COMP 575 Coursework

Name: Orion Assefaw

Student ID: 201530497

Email: o.assefaw@liverpool.ac.uk

Date: 14/05/2021

PART-1

1. Introduction and Design

A **Single Layer Perceptron (SLP)** was designed using a **Least mean squares (LMS)** algorithm in python. This learning algorithm is based on a sample-by-sample learning of the instantaneous cost function values and is used for **binary** classification problems - a common machine learning task, which involves predicting whether a given example is part of one class or the other.

The algorithm works in such a way that it iterates many times until it finds a good classifier. And in the process, it learns the weights for the input signals in order to draw a linear decision boundary. This enables it to distinguish between the two linearly separable classes, usually +1 and -1.

How the algorithm is designed is that, it Initially sets the weight and bias to zero. The perceptron then calculates the induced field by adding the products of the weights(wi's) and features (xi's), plus the bias, given by (θ) below, and passes that through an activation function ϕ, i.e. :

$$\mathrm{v_j} = \sum wixi + \theta ....(1)$$

$$\mathrm{y_j} = \phi(\mathrm{v_j}) ....(2)$$

In cases where the result of this activation score $\mathrm{y_j}$ and true (desired) output do not coincide, then the current weight gets updated to the sum of the current weight and the product of the learning rate (eta), difference between desired output and score (what is known as **error**), and the feature xi. i.e.,

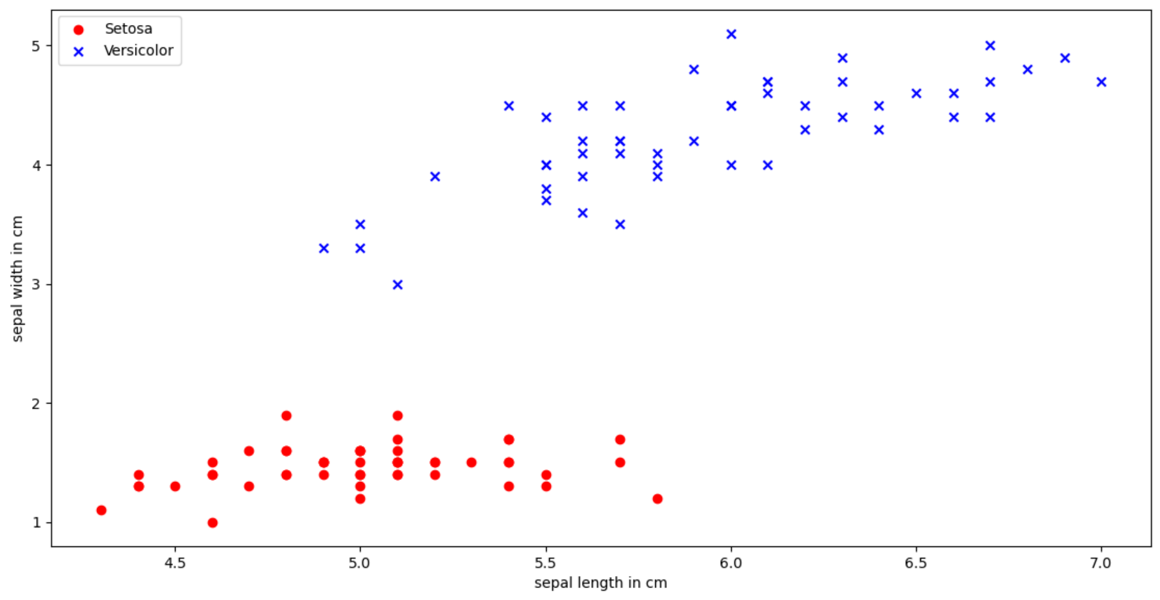
$$w(n+1) = w(n) + \eta e(n)x(n) ....(3)$$

All the above stated steps are done in the **fit** function of the python code which finally returns the adopted or trained weights and bias. For more details on how code was written and structured, the separately submitted **"mysingleperceptron.py"** file can be referred, and only a snippet code of the fit function is given below.

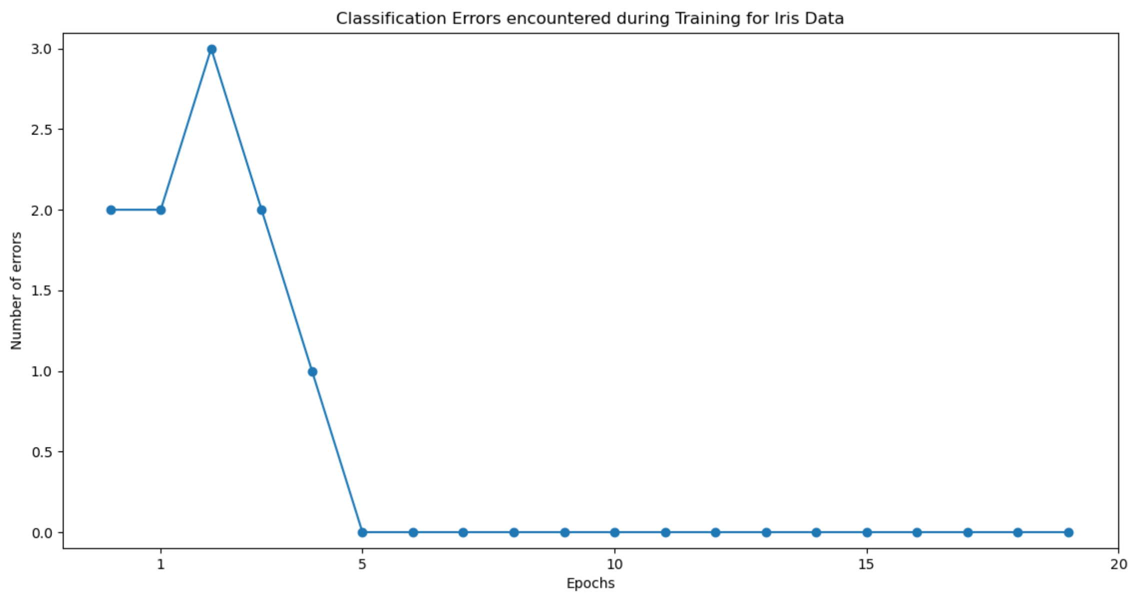
```
def fit(self, X, y):
        self.X = X
        self.y = y
        nsamples = self.X.shape[0]
        nfeatures = self.X.shape[1]
        self.weights = np.zeros(nfeatures)
        self.bias = 0
        self.errors = []

        labelsList = []
        for i in self.y:
            if i > 0:
                i = 1
                labelsList.append(i)
            else:
                i = -1
                labelsList.append(i)
        yLabel = np.asarray(labelsList)
        print("Starting weights:", self.weights)
        print("Starting bias:", self.bias)

        for epoch in range(self.iterations):
            errors = 0
            for pos, x in enumerate(self.X):
                inducedLocalField = np.dot(x,self.weights) + self.bias
                yPredicted = self.activation(inducedLocalField)

                # error = Desired output minus Predicted output
                error = yLabel[pos] - yPredicted
                if error != 0:
                    errors+=1
                    # w(n+1) = w(n) + learning rate * error * x(n)
                    self.weights += self.eta*error*x
                    # bias(n+1) = bias(n) + learning rate * error
                    self.bias += self.eta*error

            print("Epoch",epoch,"weights:", self.weights)
            print("Epoch",epoch,"bias:", self.bias)

            self.errors.append(errors)

        return self.bias, self.weights
```

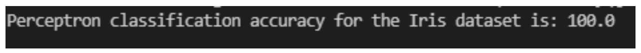
2. Results and Discussions

The designed SLP was experimented in classifying two different sets of data. The first one was the Iris Dataset - in classifying **setosa** flower species from **versicolor** ones (the third species "virginica" was not considered in this study), and the second one was a randomly created synthetic dataset of 200 data points. In both cases the input vector X was a 2 dimensional vector, where in the former case only **sepal length** and **sepal width** where considered out of the 4 original features. In order to visualise the data and understand it's pattern, a scatter plot of the Iris dataset was done, for both flower species as follows:

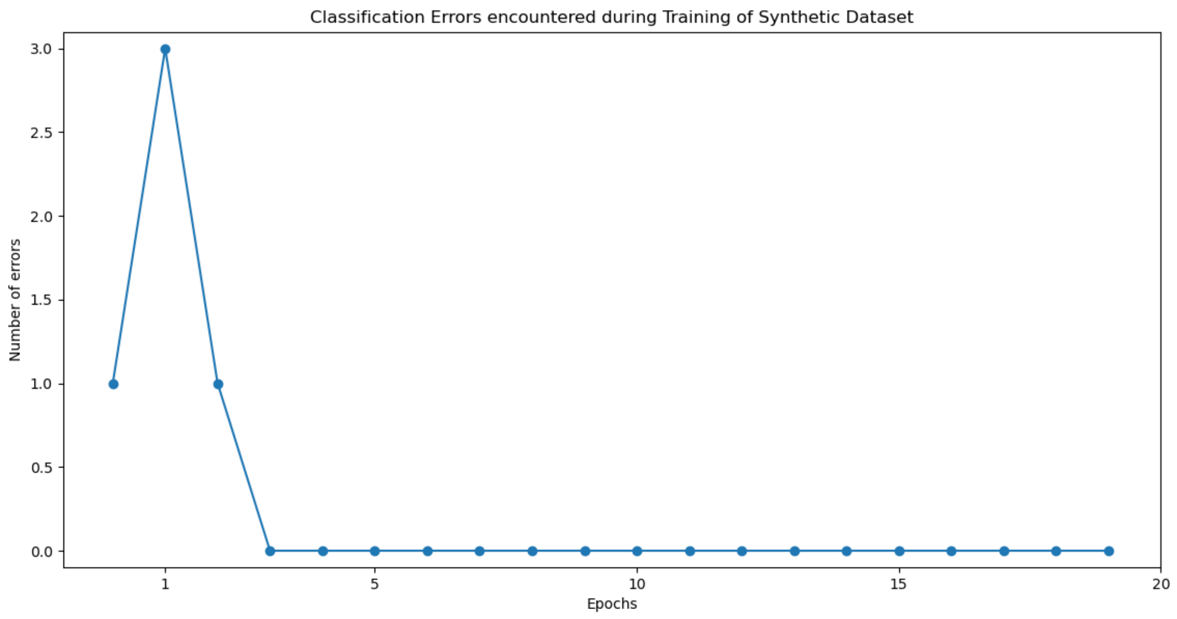


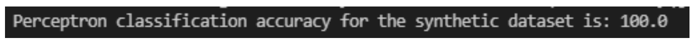
The algorithm was run for 20 iterations and as can be seen from the 2 graphs below, in both sets of data, no errors (misclassifications) occurred after the 5 iterations (or **epochs**). In fact in the case of the synthetic dataset, it took only 3 epochs (epochs 0,1, and 2) for the perceptron to accurately classify all the data points and an accuracy of 100% was reported for both datasets.





```
Perceptron classification accuracy for the Iris dataset is: 100.0
```

