Coding Assignment 2
Computational Intelligence
Dr. S. Hajipour

Mohammad Hossein Shafizadegan
99104781

December 7, 2023

Contents

1 Question 1 **2**
1.1 Dataset overview 2
1.2 Linear Regression 2
1.3 Logistic Regression 4
1.4 Neural Network 5

2 Question 2 **7**
2.1 MLP with one output neuron 7
2.2 Two output neurons 8

3 Question 3 **10**
3.1 Part 2 11
3.2 Sigmoid activation function 12

1 Question 1

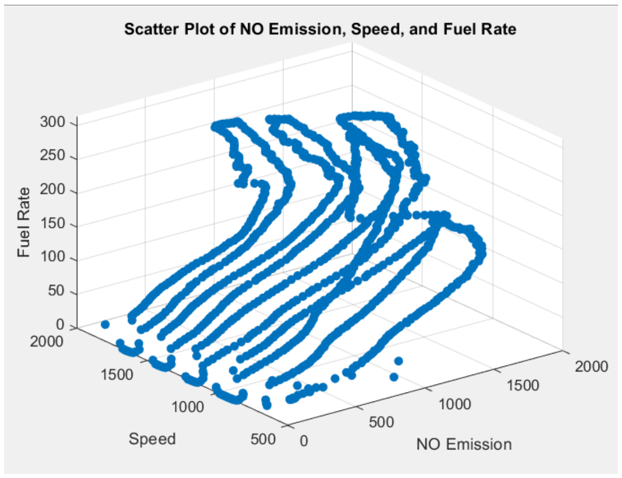
1.1 Dataset overview

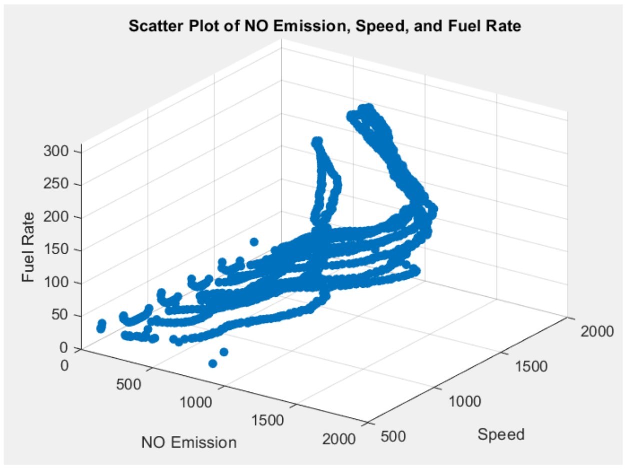
First we load the provided dataset and extract values of NO emission and speed as$X$ and values of fuel rate as Y correspondingly

```
% load data
data = load('../Data/Ex1.mat');
X = [data.NOemission' data.speed'];
Y = data.fuelrate';
```

Using ”scatter3” built-in function, we can visualize our data in the features space.

```
figure;
scatter3(X(:, 1), X(:, 2), Y, 'filled');
xlabel('NO Emission');
ylabel('Speed');
zlabel('Fuel Rate');
title('Scatter Plot of NO Emission, Speed, and Fuel Rate');
```





Now we simply choose the first 700 samples as the training set and the others as the validation set using the following code:

```
trainX = X(1:700, :);
trainY = Y(1:700);
validationX = X(701:end, :);
validationY = Y(701:end);
```

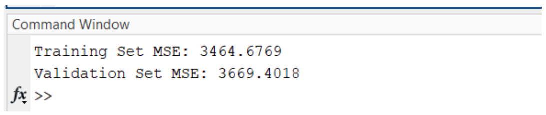
1.2 Linear Regression

In this section we aim to fit a linear regression model to our data. Using the MATLAB built-in function called ”fitlm” we can simply do so.

```
% Fitting a linear regression model using the training set
linear_model = fitlm(trainX, trainY);
```

