Manik Daya
2017140008
TEIT
FCI

Experiment 3:	**Supervised Learning** **Back Propagation in NN**

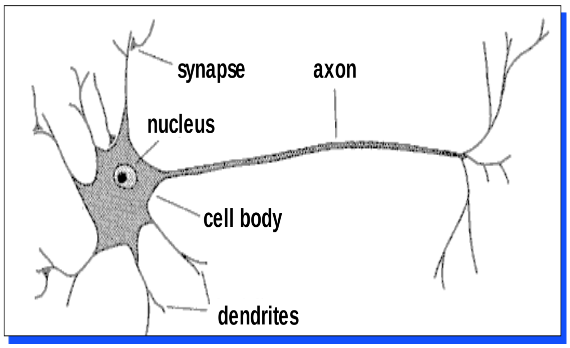
1 OBJECTIVE

Introduction to the Back propagation Algorithm

2 BACK PROPAGATION ALGORITHM

The backpropagation algorithm is a supervised learning method for multilayer feed-forward networks from the field of Artificial Neural Networks.

Feed-forward neural networks are inspired by the information processing of one or more neural cells, called a neuron. A neuron accepts input signals via its dendrites, which pass the electrical signal down to the cell body. The axon carries the signal out to synapses, which are the connections of a cell's axon to other cell's dendrites.



The principle of the backpropagation approach is to model a given function by modifying internal weightings of input signals to produce an expected output signal. The system is trained using a supervised learning method, where the error between the system's output and a known expected output is presented to the system and used to modify its internal state.

Technically, the backpropagation algorithm is a method for training the weights in a multilayer feed-forward neural network. As such, it requires a network structure to be defined of one or more layers where one layer is fully connected to the next layer. A standard network structure is one input layer, one hidden layer, and one output layer.

Backpropagation can be used for both classification and regression problems, but we will focus on classification in this tutorial.

In classification problems, best results are achieved when the network has one neuron in the output layer for each class value. For example, a 2-class or binary classification problem with the class values of A and B. These expected outputs would have to be transformed into binary vectors with one column for each class value. Such as [1, 0] and [0, 1] for A and B respectively. This is called a one hot encoding.

3 LAB TASKS

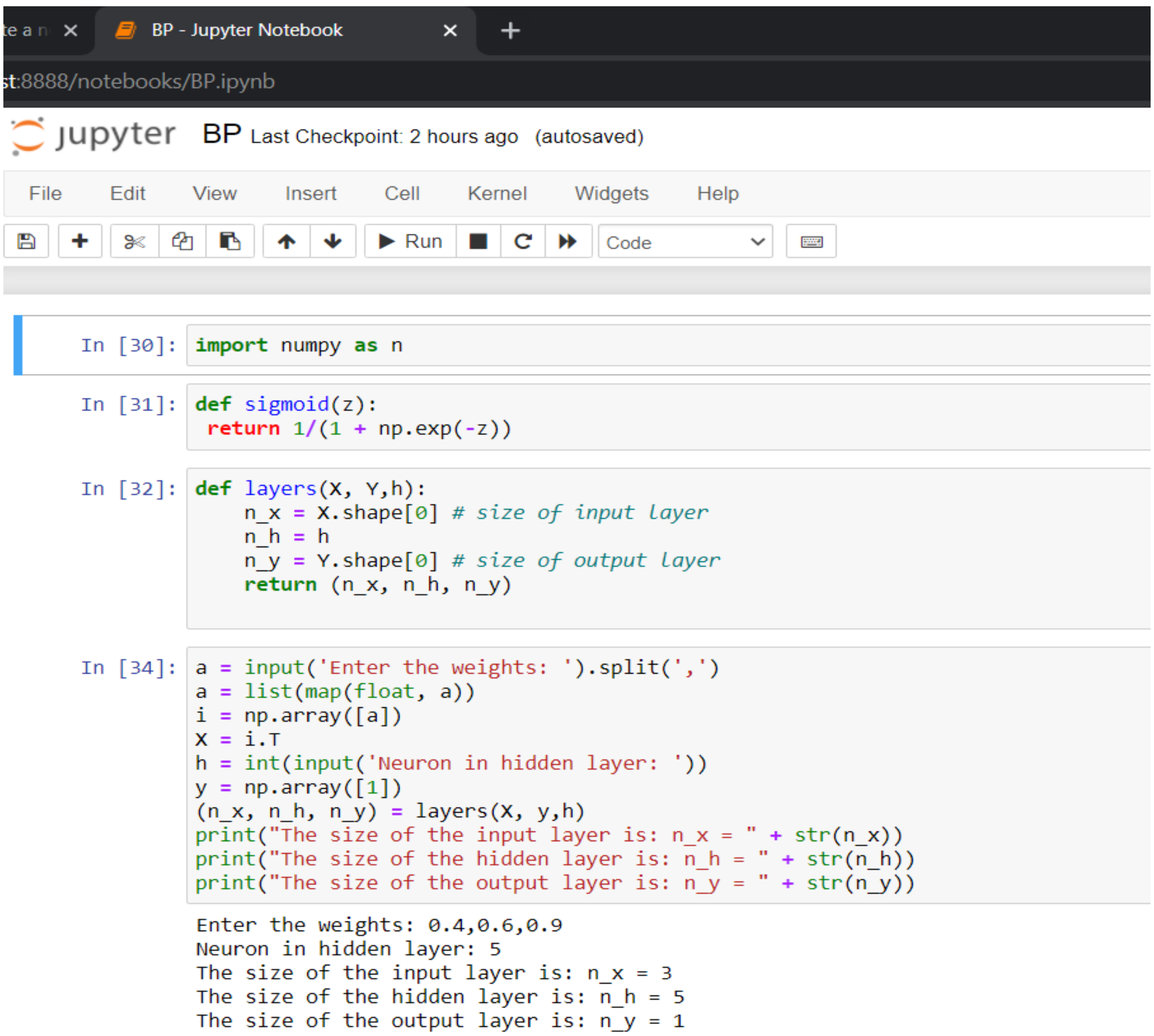
This lab is broken down into 6 parts:

1. Initialize Network.
2. Forward Propagate.

3. Back Propagate Error.
4. Train Network.
5. Predict.
6. Dataset Case Study.

4 SUBMISSION

Screenshots (including code and output)




```
In [30]: import numpy as n

In [31]: def sigmoid(z):
          return 1/(1 + np.exp(-z))

In [32]: def layers(X, Y,h):
             n_x = X.shape[0] # size of input layer
             n_h = h
             n_y = Y.shape[0] # size of output layer
             return (n_x, n_h, n_y)

In [34]: a = input('Enter the weights: ').split(',')
         a = list(map(float, a))
         i = np.array([a])
         X = i.T
         h = int(input('Neuron in hidden layer: '))
         y = np.array([1])
         (n_x, n_h, n_y) = layers(X, y,h)
         print("The size of the input layer is: n_x = " + str(n_x))
         print("The size of the hidden layer is: n_h = " + str(n_h))
         print("The size of the output layer is: n_y = " + str(n_y))

         Enter the weights: 0.4,0.6,0.9
         Neuron in hidden layer: 5
         The size of the input layer is: n_x = 3
         The size of the hidden layer is: n_h = 5
         The size of the output layer is: n_y = 1
```

In [35]:
```
def network_parameters(n_x, n_h, n_y):
    Weights1 = np.random.randn(n_h, n_x) * 0.1
    bias1 = np.random.randn(n_h,1)
    Weights2 = np.random.randn(n_y, n_h) * 0.1
    bias2 = np.random.randn(n_y,1)
    parameters = {"W1": Weights1,
                  "b1": bias1,
                  "W2": Weights2,
                  "b2": bias2}
    return parameters
```

In [36]:
```
parameters = network_parameters(n_x, n_h, n_y)
print("Weights1 = " + str(parameters["W1"]))
print("bias1 = " + str(parameters["b1"]))
print("Weights2 = " + str(parameters["W2"]))
print("bias2 = " + str(parameters["b2"]))
```

```
Weights1 = [[ 0.0625245  -0.01605134 -0.07688364]
 [-0.02300307  0.07450563  0.19761108]
 [-0.12441233 -0.06264169 -0.08037661]
 [-0.24190832 -0.0923792  -0.10238758]
 [ 0.1123978  -0.01319142 -0.16232854]]
bias1 = [[ 0.64667545]
 [-0.35627076]
 [-1.74314104]
 [-0.59664964]
 [-0.58859438]]
Weights2 = [[-0.08738823  0.00297138 -0.22482578 -0.02677619  0.10131834]]
bias2 = [[0.85279784]]
```

