Unit 7 – e portfolio activity

Simple Perception – Exercise 1

```
import numpy as np
```

importing numpy

```
inputs = np.array([45, 25])
```

```
# Check the type of the inputs

type(inputs)
```

```
numpy.ndarray
```

```
# check the value at index position 0
inputs[0]
```

```
45
```

```
# creating the weights as Numpy array

weights = np.array([0.7, 0.1])
```

```
0.7
```

The dot function is called the dot product from linear algebra. If you are dealing with a huge dataset, The processing difference between the for loop used in the last notebook and this dot product will significantly be different.

```
def sum_func(inputs, weights):
    return inputs.dot(weights)

# for weights = [0.7, 0.1]

s_prob1 = sum_func(inputs, weights)
s_prob1
```

```
34.0
```

Creating the step function

```
def step_function(sum_func):
  if (sum_func >= 1):
    print(f'The Sum Function is greater than or equal to 1')
    return 1
  else:
        print(f'The Sum Function is NOT greater')
        return 0
```

```
step_function(s_prob1 )
```

```
The Sum Function is greater than or equal to 1
1
```

```
weights = [-0.7, 0.1]
```

```
# for weights = [- 0.7, 0.1]

s_prob2 = sum_func(inputs, weights)

round(s_prob2, 2)  #round to 2 decimal places
```

```
-29.0
```

```
step_function(s_prob2 )
```

```
The Sum Function is NOT greater
0
```

If the weights were -0.1 and 0.1

```
weights = [-0.1, 0.1]
```

```
# for weights = [- 0.1, 0.1]

s_prob2 = sum_func(inputs, weights)

round(s_prob2, 2)  #round to 2 decimal places
```

```
-2.0
```

If the weights were 0.5 and 0.5

```
weights = [0.5, 0.5]
```

```
# for weights = [0.5, 0.5]

s_prob2 = sum_func(inputs, weights)

round(s_prob2, 2)  #round to 2 decimal places
```

```
35.0
```

<u>Exercise 2 – perception and operator</u>

```
import numpy as np
```

```
# Creating input values as a matrix not as a vector
inputs = np.array([[0,0], [0,1], [1,0], [1,1]])
```

```
# Chcking the shape of the inputs

inputs.shape
```

```
(4, 2)
```

```
# one weight for x1 and one for x2
weights = np.array([0.0, 0.0])
```

```
learning_rate = 0.1
```

```
# This is our Activation function

def step_function(sum):
  if (sum >= 1):
    #print(f'The Sum of Weights is Greater or equal to 1')
    return 1
  else:
        #print(f'The Sum of Weights is NOT > or = to 1')
        return 0
```

We define a function that allows us to calculate/ process the output. The function accepts an instance of our data, then calculate the sum function using Numpy. Finally, we check the output by passing it through the "Step Function."

```
def cal_output(instance):
  sum_func = instance.dot(weights)
  return step_function(sum_func)
```

We pass it as a list in a numpy array

```
cal_output(np.array([[1,1]]))
```

```
0
```

```
# Check the number of outputs

len(outputs)
```

```
# Checking the index of the input at postion 3 ..
# this is the last inpute value
inputs[3]
```

```
array([1, 1])
```

```
inputs
```

```
array([[0, 0],
       [0, 1],
       [1, 0],
       [1, 1]])
```