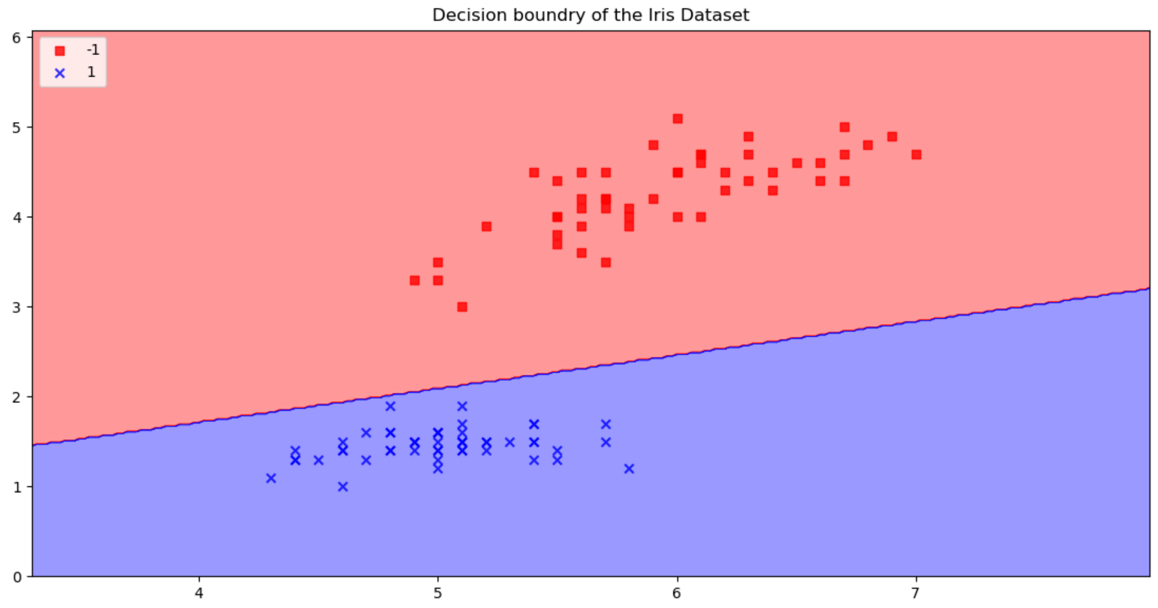


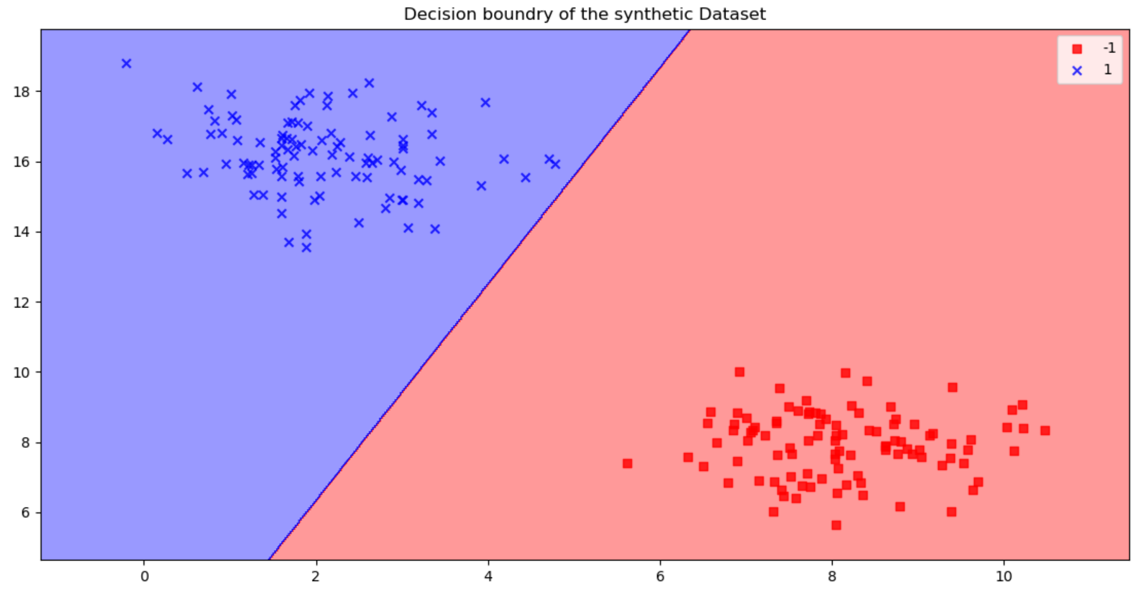


```
Perceptron classification accuracy for the synthetic dataset is: 100.0
```

Moreover, decision boundaries were plotted for both datasets and clearly show that the trained model successfully came up with a good classifier. The decision boundary is a hyperplane given by the following equation :

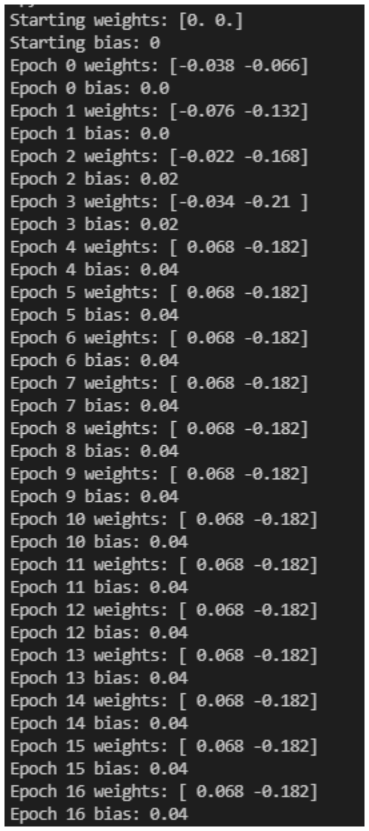
$$\sum wixi + \theta = 0 \ldots\ldots\ldots\ldots\ldots\ldots (4)$$

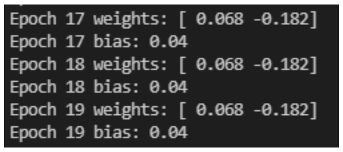


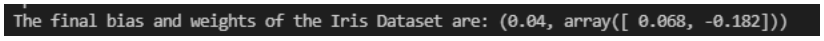


Lastly, the screen shot of the weight adaptations and biases in each of the 20 epochs is displayed so as to learn how the weights were changing in both datasets. As mentioned earlier, we do not see any weight or bias changes on and after epoch 5, as the classifier finished its learning before that.

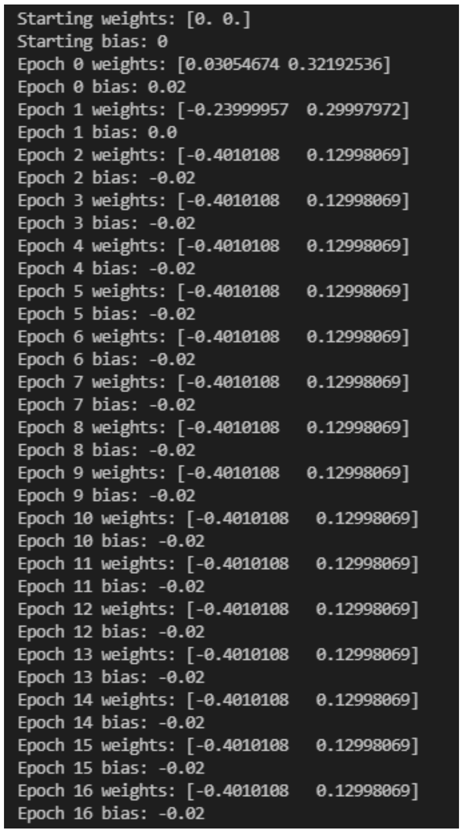
For Iris dataset:


```
Starting weights: [0. 0.]
Starting bias: 0
Epoch 0 weights: [-0.038 -0.066]
Epoch 0 bias: 0.0
Epoch 1 weights: [-0.076 -0.132]
Epoch 1 bias: 0.0
Epoch 2 weights: [-0.022 -0.168]
Epoch 2 bias: 0.02
Epoch 3 weights: [-0.034 -0.21 ]
Epoch 3 bias: 0.02
Epoch 4 weights: [ 0.068 -0.182]
Epoch 4 bias: 0.04
Epoch 5 weights: [ 0.068 -0.182]
Epoch 5 bias: 0.04
Epoch 6 weights: [ 0.068 -0.182]
Epoch 6 bias: 0.04
Epoch 7 weights: [ 0.068 -0.182]
Epoch 7 bias: 0.04
Epoch 8 weights: [ 0.068 -0.182]
Epoch 8 bias: 0.04
Epoch 9 weights: [ 0.068 -0.182]
Epoch 9 bias: 0.04
Epoch 10 weights: [ 0.068 -0.182]
Epoch 10 bias: 0.04
Epoch 11 weights: [ 0.068 -0.182]
Epoch 11 bias: 0.04
Epoch 12 weights: [ 0.068 -0.182]
Epoch 12 bias: 0.04
Epoch 13 weights: [ 0.068 -0.182]
Epoch 13 bias: 0.04
Epoch 14 weights: [ 0.068 -0.182]
Epoch 14 bias: 0.04
Epoch 15 weights: [ 0.068 -0.182]
Epoch 15 bias: 0.04
Epoch 16 weights: [ 0.068 -0.182]
Epoch 16 bias: 0.04
```

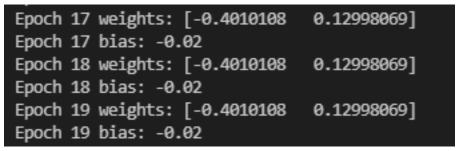

```
Epoch 17 weights: [ 0.068 -0.182]
Epoch 17 bias: 0.04
Epoch 18 weights: [ 0.068 -0.182]
Epoch 18 bias: 0.04
Epoch 19 weights: [ 0.068 -0.182]
Epoch 19 bias: 0.04
```


```
The final bias and weights of the Iris Dataset are: (0.04, array([ 0.068, -0.182]))
```

