CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

TensorFlow

Machine Learning

Machine Learning

Lecture: TensorFlow

Ted Scully

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.

It's good practice to **always** use name scopes in your TF code as shown earlier.

However, in order to compress the code we use in subsequent slides I will omit name scopes.

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 row from our training** dataset. For example, the first column contains all n features from the first row of our dataset.

Let's also assume that we have a vector W that contains all lambda values as shown.

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

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

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

Recap

- 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 preprocessing

def loadData():

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

    indexD1 = y_digits==0
    indexD2 = y_digits==1
    allindices = indexD1 | indexD2

    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)
```

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

```
# continued from previous slide
# 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)

 X_train = X_train.T
 X_test = X_test.T
 return X_train, y_train, X_test, y_test
```

Once we have extracted the relevant features and labels we split it into training and test data.

Here we reshape y_train and y_test into a true 2D matrix with a single row.

We get the transpose of the feature data (both train and test). Remember this is exactly the same as we did when we covered Logistic Regression with matrix operations.

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 row from our training** dataset. For example, the first column contains all n features from the first row of our dataset.

Let's also assume that we have a vector W that contains all lambda values as shown.

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

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

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

Creating the graph for LR using TF

```
def main():

    learning_rate = 0.01
    num_Iterations = 1000
    display_step = 100

    tr_x, tr_y, te_x, te_y = loadData()

    # We will load our training and target labels into x and y respectively
    x = tf.placeholder(tf.float32, [tr_x.shape[0], None])
    y_ = tf.placeholder(tf.float32, [1, None])

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

    # In the graph we will mutliply each training data 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)
```

Creating the graph for LR using TF

```
def main():

    learning_rate = 0.01
    num_Iterations = 1000
    display_step = 100

    tr_x, tr_y, te_x, te_y = loadData()

    # We will load our training and target labels into x and y respectively
    x = tf.placeholder(tf.float32, [tr_x.shape[0], None])
    y_ = tf.placeholder(tf.float32, [1, None])

    # We need
    w = tf.Va
    w_T = tf
    b = tf.Va

    # In the
    y_pred =
    # Pipe th
    y_pred_sigmoid = tf.sigmoid(y_pred)
```

Notice we only partially specify the shape of the training placeholder. We specify the number of rows but not the number of columns. (Remember rows are features and columns are instances). This will allow us to reuse this placeholder for test and training data. We specify the number of rows (features) in x is the same as the number of rows (features) in the training data.

Creating the graph for LR using TF

Notice we specify the same number of weights as there are features in the training data. The bias is just a single value. While I create the weights as random values they could just as easily have been set to zero. It is also important to understand that the weight will be a column vector to facilitate matrix multiplication. We then get the transpose of the weights for matrix multiplication

```
def mair

    learni
    num_
    displa

    tr_x, t

    # We will load our training and target labels into x and y respectively
    x = tf.placeholder(tf.float32, [tr_x.shape[0], None])
    y_ = tf.placeholder(tf.float32, [1, None])

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

    # In the graph we will mutliply each training data 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)
```

Creating the graph for LR using TF

```
def main():

    learning_rate = 0.01
    num_Iterations = 1000
    display_step = 100

    tr_x, tr_y, te_x, te_y = load

    # We will load our training and target labels into x and y respectively
    x = tf.placeholder(tf.float32, [tr_x.shape[0], None])
    y_ = tf.placeholder(tf.float32, [1, None])

    # We need a coefficient for each of the features and a single
    w = tf.Variable(tf.random_normal([tr_x.shape[0], 1], mean=
    w_T = tf.transpose(w)
    b = tf.Variable([0.])

    # In the graph we will mutliply each training data 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)
```

Once we have multiplied our weights by all the training data and add the bias we take the predictions and pipe them through the Sigmoid activation function. This will produce predictions in the range 0 – 1.

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

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

Creating the graph for LR using TF

```
# Continued from previous slide
# 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)

# add gradient descent
train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)



# 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.ca
# 1 if
predi

# Fina
accuracy = tf.reduce_mean(predictions_correct)
```

Once we have calculated the predictions we need to work out cost function. Remember the cost function we use in Logistic regression is the **Cross Entropy** function. We can plug this in directly with TensorFlow. This will return the cross entropy loss for each training example. We must next calculate the mean cross entropy error.

Creating the graph for LR using TF

```
# Continued from previous slide
# 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)

# add gradient descent
train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)



# 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
```

Once we have our loss function defined we apply Gradient Descent, just as we did in linear regression

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

Creating the graph for LR using TF

```
# Continued from previous slide
# Ca
 x_e

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

# add gradient descent
train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)


# 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)
```

Finally I want to calculate the prediction accuracy for the current weights and bias.

Create and run the Session

```
# Start training
  with tf.Session() as sess:

    # Initialize all variables
    sess.run(tf.global_variables_initializer())

    for i in range(num_Iterations):
      feed_train = {x: tr_x, y_: tr_y}
      sess.run(train_step, feed_dict=feed_train)

      if i % display_step == 0:
        currentLoss, train_accuracy = sess.run([loss, accuracy], feed_train)
        print('Iteration: ', i,' Loss: ', currentLoss, ' Accuracy: ',train_accuracy)


    feed_test = {x: te_x, y_: te_y}
    print('Test Accuracy: ', sess.run(accuracy, feed_test))
```

Next we run our gradient descent optimizer.

Create and run the Session

```
# Start training
  with tf.Session() as sess:

    # Initialize all variables
    sess.run(tf.global_variables_initializer())

    for i in range(num_Iterations):
        feed_train = {x: tr_x, y_: tr_y}
        sess.run(train_step, feed_dict=feed_train)

        if i % display_step == 0:
            currentLoss, train_accuracy = sess.run([loss, accuracy], feed_train)
            print('Iteration: ', i,' Loss: ', currentLoss, ' Accuracy: ',train_accuracy)


    feed_test = {x: te_x, y_: te_y}
    print('Test Accuracy: ', sess.run(accuracy, feed_test))
```

Periodically, we check the the current loss value and the accuracy on the training dataset. Finally, we calculate accuracy on the test data.

Create and run the Session

```
# Start training
  with tf.Session() as sess:

    # Initialize all variables
    sess.run(tf.global_variable

    for i in range(num_Iteratic
      feed_train = {x: tr_x, y_:
      sess.run(train_step, feed

      if i % display_step == 0:
        currentLoss, train_ac
        print('Iteration: ', i,' L


