CIS 3187 Assignment - Implementing a Multi-Layer Perceptron

Matthew Cachia - *0202398M*

FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

Declaration

Plagiarism is defined as "the unacknowledged use, as one's own, of work of another person, whether or not such work has been published, and as may be further elaborated in Faculty or University guidelines" (University Assessment Regulations, 2009, Regulation 39 (b)(i), University of Malta).

I / We*, the undersigned, declare that the [assignment / Assigned Practical Task report / Final Year Project report] submitted is my / our* work, except where acknowledged and referenced.

I / We* understand that the penalties for committing a breach of the regulations include loss of marks; cancellation of examination results; enforced suspension of studies; or expulsion from the degree programme.

Work submitted without this signed declaration will not be corrected, and will be given zero marks.

* Delete as appropriate.

(N. B. If the assignment is meant to be submitted anonymously, please sign this form and submit it to the Departmental Officer separately from the assignment).

Matthew Cachia
Student Name

Signature

Student Name

Signature

Student Name

Signature

Student Name

Signature

CPS 3187
Course Code

CIS 3187 Assignment - Implementing a Multi-Layer Perceptron
Title of work submitted

13/01/2020
Date

Statement Of Completion

Task	Completed
Creation of Boolean Function	Yes
Splitting & Randomising Data Sets	Yes
Feed Forward Method	Yes
Back Propagation Method	Yes
Sigmoid Function	Yes
Converging Network Under 1000 epochs	Yes
Bad Facts vs Epochs Graph	Yes
Working Neural Network	Yes

Artificial Neural Network

Introduction

According to Thomas [1] Artificial Neural Networks are software implementations of the neuronal structure of our brains. An Artificial Neural Network is made of a number of interconnected neurons which during training adjust how strongly they are connected.

Problems Encountered

The biggest problem encountered throughout the development of the ANN was the multiplication aspect of the matrices. At various points during early stages of development matrix multiplication was problematic as the process was not yet fully understood.
Another problem encountered was splitting and shuffling of the data set, since I opted to store the input and output into two separate arrays.

Data Set

The Data Set given to the Neural Net is a Boolean function mapping 5 bits into 3 bits through the function ABCDE $\rightarrow$ AB$\neg$E. The data set contained a total of 32 items which would then be divided into a training set and a testing set.

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

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

X is a Numpy array which contains all the inputs while Y is a Numpy array which contains all the outputs. In order to randomise and split the dataset, both arrays are shuffled by index in order to maintain the links between them, then the shuffled arrays are split into a training set and a test set.

```
arr_rand = np.random.rand(X.shape[0])
split = arr_rand < np.percentile(arr_rand, 80)
X_TRAIN = X[split]
Y_TRAIN = Y[split]
X_TEST = X[~split]
Y_TEST = Y[~split]
```

Class: Artificial Neural Network

Method: Init

In the init method of the ANN, the input, hidden and output sizes are defined. The weights are given a randomly decided number between -1 and 1, and are set to the sizes of the input, hidden and output.

```
def __init__(self):
        self.input_size = 5
        self.hidden_size = 4
        self.output_size = 3

        self.weight1 = np.random.uniform(low=-1, high=1,
                  size=(self.input_size, self.hidden_size))
        self.weight2 = np.random.uniform(low=-1, high=1,
                  size=(self.hidden_size, self.output_size))
```

Method: Sigmoid

The sigmoid method takes 2 parameters, a numeric value and a Boolean value which unless stated otherwise is false. If the Boolean value is false then the method returns the Sigmoid function of the numeric value and if the Boolean value is true it returns integral of the numeric values.

```
def sigmoid(self, S, deriv=False):
        if deriv:
            return S * (1 - S)
        return 1 / (1 + np.exp(-S))
```

Method: FeedForward

The feedforward method takes 2 parameters X and Y which are the input and the expected output respectively. The feedforward methods calculate the output of the hidden weights and the outputs of the output weights. Then returns the output values of the weights as well as the error list which is the difference between the expected output and the actual output.

