CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

TensorFlow

Machine Learning

Machine Learning

Lecture: Automatic Differentiation with TensorFlow 2

Ted Scully

Automatic Differentiation in TensorFlow

- We initially create an **instance of tf.GradientTape**, which records all (forward-pass) operations once it has been created.

- To determine the **gradients of the loss function** with respect to some variables, we call **tape.gradient**.
  - The first argument is the "**target**" for the calculation, i.e. typically the output of the loss function (or whatever function you are trying to optimize)
  - The second argument is the "**source**" these are the variables values you can alter in order to optimize the target loss function.

- The **gradient** function will **return** the partial derivatives of each of the listed trainable variables with respect to the (loss) function.

- By default, the resources held by a **GradientTape are released** as soon as GradientTape.gradient() method is called.

- We can either update the variables ourselves or more commonly we can use an existing optimizer to update the gradients for us.

Automatic Differentiation in TensorFlow

- To help illustrate the operation of GradientTape, let's take a simple example.
- Our objective is to calculate the derivative of the following function when **x = 10**.

$$3x^3 + x^2$$

Automatic Differentiation in TensorFlow

- To help illustrate the operation of GradientTape, let's take a simple example.
- Our objective is to calculate the derivative of the following function when **x = 10**.

$$3x^3 + x^2$$

```
Import tensorflow as tf

def simpleFunc(x):
  return 3*(x**3) + x**2
```

Automatic Differentiation in TensorFlow

- To help illustrate the operation of GradientTape, let's take a simple example.
- Our objective is to calculate the derivative of the following function when **x = 10**.

$$3x^3 + x^2$$

```
Import tensorflow as tf

def simpleFunc(x):
  return 3*(x**3) + x**2

# Create a tensorflow variable with an itial value of 10.0
x = tf.Variable(10.0, tf.float32)
```

Automatic Differentiation in TensorFlow

- To help illustrate the operation of GradientTape, let's take a simple example.
- Our objective is to calculate the derivative of the following function when **x = 10**.

$$3x^3 + x^2$$

```
Import tensorflow as tf

def simpleFunc(x):
  return 3*(x**3) + x**2

# Create a tensorflow variable with an itial value of 10.0
x = tf.Variable(10.0, tf.float32)

with tf.GradientTape() as tape:
  prediction = simpleFunc(x)
  current_grad = tape.gradient(prediction, x)

print (current_grad)
```

GradientTape will record all operation that take place with the with statement.

```
tf.Tensor(920.0, shape=(), dtype=float32)
```

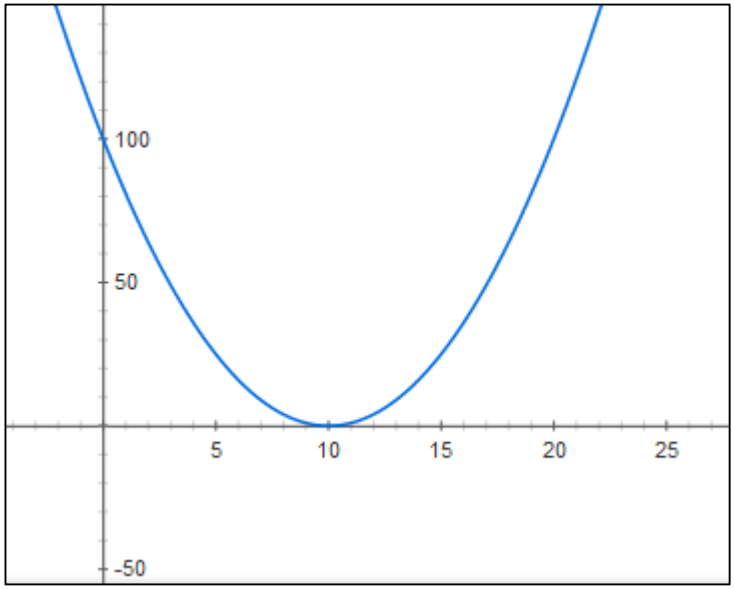
Using TensorFlow to Minimize a Function

- To illustrate the use of automatic differentiation we will start with the following example, we are going to write a TensorFlow program to find the minimum value of x in the following function:

- $X^2 – 20X + 100$

- The minimum value of X is obvious but we will illustrate the process of building a simple TensorFlow program to solve this problem.