Note that: usually, we will need to define the number of epochs, because we will never really get a value of zero when dealing with real-world data. However, for this small data, we will run the loop till we obtain zero error

```
def train():
    #
    total_error_value = 1
    # While the total_error_value is not equal to zero. we are asumming 
that at the start of running our network there will be no zero
    while (total_error_value != 0):
        #making the total_error 0 so we can do other calculations
        total_error_value = 0
        #Looping into each row of the dataset (remember indexing in python 
starts at zero hence 0-3 which are 4 values)
        for i in range(len(outputs)):
            #Calculating predictions
            prediction = cal_output(inputs[i])
            # Calculating the absolute value of the error
            error = abs(outputs[i] - prediction)
            #Updating the error
            total_error_value  += error

            if error > 0:
                for j in range(len(weights)):
                    #updating the weights for x1 and x2
                    weights[j] = weights[j] + (learning_rate * 
inputs[i][j] * error)
                    print('Weight updated to: ' + str(weights[j]))
        print('Total error Value: ' + str(total_error_value))
```

```
train()
```

```
Weight updated to: 0.1
Weight updated to: 0.1
Total error Value: 1
Weight updated to: 0.2
Weight updated to: 0.2
Total error Value: 1
Weight updated to: 0.30000000000000004
Weight updated to: 0.30000000000000004
Total error Value: 1
Weight updated to: 0.4
Weight updated to: 0.4
Total error Value: 1
Weight updated to: 0.5
Weight updated to: 0.5
Total error Value: 1
Total error Value: 0
```

```
# Now we have the final weights that will be used to classify new
instances of the data after training.

weights
```

```
array([0.5, 0.5])
```

```
cal_output(np.array([0,0]))
```

```
0
```

```
cal_output(np.array([0,1]))
```

```
0
```

```
cal_output(np.array([1,0]))
```

```
0
```

```
cal_output(np.array([1,1]))
```

```
1
```

Exercise 3 – multi-layer perception

*** If X is high, the value is approximately 1**
*** if X is small, the value is approximately 0**

```
import numpy as np
```

```
def sigmoid(sum_func):
  return 1 / (1 + np.exp(-sum_func))
```

```
sigmoid(0)
```

```
0.5
```

```
np.exp(2)
```

```
7.38905609893065
```

```
np.exp(1)
```

```
2.718281828459045
```

```
sigmoid(40)
```

```
1.0
```

```
sigmoid(-20.5)
```

```
1.2501528648238605e-09
```

```
inputs = np.array([[0,0],
                   [0,1],
                   [1,0],
                   [1,1]])
```

```
inputs
```

```
array([[0, 0],
       [0, 1],
       [1, 0],
       [1, 1]])
```

```
inputs.shape
```

```
(4, 2)
```

```
outputs = np.array([[0],
                    [1],
                    [1],
                    [0]])
```

```
(4, 1)
```

```
# First row holds the weights for x1, 2nd row contains the weights for x2

weights_0 = np.array([[-0.424, -0.740, -0.961],
                      [0.358, -0.577, -0.469]])
weights_0.shape
```

```
(2, 3)
```

```
weights_1 = np.array([[-0.017],
                      [-0.893],
                      [0.148]])
weights_1.shape
```

```
(3, 1)
```

```python
epochs = 100
learning_rate = 0.3
```

```python
input_layer = inputs
input_layer
```

```
array([[0, 0],
       [0, 1],
       [1, 0],
       [1, 1]])
```

```python
# "sum_synapse_0" This holds the sum function total of weights for the hidden layer
# For the Output: Each row holds the sum_func for each input data  [0,0,0 -> data 0,0],[0.358, -0.577, -0.469 --> 0,1]
# The dot product does the matrix multiplication and also the sum

sum_synapse_0 = np.dot(input_layer, weights_0)
sum_synapse_0
```

```
array([[ 0.   ,  0.   ,  0.   ],
       [ 0.358, -0.577, -0.469],
       [-0.424, -0.74 , -0.961],
       [-0.066, -1.317, -1.43 ]])
```

```python
# Computing the Sigmoid function for the Hidden layer

hidden_layer = sigmoid(sum_synapse_0)
hidden_layer
```

```
array([[0.5       , 0.5       , 0.5       ],
       [0.5885562 , 0.35962319, 0.38485296],
       [0.39555998, 0.32300414, 0.27667802],
       [0.48350599, 0.21131785, 0.19309868]])
```

```python
weights_1
```

```
array([[-0.017],
       [-0.893],
       [ 0.148]])
```

```
# "sum_synapse_1" This holds the sum function total of weights for the output layer
# For the Output: Each row holds the sum_func for each input data

sum_synapse_1 = np.dot(hidden_layer, weights_1)
sum_synapse_1
```

```
array([[-0.381     ],
       [-0.27419072],
       [-0.25421887],
       [-0.16834784]])
```

```
output_layer = sigmoid(sum_synapse_1)
output_layer
```

```
array([[0.40588573],
       [0.43187857],
       [0.43678536],
       [0.45801216]])
```

```
outputs
```

```
array([[0],
       [1],
       [1],
       [0]])
```

```
output_layer
```

```
array([[0.40588573],
       [0.43187857],
       [0.43678536],
       [0.45801216]])
```

```
error_output_layer = outputs - output_layer
error_output_layer
```

```
array([[-0.40588573],
       [ 0.56812143],
       [ 0.56321464],
       [-0.45801216]])
```

```
average_error = np.mean(abs(error_output_layer))
average_error
```

```
0.49880848923713045
```

```python
def sigmoid_derivative(sigmoid):
    return sigmoid * (1 - sigmoid)
```

```python
# output_layer holds the results of our application of the sigmoid, computed above

output_layer
```

```
array([[0.40588573],
       [0.43187857],
       [0.43678536],
       [0.45801216]])
```

```python
# derivative_output is our Derivative of the activation function (sigmoid) which we have on the slide
# each row is for each instance of our input dataset

derivative_output = sigmoid_derivative(output_layer)
derivative_output
```

```
array([[0.2411425 ],
       [0.24535947],
       [0.24600391],
       [0.24823702]])
```

```python
error_output_layer
```

```
array([[-0.40588573],
       [ 0.56812143],
       [ 0.56321464],
       [-0.45801216]])
```

```python
# Delta output
# each row is for each instance of our input dataset

delta_output = error_output_layer * derivative_output
delta_output
```

```
array([[-0.0978763 ],
       [ 0.13939397],
       [ 0.138553  ],
       [-0.11369557]])
```