$$\delta = OUT(1-OUT)(Target-OUT)$$

$$\Delta w_{pq,k} = \eta\ \delta_{q,k}\ OUT_{p,j}$$

$$w_{pq,k} = w_{pq,k} + \Delta w_{pq,k}$$

```
def feedforward(self, X, Y):
    self.netH = np.dot(X, self.weight1)
    outH = self.sigmoid(self.netH)
    self.netO = np.dot(outH, self.weight2)
    outO = self.sigmoid(self.netO)

    self.errorList = np.zeros((3, 1))
    i = 0
    for num in outO:
        self.update = np.subtract(Y[i], num)
        self.errorList[i] = self.update
        i += 1

    return outH, outO, self.errorList
```

Method: Backward Propagation

The backward propagation method takes 5 parameters, the input, the expected output, the output of the output and hidden layer returned from the feedforward method and the learning rate. The backward propagation algorithm aims to minimise the error between the actual output and the output returned by the Feed Forward algorithm. It does this through the use of 2 formulas.

$$\delta_{p,j} = OUTp_{,j}(1 - OUTp_{,j}) \left(\sum_{q} \delta_{q,k}\, wp_{q,k}\right)$$

$$\Delta w_{pq,j} = \eta\, \delta_{q,j}\, OUT_{p,i}$$

$$w_{pq,j} = w_{pq,j} + \Delta w_{pq,j}$$

```
def backword_propogation(self, X, Y, outH, outO, learning_rate):
    self.out_delta = self.sigmoid(outO, deriv=True) * (Y - outO)

    for i in range(self.hidden_size):
        for j in range(self.output_size):
            self.weight2[i][j] += learning_rate * self.out_delta[j] * outH[i]

    for i in range(self.input_size):
        for j in range(self.hidden_size):
            self.hidden_delta = self.sigmoid(outH, deriv=True) \
                * self.summation_delta_weight(self.out_delta, self.weight2[j])

            self.weight1[i][j] += learning_rate * self.hidden_delta[j] * X[i]
```

Method: Summation Delta Weights

This method takes 2 parameters, within the ANN this method is used to calculate the summation of the delta and weight of the output layer as described in the function below. This value is used to calculate the hidden layer's delta.

$$\left(\sum_{q} \delta_{q,k}\, wp_{q,k}\right)$$

```
def summation_delta_weight(self, out_deltas, weight):
        product = 0
        for x, out_delta in enumerate(out_deltas):
            product += out_delta * weight[x]
        return product
```

Method: Is Bad Fact

This method takes the error list returned by the feed forward method and determines if there are any bad facts, if bad facts are found then it returns False.

```
def is_bad_fact(self, error_list):
        mu = 0.2
        for error in error_list:
            if abs(error) > mu:
                return False
        return True
```

Method: Plot Graph Bad Facts vs Epochs

This method takes a single parameter and uses it to generate a graph. MatPlotLib was used to draw and generate the graph. The graph represents the number of bad facts found in each epoch.

```
def plot_graph_bad_facts_vs_epochs(self, graph):


        plt.plot(graph)
        plt.xlabel("Epochs")
        plt.ylabel("Bad Facts")
        print(plt.show())
```

Method: Train

This method takes 3 parameters the input, expected output and the amount of epochs to train the network. The network runs the inputs through the feedforward method, the error list returned via the feedforward method is then passed to the isBadFact method to verify if any bad facts exist. If there is indeed a bad fact the the weights of the network will be adjusted via the back propagation method.

```
def train(self, X, Y, epochs):
        learning_rate = 0.2
        epoch_number = 0
        ending = False
        epochs_vs_bad_facts_graph = np.zeros(epochs)
        # np.zeros(epochs_vs_bad_facts_graph)
        while epoch_number < epochs:
            bad_fact_number = 0
            if not ending:
                for j in range(len(X)):
                    output = self.feedforward(X[j], Y[j])

                    bad_fact = self.is_bad_fact(output[2])

                    if not bad_fact:
                        bad_fact_number += 1
                        self.backword_propogation(X[j], Y[j], output[0], output[1],
                              learning_rate)

                    print("Bad Facts in epoch:" + str(epoch_number) + " | "
                          + str(bad_fact_number))

                    epochs_vs_bad_facts_graph[epoch_number] = bad_fact_number

                epoch_number += 1

        self.plot_graph_bad_facts_vs_epochs(epochs_vs_bad_facts_graph)
```