    feed_test = {x: te_x, y_: te
    print('Test Accuracy: ', sess
```

```
Iteration:  0  Loss:  0.648425  Accuracy:  0.654321
Iteration:  50  Loss:  0.171431  Accuracy:  0.996914
Iteration:  100  Loss:  0.100357  Accuracy:  0.996914
Iteration:  150  Loss:  0.0724462  Accuracy:  0.996914
Iteration:  200  Loss:  0.0573896  Accuracy:  0.996914
Iteration:  250  Loss:  0.0478887  Accuracy:  0.996914
Iteration:  300  Loss:  0.0413052  Accuracy:  0.996914
Iteration:  350  Loss:  0.0364504  Accuracy:  0.996914
Iteration:  400  Loss:  0.0327079  Accuracy:  1.0
Iteration:  450  Loss:  0.0297257  Accuracy:  1.0
Iteration:  500  Loss:  0.0272873  Accuracy:  1.0
Iteration:  550  Loss:  0.0252523  Accuracy:  1.0
Iteration:  600  Loss:  0.0235251  Accuracy:  1.0
Iteration:  650  Loss:  0.0220387  Accuracy:  1.0
Iteration:  700  Loss:  0.0207443  Accuracy:  1.0
Iteration:  750  Loss:  0.0196058  Accuracy:  1.0
Iteration:  800  Loss:  0.0185957  Accuracy:  1.0
Iteration:  850  Loss:  0.0176927  Accuracy:  1.0
Iteration:  900  Loss:  0.0168799  Accuracy:  1.0
Iteration:  950  Loss:  0.0161441  Accuracy:  1.0
Test Accuracy:  1.0
```

Multi-class Classification on MINST Dataset

- The MNIST database (Modified National Institute of Standards and Technology database) is a large database of handwritten digits that is commonly used for training various image processing systems.

- Each image is a **28 x 28** pixel image, flattened to be a 1-d tensor of size **784**. Each comes with a label.

- MNIST is available as a TensorFlow Dataset object. It has
  - **55,000** instances of training data (MNIST.train),
  - **10,000** instances of test data (MNIST.test), and
  - **5,000** points of validation data (MNIST.validation).

- We are going to look at this problem in the context of multi-class classification using a single SoftMax layer.

Softmax Activation Layer

The activation in a softmax layer can be described in two steps.

$$\sigma(x_j) = \frac{e^{x_j}}{\sum_i e^{x_i}}$$

1. We calculate a new vector t, we calculate e to the power of the pre-activation outputs.

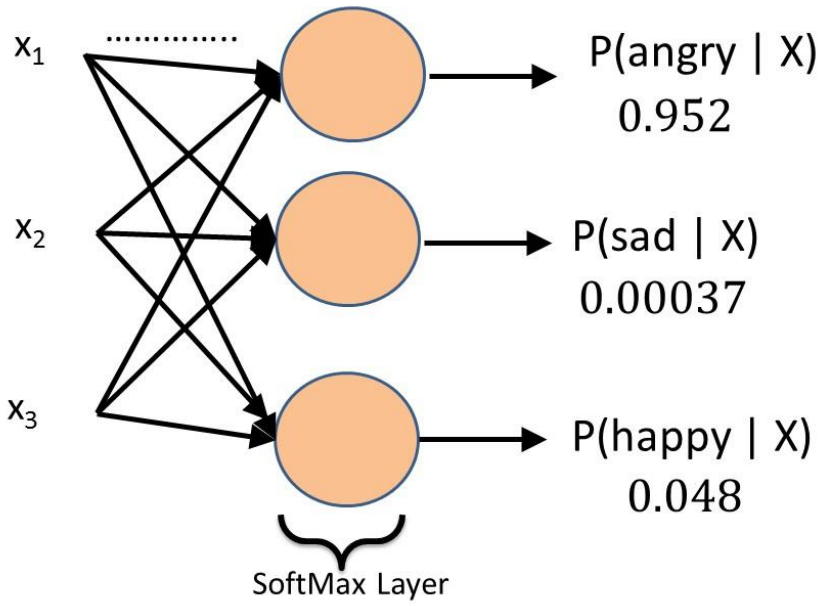
$$t = e^{A^{[1]}}$$

2. Next we calculate the output of each neuron in the softmax layer as:

$$H^{[1]} = \frac{e^{A^{[1]}}}{\sum t} = \frac{t}{\sum t}$$

One Hot Encode Target Labels for Softmax

- Notice that the output of the Softmax layer is a vector with the same number of elements as there are classes.
- For example below the output would be the vector <0.952, 0.00037, 0.048>.
- Therefore, in a classification problem you can **convert your original target labels from being numerical values to be one-hot-encoded values**.
- For example, we may have three classes, 0 for angry, 1 for sad and 2 for happy. We would represent each of these as <1,0,0> for angry, <0,1,0> for sad and <0,0,1> for happy.



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

tf.reset_default_graph()



# Read in dataset
mnist = tf.keras.datasets.mnist
(X_train, Y_train), (X_test, Y_test) = mnist.load_data()

# Reshape training dataset so that the features are flattened
X_train = X_train.reshape(X_train.shape[0], -1).astype('float32')
X_test = X_test.reshape(X_test.shape[0], -1).astype('float32')

# Normalize training data
X_train = X_train/255.0
X_test = X_test/255.0

X_train = X_train.T
X_test = X_test.T
```

The training data is originally in 3D format. We reshape it to be 2D. Essentially we flatten all the features.

Next we normalize the training data by dividing by 255. Remember pixel values are between 0-255.
Finally we get the transpose so that our training data is features * instances

```
# Convert labels to one-hot-encoded
number_of_classes = 10
Y_train = tf.keras.utils.to_categorical(Y_train, number_of_classes)
Y_test = tf.keras.utils.to_categorical(Y_test, number_of_classes)

# Transpose train labels so that dimensions
# are number of classes * number of instances
Y_train = Y_train.T
Y_test = Y_test.T

print ("Data extracted and reshaped: ")
print (X_train.shape, Y_train.shape, X_test.shape, Y_test.shape)
```

Convert the integer labels into one hot encoded labels. Therefore, we will convert Y_train from a shape of (60,000) to (60,000, 10).

In the next step we get the transpose of the labels so that they are 10 rows with 60,000 columns.

Training

(784, 60000)

(10, 60000)

Testing

(784, 10000)

(10, 10000)

Next we need placeholders for our training data. What should be the size of the place holders?

```
# Convert labels to one-hot-encoded
number_of_classes = 10
Y_train = tf.keras.utils.to_categorical(Y_train, number_of_classes)
Y_test = tf.keras.utils.to_categ

# Transpose train labels so tha
# are number of classes * num
Y_train = Y_train.T
Y_test = Y_test.T

print ("Data extracted and resl
print (X_train.shape, Y_train.s

# Model Parameters
n_epochs = 50
learningRate = 0.1


# Placeholders for the traing and label data
X = tf.placeholder(tf.float32, [784, None], name='image')
y = tf.placeholder(tf.int32, [10, None], name='label')
```

So the training feature data will have the shape 784 rows (each row is a feature) by unspecified number of columns.

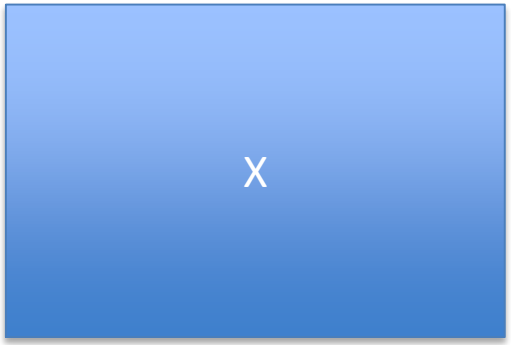
The label data will have 10 rows (number of classes) by unspecified number of columns.

What **variables** should I maintain in my program?

```
# Model Parameters
n_epochs = 50
learningRate = 0.1


# Placeholders for the traing and label data
X = tf.placeholder(tf.float32, [784, None], name='image')
y = tf.placeholder(tf.int32, [10, None], name='label')


# Create weight and bias matrices (variables) for each layer of our network
W = tf.Variable( tf.random_normal([10, 784], mean=0.0, stddev=0.05) )
b = tf.Variable( tf.zeros([10,1]) )


# Push feature data through first layer of NN
A1 = tf.add( tf.matmul(W, X), b )
```

Training

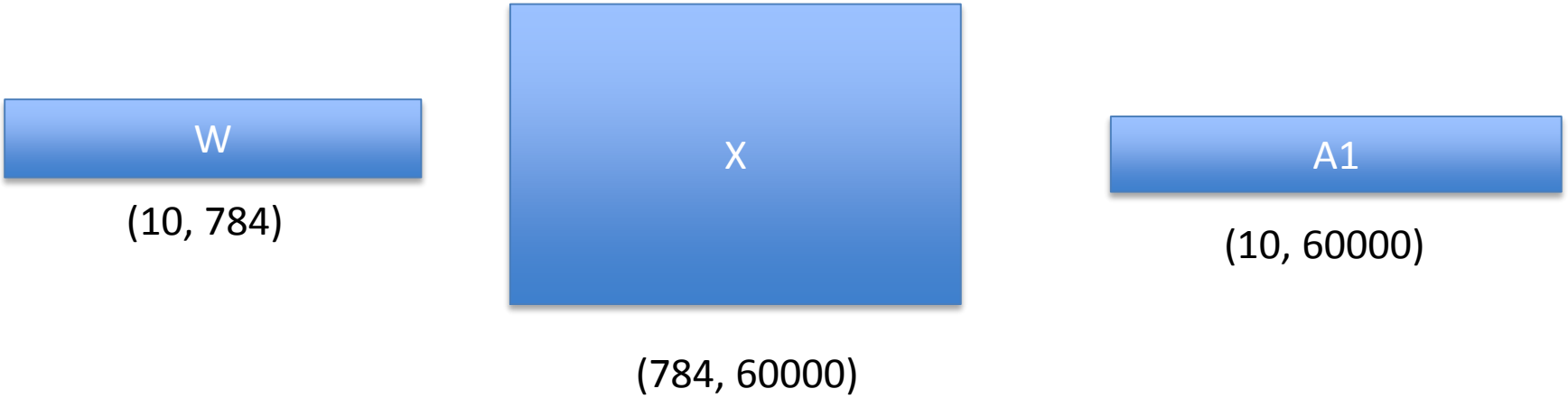
(10, 784)



