DIP: Tensor Flow _ Neural Network

Presenter: Dr. Ha Viet Uyen Synh.

PART A

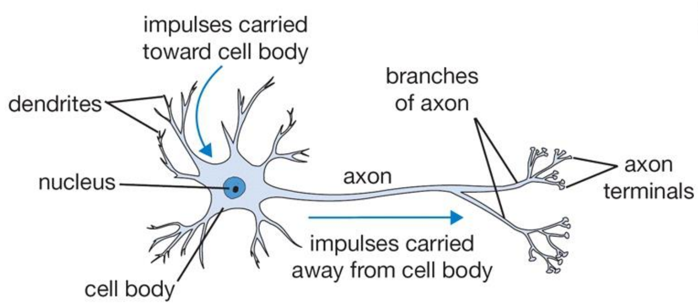
PERCEPTRON LEARNING ALGORITHM

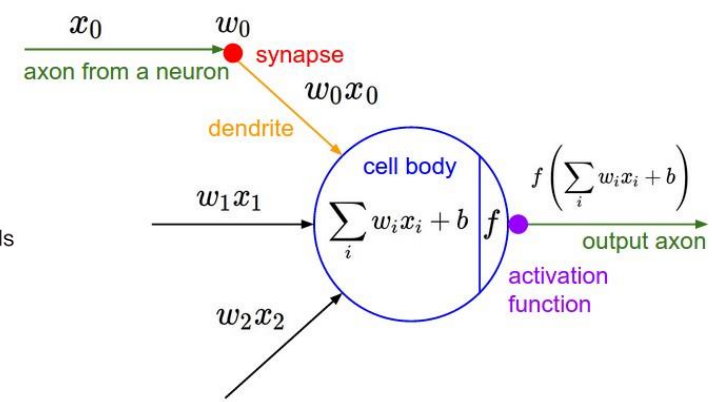
How Deep Learning Works?

In an attempt to re-engineer a human brain, Deep Learning studies the basic unit of a brain called a brain cell or a neuron.

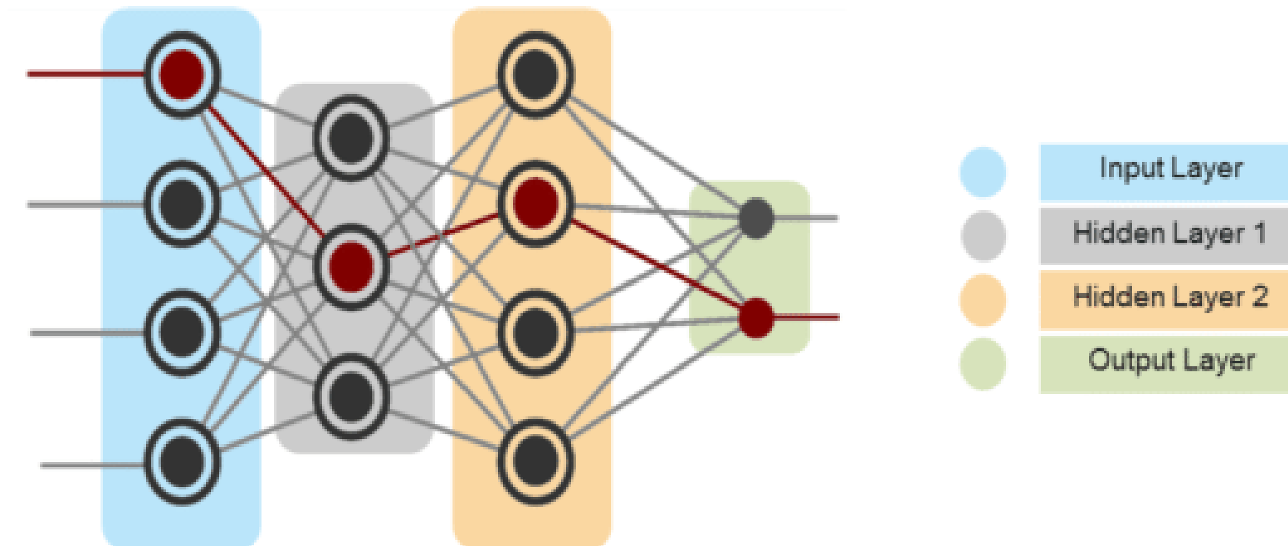
A perceptron receives multiple inputs, applies various transformations and functions and provides an output.

As we know that our brain consists of multiple connected neurons called neural network, we can also have a network of artificial neurons called perceptrons to form a Deep neural network



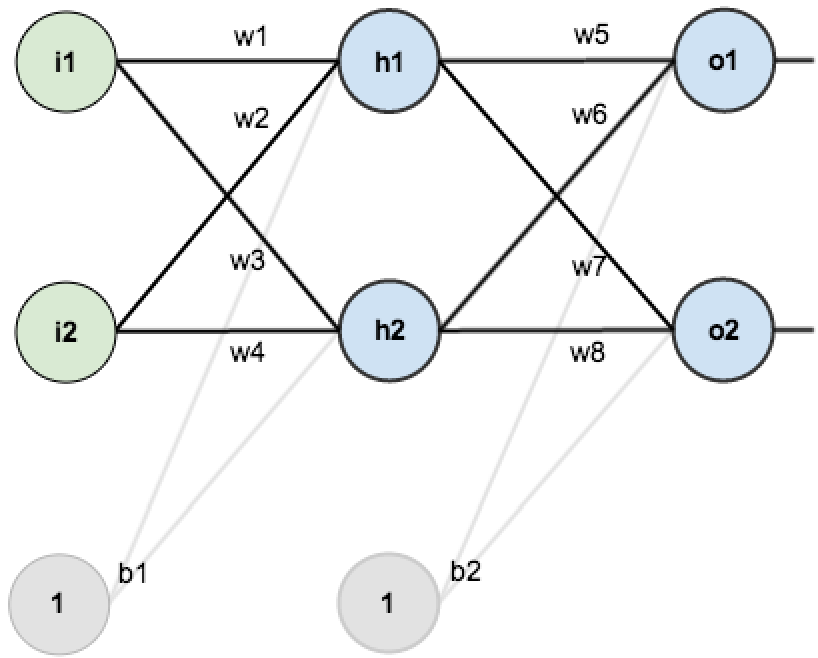


Deep neural network



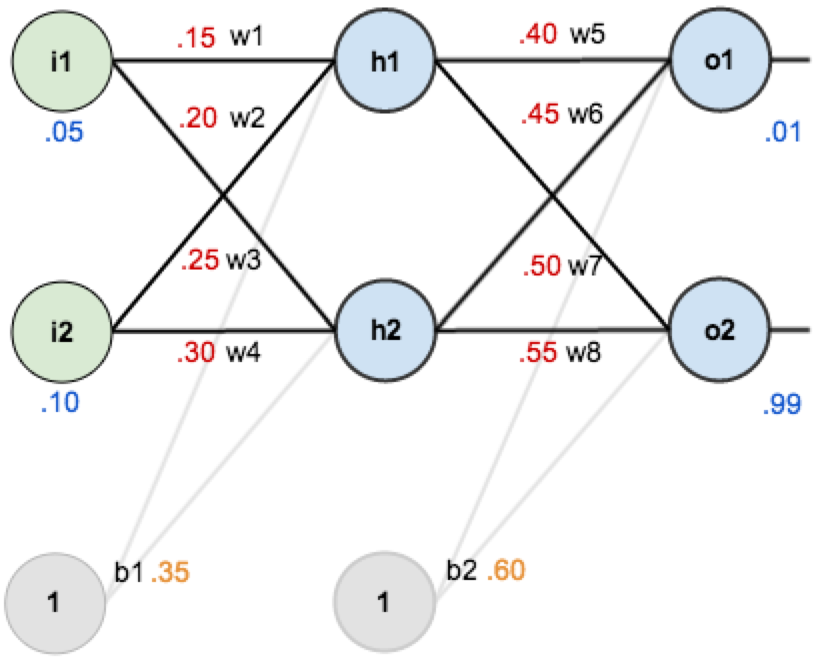
Example

We're going to use a neural network with two inputs, two hidden neurons, two output neurons. Additionally, the hidden and output neurons will include a bias.