DELTA CALCULATIONS FOR THE HIDDEN LAYER

```python
delta_output
```

```
array([[-0.0978763 ],
       [ 0.13939397],
       [ 0.138553  ],
       [-0.11369557]])
```

```
weights_1
```

```
array([[-0.017],
       [-0.893],
       [ 0.148]])
```

* Lets deal with this part first (Weight * delta_output)
* Notice that we will get an error below becuase of the shape of the weights_1 (Transpose)

```
delta_output_x_weight = delta_output.dot(weights_1)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-36-50b740e5a31c> in <cell line: 1>()
----> 1 delta_output_x_weight = delta_output.dot(weights_1)

ValueError: shapes (4,1) and (3,1) not aligned: 1 (dim 1) != 3 (dim 0)
```

```
weights_1.shape
```

```
(3, 1)
```

```
weights_1T = weights_1.T
weights_1T
```

```
array([[-0.017, -0.893,  0.148]])
```

```
weights_1T.shape
```

```
(1, 3)
```

The weights will have to be multiplied by each delta_output for each data instance

array([[-0.017],
[-0.893],
[0.148]])

```
delta_output_x_weight = delta_output.dot(weights_1T)
delta_output_x_weight
```

```
array([[ 0.0016639 ,  0.08740354, -0.01448569],
       [-0.0023697 , -0.12447882,  0.02063031],
       [-0.0023554 , -0.12372783,  0.02050584],
       [ 0.00193282,  0.10153015, -0.01682694]])
```

* Now we need to deal with the last part of the equation
* sigmoid_derivative * delta_output_x_weight

```
hidden_layer
```

```
array([[0.5       , 0.5       , 0.5       ],
       [0.5885562 , 0.35962319, 0.38485296],
       [0.39555998, 0.32300414, 0.27667802],
       [0.48350599, 0.21131785, 0.19309868]])
```

```
#  Each row in the output of delta_hidden_layer is for the data input values

delta_hidden_layer = delta_output_x_weight * sigmoid_derivative(hidden_layer)
delta_hidden_layer
```

```
array([[ 0.00041597,  0.02185088, -0.00362142],
       [-0.00057384, -0.02866677,  0.00488404],
       [-0.00056316, -0.02705587,  0.00410378],
       [ 0.00048268,  0.01692128, -0.00262183]])
```

```
hidden_layer
```


```
array([[0.5       , 0.5       , 0.5       ],
       [0.5885562 , 0.35962319, 0.38485296],
       [0.39555998, 0.32300414, 0.27667802],
       [0.48350599, 0.21131785, 0.19309868]])
```

```
delta_output
```

```
array([[-0.0978763 ],
       [ 0.13939397],
       [ 0.138553  ],
       [-0.11369557]])
```

