problems, including multi-class classification.

Building models

Here are the 10 steps to building models in TensorFlow:

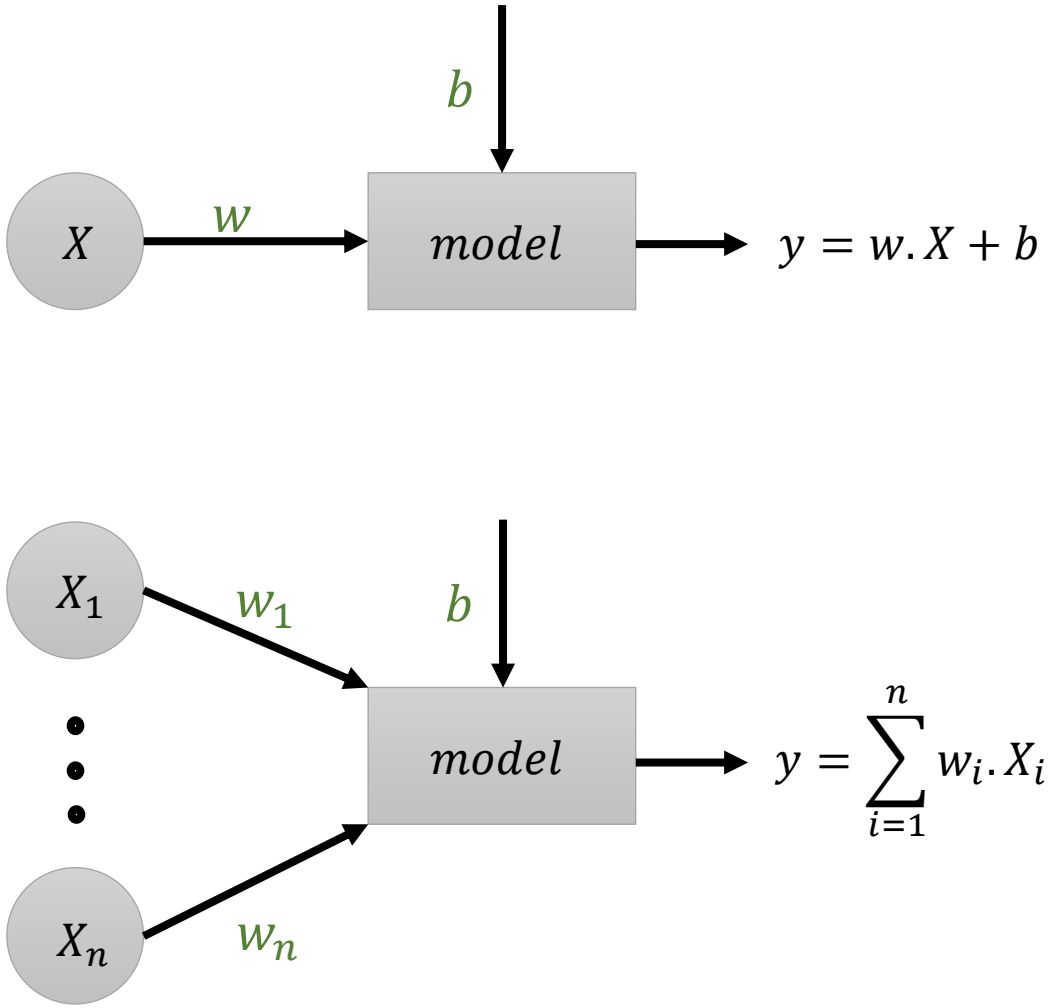
1. Load the data set that we will be using for the model.
2. Divide the data set into two parts: a training set for model training and a testing set.
3. Normalize the data if needed. This step can improve convergence during training.
4. Create placeholders that will hold the input data (predictors) and the output data (target).
5. Create variables (weights and biases) that will be tuned during training.
6. Specify the operations that will be performed in the model.
7. Declare the loss function and optimizer
8. Initialize all variables and create a session to run the graph.
9. Train the model by feeding it the training data.
10. Evaluate the model's performance using the test data.