Example

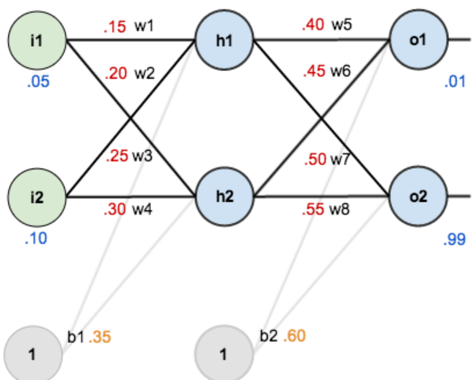
In order to have some numbers to work with, here are the initial weights, the biases, and training



Example

The Forward Pass

We figure out the total net input to each hidden layer neuron, squash the total net input using an activation function (here we use the logistic function), then repeat the process with the output layer neurons.



$$net_{h1} = w_1 * i_1 + w_2 * i_2 + b_1 * 1$$

$$net_{h1} = 0.15 * 0.05 + 0.2 * 0.1 + 0.35 * 1 = 0.3775$$

$$out_{h1} = \frac{1}{1+e^{-net_{h1}}} = \frac{1}{1+e^{-0.3775}} = 0.593269992$$

$$out_{h2} = 0.596884378$$

$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$net_{o1} = 0.4 * 0.593269992 + 0.45 * 0.596884378 + 0.6 * 1 = 1.105905967$$

$$out_{o1} = \frac{1}{1+e^{-net_{o1}}} = \frac{1}{1+e^{-1.105905967}} = 0.75136507$$

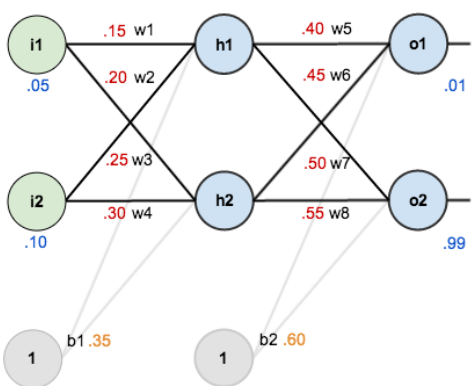
$$out_{o2} = 0.772928465$$

Example

Calculating the Total Error

We can now calculate the error for each output neuron using the squared error function and sum them to get the total error:

$$E_{total} = \sum \frac{1}{2}(target - output)^2$$



The target output for o_1 is 0.01 but the neural network output 0.75136507, therefore its error is:

$$E_{o1} = \frac{1}{2}(target_{o1} - out_{o1})^2 = \frac{1}{2}(0.01 - 0.75136507)^2 = 0.274811083$$

$$E_{o2} = 0.023560026$$

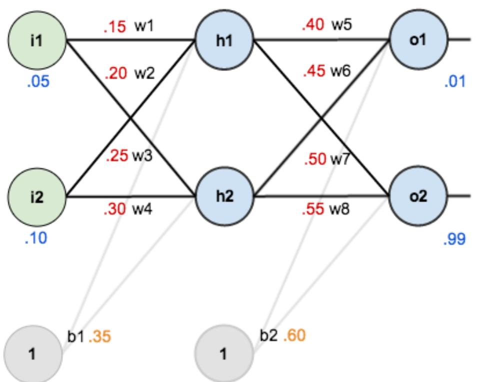
The total error for the neural network is the sum of these errors:

$$E_{total} = E_{o1} + E_{o2} = 0.274811083 + 0.023560026 = 0.298371109$$

Example

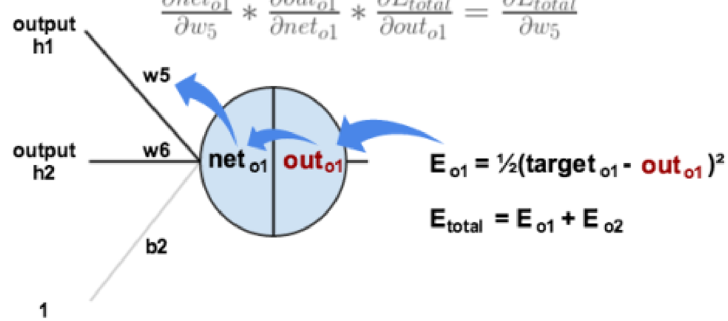
The Backwards Pass

Our goal with backpropagation is to update each of the weights in the network so that they cause the actual output to be closer the target output, thereby minimizing the error for each output neuron and the network as a whole.



Output Layer

$$\frac{\partial net_{o1}}{\partial w_5} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial E_{total}}{\partial out_{o1}} = \frac{\partial E_{total}}{\partial w_5}$$

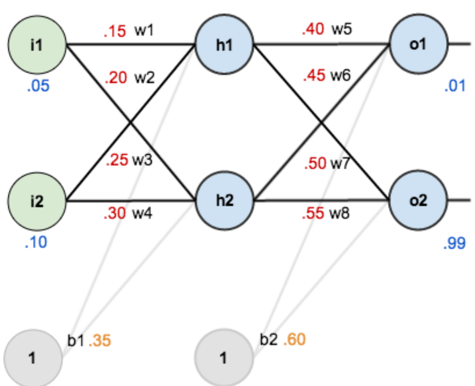


First, how much does the total error change with respect to the output?

$$E_{total} = \frac{1}{2}(target_{o1} - out_{o1})^2 + \frac{1}{2}(target_{o2} - out_{o2})^2$$

$$\frac{\partial E_{total}}{\partial out_{o1}} = 2 * \frac{1}{2}(target_{o1} - out_{o1})^{2-1} * -1 + 0$$

$$\frac{\partial E_{total}}{\partial out_{o1}} = -(target_{o1} - out_{o1}) = -(0.01 - 0.75136507) = 0.74136507$$



Example

Next, how much does the output of o_1 change with respect to its total net input?

$$out_{o1} = \frac{1}{1+e^{-net_{o1}}}$$

$$\frac{\partial out_{o1}}{\partial net_{o1}} = out_{o1}(1 - out_{o1}) = 0.75136507(1 - 0.75136507) = 0.186815602$$

Finally, how much does the total net input of o_1 change with respect to w_5?

$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$\frac{\partial net_{o1}}{\partial w_5} = 1 * out_{h1} * w_5^{(1-1)} + 0 + 0 = out_{h1} = 0.593269992$$

Putting it all together:

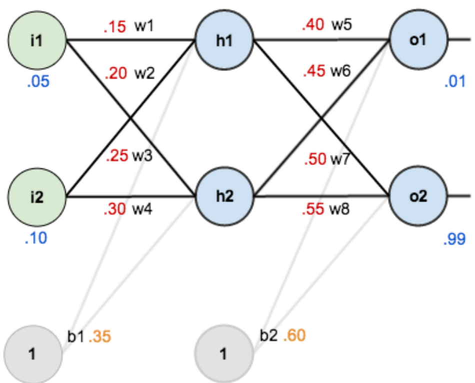
$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$$

$$\frac{\partial E_{total}}{\partial w_5} = 0.74136507 * 0.186815602 * 0.593269992 = 0.082167041$$

To decrease the error, we then subtract this value from the current weight (optionally multiplied by some learning rate, eta, which we'll set to 0.5):

$$w_5^+ = w_5 - \eta * \frac{\partial E_{total}}{\partial w_5} = 0.4 - 0.5 * 0.082167041 = 0.35891648$$

Example



We can repeat this process to get the new weights w6, w7, and w8:

$$w_6^+ = 0.408666186$$

$$w_7^+ = 0.511301270$$

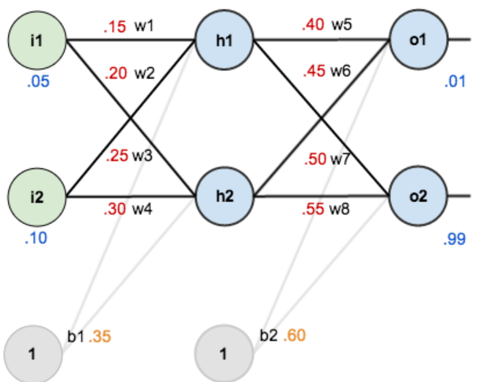
$$w_8^+ = 0.561370121$$

We perform the actual updates in the neural network after we have the new weights leading into the hidden layer neurons (ie, we use the original weights, not the updated weights, when we continue the backpropagation algorithm below).

Example

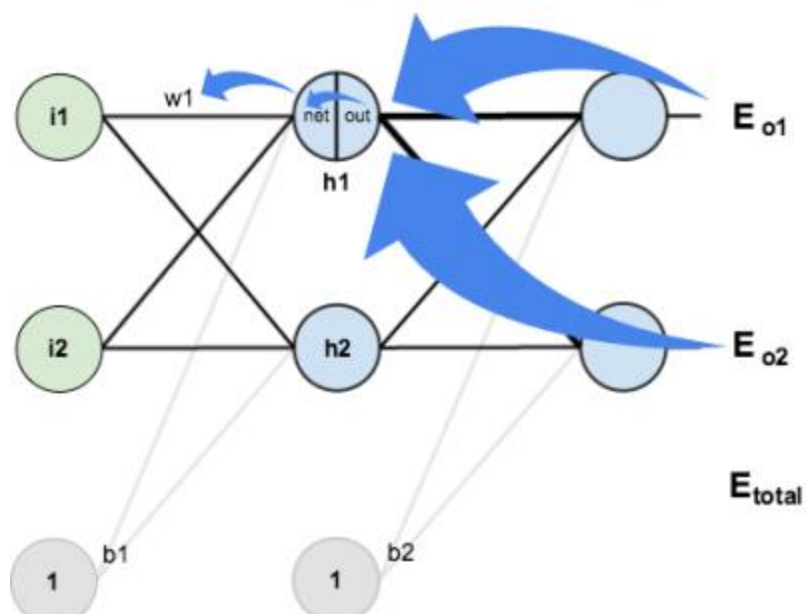
Hidden Layer

Next, we’ll continue the backwards pass by calculating new values for w_1, w_2, w_3, and w_4.



