Can Kocagil
21602218

Bilkent University

Department of Electric and Electrical Engineering

EEE443/543 Neural Networks

Mini Project 1

30.09.2020

TABLE OF CONTENTS

1 QUESTION 1

In the first question, there is single neuron that receives input from m input neurons with weights W_i, where i ϵ [1 m]. The neuron is expected to predict the probability that the output t belongs to Class A (t = 1) versus Class B (t = -1). A datasets of training samples are available with inputs x^n and outputs y^n (n ϵ [1 N]). We know that the maximum a posteriori estimate for the network weights are obtained by solving the following optimization problem:

$$\arg min_W \sum_n (y^n - h(x^n, W)^2) + \beta \sum_i W_i^2$$

where W is the vector of weights W_i, β is a scalar constant, and h(.) is the output of the neuron. According to this estimate, here is the derivation the prior probability distribution of the network weights.

To do that, I am going to use the rule of random variables that implies that the posterior probability distribution is the multiplication of its likelihood and prior probability using Bayes rule. Therefore, in order to do the multiplication, I convert the arg min_W function to arg max_W so that we can proceed.

The next step is converting, to convert arg min_W to arg max_W, I apply the exponential decaying function that is reverse operation of natural logarithm function:

$$F(x) = e^{-X}$$

The reason why I use the function f(X) is that e^{-X} a decreasing function all the time since the minus term so it can convert the function into arg max_W bases that is what we want.

Since the given function is cost function with the regularization term we can assign that (or may be abbreviated as J(W):

$$\text{Cost(W)} = \arg min_W \sum_n (y^n - h(x^n, W)^2) + \beta \sum_i W_i^2$$

Then we apply the exponential function to cost function to convert the arg min_W function to arg max_W as I discuss:

$$\begin{aligned} \text{F(Cost(W))} &= \arg max_W \, e^{-\sum_n (y^n - h(x^n, W)^2) + \beta \sum_i W_i^2} \\ &= \arg max_W \, e^{-\sum_n (y^n - h(x^n, W)^2)} e^{-\beta \sum_i W_i^2} \end{aligned}$$

In this point, I assumed that samples in the dataset is distributed by Gaussian(Normal) distribution so that we can use the rule of Maximum a Posterior (MAP) estimation of the weights W_i. By the way, for following calculations, I drop out the terms W and Y in the subscript so that $P_{W|Y}$ (W|y^n) = P(W|y^n) for the sake of simplified notation. Then, by using Bayes' rule:

$$\text{P(W}|y^n) = \frac{P(W) * P(y^n|W)}{\int P(W) * P(y^n|W) * dW}$$

Since the denominator part is constant, we can ignore and proceed without it. Therefore, we can write it as a:

$$\text{P(W}|y^n) \sim \text{P(W)} * P(y^n|W)$$

So arg max_W function implies that we can accept the term with the squared error as a likelihood of the function and the sum of the squared weights multiplied with a constant can be separated as a prior distribution of the weights with the constant term that was ignored.

Lastly, I made an assumption that the data is drawn from Gaussian Distribution, we can separate the inside of arg max as follows and assume the likelihood and prior are in the form:

Likelihood

$$L = \frac{1}{A} * e^{-\frac{\varphi * \sum n(y^n - h(x^n, W))^2}{2}}$$

Prior

$$P = A * e^{-\beta \sum_i W_i{}^2}$$

So, the next job is to find A, to do that I use the normalization axiom of probability that implies that sum over all probability distribution equals to 1, in continuous case:

$$\int_{-\infty}^{\infty} F_x(X) dX \overset{\text{def}}{=} 1$$

That yields:

$$\int_{-\infty}^{\infty} A * e^{-\beta \sum_i W_i{}^2} \mathrm{d}W_i = 1$$

Since we have m neurons, we can divide the integral into m parts then multiply all:

$$\int_{-\infty}^{\infty} A * e^{-\beta W_1{}^2} \mathrm{d}W_1 \int_{-\infty}^{\infty} A * e^{-\beta W_2{}^2} \mathrm{d}W_2 \int_{-\infty}^{\infty} A * e^{-\beta W_3{}^3} \mathrm{d}W_3 \dots \int_{-\infty}^{\infty} A * e^{-\beta W_m{}^2} \mathrm{d}W_m$$

Since we know that:

$$\int_{-\infty}^{\infty} e^{-\beta W_i{}^2} \mathrm{d}W_i = \frac{\sqrt{\pi}}{\sqrt{\beta}}$$

$$A\left(\frac{\sqrt{\pi}}{\sqrt{\beta}}\right)^m = 1$$

Finally, the Prior distribution becomes,

$$P = \left(\frac{\pi}{\beta}\right)^{\frac{m}{2}} * e^{-\beta \sum_i W_i^2}$$

2 QUESTION 2

An engineer would like to design a neural network with a single hidden layer with four input neurons (with binary inputs) and a single output neuron to implement:

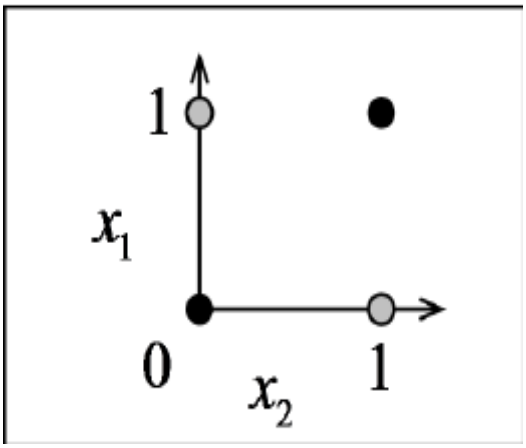
(X1 OR NOT X2) XOR (NOT X3 OR NOT X4)

For following parts, I am assuming a hidden layer with four hidden nodes, and a step function (unipolar activation function) as an activation function of the neurons as proposed.

2.1 *PART A*

In this part of the question, I derived analytically the set of inequalities based on which a set of weights. As I said that I selected the step function as an activation unit of the neural network so that weighted sum of each neurons outputs 0 or 1. In this question, there is a non-linear logic gate implementation by using feed forward neural network that consist of several AND gates, NOT gates, OR gates and XOR gate. We should design the network/weights such that decision boundary should separate the classes successfully. We have non-linear separable data points that can be interpreted from XOR gate due to having of cascade AND and OR

gates, intuitionally. However, to be concrete, the following figure visualize the feature space of 2 input XOR gate implementation to clear non-linearity.



Let X_1 and X_1 be features, then the following figure is visualization of the XOR gate. The black points represent the values of (0,0) and (1,1) that outputs 0, then gray points represent the values of (0,1) and (1,0) that outputs to 1. Finally, we see that we cannot separate the classes linearly, so we should have hyperplane to separate classes that can be implemented by adding hidden layer to perceptron model of the neural network.

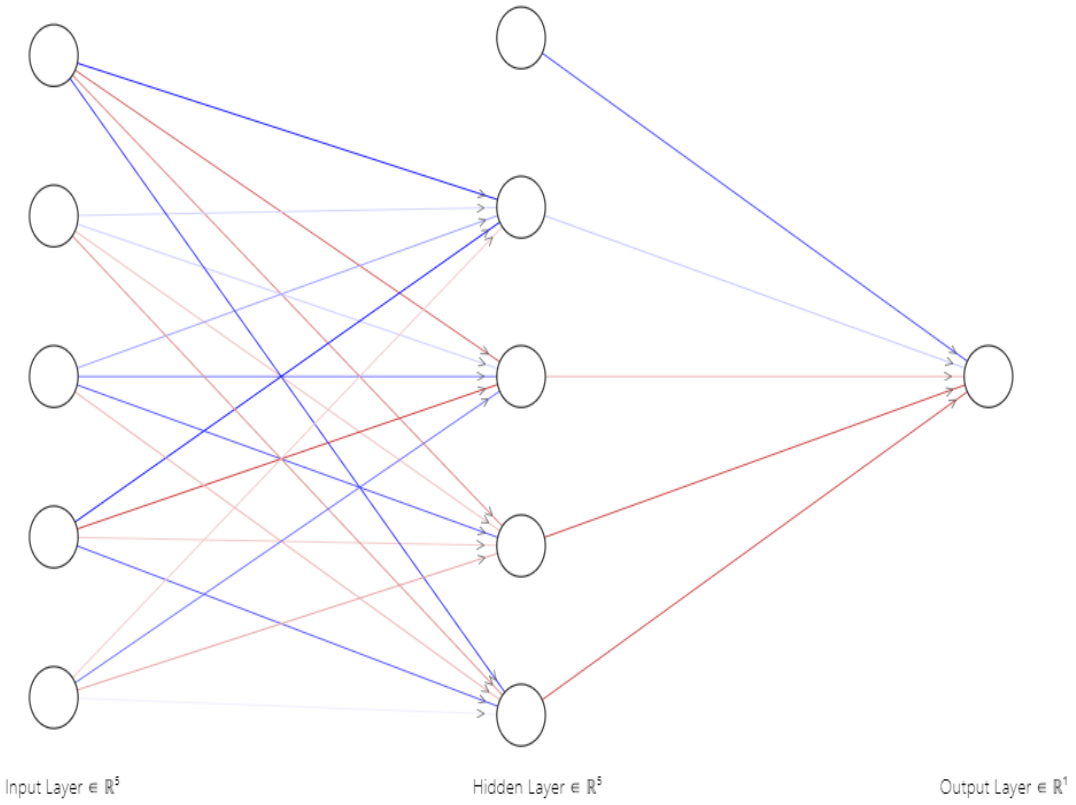
Let A and B the inputs of the XOR gate, then A XOR B can be viewed as a:

$$\text{A XOR B} = (\text{A AND } B') \text{ OR } (A' \text{ AND B})$$

Note that X' means $\overline{X}$. Then, using the above quantity, we have:

$$(X_1 + X_2\,') \oplus (X_3\,' + X_4\,') = (X_1 + X_2') (X_3 + X_4) + (X_1' X_2\ ) (X_3' + X_4')$$
$$= (X_1 X_3 X_4) + (X_2' X_3 X_4) + (X_1' X_2 X_3'\ ) + (X_1' X_2 X_4')$$

Therefore, we have four input to the network that is combination of 3 inputs. Each input can be treated separately by neurons of the networks. Moreover, the required neural network architecture is shown below:



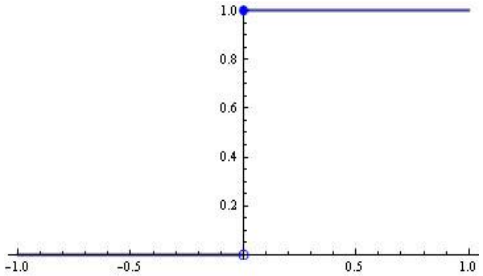
As shown, we have 4 input to the system with a bias term for an input layer. Then, we have a hidden layer that consist of 4 neurons with 1 bias term then finally an output layer with single neuron.

After that, we should find the ranges of set of inequalities for the design of the neural network that requires finding the weights of each layer with bias terms. Therefore, the following calculations describes the possible ranges of the weights and bias terms for each neuron.

2.1.1 Activation of Neurons

As we discuss, I selected the step function as an activation function of the each and every neuron. Given that number of values, step function outputs 1 if input is bigger than or equal to 0, else outputs 0 as shown below.

Step Function



$$\text{step}(x) = \begin{cases} 1, & x \geq 0 \\ 0, & x < 0 \end{cases}$$

Let X_1, X_2, X_3 and X_4 be inputs of the neural network, and let $W_{i,j}$ be weight of the i^{th} neuron in the previous layer with j^{th} neuron of the following layer, for i = 1,2,…,5 and j = 1,2,3,4. Then, A_k can be defined as output of the hidden layers neuron k^{th} for k = 1,2,3,4. Finally, $\vartheta_{1,2,3,4}$, ϑ_5 are bias term for hidden and output layer, respectively.

2.1.1.1 ***Activation of First Neuron***

In this node, we will be activate the $(X_1X_3X_4)$ term. After that, we know that $\boldsymbol{A_1}$ is a output of the first neuron in the hidden layer and can be viewed as:

$$\boldsymbol{A_1} = \text{step}(W_{11}X_1 + W_{13}X_3 + W_{14}X_4 - \vartheta_1)$$

Here, the following truth table describes the behavior of $X_1X_3X_4$ term so that corresponding inequalities is shown right side of the truth table.

X_1	X_3	X_4	A_1
0	0	0	**0**
0	0	1	**0**
0	1	0	**0**
0	1	1	**0**
1	0	0	**0**
1	0	1	**0**
1	1	0	**0**
1	1	1	**1**

Truth Table for $X_1X_3X_4$

İnequalities of 1st neuron

- $W_{13} + W_{14} - \vartheta_1 < 0$
- $W_{13} + W_{14} - \vartheta_1 < 0$
- $W_{11} + W_{13} - \vartheta_1 < 0$
- $W_{11} + W_{13} + W_{14} - \vartheta_1 \geq 0$
- $W_{11} - \vartheta_1 < 0$
- $W_{13} - \vartheta_1 < 0$
- $W_{14} - \vartheta_1 < 0$
- $-\vartheta_1 < 0$

2.1.1.2 Activation of Second Neuron

In this node, we will be activate the $(X_2{}'X_3\,X_4)$ term. Here, the following truth table describes the behavior of $(X_2{}'X_3\,X_4)$ term so that corresponding inequalities is shown right side of the truth table.

X_2	X_3	X_4	A_2
0	0	0	**0**
0	0	1	**0**
0	1	0	**0**
0	1	1	**1**
1	0	0	**0**
1	0	1	**0**
1	1	0	**0**
1	1	1	**0**

Truth Table for $X_2{}'X_3\,X_4$

İnequalities of 2nd neuron

- $W_{23} + W_{24} - \vartheta_2 \geq 0$
- $W_{22} + W_{24} - \vartheta_2 < 0$
- $W_{22} + W_{23} - \vartheta_2 < 0$
- $W_{22} + W_{23} + W_{24} - \vartheta_2 < 0$
- $W_{22} - \vartheta_2 < 0$
- $W_{23} - \vartheta_2 < 0$
- $W_{24} - \vartheta_2 < 0$
- $-\vartheta_2 < 0$

2.1.1.3 Activation of Third Neuron

In this node, we will be activate the $(X_1{}'X_2X_3{}')$term. Here, the following truth table describes the behavior of $(X_1{}'X_2X_3{}')$ term so that corresponding inequalities is shown right side of the truth table.

X_1	X_2	X_3	A_3
0	0	0	**0**
0	0	1	**0**
0	1	0	**1**
0	1	1	**0**
1	0	0	**0**
1	0	1	**0**
1	1	0	**0**
1	1	1	**0**

Truth Table for $(X_1{}'X_2X_3{}')$

İnequalities of 3rd neuron

- $W_{33} + W_{32} - \vartheta_3 < 0$
- $W_{31} + W_{33} - \vartheta_3 < 0$
- $W_{31} + W_{32} - \vartheta_3 < 0$
- $W_{31} + W_{32} + W_{33} - \vartheta_3 < 0$
- $W_{31} - \vartheta_3 < 0$
- $W_{32} - \vartheta_3 \geq 0$
- $W_{33} - \vartheta_3 < 0$

2.1.1.4 Activation of Fourth Neuron

In this node, we will be activate the $(X_1{}'X_2X_3{}')$term. Here, the following truth table describes the behavior of $(X_1{}'X_2X_3{}')$ term so that corresponding inequalities is shown right side of the truth table.

X_1	X_2	X_4	A_4
0	0	0	**0**
0	0	1	**0**
0	1	0	**1**
0	1	1	**0**
1	0	0	**0**
1	0	1	**0**
1	1	0	**0**
1	1	1	**0**

Truth Table for $X_1{}'X_2X_4{}'$

İnequalities of 4th neuron

- $W_{41} + W_{42} - \vartheta_4 < 0$
- $W_{41} + W_{44} - \vartheta_4 < 0$
- $W_{42} + W_{44} - \vartheta_4 < 0$
- $W_{41} + W_{42} + W_{44} - \vartheta_4 < 0$
- $W_{41} - \vartheta_4 < 0$
- $W_{42} - \vartheta_4 \geq 0$
- $W_{44} - \vartheta_4 < 0$
- $-\vartheta_4 < 0$

2.1.1.5 Activation of Output Neuron

Now, the implementation of the hidden layer is finished, the rest is the connect hidden layers' neurons output with OR gate to implement required system. As we discuss, A_i, where $i \in \{1,2,3,4\}$ represents the output of each neuron in the hidden layer. The mission of the output layer is to connect A_i's with OR gate. Therefore, the truth table for the implementation of the output neuron and their corresponding inequalities are shown below.