- Let’s first built a function representing our quadratic equation above.



```
def predict(x):
        # x^2 – 20x + 100
        firstTerm = x**2
        secondTerm = -20.0* x
        quadratic = firstTerm + secondTerm +100.0
        return quadratic
```

```
def predict(x):
        # x^2 – 20x + 100
        firstTerm = x**2
        secondTerm = -20.0* x
        quadratic = firstTerm + secondTerm +100.0
        return quadratic


learning_rate = 0.1
iterations = 50

x = tf.Variable(50.0, tf.float32)
```

```
def predict(x):
        # x^2 – 20x + 100
        firstTerm = x**2
        secondTerm = -20.0* x
        quadratic = firstTerm + secondTerm +100.0
        return quadratic


learning_rate = 0.1
iterations = 50

x = tf.Variable(50.0, tf.float32)

for i in range(iterations):

  with tf.GradientTape() as tape:
    prediction = predict(x)

  gradient = tape.gradient(prediction, x)
```

We want to determine the value of x that will minimize our quadratic equation. Therefore, we iterate 50 times. Each time we iterate we update the value of the variable x using our gradient descent rule.

```
def predict(x):
        # x^2 – 20x + 100
        firstTerm = x**2
        secondTerm = -20.0* x
        quadratic = firstTerm + secondTerm +100.0
        return quadratic


learning_rate = 0.1
iterations = 50

x = tf.Variable(50.0, tf.float32)

for i in range(iterations):

  with tf.GradientTape() as tape:
    prediction = predict(x)

  gradient = tape.gradient(prediction, x)
```

Gradient tape once initialized records all operations performed within it's context.

Next we call gradient, which calculate the gradients of the prediction function with respect to x the input variable.

```
def predict(x):
        # x^2 – 20x + 100
        firstTerm = x**2
        secondTerm = -20.0* x
        quadratic = firstTerm + secondTerm +100.0
        return quadratic


learning_rate = 0.1
iterations = 50

x = tf.Variable(50.0, tf.float32)

for i in range(iterations):

  with tf.GradientTape() as tape:
    prediction = predict(x)

  gradient = tape.gradient(prediction, x)

  x.assign_sub(gradient * learning_rate)

  print (x)
```

The next step is to use my normal gradient descent update rule to update the tensorflow variable x.

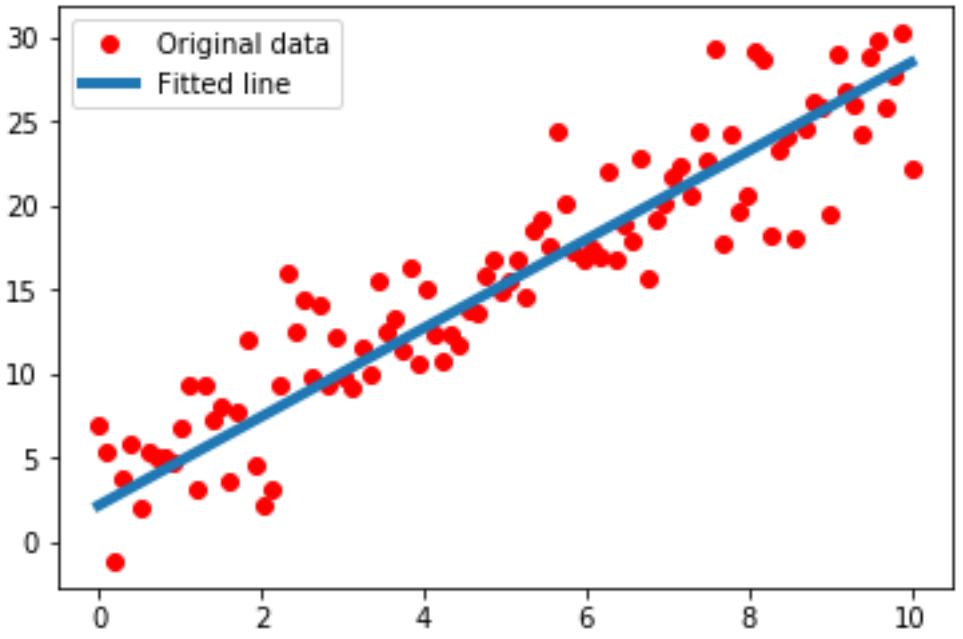
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=42.0>
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=35.6>
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=30.48>
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=26.383999>
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=23.107199>
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=20.48576>
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=18.388607>
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=16.710886>
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=15.368709>
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=14.294967>
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=13.435973>
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=12.748778>
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=12.199022>
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=11.759218>
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=11.407374>
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=11.125899>
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=10.90072>
.....
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=10.020282>
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=10.016226>
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=10.01298>
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=10.010385>

