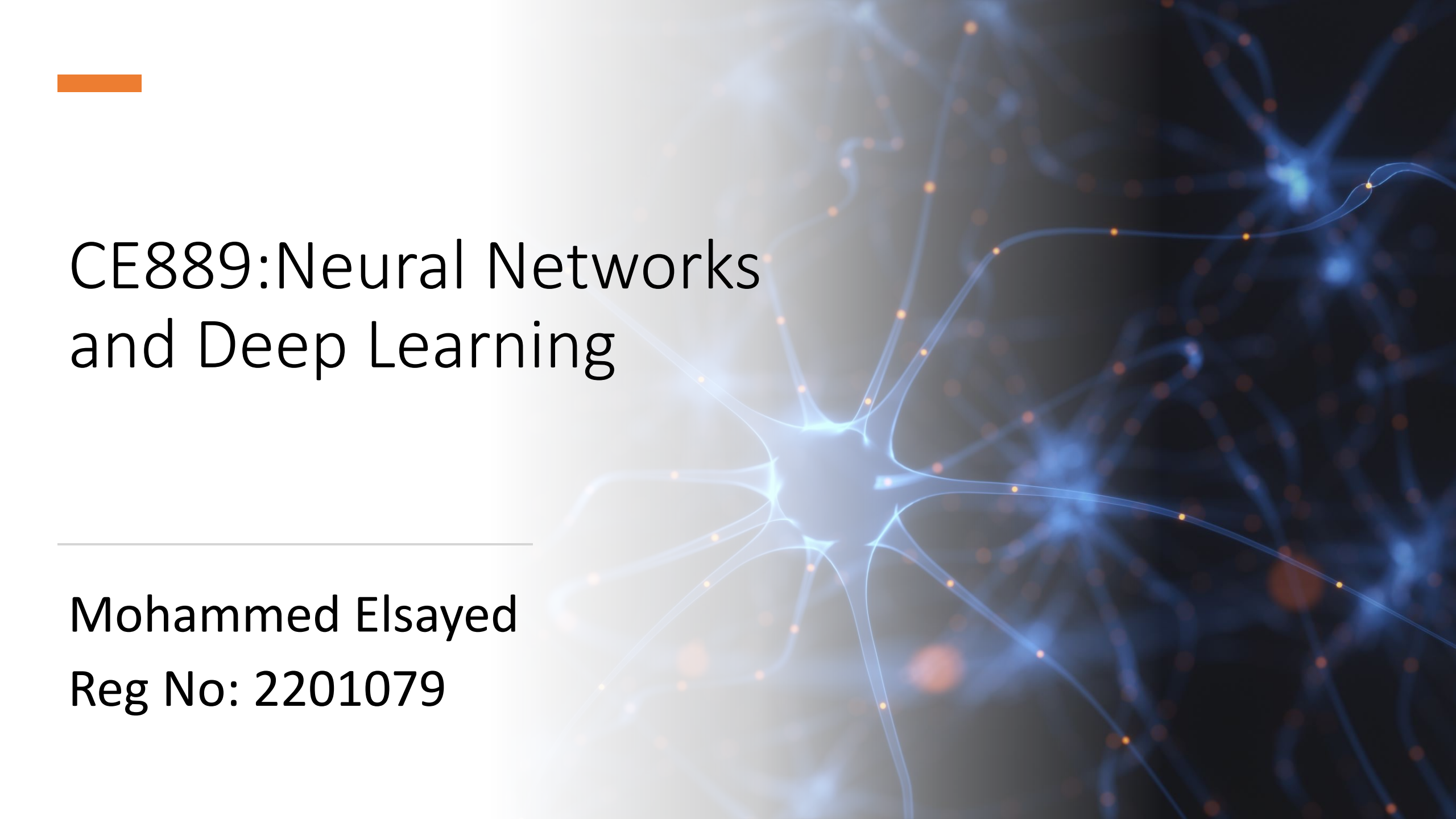
CE889:Neural Networks and Deep Learning

Mohammed Elsayed

Reg No: 2201079



Overall work

Read the CSV file for data collection after collecting the data from playing the game multiple times

Data Processing

Normalizing the Data

Split the data to Train and test

Neural network structure for feed forward and back propagation.