A_1	A_2	A_3	A_4	Output
0	0	0	0	**0**
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	**1**

Truth Table for $A_1 + A_2 + A_3 + A_4$

- $W_{51} - \vartheta_5 \geq 0$
- $W_{52} - \vartheta_5 \geq 0$
- $W_{53} - \vartheta_5 \geq 0$
- $W_{54} - \vartheta_5 \geq 0$
- $W_{51} + W_{52} - \vartheta_5 \geq 0$
- $W_{51} + W_{52} + \vartheta_5 - \vartheta_5 \geq 0$
- $W_{51} + W_{53} + W_{54} - \vartheta_5 \geq 0$
- $W_{51} + W_{53} - \vartheta_5 \geq 0$
- $W_{51} + W_{54} - \vartheta_5 \geq 0$
- $W_{52} + W_{53} - \vartheta_5 \geq 0$
- $W_{52} + W_{53} + W_{54} - \vartheta_5 \geq 0$
- $W_{51} + W_{52} + W_{53} + W_{54} \vartheta_5 \geq 0$
- $W_{52} + W_{54} - \vartheta_5 \geq 0$
- $W_{53} + W_{54} - \vartheta_5 \geq 0$
- $W_{51} + W_{52} + \vartheta_5 - \vartheta_5 \geq 0$
- $-\vartheta_5 < 0$

2.2 PART B

In part b of the question 2, we should choose a particular weight vector (including the bias term), and showing that the designed network achieves 100% performance in implementing the desired logic. Hence, my methodology of finding weight and bias parameters is the trial and error by having the knowledge of the above inequalities which means that I selected randomly integers by confirming whether they are valid or not according to the above inequalities so that my randomly selected weights and biases for hidden layer:

$$W_{hidden} = \begin{bmatrix} 2 & 1 & 4 & 3 \\ 0 & -3 & 3 & 3 \\ -3 & 3 & -2 & 0 \\ -2 & 6 & 1 & -3 \end{bmatrix} \begin{bmatrix} 6 \\ 3 \\ 2 \\ 5 \end{bmatrix} \quad where\ corresponding\ bias\ vector \quad \vartheta_{1,2,3,4} = \begin{bmatrix} 6 \\ 3 \\ 2 \\ 5 \end{bmatrix}$$

Here, I concatenating the bias term with the weights just because of the fake of simplicity of the following computations. Then, I implement same methodology for the weights of the output layer so here is the matrix of values with concatenation of bias term:

$$W_{output} = \begin{bmatrix} 2 \\ 3 \\ 4 \\ 7 \\ 1 \end{bmatrix} \quad where\ corresponding\ bias\ vector \quad \vartheta_5 = [1]$$

Hence, we should test the weights of the network whether gives accuracy of 100% or not so I created the input matrix and their testing labels in Python. Here is the Python code for implementation step by step. At the following parts, I explain the applied procedure for part B of the question 2.

2.2.1 Creating Input Matrix and Output Vector

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

```
# 4 input Truth Label :
input_features = np.array([[0,0,0,0],
                           [0,0,0,1],
                           [0,0,1,0],
                           [0,0,1,1],
                           [0,1,0,0],
                           [0,1,0,1],
                           [0,1,1,0],
                           [0,1,1,1],
                           [1,0,0,0],
                           [1,0,0,1],
                           [1,0,1,0],
                           [1,0,1,1],
                           [1,1,0,0],
                           [1,1,0,1],
                           [1,1,1,0],
                           [1,1,1,1]])

# Their correspoding outputs :
target_output = np.repeat(1,16).reshape(16,1)
target_output[0] = 0
```

After importing the basics, I created input samples, it is the 4 input truth table itself. Then, I created the testing labels for the test purposes.

```
print(f" Shape of Input Array: {input_features.shape}")
```

```
 Shape of Input Array: (16, 4)
```

```
print(f" Shape of Output Array: {target_output.shape}")
```

```
 Shape of Output Array: (16, 1)
```

Therefore, we have a (16 x 4) input matrix with (16 x 1) their outcomes/labels.

2.2.2 Creating Feed Forward Neural Network with Single Hidden Layer

According to my assumptions, in the following figure, I created feed forward neural network model, and I initiate the weights and biases of the hidden and output layer.

```
class FeedForwardNN:
    def __init__(self):

        # Introducing predetermined bias terms :

        # Hidden Layers bias term :
        self.bias = np.array([6,3,2,5]).reshape(4,1)
        # Output Layers bias term :
        self.bias2 = np.array([[1]])

        # Introducing predetermined weight terms :

        # Hidden Layers weight :
        self.W1 = np.array([[2,1,4,3],[0,-3,3,3],[-3,3,-2,1],[-2,6,1,-3]]).reshape(4,4)
        # Output Layers weight :
        self.W2 = np.array([2,3,4,7]).reshape(4,1)

        # Output of output layer :
        self.A2 = None

        # To track Mean Squared Error function :
        self.cost = []
```

```
    def step(self,X):
        """
        Given the arrays, return ;
        if {1 , X > 0
           {0 , X <= 0
        """
        return 1 * (X > 0)
```

Then, I created the step function as a activation unit of all neurons as we discuss.

Here is the main algorithm for feed forwarding through input layer to output layer. Firstly, I concatenating the weights and bias term for simplifying the matrix multiplications, then I get the weighted sum of the inputs by dot product the weights matrix and input matrix for the hidden layer. Then, I put this result in our activation function to outputs 0 or 1. After that, I

applied same procedure for the output layer so that the classify the output. Here is the Python implementation of the discussed procedure.

```
def fit(self,X:np.ndarray,Y:np.ndarray) -> None:
    """
    Given the traning dataset and their labels,
    fitting the model, and measure the performance
    by validating training dataset.
    """
    # Concatenating hidden layers weights and bias term :
    W1 = np.concatenate((self.W1,self.bias),axis = 1)

    # Feed Forwarding :
    Z1 = np.dot(X,W1)

    # Output of the activation function : (0/1)
    A1 = self.step(Z1)

    # Same procedure for output layer :
    W2 = np.concatenate((self.W2,self.bias2))
    Z2 = np.dot(A1,W2)
    self.A2 = self.step(Z2)

    # Error = Output of the output layer - Training label
    E = self.A2-Y

    # Mean Squared Error :
    MSE = (1/2) * np.abs(np.power(E,2).sum())
    self.cost.append(MSE)
```

*2.2.3 **Prediction***

Lastly, I need to predict data to do that I created the predict function to test the input samples for the following parts of the question.

```
def predict(self,X):
    """
    Feed forwarding from the input nodes, through the hidden nodes
    and to the output nodes.
    """
    return self.step(np.dot(self.step(np.dot(X,self.W1)),self.W2))
```

*2.2.4 **Evaluation of the Model***

The next step is to evaluate our model by comparing the test labels, here is the implementation of evaluation of the accuracy score.

```
    def evaluate(self,target_features):
        """
        Comparing the target labels by neural network's
        outputs, then returning accuracy score
        """
        acc = (self.A2 == target_features).all().mean()
        print(f"Accuracy of the model is: {int(acc*100)}%")
        return int(acc*100)
```

```
    def display_results(self):
        """
        Plotting and displaying the Mean Sqaured Error

        """
        print(f"MSE loss is : {self.cost[-1]}")
        plt.plot(range(len(np.squeeze(np.array(self.cost)))),np.squeeze(np.array(self.cost)))
        plt.legend([f"MSE:{round(self.cost[-1],4)}"])
        plt.xlabel('# of Iterations')
        plt.ylabel('MSE Loss Function')
        plt.title('Model evaluation')
```

```
model_partA.display_results()
```

```
MSE loss is : 0.0
```

As we expect, our Mean Squared Error is 0. Since, we have 100% accuracy, we have correctly classify the inputs.

```
if __name__ == "__main__":
    model_partA = FeedForwardNN()
    model_partA.fit(input_features,target_output)
    model_partA.evaluate(target_output)
```

```
Accuracy of the model is: 100%
```

Finally, we observe that we reach the accuracy of 100% successfully.

2.3 PART C

In this part, we assume that the input data samples are subject to small random fluctuations due to noise. We are going to answer to the question that will the network you designed in part a whether function robustly under noisy conditions or not. The exact answer is that part a solution is not the most robust solution.

In the previous part, we have a 100% accuracy that means works perfectly but the input to the network does not contain any noise. Therefore, when the input to the network contains noise,

we will see that the model could not give the accuracy of 100% due small fluctuations in the training set. Hence, we should robust our network by redesigning the weights and biases of the network to works under unstable conditions. To do that, I followed two approach; I implement the gradient descent based learning algorithm and find the best possible flexible weights and biases, secondly I solved the inequalities again by selecting the weights 1 or -1 by using the properties of orthogonality and symmetry then I arranged the bias term so that the network is not affected by noise so much. Firstly, I explain the former solution for this question. The mathematical model of the gradient descent will be analyzed in the question 3 so here is the implementation of the algorithm.

2.3.1 Creating Back Propagating Neural Network with Single Hidden Layer

In this part, I initiate the parameters (i.e., weights, bias, learning rate) by standard normal distribution over 0. In the comments of code, I describe the parameters. Again, for the sake of simplicity, I concatenated the weight and bias matrices.

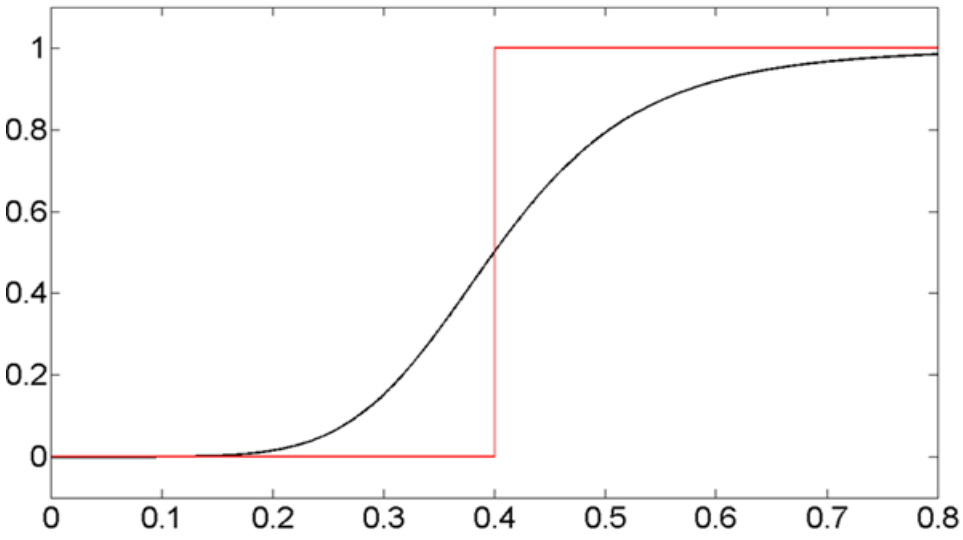
```
class MultiLayerPerceptron:
    def __init__(self):

        # Introducing weight and bias terms,
        # by standart normal distribution over 0

        np.random.seed(42)

        # Hidden Layers bias term :
        self.B1 = np.random.randn(4,1)
        # Output Layers bias term :
        self.B2 = np.array([[1]])

        # Learning rate for gradient descent :
        self.lr = 0.05

        # Hidden Layers weight :
        self.W1 = np.random.randn(4,4) * 0.01
        # Output layers weight
        self.W2 = np.random.randn(4,1) * 0.01

        # To track Mean Squared Error function :
        self.cost = []

        # Concatenating weights and bias terms :
        self.W1 = np.concatenate((self.W1,self.B1),axis = 1)
        self.W2 = np.concatenate((self.W2,self.B2))

        # Output of the output layer
        self.A2 = None
```

2.3.2 Activation of the Neurons

I selected sigmoid function as an activation unit of neural network. Here, main logic is the robust the network, so another activation unit may be selected. Then, I created a function of the derivative of the sigmoid function that is used in the back propagating part.

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

    def sigmoid_der(self,X:np.ndarray):
        """
        The derivative of Sigmoid function for
        gradient descent while backproparagation
        """
        return self.sigmoid(X)*(1-self.sigmoid(X))
```

The reason for the selection of the sigmoid function as activation unit is that it is approximation of the step function, in the following figure, black color represents the sigmoid function, whereas red one represents the step function.



2.3.3 Fitting the Model and Back Propagation

In the following code, I trained the weights and biases of the network over 10 000 epochs with firstly forward propagating then calculating the Mean Squared Function, finally updated the weights and biases according to gradient descent algorithm.

2.3.3.1 Forward Propagating

```
    def fit(self,X:np.ndarray,Y:np.ndarray,iterations:int,verbose:True):
        """
        Given the traning dataset,their labels and number of epochs
        fitting the model, and measure the performance
        by validating training dataset.
        """

        for epoch in range(iterations):

            # Feed forwarding :
            Z1 = np.dot(X,self.W1)

            # Output of the hidden layer :
            A1 = self.sigmoid(Z1)

            # İnput of the output layer :
            Z2 = np.dot(A1,self.W2)
            # Output of the output layer :
            self.A2 = self.sigmoid(Z2)

            # Calculating error
            E = self.A2-Y

            # Mean sqaured error :
            MSE = (1/2) * np.abs(np.power(E,2).sum())
            self.cost.append(MSE)

            if verbose == True:
                if epoch % 500 == 0:
                    print(f"The epoch num is: {epoch} ------> MSE is : {MSE}")
```

2.3.3.2 Backward Propagating

I divided the back propagating algorithm into two parts. The first part is the calculation and derivation of the effect of the weights of the output layer on loss function (i.e., Mean Squared Error).

```
            # Back propagation part, updating weights and bias term
            # according to the gradient descent algorithm, in the first phase
            # I computed effect of the outputs layers weights on the loss function.
            # Them, in the second phase, I computed the effect of the hidden layers
            # weight on the Error function. Finally, update the weights and biases.

#---------------------- First Phase ----------------------------------#
            # The derivative of the Error with respect to the output :
            dE_dA2 = E
            # The derivative of output with respect to input to the output layer :
            dA2_dZ2 = self.sigmoid_der(Z2)
            dE_dZ2 = E * dA2_dZ2

            # The derivative of the input function of output layer with respect to weights of
            # the output layer :
            dZ2_dW2 = A1.T

            # Total derivate :
            dE_dW2 = np.dot(dZ2_dW2,dE_dZ2)
```

The next phase is the calculation and derivation of the effect of the weights of the hidden layer on loss function.

```
#---------------------- Second Phase ----------------------------------#
            # Goal is to find the effect of the hidden layers weight
            # on the output of neural network so we should find

            # The derivate of the Error function with respect to output of the hidden layer
            dE_dA1 = np.dot(dE_dZ2,self.W2.T)

            # The derivative of the outputs of the hidden layer with respect to input to the
            # hidden layer :
            dA1_dZ1 = self.sigmoid_der(Z1)

            # Multiplication of above two :
            dE_dZ1= dE_dA1 * dA1_dZ1

            # The derivative of the input of the hidden layer with respect to the weights of
            # the hidden layer :
            dZ1_dW1 = X.T

            # Total derivative:
            dE_dW1 = np.dot(dZ1_dW1,dE_dZ1)
```

When finding the required derivatives, I updated the weights according to gradient descent rule.

```
            # Updating weights and biases
            self.W1 -= self.lr * dE_dW1
            self.W2 -= self.lr * dE_dW2
```

2.3.4 Model Evaluation

The next step is to evaluate our model by comparing the test labels, here is the implementation of evaluation of the accuracy score or Mean Squared Error (MSE).