Linear Regression Using TensorFlow

- In this example we are going to build a Linear Regression algorithm using TensorFlow using GradientTape to calculate the derivatives.

  - $h(x) = \lambda_1 x + b$

Which of the above should be a variable in our TensorFlow program?

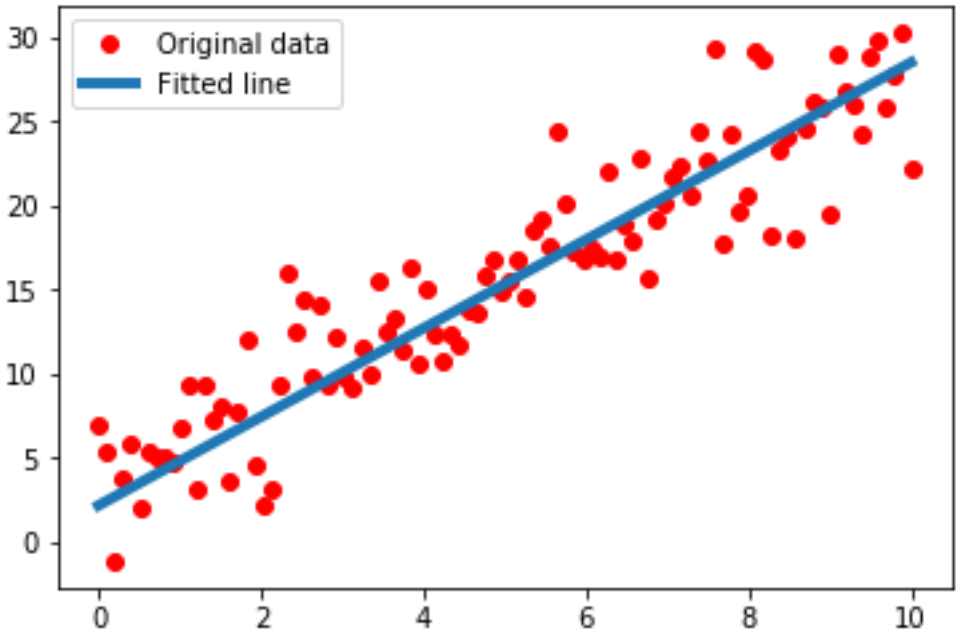


Linear Regression Using TensorFlow

- In this example we are going to build a Linear Regression algorithm using TensorFlow using GradientTape to calculate the derivitatives.