Train the Model and calculate MSE and stopping criteria

validate the model and get MSE overall

saving the weighs to a CSV file

Prediction Function

Neural Holder to integrate with the game

Data processing

```
[ ]  def normalization_process (X, max_X,min_X):
       Value = (X-min_X)/(max_X - min_X)
       return Value
```

```
[ ] from sklearn.model_selection import train_test_split
    X_train, X_test,y_train, y_test = train_test_split(tem_data_norm[['X1', 'X
```

```
[ ] X_train.shape,X_test.shape,y_train.shape,y_test.shape

    ((3054, 2), (1309, 2), (3054, 2), (1309, 2))
```

```
data.skew()

X1    0.036520
X2    0.253583
Y1    0.634892
Y2    0.207972
dtype: float64
```

```
data.describe()
```

	X1	X2	Y1	Y2
count	6816.000000	6816.0	6816.0	6816.0
unique	6519.000000	6407.0	6389.0	5465.0
top	132.512816	307.1	-0.1	0.0
freq	2.000000	42.0	25.0	298.0

```
[ ] data.info()
```

```
[ ] data.dtypes

    X1    object
    X2    object
    Y1    object
    Y2    object
    dtype: object
```

```
data.drop(0,axis = 0,inplace = True)
```

```
[ ] data["X1"] = data["X1"].astype(float)
    data["X2"] = data["X2"].astype(float)
    data["Y1"] = data["Y1"].astype(float)
    data["Y2"] = data["Y2"].astype(float)
```

```
data.head()
```

	-87.8088028089586	307.1	0.0	0.0.1
0	-87.808803	307.1	-0.1	0.04
1	-87.848803	307.2	-0.2	0.00
2	-87.848803	307.4	-0.3	0.04
3	-87.888803	307.7	-0.4	0.00
4	-87.888803	308.1	-0.5	0.04

```
[ ] data = pd.DataFrame(np.vstack([data.columns, data]))
```

```
[ ] data.reset_index(inplace = True, drop = True)
```

```
[ ] data.rename({0:"X1",1:"X2",2:"Y1",3:"Y2"},axis = 1,inplace = True)
```

Reading the data from CSV file and rename the columns X1,X2,Y1,Y2

Describing and getting some information from the data

Check the null values and duplicated values

Covert the data type form object to Float

Check the normal distribution of the data found all the column are within the range of -.5 to 0.5

Normalizing the data between 0-1 using the equation

Splitting the data to 70% train and 30% test and first 2 column is input and second 2 column is output.

Neural Network structure

Input neurons
2 neurons

Hidden layers
neurons is variables

Output neurons
2 neurons

neural
network class

Initializing
weights

Hyperparameters
Stopping criteria

Create loop that take every row to (forward →then back propagation →update weights)→ next row inside one epoch

```
# ****************input to neural network X_train,y_train// X_test,y_test**************************
inputx=np.array(X_train)
input_p = np.array (X_test)
outy=np.array(y_train)
out_p= np.array(y_test)

#*********************************structure of the neural network **********************************
inputlayer_neurons  = inputx.shape[1] # number of input layers neurons--> the shape for input data,2 neurons.
hiddenlayer_neurons = 20              # number of hidden layers neurons--> get the values from MATLAP
output_neurons      = outy.shape[1]   # number of output layers neurons---> the shape of output data,2 neurons

#*************** Hyper prameters get the values from MATLAP*****************************************
lr= 0.0001 # learning rate
lamda= 0.6 # the value of lamda
momentum=0.6
epoch = 157 # number of iteration
stopping_criteria = 0.0 # stoppin cretaria

#************************************ class Neural network ***************************************#
class NeuralNetwork:
    def __init__(self,  epoch,inputlayer_neurons, hiddenlayer_neurons, output_neurons ):
        self.inputx = inputx
        self.outy = outy
        self.lr = lr
        self.epoch = epoch
        self.lamda= lamda
        self.momentum = momentum
# structure of neural network:
        self.inputlayer_neurons  = inputlayer_neurons
        self.hiddenlayer_neurons = hiddenlayer_neurons
        self.output_neurons      = output_neurons
# intialize the weights and bias:
        global wh,bh,wout,bout
        self.wh=np.random.uniform(size=(self.inputlayer_neurons,self.hiddenlayer_neurons))
        self.bh=np.random.uniform(size=(1,self.hiddenlayer_neurons))
        self.wout=np.random.uniform(size=(self.hiddenlayer_neurons,self.output_neurons))
        self.bout=np.random.uniform(size=(1,self.output_neurons))
```