```
    def accuracy(self,Y):
        """
        Given the test labels of the training test, comparing and
        returning the accuracy of the model
        """
        acc = (self.A2 == target_features).all().mean()
        print(f"Accuracy of the model is: {int(acc*100)}%")
        return int(acc*100)
```

```
    def evaluate(self):
        """Plotting and displaying the Mean Sqaured Error :
        """
        print(f"MSE loss is : {self.cost[-1]}")
        plt.plot(range(len(np.squeeze(np.array(self.cost)))),np.squeeze(np.array(self.cost)))
        plt.legend([f"MSE:{round(self.cost[-1],4)}"])
        plt.xlabel('# of Iterations')
        plt.ylabel('MSE Loss Function')
        plt.title('Model evaluation')
```

*2.3.5 **Implementation***

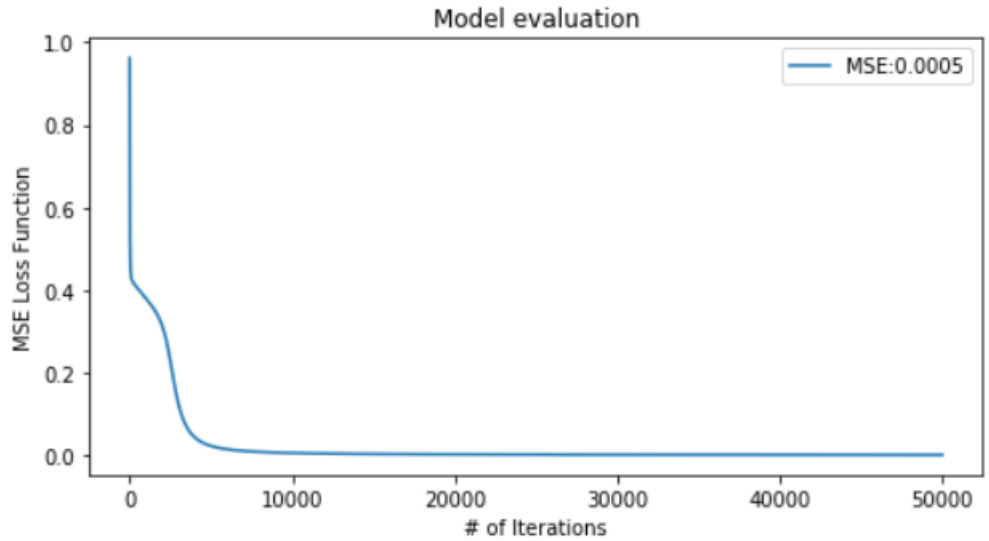
In this part, I created the model for part C, and fitting the model as we discuss.

```
model_partC = MultiLayerPerceptron()
model_partC.fit(X = input_features,Y = target_output,iterations = 50000,verbose = True )
```

```
The epoch num is: 0 ------> MSE is : 0.9619788868874422
The epoch num is: 500 ------> MSE is : 0.40325494024751635
The epoch num is: 1000 ------> MSE is : 0.3795212467800225
The epoch num is: 1500 ------> MSE is : 0.35341330636682233
The epoch num is: 2000 ------> MSE is : 0.31322025970720063
The epoch num is: 2500 ------> MSE is : 0.2280402563837776
The epoch num is: 3000 ------> MSE is : 0.1236457204689303
The epoch num is: 3500 ------> MSE is : 0.06851007874362836
The epoch num is: 4000 ------> MSE is : 0.043034581485915976
The epoch num is: 4500 ------> MSE is : 0.02982426525555424
The epoch num is: 5000 ------> MSE is : 0.022157507630476393
```

```
plt.figure(figsize = (8,4))
model_partC.evaluate()
```

```
MSE loss is : 0.0005240490481591191
```



As we see, our model is more robust to be work under noisy conditions. Final weights of the hidden layer:

```
print(f"Most Robust Weights of Hidden Layer \n {model_partC.W1}")
```

```
Most Robust Weights of Hidden Layer
 [[-1.26240984 -1.46796279 -1.43157073 -1.60343491  2.90054917]
 [-1.26238999 -1.46804527 -1.44813112 -1.61921153  2.85520524]
 [-1.2505314  -1.46779474 -1.44178554 -1.5993274   2.9085472 ]
 [-1.24683938 -1.43399215 -1.41755183 -1.57873098  3.03688246]]
```

Final weights of the Output Layer:

```
print(f"Most Robust Weights of Output Layer \n {model_partC.W2}")
```

```
Most Robust Weights of Output Layer
 [[-3.32662012]
 [-3.86788154]
 [-3.79990092]
 [-4.26749739]
 [ 7.80405911]]
```

We see that our network learns very good this is a logic gate implementation, our inputs to the gate have same importance in digital circuit designs so we can see that our networks weights are nearly same and bias have big role to classify inputs.

*2.3.6 **Prediction***

This section will be used in the following part.

```
    def predict(self,X):
        """
        Feed forwarding from the input nodes, through the hidden nodes
        and to the output nodes.
        """
        return self.sigmoid(np.dot(self.sigmoid(np.dot(X,self.W1)),self.W2))
```

2.3.7 Robust the Network by Conditioning on Weight and Bias Term

In this section, I am going to introduce the 2nd method for powering the network by conditioning weights and bias term. To find the most robust weights, I select the weights as ± 1 to use . The main algorithm for strengthen the network is to give -1 when the input within NOT operator, 1 for input with no NOT operator, and finally 0 for that input does not exist. Then, I put the condition that bias term should be in the middle of the equalities. By doing so, we can robust the network so that the effect of noisy inputs is decreased.

İnequalities of 1st neuron

- $W_{11} + W_{13} - \vartheta_1 < 0$
- $W_{11} + W_{13} + W_{14} - \vartheta_1 \geq 0$

After the weights are given we can easily calculate the range of bias term by solving equations according to ($X_1 X_3 X_4$) term so that

$$\boldsymbol{W_{13} = 1 \; W_{14} = 1 \; W_{11} = 1}$$
$$\boldsymbol{2 < \vartheta_1 \leq 3}$$

İnequalities of 2nd neuron

- $W_{23} + W_{24} - \vartheta_2 \geq 0$
- $W_{22} + W_{23} + W_{24} - \vartheta_2 < 0$

After the weights are given we can easily calculate the range of bias term by solving equations according to ($X_2' X_3 X_4$) term so that

$$\boldsymbol{-W_{22} = W_{23} = W_{24} = 1}$$
$$\boldsymbol{1 < \vartheta_2 \leq 2}$$

İnequalities of 3rd neuron

- $W_{31} - \vartheta_3 < 0$
- $W_{32} - \vartheta_3 \geq 0$

After the weights are given we can easily calculate the range of bias term by solving equations according to ($X_1' X_2 X_3'$) term so that

$$\boldsymbol{W_{31}, W_{33} = -1 \; W_{32} = 1}$$
$$\boldsymbol{0 < \vartheta_3 \leq 1}$$

İnequalities of 4th neuron

- $W_{41}+ W_{42}+ W_{44}- \vartheta_4 < 0$
- $W_{42}- \vartheta_4 \geq 0$

After the weights are given we can easily calculate the range of bias term by solving equations according to $(X_1{}'X_2X_4{}')$ term so that

$$\boldsymbol{W_{41}, W_{44} = -1 \ W_{42}=1}$$
$$\boldsymbol{0 < \vartheta_4 \leq 1}$$

İnequalities of Output neuron

- $W_{51}+ W_{52} + W_{53} + W_{54} - \vartheta_5 \geq 0$
- $-\vartheta_5 < 0$

After the weights are given we can easily calculate the range of bias term by solving equations according to $(X_1X_3X_4) + (X_2{}'X_3X_4) + (X_1{}'X_2X_3{}') + (X_1{}'X_2X_4{}')$ term so that

$$\boldsymbol{W_{51} = W_{52} = W_{53} = W_{54} = 1}$$
$$\boldsymbol{0 < \vartheta_5 \leq 4}$$

Note that interval of bias term of output layer can be vary in $0 < \vartheta_5 \leq$ i for i =1,2,3,4 due to inequalities, wise choice for the bias term is in $0 < \vartheta_5 \leq 1$ but as we discuss I choose the number that is in middle of the interval. Lastly, here I gathered what I did in part C.

$$W = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 0 & -1 & 1 & 1 \\ -1 & 1 & -1 & 0 \\ -1 & 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} 2.5 \\ 1.5 \\ 0.5 \\ 0.5 \end{bmatrix} \quad where\ corresponding\ bias\ vector \quad \vartheta_{1,2,3,4} = \begin{bmatrix} 2.5 \\ 1.5 \\ 0.5 \\ 0.5 \end{bmatrix}$$

Here, I concatenating the bias term with the weights just because of the fake of simplicity. Then, I implement same methodology for the weights of the output layer so here is the matrix of values with concatenation of bias term:

$$W = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 0.5 \end{bmatrix} \textit{where bias vector} \quad \vartheta_5 = [0.5]$$

Finally, note that inequalities can be solve infinitely many ways but here I choose the most rational ones to our network.

2.4 PART D

In this part, we are going to generate 400 input samples by first concatenating 25 samples from each input vector with random noise vector of length 4 for each training sample, assuming a zero-mean Gaussian distribution with a standard deviation of 0.2. Then, we are going to form validation samples for testing the neural network by linearly superposing the input samples and the random noise samples. Finally, I evaluate the classification performance (i.e., percentage correct) of the models designed in parts a and c on the validation samples.

2.4.1 Creating Noisy Input Vector

Firstly, I created stable input vector and noise vector with Gaussian distribution zero mean, 0.2 of standard deviation then I concatenate the input vector and noise vector to form a validation samples.

```
# Creating 400 x 4 zero matrix :
input = np.zeros(1600).reshape(400,4)

# Generating 400 input sample = 16 x 25
for i in range(16):
    for j in range(25):
        input[i*25+j,:] = input_features[i]

# Noice matrix with Gaussion distribution zero mean 0.2 std.
noise = np.random.normal(loc = 0 , scale = 0.2 ,size = (400,4))

# Concetaneting
noisy_samples = input + noise
```

```
print(f" The shape of noisy samples: {noisy_samples.shape}")
```

```
 The shape of noisy samples: (400, 4)
```

2.4.2 Predictions of Model Created in Part A

As we discuss, I implemented a prediction function in the feed forward neural network in the part A, now I simply use this function to get prediction for validation samples.

```
# The predictions on the model created in the Part A :
model_A_pred = model_partA.predict(noisy_samples)
```

```
model_A_pred.shape
```

```
(400, 1)
```

2.4.3 Predictions of Model Created in Part C

As I said, I implemented a prediction function in the back propagating neural network in the part C, now I simply use this function to get prediction for validation samples.

```
# # The predictions on the model created in the Part C :
model_C_pred = model_partC.predict(noisy_samples)
```

```
model_C_pred.shape
```

```
(400, 1)
```

Then, I created test labels for the reason of comparison between predictions and true labels.

```
# Generating testing labels :
test_labels = np.repeat(1,400).reshape(400,1)
test_labels[:25] = 0
```

```
test_labels.shape
```

```
(400, 1)
```

Before the comparison between predictions and test labels, I rounded the result coming from model part C because remember that I used sigmoid function that outputs in the interval [0,1] so I simply round the numbers.

```
# Rounding the predictions of the model created
# in the Part C since sigmoid function returns the
# predictions in the interval [0,1].
model_C_pred = np.round(model_C_pred)
```

2.4.4 Testing Performance

Lastly, I compare the model created in the part A and C by accuracy score that means the rate of total true predictions and the number of the prediction.

```
def performance(X1,X2):
    """
    Given two arrays, prediction matrix and
    test matrix, comparing and returning the number
    of true predictions
    """

    assert(X1.shape == X2.shape)

    true_pred = 0

    for i in range(X1.shape[0]):
        if int(X1[i]) == 1*int(X2[i]):
            true_pred += 1
    return true_pred
```

```
model_A_perf = performance(model_A_pred,test_labels)
model_C_perf = performance(model_C_pred,test_labels)
```

```
print(f"The Accuracy of Model Part A is: {(model_A_perf/test_labels.shape[0]) * 100}% \n")
print(f"The Accuracy of Model Part C is: {(model_C_perf/test_labels.shape[0]) * 100}% ")
```

```
The Accuracy of Model Part A is: 94.0%

The Accuracy of Model Part C is: 97.75%
```

Therefore, we see that the accuracy of the model created in part A gives 94% accuracy that means it have 4* 96 = 376 true predictions whereas the robust model created in the part C by powered gradient descent algorithm gives 97.75% accuracy that means it have 3*97.95 =391 true predictions.

3 QUESTION 3

In the following sections, I am going to explain and discuss the process and results of the question 3.

3.1 PART A

A researcher would like to process images of alphabet letters with a perceptron. A collection of images were compiled for training and testing the perceptron. In the part A, I visualize a sample image for each class. Then, Find correlation coefficients between pairs of sample images.

3.1.1 Loading Dataset

```
import numpy as np
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings("ignore")
import h5py
import seaborn as sns
%matplotlib inline
```

After importing the required libraries, I load the dataset and split into training and testing data and their labels respectively.

```
def load_dataset(path_dataset:str) -> tuple :
    """
    Given the path of the dataset, return
    training and testing images with respective
    labels.
    """

    with h5py.File(path_dataset,'r') as F:
        # Names variable contains the names of training and testing file
        names = list(F.keys())

        X_train = np.array(F[names[2]][:])
        y_train = np.array(F[names[3]][:])
        X_test = np.array(F[names[0]][:])
        y_test = np.array(F[names[1]][:])

    y_train = y_train.reshape(y_train.shape[0],1)
    y_test = y_test.reshape(y_test.shape[0],1)

    return X_train,y_train,X_test,y_test
```

```
X_train,y_train,X_test,y_test = load_dataset('assign1_data1.h5')
```

```
def print_shapes():
    print(f"X_train has a shape  : {X_train.shape} and contains {X_train.shape[0]} training images with 28x28 pixel" )
    print(f"y_train has a shape  : {y_train.shape}")
    print(f"X_test  has a shape  : {X_test.shape}  and contains {X_test.shape[0]} training images with 28x28 pixel" )
    print(f"y_test  has a shape  : {y_test.shape}")
print_shapes()
```

```
X_train has a shape  : (5200, 28, 28) and contains 5200 training images with 28x28 pixel
y_train has a shape  : (5200, 1)
X_test  has a shape  : (1300, 28, 28)  and contains 1300 training images with 28x28 pixel
y_test  has a shape  : (1300, 1)
```

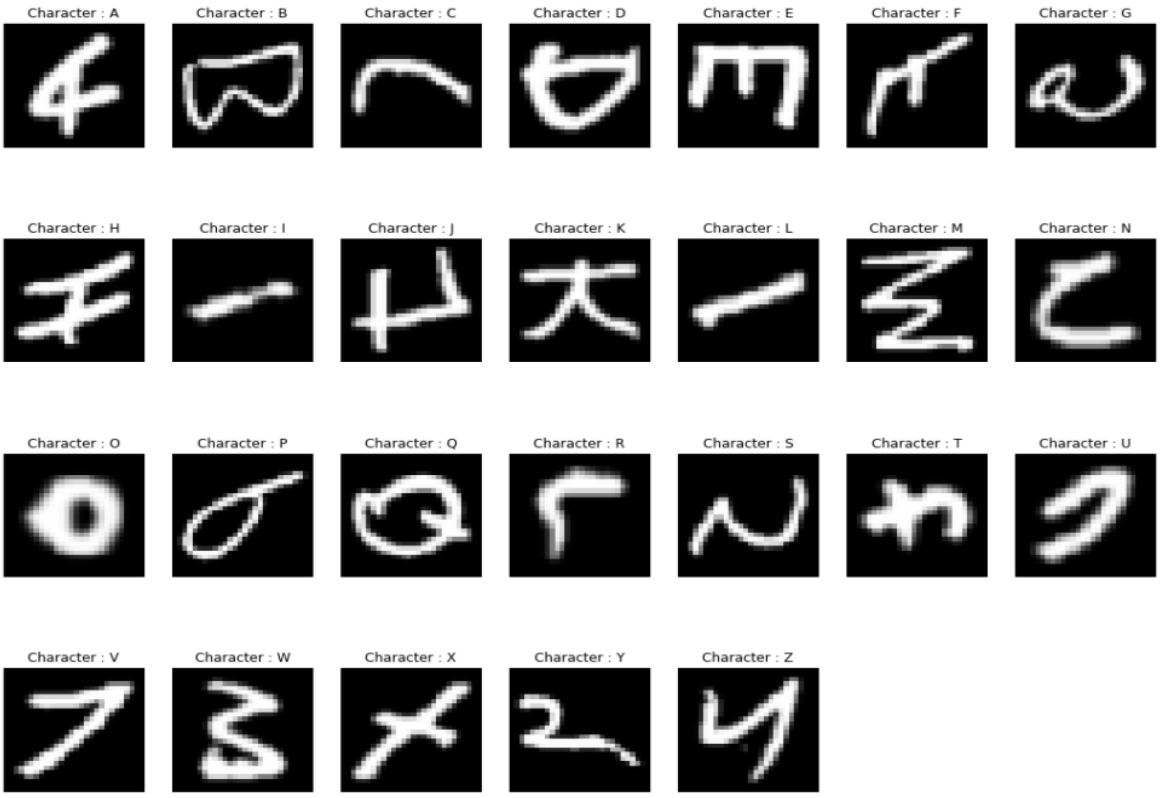
Therefore, we have 5200 training images with 28x28 pixel with 1 color channel, and 1300 testing images with same pixels. Then, I visualize each sample class in the data.

3.1.2 Visualization of the Letters