$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$



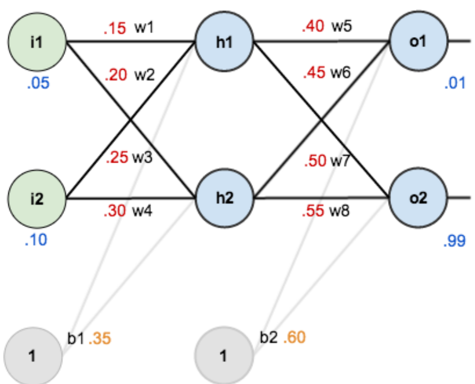
$E_{total} = E_{o1} + E_{o2}$

Example

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$

$$\frac{\partial E_{o1}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial out_{h1}}$$

$$\frac{\partial E_{o1}}{\partial net_{o1}} = \frac{\partial E_{o1}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} = 0.74136507 * 0.186815602 = 0.138498562$$



$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$\frac{\partial net_{o1}}{\partial out_{h1}} = w_5 = 0.40$$

$$\frac{\partial E_{o1}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial out_{h1}} = 0.138498562 * 0.40 = 0.055399425$$

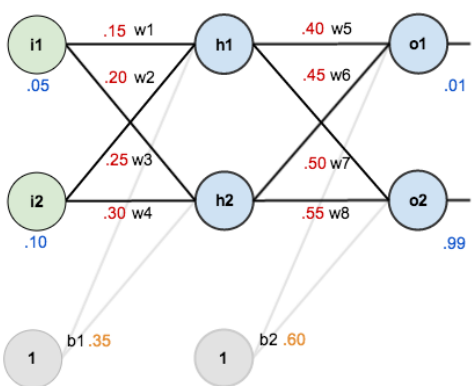
$$\frac{\partial E_{o2}}{\partial out_{h1}} = -0.019049119$$

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}} = 0.055399425 + -0.019049119 = 0.036350306$$

$$out_{h1} = \frac{1}{1+e^{-net_{h1}}}$$

$$\frac{\partial out_{h1}}{\partial net_{h1}} = out_{h1}(1 - out_{h1}) = 0.59326999(1 - 0.59326999) = 0.241300709$$

Example



We calculate the partial derivative of the total net input to h_1 with respect to w_1 the same as we did for the output neuron:

$$net_{h1} = w_1 * i_1 + w_2 * i_2 + b_1 * 1$$

$$\frac{\partial net_{h1}}{\partial w_1} = i_1 = 0.05$$

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\frac{\partial E_{total}}{\partial w_1} = 0.036350306 * 0.241300709 * 0.05 = 0.000438568$$

We can now update w_1:

$$w_1^+ = w_1 - \eta * \frac{\partial E_{total}}{\partial w_1} = 0.15 - 0.5 * 0.000438568 = 0.149780716$$

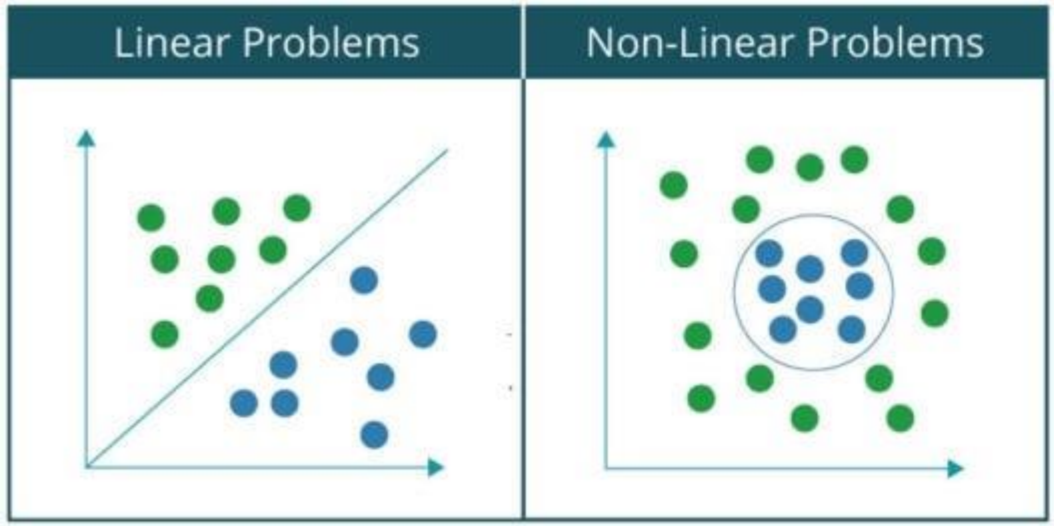
Repeating this for w_2, w_3, and w_4

$$w_2^+ = 0.19956143$$

$$w_3^+ = 0.24975114$$

$$w_4^+ = 0.29950229$$

Types of Classification Problems



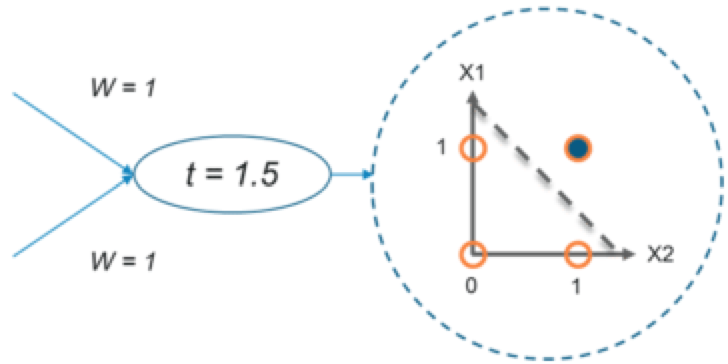
Perceptron

Therefore, a perceptron can be used as a separator or a decision line that divides the input set of AND Gate, into two classes:

Class 1: Inputs having output as 0 that lies below the decision line.

Class 2: Inputs having output as 1 that lies above the decision line or separator.

x1	x2	y
0	0	0
0	1	0
1	0	0
1	1	1



$$\underbrace{f(x)}_{\text{Transfer Function}} = \underbrace{w}_{\text{Weight Vector}} . \underbrace{x}_{\text{Input Vector}} + \underbrace{b}_{\text{Bias}}$$

Implementation of AND Gate

1. Import all the required library

```
#import required library
import tensorflow as tf
```

2. Define Variables for Input and Output

```
#input1, input2 and bias
train_in = [
    [1., 1.,1],
    [1., 0,1],
    [0, 1.,1],
    [0, 0,1]]

#output
train_out = [
[1.],
[0],
[0],
[0]]
```

Implementation of AND Gate

3. Define Weight Variable

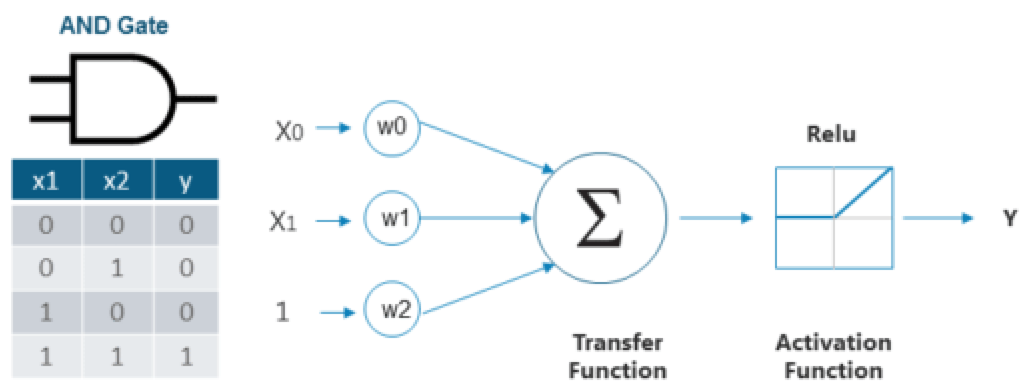
```
#weight variable initialized with random values using random_normal()
w = tf.Variable(tf.random_normal([3, 1], seed=12))
```

4. Define placeholders for Input and Output

```
#Placeholder for input and output
x = tf.placeholder(tf.float32,[None,3])
y = tf.placeholder(tf.float32,[None,1])
```

5. Calculate Output and Activation Function

```
#calculate output
output = tf.nn.relu(tf.matmul(x, w))
```



Implementation of AND Gate

6. Calculate the Cost or Error

```
#Mean Squared Loss or Error
loss = tf.reduce_sum(tf.square(output - y))
```

7. Minimize Error

```
#Minimize loss using GradientDescentOptimizer with a learning rate of 0.01
optimizer = tf.train.GradientDescentOptimizer(0.01)
train = optimizer.minimize(loss)
```

8. Initialize all the variables

```
#Initialize all the global variables
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)
```

9. Training Perceptron in Iterations

```
#Compute output and cost w.r.t to input vector
sess.run(train, {x:train_in,y:train_out})
cost = sess.run(loss,feed_dict={x:train_in,y:train_out})
print('Epoch--',i,'--loss--',cost)
```