✓ 45s completed at 17:11

Feed forward

```
#************************Feed Forward ***********************************
    def forward (self, input):
    # from input to hidden layer output
        hidden_layer_input1 = self.weightMulitply(input,self.wh)
        hidden_layer_input1= hidden_layer_input1+self.bh
        hidden_layer_activation = self.sigActivationFunc(hidden_layer_input1)
        output_hidden_layer= hidden_layer_activation
    # from hidden to output layer
        output_layer_input2 = self.weightMulitply(hidden_layer_activation,self.wout)
        output_layer_input2=output_layer_input2+self.bout
        output_layer_activation = self.sigActivationFunc(output_layer_input2)
        output = output_layer_activation
        return output, output_hidden_layer
# weight multiplication
    def weightMulitply(self,input,weight):
      V=np.dot(input,weight)
      return V

# activation function
    def sigActivationFunc(self,V):
      activation=self.sigmoid(V)
      return activation

# defining the Sigmoid Function
    def sigmoid (self, z):
        return 1/(1 + np.exp(-self.lamda*z))

#++++++++++++++++++++++++++ Back Propogation ++++++++++++++++++++++++++++++++++
```

Multiply — Input by the weights assigned randomly before to get the value of V

↓

Run — Activation function of V values to get the hidden layers values

↓

Multiply — Hidden layers values to get the values of V and then run the activation function to get the output values from the model.

↓

Run — Run the activation function of V values to get the Output

Back propagation

```
#************************* Back Propogation ************************************
    def back_propgation(self,output,error,out_hidden,row,i):
        gradient_descent = self.gradient_descent(output,error)
        local_gradient = self.local_gradient(gradient_descent,out_hidden)

        self.wout = self.output_weight_updation(out_hidden, gradient_descent,row,i)
        self.wh = self.input_weight_updation(inputx[row], gradient_descent,row,i)
        return self.wout,self.wh

# clculate The gradients (output layer and hidden layer)
    def gradient_descent(self, output, error):    # out layer:
        gradient_descent = self.derivatives_sigmoid(output) * error

        return gradient_descent

    def local_gradient(self,gradient_descent,out_hidden):  #hidden layer:
        a =  np.dot(gradient_descent,self.wout.T)
        local_gradient = a * self.derivatives_sigmoid(out_hidden)
        return local_gradient

# derivative of Sigmoid Function
    def derivatives_sigmoid(self, z):
        return lamda *z * (1 - z)
#************************ Error calculation ***************************************
# calculate the error:
    def error (self,output, actualOutput):
        E = actualOutput - output
        E_sq = E**2
        return E, E_sq
```

```
#********************** Updating the weights ************************************
#updating weights:
    def output_weight_updation(self,output, gradient_descent,row,i):  # output weights.

        Delta_weight_out1 = np.dot(output.T, gradient_descent) * self.lr
        Delta_weight_out2 = Delta_weight_out1 + self.momentum * Delta_weight_out1
        if row==0 and i==0:#for first row in first epoic
          self.wout += np.sum(Delta_weight_out1,axis=0,keepdims=True)
        if row>0 :
          self.wout += np.sum(Delta_weight_out2,axis=0,keepdims=True)
        return self.wout

    def input_weight_updation(self,input, local_gradient,row,i): # input weights.

        Delta_weight_inp1 = np.dot(input, local_gradient.T) * self.lr
        Delta_weight_inp2 = Delta_weight_inp1 + self.momentum * Delta_weight_inp1
        if row == 0 and i ==0: #for first row in first epoic
          self.wh += np.sum( Delta_weight_inp1,axis =0, keepdims=True )
        if row > 0 :
          self.wh += np.sum( Delta_weight_inp2,axis =0, keepdims=True )
        return self.wh
```

Calculate	Error between the actual output on the train data and the output from the model.
Calculate	Gradients δ output layers =λ*y(1-y)*error
Calculate	Delta weights $\nabla \mathbb{W}=\alpha\ \delta$ Y → where α is the learning rate
update	weights at output layers
Calculate	local gradients δ at hidden layers =λ *h(1-h)*Error*$\sum(\delta * \mathbb{W})$ → where $\mathbb{W}$ is the original weights from previous iteration.
Calculate	Delta weights at input layers $\nabla \mathbb{W}=\alpha\ \delta$ X → where α is the learning rate
Update	weights at input layers
Note	In second iteration " multiply Delta weights by (momentum to the Delta weights previous) to increase the speed to global minima.