```
def unique_images(X: np.ndarray,Y:np.ndarray) -> list:
    """
    Given traning and testing images,
    return  images of unique alphabet letters
    """
    size = X.shape[0]
    letter = 1
    uniqe_letter = []
    unique_image = None
    uniqe_index = []

    for i in range(size):
        if letter == int(Y[i]):
            unique_image = X[i]
            uniqe_letter.append(unique_image)
            uniqe_index.append(i)
            letter += 1

    return uniqe_letter,uniqe_index
```

```
images,indexes = unique_images(X_test,y_test)
```

```
alphabet = ['A', 'B', 'C', 'D', 'E', 'F', 'G',
       'H', 'I', 'J', 'K', 'L', 'M', 'N',
       'O', 'P', 'Q', 'R', 'S', 'T', 'U',
       'V', 'W', 'X', 'Y', 'Z']

plt.figure(figsize=(18,15))

for letter,image in enumerate(images):
    plt.subplot(4,7,letter+1)
    plt.imshow(image,cmap='gray')
    plt.axis('off')
    plt.title(f"Character : {alphabet[letter]}")
```



So there we see that we have a 26 class with corresponding alphabet letters. Then, we are realized that in the images there are upper case and lower case letter. Moreover, each letter exists of different version that means images can be reversed, flipped that is data augmentation methods to generalize the model further.

3.1.3 Correlation Coefficient Matrix

In this section, our aim is to find correlation coefficient matrix between two samples in the dataset. The correlational coefficient matrix can be defined as a following way:

$$P_{X,Y} = \frac{cov(X,Y)}{\sigma_X \sigma_Y}$$

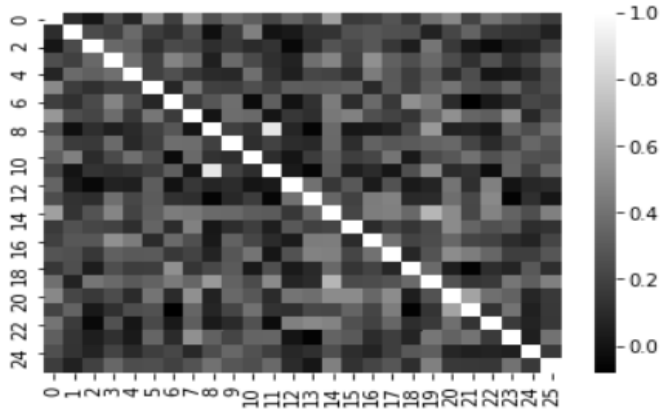
Pearson's correlation coefficient is covariance of the random variables divided by their multiplication of standard deviations where cov is the covariance, σ_X is the standard deviation of the random variable X, σ_Y is the standard deviation of the random variable Y. The formula can be seen as a:

$$P_{X,Y} = \frac{E[(X-\mu_X)(Y-\mu_Y)]}{\sigma_X \sigma_Y}$$

Where E is the expectations, μ_X and μ_Y are the mean of X and Y, respectively. Here is the implementation of Pearson's correlation matrix in Python.

```
def corr(X,nclass,pos):
    mat = np.zeros(nclass*nclass).reshape(nclass,nclass)
    x = X.T
    for i in range(nclass):
        for j in range(nclass):
            mat[i,j] = np.corrcoef(x[:,:,pos[i]].flat, x[:,:,pos[j]].flat)[0,1]
    return mat
```

```
df_correlation_same = pd.DataFrame(corr(X_test,nclass=26,pos = indexes))
sns.heatmap(df_correlation, annot = False , cmap = 'gray')
plt.show()
```



In this figure, I try to explore the correlation between the same sample image. As expected, the diagonal part of the matrix is showing a correlation 1 (white area). This expectation can be proved as a:

$$P_{X,Y} = \frac{cov(X,Y)}{\sigma_X \sigma_Y} = \frac{cov(X,X)}{\sigma_X \sigma_X} = \frac{\sigma_X \sigma_X}{\sigma_X \sigma_X} = 1$$

Therefore, we can say that within-class variability lower than across-class variability.

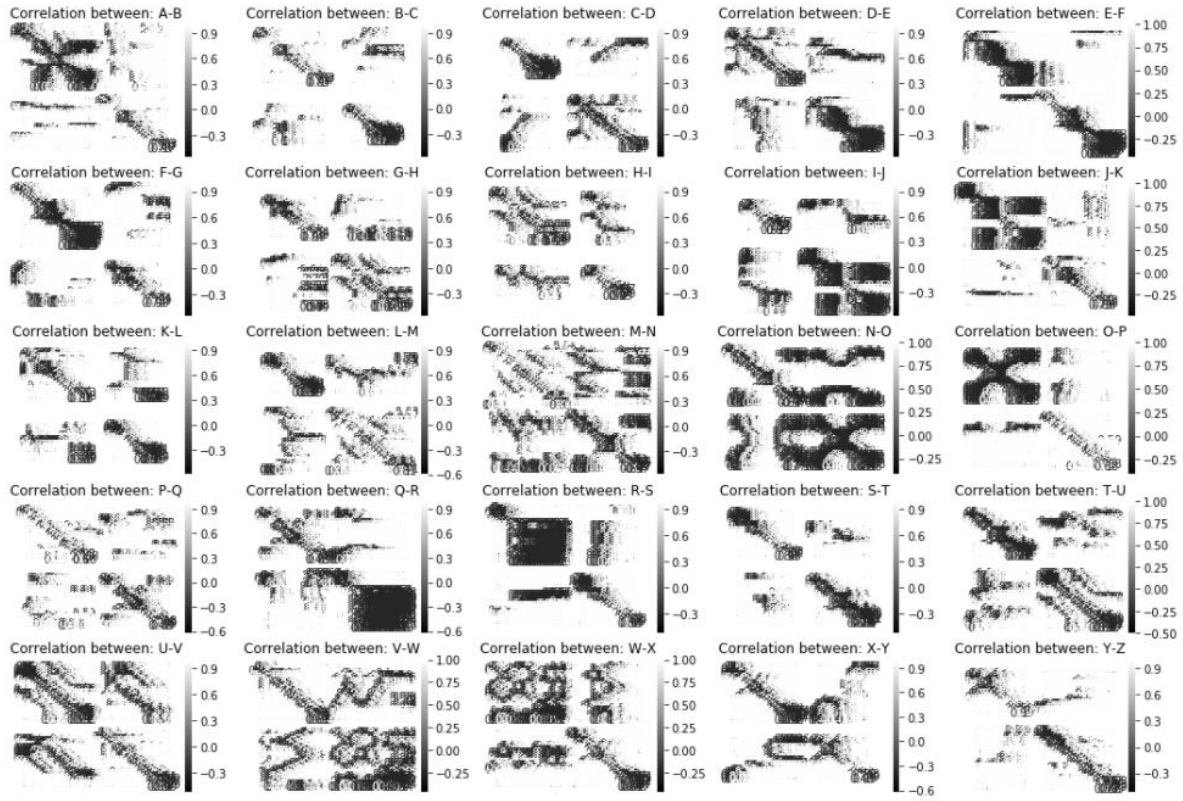
Next, we see that there are darker places that corresponds a lower correlation between the images so that across class variability is higher. The middle of the white and black points, (i.e., gray points) represents the not quite high correlation but also not quite low correlations so one can say that there are some medium correlations.

After that, I compared the different letter samples to see the correlation between them, and dive into within class variability more. Here is the Python implementation:

```python
def correlation_matrix(X1,X2 : np.ndarray):
    """
    Given arrays of two images, returning
    correlation coefficient matrix
    """
    x1,x2 = flatten_images(X1,X2)

    cor = np.corrcoef(x1,x2)

    return cor
```

```python
plt.figure(figsize=(18,15))

for letter,image in enumerate(images):
    if letter == 25:
        break
    plt.subplot(6,5,letter+1)
    corr = correlation_matrix(images[letter],images[letter+1])
    sns.heatmap(corr, annot = True, cmap = 'gray')
    plt.axis('off')
    plt.title(f"Correlation between: {alphabet[letter]}-{alphabet[letter+1]}")
```

The assignment required the displaying of the correlation coefficient matrix from sample images. Hence, here are the visualizations.



Here we see that some letters have higher correlation between them, as well as some are not so much. But the visualization of the correlation, give huge understanding of the correlation. Here, the diagonal is not much whiter that yields within class variability is higher with respect to what we did last time.

3.2 PART B

3.2.1 Data Preprocessing

In the following section, basic image preprocessing techniques will be discussed to feed the network in proper way.

3.2.1.1 One Hot Encode

In this part, I designed a perceptron model that consist of 28*28 (flatten images) input neurons with 26(number of class) neurons in the output layer so that each class corresponds a unique letter in the alphabet. To do so, first step is the one hot encode the training labels so that we can convert number of classes with number of indexes in the (number of sample x number of class) matrix that means I converted the labels with 5200 x 26 matrix where each row is the sample image and columns represents the corresponding classes.

```
def one_hot_encoder(Y):
    classes = int(np.max(Y))
    dim = Y.shape[0]
    val = np.zeros(classes*dim).reshape(dim,classes)

    for num in range(dim):
        val[num,int(Y[num])-1] = 1

    return val
```

```
y_train = one_hot_encoder(y_train)
y_test = one_hot_encoder(y_test)
```

```
print_shapes()
```

```
X_train has a shape  : (5200, 28, 28) and contains 5200 training images with 28x28 pixel
y_train has a shape  : (5200, 26)
X_test  has a shape  : (1300, 28, 28)  and contains 1300 training images with 28x28 pixel
y_test  has a shape  : (1300, 26)
```

3.2.1.2 Flattening Images

Therefore, our labels now 0 or 1. Then, the next step is to implement flattening images so that they can be treated in neural network easily. The process involves the multiplying the pixels itself 28*28 = 784 then images are ready to input to the network.

```
def flatten_images(X1,X2):
    X1 = X1.reshape(X1.shape[0],-1)
    X2 = X2.reshape(X2.shape[0],-1)

    return X1,X2
```

```
X_train,X_test = flatten_images(X_train,X_test)
```

3.2.1.3 *Standardization the data*

Before the training, data preprocessing is important so that the most known preprocessing for images is to normalize them because they have a range between [0,255]. The main task is to scale them between [0,1] by dividing its maximum value.

```
def normalize(X):
    return X/255
```

```
X_train = normalize(X_train)
X_test = normalize(X_test)
```

3.2.2 *Single Layer perceptron*

Then, we can feed images into network to do that I initiate the class that takes only learning rate. I introduce bias and weight of the perception model according to zero mean Gaussian distribution with 0.01 standard deviation.

```
class SingleLayerNN():
    def __init__(self, learning_rate):

        # Introducing bias and weight terms with  Gaussian distribution N(0; 0:01)
        self.bias = np.random.normal(loc = 0 ,scale = 0.01 ,size = (1,26))
        self.W = np.random.normal(loc = 0 ,scale = 0.01 ,size =(784,26))

        # Learning rate :
        self.lr = learning_rate

        # To keep track loss function :
        self.cost = []
```

For the activation unit of the net, I selected sigmoid function as proposed. I and initiate the derivative of the sigmoid function to be use later in the back propagation.

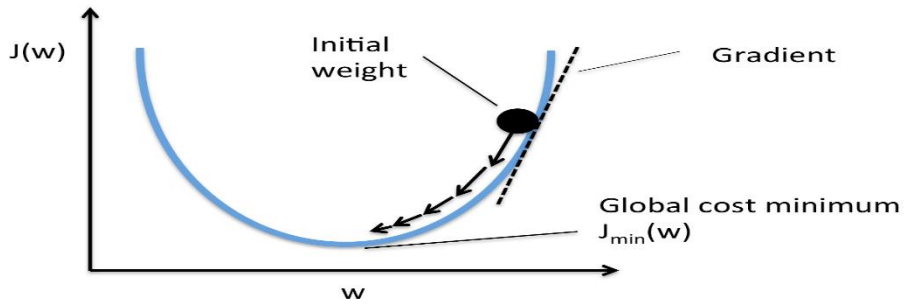
```
    def sigmoid(self,X):
        return 1/(1+np.exp(-X))

    def sigmoid_der(self,X):
        return self.sigmoid(X)*(1-self.sigmoid(X))
```

Most crucial part is to fitting part because it arranges the weights and biases what we call learning. Here, I feed randomly selected training images into the perceptron model and calculate the loss function. In our case, I selected mean squared function as proposed.

3.2.3 Back Propagation

After forwarding the input to the output neurons, I calculate the loss. After that, we should implement back propagation to learn best possible weights and biases so I use the gradient descent algorithm. In the gradient descent algorithm, we are trying to update weights so that the derivative of the cost function (i.e., mean squared error) reaches to global minimum. So here is the mathematical model of the gradient descent:

Learning rate is a parameter of the gradient descent that optimizes the derivative step that can be visualize as a:



To reach the global minimum in the cost function, learning rate should not be much higher or lower because if it is substantially low, it can be too slow to reach global minimum or if much higher than the optimal it can be miss the global minimum because the step size of the derivative is increases.

Weights are updated until convergence according the derivative of the loss function with respect to weights that means that effect of the weights on the loss function.

$$W := W\text{-}learning\ rate * \frac{d}{dW} f(W)$$

The derivative of the loss function can be calculated by chain rule because the network consists of cascade systems:

$$\frac{\partial Error}{\partial W} = \frac{\partial Error}{\partial Out} \frac{\partial Out}{\partial in} \frac{\partial in}{\partial W}$$

We can treat each one separately for convenience, so first derivate can be calculated easily by:

First Derivative:

$$\frac{\partial Error}{\partial Out} = \frac{\partial}{\partial Out}\left(\frac{1}{m} * \sum_{i=1}^{m}(target - output)^2\right)$$

$$\frac{\partial Error}{\partial Out} = output - target$$

Second Derivative:

$$\frac{\partial Out}{\partial in} = sigmoid(\alpha) * (1 - sigmoid(\alpha))$$

Third Derivative

$$\frac{\partial in}{\partial W} = \frac{\partial(\sum_{i=1}^{m}(W^T X + \vartheta))}{\partial W} = input\ values$$

Combining Together

$$W = W - learning\ rate * \frac{\partial Error}{\partial W}$$

Here is the implementation of feed forwarding the input and calculating the loss on Python:

```
def fit(self,X,Y,iterations,verbose = True):
    """
    Given the training images and their corresponding
    labels, fitting the model by :
    1) Forwarding the input to the output layer
    2) Calculate the loss
    3) Back propagation
    4) Update the weights and bias according to the
    gradient descent rule

    """
    # As proposed , 10000 epoch :
    for iter in range(iterations):

        # Randomly selected numbers between 0 and 5200
        random_index = np.random.randint(0,X.shape[0])

        # Indexing randomly for training image
        X_temp = X[random_index].reshape(1,X.shape[1])

        # Indexing randomly for training image's label
        y_temp = Y[random_index].reshape(1,Y.shape[1])

        # Feed forwarding input
        Z = np.dot(X_temp,self.W) + self.bias

        # Passing activation unit
        A = self.sigmoid(Z)


        # Calculating the error
        error = A - y_temp

        # Loss function : Mean Squared Error
        MSE = np.square(error).mean()

        # Adding losses to visualize
        self.cost.append(MSE)
```

Here is the implementation of backpropagation in Python:

```
            # The derivative of Error with respect to the output
            derror_douto = error

            # The derivative of output with respect to the input
            douto_dino = self.sigmoid_der(Z)


            deriv = derror_douto * douto_dino

            # The derivative of input with respect to the weight , then multiply all
            deriv_final = np.dot(X_temp.T,deriv)

            # Updated weights and bias term according to the gradient descent

            self.W -= self.lr * deriv_final
            self.bias -= self.lr * deriv
```

Therefore, the weights and bias term are updated according to the gradient descent update rule.

3.2.4 Tuning the learning rate

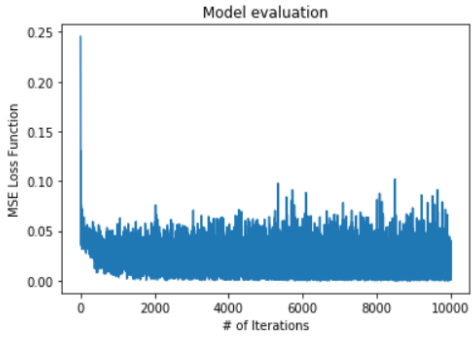
Here is implementation of a grid search algorithm that takes training data, test data and some sort of learning rates, fitting the model then calculating the loss. According to loss, it returns the best possible learning rate.