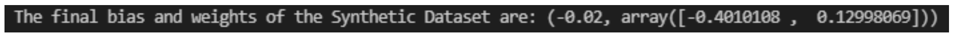
For Synthetic Dataset:



```
Starting weights: [0. 0.]
Starting bias: 0
Epoch 0 weights: [0.03054674 0.32192536]
Epoch 0 bias: 0.02
Epoch 1 weights: [-0.23999957  0.29997972]
Epoch 1 bias: 0.0
Epoch 2 weights: [-0.4010108   0.12998069]
Epoch 2 bias: -0.02
Epoch 3 weights: [-0.4010108   0.12998069]
Epoch 3 bias: -0.02
Epoch 4 weights: [-0.4010108   0.12998069]
Epoch 4 bias: -0.02
Epoch 5 weights: [-0.4010108   0.12998069]
Epoch 5 bias: -0.02
Epoch 6 weights: [-0.4010108   0.12998069]
Epoch 6 bias: -0.02
Epoch 7 weights: [-0.4010108   0.12998069]
Epoch 7 bias: -0.02
Epoch 8 weights: [-0.4010108   0.12998069]
Epoch 8 bias: -0.02
Epoch 9 weights: [-0.4010108   0.12998069]
Epoch 9 bias: -0.02
Epoch 10 weights: [-0.4010108   0.12998069]
Epoch 10 bias: -0.02
Epoch 11 weights: [-0.4010108   0.12998069]
Epoch 11 bias: -0.02
Epoch 12 weights: [-0.4010108   0.12998069]
Epoch 12 bias: -0.02
Epoch 13 weights: [-0.4010108   0.12998069]
Epoch 13 bias: -0.02
Epoch 14 weights: [-0.4010108   0.12998069]
Epoch 14 bias: -0.02
Epoch 15 weights: [-0.4010108   0.12998069]
Epoch 15 bias: -0.02
Epoch 16 weights: [-0.4010108   0.12998069]
Epoch 16 bias: -0.02
```



```
Epoch 17 weights: [-0.4010108   0.12998069]
Epoch 17 bias: -0.02
Epoch 18 weights: [-0.4010108   0.12998069]
Epoch 18 bias: -0.02
Epoch 19 weights: [-0.4010108   0.12998069]
Epoch 19 bias: -0.02
```



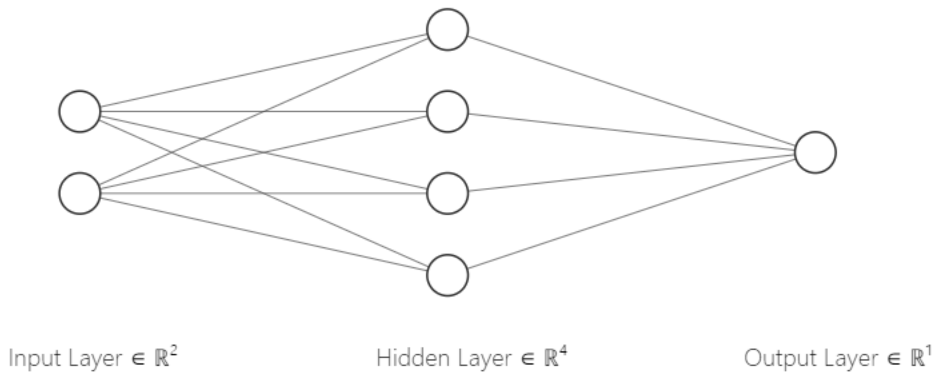
```
The final bias and weights of the Synthetic Dataset are: (-0.02, array([-0.4010108 ,  0.12998069]))
```

Part-2

1. Introduction and Design

A **multi-layer perceptron (MLP)** consists of at least three layers of nodes: an input layer, a hidden layer and an output layer. Except for the input nodes, each node is a neuron that uses a nonlinear activation function. This allows MLPs to learn more complex patterns and data that is not linearly separable.

In this project, a fully connected **3 layer** MLP was designed, using python code, with the following architechture: *2* input nodes in the **input layer**, *4* nodes in the **hidden layer**, and *1* node in the **output layer** making the entire network a 2 X 4 X 1.



The network was trained using a "backpropagation" algorithm (a gradient descent method, which, given an artificial neural network (ANN) and an error function, calculates the gradient of the error function with respect to the neural network's weights) to try and optimise the weights.

Training of the network was done in such a way that the weights and biases are all first randomly initialised and then go through the two passes of backpropagation : a **Forward Pass** and a **Backward Pass**.

A. Forward Pass

During this pass, a straightforward dot matrix operation of the weights and inputs was performed, followed up by adding the bias term, as shown in the code snippet below:

```
def forwardPropagation(self, X, sigmoid = True):
        # 1. Input Layer-to-Hidden Layer

        # Induced Local field (summation of (Input(X)*weights1) + bias1)
        self.v1 = np.dot(X,self.weights1) + self.bias1.T
        # Activation of the induced local field
        if sigmoid:
            self.a1 = self.sigmoid(self.v1)
        else:
            self.a1 = self.tanh(self.v1)

        # 2. Hidden Layer-to-Output Layer

        # Induced Local field (summation of (a1*weights2) + bias2 )
        self.v2 = np.dot(self.a1,self.weights2) + self.bias2.T
        # Activation of the induced local field
        if sigmoid:
            self.a2 = self.sigmoid(self.v2)
        else:
            self.a2 = self.tanh(self.v2)

        return self.a2
```

The weights were either **randomly** initialised or initialised using **Xavier's** weight initialisation method. Xavier's method for weight initialisation is when weights are picked from a Gaussian distribution with zero mean and a variance of 1/N, where N specifies the number of input neurons.

The result of this dot matrix of the weights and inputs then yields to the pre-activation value (induced local field, denoted by self.v1 in the code snippet above) for the hidden layer. That value is then passed through a choice of non-linearity activation function **sigmoid** or the **hyperbolic tangent function** (**tanh**), before being fed forward as an input to the last (output) layer's dot matrix operation of weights and that value. Mathematically speaking:

In the hidden layer : $\mathrm{v_j} = \sum wixi + \theta$, then $\mathrm{y_j} = \phi(\mathrm{v_j})$

and In the output layer : $\mathrm{v_j} = \sum wi\mathrm{y_j} + \theta \quad , then \quad \mathrm{y_j} = \phi(\mathrm{v_j})$

B. Backward Pass

The backward pass process begins from the last (output) layer and works backwards in a layer-by-layer fashion, until it reaches the first (input) layer. It first starts by computing the network error, which is the desired output minus the output produced by the network ,i.e.,

$$\mathrm{e_j}(n) = \mathrm{d_j}(n) - \mathrm{y_j}(n)\text{..........................}(5)$$

Then the derivatives(gradients) of the weights and biases are computed and propagated across the network. In this way, the network gets a feel of the contributions of each individual units, and adjusts itself accordingly so that the weights and biases are optimal. Mathematically,

In the output layer:

$$\delta_\mathrm{j}(n) = \mathrm{e_j}(n)\Phi'[\mathrm{v_j}(n)]\text{..............................}(6)$$

In the hidden layer:

$$\delta_\mathrm{j}(n) = \Phi'[\mathrm{v_j}(n)]\sum \delta_\mathrm{k}(n)\mathrm{w_{kj}}(n)\text{...........................}(7)$$

All the local gradients are then used to calculate Δwji and adapt the synaptic weights so that to reduce the network error, as follows:

$$\Delta\mathrm{w_{ji}}(n) = -\eta\partial E(n)/\partial\mathrm{w_{ji}} = \eta\delta_\mathrm{j}(n)\mathrm{y_i}(n)\text{........................}(8)$$

The code snippet for the backpropagation is given below, to show how the above discussed mathematical computations were dealt with by in python:

```
def backwardPropagation(self, X, y, sigmoid = True):
        # 1. Output Layer

        #  e(n) = d(n) - y(n)
        self.error_a2 = self.a2 - y

        if sigmoid:
            # delta = e * sigmoid_derivative(a2)
            self.delta_a2 = self.error_a2 * self.sigmoid(self.a2, dev= True)
        else:
            # delta = e * tanh_derivative(a2)
            self.delta_a2 = self.error_a2 * self.tanh(self.a2, dev= True)


        # 2. Hidden Layer

        # error = summation(delta*weight)
        self.error_a1 = np.dot(self.delta_a2,self.weights2.T)
        if sigmoid:
            # delta = sigmoid_derivative(a1) * summation(delta*weight)
            self.delta_a1 = self.sigmoid(self.a1, dev = True) * self.error_a1
        else:
            # delta = tanh_derivative(a1) * summation(delta*weight)
            self.delta_a1 = self.tanh(self.a1, dev = True) * self.error_a1

        #Updating Weights

        # New weight = old weight - learning rate * summation(previous layer input * present layer delta)

        self.weights2 -= self.learningRate * np.dot(self.a1.T,self.delta_a2)
        self.weights1 -= self.learningRate * np.dot(X.T,self.delta_a1)

        self.bias2 -=  np.sum(self.learningRate * self.delta_a2)

        self.bias1 -= np.sum(self.learningRate * self.delta_a1)
        return np.sum(((self.error_a2)**2) / 2.0)
```

For further details the separately submitted **"mlp.py"** file can be referred.

2. Results and Discussion

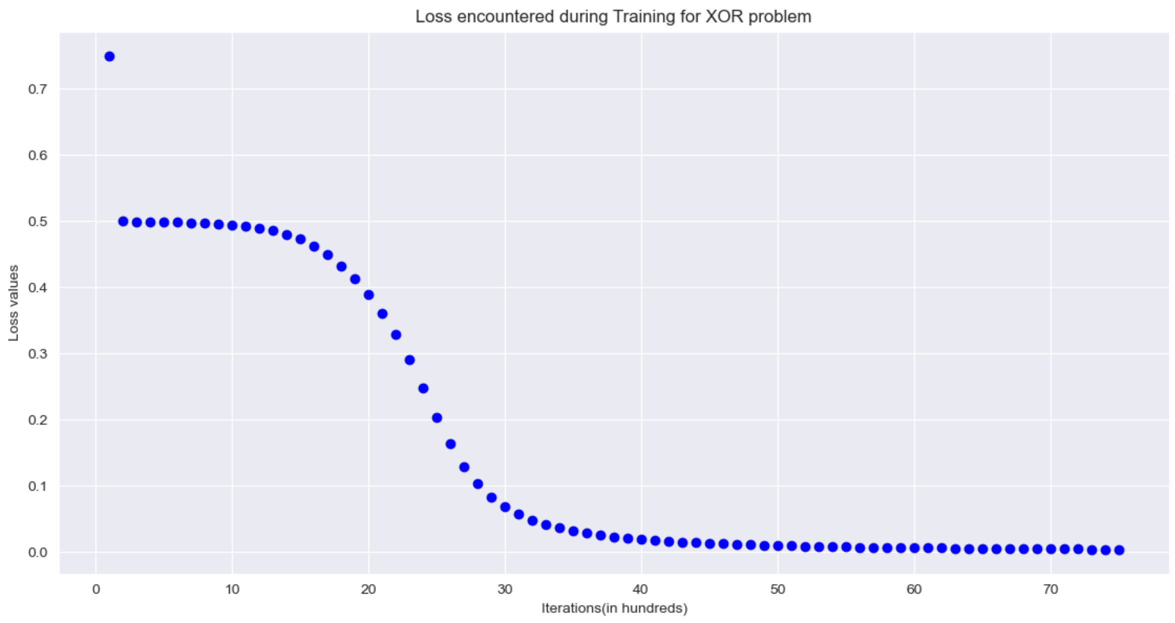
The designed MLP was experimented in classifying two different famous and classic ANN problems - the **XOR** (**exclusive or**) and the **XNOR** (**not exclusive or**). That way, the network was assessed whether it can predict the outputs of XOR and XNOR logic gates correctly or not.