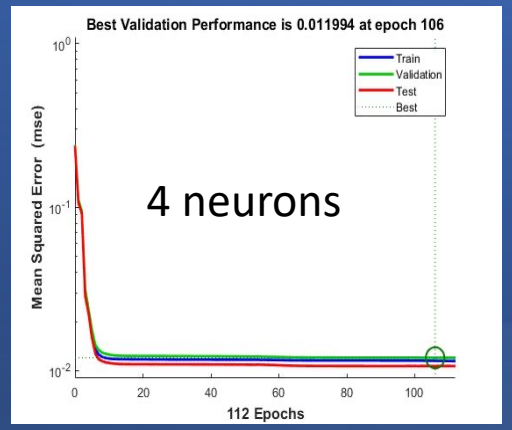
Training the Model

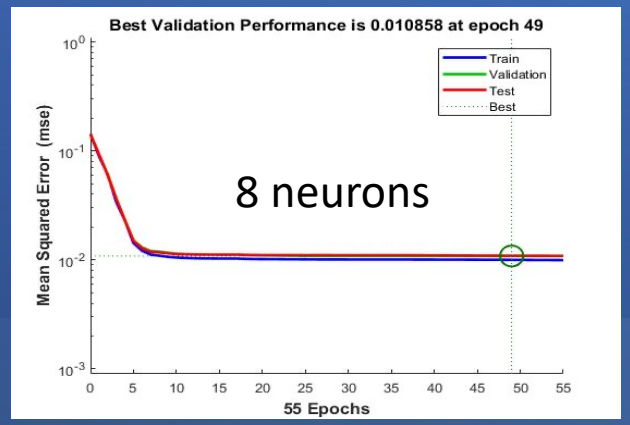
```
#********************************** Tain The Model by (X_train) and Y_train) ************************************

    def train(self,inputx, outy):
      global stopping_criteria
      self.MSE_list =[]
      self.nu_it_lst=[]
      for i in range(epoch):
        self.nu_it_lst.append(i)
        Error=[]
        Error_squared =[]
        for row in range(len(inputx)):
            output, out_hidden = self.forward(inputx[row])         # calling feed forward
            error,E_sq = self.error(output,outy[row])              # calculate the error
            Error.append(error)                                    # saving the error after each epoch
            Error_squared.append(E_sq)                             # saving the error square after epoch to calculate MSE
            self.back_propgation(output,error,out_hidden,row,i)    # calling back Propgation

        MSE = np.mean(Error_squared)                               # calculate Mean square error
        print("Epoch--",i,"--MSE--------------", MSE)              # print MSE with every epoic
        self.MSE_list.append(MSE)
        if float("{:.8f}".format(MSE)) == float("{:.8f}".format(stopping_criteria)): # Stopping criteria
            print('Stopping')
            break
        else:
            stopping_criteria = MSE                                 # saving mean square error to plot the graph with each
#********************************** validate the Model ************************************************************
```

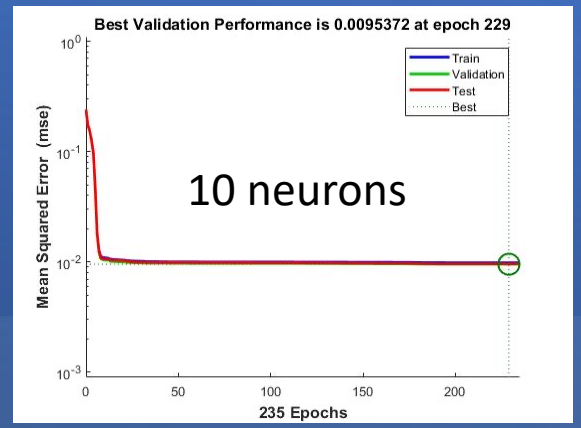
```
#**************************# calling the Neural network #**********************************
network = NeuralNetwork(epoch,inputlayer_neurons,hiddenlayer_neurons,output_neurons) # numb
network.train(inputx,outy)    # train the model in input and output from the data"X_train,y
network.savingweights()       # saving weights after train
network.plottingGraph()       # plotting the graph for MSE with every epoch.
print( " \n *********Testing of the Model********** \n ")
network.Testing(input_p,out_p)# validate the model and get Mse
network.plottingGraph_V()     # plotting a graph for distribution of MSE on testin data
```