After that we predict the y values for our validation data and training data separately using our linear model. Finally we calculate the Mean Square Error (MSE) and report the results. Here is the code developed for this section

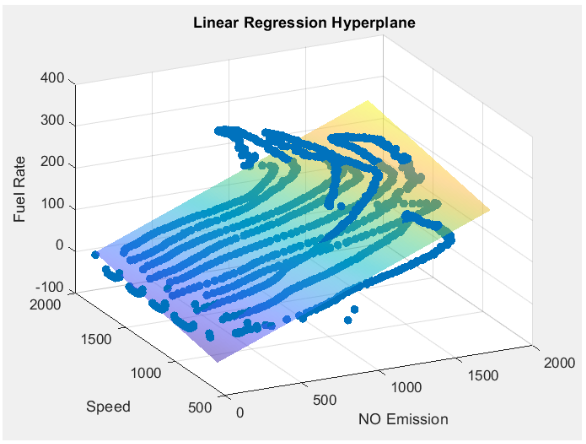
```
% Predicting the fuel rate for the training and validation sets
trainY_pred = predict(linear_model, trainX);
validationY_pred = predict(linear_model, validationX);

% Calculating MSE for the training and validation sets
trainMSE = mean((trainY - trainY_pred).^2);
validationMSE = mean((validationY - validationY_pred).^2);

disp(['Training Set MSE: ' num2str(trainMSE)]);
disp(['Validation Set MSE: ' num2str(validationMSE)]);
```

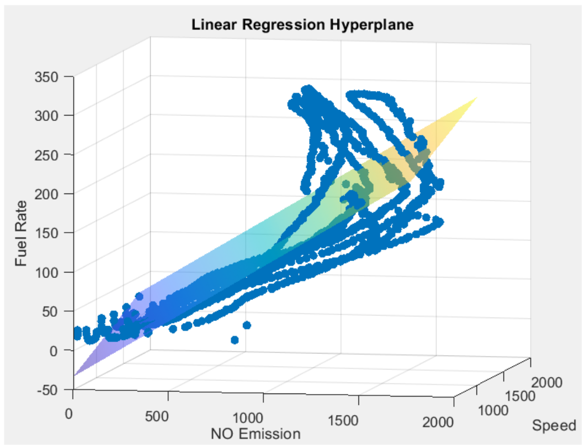

Command Window

```
Training Set MSE: 3464.6769
Validation Set MSE: 3669.4018
fx >>
```

Now in order to check if our results make sense, we will plot the resulting hyperplane. In order to do so, we first extract the coefficients of the plane formula from the trained linear model. The next step is to form and plot the hyperplane regarding the coefficients we found before. The code will be as follows:

```
% Extracting the coefficients and intercept from the linear regression model
coefficients = linear_model.Coefficients.Estimate(2:end);
intercept = linear_model.Coefficients.Estimate(1);

% Creating a meshgrid for the hyperplane visualization
x1 = linspace(min(X(:, 1)), max(X(:, 1)), 100);
x2 = linspace(min(X(:, 2)), max(X(:, 2)), 100);
[X1, X2] = meshgrid(x1, x2);
Y_pred = coefficients(1) * X1 + coefficients(2) * X2 + intercept;

% Scatter plot of the data points
figure;
scatter3(X(:, 1), X(:, 2), Y, 'filled'); hold on;

% Plotting the linear regression hyperplane
surf(X1, X2, Y_pred, 'FaceAlpha', 0.5, 'EdgeColor', 'none');
```





It can be seen and inferred that the results are quite reasonable. Also we note that if we had normalized data and then calculate the MSE error, the values for MSE would be much smaller.