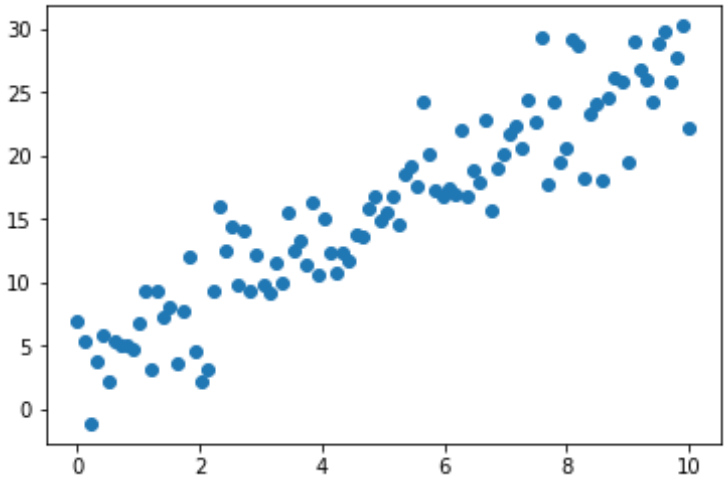
  - $h(x) = \lambda_1 x + b$

- You will remember our equation for linear regression above. There are two variables that we must adjust in order to minimize the loss function.
- Those are $\boldsymbol{\lambda_1}$ and **b**. Note in this example x is just the training data, which will not change.



Linear Regression – Preparing Data

- In the code we create some sample data for our linear regression model.
- Notice we also specify some of the parameters for our linear regression model.




```
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np


def generateData():

  np.random.seed(10)
  train_X=np.linspace(0,10,100)
  noise=np.random.normal(0,1.5,100)

  train_Y=((2.5*train_X)+3) +noise

  return train_X, train_Y


x_train, y_train = generateData()
plt.scatter(x_train,y_train)
plt.legend()
plt.show()
```

Linear Regression – Model and Loss Function

- Next we create a function for our linear regression model (takes in training data and output results)

- We also create a function for our squared error cost. . Notice we are comparing the predicted output with the actual labels Y.

```
def predict(x, w, c):
  y = w * x + c
  return y


def loss_func(y_pred, y_true):
  error = y_pred - y_true
  sumSqErrors = tf.reduce_sum(error**2)
  currentLoss = sumSqErrors/(2*len(y_pred))
  return currentLoss
```

$$\frac{1}{2m}\sum_{i=0}^{m}((h(x^i)-y^i))^2$$

Linear Regression – Training Process

- We create a variable w and c and set their values to 0.
- As with the previous example each time we iterate, we create an instance of GradientTape, which tracks the forward pass operations. We then calculate the derivatives of our loss function with respect to the variables w and c. Next we use these gradients in the normal way to the variable values for our linear regression model.

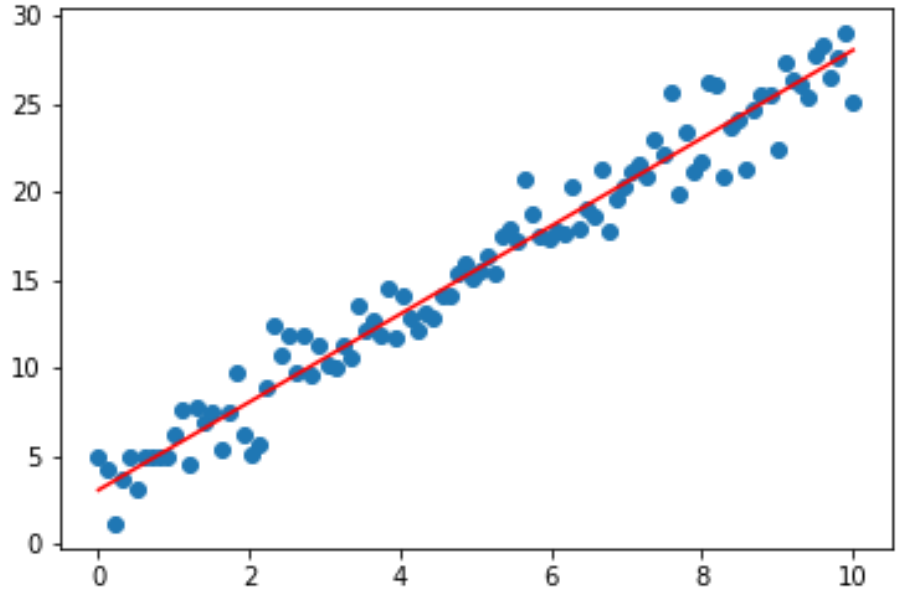
```
w = tf.Variable(0.)
c = tf.Variable(0.)

learning_rate = 0.001
steps = 10000

for i in range(steps):

  with tf.GradientTape() as tape:
    predictions = predict(x_train, w, c)
    loss = loss_func(predictions, y_train)

  gradients = tape.gradient(loss, [w, c])

  m.assign_sub(gradients[0] * learning_rate)
  c.assign_sub(gradients[1] * learning_rate)

  if i % 100 == 0:
        print("Step ",i,", Loss ",loss.numpy())
```