Linear Regression model

In a linear regression model, the weight and bias are the parameters that should be optimized during the training process. Therefore, they will be declared as variables in the model.

It is also necessary to create a placeholder for the input variable X.

If the model has more than one input variable, the process is similar, but the dimensions of the placeholders and variables will change accordingly.



Linear Regression model

Now, let us examine an example of a linear regression model built using TensorFlow. In this example, we will utilize the Iris dataset from Seaborn, which contains five attributes.
Our objective is to predict the petal length when provided with the sepal length. Therefore, we will consider the sepal length as our input X, and the petal length as our output value y.
We will first import the necessary libraries.

```
import matplotlib as plt
import tensorflow as tf
from sklearn import datasets
import numpy as np
from sklearn.cross_validation import train_test_split
from matplotlib import pyplot
```

We will next load the Iris data set and extract the predictors and targets.

```
#1. Load the data
iris = datasets.load_iris()

predictors_vals = np.array([predictors[0] for predictors in iris.data])
target_vals = np.array([predictors[2] for predictors in iris.data])
```

Linear Regression model

Now, we will split the data set into training and testing data sets. Note that we are dedicating 80% of the data to the training data; that leaves 20% for the test data.

```
#2. Split the data into training and testing data sets
x_train, x_test, y_train, y_test = train_test_split(predictors_vals, target_vals,
                                                    test_size = 0.2, random_state = 12)
```

Next, we will create placeholders that will hold the predictors and the target.

```
#3. Normalize the data if neede
#4. Create placeholders that  will hold the predictors and the target

predictor = tf.placeholder(shape = [None, 1], dtype = tf.float32)
target = tf.placeholder(shape = [None, 1], dtype = tf.float32)
```

We will next create variables (weight and bias).

```
#5. Create variables (Weights and bias) that will be tuned during training

w = tf.Variable(tf.zeros(shape = [1, 1]))
b = tf.Variable(tf.ones(shape = [1, 1]))
```

Linear Regression model

Now, we will specify the operation of the model.

```
#6. Specify the operations that will be performed in the model

model_output = tf.add(tf.matmul(predictor, w), b)
```

Next, we will declare the model's loss function and optimizer.

```
#7 Declare the loss function and optimizer

loss = tf.reduce_mean(tf.abs(target - model_output))

opt = tf.train.GradientDecentOptimizer(0.01)

train_step = opt.minimize(loss)
```

We will next initialize the variables of the model and create a session.

```
#8. Initialize all variables and create a session to run the graph

sess = tf.Session()
init = tf.global_variables_initializer()

sess.run(init)
```

Linear Regression model

Now, we will train the model using the training data.

We will next evaluate the model using the test data.

```
#9. Train the model by feeding it the training data

loss_array = []
batch_size = 40

for i in range(200):
  rand_rows = np.random.randint(0, len(x_train) - 1, size = batch_size)
  batch_X = np.transpose([x_train[rand_rows]])
  batch_y = np.transpose([y_train[rand_rows]])
  sess.run(train_step, feed_dict = {predictor: batch_x, target: batch_y})
  batch_loss = sess.run(loss, feed_dict = {predictor: batch_x, target: batch_y})
  loss_array.append(batch_loss)

  if (i + 1) % 50 == 0:
    print('Step Number' + str(i + 1) + ' A = ' + str(sess.run(w)) + ' b = ' + str(sess.run(b)))
    print('L1 Loss = ' + str(batch_loss))

[slope] = sess.run(w)
[y_intercept] = sess.run(b)
```

```
#10. Evaluate the model's performance using the test data

loss_array = []
batch_size = 30
for i in range(100):
  rand_rows = np.random.randint(0, len(x_test) - 1, size = batch_size)
  batch_X = np.transpose([x_test[rand_rows]])
  batch_y = np.transpose([y_test[rand_rows]])
  sess.run(train_step, feed_dict = {predictor: batch_x, target: batch_y})
  batch_loss = sess.run(loss, feed_dict = {predictor: batch_x, target: batch_y})
  loss_array.append(batch_loss)

  if (i + 1) % 20 == 0:
    print('Step Number' + str(i + 1) + ' A = ' + str(sess.run(w)) + ' b = ' + str(sess.run(b)))
    print('L1 Loss = ' + str(batch_loss))

[slope] = sess.run(w)
[y_intercept] = sess.run(b)
```

Linear Regression model

Having evaluated our model, we will now proceed to plot the predicted line, along with the original data points to facilitate a visual representation of the results.