Implementation of AND Gate

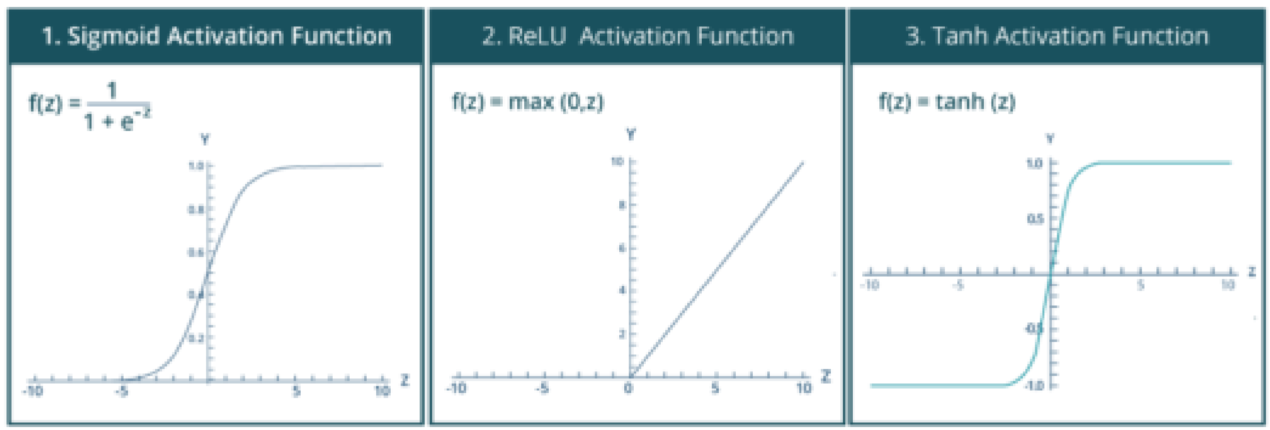
10. Output

```
Epoch-- 995 --loss-- 0.000373111
Epoch-- 996 --loss-- 0.000370556
Epoch-- 997 --loss-- 0.000368017
Epoch-- 998 --loss-- 0.000365496
Epoch-- 999 --loss-- 0.000362991
Final Weights: [[ 0.97700185]
 [ 0.97700185]
 [-0.96747559]]
Final Output: [[ 0.9865281 ]
 [ 0.00952625]
 [ 0.00952625]
 [ 0.        ]]
```

Activation Functions

$$\text{logit, } z = x_1.w_1 + \dots + x_n.w_n$$

$$\underbrace{y}_{\textit{Final Output}} = \underbrace{f(z)}_{\textit{Activation Function}}$$



Use Case

In this use case, we have been provided with a SONAR data set which contains the data about 208 patterns obtained by bouncing sonar signals off a metal cylinder (naval mine) and a rock at various angles and under various conditions. So, our goal is to build a model that can predict whether the object is a naval mine or rock based on our data set.

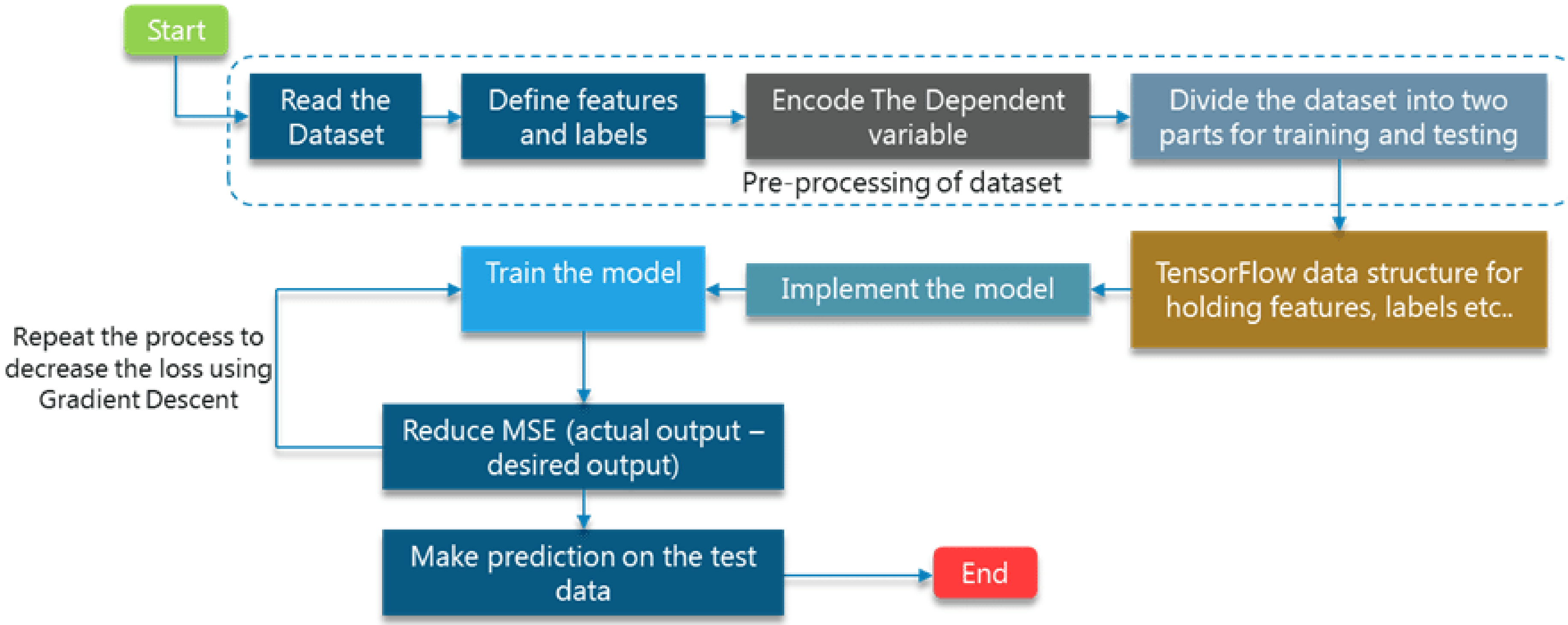
SONAR data set

60 columns representing different features obtained by bouncing sonar signals off a metal cylinder and a rock

0.0453, 0.0523, 0.0843, 0.0689, 0.1183, 0.2583, 0.2156, 0.3481, 0.3337, 0.2872, 0.4918, 0.6552, 0.6919, 0.7797, 0.7464, 0.9444, 1, 0.8874, 0.8024, 0.7818, 0.5212, 0.4052, 0.3957, 0.3914, 0.325, 0.32, 0.3271, 0.2767, 0.4423, 0.2028, 0.3788, 0.2947, 0.1984, 0.2341, 0.1306, 0.4182, 0.3835, 0.1057, 0.184, 0.197, 0.1674, 0.0583, 0.1401, 0.1628, 0.0621, 0.0203, 0.053, 0.0742, 0.0409, 0.0061, 0.0125, 0.0084, 0.0089, 0.0048, 0.0094, 0.0191, 0.014, 0.0049, 0.0052, 0.0044, R

Label associated with each record contains the letter **"R"** if the object is a rock and **"M"** if it is a mine

The model using TensorFlow



Implementation of SONAR data classification

1. Import all the required Libraries:

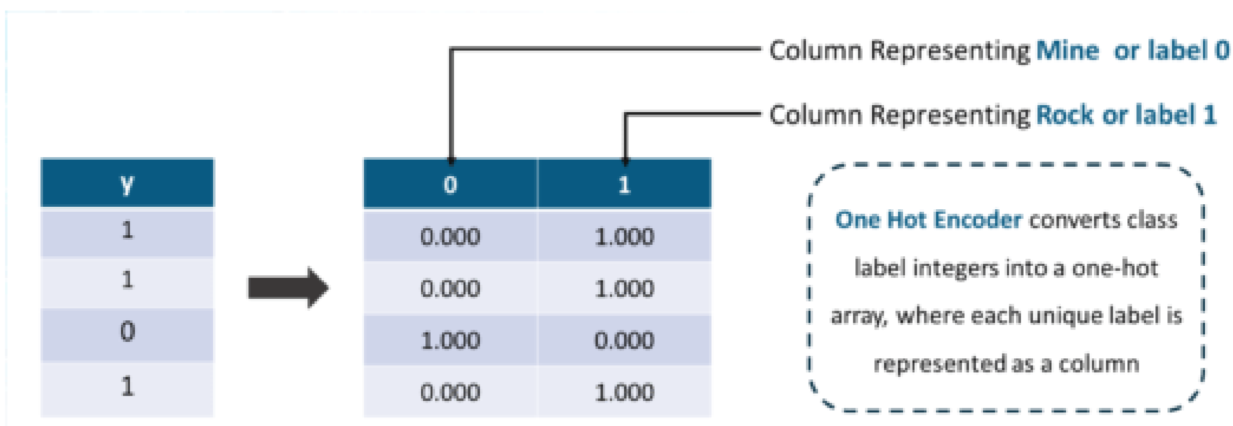
```
#import the required libraries
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.utils import  shuffle
from sklearn.model_selection import train_test_split
```

2. Read and Pre-process the data set:

```
#Read the sonar dataset
df = pd.read_csv("sonar.csv")
print(len(df.columns))
X = df[df.columns[0:60]].values
y = df[df.columns[60]]
#encode the dependent variable as it has two categorical values
encoder = LabelEncoder()
encoder.fit(y)
y = encoder.transform(y)
Y = one_hot_encode(y)
```

3. Function for One Hot Encoder:

```python
#function for applying one_hot_encoder
def one_hot_encode(labels):
  n_labels = len(labels)
  n_unique_labels = len(np.unique(labels))
  one_hot_encode = np.zeros((n_labels,n_unique_labels))
  one_hot_encode[np.arange(n_labels), labels] = 1
  return one_hot_encode
```