(784, 60000)

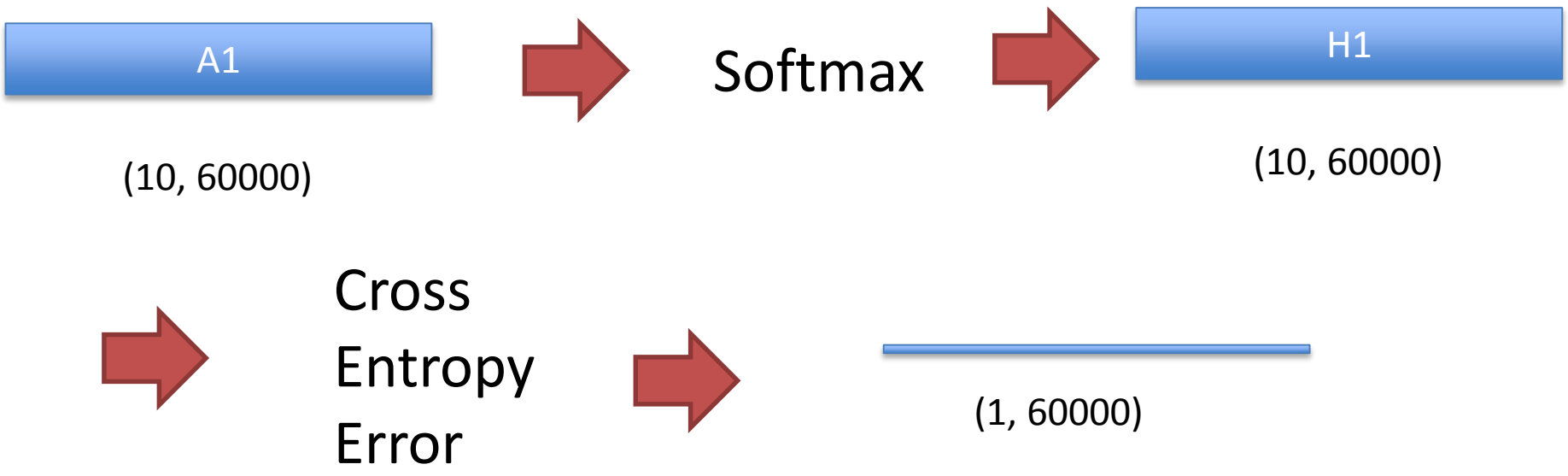
Training

Please note to simplify the illustration below I have omitted the addition of bias.



Softmax

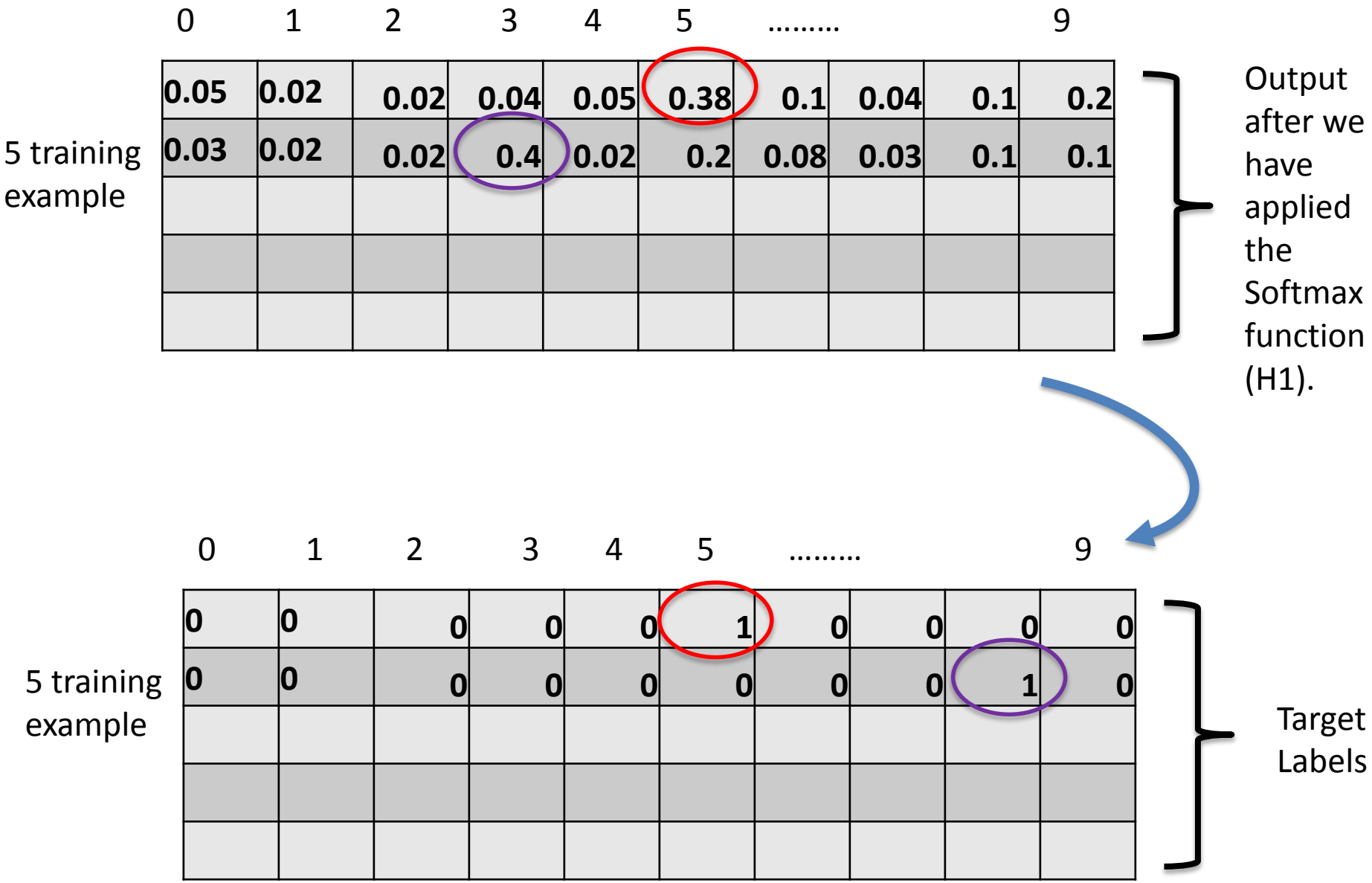
- In our code we are going to pass A1 to a TensorFlow method **tf.nn.softmax_cross_entropy_with_logits_v2** which will take the pre-activation outputs, calculate the softmax output and **<u>return the cross entropy error</u>** for each training instance.



- The above function takes in the logits and labels in the following shape **[number_instance, num_classes].** Therefore, before we pass A1 and our labels to this method we must first transpose them.

	0	1	2	3	4	5	.....		9
5 training example	0.05	0.02	0.02	0.04	0.05	0.38	0.1	0.04	0.1	0.2
	0.03	0.02	0.02	0.4	0.02	0.2	0.08	0.03	0.1	0.1

Output after we have applied the SoftMax function.