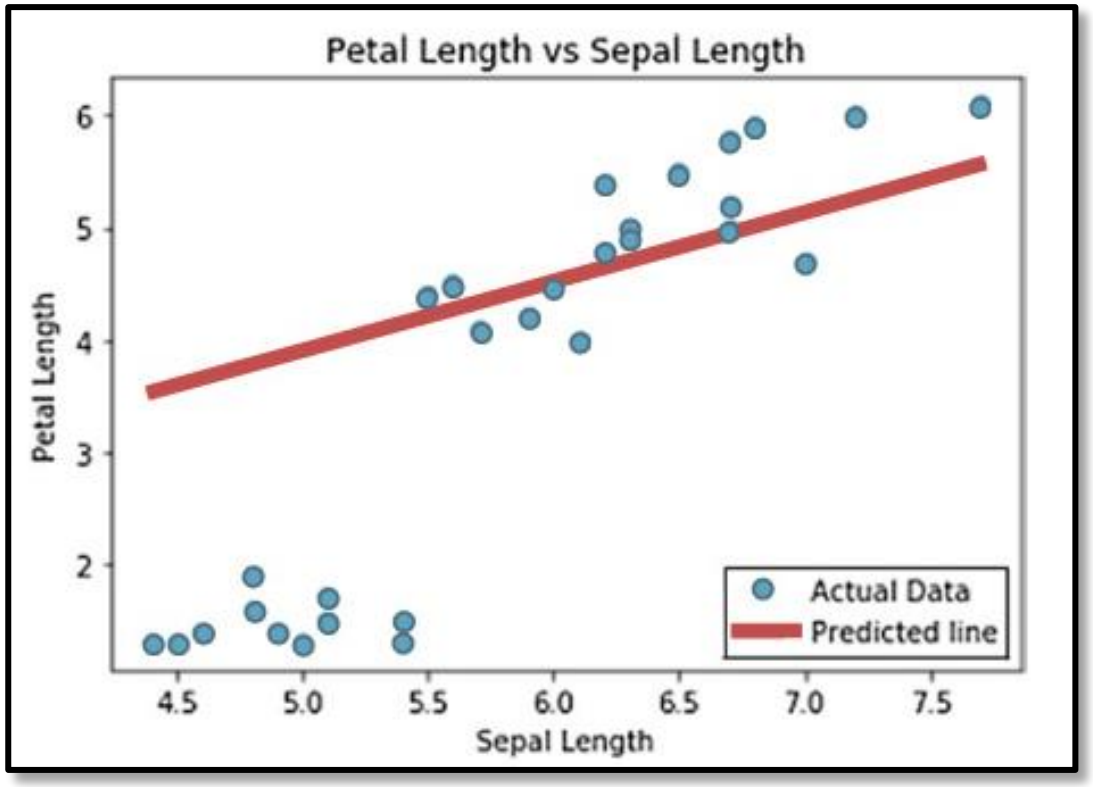
```
plt.plot(x_test, y_test, 'o', label = 'Actual Data')
test_fit = []
for i in x_test:
  test_fit.append(slope * i + y_intercept)

  plt.plot(x_test, test_fit, 'r-', label = 'Predicted line', linewidth = 3)
  plt.legend(loc = 'lower right')
  plt.title('Petal Length vs Sepla Length')
  plt.ylabel('Petal Length')
  plt.xlable('Sepal Length')
  plt.show()
```