4. Dividing data set into Training and Test Subset

```
#Divide the data in training and test subset
#use train_test_split() function from the sklearn library for dividing the dataset
X,Y = shuffle(X,Y,random_state=1)
train_x,test_x,train_y,test_y = train_test_split(X,Y,test_size=0.20, random_state=42)
```

5. Define Variables and Placeholders

```
#define and initialize the variables to work with the tensors
learning_rate = 0.1
training_epochs = 1000

#Array to store cost obtained in each epoch
cost_history = np.empty(shape=[1],dtype=float)

n_dim = X.shape[1]
n_class = 2

x = tf.placeholder(tf.float32,[None,n_dim])
W = tf.Variable(tf.zeros([n_dim,n_class]))
b = tf.Variable(tf.zeros([n_class]))

#initialize all variables.
init = tf.global_variables_initializer()
```

6. Calculate the Cost or Error

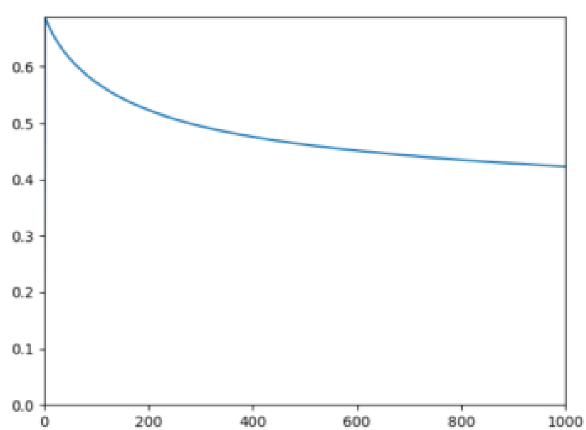
```
y_ = tf.placeholder(tf.float32,[None,n_class])
y = tf.nn.softmax(tf.matmul(x, W)+ b)
cost_function = tf.reduce_mean(-tf.reduce_sum((y_ * tf.log(y)),
reduction_indices=[1]))
training_step =
tf.train.GradientDescentOptimizer(learning_rate).minimize(cost_fun
ction)
```

7. Training the Perceptron Model in Successive Epochs

```
#initialize the session
sess = tf.Session()
sess.run(init)
mse_history = []

#calculate the cost for each epoch
for epoch in range(training_epochs):
    sess.run(training_step,feed_dict={x:train_x,y_:train_y})
    cost = sess.run(cost_function,feed_dict={x: train_x,y_: train_y})
    cost_history = np.append(cost_history,cost)
    print('epoch : ', epoch,  ' - ', 'cost: ', cost)
```

8. Validation of the Model based on Test Subset

```
#Run the trained model on test subset
pred_y = sess.run(y, feed_dict={x: test_x})

#calculate the correct predictions
correct_prediction = tf.equal(tf.argmax(pred_y,1),
tf.argmax(test_y,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
print(&amp;amp;amp;quot;Accuracy:
&amp;amp;amp;quot;,sess.run(accuracy))

plt.plot(range(len(cost_history)),cost_history)
plt.axis([0,training_epochs,0,np.max(cost_history)])
plt.show()
```




```
epoch :  986  -  cost:  0.423809
epoch :  987  -  cost:  0.423757
epoch :  988  -  cost:  0.423704
epoch :  989  -  cost:  0.423652
epoch :  990  -  cost:  0.4236
epoch :  991  -  cost:  0.423548
epoch :  992  -  cost:  0.423496
epoch :  993  -  cost:  0.423444
epoch :  994  -  cost:  0.423392
epoch :  995  -  cost:  0.42334
epoch :  996  -  cost:  0.423288
epoch :  997  -  cost:  0.423236
epoch :  998  -  cost:  0.423185
epoch :  999  -  cost:  0.423133
Accuracy:  0.833333
```

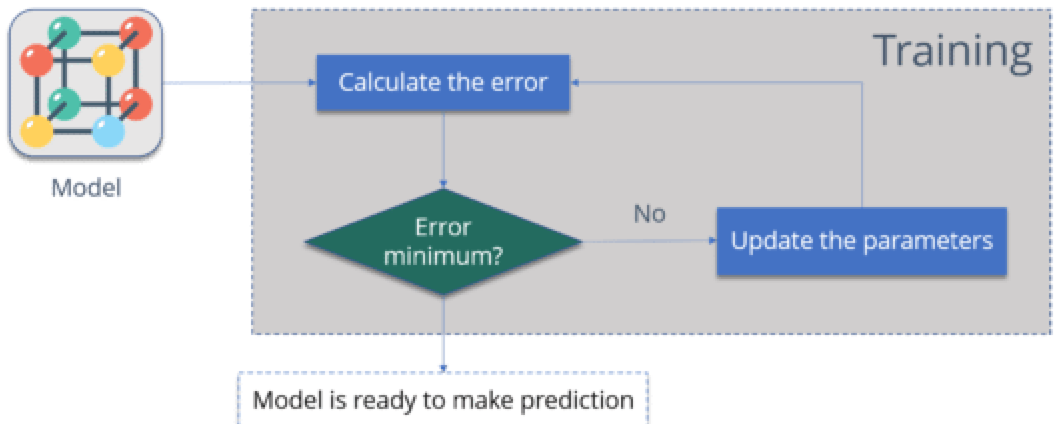
PART B

BACKPROPAGATION (RECALL)

What is Backpropagation?

The Backpropagation algorithm looks for the minimum value of the error function in weight space using a technique called the delta rule or gradient descent.

The weights that minimize the error function is then considered to be a solution to the learning problem.



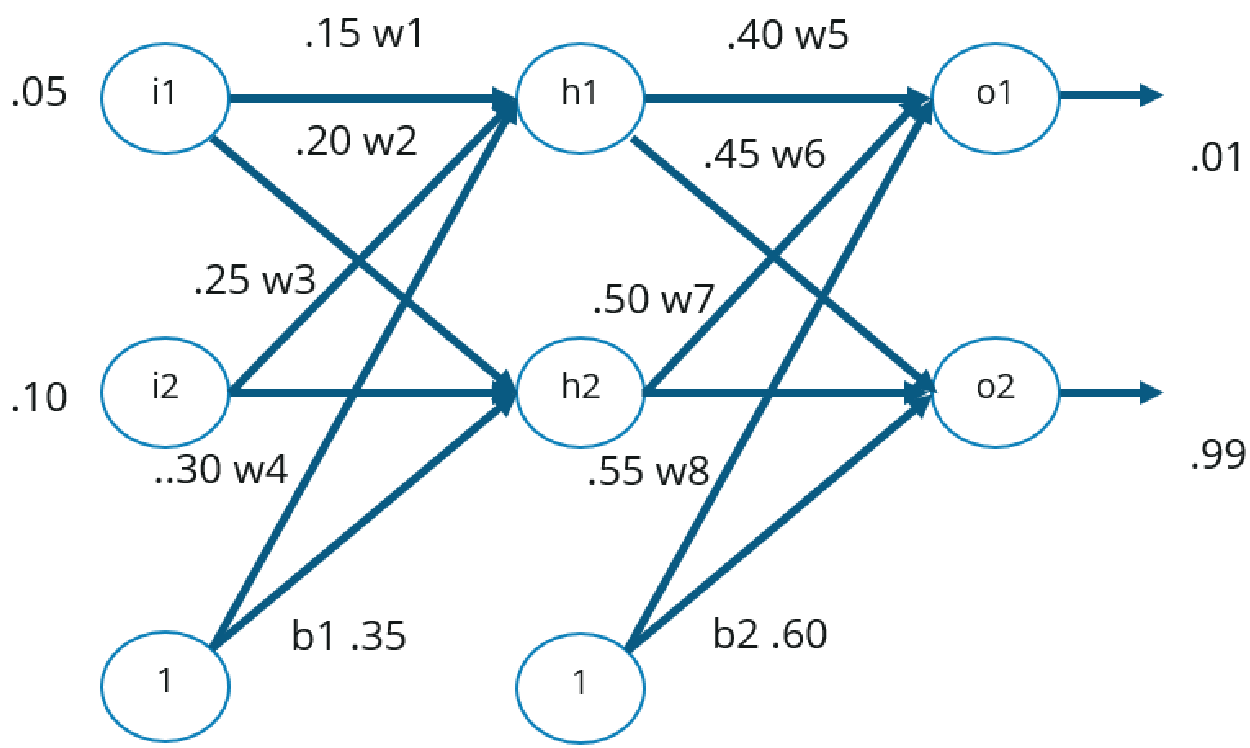
How Backpropagation Works?