In [37]:
```
#Forward propogation
```

In [38]:
```
def propagate(X, parameters):
    Weights1 = parameters["W1"]
    bias1 = parameters["b1"]
    Weights2 = parameters["W2"]
    bias2 = parameters["b2"]

    Z1 = np.dot(Weights1,X) + bias1
    A1 = sigmoid(Z1)
    Z2 = np.dot(Weights2, A1) + bias2
    A2 = sigmoid(Z2)
    cache = {"Z1": Z1,
             "A1": A1,
             "Z2": Z2,
             "A2": A2}
    return A2, cache
```

In [39]:
```
A2, cache = propagate(X, parameters)
print(cache['Z1'], cache['A1'], cache['Z2'], cache['A2'])
```

```
[[ 0.59285918]
 [-0.14291864]
 [-1.90282993]
 [-0.84098931]
 [-0.69764581]] [[0.6440209 ]
 [0.46433103]
 [0.12978852]
 [0.30132647]
 [0.33233439]] [[0.79432109]] [[0.6887584]]
```

In [40]:
```
#Compute cost of network
```

```
In [40]: #Compute cost of network
```

```
In [41]: def compute_loss(A2, Y, parameters):
             if len(Y.shape)==2:
                 m = Y.shape[1]
             else:
                 m = 1
             logprobs = np.multiply(np.log(A2),Y) + np.multiply(np.log(1-A2),(1-Y))
             cost = - (np.sum(logprobs)/m)
             cost = float(np.squeeze(cost))
             return cost
```

```
In [42]: print("cost = " + str(compute_loss(A2, y, parameters)))
```

Run Code

```
In [43]: #backpropogate the cost
```

```
In [44]: def backward_propagation(parameters, cache, X, Y):
             m = X.shape[1]
             Weights1 = parameters["W1"]
             Weights2 = parameters["W2"]
             A1 = cache['A1']
             A2 = cache['A2']
             dZ2 = A2-Y
             dW2 = (np.dot(dZ2,A1.T)/m)
             db2 = (np.sum(dZ2, axis=1,keepdims=True)/m)
             dZ1 = np.dot(Weights2.T, dZ2) * (1-np.power(A1,2))
             dW1 = (np.dot(dZ1, X.T)/m)
             db1 = (np.sum(dZ1, axis=1,keepdims=True)/m)
             grads = {"dW1": dW1,
                      "db1": db1,
                      "dW2": dW2,
                      "db2": db2}
             return grads
```

```
In [45]: grads = backward_propagation(parameters, cache, X, y)
         print ("dW1 = "+ str(grads["dW1"]))
         print ("db1 = "+ str(grads["db1"]))
         print ("dW2 = "+ str(grads["dW2"]))
         print ("db2 = "+ str(grads["db2"]))

         dW1 = [[ 0.00636711  0.00955067  0.014326   ]
          [-0.00029017 -0.00043525 -0.00065288]
          [ 0.02751856  0.04127784  0.06191676]
          [ 0.00303087  0.0045463   0.00681945]
          [-0.01122065 -0.01683097 -0.02524646]]
         db1 = [[ 0.01591778]
          [-0.00072542]
          [ 0.0687964 ]
          [ 0.00757717]
          [-0.02805162]]
         dW2 = [[-0.2004461  -0.14451913 -0.04039559 -0.09378533 -0.10343629]]
```

```
In [47]: def update_parameters(parameters, grads, learning_rate = 0.2):
    Weights1 = parameters['W1']
    bias1 = parameters['b1']
    Weights2 = parameters['W2']
    bias2 = parameters['b2']
    dW1 = grads['dW1']
    db1 = grads['db1']
    dW2 = grads['dW2']
    db2 = grads['db2']

    Weights1 = Weights1 - (learning_rate*dW1)
    bias1 = bias1 - (learning_rate*db1)
    Weights2 = Weights2 - (learning_rate*dW2)
    bias2 = bias2 - (learning_rate*db2)
    parameters = {"W1": Weights1,
                  "b1": bias1,
                  "W2": Weights2,
                  "b2": bias2}
    return parameters
```

```
In [48]: parameters = update_parameters(parameters, grads)
print("Weights1 = " + str(parameters["W1"]))
print("bias1 = " + str(parameters["b1"]))
print("Weights2 = " + str(parameters["W2"]))
print("bias2 = " + str(parameters["b2"]))
```

```
Weights1 = [[ 0.06125107 -0.01796147 -0.07974883]
 [-0.02294504  0.07459268  0.19774165]
 [-0.12991604 -0.07089726 -0.09275996]
 [-0.24251449 -0.09328846 -0.10375147]
 [ 0.11464193 -0.00982523 -0.15727925]]
bias1 = [[ 0.6434919 ]
 [-0.35612567]
 [-1.75690032]
 [-0.59816508]
 [-0.58298406]]
```