Linear Regression – Training Process

- As we run the code we should see the loss gradually decline.

```
plt.scatter(x_train,y_train)
plt.plot(x_train,predict(x_train),"r")
plt.show()
```

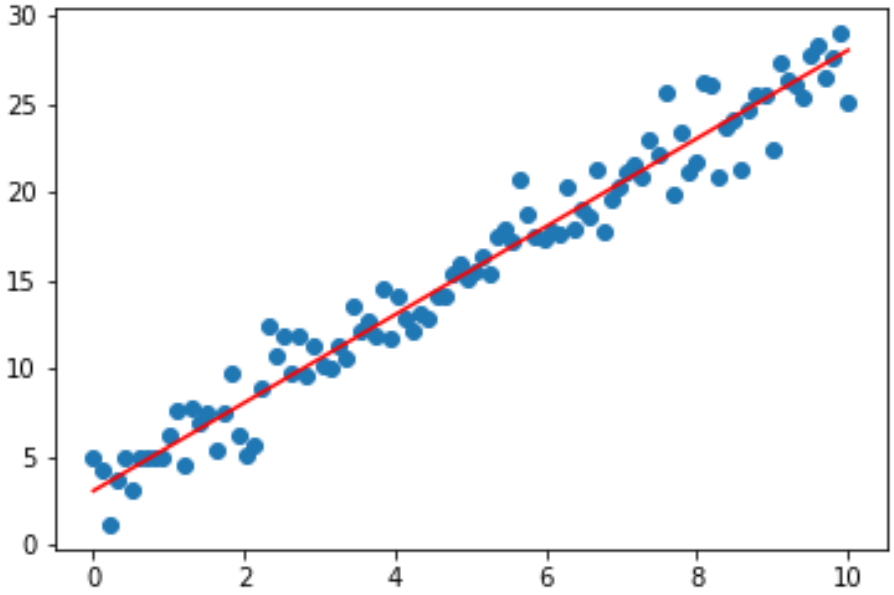
- The following code generates a graph showing the linear regression model generated after our gradient descent has updated the values of m and c.



Linear Regression – Training Process

- In the previous examples we have only used standard gradient descent.
- However, TensorFlow offers a host of in-built optimizers (Adam, Adagrad, RMSProp, etc) that we can plug into our code.
- You can find the full list of optimizers at tf.keras.optimizers

- The code on the next slide will demonstrate how to alter our linear regression model to use the inbuilt Adam optimizer.



Linear Regression – Training with Adam Optimizer

```
w = tf.Variable(0.)
c = tf.Variable(0.)
adam_optimizer = tf.keras.optimizers.Adam()


learning_rate = 0.001
steps = 10000

for i in range(steps):

  with tf.GradientTape() as tape:
    predictions = predict(x_train,w, c)
    loss = loss_func(predictions, y_train)

  gradients = tape.gradient(loss, [w, c])

  adam_optimizer.apply_gradients(zip(gradients, [w,c]))
```

As you can see there has been little change to the code from the previous example.

1. We create an instance of the Adam optimizers.
2. Each time we iterate through out training loop we calculate the gradient for m and c. We then call the Adam apply_gradients method in the Adam optimizer and pass it the gradient and the associate variables and that's it!

TF2.0 -Logistic Regression for Binary Classification

- In the following example, we are going to build a Logistic Regression model using TensorFlow for the digits dataset (This dataset is made up of 1797 8x8 images).

- We previously tackled the problem of digit recognition in a practical lab.

- As with the original lab we will focus on binary classification and pick the images corresponding to two digits.

- You can find the full code for this example here.