There are 3 main steps in Backpropagation:

Step – 1: Forward Propagation

Step – 2: Backward Propagation

Step – 3: Putting all the values together and calculating the updated weight value



Step – 1: Forward Propagation

Net Input For h1:

net h1 = w1*i1 + w2*i2 + b1*1 → net h1 = 0.15*0.05 + 0.2*0.1 + 0.35*1 = 0.3775

Output Of h1:

out h1 = $1/1+e^{-\text{net h1}}$ → $1/1+e^{.3775}$ = 0.593269992

Output Of h2:

out h2 = 0.596884378

Output For o1:

net o1 = w5*out h1 + w6*out h2 + b2*1 → 0.4*0.593269992 + 0.45*0.596884378 + 0.6*1 = 1.105905967

Out o1 = $1/1+e^{-\text{net o1}}$ → $1/1+e^{-1.105905967}$ = 0.75136507

Output For o2:

Out o2 = 0.772928465

Error For o1:

$E_{o1} = \Sigma 1/2(target - output)^2$ → ½ (0.01 – 0.75136507)² = 0.274811083

Error For o2:

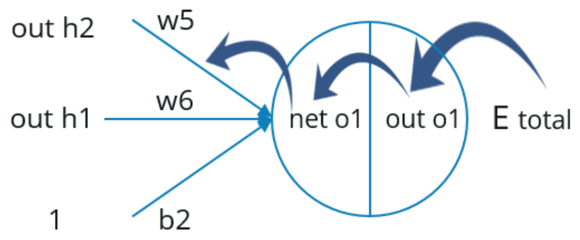
E_{o2} = 0.023560026

Total Error:

$E_{total} = E_{o1} + E_{o2}$ → 0.274811083 + 0.023560026 = 0.298371109

Step – 2: Backward Propagation

$$\frac{\delta Etotal}{\delta w5} = \frac{\delta Etotal}{\delta out\ o1} * \frac{\delta out\ o1}{\delta net\ o1} * \frac{\delta net\ o1}{\delta w5}$$



$$E_{total} = 1/2(\text{target o1} - \text{out o1})^2 + 1/2(\text{target o2} - \text{out o2})^2$$

$$\frac{\delta Etotal}{\delta out\ o1} = -(\text{target o1} - \text{out o1}) = -(0.01 - 0.75136507) = 0.74136507$$

$$\text{out o1} = 1/1+e^{-neto1}$$

$$\frac{\delta out\ o1}{\delta net\ o1} = \text{out o1 } (1 - \text{out o1}) = 0.75136507\ (1 - 0.75136507) = 0.186815602$$

$$\text{net o1} = w5 * \text{out h1} + w6 * \text{out h2} + b2 * 1$$

$$\frac{\delta net\ o1}{\delta w5} = 1 * \text{out h1 } w5^{(1-1)} + 0 + 0 = 0.593269992$$

Step – 3: Calculating the updated weight value

$$\frac{\delta Etotal}{\delta w5} = \frac{\delta Etotal}{\delta out\ o1} * \frac{\delta out\ o1}{\delta net\ o1} * \frac{\delta net\ o1}{\delta w5} \rightarrow 0.082167041$$

$$w5^{+} = w5 - n\frac{\delta Etotal}{\delta w5} \rightarrow w5^{+} = 0.4 - 0.5 * 0.082167041$$

Updated w5 → 0.35891648

Backpropagation Algorithm

initialize network weights (often small random values)

do

    for Each training example named ex

        // forward pass

        prediction = neural-net-output(network, ex)

        actual = teacher-output(ex)

        compute error (prediction - actual) at the output units

        // backward pass

        compute Δw_h for all weights from hidden layer to output layer

        compute Δw_i for all weights from input layer to hidden layer

        update network weights

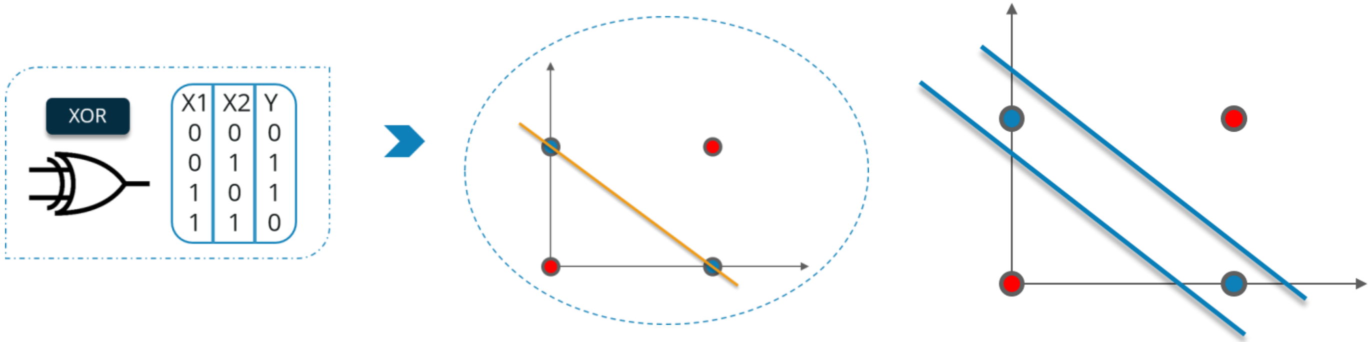
until all examples classified correctly or another stopping criterion satisfied

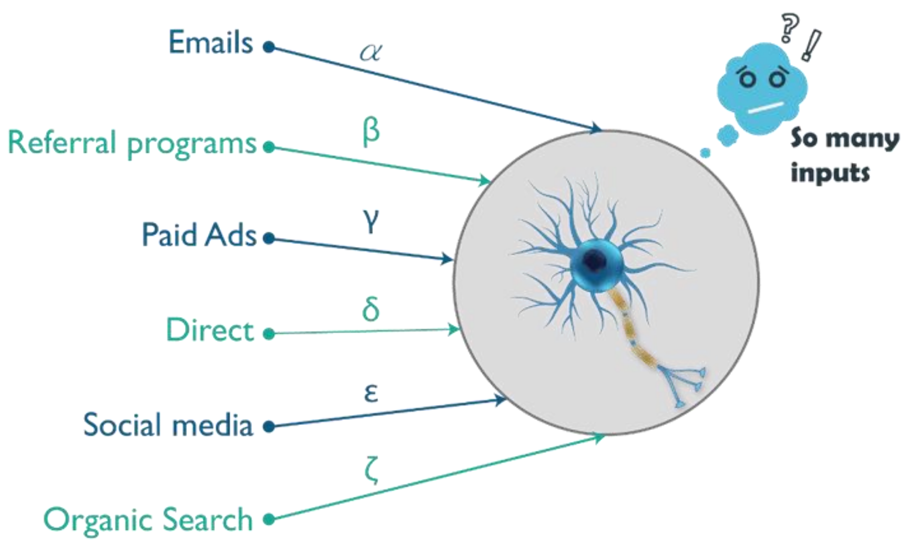
return the network

PART C

MULTI LAYER PERCEPTRON

Limitations of Single-Layer Perceptron



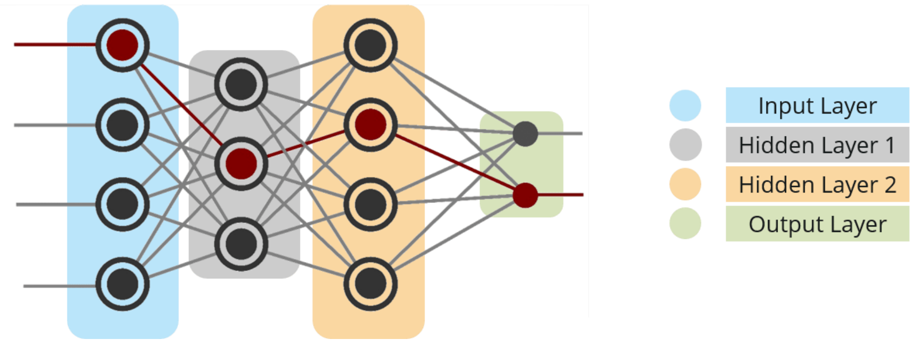


What is Multi-Layer Perceptron?

Input Nodes – The Input nodes provide information from the outside world to the network and are together referred to as the "Input Layer". No computation is performed in any of the Input nodes – *they just pass on the information to the hidden nodes.*

Hidden Nodes – The Hidden nodes have no direct connection with the outside world. *They perform computations and transfer information from the input nodes to the output nodes*. A collection of hidden nodes forms a "Hidden Layer". While a network will only have a single input layer and a single output layer, it can have zero or multiple Hidden Layers. ***A Multi-Layer Perceptron has one or more hidden layers***.

Output Nodes – The Output nodes are collectively referred to as the "Output Layer" and are responsible for *computations and transferring information from the network to the outside world*.