0 1 2 3 4 5 9
0.05 0.02 0.02 0.04 0.05 0.38 0.1 0.04 0.1 0.2
0.03 0.02 0.02 0.4 0.02 0.2 0.08 0.03 0.1 0.1
5 training example
Output after we have applied the Softmax function (H1).
0 1 2 3 4 5 9
0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0 1 0
5 training example
Target Labels

```
# Transpose the train and label data again to revert back to
# shape number of instances * number of classes (see previous slide)
logits = tf.transpose(A1)
labels = tf.transpose(y)


# calculate the cross entropy error for each training example
error = tf.nn.softmax_cross_entropy_with_logits_v2(logits=logits, labels=labels)
loss = tf.reduce_mean(error)


# Point our gradient de                    t the loss node
optimizer = tf.train                    Optimizer(learningRate).minimize(loss)


#
c

#
a
```

The **softmax_cross_entropy_with_logits** function takes in the logits for each neuron for each training example as well as the actual values of y and calculates the **cross entropy loss** for each training example.
We then get the mean of these values. This is our loss.

In this code we plug in our optimizer and point it at the loss node.

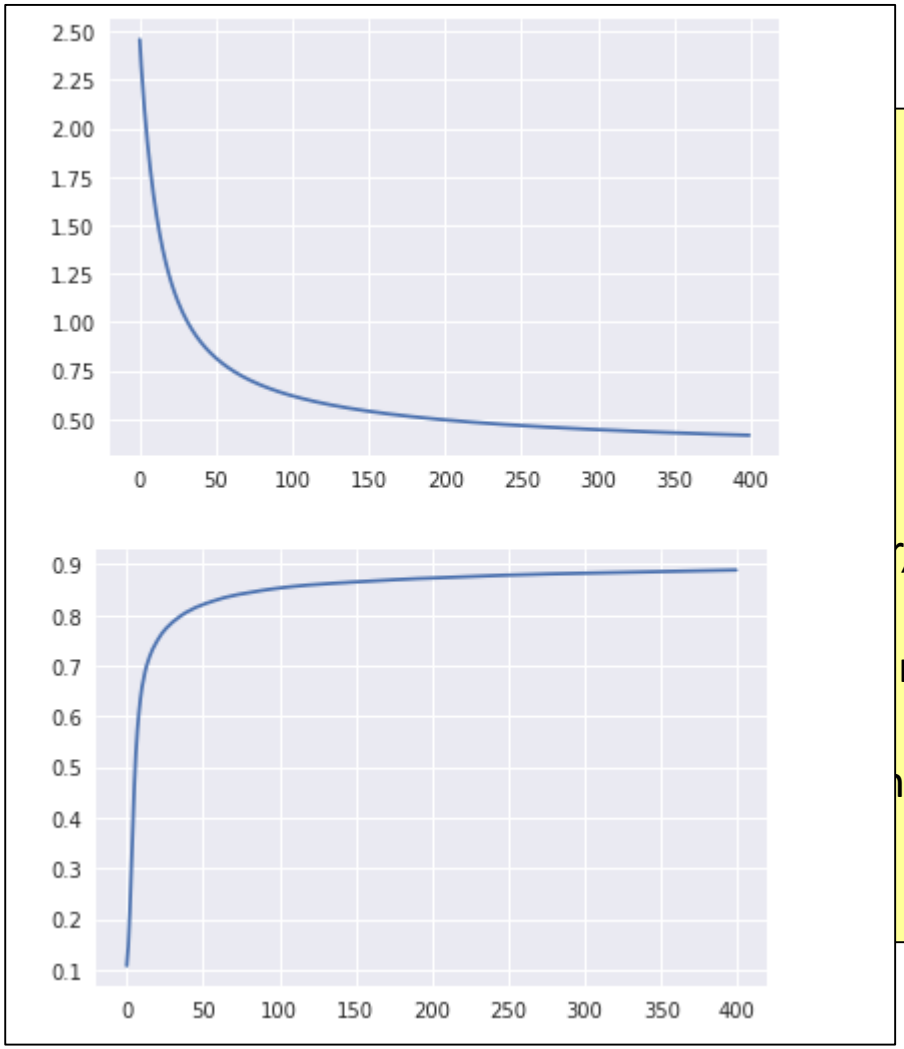
Finally we calculate the accuracy for the current weights and bias.

```
# calculate the cross entropy error for each training example
error = tf.nn.softmax_cross_entropy_with_logits_v2(logits=logits, labels=labels)
loss = tf.reduce_mean(error)


# Point our gradient descent solver at the loss node
optimizer = tf.train.GradientDescentOptimizer(learningRate).minimize(loss)

# Calculate the correct predictions
correct_prediction = tf.equal(tf.argmax(A1), tf.argmax(y))

# Calculate accuracy on the test set
accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
```

We iterated for 400 epochs and the code took **80 seconds** to complete and obtained a reasonable level of accuracy with 0.88. Now consider a case where we have a large neural network, the time taken to train the network would be prohibitive.

```
with tf.Session() as sess:


    sess.run(tf.global_variables_initializer())

    for epoch in range(n_epochs):

        _, currentLoss, acc  = sess.run([optimizer, loss, accuracy], feed_dict={X: X_train, y:
Y_train})
        print ("Iteration ", epoch, " Loss: ", currentLoss, "  Train Acc: ", acc)

    print ("Final Validation Accuracy ", sess.run(accuracy, feed_dict={X: X_test, y: Y_test}))
```

Accuracy 0.88



We iterated for 400 epochs and the code took **80 seconds** to complete and obtained a reasonable level of accuracy with 0.88. Now consider a case where we have a large neural network, the time taken to train the network would be prohibitive.

```
r, loss, accuracy], feed_dict={X: X_train, y:

ntLoss, "  Train Acc: ", acc)

n(accuracy, feed_dict={X: X_test, y: Y_test}))
```

Accuracy:
0.88 Training Data
0.89 Testing Data

Full code is available here.

Implementing a Neural Network in TF.

- In the following Neural Network we are going to look at implementing a neural network for tackling the MNIST dataset.

- The network we will build will consist of:
  - A layer of 300 ReLU neurons
  - A SoftMax layer

- Before, we look at the code let look at the quick reminder of how we tackle this problem in a vectorised manner.

Vectorized Forward Pass for ANNs

$$A^{[1]} = w^{[1]}X + b^{[1]}$$
$$H^{[1]} = act(A^{[1]})$$
$$A^{[2]} = w^{[2]}\,H^{[1]} + b^{[2]}$$
$$H^{[2]} = act(A^{[2]})$$

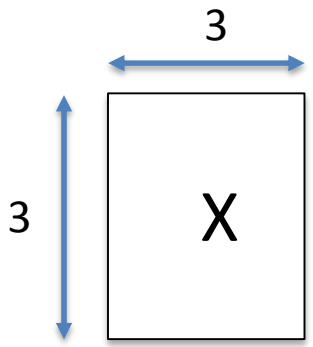
- **(m)** num of training examples
- **(n)** num of features
- **(p)** number of nodes in layer

The first operation we perform in the vectorised code is to multiply the **weight matrix** by the training **data matrix**.