The experiment was done using numpy random seed 16 and the network was trained for 7500 iterations. It was performed under four different settings or architectures for both the XOR and XNOR data classification problem :

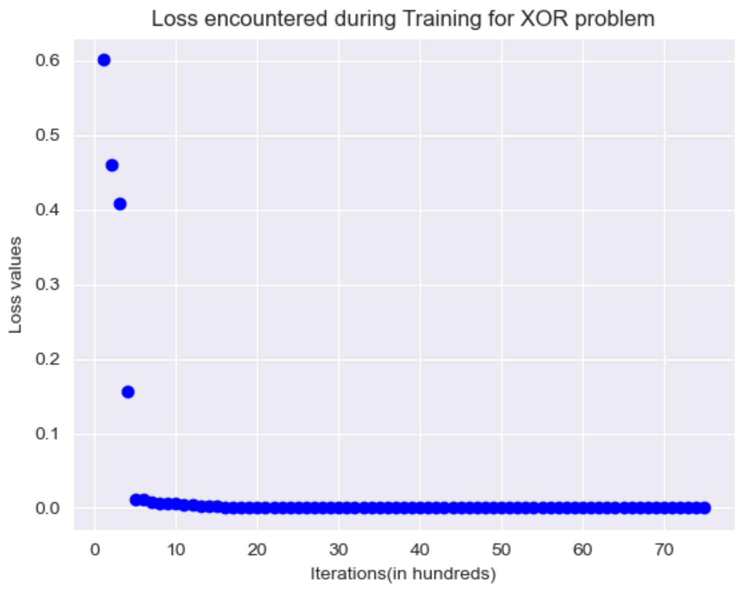
- Using **randomly initialised** weights and **Tanh** non linearity
- Using **"Xavier's" weight initialisation** technique and **Tanh** non linearity
- Using **randomly initialised** weights and **Sigmoid** non linearity
- Using **"Xavier's"** weight initialisation technique and **Sigmoid** non linearity

I. XOR Results

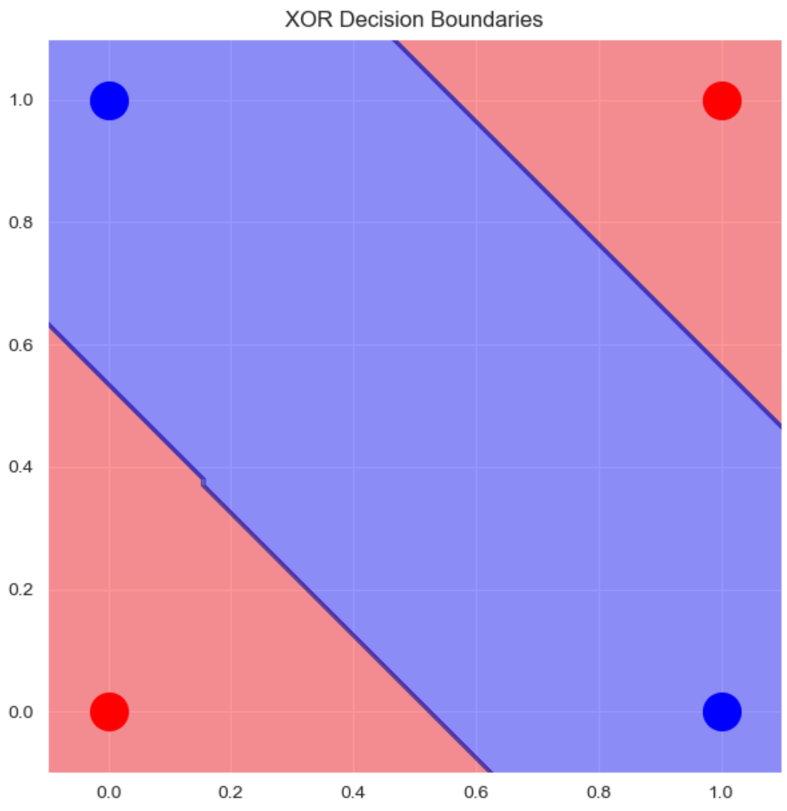
For the XOR problem, as can be seen from the graph below, the network seems to learn it's trade and produce significantly low loss values and more and more accurate predicted output starting from around the 3000th iteration on wards (eventually yielding a loss of only 0.00429 starting from about 0.75 in the first iteration). That was particularly observed in the case where the sigmoid activation was used and weights were randomly initialised. In the same non-linearity case of sigmoid but when applying the Xavier's weight initialising method, a very similar loss was observed, although the the network seemed to learn a bit slower in the latter's case, i.e., after 4000th iteration (eventually yielding a loss of only 0.00505).

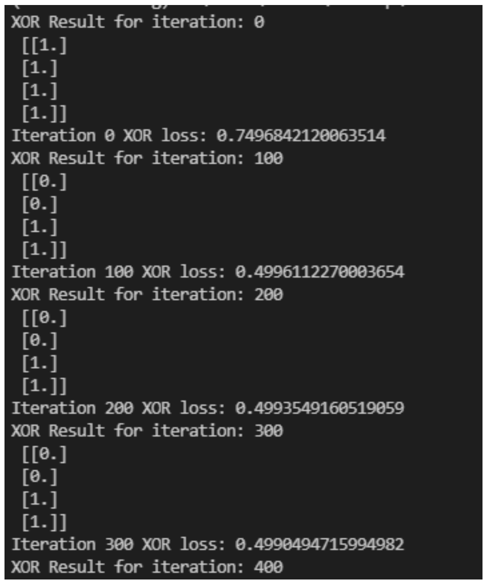


In the case of Tanh non linearity, however, the network seems to suffer from premature learning or underfitting and thus resulting in loss curve that remains flat regardless of training. That was true irrespective of which weight initialising method was used. The loss curve observed using Tanh non linearity is given below:

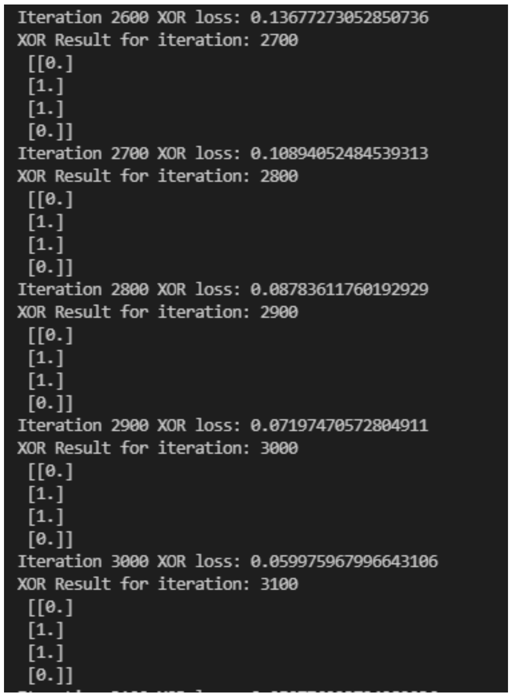


Finally, a XOR decision boundary was plotted for the sigmoid non linearity and randomly initialised weight case, as can be seen below. It correctly classified the present patterns as indicated by the blue region, for XOR with output = 1, and red region, for XOR with output = 0. It, however, failed to produce accurate decision boundaries in the case where Tanh activation was used, for reasons stated earlier, i.e., underfitting.

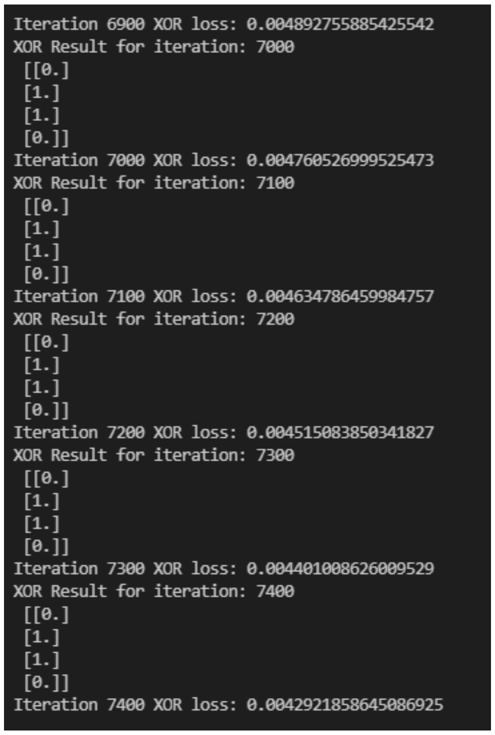




```
XOR Result for iteration: 0
 [[1.]
 [1.]
 [1.]
 [1.]]
Iteration 0 XOR loss: 0.7496842120063514
XOR Result for iteration: 100
 [[0.]
 [0.]
 [1.]
 [1.]]
Iteration 100 XOR loss: 0.4996112270003654
XOR Result for iteration: 200
 [[0.]
 [0.]
 [1.]
 [1.]]
Iteration 200 XOR loss: 0.4993549160519059
XOR Result for iteration: 300
 [[0.]
 [0.]
 [1.]
 [1.]]
Iteration 300 XOR loss: 0.4990494715994982
XOR Result for iteration: 400
```



```
Iteration 2600 XOR loss: 0.13677273052850736
XOR Result for iteration: 2700
 [[0.]
 [1.]
 [1.]
 [0.]]
Iteration 2700 XOR loss: 0.10894052484539313
XOR Result for iteration: 2800
 [[0.]
 [1.]
 [1.]
 [0.]]
Iteration 2800 XOR loss: 0.08783611760192929
XOR Result for iteration: 2900
 [[0.]
 [1.]
 [1.]
 [0.]]
Iteration 2900 XOR loss: 0.07197470572804911
XOR Result for iteration: 3000
 [[0.]
 [1.]
 [1.]
 [0.]]
Iteration 3000 XOR loss: 0.059975967996643106
XOR Result for iteration: 3100
 [[0.]
 [1.]
 [1.]
 [0.]]
```