Calling and Testing the Network

Calling the Network

The class artificial_neural_network is instantiated, the training sets and the number of epochs are passed to the train method.

```
ANN = artificial_neural_network()
ANN.train(X_TRAIN, Y_TRAIN, 999)
```

Testing the Network

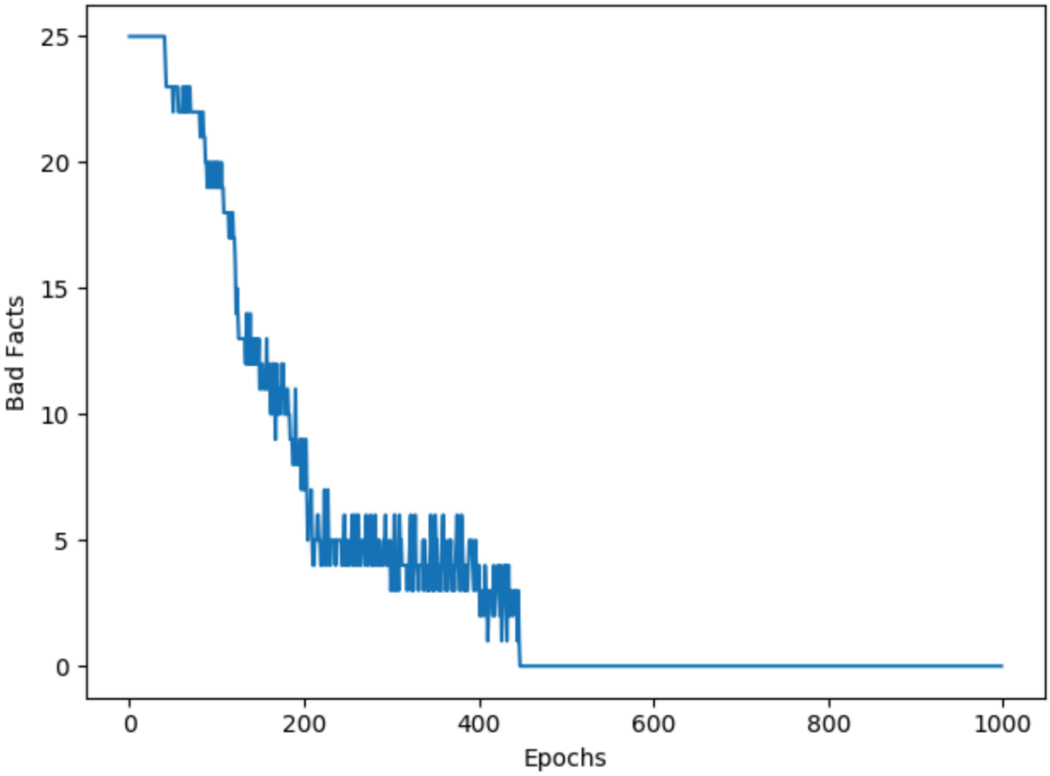
The network is tested by passing the test data sets to the feedforward method and verifying that the expected output is the same as the actual output.