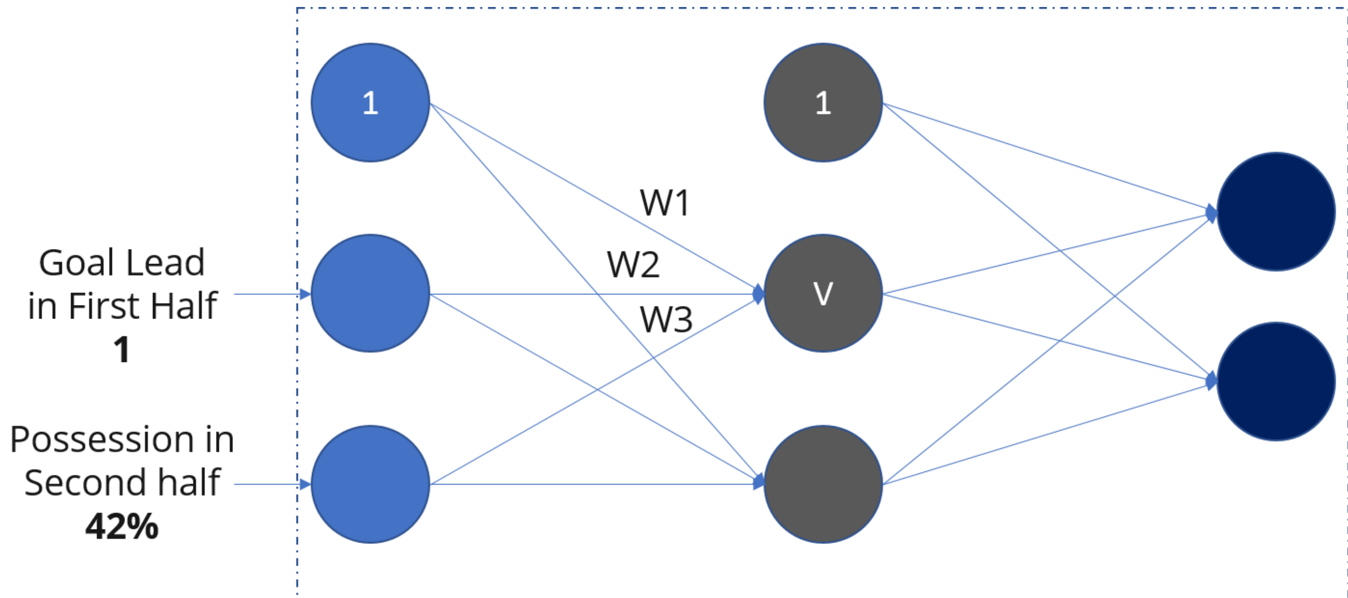
Example

Suppose we have data of a football team, Chelsea. The data contains three columns. The last column tells whether Chelsea won the match or they lost it. The other two columns are about, goal lead in the first half and possession in the second half. Possession is the amount of time for which the team has the ball in percentage.

We want to predict whether Chelsea will win the match or not, if the goal lead in the first half is 2 and the possession in the second half is 32%.

Goal Lead in First Half	Possession in Second Half	Won or Lost (1,0)?
0	80%	1
0	35%	0
1	42%	1
2	20%	0
-1	75%	1

Example _ Forward Propagation:



Probability of winning = 0.4   Target = 1

Error = (1-0.4) = 0.6

Probability of losing = 0.6   Target = 0

Error = (0-0.6) = -0.6

Example _ Backward Propagation

Network Output

Error/Loss

$$J(\mathbf{w}) = \frac{1}{2}\sum_i (\text{target}^{(i)} - \text{output}^{(i)})^2$$

Actual Output

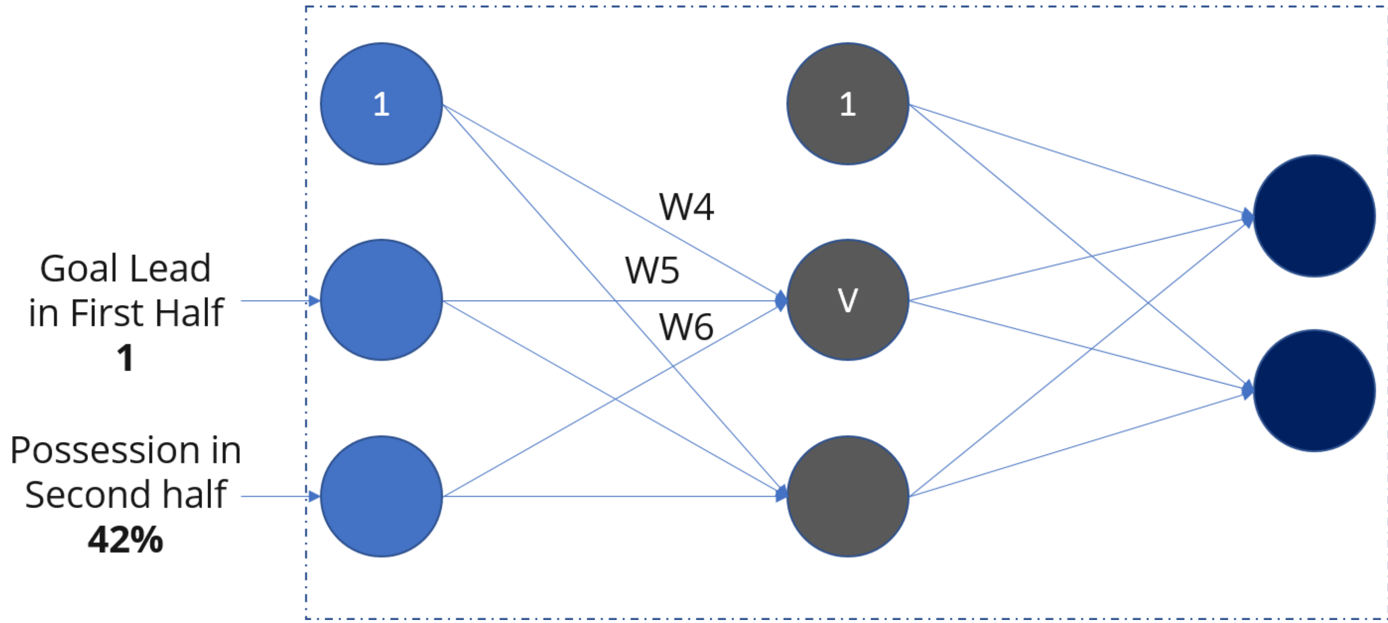
Learning Rate

Change in Weight

$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j},$$

Gradient

Example _ Weight Updation



Probability of winning = 0.8   Target = 1

Error = (1-0.8) = 0.2

Probability of losing = 0.2   Target = 0

Error = (0-0.2) = -0.2

Use- Case

Using Artificial Neural Network, we need to figure out, if the bank notes are real or fake?

4.0127	10.1477	-3.9366	-4.0728	0
2.6606	3.1681	1.9619	0.18662	0
3.931	1.8541	-0.02343	1.2314	0
0.01727	8.693	1.3989	-3.9668	0
3.2414	0.40971	1.4015	1.1952	0
2.2504	3.5757	0.35273	0.2836	0
-1.3971	3.3191	-1.3927	-1.9948	1
0.39012	-0.14279	-0.03199	0.35084	1
-1.6677	-7.1535	7.8929	0.96765	1
-3.8483	-12.8047	15.6824	-1.281	1
-3.5681	-8.213	10.083	0.96765	1
-2.2804	-0.30626	1.3347	1.3763	1
-1.7582	2.7397	-2.5323	-2.234	1
-0.89409	3.1991	-1.8219	-2.9452	1
0.3434	0.12415	-0.28733	0.14654	1

Features

- Variance of Wavelet Transformed image
- Skewness of Wavelet Transformed image
- Curtosis of Wavelet Transformed image
- Entropy of image

Label

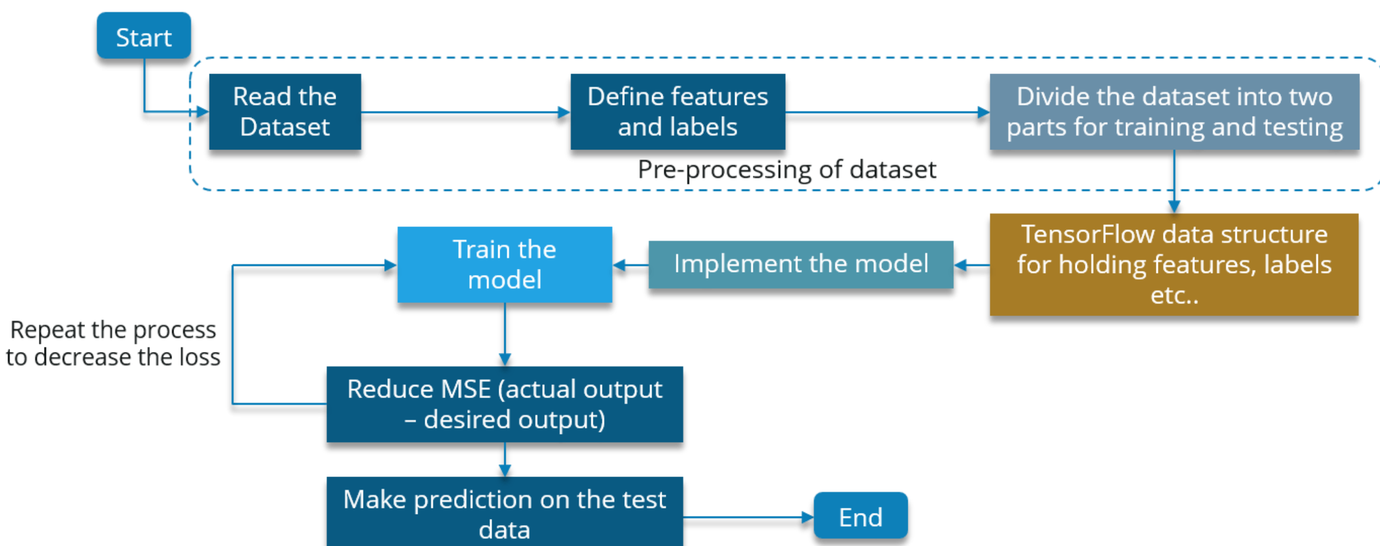
1 – Real, 0 - Fake

Use-Case

Data were extracted from images that were taken from genuine and forged banknote-like specimens.