```
def grid_search(X_train,Y_train,X_test,y_test,learning_rates):
    acc = []
    for lr in learning_rates:
        temp_model = SingleLayerNN(lr)
        temp_model.fit(X_train,y_train,10000,False)
        pred = temp_model.predict(X_test)
        acc.append(accuracy(y_test,pred))

    val = learning_rates[np.argmax(acc)]

    return val,SingleLayerNN(val)

# Trying the learning rate for = 0.1,0.2,...,0.9
learning_rates = [i*0.01 for i in range(1,10)]

best_lr, best_model = grid_search(X_train,y_train,X_test,y_test,learning_rates)
print(f" Optimal learning rate is {best_lr}")
best_model.fit(X_train,y_train,10000,False)
# Predictions
best_predictions = best_model.predict(X_test)


# Accuracy Score :
accuracy(y_test,best_predictions)
```

```
 Optimal learning rate is 0.06

60.38461538461538
```

Note that, learning rate can be changed by %3-5 because of the random processes in the learning algorithm. This method finds the best possible learning rate when random distribution. So the best possible accuracy is ~ **60%** with **0.06** learning rate. Evaluation of the model shown below:

```
def evaluate(self):
    """
    Evaluation of the model by visualization of MSE
    """
    print(f"MSE loss is : {self.cost[-1]}")
    plt.plot(range(len(self.cost)),self.cost)
    plt.xlabel('# of Iterations')
    plt.ylabel('MSE Loss Function')
    plt.title('Model evaluation')
```



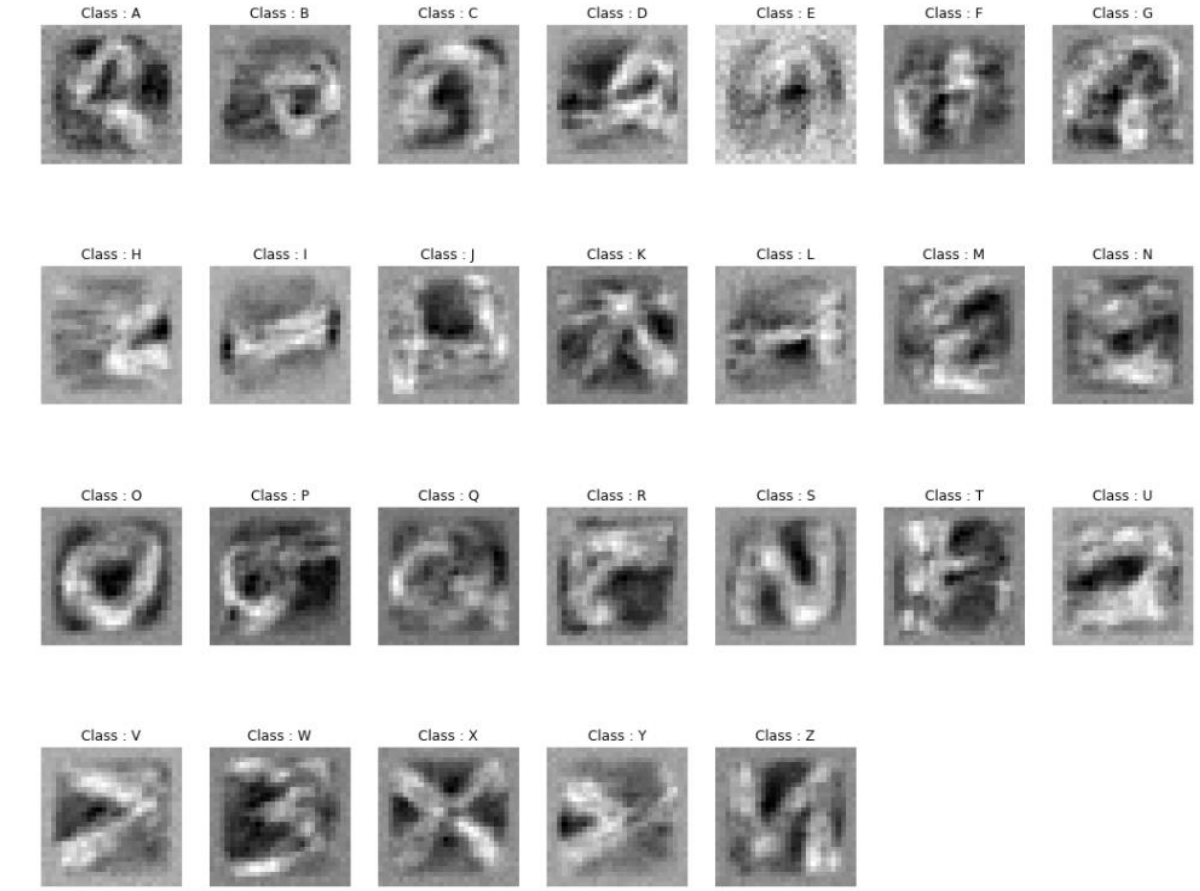
Since we implement step-wise (On-line) learning algorithm, variations in the training process is quite normal.

3.2.5 Displaying Final Networks Weight

Then, assignment requires displaying the weights, the figure is shown below:

```
def display_weight(self):
    """
    Visualization of the final weight elements
    """

    plt.figure(figsize=(18,15))
    for i in range(26):
        plt.subplot(4,7,i+1)
        plt.imshow(self.W.T[i,:].reshape(28,28),cmap='gray')
        plt.axis('off')
        plt.title(f"Class : {alphabet[i]}")
```

```
model.display_weight()
```



So we can easily see that our neural network learns from the training images. The reason behind this is to weights of the model mimics the training data. They are not hundred percent right due to certain reason like there are upper cases, lower cases, flipping letter, skewed letters and shifted letter that result in generalization of the model but decreased the learning accuracy so one can say that there is trade-off between them. Note that further trials of the tuning the learning rate can give higher accuracy scores such as 61% with learning rate 0.054. But, if one wants higher, should optimize the network by further optimization techniques.

```
model = SingleLayerNN(0.054)
model.fit(X_train,y_train,10000,False)
accuracy(y_test,model.predict(X_test))
```

```
61.0
```

Lastly, I created the prediction method for future use but here is the implementation that takes an input and feeding forward.

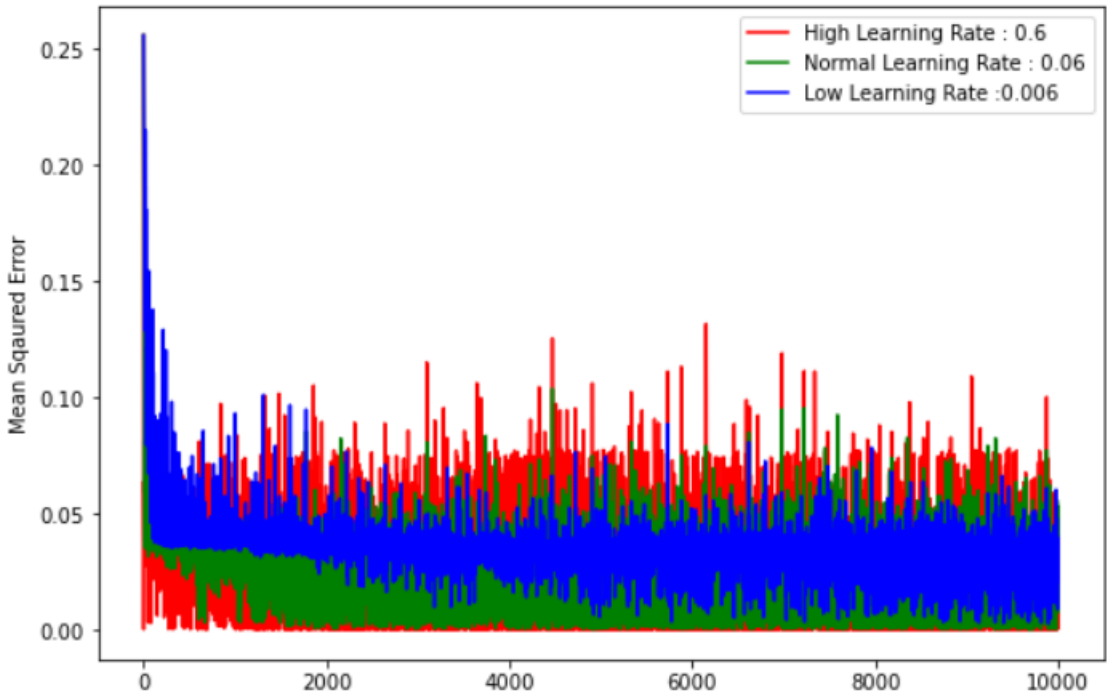
```
    def predict(self,X):
        """
        Feed forwarding to test new images and classification
        """
        return self.sigmoid(np.dot(X,self.W)+self.bias)
```

3.3 PART C

In this part of the question, I tried 3 learning rates that is 10 times of the optimal one, optimal one and %10 percent of the optimal one. Here is the Python code for implementation:

```
# substantially higher and a subtantially lower learning rate
# so multiplying learning rate by 10 and divided by 10, respectively


learningRate = [best_lr * i for i in [10,1,0.1]]
MeanSqrtError = []

for i in learningRate:
    model_temp = SingleLayerNN(learning_rate = i)
    model_temp.fit(X_train,y_train,iterations = 10000, verbose = False)
    MeanSqrtError.append(model_temp.cost)
```

```
plt.figure(figsize=(9,6))
plt.plot(MeanSqrtError[0], color = 'r')
plt.plot(MeanSqrtError[1],color = 'g')
plt.plot(MeanSqrtError[2],color = 'b')
plt.ylabel('Mean Sqaured Error')
plt.legend([f'High Learning Rate : {learningRate[0]}',f'Normal Learning Rate : {learningRate[1]}',
            f'Low Learning Rate :{learningRate[2]}'])
plt.show()
```



As we expect, with the high learning rate – the red one – is converged local minimum too fast and does not change with the number of iteration that result in not convergence of the gradient so lead to decrease the performance of the model. Moreover, with the lower learning rate, gradient is too slow find the global minimum that is also unwanted. The reason behind is that gradient's step size is smaller so that results in the problems of convergence.

3.4 PART D

The final part is the evaluation part, here I tested the model initiated by different learning rates as we discuss in part C. Previously, I defined the predict function so here we are just call the pre-defined function to predict classes.

```
# I created 3 model with different learning rates as proposed,
# from higher to lower.
model_1 = SingleLayerNN(learningRate[0])
model_2 = SingleLayerNN(learningRate[1])
model_3 = SingleLayerNN(learningRate[2])

# Fitting the model
model_1.fit(X_train,y_train,10000,False)
model_2.fit(X_train,y_train,10000,False)
model_3.fit(X_train,y_train,10000,False)
```

```
# Predictions :
y_pred_1 = model_1.predict(X_test)
y_pred_2 = model_2.predict(X_test)
y_pred_3 = model_3.predict(X_test)
```

Lastly, I evaluate the model by comparing their accuracy score that can be calculated by:

$$\text{Accuracy score} \triangleq \frac{\#\ of\ true\ predictions}{\#\ of\ predictions} * 100$$

Therefore, here is the Python implementation:

```
def accuracy(Y1,Y2):
    """
    Given one prediction and their test label
    counting how many correct outputs do model have
    then dividing it by the total number to give
    accuracy score
    """
    assert(Y1.shape == Y2.shape)
    size = Y1.shape[0]
    count = 0
    for i in range(size):
        if np.argmax(Y1[i]) == np.argmax(Y2[i]):
            count += 1
    return (count/size)*100
```

```
print(f" Model 1 Accuracy : {accuracy(y_pred_1,y_test)}")
print(f" Model 2 Accuracy : {accuracy(y_pred_2,y_test)}")
print(f" Model 3 Accuracy : {accuracy(y_pred_3,y_test)}")
```

```
Model 1 Accuracy : 52.46153846153846
Model 2 Accuracy : 59.692307692307686
Model 3 Accuracy : 34.15384615384615
```

As we expect, non-optimal learning rates gives worse result due to the problems with convergence of the global minimum in gradient descent. Here we see that optimization of the network by fast learning rate while stochastic gradient descent may or may not converge the expected minimum value of error, in this case our model with high learning rate gives ~ 52% accuracy. The reason behind that 0.6 is not a huge learning, so it is quite normal see not worst results. Our best model, gives ~ 60% since we know that with optimal learning rate, we can tend to global minimum of the loss function, without decaying learning gradually in the iterations. Finally, model 3 with low learning rate gives the worst error rate, low accuracy that is too slow to converge the global minimum. With the increment of iteration, model 3 may converge the global minimum. However, this is not the case with 10 000 iterations.

4 QUESTION 4

In this part of the assignment, I analyze the *two_layer_net.*ipynb Python code for implementation of two layer fully connected neural network, here are the parts notebook involved.

4.1 IMPLEMENTING NEURAL NETWORK

In this part, a small net is created by input size 4, neuron size 10, number of classes 3 and number of inputs. Input and output matrices is given randomly with 5x4 input size, (5,) of output size that is a broadcast version of (5x1) matrix. Then, model is created with specifying its neural architecture.

The code is shown below:

Implementing a Neural Network

In this exercise we will develop a neural network with fully-connected layers to perform classification, and test it out on the CIFAR-10 dataset.

In [1]:

```
# A bit of setup
from __future__ import print_function
import numpy as np
import matplotlib.pyplot as plt

from cs231n.classifiers.neural_net import TwoLayerNet


%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2

def rel_error(x, y):
    """ returns relative error """
    return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))
```

In [ ]:

```

```

We will use the class `TwoLayerNet` in the file `cs231n/classifiers/neural_net.py` to represent instances of our network. The network parameters are stored in the instance variable `self.params` where keys are string parameter names and values are numpy arrays. Below, we initialize toy data and a toy model that we will use to develop your implementation.

In [2]:

```
# Create a small net and some toy data to check your implementations.
# Note that we set the random seed for repeatable experiments.

input_size = 4
hidden_size = 10
num_classes = 3
num_inputs = 5

def init_toy_model():
    np.random.seed(0)
    return TwoLayerNet(input_size, hidden_size, num_classes, std=1e-1)

def init_toy_data():
    np.random.seed(1)
    X = 10 * np.random.randn(num_inputs, input_size)
    y = np.array([0, 1, 2, 2, 1])
    return X, y

net = init_toy_model()
X, y = init_toy_data()
```

Therefore, model is created by TwoLayerNet class with mentioned architecture.

4.2 FORWARD PASS: COMPUTE SCORES

In this part, we are going to take the data and weights and compute the class scores, the loss, and the gradients on the parameters, finally compare our scores and expected score that should be lower than 10^{-7}. Here the following figure shows both our scores and correct scores.

Forward pass: compute scores

Open the file `cs231n/classifiers/neural_net.py` and look at the method `TwoLayerNet.loss`. This function is very similar to the loss functions you have written for the SVM and Softmax exercises: It takes the data and weights and computes the class scores, the loss, and the gradients on the parameters.

Implement the first part of the forward pass which uses the weights and biases to compute the scores for all inputs.

In [4]:

```
scores = net.loss(X)
print('Your scores:')
print(scores)
print()
print('correct scores:')
correct_scores = np.asarray([
  [-0.81233741, -1.27654624, -0.70335995],
  [-0.17129677, -1.18803311, -0.47310444],
  [-0.51590475, -1.01354314, -0.8504215 ],
  [-0.15419291, -0.48629638, -0.52901952],
  [-0.00618733, -0.12435261, -0.15226949]])
print(correct_scores)
print()

# The difference should be very small. We get < 1e-7
print('Difference between your scores and correct scores:')
print(np.sum(np.abs(scores - correct_scores)))
```

```
Your scores:
[[-0.81233741 -1.27654624 -0.70335995]
 [-0.17129677 -1.18803311 -0.47310444]
 [-0.51590475 -1.01354314 -0.8504215 ]
 [-0.15419291 -0.48629638 -0.52901952]
 [-0.00618733 -0.12435261 -0.15226949]]

correct scores:
[[-0.81233741 -1.27654624 -0.70335995]
 [-0.17129677 -1.18803311 -0.47310444]
 [-0.51590475 -1.01354314 -0.8504215 ]
 [-0.15419291 -0.48629638 -0.52901952]
 [-0.00618733 -0.12435261 -0.15226949]]

Difference between your scores and correct scores:
3.6802720496109664e-08
```

As we see that, difference between scores is less than 10^{-7} as expected.

4.3 FORWARD PASS: COMPUTE LOSS

In this part, the regularization loss will be examining, since we add the regularization term to a loss function, we expect that the loss should be lower than what we get before. As I expected, the difference between losses are:

Forward pass: compute loss

In the same function, implement the second part that computes the data and regularizaion loss.

In [5]:

```
loss, _ = net.loss(X, y, reg=0.05)
correct_loss = 1.30378789133

# should be very small, we get < 1e-12
print('Difference between your loss and correct loss:')
print(np.sum(np.abs(loss - correct_loss)))
```

```
Difference between your loss and correct loss:
1.7985612998927536e-13
```

4.4 BACKWARD PASS

We will be examining the effects of the weights and biases to the loss function in this section that will be form a basis for gradient descent algorithm.

Backward pass

Implement the rest of the function. This will compute the gradient of the loss with respect to the variables `W1`, `b1`, `W2`, and `b2`. Now that you (hopefully!) have a correctly implemented forward pass, you can debug your backward pass using a numeric gradient check:

In [6]:

```
from cs231n.gradient_check import eval_numerical_gradient

# Use numeric gradient checking to check your implementation of the backward pass.
# If your implementation is correct, the difference between the numeric and
# analytic gradients should be less than 1e-8 for each of W1, W2, b1, and b2.

loss, grads = net.loss(X, y, reg=0.05)

# these should all be less than 1e-8 or so
for param_name in grads:
    f = lambda W: net.loss(X, y, reg=0.05)[0]
    param_grad_num = eval_numerical_gradient(f, net.params[param_name], verbose=False)
    print('%s max relative error: %e' % (param_name, rel_error(param_grad_num, grads[pa
ram_name])))
```

```
W1 max relative error: 3.561318e-09
b1 max relative error: 1.555470e-09
W2 max relative error: 3.440708e-09
b2 max relative error: 3.865091e-11
```

```
W1 max relative error: 3.561318e-09
b1 max relative error: 1.555470e-09
W2 max relative error: 3.440708e-09
b2 max relative error: 3.865091e-11
```

Relative error means the partial derivative with respect to the loss function(one can call gradients), so here is the result as shown in above.

4.5 *TRAIN THE NETWORK*

This is the training part of process so we take the training and testing data, also using them for validating the training process with Stochastic Gradient Descent algorithm. To feed SGD, learning rate specified as a 0.1 with regularization parameter β (or lambda is used also) $5x10^{-6}$ on 100 epoch. Here are the results of the training:

Train the network

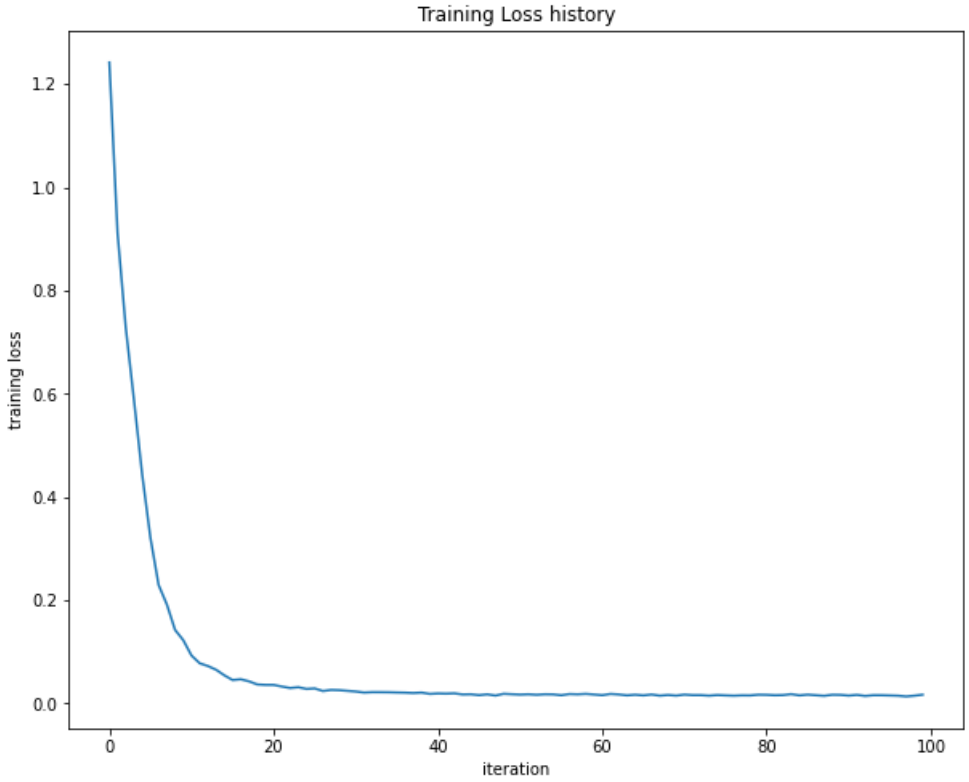
To train the network we will use stochastic gradient descent (SGD), similar to the SVM and Softmax classifiers. Look at the function `TwoLayerNet.train` and fill in the missing sections to implement the training procedure. This should be very similar to the training procedure you used for the SVM and Softmax classifiers. You will also have to implement `TwoLayerNet.predict`, as the training process periodically performs prediction to keep track of accuracy over time while the network trains.

Once you have implemented the method, run the code below to train a two-layer network on toy data. You should achieve a training loss less than 0.2.

In [7]:

```
net = init_toy_model()
stats = net.train(X, y, X, y,
            learning_rate=1e-1, reg=5e-6,
            num_iters=100, verbose=False)

print('Final training loss: ', stats['loss_history'][-1])

# plot the loss history
plt.plot(stats['loss_history'])
plt.xlabel('iteration')
plt.ylabel('training loss')
plt.title('Training Loss history')
plt.show()
```

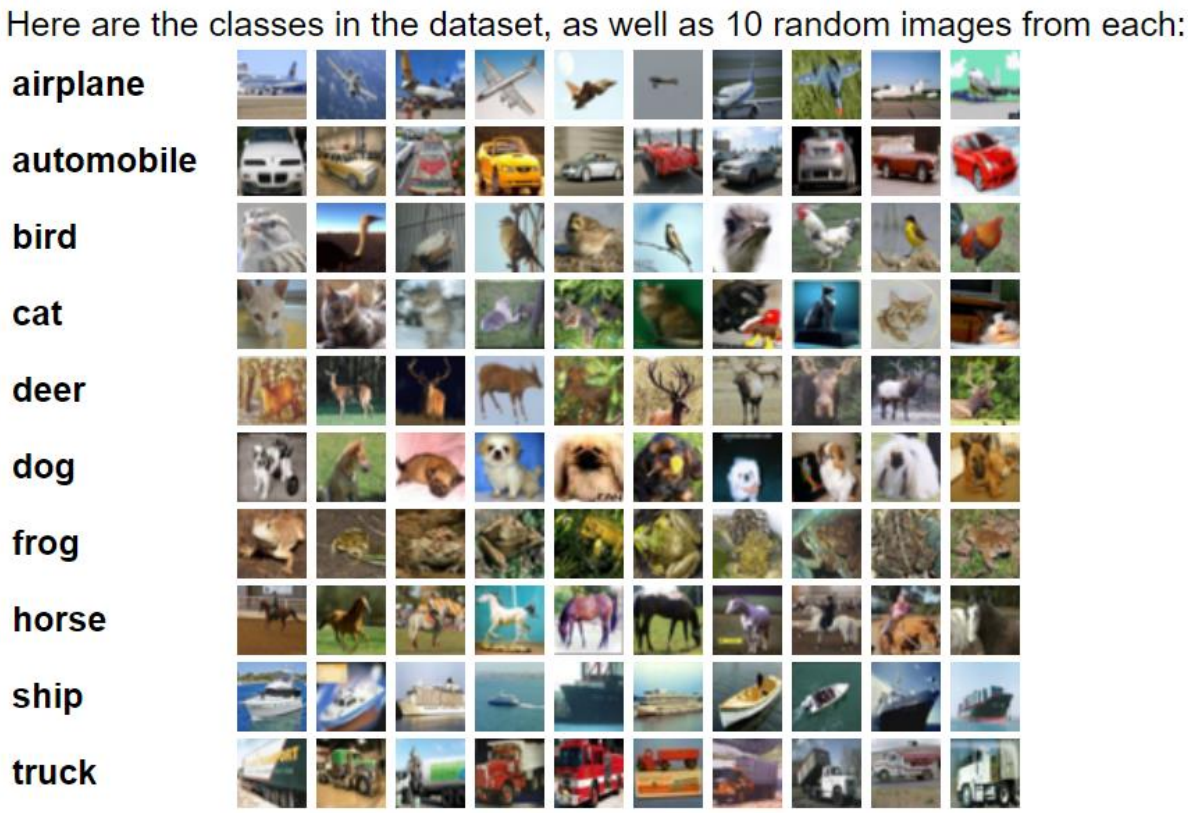
```
Final training loss:  0.017149607938732093
```



This is expected since the training loss should decrease with the number of iterations until a certain point so we see that loss is decreasing gradually. This means that our learning rate and regularization term is optimal.

4.6 CIFAR-10 DATASET

In this part of the question, we will work on well-known CIFAR-10 dataset. CIFAR-10 dataset consist of 32x32 images with 3 color channel RGB. In our case, we have a 49 000 training images and 1000 testing images. So the dataset is split into 98% of the dataset is training data and 2% of the dataset is testing data that will cause problems as we will see.



4.6.1 Load Data

In this part, we extract the data from the website https://www.cs.toronto.edu/~kriz/cifar.html that is official page of CIFAR-1xx datasets. Then, we split into training and testing data with corresponding labels. After that, we normalize the images by subtracting the mean of the image itself. The final step for preprocessing is the flattening the images since we have a 32x32 pixel with RGB color channels, we should create a vector per sample by forming pixels and color channel so that we have a 32*32*3 features that implies that we have 3072 dimension in the data. In the following figure, we see that the final version of the training and testing data consist of 49 000 sample with 3072 features and 1000 sample with 3072 features, respectively.

In [13]:

```
from cs231n.data_utils import load_CIFAR10

def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000):
    """
    Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
    it for the two-layer neural net classifier. These are the same steps as
    we used for the SVM, but condensed to a single function.
    """
    # Load the raw CIFAR-10 data
    cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'

    X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

    # Subsample the data
    mask = list(range(num_training, num_training + num_validation))
    X_val = X_train[mask]
    y_val = y_train[mask]
    mask = list(range(num_training))
    X_train = X_train[mask]
    y_train = y_train[mask]
    mask = list(range(num_test))
    X_test = X_test[mask]
    y_test = y_test[mask]

    # Normalize the data: subtract the mean image
    mean_image = np.mean(X_train, axis=0)
    X_train -= mean_image
    X_val -= mean_image
    X_test -= mean_image

    # Reshape data to rows
    X_train = X_train.reshape(num_training, -1)
    X_val = X_val.reshape(num_validation, -1)
    X_test = X_test.reshape(num_test, -1)

    return X_train, y_train, X_val, y_val, X_test, y_test


# Cleaning up variables to prevent loading data multiple times (which may cause memory
 issue)
try:
    del X_train, y_train
    del X_test, y_test
    print('Clear previously loaded data.')
except:
   pass

# Invoke the above function to get our data.
X_train, y_train, X_val, y_val, X_test, y_test = get_CIFAR10_data()
print('Train data shape: ', X_train.shape)
print('Train labels shape: ', y_train.shape)
print('Validation data shape: ', X_val.shape)
print('Validation labels shape: ', y_val.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
```

```
Train data shape:  (49000, 3072)
Train labels shape:  (49000,)
Validation data shape:  (1000, 3072)
Validation labels shape:  (1000,)
Test data shape:  (1000, 3072)
Test labels shape:  (1000,)
```

Therefore, in these piece of code, we retrieve the data from the official website, then apply basic image preprocessing techniques to feed the network by normalizing the data by subtracting the mean of the images and flattening the images (i.e., gathering feature space)

4.6.2 Train a Network

We have finalized the preprocessing part so that we can feed them into network with optimization algorithm Stochastic Gradient Descent with decaying learning rate. The reason behind this is to not miss the global optimum in the loss function when we take derivatives with respect to the weights. When we are closer to the global minimum, we are decreasing the step size of the gradient so that we increase the likelihood of catch the global minimum. Next, we add 50 neurons to the hidden layer. When training, we set epoch to 1000, with batch size 200 and initialized the learning rate with 0.0001 and we multiply the learning rate with 0.95 in every update of the weights and biases. Finally, we set regularization parameter β = 0.25. The training process is shown below:

Train a network

To train our network we will use SGD. In addition, we will adjust the learning rate with an exponential learning rate schedule as optimization proceeds; after each epoch, we will reduce the learning rate by multiplying it by a decay rate.

In [14]:

```
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
net = TwoLayerNet(input_size, hidden_size, num_classes)

# Train the network
stats = net.train(X_train, y_train, X_val, y_val,
            num_iters=1000, batch_size=200,
            learning_rate=1e-4, learning_rate_decay=0.95,
            reg=0.25, verbose=True)

# Predict on the validation set
val_acc = (net.predict(X_val) == y_val).mean()
print('Validation accuracy: ', val_acc)
```

```
iteration 0 / 1000: loss 2.302978
iteration 100 / 1000: loss 2.302503
iteration 200 / 1000: loss 2.298749
iteration 300 / 1000: loss 2.272868
iteration 400 / 1000: loss 2.242846
iteration 500 / 1000: loss 2.117056
iteration 600 / 1000: loss 2.109445
iteration 700 / 1000: loss 2.038701
iteration 800 / 1000: loss 1.986945
iteration 900 / 1000: loss 1.984841
Validation accuracy:  0.283
```

We see that our validation accuracy is too low. To explore what happened in the training process I analyze the loss function and accuracy together in the following part.

4.6.3 Debug the Training

Here we see that the progress loss function with the number of iterations, loss is decreasing more or less in the training process but this is not the want we want. We want that loss function should decrease exponentially.

Debug the training

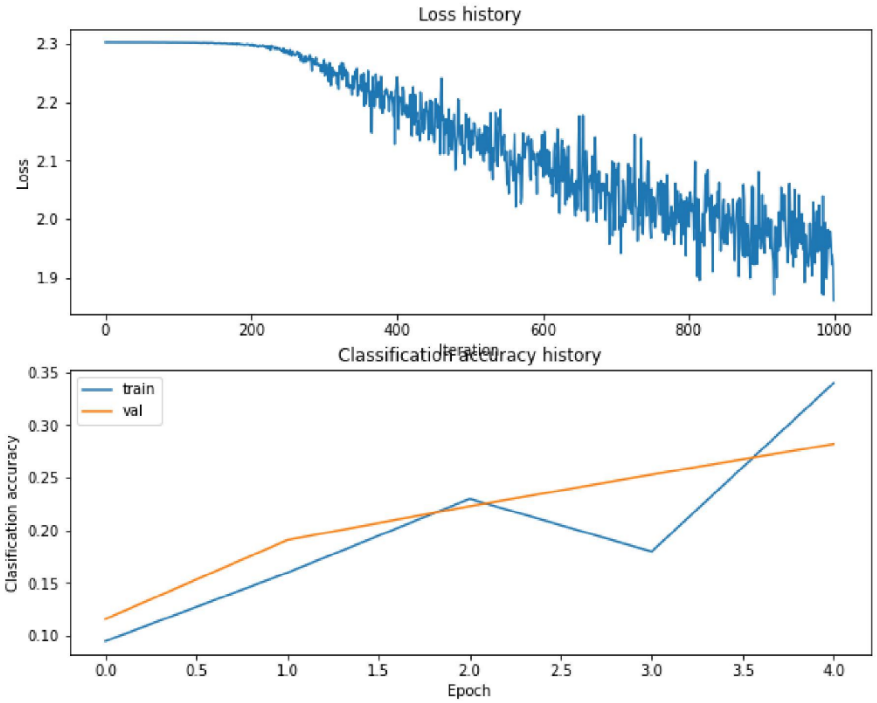
With the default parameters we provided above, you should get a validation accuracy of about 0.29 on the validation set. This isn't very good.

One strategy for getting insight into what's wrong is to plot the loss function and the accuracies on the training and validation sets during optimization.

Another strategy is to visualize the weights that were learned in the first layer of the network. In most neural networks trained on visual data, the first layer weights typically show some visible structure when visualized.

In [15]:

```
# Plot the loss function and train / validation accuracies
plt.subplot(2, 1, 1)
plt.plot(stats['loss_history'])
plt.title('Loss history')
plt.xlabel('Iteration')
plt.ylabel('Loss')

plt.subplot(2, 1, 2)
plt.plot(stats['train_acc_history'], label='train')
plt.plot(stats['val_acc_history'], label='val')
plt.title('Classification accuracy history')
plt.xlabel('Epoch')
plt.ylabel('Clasification accuracy')
plt.legend()
plt.show()
```

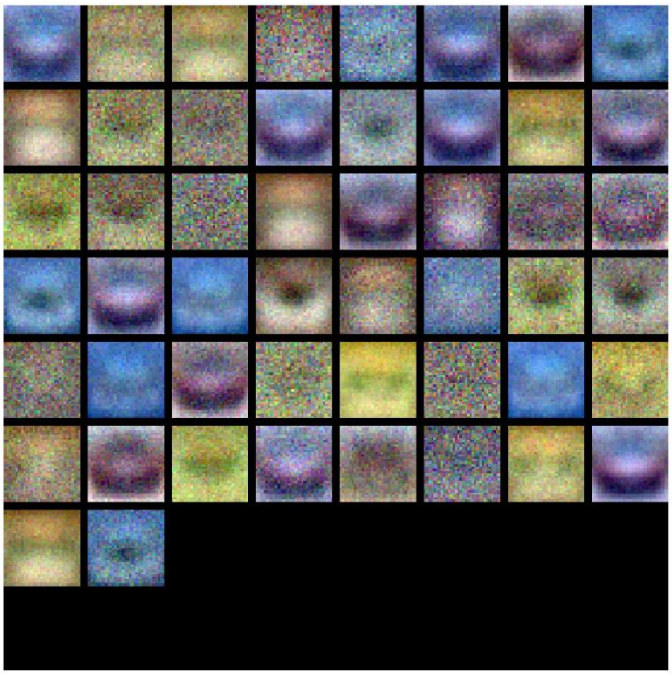


We see that validation accuracy is following a training accuracy until a certain point that the point we should since after that point, we are more likely to face with overfitting of the model. So we see that our model is facing high variance with low bias what we call overfitting.

Another way to track the process and debug it is to visualization of the weights since they reflect the learned features of the data. Here is the visualization of the learned features of the hidden layer's weights

In [16]:

```
from cs231n.vis_utils import visualize_grid

# Visualize the weights of the network

def show_net_weights(net):
    W1 = net.params['W1']
    W1 = W1.reshape(32, 32, 3, -1).transpose(3, 0, 1, 2)
    plt.imshow(visualize_grid(W1, padding=3).astype('uint8'))
    plt.gca().axis('off')
    plt.show()

show_net_weights(net)
```



4.6.4 *Tuning the Hyperparamaters*

Tune your hyperparameters

What's wrong?. Looking at the visualizations above, we see that the loss is decreasing more or less linearly, which seems to suggest that the learning rate may be too low. Moreover, there is no gap between the training and validation accuracy, suggesting that the model we used has low capacity, and that we should increase its size. On the other hand, with a very large model we would expect to see more overfitting, which would manifest itself as a very large gap between the training and validation accuracy.

Tuning. Tuning the hyperparameters and developing intuition for how they affect the final performance is a large part of using Neural Networks, so we want you to get a lot of practice. Below, you should experiment with different values of the various hyperparameters, including hidden layer size, learning rate, numer of training epochs, and regularization strength. You might also consider tuning the learning rate decay, but you should be able to get good performance using the default value.

Approximate results. You should be aim to achieve a classification accuracy of greater than 48% on the validation set. Our best network gets over 52% on the validation set.

Experiment: You goal in this exercise is to get as good of a result on CIFAR-10 as you can, with a fully-connected Neural Network. Feel free implement your own techniques (e.g. PCA to reduce dimensionality, or adding dropout, or adding features to the solver, etc.).

In this part, we are tuning the Hyperparamaters such as including hidden layer size, learning rate, number of training epochs, and regularization strength. Grid search algorithm is implemented to find the best possible Hyperparamaters. So main algorithm behind the finding optimal solutions to the problem is to try and record the process and keep the model that yields best accuracy and low error. Here are the results:

In [18]:

```
best_net = None # store the best model into this

################################################################################
# TODO: Tune hyperparameters using the validation set. Store your best trained  #
# model in best_net.                                                            #
#                                                                               #
# To help debug your network, it may help to use visualizations similar to the  #
# ones we used above; these visualizations will have significant qualitative    #
# differences from the ones we saw above for the poorly tuned network.          #
#                                                                               #
# Tweaking hyperparameters by hand can be fun, but you might find it useful to  #
# write code to sweep through possible combinations of hyperparameters          #
# automatically like we did on the previous exercises.                          #
################################################################################

input_size = X_train.shape[1]
hidden_size = 100
output_size = 10

# learning_rates = [1, 1e-1, 1e-2, 1e-3]
# regularization_strengths = [1e-6, 1e-5, 1e-4, 1e-3]

# Magic lrs and regs?
# Just copy from: https://github.com/lightaime/cs231n/blob/master/assignment1/two_layer
_net.ipynb
# :(
learning_rates = np.array([0.7, 0.8, 0.9, 1, 1.1])*1e-3
regularization_strengths = [0.75, 1, 1.25]

best_val = -1

for lr in learning_rates:
    for reg in regularization_strengths:
        net = TwoLayerNet(input_size, hidden_size, output_size)
        net.train(X_train, y_train, X_val, y_val, learning_rate=lr, reg=reg,
                  num_iters=1500)

        y_val_pred = net.predict(X_val)
        val_acc = np.mean(y_val_pred == y_val)

        print('lr: %f, reg: %f, val_acc: %f' % (lr, reg, val_acc))

        if val_acc > best_val:
            best_val = val_acc
            best_net = net

print('Best validation accuracy: %f' % best_val)
################################################################################
#                              END OF YOUR CODE                                 #
################################################################################
```

Here we see that best possible validation accuracy is 50.5% and the weights can be visualized as a:

```
lr: 0.000700, reg: 0.750000, val_acc: 0.484000
lr: 0.000700, reg: 1.000000, val_acc: 0.469000
lr: 0.000700, reg: 1.250000, val_acc: 0.471000
lr: 0.000800, reg: 0.750000, val_acc: 0.478000
lr: 0.000800, reg: 1.000000, val_acc: 0.463000
lr: 0.000800, reg: 1.250000, val_acc: 0.477000
lr: 0.000900, reg: 0.750000, val_acc: 0.487000
lr: 0.000900, reg: 1.000000, val_acc: 0.481000
lr: 0.000900, reg: 1.250000, val_acc: 0.475000
lr: 0.001000, reg: 0.750000, val_acc: 0.465000
lr: 0.001000, reg: 1.000000, val_acc: 0.484000
lr: 0.001000, reg: 1.250000, val_acc: 0.458000
lr: 0.001100, reg: 0.750000, val_acc: 0.505000
lr: 0.001100, reg: 1.000000, val_acc: 0.481000
lr: 0.001100, reg: 1.250000, val_acc: 0.477000
Best validation accuracy: 0.505000
```

In [16]:

```
# visualize the weights of the best network
show_net_weights(best_net)
```

Here we see that best possible validation accuracy is 50.5% and the weights can be visualized as shown above. Hence, it is easy to see that model is learnt features more with respect to previous model since as we discuss the weights are tending to mimic the features of the dataset.

4.6.5 *Running on the test set*

Finally, we use the testing images and their corresponding labels to test our model with the data that is unseen by the model.

Run on the test set

When you are done experimenting, you should evaluate your final trained network on the test set; you should get above 48%.

In [20]:

```
test_acc = (best_net.predict(X_test) == y_test).mean()
print('Test accuracy: ', test_acc)
```

```
Test accuracy:  0.482
```

Inline Question

Now that you have trained a Neural Network classifier, you may find that your testing accuracy is much lower than the training accuracy. In what ways can we decrease this gap? Select all that apply.

1. Train on a larger dataset.
2. Add more hidden units.
3. Increase the regularization strength.
4. None of the above.

Your answer:

Your explanation:

So this is the results of final model that gives 48.2% testing accuracy.

4.7 *INLINE QUESTIONS*

In this part of the question, we face to overfitting of the model that is caused by high variance so that testing accuracy is much lower than the training accuracy. The question asks 'In what ways can we decrease this gap?'.

4.7.1 Train on a larger dataset

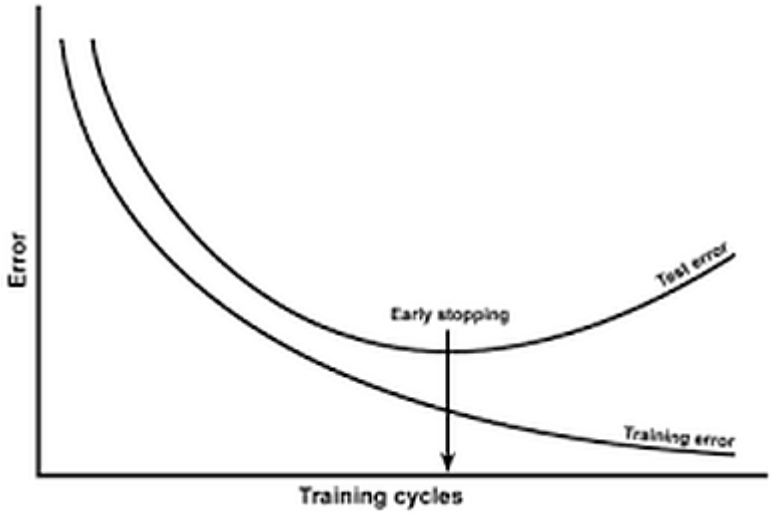
Train on a larger dataset can increase the model performance at a certain point of accuracy, but when the data increases the model should be enough complex to handle large data, i.e., adding hidden layers, neurons, to do that one of the best algorithm is data augmentation that is implementing flipping, translating, rotating, changing its brightness, scaling, adding noises (i.e., blurring, Gaussian filtering ext.) to the existing images in the dataset so that the size of both training and testing data are increased. Therefore, increasing the size of dataset may be better, but the model should be enough complex to learn and generalize properly. In our case, our validation accuracy is around 50%, so increasing the size of dataset may help model to learn better.

4.7.2 Add more hidden units

In our case, validation accuracy is low so that we increased the size of hidden units to handle large data, from 50 hidden units to 100 units, and see that model is performed better. The reason behind this is that we have a large dataset but don't have proper model to learn from it. To learn better, we increased the model complexity by adding neurons to the hidden layers. Adding layers can be also optimal way to get better performance. On the other hand, in general, when we encounter that testing accuracy is much lower than the training accuracy, one can say that model is overfitting so we need the simplify the model. Simplifying the model can consist of several parts but removing layers or neurons in the layer, can help the model to prevent overfitting. Therefore, adding more neurons to the layer may or may not be the good solutions for solving the overfitting model, it depends on the size of dataset, training and testing split of the dataset and much more criteria's.

4.7.3 Increase the regularization strength

Increasing the regularization strength is a powerful way of avoiding overfitting and getting better performance for training accuracy. We can increase the regularization strength by several methods but I am going to talk about four of them which are early stopping ,L_1, L_2 and adding dropout regularization techniques. Early stopping technique provides guidance the model by finding the best iterations number or epoch before the model starts overfitting. In the figure, we see that, at a certain point, we should stop the training because the loss is increasing gradually so early stopping regularization technique can help to prevent the model from overfitting.



Next, we can use the $L_1$and L_2 regularization techniques to reduce the complexity of the model by the following mathematical expression :

$$L(x, y) = \sum_n (y^n - h(x^n, W)^2) + \beta \sum_i W_i^2$$

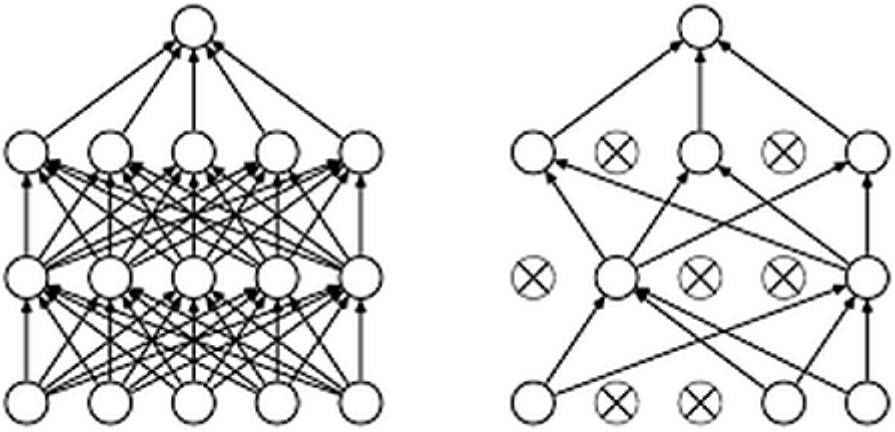
The term $\beta \sum_i W_i^2$ is a regularization term that is adding penalties to the cost function as we show in the above formula. This is the L_2 regularization technique. Then, here the L_1 approach:

$$L(x, y) = \sum_n (y^n - h(x^n, W)^2) + \beta \sum_i |W_i|$$

The difference between them can be sequenced by:

L_2 regularization penalizes the squared sums of the weights wherase L_1 sum of absolute values. L_1 can be more stable and robust the outliers in the data but L_2 not.

Lastly, adding dropouts is also powerful regularization technique that is arranging loss function by randomly drops the hidden units in the layers of the network while training process. Adding dropouts can be visualized by the following figure:



(a) Standard Neural Net (b) After applying dropout.

5 APPENDIX

In this part, code is provided by labeled question numbers.

5.1 QUESTION 2

5.1.1 Part B

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

# 4 input Truth Label :
input_features = np.array([[0,0,0,0],
                           [0,0,0,1],
                           [0,0,1,0],
                           [0,0,1,1],
                           [0,1,0,0],
                           [0,1,0,1],
                           [0,1,1,0],
                           [0,1,1,1],
                           [1,0,0,0],
                           [1,0,0,1],
                           [1,0,1,0],
                           [1,0,1,1],
                           [1,1,0,0],
                           [1,1,0,1],
                           [1,1,1,0],
                           [1,1,1,1]])

# Their correspoding outputs :
target_output = np.repeat(1,16).reshape(16,1)
target_output[0] = 0
```

```
print(f" Shape of Input Array: {input_features.shape}")

print(f" Shape of Output Array: {target_output.shape}")
```

```
class FeedForwardNN:
    def __init__(self):

        # Introducing predetermined bias terms :

        # Hidden Layers bias term :
        self.bias = np.array([6,3,2,5]).reshape(4,1)
        # Output Layers bias term :
        self.bias2 = np.array([[1]])

        # Introducing predetermined weight terms :

        # Hidden Layers weight :
        self.W1 = np.array([[2,1,4,3],[0,-3,3,3],[-3,3,-2,1],[-2,6,1,-3]]).
reshape(4,4)
        # Output Layers weight :
        self.W2 = np.array([2,3,4,7]).reshape(4,1)

        # Output of output layer :
        self.A2 = None

        # To track Mean Squared Error function :
        self.cost = []


    def step(self,X):
        """
        Given the arrays, return ;
        if {1 , X > 0
           {0 , X <= 0
        """
        return 1 * (X > 0)

    def fit(self,X:np.ndarray,Y:np.ndarray) -> None:
        """
        Given the traning dataset and their labels,
        fitting the model, and measure the performance
        by validating training dataset.
        """
        # Concatenating hidden layers weights and bias term :
        W1 = np.concatenate((self.W1,self.bias),axis = 1)

        # Feed Forwarding :
        Z1 = np.dot(X,W1)
```

```
        # Output of the activation function : (0/1)
        A1 = self.step(Z1)

        # Same procedure for output layer :
        W2 = np.concatenate((self.W2,self.bias2))
        Z2 = np.dot(A1,W2)
        self.A2 = self.step(Z2)

        # Error = Output of the output layer - Training label
        E = self.A2-Y

        # Mean Squared Error :
        MSE = (1/2) * np.abs(np.power(E,2).sum())
        self.cost.append(MSE)

    def evaluate(self,target_features):
        """
        Comparing the target labels by neural network's
        outputs, then returning accuracy score
        """
        acc = (self.A2 == target_features).all().mean()
        print(f"Accuracy of the model is: {int(acc*100)}%")
        return int(acc*100)

    def predict(self,X):
        """
        Feed forwarding from the input nodes, through the hidden nodes
        and to the output nodes.
        """
        return self.step(np.dot(self.step(np.dot(X,self.W1)),self.W2))

    def display_results(self):
        """
        Plotting and displaying the Mean Sqaured Error

        """
        print(f"MSE loss is : {self.cost[-1]}")
        plt.plot(range(len(np.squeeze(np.array(self.cost)))),np.squeeze(np.
array(self.cost)))
        plt.legend([f"MSE:{round(self.cost[-1],4)}"])
        plt.xlabel('# of Iterations')
        plt.ylabel('MSE Loss Function')
        plt.title('Model evaluation')



model_partA = FeedForwardNN()
model_partA.fit(input_features,target_output)
model_partA.evaluate(target_output)
```