This is a (p*n) matrix multiplied by a (n*m) matrix, which gives us back a (p*m) matrix. Notice rather than just multiplying a single example by the weights associated with each node (as we did previously) we are now multiplying all examples by the weights (vectorising the entire operation).

$$\begin{bmatrix} w_1^{[1](1)} & \cdots & w_1^{[1](n)} \\ \vdots & \ddots & \vdots \\ w_p^{[1](1)} & \cdots & w_p^{[1](n)} \end{bmatrix} \begin{bmatrix} x_1^1 & \cdots & x_1^m \\ \vdots & \ddots & \vdots \\ x_n^1 & \cdots & x_n^m \end{bmatrix} = \begin{bmatrix} t_1^1 & \cdots & t_1^m \\ \vdots & \ddots & \vdots \\ t_p^1 & \cdots & t_p^m \end{bmatrix}$$

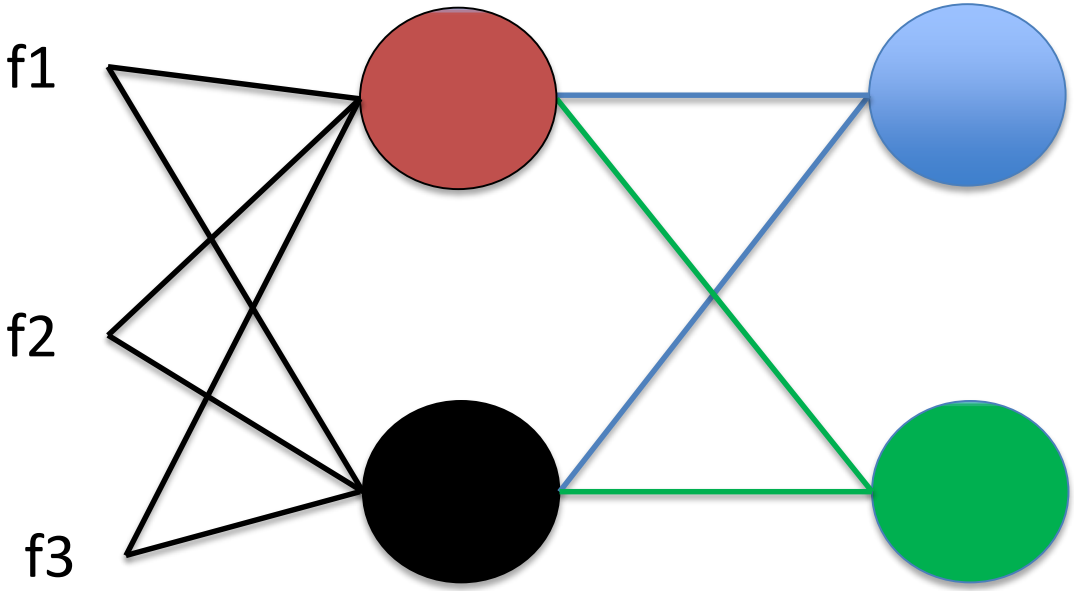
Let's look at a problem where we have three training examples that we want to push through the following neural network.



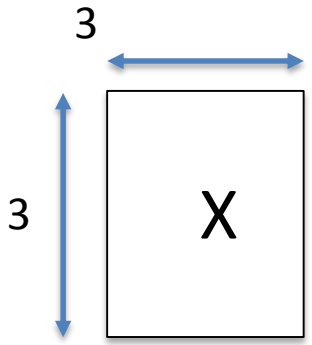
Each training example is defined by 3 feature values

The input training data will be 3*3.

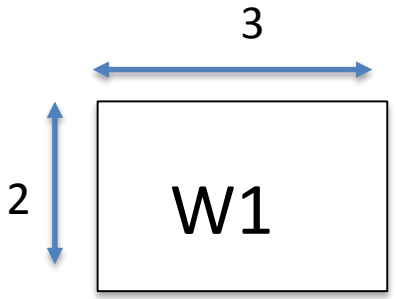
Each **training example is one column** and each of the **features are rows**.

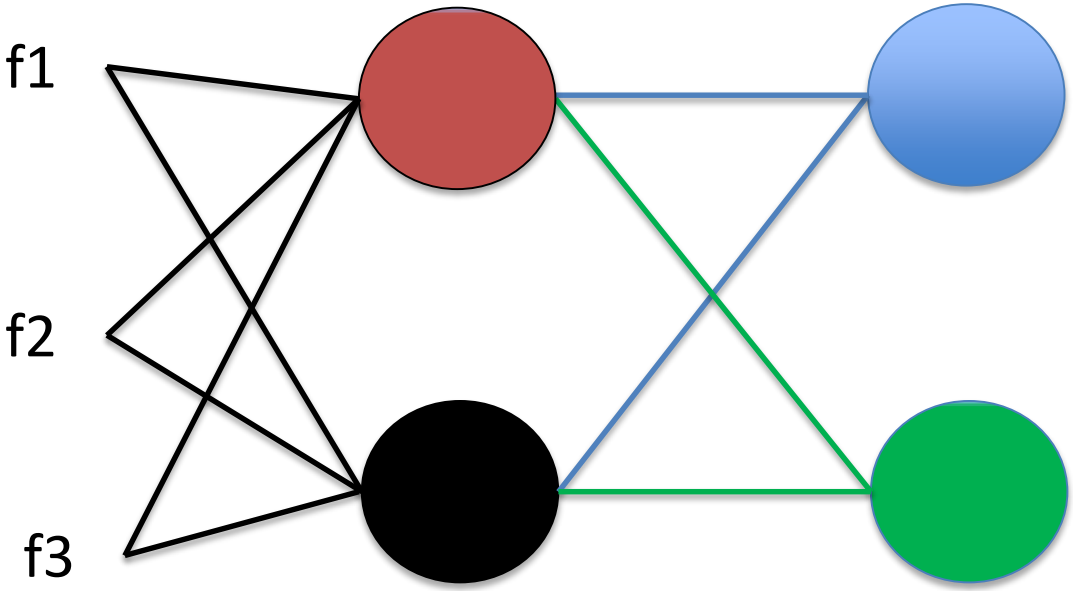


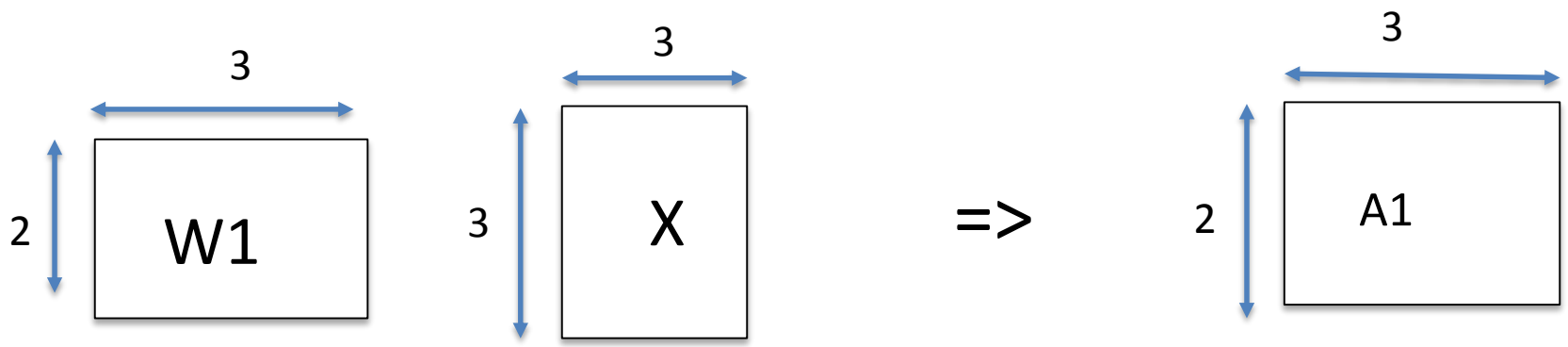
Let's look at a problem where we have three training examples that we want to push through the following neural network. The input training data will be 3*3. Each **training example is one column** and each of the **features are rows**.