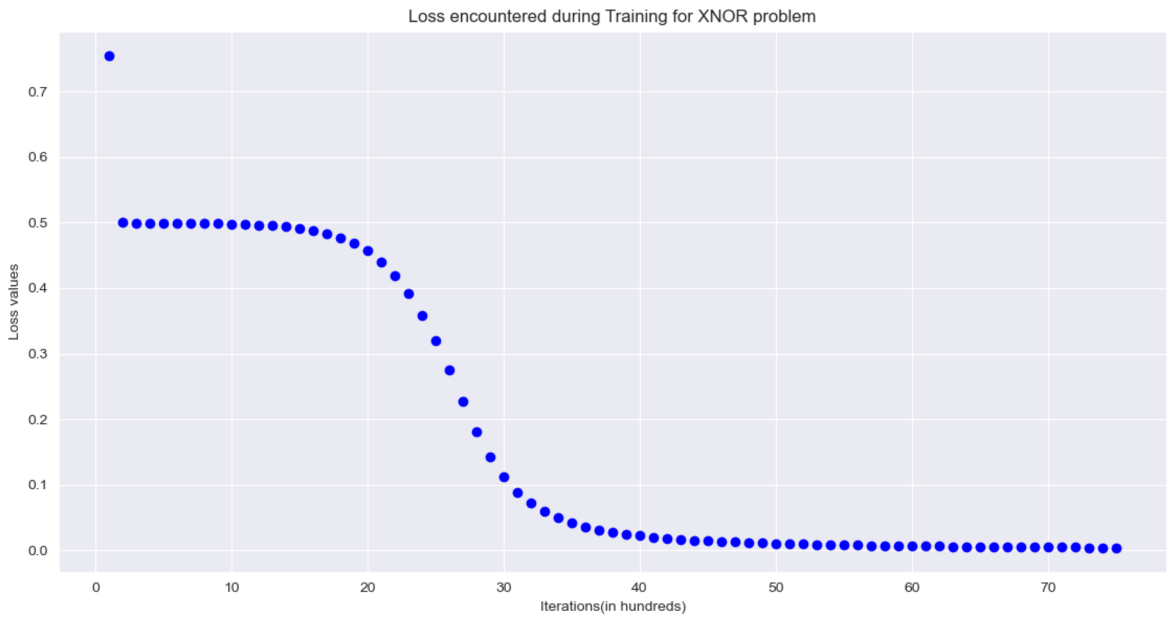


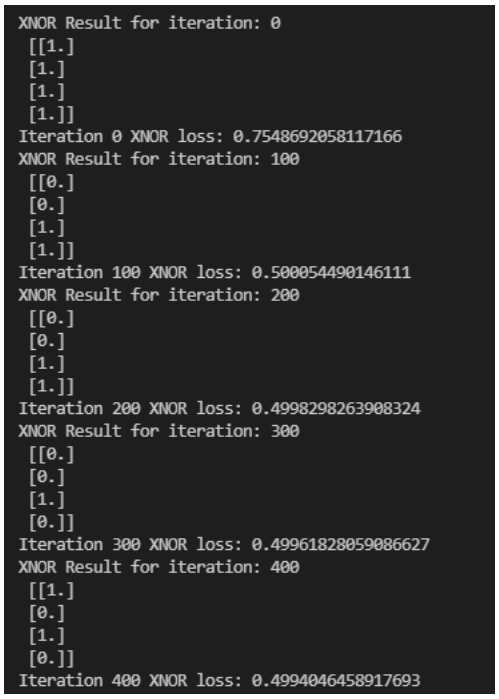
```
Iteration 6900 XOR loss: 0.004892755885425542
XOR Result for iteration: 7000
 [[0.]
 [1.]
 [1.]
 [0.]]
Iteration 7000 XOR loss: 0.004760526999525473
XOR Result for iteration: 7100
 [[0.]
 [1.]
 [1.]
 [0.]]
Iteration 7100 XOR loss: 0.004634786459984757
XOR Result for iteration: 7200
 [[0.]
 [1.]
 [1.]
 [0.]]
Iteration 7200 XOR loss: 0.004515083850341827
XOR Result for iteration: 7300
 [[0.]
 [1.]
 [1.]
 [0.]]
Iteration 7300 XOR loss: 0.004401008626009529
XOR Result for iteration: 7400
 [[0.]
 [1.]
 [1.]
 [0.]]
Iteration 7400 XOR loss: 0.0042921858645086925
```

II. **XNOR Results**

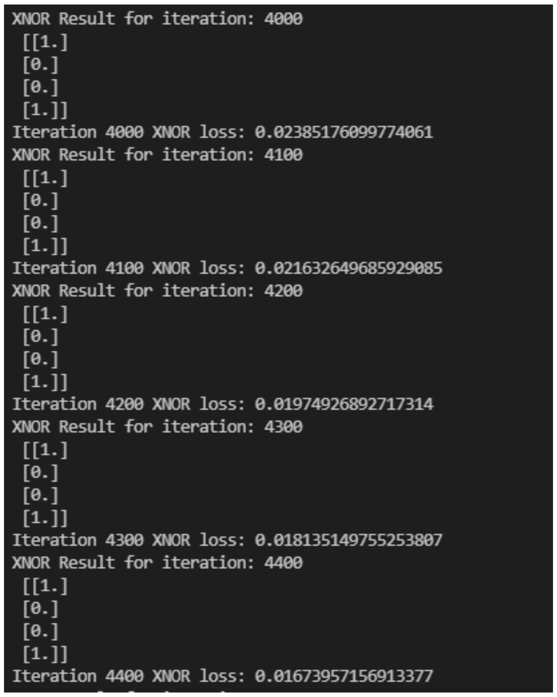
XNOR problem is the exact opposite of XOR, and so if the input was a list given by [[0,0],[0,1],[1,0],[1,1]], then the network is expected to produce and output of [[1],[0],[0],[1]].

It can be shown in the graph below that, when applying a sigmoid non linearity and random weight initialisation, the network seems to perform well. The loss, starting at a value of 0.76 (at iteration 1), continues to decrease and eventually reaches 0.0045. It generally started producing very low loss values and accurate outputs starting from around 4000th iteration. The loss curve and a sample screenshot of the out put produced or printed by the code is given below.

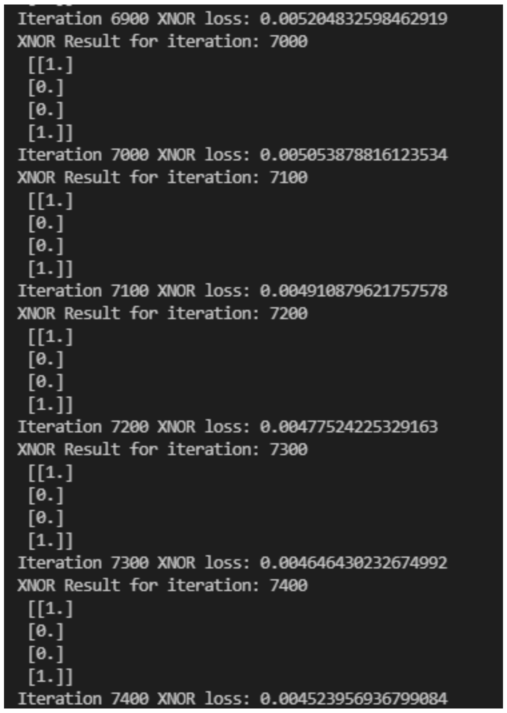




```
XNOR Result for iteration: 0
 [[1.]
 [1.]
 [1.]
 [1.]]
Iteration 0 XNOR loss: 0.7548692058117166
XNOR Result for iteration: 100
 [[0.]
 [0.]
 [1.]
 [1.]]
Iteration 100 XNOR loss: 0.500054490146111
XNOR Result for iteration: 200
 [[0.]
 [0.]
 [1.]
 [1.]]
Iteration 200 XNOR loss: 0.4998298263908324
XNOR Result for iteration: 300
 [[0.]
 [0.]
 [1.]
 [0.]]
Iteration 300 XNOR loss: 0.49961828059086627
XNOR Result for iteration: 400
 [[1.]
 [0.]
 [1.]
 [0.]]
Iteration 400 XNOR loss: 0.4994046458917693
```

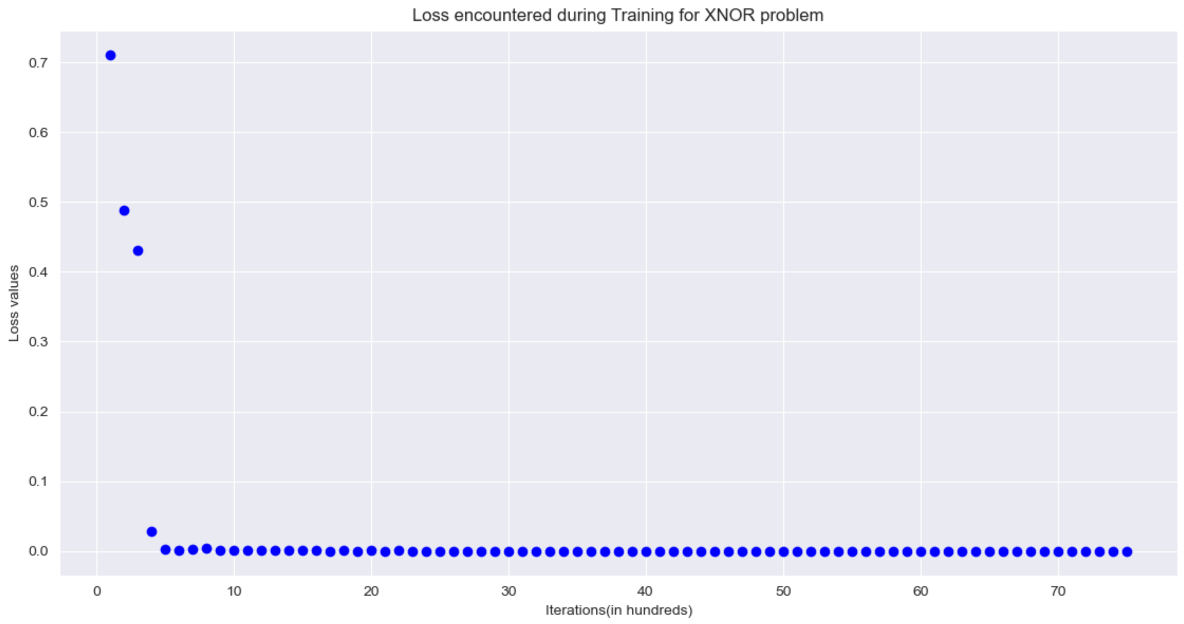


```
XNOR Result for iteration: 4000
 [[1.]
 [0.]
 [0.]
 [1.]]
Iteration 4000 XNOR loss: 0.02385176099774061
XNOR Result for iteration: 4100
 [[1.]
 [0.]
 [0.]
 [1.]]
Iteration 4100 XNOR loss: 0.021632649685929085
XNOR Result for iteration: 4200
 [[1.]
 [0.]
 [0.]
 [1.]]
Iteration 4200 XNOR loss: 0.01974926892717314
XNOR Result for iteration: 4300
 [[1.]
 [0.]
 [0.]
 [1.]]
Iteration 4300 XNOR loss: 0.018135149755253807
XNOR Result for iteration: 4400
 [[1.]
 [0.]
 [0.]
 [1.]]
Iteration 4400 XNOR loss: 0.01673957156913377
```