1.3 Logistic Regression

For implementing the logistic regression model, we first have to perform the Logit-Transformation for the y values.

$$z = \ln\left(\frac{Y - y}{y}\right) = ax + b$$

```
    Y = max(1.1*trainY);
    log_trainY = log((Y - trainY)./trainY);
```

Now we create the model just like what we do before:

```
    % Fitting a logistic regression model using the training set
    logistic_model = fitlm(trainX, log_trainY);
```

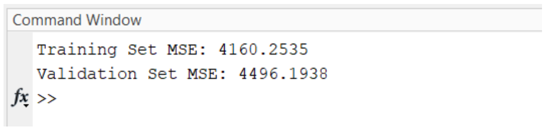
Now we use the model to predict the results for train and validation set. We note that we have to transform the outputs of the model to the previous format by performing the inverse logit-Transform.

$$y = \frac{Y}{1 + e^z}$$

```
    % Predicting the fuel rate probabilities for the training and validation sets
    trainY_pred = predict(logistic_model, trainX);
    trainY_pred = Y ./ (1 + exp(trainY_pred));
    validationY_pred = predict(logistic_model, validationX);
    validationY_pred = Y ./ (1 + exp(validationY_pred));
```

Now we calculate the MSE correspondingly

```
    % Calculating MSE for the training and validation sets
    trainMSE = mean((trainY - trainY_pred).^2);
    validationMSE = mean((validationY - validationY_pred).^2);

    disp(['Training Set MSE: ' num2str(trainMSE)]);
    disp(['Validation Set MSE: ' num2str(validationMSE)]);
```

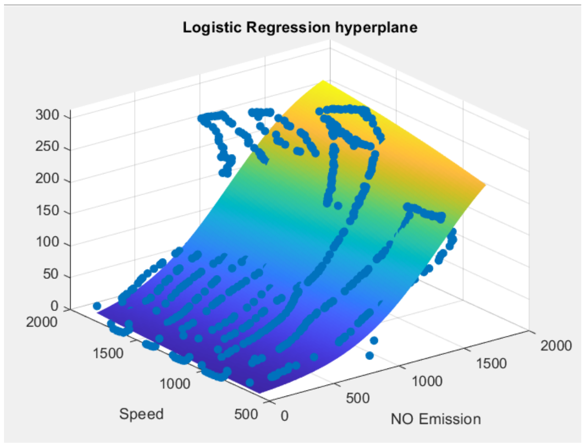


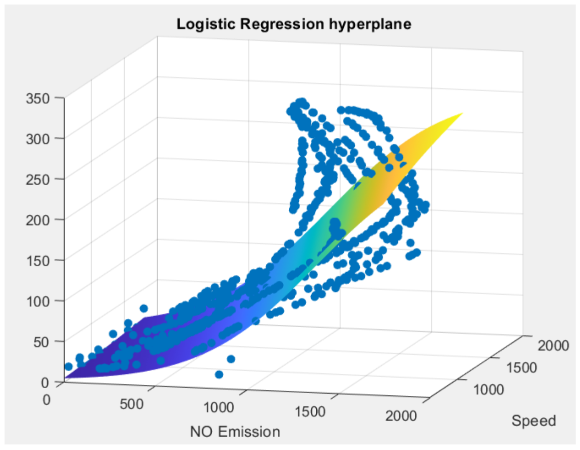
```
Training Set MSE: 4160.2535
Validation Set MSE: 4496.1938
fx >>
```

Like what we did before, in order to see how the model work, we have plotted the resulting hyperplane after extracting the hyperplane parameters and forming the hyperplane.

```
    % Extracting model parameters
    coef = logistic_model.Coefficients.Estimate;
    intercept = coef(1);
    weights = coef(2:end);

    % Calculating the log-odds (linear combination) for the grid points
    log_odds = intercept + X_grid * weights;

    % Applying the logistic (sigmoid) function to obtain probabilities
    Y_grid = max(trainY) ./ (1 + exp(log_odds));

    % Reshaping the predicted probabilities to match the grid dimensions
    Y_grid = reshape(Y_grid, size(X1));

    % Plotting the decision boundary
    surf(X1, X2, Y_grid, 'EdgeColor', 'none');
```