Vectorised Version of Logistic Regression

Let's assume we have a training set with m rows and n columns (features).

$$X = \begin{bmatrix} x_1^1 & \cdots & x_1^m \\ \vdots & \ddots & \vdots \\ x_n^1 & \cdots & x_n^m \end{bmatrix}$$

We create a matrix **X** as follows, where each **column is a training instance** and each row is a feature.

Let's also assume that we have a vector W that contains all lambda values as shown. As usual the number of weights n is the same as the number of features

$$W = \begin{bmatrix} \boldsymbol{\lambda_1} \\ \boldsymbol{\lambda_2} \\ \vdots \\ \boldsymbol{\lambda_n} \end{bmatrix}$$

Finally, let's assume we have a row vector Y that contains the actual regression value for each training instance i.

$$\mathrm{Y} = [\, y^1, y^2, \ \ldots ym]$$

Reminder of Logistic Regression

- The first step using vectors is to perform vector matrix multiplication as follows (notice this allows us to multiply the vector of lambda weights by each training example and add the bias):

$$A = W^T X + b$$

- The resulting vector A will look like this:

$$A = \langle a^1, a^2, \ldots, a^m \rangle$$

- Where a^1 is $x_1^1 \boldsymbol{\lambda_1} + x_2^1 \boldsymbol{\lambda_2} + \ldots x_n^1 \boldsymbol{\lambda_m} + \boldsymbol{b}$

- To obtain the predicted output for the current values of $\boldsymbol{\lambda}$ and **b** we can do the following.

$$H = \text{logistic}(A)$$

Prepare Data

```
from sklearn import datasets
from sklearn import preprocessing
import tensorflow as tf

def loadData():

    digits = datasets.load_digits()
    X_digits = digits.data
    y_digits = digits.target

    # As this is a binary classification
    # we are only going to pull out two classes
    indexD1 = y_digits==0
    indexD2 = y_digits==1
    allindices = indexD1 | indexD2

    # Standarize the data
    X_digits = X_digits[allindices]
    scaler = preprocessing.StandardScaler()
    X_digits = scaler.fit_transform(X_digits)

    y_digits = y_digits[allindices]
    n_samples = len(X_digits)
    return X_digits, y_digits
```

In this program we only want to perform **binary** classification. Therefore, we load the digits dataset and an extract only those images that correspond to 0 and 1.

The size of X_digits and y_digits is (360, 64) and (360,) respectively.

Prepare Data

```
def splitData(X_digits, y_digits):

  n_samples = len(y_digits)
  # Training data
  X_train = X_digits[:int(.9 * n_samples)]
  y_train = y_digits[:int(.9 * n_samples)]

  # Test data
  X_test = X_digits[int(.9 * n_samples):]
  y_test = y_digits[int(.9 * n_samples):]

  # Reshape the label data so that it is a real
  # column vector
  y_train = y_train.reshape(1,-1)
  y_test = y_test.reshape(1,-1)

  # Get the transpose of the feature data
  X_train = X_train.T
  X_test = X_test.T
  return X_train, y_train, X_test, y_test
```

In this code we extract a training and validation set. We then reshape the label data so that it is a true 2D NumPy array containing just one row. We also transpose our feature data (remember we want our feature data in a matrix where each row is a feature).

The Forward Pass and Loss

```
def predict(x, w_T, b):
    # We need to mutliply each training example by the weights and add bias
    y_pred = tf.matmul(w_T, x) + b
    # Pipe the results through the signmoid activiation function.
    y_pred_sigmoid = tf.sigmoid(y_pred)
    return y_pred_sigmoid


def loss_funct(y, y_pred):
   # Calculate the cross entropy error for all training data
    x_entropy = tf.nn.sigmoid_cross_entropy_with_logits(logits=y_pred, labels=y)

    # Calculate the mean cross entropy error
    loss = tf.reduce_mean(x_entropy)

    return loss
```

In the code above specified our forward pass (where we push all data through out logistic neuron) and calculate the cross entropy loss.