- We need to multiply the "inputs" by "delta" however, for the matrix multiplication we need to transpose the values in the hidden_layer, so we have all of them on one row for each neuron

```
hidden_layerT = hidden_layer.T
hidden_layerT
```

```
array([[0.5       , 0.5885562 , 0.39555998, 0.48350599],
       [0.5       , 0.35962319, 0.32300414, 0.21131785],
       [0.5       , 0.38485296, 0.27667802, 0.19309868]])
```

```
input_x_delta1 = hidden_layerT.dot(delta_output)
input_x_delta1
```

```
array([[0.03293657],
       [0.02191844],
       [0.02108814]])
```

```
weights_1 = weights_1 + (input_x_delta1 * learning_rate)
weights_1
```

```
array([[-0.00711903],
       [-0.88642447],
       [ 0.15432644]])
```

```
# First column is X1, and 2nd column is X2 (our input values )

input_layer
```

```
array([[0, 0],
       [0, 1],
       [1, 0],
       [1, 1]])
```

```
delta_hidden_layer
```

```
array([[ 0.00041597,  0.02185088, -0.00362142],
       [-0.00057384, -0.02866677,  0.00488404],
       [-0.00056316, -0.02705587,  0.00410378],
       [ 0.00048268,  0.01692128, -0.00262183]])
```

```
# we need to transpose the values just as we did before

input_layerT = input_layer.T
input_layerT
```

```
array([[0, 0, 1, 1],
       [0, 1, 0, 1]])
```

```
input_x_delta0 = input_layerT.dot(delta_hidden_layer)
input_x_delta0
```

```
array([[-8.04778516e-05, -1.01345901e-02,  1.48194623e-03],
       [-9.11603819e-05, -1.17454886e-02,  2.26221011e-03]])
```

```
weights_0 = weights_0 + (input_x_delta0 * learning_rate)
weights_0
```

```
array([[-0.42402414, -0.74304038, -0.96055542],
       [ 0.35797265, -0.58052365, -0.46832134]])
```

So all the lines of code above, has allowed us to complete our first epoch. we will need to put all the code together so we can run multiple epochs

```python
#Importing Numpy
import numpy as np

# This is the sigmoid Function
def sigmoid(sum):
  return 1 / (1 + np.exp(-sum))

#This is the sigmoid derivative as used before
def sigmoid_derivative(sigmoid):
  return sigmoid * (1 - sigmoid)

# Our input values
inputs = np.array([[0,0],
                   [0,1],
                   [1,0],
                   [1,1]])
#Our output values
outputs = np.array([[0],
                    [1],
                    [1],
                    [0]])
```

```python
# weights_0 = np.array([[-0.424, -0.740, -0.961],
#                       [0.358, -0.577, -0.469]])

# weights_1 = np.array([[-0.017],
#                       [-0.893],
#                       [0.148]])
```