The final images have 400×400 pixels. Due to the object lens and distance to the investigated object gray-scale, pictures with a resolution of about 660 dpi were gained.

Wavelet Transform tool were used to extract features from images.



Implementation

```python
# Import all the required libraries
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.utils import shuffle
from sklearn.model_selection import train_test_split

# Reading the dataset
def read_dataset():
    df = pd.read_csv("C:\\Users\\Saurabh\\PycharmProjects\\Neural Network Tutorial\\banknote.csv")
    # print(len(df.columns))
    X = df[df.columns[0:4]].values
    y = df[df.columns[4]]

    # Encode the dependent variable
    Y = one_hot_encode(y)
    print(X.shape)
    return (X, Y)
```

```
# Define the encoder function.
def one_hot_encode(labels):
    n_labels = len(labels)
    n_unique_labels = len(np.unique(labels))
    one_hot_encode = np.zeros((n_labels, n_unique_labels))
    one_hot_encode[np.arange(n_labels), labels] = 1
    return one_hot_encode


# Read the dataset
X, Y = read_dataset()

# Shuffle the dataset to mix up the rows.
X, Y = shuffle(X, Y, random_state=1)

# Convert the dataset into train and test part
train_x, test_x, train_y, test_y = train_test_split(X, Y, test_size=0.20,
random_state=415)

# Inpect the shape of the training and testing.
print(train_x.shape)
print(train_y.shape)
print(test_x.shape)
```

```
# Define the important parameters and variable to work with the
tensors
learning_rate = 0.3
training_epochs = 100
cost_history = np.empty(shape=[1], dtype=float)
n_dim = X.shape[1]
print("n_dim", n_dim)
n_class = 2
model_path = "C:\\Users\\Saurabh\\PycharmProjects\\Neural
Network Tutorial\\BankNotes"

# Define the number of hidden layers and number of neurons for
each layer
n_hidden_1 = 4
n_hidden_2 = 4
n_hidden_3 = 4
n_hidden_4 = 4

x = tf.placeholder(tf.float32, [None, n_dim])
W = tf.Variable(tf.zeros([n_dim, n_class]))
b = tf.Variable(tf.zeros([n_class]))
y_ = tf.placeholder(tf.float32, [None, n_class])
```

```
# Define the model
def multilayer_perceptron(x, weights, biases):

    # Hidden layer with RELU activationsd
    layer_1 = tf.add(tf.matmul(x, weights['h1']), biases['b1'])
    layer_1 = tf.nn.relu(layer_1)

    # Hidden layer with sigmoid activation
    layer_2 = tf.add(tf.matmul(layer_1, weights['h2']), biases['b2'])
    layer_2 = tf.nn.relu(layer_2)

    # Hidden layer with sigmoid activation
    layer_3 = tf.add(tf.matmul(layer_2, weights['h3']), biases['b3'])
    layer_3 = tf.nn.relu(layer_3)

    # Hidden layer with RELU activation
    layer_4 = tf.add(tf.matmul(layer_3, weights['h4']), biases['b4'])
    layer_4 = tf.nn.sigmoid(layer_4)

    # Output layer with linear activation
    out_layer = tf.matmul(layer_4, weights['out']) + biases['out']
    return out_layer
```

```
# Define the weights and the biases for each layer

weights = {
    'h1': tf.Variable(tf.truncated_normal([n_dim, n_hidden_1])),
    'h2': tf.Variable(tf.truncated_normal([n_hidden_1, n_hidden_2])),
    'h3': tf.Variable(tf.truncated_normal([n_hidden_2, n_hidden_3])),
    'h4': tf.Variable(tf.truncated_normal([n_hidden_3, n_hidden_4])),
    'out': tf.Variable(tf.truncated_normal([n_hidden_4, n_class]))
}
biases = {
    'b1': tf.Variable(tf.truncated_normal([n_hidden_1])),
    'b2': tf.Variable(tf.truncated_normal([n_hidden_2])),
    'b3': tf.Variable(tf.truncated_normal([n_hidden_3])),
    'b4': tf.Variable(tf.truncated_normal([n_hidden_4])),
    'out': tf.Variable(tf.truncated_normal([n_class]))
}
```

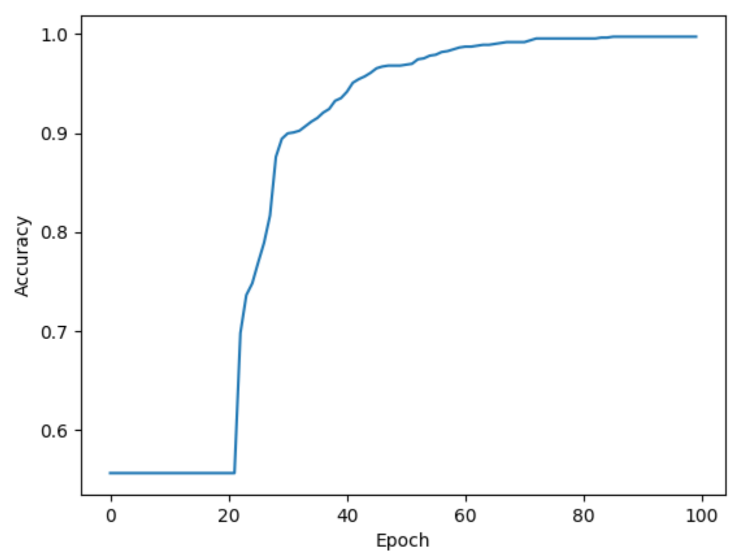
```
# Initialize all the variables

init = tf.global_variables_initializer()

saver = tf.train.Saver()

# Call your model defined
y = multilayer_perceptron(x, weights, biases)

# Define the cost function and optimizer
cost_function = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=y, labels=y_))
training_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost_function)

sess = tf.Session()
sess.run(init)
```

```
# Calculate the cost and the accuracy for each epoch

mse_history = []
accuracy_history = []

for epoch in range(training_epochs):
    sess.run(training_step, feed_dict={x: train_x, y_: train_y})
    cost = sess.run(cost_function, feed_dict={x: train_x, y_: train_y})
    cost_history = np.append(cost_history, cost)
    correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
    # print("Accuracy: ", (sess.run(accuracy, feed_dict={x: test_x, y_: test_y})))
    pred_y = sess.run(y, feed_dict={x: test_x})
    mse = tf.reduce_mean(tf.square(pred_y - test_y))
    mse_ = sess.run(mse)
    mse_history.append(mse_)
    accuracy = (sess.run(accuracy, feed_dict={x: train_x, y_: train_y}))
    accuracy_history.append(accuracy)

    print('epoch : ', epoch, ' - ', 'cost: ', cost, " - MSE: ", mse_, "- Train Accuracy: ", accuracy)

save_path = saver.save(sess, model_path)
print("Model saved in file: %s" % save_path)
```

```
#Plot Accuracy Graph
plt.plot(accuracy_history)
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.show()

# Print the final accuracy

correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
print("Test Accuracy: ", (sess.run(accuracy, feed_dict={x: test_x, y_: test_y})))

# Print the final mean square error

pred_y = sess.run(y, feed_dict={x: test_x})
mse = tf.reduce_mean(tf.square(pred_y - test_y))
print("MSE: %.4f" % sess.run(mse))
```

Output

```
epoch :  86  -  cost:  0.104916  - MSE:  1.96544271492 - Train Accuracy:  0.997263
epoch :  87  -  cost:  0.103177  - MSE:  1.98525729895 - Train Accuracy:  0.997263
epoch :  88  -  cost:  0.1015  - MSE:  2.00468696572 - Train Accuracy:  0.997263
epoch :  89  -  cost:  0.0998027  - MSE:  2.02396983975 - Train Accuracy:  0.997263
epoch :  90  -  cost:  0.0980679  - MSE:  2.04333056024 - Train Accuracy:  0.997263
epoch :  91  -  cost:  0.0964177  - MSE:  2.06350608599 - Train Accuracy:  0.997263
epoch :  92  -  cost:  0.0948272  - MSE:  2.08376747827 - Train Accuracy:  0.997263
epoch :  93  -  cost:  0.0932706  - MSE:  2.10399607284 - Train Accuracy:  0.997263
epoch :  94  -  cost:  0.0917953  - MSE:  2.12376526339 - Train Accuracy:  0.997263
epoch :  95  -  cost:  0.0903142  - MSE:  2.1436460348 - Train Accuracy:  0.997263
epoch :  96  -  cost:  0.0889434  - MSE:  2.16332118965 - Train Accuracy:  0.997263
epoch :  97  -  cost:  0.0876444  - MSE:  2.18238911044 - Train Accuracy:  0.997263
epoch :  98  -  cost:  0.0863936  - MSE:  2.20116683898 - Train Accuracy:  0.997263
epoch :  99  -  cost:  0.0851813  - MSE:  2.21978664866 - Train Accuracy:  0.997263
Model saved in file: C:\Users\Saurabh\PycharmProjects\Neural Network Tutorial\BankNotes
Test Accuracy:  0.996364
MSE: 2.2198
```



Questions? More Information?

✉ hvusynh@hcmiu.edu.vn