BILKENT UNIVERSITY
ENGINEERING FACULTY
DEPARTMENT OF ELECTRICAL AND ELECTRONICS ENGINEERING

EEE 443/543
Neural Networks – Fall 2021-2022
Assignment 1

Atakan Topcu
21803095

Due 05 November 2021, 17:00PM

Contents

Question 1.

The question asks us to analytically find the prior probability distribution of the network weights of m input neurons with m weights. The loss function for calculating a posteriori distribution of weights are given as

$$J(W) = argmin_W \sum_n (y^n - h(x^n, W)^2 + \beta \sum_i w_i^2$$

We know that posterior distribution is

$p_{W|\mathbf{Y}}(W|\mathbf{Y}) = \frac{p_{Y|\mathbf{W}}(Y|\mathbf{W})p_W(W)}{p_Y(Y)} = \boldsymbol{Constant} * p_{Y|\mathbf{W}}(Y|\mathbf{W})p_W(W)$ where $p_{Y|\mathbf{W}}(Y|\mathbf{W})$ is the likelihood function of W and $p_W(W)$ is the prior distribution. Since the denominator is essentially a constant value, it does not have any effect on the maximization problem.

$$argmax_W\ p_{W|\mathbf{Y}}(W|\mathbf{Y}) = \ argmax_W\ p_{Y|\mathbf{W}}(Y|\mathbf{W})p_W(W)$$

$$\xrightarrow{\boldsymbol{yields}} argmax_W\ p_{Y|\mathbf{W}}(Y|\mathbf{W})p_W(W) = argmax_W \log(\ p_{Y|\mathbf{W}}(Y|\mathbf{W})p_W(W))$$

$$\xrightarrow{\boldsymbol{yields}} argmax_W \log(p_{Y|\boldsymbol{W}}(Y|\boldsymbol{W})) + log(p_W(W)$$
$$= \ argmin_W(\ -\log(p_{Y|\boldsymbol{W}}(Y|\boldsymbol{W})) - log(p_W(W))$$

We can see that the final version of the equation is similar to the given optimization problem. Thus, we can accept the term with the squared error as the likelihood function and the sum of the squared weights multiplied with a constant as the prior distribution of the weights. Thus,

$$-log(p_W(W)) = \beta \sum_i w_i^2$$
$$\xrightarrow{\boldsymbol{yields}} p_W(W) = Ae^{-\beta \sum_i w_i^2}$$

We have to add a constant A to the prior distribution as the data can be discarded during the maximization/minimization process. Thus, we have to first find the coefficient term A. To find the coefficient A, we can use the fact that the sum over all probability distribution equals 1, which equals to

$$\int_{-\infty}^{\infty} Ae^{-\beta \sum_i w_i^2} dw_i = 1$$
$$\xrightarrow{yields} A \int_{-\infty}^{\infty} e^{-\beta \sum_i w_i^2}\ dw_i = 1$$

Furthermore, we know that there are m weights which means we can separate these weights as the multiplication of m integrals.

$$A\int_{-\infty}^{\infty} e^{-\beta w_1^2}\, dw_1 * \int_{-\infty}^{\infty} e^{-\beta w_2^2}\, dw_2 \ldots \int_{-\infty}^{\infty} e^{-\beta w_m^2}\, dw_m = 1$$

Moreover, it is known that

$$A\int_{-\infty}^{\infty} e^{-\beta w_i^2}\, dw_i = \frac{\sqrt{\pi}}{\sqrt{\beta}}$$

This entails that

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

$$\xrightarrow{\textbf{yields}} A = \left(\frac{\sqrt{\beta}}{\sqrt{\pi}}\right)^m = \left(\frac{\beta}{\pi}\right)^{m/2}$$

In the end, our prior probability distribution for network weights becomes,

$$\boldsymbol{p_W(W) = \left(\frac{\beta}{\pi}\right)^{m/2} e^{-\beta \sum_i w_i^2}}$$

Question 2.

In this question, we are given information about an engineer who wants to implement a neural network with a single layer with four input neurons and one output neuron. The engineer wants to implement,

$$(x_1 or \overline{x_2})\ xor\ (\overline{x_3}\ or\ \overline{x_4})$$

For the hidden layer, we are informed to assume four units, and the step function for activation function.

Part A.