The results can be seen below which demonstrate the performance of the logistic regression vividly





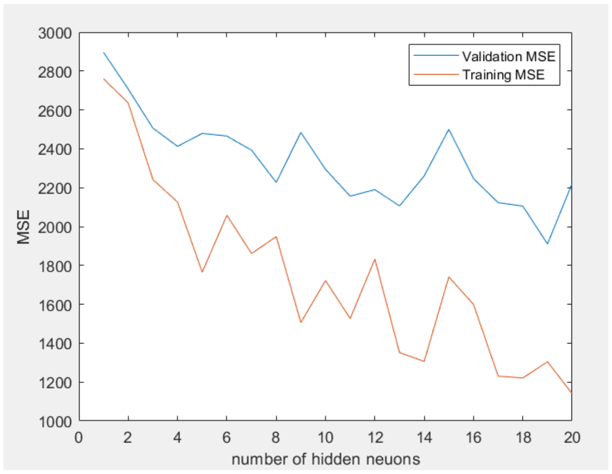
1.4 Neural Network

For this section we have developed a function called ”MLP”. In this function, we create the network using the built-in function ”fitnet” and train it respectively. Then after predicting the y values for the train and validation data, we calculate the MSE error for them and return the resulting values. The code of this function can be seen below:

```
    function [trainMSE, valMSE] = MLP(n_hidden, trainX, validationX, trainY, validationY)

      % Create the MLP model
      net = fitnet(n_hidden);

      % Set the training parameters
      net.trainParam.showWindow = false; % Disable training window display
      net.divideParam.trainRatio = 100/100;
      net.divideParam.testRatio = 0/100;
      net.divideParam.valRatio = 0/100;

      % Train the MLP model
      net = train(net, trainX', trainY');
      % Make predictions on the training and validation sets
      trainPred = net(trainX');
      valPred = net(validationX');

      % Calculate the MSE for the training and validation sets
      trainMSE = mean((trainPred - trainY').^2);
      valMSE = mean((valPred - validationY').^2);

    end
```

Now for different numbers of hidden neurons, we calculate and plot the MSE errors for validation utilizing the previous function we developed.

```
n_hidden = 1:20;
val_MSE_errors = zeros(1,length(n_hidden));
train_MSE_errors = zeros(1,length(n_hidden));

for i=1:length(n_hidden)
   [train_MSE_errors(i), val_MSE_errors(i)] = MLP(n_hidden(i), trainX, validationX, trainY,
       validationY);
end
```



The training and validation error can be seen in the above figure. Regarding both of the above curves, we can infer that perhaps about 13 hidden neurons is the best choice since the validation error has increased after that and the training error is relatively small.

It can be vividly inferred from this question that the MLP network with one hidden layer has a much better performance for these data as the MSE values resulting from MLP are much smaller relatively.

Linear regression is best for predicting continuous variables and is highly interpretable, but it assumes a linear relationship between inputs and outputs, which limits its use in complex, non-linear scenarios.

An MLP with one hidden layer, however, introduces non-linearity through activation functions, allowing it to capture more complex patterns and relationships in the data, making it versatile for both regression and classification.