In [56]:
```python
def nn_model(X, Y, n_h, num_iterations = 10000, print_cost=False):
    np.random.seed(3)
    n_x = layers(X, Y,n_h)[0]
    n_y = layers(X, Y,n_h)[2]

    parameters = network_parameters(X.shape[0], n_h, Y.shape[0])

    # Loop (gradient descent)

    for i in range(0, num_iterations):
        A2, cache = propagate(X, parameters)
        cost = compute_loss(A2, Y, parameters)
        grads = backward_propagation(parameters, cache, X, Y)
        parameters = update_parameters(parameters, grads)
        if print_cost and i%1000 == 0:
            print ("Cost after iteration %i: %f" %(i, cost))
    return parameters
```

In [57]:
```python
parameters = nn_model(X, y, h, num_iterations=10000, print_cost=True)
print("Weights1 = " + str(parameters["W1"]))
print("bias1 = " + str(parameters["b1"]))
print("Weights2 = " + str(parameters["W2"]))
print("bias2 = " + str(parameters["b2"]))
```

```
Cost after iteration 0: 1.141313
Cost after iteration 1000: 0.001117
Cost after iteration 2000: 0.000523
Cost after iteration 3000: 0.000338
Cost after iteration 4000: 0.000248
Cost after iteration 5000: 0.000196
Cost after iteration 6000: 0.000162
Cost after iteration 7000: 0.000137
Cost after iteration 8000: 0.000119
Cost after iteration 9000: 0.000106
Weights1 = [[0.54045013 0.58603192 0.82322114]
```


```
Cost after iteration 7000: 0.000137
Cost after iteration 8000: 0.000119
Cost after iteration 9000: 0.000106
Weights1 = [[0.54045013 0.58603192 0.82322114]
```


```
 [0.22171776 0.58436173 0.88267493]
 [0.47974685 0.66933143 1.09366543]
 [0.26922321 0.34403104 0.80158852]
 [0.41074391 0.65487547 0.73088061]]
bias1 = [[ 0.4992908 ]
 [ 0.47480764]
 [-0.32642482]
 [ 1.77472997]
 [-0.29453735]]
Weights2 = [[1.63965598 1.59481657 1.34490308 2.53282633 1.09274976]]
bias2 = [[2.2911678]]
```

In [58]:
```python
def predict(parameters, X):
    A2, cache = propagate(X, parameters)
    predictions = np.where(A2 > 0.5, 1, 0)
    return predictions
```

In [59]:
```python
predictions = predict(parameters, X)
print("predictions = " + str(predictions))
```

```
predictions = [[1]]
```

Observations:

1) Weights and number of neurons in hidden layer input was taken from user
2) Random biases were applied to neurons in the hidden and in the output layer
3) Sigmoid activation function is used for better accuracy
4) On increasing the iteration number, the loss function was reduced significantly.
5)Since it is binary classification so output will always have one neuron
6)The classification is based on the threshold value which is set as 0.5.

Conclusion: Back propagation is learned and implemented in python using random weights and random bias. The weights and bias are randomly generated and the cost is 1.141313 which is a bit high so to reduce the cost weights are updated and about 10,000 iterations are done and the cost is reduced after every iteration. After 9000th iteration the cost came down to 0.000106 which is very low as compared to 1.141313, weights are updated accordingly and the desired output of 1 is achieved.