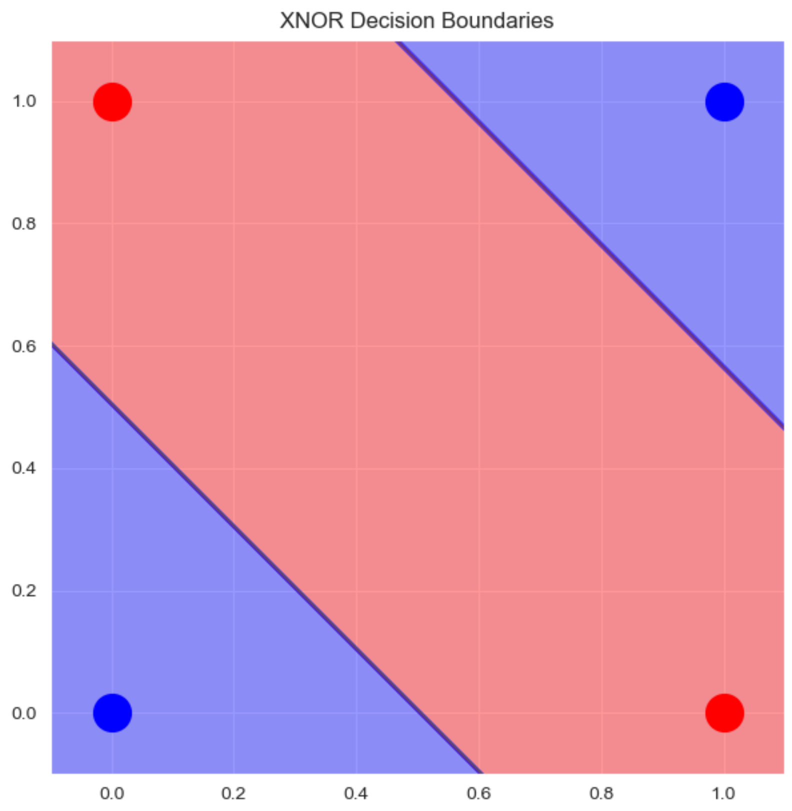


```
Iteration 6900 XNOR loss: 0.005204832598462919
XNOR Result for iteration: 7000
 [[1.]
 [0.]
 [0.]
 [1.]]
Iteration 7000 XNOR loss: 0.005053878816123534
XNOR Result for iteration: 7100
 [[1.]
 [0.]
 [0.]
 [1.]]
Iteration 7100 XNOR loss: 0.004910879621757578
XNOR Result for iteration: 7200
 [[1.]
 [0.]
 [0.]
 [1.]]
Iteration 7200 XNOR loss: 0.00477524225329163
XNOR Result for iteration: 7300
 [[1.]
 [0.]
 [0.]
 [1.]]
Iteration 7300 XNOR loss: 0.004646430232674992
XNOR Result for iteration: 7400
 [[1.]
 [0.]
 [0.]
 [1.]]
Iteration 7400 XNOR loss: 0.004523956936799084
```

When applying the same non-linearity function of sigmoid to the weights initialised using Xavier's Method, however, a constant loss curve was observed (irrespective of the training). In deed that was also the case whenever the other non-linearity function Tanh was used, both with randomly initialised weights and weights initialised using Xavier's method. The graph below illustrates that:



Lastly, similar to the XOR problem's case, a decision boundary was plotted for the sigmoid non linearity and randomly initialised weight case (the best observed setting of the architecture experimentation). The decision boundary successfully classifies the 4 input data points in the right regions.



Part-3

1. Introduction and Design

In this part, **Genetic Algorithm (GA)** and **Particle Swarm Optimisation (PSO)** methods were used to train the multi layer neural network of the XOR and XNOR problem, discussed in the previous part. In order to do that, **DEAP** was used for the GA part and **pyswarms** for PSO, both using python code.

For both implementations of GA and pyswarms, the usual forward pass was first implemented, followed by the respective weight optimisation. To take care of the non linearity, sigmoid function was used in the forward pass. In both cases, the network had 17 weights, in total, to optimise. Throughout the experimentations, random seed 50 was used as the default seed.

A code snippet from the GA forward pass looks as follows:

```
def sigmoid(self, z):
        return 1 / (1 + np.exp((-1) * z))

    def hidden(self, IP1, IP2, wt1, wt2, wt0):
        inducedfield = IP1*wt1 + IP2*wt2 + self.bias*wt0
        return self.sigmoid(inducedfield)

    def finaloutput(self, IP1, IP2, IP3, IP4, wt1, wt2, wt3, wt4, wt0):
        inducedfield = IP1*wt1 + IP2*wt2 + IP3*wt3 + IP4*wt4 + self.bias*wt0
        return self.sigmoid(inducedfield)
```

Similarly, the PSO forward pass code snippet looks as follows:

```
def forwardPass(self, p):
        # Roll-back the weights and biases
        W1 = p[0:8].reshape((self.n_inputs,self.n_hidden))
        b1 = p[8:12].reshape((self.n_hidden,))
        W2 = p[12:16].reshape((self.n_hidden,self.n_classes))
        b2 = p[16:17].reshape((self.n_classes,))

        # Pre-activation in Layer 1
        v1 = self.X.dot(W1) + b1

        # Activation in Layer 1
        a1 = self.sigmoid(v1)

        # Pre-activation in Layer 2
        v2 = a1.dot(W2) + b2

        # Activation in Layer 2
        a2 = self.sigmoid(v2)

        return a2


    def sigmoid(self, Z):
        return 1 / (1 + np.exp((-1) * Z))
```

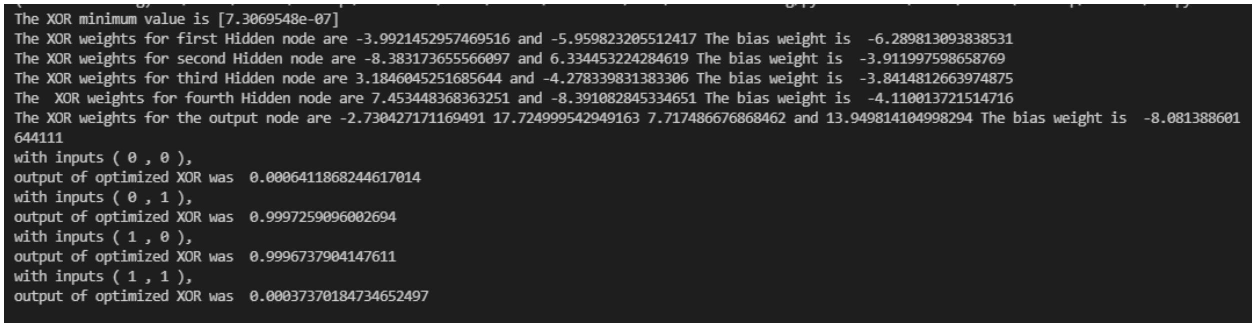
For more details on how the code is structured, the separately attached python file of **"evolutionaryOptimisation.py"** can be referred.

2. Results and Discussions

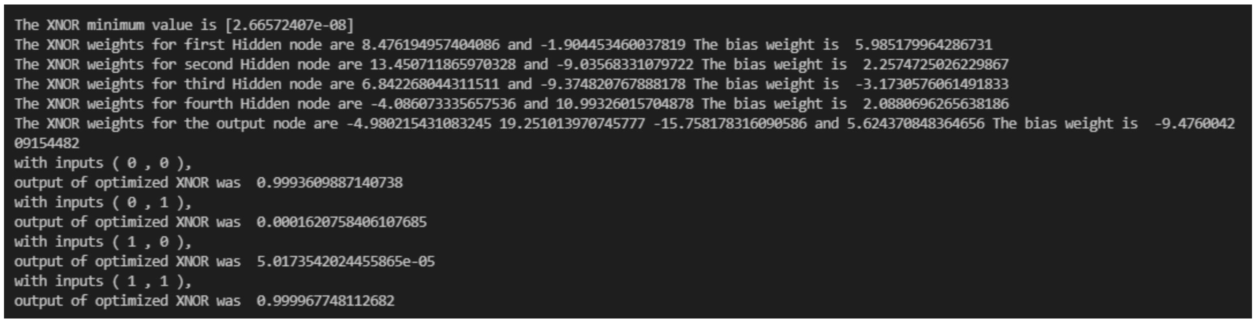
I. GA

As stated earlier the GA optimisation of the MLP was done using the DEAP library. The network was tested under different varieties of mutation and cross over rates, to see which one performs the best.