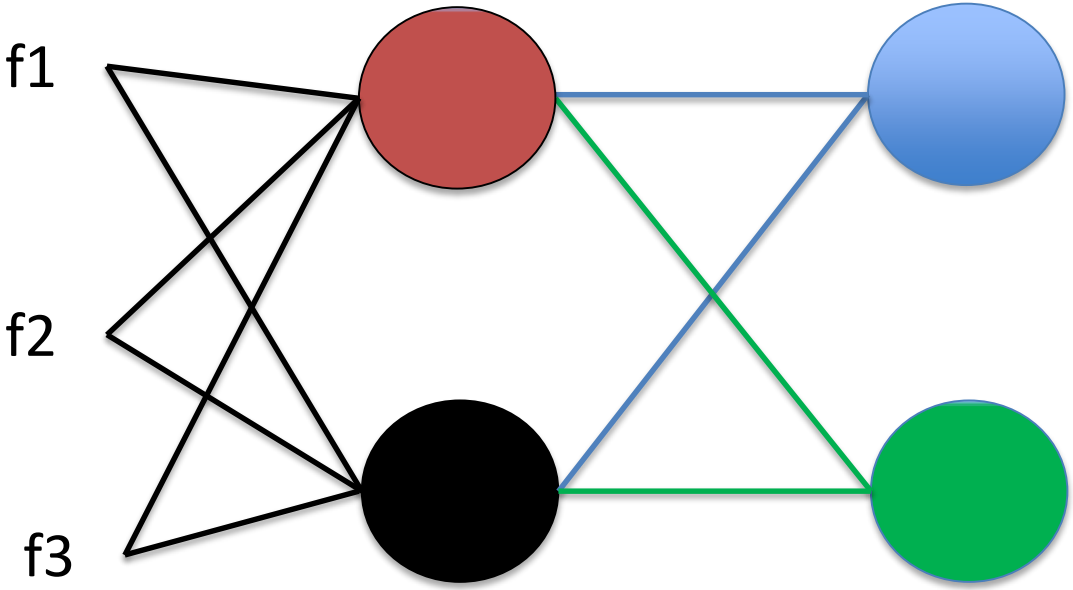
The weights for the first layer are 2*3. Each row represents the weights for a specific neuron.

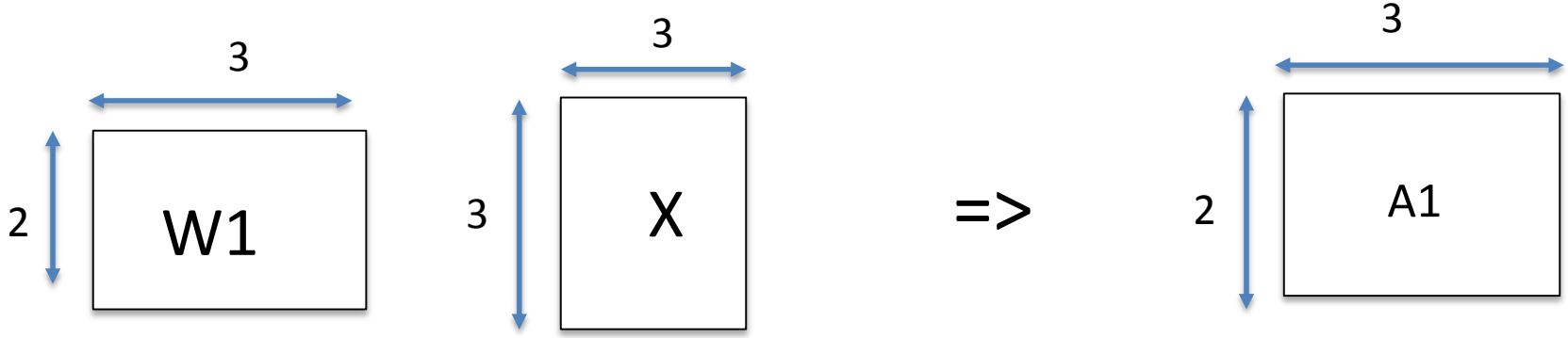




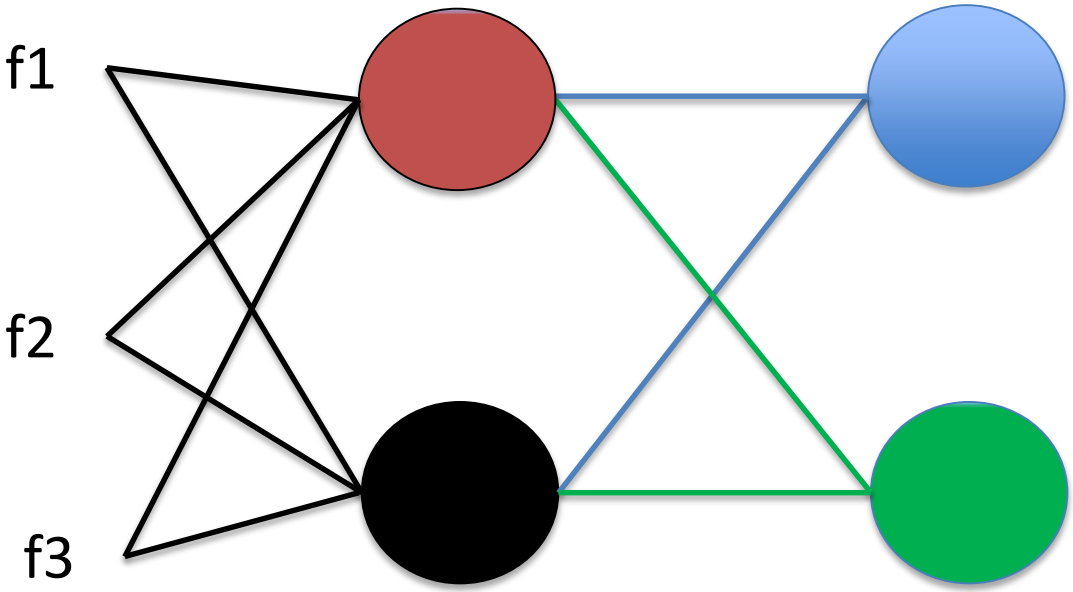


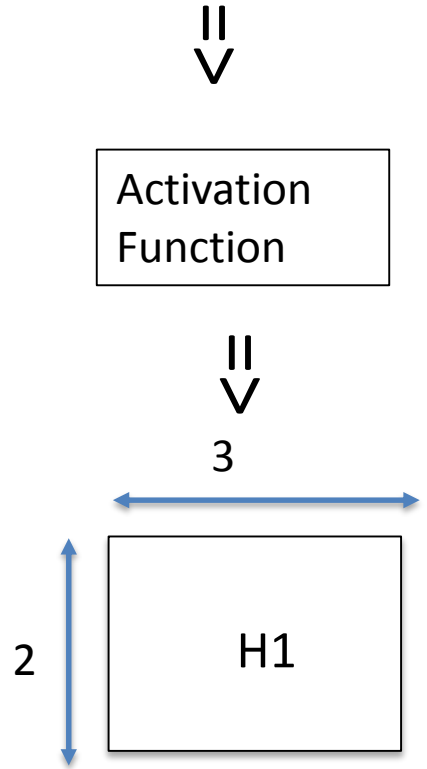
We do our usual dot **matrix multiplication** and it provides us with the pre-activation values for the first layer of neurons for each training example (again note we just omit adding the bias for simplicity).
Each column in the output matrix above relates to one training instance and the two values specify the output from the two neurons.



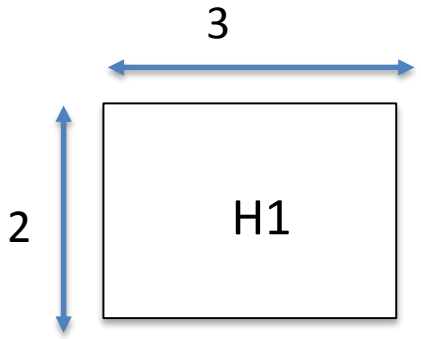


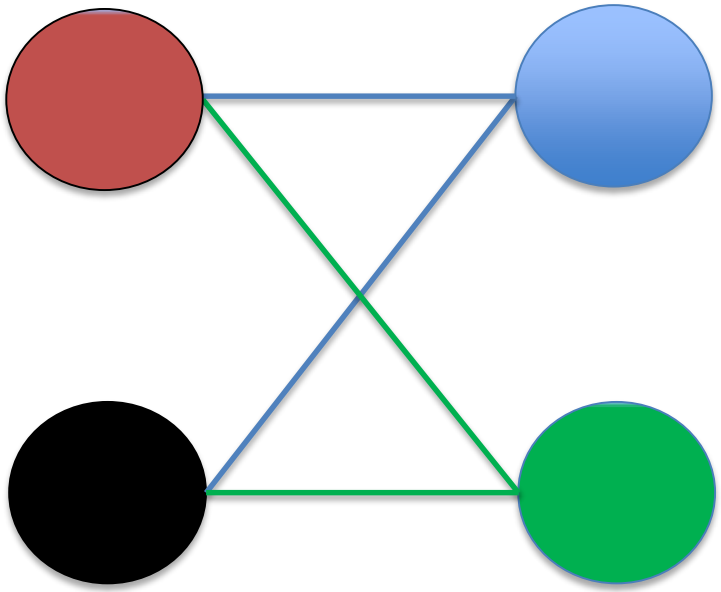
The pre-activation values in A1 are piped through an activation function and the output is H1, which in turn flows into the next layer of neurons.