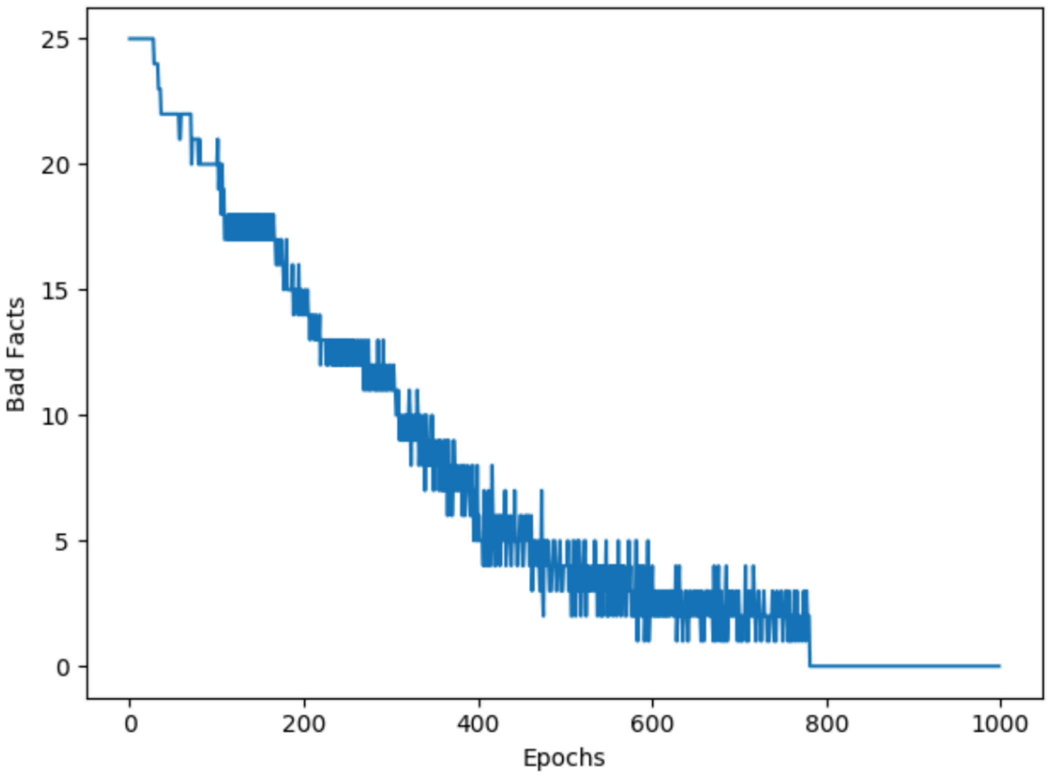
```
for x in range(len(X_TEST)):
    output = ANN.feedforward(X_TEST[x], Y_TEST[x])
    print(" Input " + str(X_TEST[x]) + " Expected Output " +
          str(Y_TEST[x]) + " Actual Output" + str(np.around(output[1])))
```

```
Input [0 1 0 0 0] Expected Output [1 0 1] Actual Output[1. 0. 1.]
Input [0 1 1 0 0] Expected Output [1 1 1] Actual Output[1. 1. 1.]
Input [1 0 0 0 0] Expected Output [1 0 1] Actual Output[1. 0. 1.]
Input [1 0 0 1 0] Expected Output [1 1 1] Actual Output[1. 1. 1.]
Input [1 0 1 0 0] Expected Output [1 1 1] Actual Output[1. 1. 1.]
Input [1 1 0 0 0] Expected Output [1 0 1] Actual Output[1. 0. 1.]
Input [1 1 1 0 0] Expected Output [1 1 1] Actual Output[1. 1. 1.]
```

Bad Facts vs Epochs Graph

The graph measures the amount of bad facts within an epoch. As can be seen in the 2 examples below, the amount of bad facts keeps reducing until 0 bad facts are found. The graph generated is non-monotonic.





CODE

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

X = np.array(([0, 0, 0, 0, 0], [0, 0, 0, 0, 1]
              , [0, 0, 0, 1, 0], [0, 0, 0, 1, 1]
              , [0, 0, 1, 0, 0], [0, 0, 1, 0, 1]
              , [0, 0, 1, 1, 0], [0, 0, 1, 1, 1]
              , [0, 1, 0, 0, 0], [0, 1, 0, 0, 1]
              , [0, 1, 0, 1, 0], [0, 1, 0, 1, 1]
              , [0, 1, 1, 0, 0], [0, 1, 1, 0, 1]
              , [0, 1, 1, 1, 0], [0, 1, 1, 1, 1]
              , [1, 0, 0, 0, 0], [1, 0, 0, 0, 1]
              , [1, 0, 0, 1, 0], [1, 0, 0, 1, 1]
              , [1, 0, 1, 0, 0], [1, 0, 1, 0, 1]
              , [1, 0, 1, 1, 0], [1, 0, 1, 1, 1]
              , [1, 1, 0, 0, 0], [1, 1, 0, 0, 1]
              , [1, 1, 0, 1, 0], [1, 1, 0, 1, 1]
              , [1, 1, 1, 0, 0], [1, 1, 1, 0, 1]
              , [1, 1, 1, 1, 0], [1, 1, 1, 1, 1]))