- **Note:** Multiplying the random number by 2 and subtracting by 1, allows us to have a mix of both positive and negative random numbers for the weights

```python
weights_0 = 2 * np.random.random((2, 3)) - 1
weights_1 = 2 * np.random.random((3, 1)) - 1
```

```python
epochs = 400000
learning_rate = 0.6
error = []

for epoch in range(epochs):
  input_layer = inputs
  sum_synapse0 = np.dot(input_layer, weights_0)
  hidden_layer = sigmoid(sum_synapse0)

  sum_synapse1 = np.dot(hidden_layer, weights_1)
  output_layer = sigmoid(sum_synapse1)

  error_output_layer = outputs - output_layer
  average = np.mean(abs(error_output_layer))
    #print after every specified range of the value
  if epoch % 100000 == 0:
    print('Epoch: ' + str(epoch + 1) + ' Error: ' + str(average))
    error.append(average)

  derivative_output = sigmoid_derivative(output_layer)
  delta_output = error_output_layer * derivative_output

  weights1T = weights1.T
  delta_output_weight = delta_output.dot(weights1T)
  delta_hidden_layer = delta_output_weight * 
sigmoid_derivative(hidden_layer)

  hidden_layerT = hidden_layer.T
  input_x_delta1 = hidden_layerT.dot(delta_output)
  weights_1 = weights_1 + (input_x_delta1 * learning_rate)

  input_layerT = input_layer.T
  input_x_delta0 = input_layerT.dot(delta_hidden_layer)
  weights_0 = weights_0 + (input_x_delta0 * learning_rate)
```

The value is low after running 1 million epochs

```python
#1 million epochs with a learning rate of 0.3
1 - 0.009670967930930745
```

```
0.9903290320690693
```

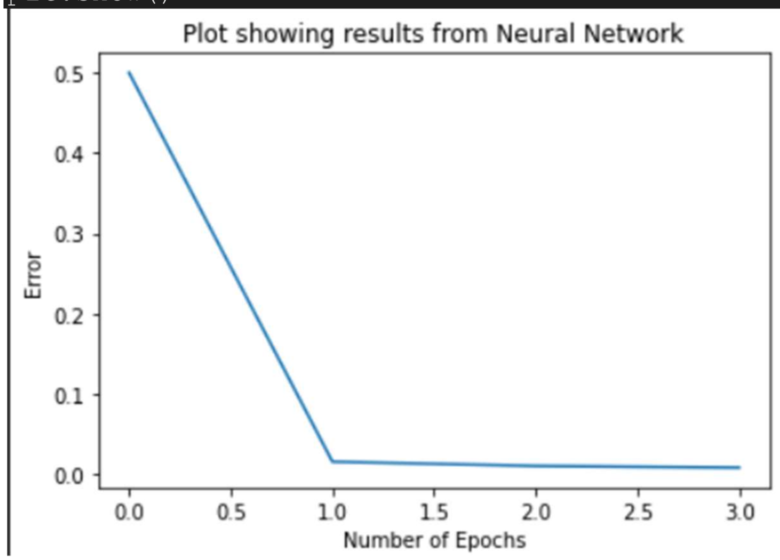
```
#after 400,000 epochs, with a learning rate of 0.6
1- 0.008192022809586367
```

```
0.9918079771904136
```

To visualise the result we will now import matplotlib

```
import matplotlib.pyplot as plt
```

```
plt.xlabel('Number of Epochs')
plt.ylabel('Error')
plt.title('Plot showing results from Neural Network')
plt.plot(error)
plt.show()
```




```
outputs
```

```
array([[0],
       [1],
       [1],
       [0]])
```

We are now comparing the outputs and predictions

```
output_layer
```

```
array([[0.36635804],
       [0.37941907],
       [0.38426045],
       [0.39649774]])
```

We see that our neural network was able to get values close to the actual values from the results.

This shows that our neural network can handle the complexity of the XOR operator dataset.

Let us see the updated weights. These are the weights we will require if we want to make future predictions

```
weights_0
```

```
array([[-0.83235059, -0.38039511,  0.81885728],
       [ 0.32052693, -0.54903926, -0.24576517]])
```

```
weights_1
```

```
array([[-0.35156863],
       [-0.52620914],
       [-0.21796815]])
```

```
# This function accepts an instance of a dataset

def calculate_output(instance):
    #input to hidden layer
    hidden_layer = sigmoid(np.dot(instance, weights_0))
    #hidden to output layer
    output_layer = sigmoid(np.dot(hidden_layer, weights_1))
    return output_layer[0]
```

```
round(calculate_output(np.array([0, 0])))
```

```
0
```

```
round(calculate_output(np.array([0, 1])))
```

```
0
```

```
round(calculate_output(np.array([1, 0])))
```

```
0
```

```
round(calculate_output(np.array([1, 1])))
```

```
0
```