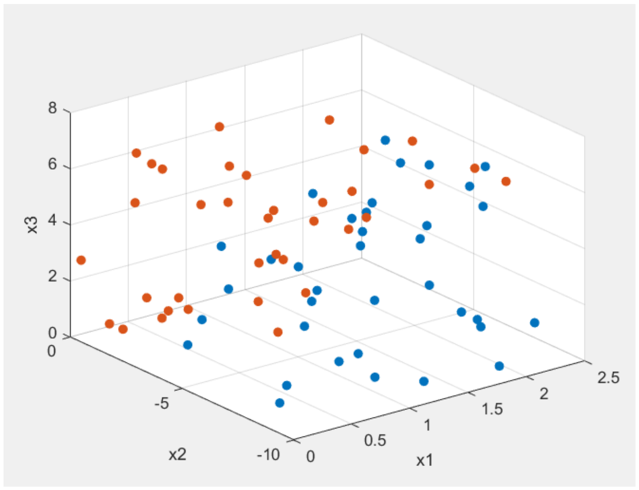
2 Question 2

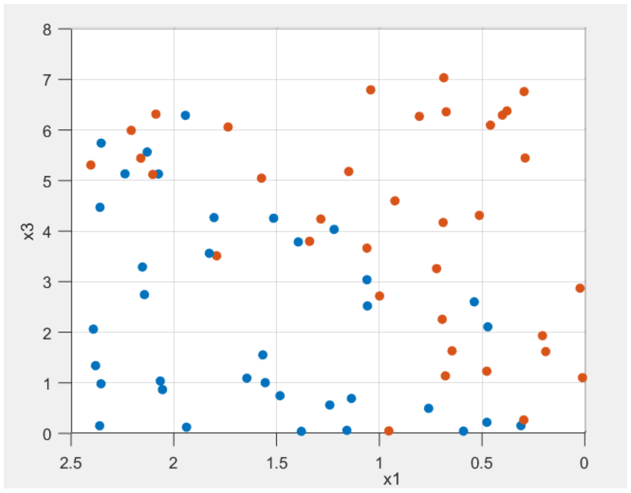
As of the first step we load the provided dataset and randomly divide the training and validation data. The code for doing so can be seen below:

```
% load data
data = load('../Data/Ex2.mat');
trainVal_data = data.TrainData;
test_data = data.TestData;

% Split into training and validation sets (80% training, 20% validation)
[trainInd, valInd, ~] = dividerand(size(trainVal_data, 2), 0.8, 0.2, 0);
trainX = trainVal_data(1:3, trainInd);
trainY = trainVal_data(4, trainInd);
valX = trainVal_data(1:3, valInd);
valY = trainVal_data(4, valInd);
```

Now for having a better understanding of our data, here we have visualize the training data for both classes using the following code.

```
cls1_idx = find(trainY == 1);
class1X = trainX(:, cls1_idx);

cls2_idx = find(trainY == 0);
class2X = trainX(:, cls2_idx);

figure;
scatter3(class1X(1,:), class1X(2,:), class1X(3,:), 'filled'); hold on;
scatter3(class2X(1,:), class2X(2,:), class2X(3,:), 'filled');
```





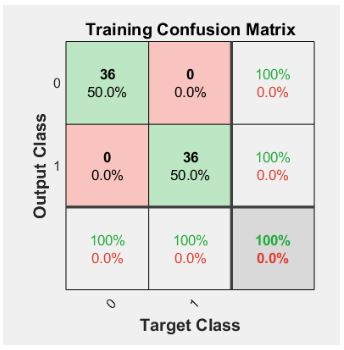
2.1 MLP with one output neuron

In this section we aim to design a neural network containing only one output neuron. The network is simply created using built-in function ”patternnet()”.

```
% Define the number of neurons in the hidden layer
hiddenSize = 10;
% Create the MLP model
```

```
    net = patternnet(hiddenSize);
    net.divideParam.trainRatio = 100/100;
    net.divideParam.testRatio = 0/100;
    net.divideParam.valRatio = 0/100;

    % Train the MLP model
    net = train(net, trainX, trainY);

    val_predict = round(net(valX));
    error = sum(abs(val_predict - valY));
    disp(error)
```

Here in order to assess the performance of the network we have calculated the number of the cases the network has estimated the class labels wrongly for validation set and print this value. Also the confusion matrix for training data can be seen below.



Now we simply utilize the trained model for the test data and finally save the predicted labels.