After running these codes, we will get the following result:

```
Step Number50 A = [[ 0.57060003]] b = [[ 1.05275011]]
L1 Loss = 0.965698
Step Number100 A = [[ 0.56647497]] b = [[ 1.00924981]]
L1 Loss = 1.124
Step Number150 A = [[ 0.56645012]] b = [[ 0.96424991]]
L1 Loss = 1.18043
Step Number200 A = [[ 0.58122498]] b = [[ 0.92174983]]
L1 Loss = 1.20376
Step Number: 20 A = [[ 0.58945829]] b = [[ 0.90308326]]
L1 Loss = 1.18207
Step Number: 40 A = [[ 0.62599164]] b = [[ 0.88975]]
L1 Loss = 0.826957
Step Number: 60 A = [[ 0.63695836]] b = [[ 0.87108338]]
L1 Loss = 0.838114
Step Number: 80 A = [[ 0.60072505]] b = [[ 0.8450833]]
L1 Loss = 1.52654
Step Number: 100 A = [[ 0.6150251]] b = [[ 0.8290832]]
L1 Loss = 1.25477
```

Linear Regression model

Finally, here is the predicted line, along with the original data points.



Multi-layer perceptron

Now, let us examine an example of a multi-layer perceptron (MLP) built using TensorFlow.
In this example, we will utilize the same dataset as the previous example, the Iris dataset. Our objective remains the same: predicting the petal length based on the sepal length. Therefore, we will consider the sepal length as our input X, and the petal length as our output value y.
We will first import the necessary libraries.

```
import matplotlib as plt
import tensorflow as tf
from sklearn import datasets
import numpy as np
from sklearn.cross_validation import train_test_split
from matplotlib import pyplot
```

We will next load the Iris data set and extract the predictors and targets.

```
#1. Load the data
iris = datasets.load_iris()

predictors_vals = np.array([predictors[0] for predictors in iris.data])
target_vals = np.array([predictors[2] for predictors in iris.data])
```

Linear Regression model

Now, we will split the data set into training and testing data sets. Note that we are dedicating 80% of the data to the training data; that leaves 20% for the test data.

```
#2. Split the data into training and testing data sets
predictors_vals_train, predictors_vals_test, target_vals_train, target_vals_test =
          train_test_split(predictors_vals, target_vals, test_size = 0.2, random_state = 12)
```

Next, we will create placeholders that will hold the predictors and the target.

```
#3. Normalize the data if neede
#4. Create placeholders that  will hold the predictors and the target

x_data = tf.placeholder(shape = [None, 3], dtype = tf.float32)
y_target = tf.placeholder(shape = [None, 1], dtype = tf.float32)
```

Linear Regression model

We will next create variables (weights and biases).

```
#5. Create variables (Weights and bias) that will be tuned during training

hidden_layer_nodes = 10

#first layer
w1 = tf.Variable(tf.ones(shape = [3, hidden_layer_nodes]))
b1 = tf.Variable(tf.ones(shape = [hidden_layer_nodes]))

#second layer
w2 = tf.Variable(tf.ones(shape = [hidden_layer_nodes, 1]))
b2 = tf.Variable(tf.ones(shape = [1]))
```

Now, we will specify the operation of the model.

```
#6. Specify the operations that will be performed in the model

hidden_output = tf.nn.relu(tf.add(tf.matmul(x_data, w1), b1))
final_output = tf.nn.relu(tf.add(matmul(hidden_output, w2), b2))
```

Next, we will declare the model's loss function and optimizer.

```
#7 Declare the loss function and optimizer

loss = tf.reduce_mean(tf.square(y_target - final_output))

opt = tf.train.AdamOptimizer(0.02)

train_step = opt.minimize(loss)
```

Linear Regression model

We will next initialize the variables of the model and create a session.

```
#8. Initialize all variables and create a session to run the graph

sess = tf.Session()
init = tf.global_variables_initializer()

sess.run(init)
```

Now, we will train the model using the training data.