Experimentation was started by considering an initial population of 500 individuals and 80 generations . However, to reduce the chance of getting a good solution from the random first generation, and to make sure to show off the evolutionary process, the number of population was later reduced to 300. For the XOR problem a crossover rate of 0.5 and mutation rate of 0.15 produced the best result, whereas for the XNOR problem a cross over rate of 0.6 seems to work out to be the best. Below is a screenshot of the output displayed in the python terminal, when the experiment was done using those parameters:



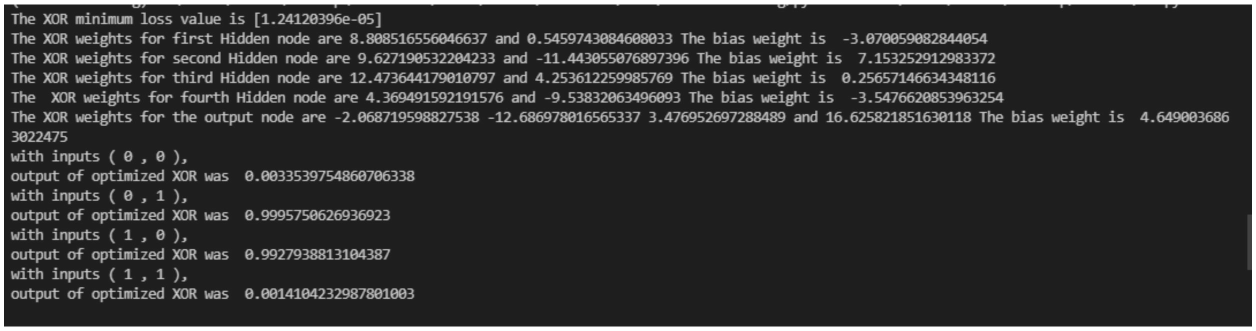
```
The XOR minimum value is [7.3069548e-07]
The XOR weights for first Hidden node are -3.9921452957469516 and -5.959823205512417 The bias weight is  -6.289813093838531
The XOR weights for second Hidden node are -8.383173655566097 and 6.334453224284619 The bias weight is  -3.911997598658769
The XOR weights for third Hidden node are 3.1846045251685644 and -4.278339831383306 The bias weight is  -3.8414812663974875
The  XOR weights for fourth Hidden node are 7.453448368363251 and -8.391082845334651 The bias weight is  -4.110013721514716
The XOR weights for the output node are -2.730427171169491 17.724999542949163 7.717486676868462 and 13.949814104998294 The bias weight is  -8.081388601
644111
with inputs ( 0 , 0 ),
output of optimized XOR was  0.0006411868244617014
with inputs ( 0 , 1 ),
output of optimized XOR was  0.9997259096002694
with inputs ( 1 , 0 ),
output of optimized XOR was  0.9996737904147611
with inputs ( 1 , 1 ),
output of optimized XOR was  0.00037370184734652497
```



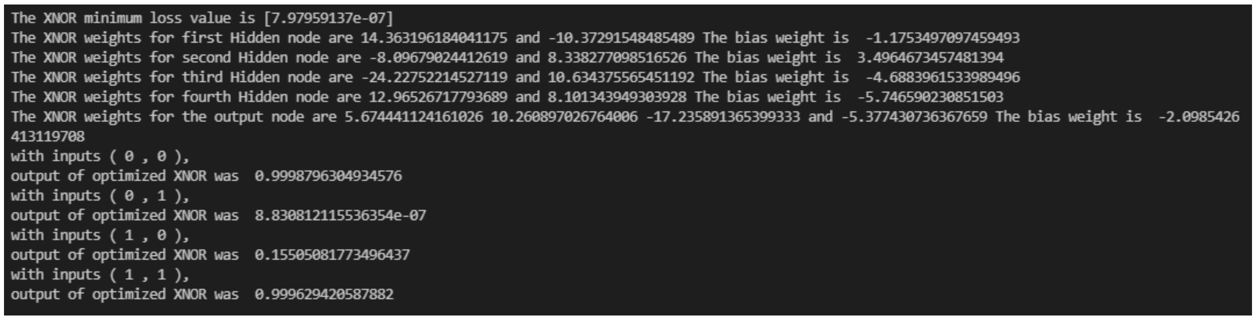
```
The XNOR minimum value is [2.66572407e-08]
The XNOR weights for first Hidden node are 8.476194957404086 and -1.904453460037819 The bias weight is  5.985179964286731
The XNOR weights for second Hidden node are 13.450711865970328 and -9.03568331079722 The bias weight is  2.2574725026229867
The XNOR weights for third Hidden node are 6.842268044311511 and -9.374820767888178 The bias weight is  -3.1730576061491833
The XNOR weights for fourth Hidden node are -4.086073335657536 and 10.99326015704878 The bias weight is  2.0880696265638186
The XNOR weights for the output node are -4.980215431083245 19.251013970745777 -15.758178316090586 and 5.624370848364656 The bias weight is  -9.4760042
09154482
with inputs ( 0 , 0 ),
output of optimized XNOR was  0.9993609887140738
with inputs ( 0 , 1 ),
output of optimized XNOR was  0.0001620758406107685
with inputs ( 1 , 0 ),
output of optimized XNOR was  5.0173542024455865e-05
with inputs ( 1 , 1 ),
output of optimized XNOR was  0.999967748112682
```

As can be seen above, these hyperparameters produced almost perfect prediction of output in both classification problems, as the observed loss of $7.3X10^-7$ and $2.66X10^-8$, respectively for XOR and XNOR.

Altering the initial population size down to 200 seemed to produce a bit more loss (error) and especially in the XNOR problem case, the third digit of the predicted output was around 0.15 higher than the desired output of 0. The following screenshot shows that:



```
The XOR minimum loss value is [1.24120396e-05]
The XOR weights for first Hidden node are 8.808516556046637 and 0.5459743084608033 The bias weight is  -3.070059082844054
The XOR weights for second Hidden node are 9.627190532204233 and -11.443055076897396 The bias weight is  7.153252912983372
The XOR weights for third Hidden node are 12.473644179010797 and 4.253612259985769 The bias weight is  0.25657146634348116
The  XOR weights for fourth Hidden node are 4.369491592191576 and -9.53832063496093 The bias weight is  -3.5476620853963254
The XOR weights for the output node are -2.068719598827538 -12.686978016565337 3.476952697288489 and 16.625821851630118 The bias weight is  4.649003686
3022475
with inputs ( 0 , 0 ),
output of optimized XOR was  0.0033539754860706338
with inputs ( 0 , 1 ),
output of optimized XOR was  0.9995750626936923
with inputs ( 1 , 0 ),
output of optimized XOR was  0.9927938813104387
with inputs ( 1 , 1 ),
output of optimized XOR was  0.0014104232987801003
```

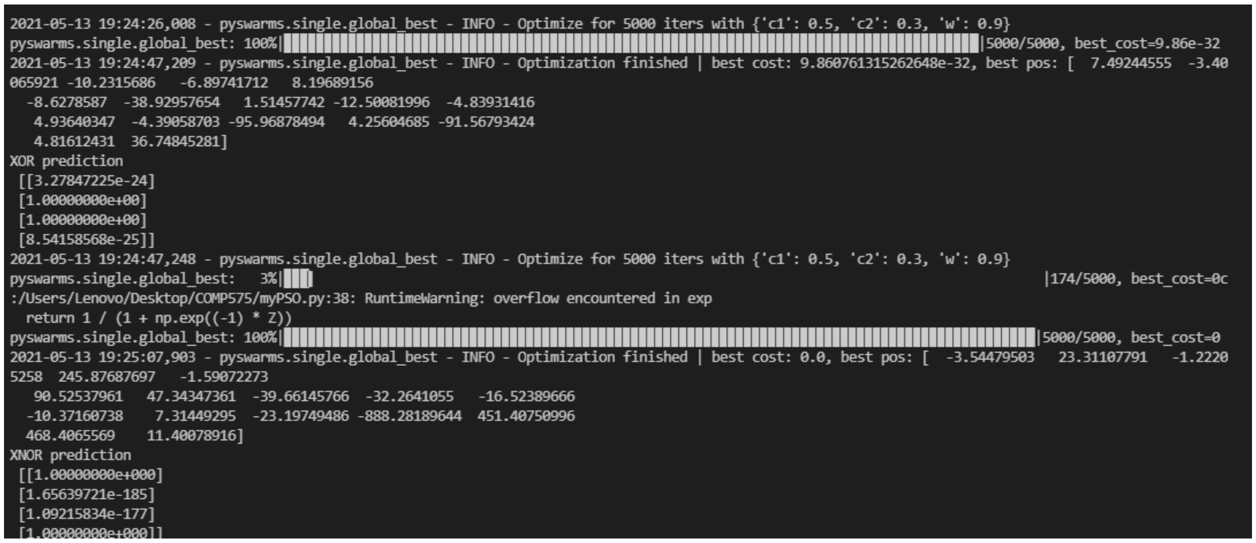


```
The XNOR minimum loss value is [7.97959137e-07]
The XNOR weights for first Hidden node are 14.363196184041175 and -10.37291548485489 The bias weight is  -1.1753497097459493
The XNOR weights for second Hidden node are -8.09679024412619 and 8.338277098516526 The bias weight is  3.4964673457481394
The XNOR weights for third Hidden node are -24.22752214527119 and 10.634375565451192 The bias weight is  -4.6883961533989496
The XNOR weights for fourth Hidden node are 12.96526717793689 and 8.101343949303928 The bias weight is  -5.746590230851503
The XNOR weights for the output node are 5.674441124161026 10.260897026764006 -17.235891365399333 and -5.377430736367659 The bias weight is  -2.0985426
413119708
with inputs ( 0 , 0 ),
output of optimized XNOR was  0.9998796304934576
with inputs ( 0 , 1 ),
output of optimized XNOR was  8.830812115536354e-07
with inputs ( 1 , 0 ),
output of optimized XNOR was  0.15505081773496437
with inputs ( 1 , 1 ),
output of optimized XNOR was  0.999629420587882
```

II. PSO

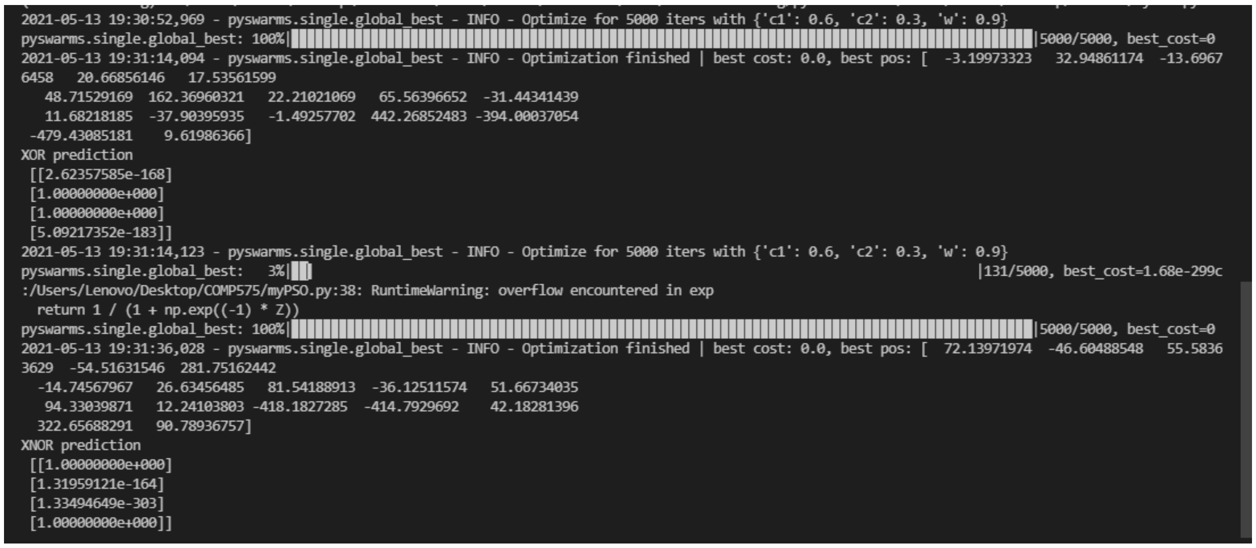
PSO optimisation was implemented by using pyswarms library. Experiments were done for different number of particles, cognitive and social parameters, and inertia parameter.