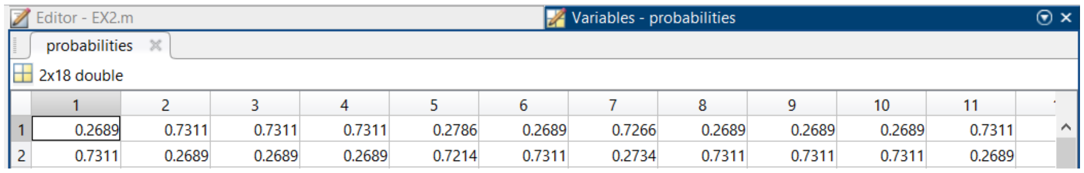
```
    % Load or define your test data
    testX = test_data(1:3,:); % Test data features

    % Convert the continuous output to binary labels
    predictedLabels = round(net(testX));
    save("Testlabel_a.mat", 'predictedLabels');
```

2.2 Two output neurons

The code for this section with appropriate comments for further explanations is provided below:

```
    % Set the number of output neurons to 2
    net.layers{end}.size = 2;
    net.divideParam.trainRatio = 100/100;
    net.divideParam.testRatio = 0/100;
    net.divideParam.valRatio = 0/100;
    % Train the MLP model
    net = train(net, trainX, [1-trainY; trainY]);

    val_predict = net(valX);
    % Apply softmax function to obtain class probabilities
    probabilities = softmax(val_predict);
    % Determine the predicted labels based on the maximum probability
    [~, valLabels] = max(probabilities);
    valLabels = valLabels - 1;
```

The first output neuron determines the probability of the data to belong to the first class with label 0 and the second output layer to the same for the second layer After training the model with trainX (input features) and trainY (target labels), it predicts the class probabilities for the validation set valX using the softmax function. Finally, it determines the predicted labels by selecting the class with the highest probability for each example in the validation set. The max function is used to find these labels, which are stored in valLabels. The predicted probabilities can be seen below



2x18 double

	1	2	3	4	5	6	7	8	9	10	11
1	0.2689	0.7311	0.7311	0.7311	0.2786	0.2689	0.7266	0.2689	0.2689	0.2689	0.7311
2	0.7311	0.2689	0.2689	0.2689	0.7214	0.7311	0.2734	0.7311	0.7311	0.7311	0.2689

Now we apply the network for out test data and stores the resulting labels:

```
% Make predictions on the test data
testY = net(testX);

Y_probabilities = softmax(testY);
% Determine the predicted labels based on the maximum probability
[~, yLabels] = max(Y_probabilities);
yLabels = yLabels - 1;

save("Testlabel_b.mat", 'yLabels');
```

There's no definitive answer as to which design is better; it depends on the context. If interpretability and computational efficiency are priorities, a single output neuron might be preferable. If we want a more detailed probability distribution over the classes, two output neurons might be the better choice.

3 Question 3

First we complete the definition of of the derivative of the functions declared in the begging of the notebook as follows:

```
def sigmoid_prime(x):
  return sigmoid(x) * (1 - sigmoid(x))

def tanh_prime(x):
  return 1 - np.tanh(x)**2
```

In the main function of the code, we simply define the inputs and outputs based on the truth table:

```
if __name__ == '__main__':

  nn = NeuralNetwork([2,2,1],'tanh')

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

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

Now we complete the updating rule based on error back propagation algorithm:

```
for i in range(len(self.weights)):
  layer = np.atleast_2d(a[i])
  delta = np.atleast_2d(deltas[i])
  self.weights[i] += learning_rate * layer.T.dot(delta)
```

Also in the code, we print the value of error every 500 epochs. Also we have a defined an array to store the absolute value of the errors :

```
error = y[i] - a[-1]
if (k%500 == 0):
  error_arr.append(abs(error))
  print(f"Epoch {k}, error = {error} \n")
```

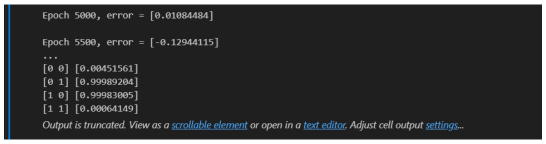
Finally we assign this array as an attribute of the class after the for loop terminated:

```
self.error_arr = error_arr
```

As of the next step, we create a function for the NeuralNetwork class to plot and visualize the error through epochs

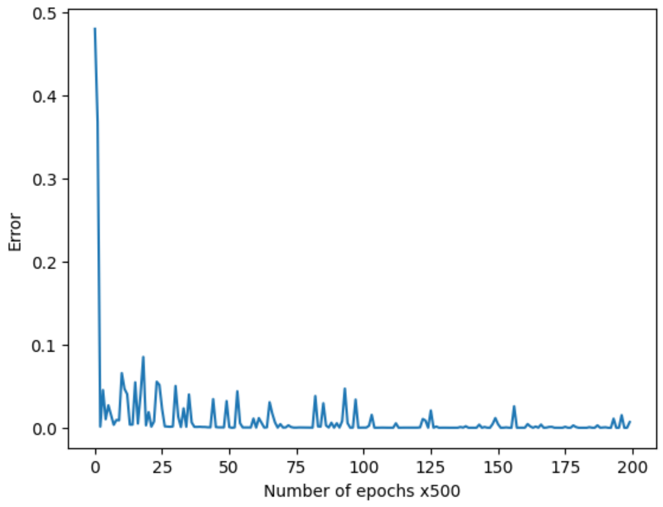
```
def plot_errors(self):
  plt.plot(self.error_arr)
  plt.xlabel('Number of epochs x500')
  plt.ylabel('Error')
```

Now we execute the main function and observe the results below:



```
Epoch 5000, error = [0.01084484]

Epoch 5500, error = [-0.12944115]
...
[0 0] [0.00451561]
[0 1] [0.99989204]
[1 0] [0.99983005]
[1 1] [0.00064149]
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```

The curve demonstrating the error for epochs can be also seen below:

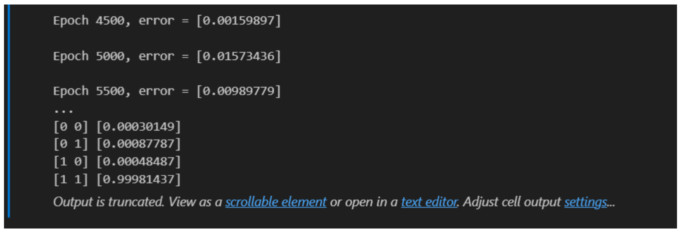


It can be seen vividly that the absolute value of the error has decreased as the model examine more epochs and finally the error will tend to be so close to zero.

3.1 Part 2

In this part we examine the network for logical AND function. We define the output (y) correspondingly as follows:

```
    if __name__ == '__main__':

        nn = NeuralNetwork([2,2,1],'tanh')

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

        nn.fit(X, y)

        for e in X:
            print(e,nn.predict(e))
```

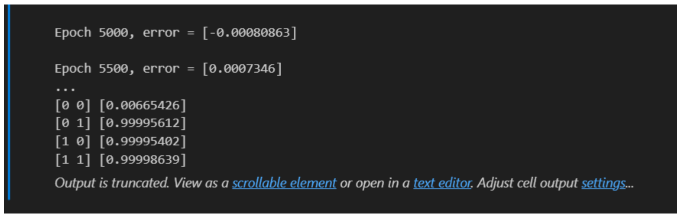


```
Epoch 4500, error = [0.00159897]

Epoch 5000, error = [0.01573436]

Epoch 5500, error = [0.00989779]
...
[0 0] [0.00030149]
[0 1] [0.00087787]
[1 0] [0.00048487]
[1 1] [0.99981437]
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```

We do the same for logical OR function too