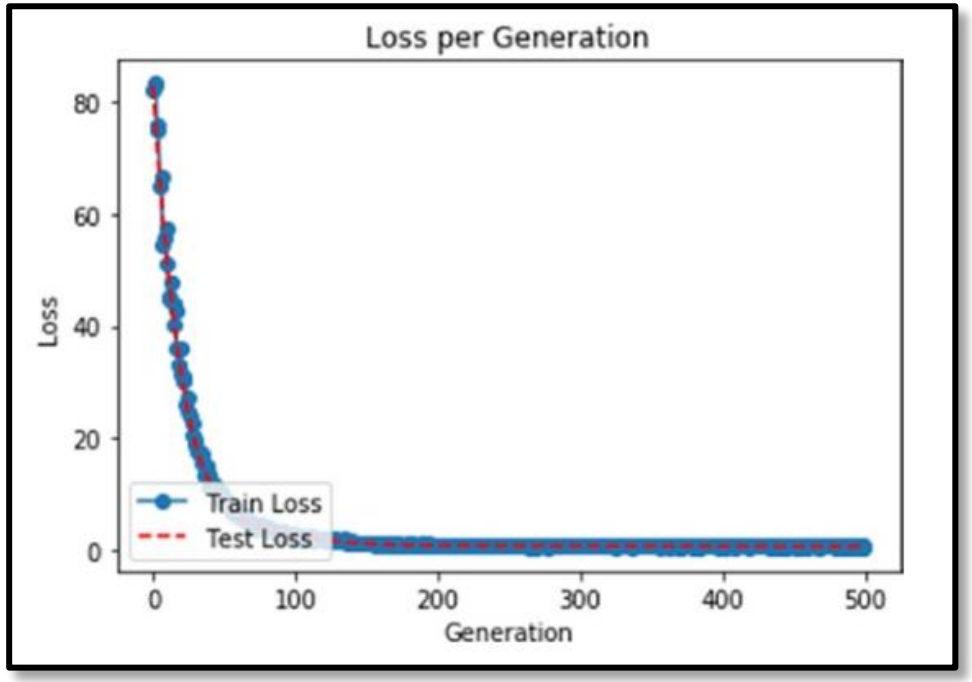
```
#9. Train the model by feeding it the training data

loss_array = []
test_loss = []
batch_size = 20

for i in range(500):
  batch_index = np.random.choice(len(predictors_vals_train), size = batch_size)
  batch_X = predictors_vals_train[batch_index]
  batch_y = np.transpose([target_vals_train[batch_index]])
  sess.run(train_step, feed_dict = {x_data: batch_x, y_target: batch_y})
  batch_loss = sess.run(loss, feed_dict = {x_data: batch_x, y_target: batch_y})
  loss_array.append(np.sqrt(batch_loss))

  test_temp_loss = sess.run(loss, feed_dict = {x_data: predictors_vals_test,
                                               y_target: np.transpose([target_vals_test])})
  test_loss.append(np.sqrt(test_temp_loss))
  if (i + 1) % 50 == 0:
    print('Loss = ' + str(batch_loss))
```

Linear Regression model

We will now evaluate the model by examining the plot of its loss.

```
#Plot the model's loss

pyplot.plot(loss_array, 'o-', label = 'Train Loss')
pyplot.plot(test_loss, 'r--', label = 'Test Loss')
pyplot.title('Loss per Generation')
pyplot.legend(loc = 'lower left')
pyplot.xlabel('Generation')
pyplot.ylabel('Loss')
pyplot.show()
```

Here is the plot of the model's loss. As we can observe, the loss is reducing over time.



Conclusion

In conclusion, the implementation of both a linear regression model and a multi-layer perceptron (MLP) using TensorFlow demonstrates the versatility and power of this framework for machine learning applications. Through our exploration, we have seen how linear regression provides a foundational approach for understanding relationships between variables, while the MLP offers a more complex and flexible solution for-

capturing intricate patterns within data. By leveraging TensorFlow's robust features, we can efficiently train and evaluate these models, leading to valuable insights and predictions.