In this part, we are asked to analytically derive the set of inequalities for each hidden unit. Before we go on to derive the inequalities, we have to first find the inputs of each hidden unit. To achieve that we first have to expand the given XOR equation. Since the XOR function cannot be separated by a single decision boundary, we have to first write the XOR as the combination of AND, OR functions. Thus,

$$(X\ XOR\ Y) = (X\ AND\ \bar{Y}\ )\ OR\ (\ \bar{X}\ AND\ Y)$$

In this case, X equals x_1 or not x_2. Y equals to not x_3 or not x_4. Thus, we can write the logic operation as follows

$$((x_1 \vee \overline{x_2}) \wedge (x_3 \wedge x_4)) \vee ((x_2 \wedge \overline{x_1}) \wedge (\overline{x_3} \vee \overline{x_4}))$$

$$\xrightarrow{yields} (x_1 \wedge x_3 \wedge x_4) \vee (\overline{x_2} \wedge x_3 \wedge x_4) \vee (\overline{x_1} \wedge x_2 \wedge \overline{x_3}) \vee (\overline{x_1} \wedge x_2 \wedge \overline{x_4})$$

We can see that as we expand the logical operation we have four parts we have to consider which aligns with our four hidden units. So, we have three inputs for each hidden unit without biases. Now, we can move on to finding the inequalities for each neuron.

1st Neuron:

In the first hidden neuron, we will implement the logic function (x_1 and x_3, and x_4).

X_1	X_3	X_4	Out_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

Figure 1. Truth table for the 1st hidden neuron.

From figure 1, we can define the output of this neuron as follows.

$$Out_1 = h(w_{11}x_1 + w_{13}x_3 + w_{14}x_4 + b_1)$$

Also, our activation function (h ()) can be expressed as follows.

$$h(x) = \begin{cases} 1, x \geq 0 \\ 0, otherwise \end{cases}$$

Thus, the inequalities of the first hidden neuron can be written as follows.

$$(\boldsymbol{x_1, x_3, x_4}) = (\mathbf{0,0,0}) \xrightarrow{\textbf{yields}} \; b_1 \; < 0$$

$$= (\mathbf{0,0,1}) \xrightarrow{\textbf{yields}} \; w_{14} + \; b_1 \; < 0$$
$$= (\mathbf{0,1,0}) \xrightarrow{\textbf{yields}} \; w_{13} + \; b_1 \; < 0$$
$$= (\mathbf{0,1,1}) \xrightarrow{\textbf{yields}} \; w_{13} + \; w_{14} + \; b_1 \; < 0$$
$$= (\mathbf{1,0,0}) \xrightarrow{\textbf{yields}} \; w_{11} + \; b_1 \; < 0$$
$$= (\mathbf{1,0,1}) \xrightarrow{\textbf{yields}} w_{11} + \; w_{14} + \; b_1 \; < 0$$
$$= (\mathbf{1,1,0}) \xrightarrow{\textbf{yields}} w_{11} + \; w_{13} + \; b_1 \; < 0$$
$$= (\mathbf{1,1,1}) \xrightarrow{\textbf{yields}} w_{11} + \; w_{13} + \; w_{14} + \; b_1 \; \geq 0$$

2nd Neuron:

In the second hidden neuron, we will implement the logic function (not x_2 and x_3 and x_4).

X_2	X_3	X_4	Out_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**

Figure 2. Truth table for the 2nd hidden neuron.

From figure 2, we can define the output of this neuron as follows.

$$Out_2 = h(w_{22}x_2 + w_{23}x_3 + w_{24}x_4 + b_2)$$

Also, our activation function (h ()) can be expressed as follows.

$$h(x) = \begin{cases} 1, x \geq 0 \\ 0, otherwise \end{cases}$$

Thus, the inequalities of the second hidden neuron can be written as follows.

$$(\boldsymbol{x_2, x_3, x_4}) = (\mathbf{0,0,0}) \xrightarrow{\boldsymbol{yields}} \; b_2 < 0$$

$$= (\mathbf{0,0,1}) \xrightarrow{\boldsymbol{yields}} \; w_{24} + b_2 < 0$$
$$= (\mathbf{0,1,0}) \xrightarrow{\boldsymbol{yields}} \; w_{23} + b_2 < 0$$
$$= (\mathbf{0,1,1}) \xrightarrow{\boldsymbol{yields}} \; w_{23} + w_{24} + b_2 \geq 0$$
$$= (\mathbf{1,0,0}) \xrightarrow{\boldsymbol{yields}} \; w_{22} + b_2 < 0$$
$$= (\mathbf{1,0,1}) \xrightarrow{\boldsymbol{yields}} w_{22} + w_{24} + b_2 < 0$$
$$= (\mathbf{1,1,0}) \xrightarrow{\boldsymbol{yields}} w_{22} + w_{23} + b_2 < 0$$
$$= (\mathbf{1,1,1}) \xrightarrow{\boldsymbol{yields}} w_{22} + w_{23} + w_{24} + b_2 < 0$$

3rd Neuron:

In the third hidden neuron, we will implement the logic function (not x_1 and x_2 and not x_3).

X_1	X_2	X_3	Out_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**

Figure 3. Truth table for the 3rd hidden neuron.