Y = np.array(([0, 0, 1], [0, 0, 0]
              , [0, 1, 1], [0, 1, 0]
              , [0, 1, 1], [0, 1, 0]
              , [0, 1, 1], [0, 1, 0]
              , [1, 0, 1], [1, 0, 0]
              , [1, 1, 1], [1, 1, 0]
              , [1, 1, 1], [1, 1, 0]
              , [1, 1, 1], [1, 1, 0]
              , [1, 0, 1], [1, 0, 0]
              , [1, 1, 1], [1, 1, 0]
              , [1, 1, 1], [1, 1, 0]
              , [1, 1, 1], [1, 1, 0]
              , [1, 0, 1], [1, 0, 0]
              , [1, 1, 1], [1, 1, 0]
              , [1, 1, 1], [1, 1, 0]
              , [1, 1, 1], [1, 1, 0]))

arr_rand = np.random.rand(X.shape[0])
split = arr_rand < np.percentile(arr_rand, 80)
X_TRAIN = X[split]
Y_TRAIN = Y[split]
```

```
X_TEST = X[~split]
Y_TEST = Y[~split]


class artificial_neural_network(object):

    def __init__(self):
        self.input_size = 5
        self.hidden_size = 4
        self.output_size = 3

        self.weight1 = np.random.uniform(low=-1, high=1, size=(self.input_size,
self.hidden_size))
        self.weight2 = np.random.uniform(low=-1, high=1, size=(self.hidden_size,
self.output_size))

    def sigmoid(self, S, deriv=False):
        if deriv:
            return S * (1 - S)
        return 1 / (1 + np.exp(-S))

    def feedforward(self, X, Y):
        self.netH = np.dot(X, self.weight1)
        outH = self.sigmoid(self.netH)
        self.netO = np.dot(outH, self.weight2)
        outO = self.sigmoid(self.netO)

        self.errorList = np.zeros((3, 1))
        i = 0
        for num in outO:
            self.update = np.subtract(Y[i], num)
            self.errorList[i] = self.update
            i += 1

        return outH, outO, self.errorList

    def summation_delta_weight(self, out_deltas, weight):
        product = 0
        for x, out_delta in enumerate(out_deltas):
            product += out_delta * weight[x]
        return product

    def backword_propogation(self, X, Y, outH, outO, learning_rate):
        self.out_delta = self.sigmoid(outO, deriv=True) * (Y - outO)
```

```
        for i in range(self.hidden_size):
            for j in range(self.output_size):
                self.weight2[i][j] += learning_rate * self.out_delta[j] *
outH[i]

        for i in range(self.input_size):
            for j in range(self.hidden_size):
                self.hidden_delta = self.sigmoid(outH, deriv=True) \
                                    *
self.summation_delta_weight(self.out_delta, self.weight2[j])

                self.weight1[i][j] += learning_rate * self.hidden_delta[j] *
X[i]

    def is_bad_fact(self, error_list):
        mu = 0.2
        for error in error_list:
            if abs(error) > mu:
                return False
        return True

    def plot_graph_bad_facts_vs_epochs(self, graph):

        plt.plot(graph)
        plt.xlabel("Epochs")
        plt.ylabel("Bad Facts")
        print(plt.show())

    def train(self, X, Y, epochs):
        learning_rate = 0.2
        epoch_number = 0
        ending = False
        epochs_vs_bad_facts_graph = np.zeros(epochs)
        # np.zeros(epochs_vs_bad_facts_graph)
        while epoch_number < epochs:
            bad_fact_number = 0
            if not ending:
                for j in range(len(X)):
                    output = self.feedforward(X[j], Y[j])

                    bad_fact = self.is_bad_fact(output[2])

                    if not bad_fact:
```

```
                        bad_fact_number += 1
                        self.backword_propogation(X[j], Y[j], output[0],
output[1], learning_rate)

                    print("Bad Facts in epoch:" + str(epoch_number) + " | "
                          + str(bad_fact_number))

                    epochs_vs_bad_facts_graph[epoch_number] = bad_fact_number

                epoch_number += 1

        self.plot_graph_bad_facts_vs_epochs(epochs_vs_bad_facts_graph)


ANN = artificial_neural_network()
ANN.train(X_TRAIN, Y_TRAIN, 999)
for x in range(len(X_TEST)):
    output = ANN.feedforward(X_TEST[x], Y_TEST[x])
    print(" Input " + str(X_TEST[x]) + " Expected Output " +
          str(Y_TEST[x]) + " Actual Output" + str(np.around(output[1])))
```

References

[1] A. Thomas, An introduction to neural networks for beginners. 2019.