Now lets look at layer two of the network. As we saw the first layer produced a matrix (H1) containing 2 rows and 3 columns (number of neurons by number of training examples)





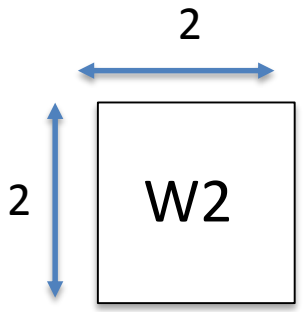
For example, H1 might look like the following

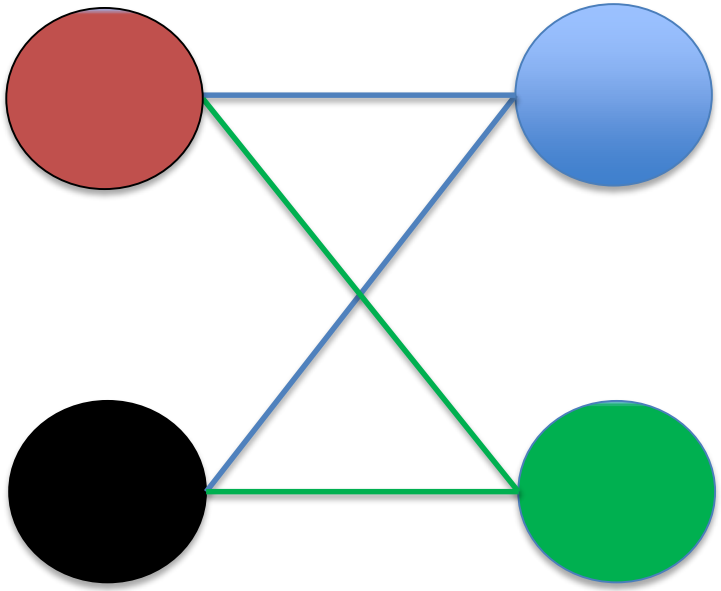
0.23	-0.41	0.12
0.32	0.98	-0.43

Each row corresponds to a output of a neuron in layer 1

What will be the dimensions of the weights for layer 2?

The weights matrix for the second layer in this example will be a 2 by 2 matrix (number of neurons, number of inputs).