Calculating Accuracy

```
def calculate_accuracy(x, y, w, b):

  y_pred_sigmoid = predict(x, w, b)
  # Round the predictions by the logistical unit to either 1 or 0
  predictions = tf.round(y_pred_sigmoid)

  # tf.equal will return a boolean array: True if prediction correct, False otherwise
  # tf.cast converts the resulting boolean array to a numerical array
  # 1 if True (correct prediction), 0 if False (incorrect prediction)
  predictions_correct = tf.cast(tf.equal(predictions, y), tf.float32)

  # Finally, we just determine the mean value of predictions_correct
  accuracy = tf.reduce_mean(predictions_correct)
  return accuracy
```

In this code we calculate the accuracy of our model given the current values of w and b

Logistic Reg Model

```
def main():

    learning_rate = 0.01
    num_Iterations = 70
    adam_optimizer = tf.keras.optimizers.Adam()

    X_digits, y_digits = loadData()
    tr_x, tr_y, te_x, te_y  = splitData(X_digits, y_digits)

    tr_x = tf.cast(tr_x, tf.float32)
    te_x = tf.cast(te_x, tf.float32)
    tr_y = tf.cast(tr_y, tf.float32)
    te_y = tf.cast(te_y, tf.float32)

    # We need a coefficient for each of the features and a single bias value
    w = tf.Variable(tf.random.normal([ 1,tr_x.shape[0]], mean=0.0, stddev=0.05))
    b = tf.Variable([0.])
```

Notice here we cast all training data so that it is all float 32 data types. This will avoid any problems with automatic type conversion.

We also create a tf.Variable for both the training data and the bias.

Gradient Descent Loop

```
# Iterate our training loop
for i in range(num_Iterations):

  # Create an instance of GradientTape to monitor the forward pass
  # and calcualte the gradients for each of the variables m and c
  with tf.GradientTape() as tape:
    y_pred = predict(tr_x, w, b)
    currentLoss = loss_funct(tr_y, y_pred)

  gradients = tape.gradient(currentLoss, [w, b])
  accuracy = calculate_accuracy(tr_x, tr_y, w, b)
  print ("Iteration ", i, ": Loss = ",currentLoss.numpy(), "  Acc: ", accuracy.numpy())
  adam_optimizer.apply_gradients(zip(gradients, [w,b]))

test_accuracy = calculate_accuracy(te_x, te_y, w, b)
print ("Test Accuracy : ", test_accuracy)
```

Notice above we iterate using our gradient descent rule. We use GradientTape to track the forward pass and loss calculation. We then get the gradient of the variables and use this to update the variable values using Adam.

Iteration 0 : Loss = 0.70315564 Acc: 0.7407407
Iteration 1 : Loss = 0.7003976 Acc: 0.7808642
Iteration 2 : Loss = 0.69765574 Acc: 0.8055556
Iteration 3 : Loss = 0.6949312 Acc: 0.8240741
Iteration 4 : Loss = 0.69222474 Acc: 0.8333333
Iteration 5 : Loss = 0.6895374 Acc: 0.85493827
Iteration 6 : Loss = 0.6868702 Acc: 0.86728394
Iteration 7 : Loss = 0.68422437 Acc: 0.89506173
Iteration 8 : Loss = 0.68160063 Acc: 0.904321
Iteration 9 : Loss = 0.6790001 Acc: 0.91358024
Iteration 10 : Loss = 0.6764237 Acc: 0.9228395
Iteration 11 : Loss = 0.67387235 Acc: 0.92901236
Iteration 12 : Loss = 0.67134696 Acc: 0.9382716
Iteration 13 : Loss = 0.6688484 Acc: 0.9506173
Iteration 14 : Loss = 0.66637725 Acc: 0.9537037
Iteration 15 : Loss = 0.66393447 Acc: 0.95987654
Iteration 16 : Loss = 0.66152066 Acc: 0.95987654
……

Iteration 65 : Loss = 0.5812083 Acc: 0.9907407
Iteration 66 : Loss = 0.58022994 Acc: 0.9907407
Iteration 67 : Loss = 0.5792707 Acc: 0.9907407
Iteration 68 : Loss = 0.57833004 Acc: 0.9907407
Iteration 69 : Loss = 0.5774078 Acc: 0.99382716
Test Accuracy : tf.Tensor(1.0, shape=(), dtype=float32)