```
    if __name__ == '__main__':

        nn = NeuralNetwork([2,2,1],'tanh')

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

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

        nn.fit(X, y)

        for e in X:
            print(e,nn.predict(e))
```

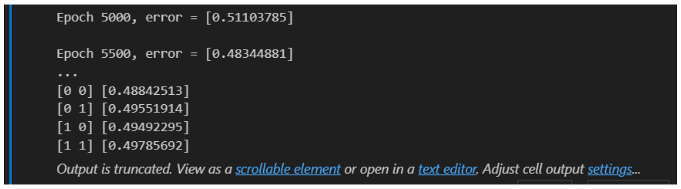


```
Epoch 5000, error = [-0.00080863]

Epoch 5500, error = [0.0007346]
...
[0 0] [0.00665426]
[0 1] [0.99995612]
[1 0] [0.99995402]
[1 1] [0.99998639]
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```

3.2 Sigmoid activation function

In this part we simply change the activation function to sigmoid one as follows:

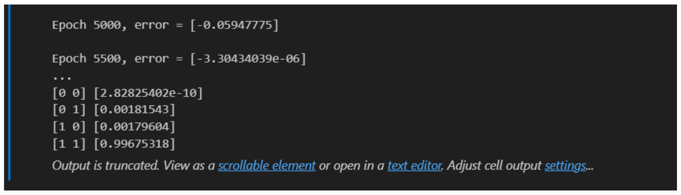
```python
    if __name__ == '__main__':

        nn = NeuralNetwork([2,2,1],'sigmoid')

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

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

        nn.fit(X, y)

        for e in X:
            print(e,nn.predict(e))
```



```
Epoch 5000, error = [0.51103785]

Epoch 5500, error = [0.48344881]
...
[0 0] [0.48842513]
[0 1] [0.49551914]
[1 0] [0.49492295]
[1 1] [0.49785692]
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```

Here we assess the logistic activation function for logical AND problem

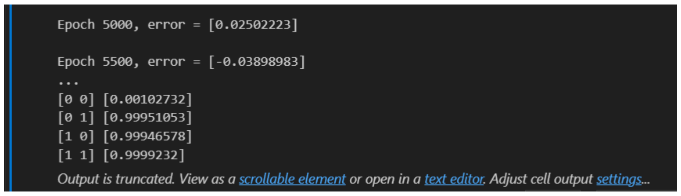
```python
    if __name__ == '__main__':

        nn = NeuralNetwork([2,2,1],'sigmoid')

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

        nn.fit(X, y)

        for e in X:
            print(e,nn.predict(e))
```



```
Epoch 5000, error = [-0.05947775]

Epoch 5500, error = [-3.30434039e-06]
...
[0 0] [2.82825402e-10]
[0 1] [0.00181543]
[1 0] [0.00179604]
[1 1] [0.99675318]
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```

Here we assess the logistic activation function for logical OR problem

```python
if __name__ == '__main__':

  nn = NeuralNetwork([2,2,1],'sigmoid')

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

  nn.fit(X, y)

  for e in X:
    print(e,nn.predict(e))
```



```
Epoch 5000, error = [0.02502223]

Epoch 5500, error = [-0.03898983]
...
[0 0] [0.00102732]
[0 1] [0.99951053]
[1 0] [0.99946578]
[1 1] [0.9999232]
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```

As it can be seen below, this code randomly initialize the weight matrices between the input and hidden layers of a neural network every time we create a network. Therefor the results we achieve every time we run the code of this section we expect to see different results.

```python
# input and hidden layers - random((2+1, 2+1)) : 3 x 3
for i in range(1, len(layers) - 1):
  r = 2*np.random.random((layers[i-1] + 1, layers[i] + 1)) -1
  self.weights.append(r)
# output layer - random((2+1, 1)) : 3 x 1
r = 2*np.random.random( (layers[i] + 1, layers[i+1])) - 1
self.weights.append(r)
```