5.1.2 Part C

```
class MultiLayerPerceptron:
    def __init__(self):

        # Introducing weight and bias terms,
        # by standart normal distribution over 0

        np.random.seed(42)

        # Hidden Layers bias term :
        self.B1 = np.random.randn(4,1)
        # Output Layers bias term :
        self.B2 = np.array([[1]])

        # Learning rate for gradient descent :
        self.lr = 0.05

        # Hidden Layers weight :
        self.W1 = np.random.randn(4,4) * 0.01
        # Output layers weight
        self.W2 = np.random.randn(4,1) * 0.01

        # To track Mean Squared Error function :
        self.cost = []

        # Concatenating weights and bias terms :
        self.W1 = np.concatenate((self.W1,self.B1),axis = 1)
        self.W2 = np.concatenate((self.W2,self.B2))

        # Output of the output layer
        self.A2 = None

    def sigmoid(self,X:np.ndarray):
        return 1/(1+np.exp(-X))

    def sigmoid_der(self,X:np.ndarray):
        """
        The derivative of Sigmoid function for
        gradient descent while backproparagation
        """
        return self.sigmoid(X)*(1-self.sigmoid(X))


    def fit(self,X:np.ndarray,Y:np.ndarray,iterations:int,verbose:True):
        """
        Given the traning dataset,their labels and number of epochs
        fitting the model, and measure the performance
        by validating training dataset.
        """

        for epoch in range(iterations):

            # Feed forwarding :
```

```
            Z1 = np.dot(X,self.W1)

            # Output of the hidden layer :
            A1 = self.sigmoid(Z1)

            # İnput of the output layer :
            Z2 = np.dot(A1,self.W2)
            # Output of the output layer :
            self.A2 = self.sigmoid(Z2)

            # Calculating error
            E = self.A2-Y

            # Mean sqaured error :
            MSE = (1/2) * np.abs(np.power(E,2).sum())
            self.cost.append(MSE)

            if verbose == True:
                if epoch % 500 == 0:
                    print(f"The epoch num is: {epoch} ------> MSE is : {MSE
}")

            # Back propagation part, updating weights and bias term
            # according to the gradient descent algorithm, in the first pha
se
            # I computed effect of the outputs layers weights on the loss f
unction.
            # Them, in the second phase, I computed the effect of the hidde
n layers
            # weight on the Error function. Finally, update the weights and
 biases.

#---------------------- First Phase ----------------------------------#
            # The derivative of the Error with respect to the output :
            dE_dA2 = E
            # The derivative of output with respect to input to the output
layer :
            dA2_dZ2 = self.sigmoid_der(Z2)
            dE_dZ2 = E * dA2_dZ2

            # The derivative of the input function of output layer with res
pect to weights of
            # the output layer :
            dZ2_dW2 = A1.T

            # Total derivate :
            dE_dW2 = np.dot(dZ2_dW2,dE_dZ2)

#---------------------- Second Phase ----------------------------------#

            # Goal is to find the effect of the hidden layers weight
            # on the output of neural network so we should find
```

```
            # The derivate of the Error function with respect to output of
the hidden layer
            dE_dA1 = np.dot(dE_dZ2,self.W2.T)

            # The derivative of the outputs of the hidden layer with respec
t to input to the
            # hidden layer :
            dA1_dZ1 = self.sigmoid_der(Z1)

            # Multiplication of above two :
            dE_dZ1= dE_dA1 * dA1_dZ1

            # The derivative of the input of the hidden layer with respect
to the weights of
            # the hidden layer :
            dZ1_dW1 = X.T

            # Total derivative:
            dE_dW1 = np.dot(dZ1_dW1,dE_dZ1)

            # Updating weights and biases
            self.W1 -= self.lr * dE_dW1
            self.W2 -= self.lr * dE_dW2


    def accuracy(self,Y):
        """
        Given the test labels of the training test, comparing and
        returning the accuracy of the model
        """
        acc = (self.A2 == target_features).all().mean()
        print(f"Accuracy of the model is: {int(acc*100)}%")
        return int(acc*100)

    def predict(self,X):
        """
        Feed forwarding from the input nodes, through the hidden nodes
        and to the output nodes.
        """
        return self.sigmoid(np.dot(self.sigmoid(np.dot(X,self.W1)),self.W2)
)

    def evaluate(self):
        """Plotting and displaying the Mean Sqaured Error :
        """
        print(f"MSE loss is : {self.cost[-1]}")
        plt.plot(range(len(np.squeeze(np.array(self.cost)))),np.squeeze(np.
array(self.cost)))
        plt.legend([f"MSE:{round(self.cost[-1],4)}"])
        plt.xlabel('# of Iterations')
        plt.ylabel('MSE Loss Function')
        plt.title('Model evaluation')
```

```
model_partC = MultiLayerPerceptron()
model_partC.fit(X = input_features,Y = target_output,iterations = 50000,ver
bose = True )

plt.figure(figsize = (8,4))
model_partC.evaluate()

print(f"Most Robust Weights of Hidden Layer \n {model_partC.W1}")

print(f"Most Robust Weights of Output Layer \n {model_partC.W2}")
```

5.1.3 Part D

```
# Creating 400 x 4 zero matrix :
input = np.zeros(1600).reshape(400,4)

# Generating 400 input sample = 16 x 25
for i in range(16):
    for j in range(25):
        input[i*25+j,:] = input_features[i]

# Noice matrix with Gaussion distribution zero mean 0.2 std.
noise = np.random.normal(loc = 0 , scale = 0.2 ,size = (400,4))

# Concetaneting
noisy_samples = input + noise

print(f" The shape of noisy samples: {noisy_samples.shape}")

# The predictions on the model created in the Part A :
model_A_pred = model_partA.predict(noisy_samples)


# # The predictions on the model created in the Part C :
model_C_pred = model_partC.predict(noisy_samples)


# Generating testing labels :
test_labels = np.repeat(1,400).reshape(400,1)
test_labels[:25] = 0


# Rounding the predictions of the model created
# in the Part C since sigmoid function returns the
# predictions in the interval [0,1].
model_C_pred = np.round(model_C_pred)

def performance(X1,X2):
    """
    Given two arrays, prediction matrix and
    test matrix, comparing and returning the number
    of true predictions
    """
```

```
    assert(X1.shape == X2.shape)

    true_pred = 0

    for i in range(X1.shape[0]):
        if int(X1[i]) == 1*int(X2[i]):
            true_pred += 1
    return true_pred

model_A_perf = performance(model_A_pred,test_labels)
model_C_perf = performance(model_C_pred,test_labels)

print(f"The Accuracy of Model Part A is: {(model_A_perf/test_labels.shape[0
]) * 100}% \n")
print(f"The Accuracy of Model Part C is: {(model_C_perf/test_labels.shape[0
]) * 100}% ")
```

5.2 QUESTION 3

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import h5py
import seaborn as sns
```

5.2.1 Part A

```
def load_dataset(path_dataset:str) -> tuple :
    """
    Given the path of the dataset, return
    training and testing images with respective
    labels.
    """

    with h5py.File(path_dataset,'r') as F:
        # Names variable contains the names of training and testing file
        names = list(F.keys())

        X_train = np.array(F[names[2]][:])
        y_train = np.array(F[names[3]][:])
        X_test = np.array(F[names[0]][:])
        y_test = np.array(F[names[1]][:])

    y_train = y_train.reshape(y_train.shape[0],1)
    y_test = y_test.reshape(y_test.shape[0],1)

    return X_train,y_train,X_test,y_test

X_train,y_train,X_test,y_test = load_dataset('assign1_data1.h5')
```

```
def print_shapes():
    print(f"X_train has a shape  : {X_train.shape} and contains {X_train.sh
ape[0]} training images with 28x28 pixel" )
    print(f"y_train has a shape  : {y_train.shape}")
    print(f"X_test  has a shape  : {X_test.shape}  and contains {X_test.sha
pe[0]} training images with 28x28 pixel" )
    print(f"y_test  has a shape  : {y_test.shape}")
print_shapes()


def unique_images(X: np.ndarray,Y:np.ndarray) -> list:
    """
    Given traning and testing images,
    return  images of unique alphabet letters
    """
    size = X.shape[0]
    letter = 1
    uniqe_letter = []
    unique_image = None
    uniqe_index = []

    for i in range(size):
        if letter == int(Y[i]):
            unique_image = X[i]
            uniqe_letter.append(unique_image)
            uniqe_index.append(i)
            letter += 1

    return uniqe_letter,uniqe_index

images,indexes = unique_images(X_test,y_test)


alphabet = ['A', 'B', 'C', 'D', 'E', 'F', 'G',
       'H', 'I', 'J', 'K', 'L', 'M', 'N',
       'O', 'P', 'Q', 'R', 'S', 'T', 'U',
       'V', 'W', 'X', 'Y', 'Z']

plt.figure(figsize=(18,15))

for letter,image in enumerate(images):
    plt.subplot(4,7,letter+1)
    plt.imshow(image,cmap='gray')
    plt.axis('off')
    plt.title(f"Character : {alphabet[letter]}")


def corr(X,nclass,pos):
    mat = np.zeros(nclass*nclass).reshape(nclass,nclass)
    x = X.T
    for i in range(nclass):
        for j in range(nclass):
            mat[i,j] = np.corrcoef(x[:,:,pos[i]].flat, x[:,:,pos[j]].flat)[
```

```
0,1]
    return mat

df_correlation_same = pd.DataFrame(corr(X_test,nclass=26,pos = indexes))
sns.heatmap(df_correlation_same, annot = False , cmap = 'gray')
plt.show()
```

5.2.2 Part B

```
def one_hot_encoder(Y):
    classes = int(np.max(Y))
    dim = Y.shape[0]
    val = np.zeros(classes*dim).reshape(dim,classes)

    for num in range(dim):
        val[num,int(Y[num])-1] = 1

    return val

y_train = one_hot_encoder(y_train)
y_test = one_hot_encoder(y_test)

print_shapes()

def flatten_images(X1,X2):
    X1 = X1.reshape(X1.shape[0],-1)
    X2 = X2.reshape(X2.shape[0],-1)

    return X1,X2

X_train,X_test = flatten_images(X_train,X_test)

def normalize(X):
    return X/255

X_train = normalize(X_train)
X_test = normalize(X_test)




class SingleLayerNN():
    def __init__(self, learning_rate):
        np.random.seed(0)
        # Introducing bias and weight terms with  Gaussian distribution N(0
; 0:01)
        self.bias = np.random.normal(loc = 0 ,scale = 0.01 ,size = (1,26))

        self.W = np.random.normal(loc = 0 ,scale = 0.01 ,size =(784,26))

        # Learning rate :
        self.lr = learning_rate

        # To keep track loss function :
```

```
        self.cost = []


    def sigmoid(self,X):
        return 1/(1+np.exp(-X))

    def sigmoid_der(self,X):
        return self.sigmoid(X)*(1-self.sigmoid(X))

    def fit(self,X,Y,iterations,verbose = True):
        """
        Given the training images and their corresponding
        labels, fitting the model by :
        1) Forwarding the input to the output layer
        2) Calculate the loss
        3) Back propagation
        4) Update the weights and bias according to the
        gradient descent rule

        """
        # As proposed , 10000 epoch :
        for iter in range(iterations):

            # Randomly selected numbers between 0 and 5200
            random_index = np.random.randint(0,X.shape[0])


            # Indexing randomly for training image
            X_temp = X[random_index].reshape(1,X.shape[1])

            # Indexing randomly for training image's label
            y_temp = Y[random_index].reshape(1,Y.shape[1])

            # Feed forwarding input
            Z = np.dot(X_temp,self.W) + self.bias

            # Passing activation unit
            A = self.sigmoid(Z)


            # Calculating the error
            error = A - y_temp

            # Loss function : Mean Squared Error
            MSE = np.square(error).mean()

            # Adding losses to visualize
            self.cost.append(MSE)

            if verbose == True:
                if epoch % 100 == 0:
                    print(f"The epoch number : {iter} ----> MSE : {MSE} ")
```

```
            # Back propagation part

            # Calculation of derivatives

            # The derivative of Error with respect to the output
            derror_douto = error

            # The derivative of output with respect to the input
            douto_dino = self.sigmoid_der(Z)


            deriv = derror_douto * douto_dino

            # The derivative of input with respect to the weight , then mul
tiply all
            deriv_final = np.dot(X_temp.T,deriv)

            # Updated weights and bias term according to the gradient desce
nt

            self.W -= self.lr * deriv_final
            self.bias -= self.lr * deriv

    def display_weight(self):
        """
        Visualization of the final weight elements
        """

        plt.figure(figsize=(18,15))
        for i in range(26):
            plt.subplot(4,7,i+1)
            plt.imshow(self.W.T[i,:].reshape(28,28),cmap='gray')
            plt.axis('off')
            plt.title(f"Class : {alphabet[i]}")


    def predict(self,X):
        """
        Feed forwarding to test new images and classification
        """
        return self.sigmoid(np.dot(X,self.W)+self.bias)

    def evaluate(self):
        """
        Evaluation of the model by visualization of MSE
        """
        print(f"MSE loss is : {self.cost[-1]}")
        plt.plot(range(len(self.cost)),self.cost)
        plt.xlabel('# of Iterations')
        plt.ylabel('MSE Loss Function')
        plt.title('Model evaluation')
```

```
def accuracy(Y1,Y2):
    """
    Given one prediction and their test label
    counting how many correct outputs do model have
    then dividing it by the total number to give
    accuracy score
    """
    assert(Y1.shape == Y2.shape)
    size = Y1.shape[0]
    count = 0
    for i in range(size):
        if np.argmax(Y1[i]) == np.argmax(Y2[i]):
            count += 1
    return (count/size)*100

def grid_search(X_train,Y_train,X_test,y_test,learning_rates):
    acc = []
    for lr in learning_rates:
        temp_model = SingleLayerNN(lr)
        temp_model.fit(X_train,y_train,10000,False)
        pred = temp_model.predict(X_test)
        acc.append(accuracy(y_test,pred))

    val = learning_rates[np.argmax(acc)]

    return val,SingleLayerNN(val)

# Trying the learning rate for = 0.1,0.2,...,0.9
learning_rates = [i*0.01 for i in range(1,10)]

best_lr, best_model = grid_search(X_train,y_train,X_test,y_test,learning_ra
tes)
print(f" Optimal learning rate is {best_lr}")
best_model.fit(X_train,y_train,10000,False)
# Predictions
best_predictions = best_model.predict(X_test)


# Accuracy Score :
print(f"Accuracy is {accuracy(y_test,best_predictions)}")



best_model.evaluate()

best_model.display_weight()

model = SingleLayerNN(0.054)
model.fit(X_train,y_train,10000,False)
print(f"Best possible accuracy is {accuracy(y_test,model.predict(X_test))}"
)
```

5.2.3 Part C

```
# substantially higher and a subtantially lower learning rate
# so multiplying learning rate by 10 and divided by 10, respectively


learningRate = [best_lr * i for i in [10,1,0.1]]
MeanSqrtError = []

for i in learningRate:
    model_temp = SingleLayerNN(learning_rate = i)
    model_temp.fit(X_train,y_train,iterations = 10000, verbose = False)
    MeanSqrtError.append(model_temp.cost)

plt.figure(figsize=(9,6))
plt.plot(MeanSqrtError[0], color = 'r')
plt.plot(MeanSqrtError[1], color = 'g')
plt.plot(MeanSqrtError[2], color = 'b')
plt.ylabel('Mean Sqaured Error')
plt.legend([f'High Learning Rate : {learningRate[0]}',f'Normal Learning Rat
e : {learningRate[1]}',
            f'Low Learning Rate :{learningRate[2]}'])
plt.show()
```

5.2.4 Part D

```
# I created 3 model with different learning rates as proposed,
# from higher to lower.
model_1 = SingleLayerNN(learningRate[0])
model_2 = SingleLayerNN(learningRate[1])
model_3 = SingleLayerNN(learningRate[2])

# Fitting the model
model_1.fit(X_train,y_train,10000,False)
model_2.fit(X_train,y_train,10000,False)
model_3.fit(X_train,y_train,10000,False)

# Predictions :
y_pred_1 = model_1.predict(X_test)
y_pred_2 = model_2.predict(X_test)
y_pred_3 = model_3.predict(X_test)

print(f" Model 1 Accuracy : {accuracy(y_pred_1,y_test)}")
print(f" Model 2 Accuracy : {accuracy(y_pred_2,y_test)}")
print(f" Model 3 Accuracy : {accuracy(y_pred_3,y_test)}")
```

6 REFERENCES

[1] M. A. Nielsen, “Neural Networks and Deep Learning,” *Neural networks and deep learning*, 01-Jan-1970. [Online]. Available: http://neuralnetworksanddeeplearning.com/index.html. [Accessed: 16-Oct-2020].

[2] *Deep Learning*. [Online]. Available: https://www.deeplearningbook.org/. [Accessed: 16-Oct-2020].

[3] R. Paudel, “Building a Neural Network with a Single Hidden Layer using Numpy,” *Medium*, 18-May-2020. [Online]. Available: https://towardsdatascience.com/building-a-neural-network-with-a-single-hidden-layer-using-numpy-923be1180dbf. [Accessed: 16-Oct-2020].

[4] T. A. I. Team, “Building Neural Networks with Python Code and Math in Detail - II,” *Medium*, 28-Aug-2020. [Online]. Available: https://medium.com/towards-artificial-intelligence/building-neural-networks-with-python-code-and-math-in-detail-ii-bbe8accbf3d1. [Accessed: 16-Oct-2020].