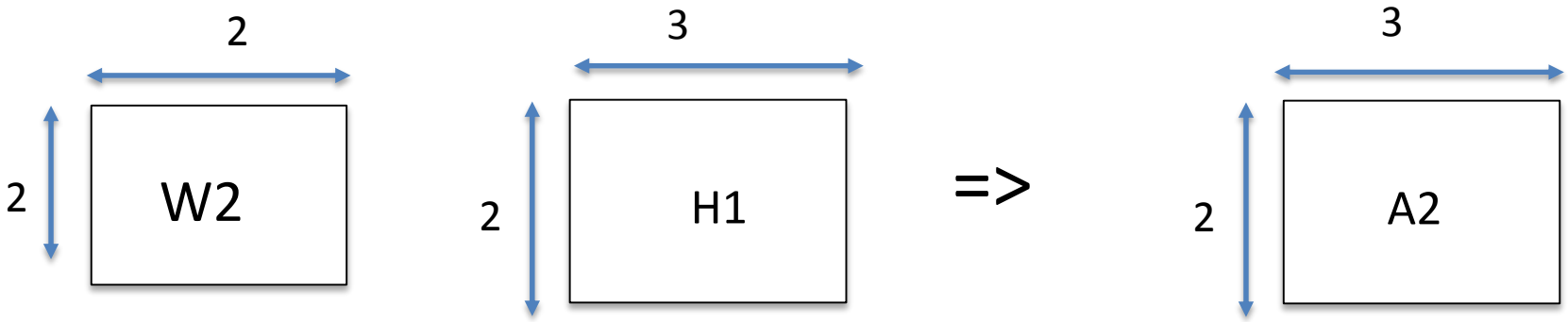


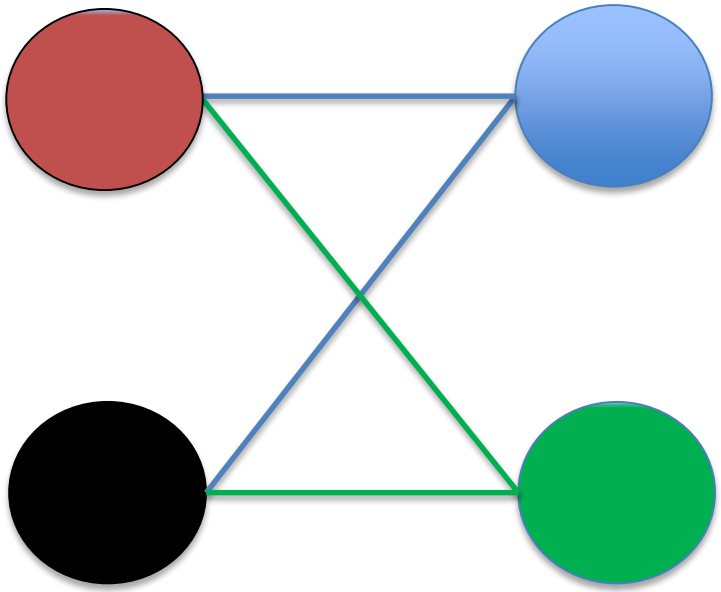


For example, W might look like the following

0.02	0.004
0.1	-0.2

Each **row** corresponds to the weights coming in to each **neuron.**

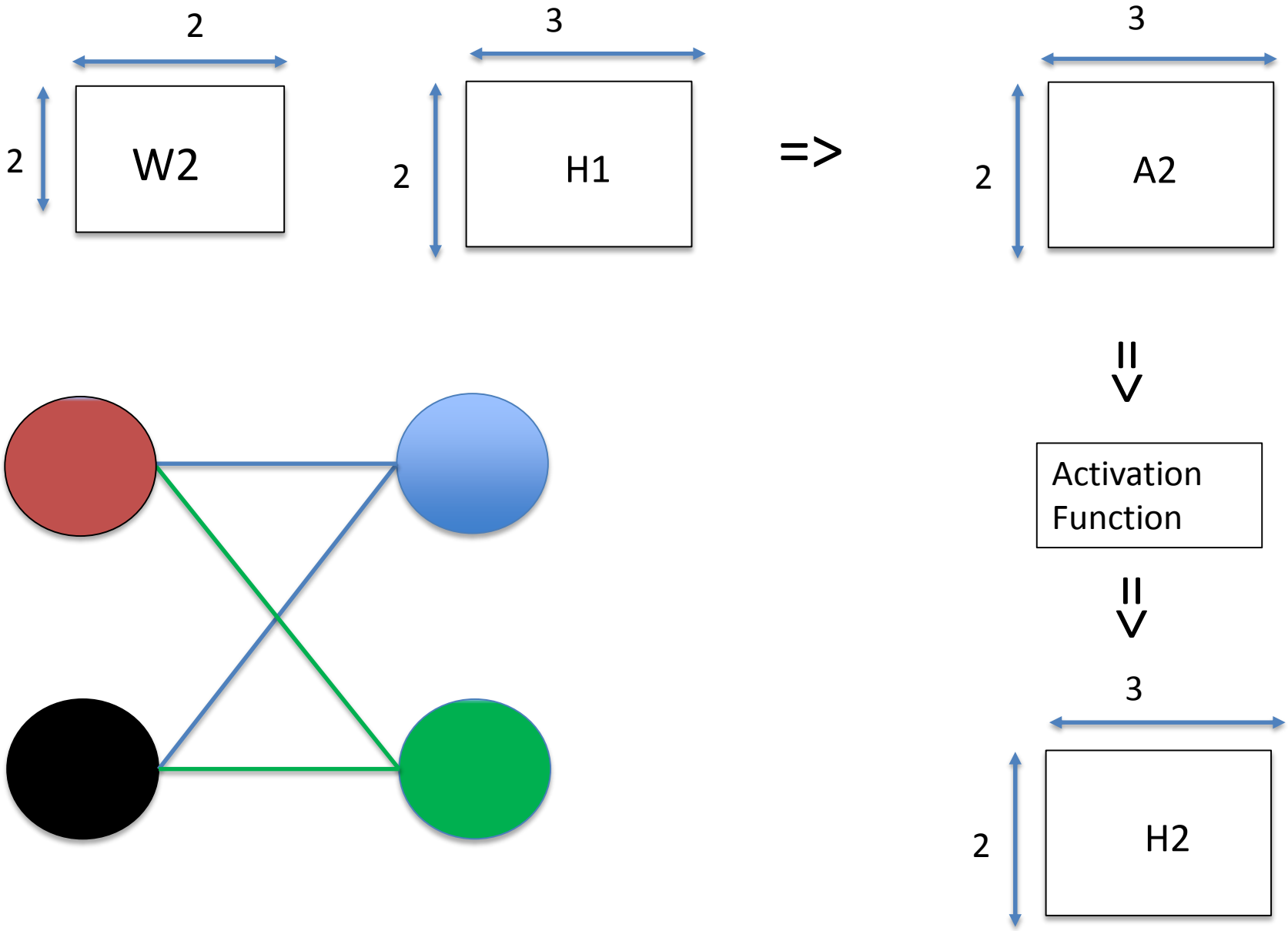




0.02	0.004
0.1	-0.2

0.23	-0.41	0.12
0.32	0.98	-0.43

- Each row in W2 corresponds to the weights for single neuron.
- Each column in H1 corresponds to a output for a single training example.
- Therefore, we multiple the output from a single training instance from layer 1 by the weights of a neuron for layer 2.


2
2
W2
3
2
H1
=>
3
2
A2
Activation
Function
3
2
H2

- The network we will build will consist of:
  - A layer of 300 ReLU neurons
  - A SoftMax layer

Here we load the MNIST dataset. This code is the same as the Softmax example.

```
import tensorflow as tf
import numpy as np

tf.reset_default_graph()

# Read in dataset
mnist = tf.keras.datasets.mnist
(X_train, Y_train), (X_test, Y_test) = mnist.load_data()

# Reshape training dataset so that the features are flattened
X_train = X_train.reshape(X_train.shape[0], -1).astype('float32')
X_test = X_test.reshape(X_test.shape[0], -1).astype('float32')

# Normalize training data
X_train = X_train/255.0
X_test = X_test/255.0

X_train = X_train.T
X_test = X_test.T
```

```
# continued from previous slide

# Convert labels to one-hot-encoded
number_of_classes = 10
Y_train = tf.keras.utils.to_categorical(Y_train, number_of_classes)
Y_test = tf.keras.utils.to_categorical(Y_test, number_of_classes)

# Transpose train labels so that dimensions
# are number of classes * number of instances
Y_train = Y_train.T
Y_test = Y_test.T

print ("Data extracted and reshaped: ")
print (X_train.shape, Y_train.shape, X_test.shape, Y_test.shape)
```

Again the code we have included here is exactly the same as for the Softmax example.

```
# Model Parameters
n_inputs = 784
n_hidden = 300
n_outputs = 10
learningRate = 0.1
n_epochs = 40
```

Create a neural network with a layer of 300 neurons followed by a softmax layer containing 10 neurons. Notice the dimensions of the placeholders is the same as we previously used in softmax.

```
# Placeholders for the traing and label data
X = tf.placeholder(tf.float32, [n_inputs, None], name='image')
y = tf.placeholder(tf.int32, [n_outputs, None], name='label')

# Create weight and bias matrices (variables) for each layer of our network

W1 = tf.get_variable("W1", [n_hidden, n_inputs],initializer =
tf.glorot_uniform_initializer(seed=1) )
b1 = tf.get_variable("b1", [n_hidden, 1], initializer = tf.zeros_initializer())

W2 = tf.get_variable("W2", [n_outputs, n_hidden], initializer =
tf.glorot_uniform_initializer(seed=1))
b2 = tf.get_variable("b2", [n_outputs, 1], initializer = tf.zeros_initializer())
```

```
# Model Parameters
n_inputs = 784
n_hidden = 300
n_outputs = 10
learningRate = 0.1
n_epochs = 40
```

Notice we can create a set of weights and bias values for the first layer (W1 and b1) as well as for the second layer (W2 and b2). It is important to notice that the number of weight fed into the second layer is dependent on the number of neurons in the second layer.

```
# Placeholders for the traing and label data
X = tf.placeholder(tf.float32, [n_inputs, None], name='image')
y = tf.placeholder(tf.int32, [n_outputs, None], name='label')

# Create weight and bias matrices (variables) for each layer of our network

W1 = tf.get_variable("W1", [n_hidden, n_inputs],initializer =
tf.glorot_uniform_initializer(seed=1) )
b1 = tf.get_variable("b1", [n_hidden, 1], initializer = tf.zeros_initializer())

W2 = tf.get_variable("W2", [n_outputs, n_hidden], initializer =
tf.glorot_uniform_initializer(seed=1))
b2 = tf.get_variable("b2", [n_outputs, 1], initializer = tf.zeros_initializer())
```

```
# Push feature data through first layer of NN
A1 = tf.add(tf.matmul(W1, X), b1)
H1  = tf.nn.relu(A1)

# Calculate preactivation value for third layer
A2 = tf.add(tf.matmul(W2, H1), b2)

# Revert back to shape number of instances * number of classes
logits = tf.transpose(A2)
labels = tf.transpose(y)

# calculate the cross entropy
error = tf.nn.softmax_cross_e
loss = tf.reduce_mean(error)

# Point our gradient descent solver at the loss node
optimizer = tf.train.GradientDescentOptimizer(learningRate).minimize(loss)


# Calculate the correct predictions
correct_prediction = tf.equal(tf.argmax(A2), tf.argmax(y))

# Calculate accuracy on the test set
accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
```

We push the training data through the first layer (both matrix multiplication and activation function). We take the output of the first layer and push it through the second layer and collection the pre-activation values.

```
# Push feature data through first layer of NN
A1 = tf.add(tf.matmul(W1, X
H1  = tf.nn.relu(A1)

# Calculate preactivation val
A2 = tf.add(tf.matmul(W2, H

# Revert back to shape numbe
logits = tf.transpose(A2)
labels = tf.transpose(y)

# calculate the cross entropy error for each training example
error = tf.nn.softmax_cross_entropy_with_logits_v2(logits=logits, labels=labels)
loss = tf.reduce_mean(error)

# Point our gradient descent solver at the loss node
optimizer = tf.train.GradientDescentOptimizer(learningRate).minimize(loss)


# Calculate the correct predictions
correct_prediction = tf.equal(tf.argmax(A2), tf.argmax(y))

# Calculate accuracy on the test set
accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
```

Next we calculate the softmax value and the cross entropy error. We plug in our gradient descent optimizer and then we calculate the accuracy.

```
with tf.Session() as sess:


    sess.run(tf.global_variables_initializer())

    for epoch in range(n_epochs):

        _, currentLoss, acc  = sess.run([optimizer, loss, accuracy], feed_dict={X: X_train, y: Y_train})
        print (currentLoss, "  ", acc)

    print ("Final Validation Accuracy ", sess.run(accuracy, feed_dict={X: X_test, y: Y_test}))
```

The full code for the example above is available here.