Initially, 80 particles and a cognitive parameter of 0.5, a social parameter of 0.3 and an inertia of 0.9 were used. The network did an excellent job in predicting the correct outputs for both the XOR and XNOR problems. A screenshot of the results is as follows:



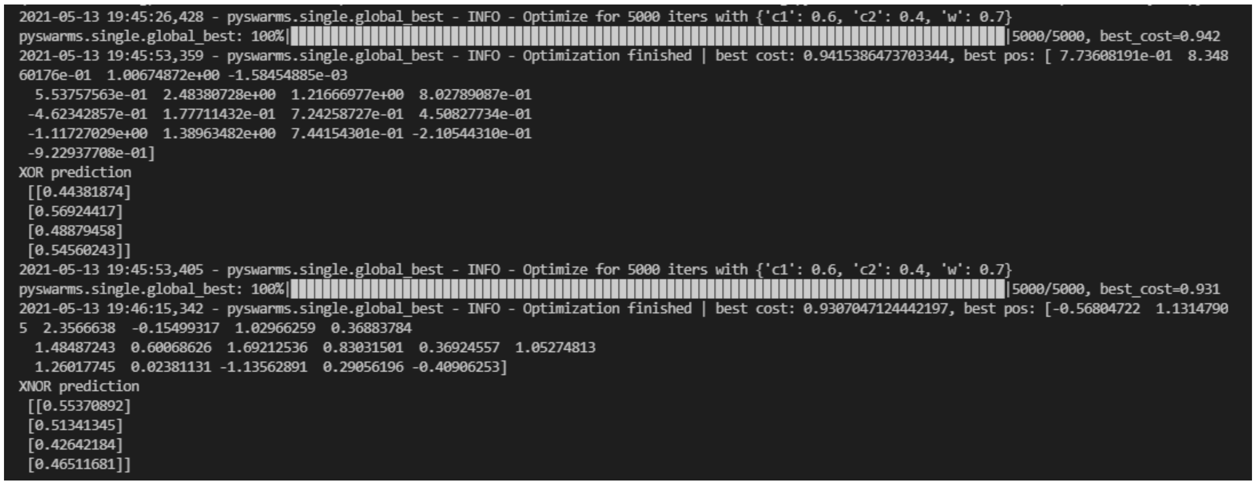
```
2021-05-13 19:24:26,008 - pyswarms.single.global_best - INFO - Optimize for 5000 iters with {'c1': 0.5, 'c2': 0.3, 'w': 0.9}
pyswarms.single.global_best: 100%|████████████████████████████████████████████████████████████████████████|5000/5000, best_cost=9.86e-32
2021-05-13 19:24:47,209 - pyswarms.single.global_best - INFO - Optimization finished | best cost: 9.860761315262648e-32, best pos: [  7.49244555  -3.40
065921 -10.2315686   -6.89741712   8.19689156
  -8.6278587  -38.92957654   1.51457742 -12.50081996  -4.83931416
   4.93640347  -4.39058703 -95.96878494   4.25604685 -91.56793424
   4.81612431  36.74845281]
XOR prediction
 [[3.27847225e-24]
 [1.00000000e+00]
 [1.00000000e+00]
 [8.54158568e-25]]
2021-05-13 19:24:47,248 - pyswarms.single.global_best - INFO - Optimize for 5000 iters with {'c1': 0.5, 'c2': 0.3, 'w': 0.9}
pyswarms.single.global_best:   3%|██▌                                                                          |174/5000, best_cost=0c
:/Users/Lenovo/Desktop/COMP575/myPSO.py:38: RuntimeWarning: overflow encountered in exp
  return 1 / (1 + np.exp((-1) * Z))
pyswarms.single.global_best: 100%|█████████████████████████████████████████████████████████████████████████████|5000/5000, best_cost=0
2021-05-13 19:25:07,903 - pyswarms.single.global_best - INFO - Optimization finished | best cost: 0.0, best pos: [  -3.54479503   23.31107791   -1.2220
5258  245.87687697   -1.59072273
   90.52537961   47.34347361  -39.66145766  -32.2641055   -16.52389666
  -10.37160738    7.31449295  -23.19749486 -888.28189644  451.40750996
  468.4065569    11.40078916]
XNOR prediction
 [[1.00000000e+000]
 [1.65639721e-185]
 [1.09215834e-177]
 [1.00000000e+000]]
```

Later, a little tweak was made by changing the cognitive parameter to 0.6, whilst keeping the other parameters constant. As a result of that, a much better result was obtained in the case of XOR problem and a little improvement in the case of XNOR. The following screenshot depicts that:



```
2021-05-13 19:30:52,969 - pyswarms.single.global_best - INFO - Optimize for 5000 iters with {'c1': 0.6, 'c2': 0.3, 'w': 0.9}
pyswarms.single.global_best: 100%|█████████████████████████████████████████████████████████████████████████|5000/5000, best_cost=0
2021-05-13 19:31:14,094 - pyswarms.single.global_best - INFO - Optimization finished | best cost: 0.0, best pos: [  -3.19973323   32.94861174  -13.6967
6458   20.66856146   17.53561599
   48.71529169  162.36960321   22.21021069   65.56396652  -31.44341439
   11.68218185  -37.90395935   -1.49257702  442.26852483 -394.00037054
 -479.43085181    9.61986366]
XOR prediction
 [[2.62357585e-168]
 [1.00000000e+000]
 [1.00000000e+000]
 [5.09217352e-183]]
2021-05-13 19:31:14,123 - pyswarms.single.global_best - INFO - Optimize for 5000 iters with {'c1': 0.6, 'c2': 0.3, 'w': 0.9}
pyswarms.single.global_best:   3%|██                                                                    |131/5000, best_cost=1.68e-299c
:/Users/Lenovo/Desktop/COMP575/myPSO.py:38: RuntimeWarning: overflow encountered in exp
  return 1 / (1 + np.exp((-1) * Z))
pyswarms.single.global_best: 100%|█████████████████████████████████████████████████████████████████████████|5000/5000, best_cost=0
2021-05-13 19:31:36,028 - pyswarms.single.global_best - INFO - Optimization finished | best cost: 0.0, best pos: [  72.13971974  -46.60488548   55.5836
3629  -54.51631546  281.75162442
  -14.74567967   26.63456485   81.54188913  -36.12511574   51.66734035
   94.33039871   12.24103803 -418.1827285  -414.7929692    42.18281396
  322.65688291   90.78936757]
XNOR prediction
 [[1.00000000e+000]
 [1.31959121e-164]
 [1.33494649e-303]
 [1.00000000e+000]]
```

Lastly, the number of particles in the swarm was increased to 100 and the inertia was decreased to 0.7, to assess what impact it might have on the results. As can be seen below, it produced the worst performance, in terms of the loss (error). The predicted numbers were off the desired ones by almost ∓ 0.5 in each of 4 numbers in the XOR and XNOR problems.



```
2021-05-13 19:45:26,428 - pyswarms.single.global_best - INFO - Optimize for 5000 iters with {'c1': 0.6, 'c2': 0.4, 'w': 0.7}
pyswarms.single.global_best: 100%|████████████████████████████████████████████████████████████████████████████████|5000/5000, best_cost=0.942
2021-05-13 19:45:53,359 - pyswarms.single.global_best - INFO - Optimization finished | best cost: 0.9415386473703344, best pos: [ 7.73608191e-01  8.348
60176e-01  1.00674872e+00 -1.58454885e-03
  5.53757563e-01  2.48380728e+00  1.21666977e+00  8.02789087e-01
 -4.62342857e-01  1.77711432e-01  7.24258727e-01  4.50827734e-01
 -1.11727029e+00  1.38963482e+00  7.44154301e-01 -2.10544310e-01
 -9.22937708e-01]
XOR prediction
 [[0.44381874]
 [0.56924417]
 [0.48879458]
 [0.54560243]]
2021-05-13 19:45:53,405 - pyswarms.single.global_best - INFO - Optimize for 5000 iters with {'c1': 0.6, 'c2': 0.4, 'w': 0.7}
pyswarms.single.global_best: 100%|████████████████████████████████████████████████████████████████████████████████|5000/5000, best_cost=0.931
2021-05-13 19:46:15,342 - pyswarms.single.global_best - INFO - Optimization finished | best cost: 0.9307047124442197, best pos: [-0.56804722  1.1314790
5  2.3566638  -0.15499317  1.02966259  0.36883784
  1.48487243  0.60068626  1.69212536  0.83031501  0.36924557  1.05274813
  1.26017745  0.02381131 -1.13562891  0.29056196 -0.40906253]
XNOR prediction
 [[0.55370892]
 [0.51341345]
 [0.42642184]
 [0.46511681]]
```

In conclusion, it is worth mentioning that overall, comparing the best performances of the PSO against those of GA, by far the PSO seems to be the best, for both the XOR and XNOR problems. Not only did it produce almost 0 loss in it's best hyperparameter combinations, but also as can be seen from the screenshots, in cases where the output was 1 it constantly and accurately predicted that (as opposed to a prediction of like 0.9997 in the case of GA). And even in cases where the output was 0, it was predicting it almost as $1.85X10^-170$ and at times even much better as $1.33X10^-303$.

Moreover, if it's needed to re-do the experiments, just go to the separately submitted python files for parts 1,2, and 3 and hit run on each one of them and that should produce a similar output to what has been discussed here (the files already contain specific test case, towards the end of the code, that is ready to be run) .

References

1. Ayberk Kanberoglu,(2016), GitHub repository, https://github.com/kanayb/XOR-with-neural-networks-and-genetic-algorithms/blob/master/NeuralNetworkXor.py
2. Deap Documentation Example,(2021),Particle Swarm Optimization Basics. Retrieved from https://deap.readthedocs.io/en/master/examples/pso_basic.html#id1https://deap.readthedocs.io/en/master/examples/ps
3. Lester James V.Miranda. (2017),Training a Neural Network. Retrieved from https://pyswarms.readthedocs.io/en/latest/examples/usecases/train_neural_network.html#Neural-network-architecture
4. Python Engineer (2021). *Machine Learning From Scratch In Python*. Available at: https://www.youtube.com/watch?v=rLOyrWV8gmA (Accessed: 10 May 2021).
5. ‘Least mean squares filter’ (2020). Wikipedia. Available at https://en.wikipedia.org/wiki/Least_mean_squares_filter (Accessed: 08 May 2021).
6. Weimin,(2020), GitHub repository, https://weimin17.github.io/2017/09/Implement-The-Perceptron-Algorithm-in-Python-version2/
7. ‘Multilayer perceptron’ (2021). Wikipedia. Available at https://en.wikipedia.org/wiki/Multilayer_perceptron (Accessed: 10 May 2021).