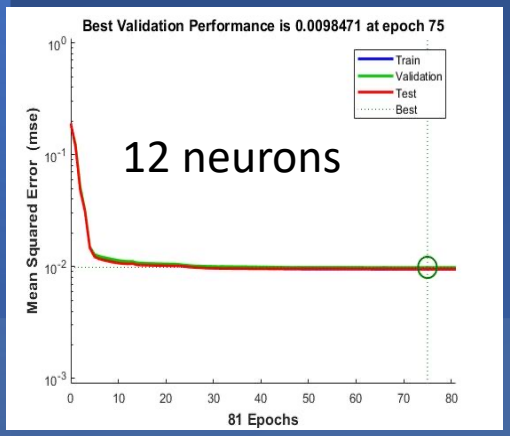
- Applying the structure of the class to pass throw tha data each row for feed forward → backward→ updating weights → next row
- After finish all the rows for the training samples repeat again for multiple number of epochs.
- Calculating Mean Square Error for every epoch and plotting on a graph to visualize the performance of the training process
- Stopping Criteria either the number of epoch reached or the MSE become similar so no significant change in MSE
- Saving the final weights to a CSV files for using farther

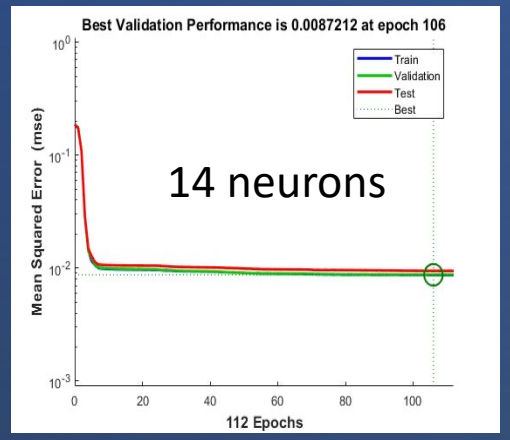
Uploading the normalized Data to MATLAP to adjust hyper parameters " LAMDA,EITA,MOMUNTUM,EPOCH)

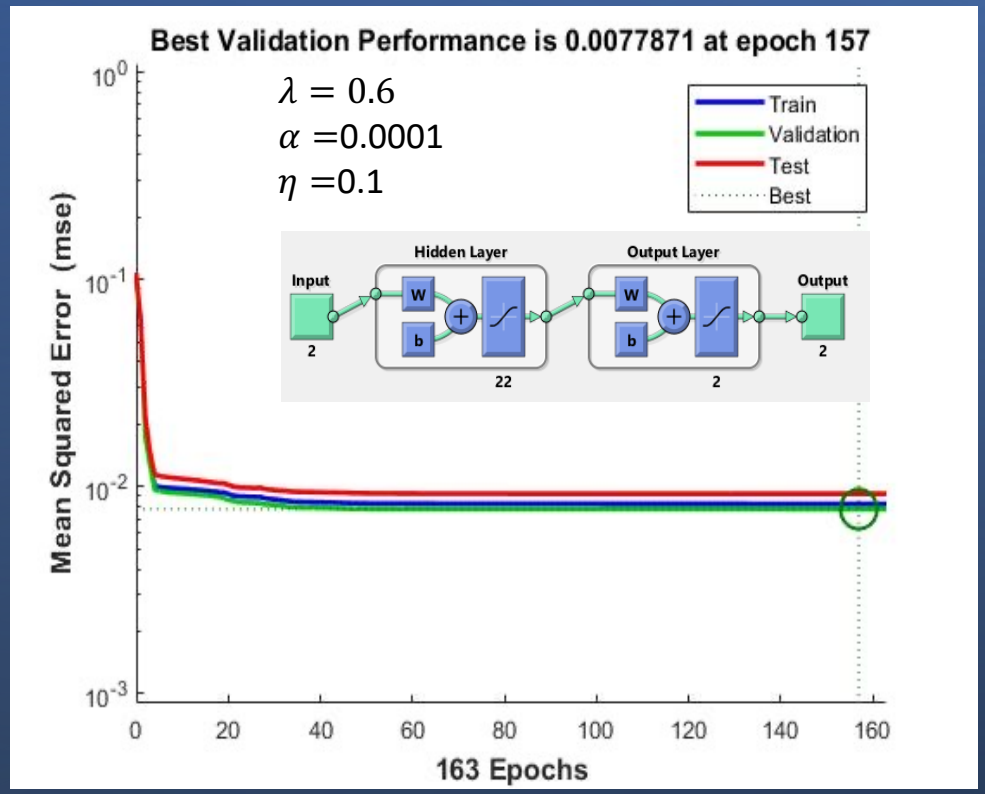


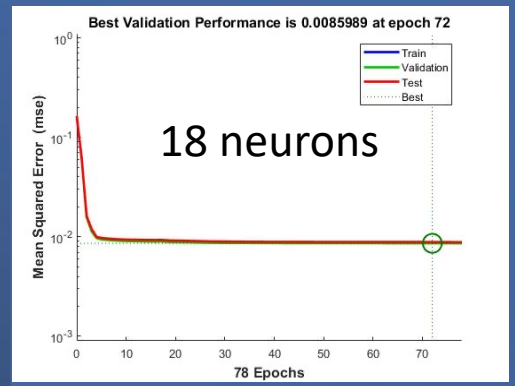


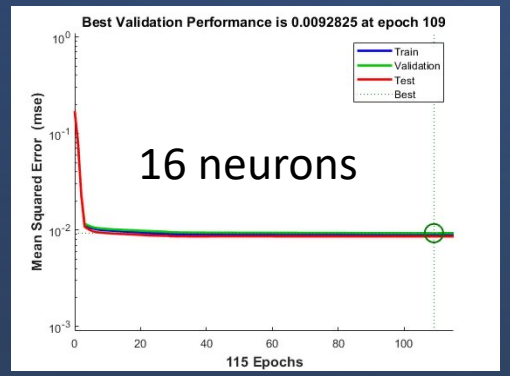


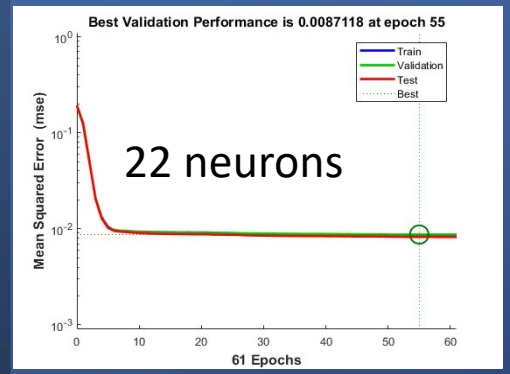




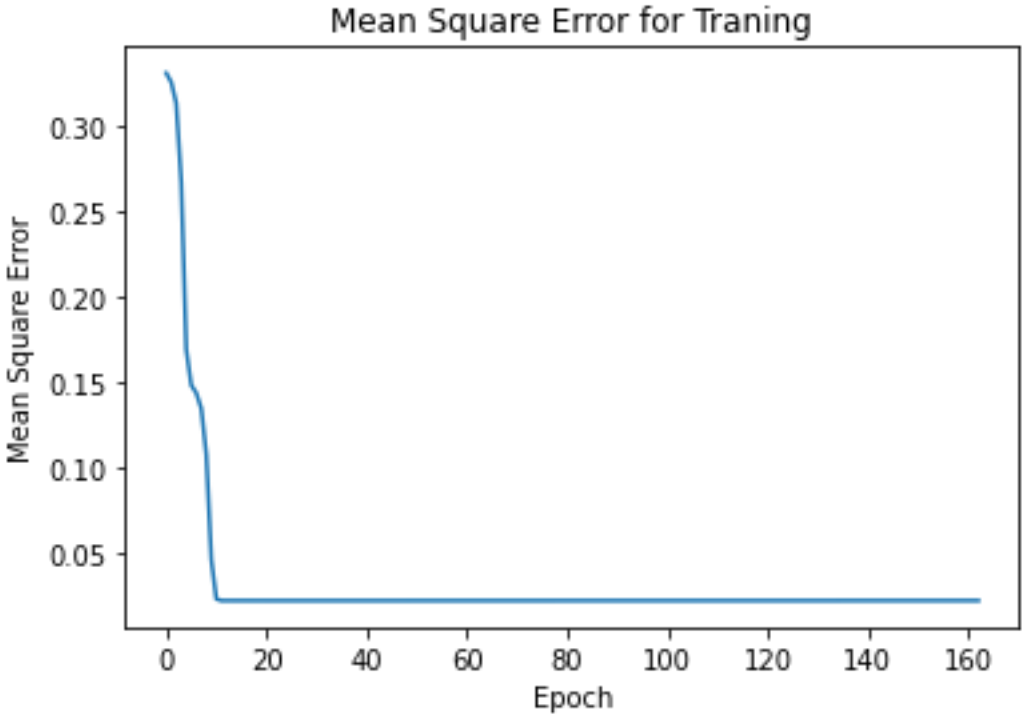


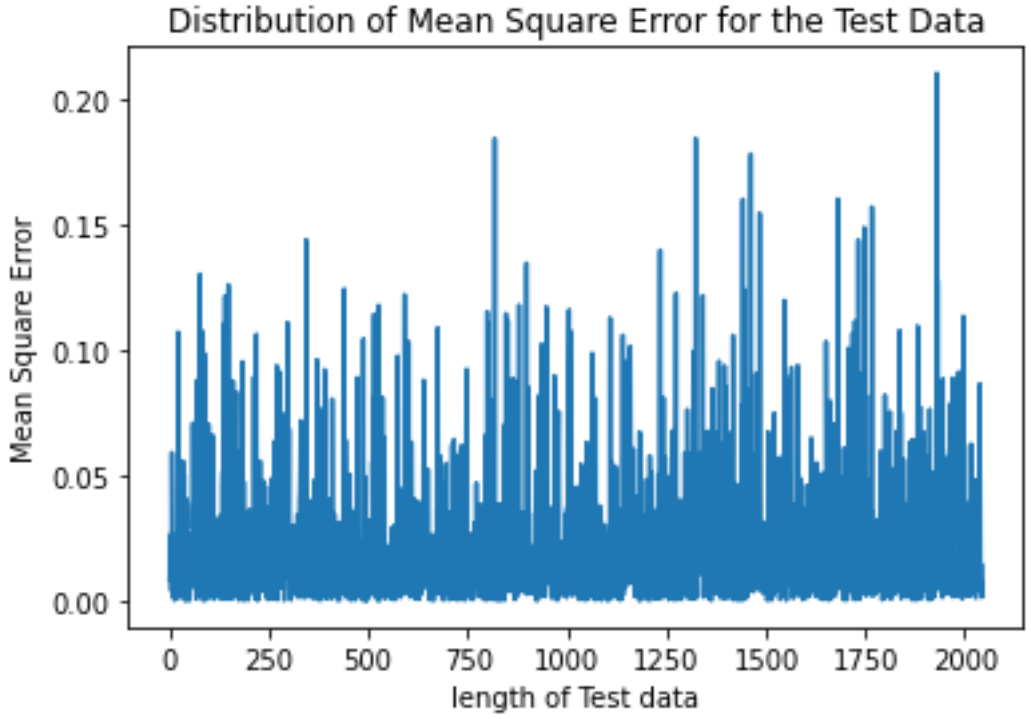






Mean Square Error






```
*********Testing of the Model**************
|*** The Average of Mean Square error : =  0.021532979653388875
#### The graph for Distribution of MSE at test data #########
```

Integration with the game

- Create a prediction files containing:
  - Reading the weights from CSV files
  - prediction function that will calculate the output through a feed forward
- Neural Holder file:
  - Normalize the input _row from the game
  - Calculate the output from prediction function
  - De-Normalize the output
  - Return the output in two values

Thank you