From figure 3, we can define the output of this neuron as follows.

$$Out_3 = h(w_{31}x_1 + w_{32}x_2 + w_{33}x_3 + b_3)$$

Also, our activation function (h ()) can be expressed as follows.

$$h(x) = \begin{cases} 1, x \geq 0 \\ 0, otherwise \end{cases}$$

Thus, the inequalities of the third hidden neuron can be written as follows.

$$(\boldsymbol{x_1, x_2, x_3}) = (\mathbf{0,0,0}) \xrightarrow{\boldsymbol{yields}} \; b_3 < 0$$

$$= (\mathbf{0,0,1}) \xrightarrow{\boldsymbol{yields}} \; w_{33} + b_3 < 0$$
$$= (\mathbf{0,1,0}) \xrightarrow{\boldsymbol{yields}} \; w_{32} + b_3 \geq 0$$

$$= (\mathbf{0},\mathbf{1},\mathbf{1}) \xrightarrow{\textbf{yields}} w_{32} + w_{33} + b_3 < 0$$
$$= (\mathbf{1},\mathbf{0},\mathbf{0}) \xrightarrow{\textbf{yields}} w_{31} + b_3 < 0$$
$$= (\mathbf{1},\mathbf{0},\mathbf{1}) \xrightarrow{\textbf{yields}} w_{31} + w_{33} + b_3 < 0$$
$$= (\mathbf{1},\mathbf{1},\mathbf{0}) \xrightarrow{\textbf{yields}} w_{31} + w_{32} + b_3 < 0$$
$$= (\mathbf{1},\mathbf{1},\mathbf{1}) \xrightarrow{\textbf{yields}} w_{31} + w_{32} + w_{33} + b_3 < 0$$

4th Neuron:

In the fourth hidden neuron, we will implement the logic function (not x_1 and x_2 and not x_4).

X_1	X_2	X_4	Out_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**

Figure 4. Truth table for the 4th hidden neuron.

From figure 4, we can define the output of this neuron as follows.

$$Out_4 = h(w_{41}x_1 + w_{42}x_2 + w_{44}x_4 + b_4)$$

Also, our activation function (h ()) can be expressed as follows.

$$h(x) = \begin{cases} 1, x \geq 0 \\ 0, otherwise \end{cases}$$

Thus, the inequalities of the fourth hidden neuron can be written as follows.

$$(\boldsymbol{x_1}, \boldsymbol{x_2}, \boldsymbol{x_4}) = (\mathbf{0},\mathbf{0},\mathbf{0}) \xrightarrow{\textbf{yields}} b_4 < 0$$

$$= (\mathbf{0},\mathbf{0},\mathbf{1}) \xrightarrow{\textbf{yields}} w_{44} + b_4 < 0$$
$$= (\mathbf{0},\mathbf{1},\mathbf{0}) \xrightarrow{\textbf{yields}} w_{42} + b_4 \geq 0$$
$$= (\mathbf{0},\mathbf{1},\mathbf{1}) \xrightarrow{\textbf{yields}} w_{42} + w_{44} + b_4 < 0$$
$$= (\mathbf{1},\mathbf{0},\mathbf{0}) \xrightarrow{\textbf{yields}} w_{41} + b_4 < 0$$
$$= (\mathbf{1},\mathbf{0},\mathbf{1}) \xrightarrow{\textbf{yields}} w_{41} + w_{44} + b_4 < 0$$
$$= (\mathbf{1},\mathbf{1},\mathbf{0}) \xrightarrow{\textbf{yields}} w_{41} + w_{42} + b_4 < 0$$
$$= (\mathbf{1},\mathbf{1},\mathbf{1}) \xrightarrow{\textbf{yields}} w_{41} + w_{42} + w_{44} + b_4 < 0$$

Output Neuron:

In the output neuron, we will implement the logic function (Out_1 or Out_2 or Out_3 or Out_4).

Out_1	Out_2	Out_3	Out_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

Figure 5. Truth table for the output neuron.

From figure 5, we can define the output of this neuron as follows.

$$Output = h(w_{51}Out_1 + w_{52}Out_2 + w_{53}Out_3 + w_{54}Out_4 + b_5)$$

Also, our activation function (h ()) can be expressed as follows.

$$h(x) = \begin{cases} 1, x \geq 0 \\ 0, otherwise \end{cases}$$

Thus, the inequalities of the output neuron can be written as follows.

$$\begin{aligned}
(\boldsymbol{Out_1, Out_2, Out_3, Out_4}) = (\mathbf{0,0,0,0}) &\xrightarrow{\textbf{yields}} \; b_5 < 0 \\
= (\mathbf{0,0,0,1}) &\xrightarrow{\textbf{yields}} \; w_{54} + b_5 \geq 0 \\
= (\mathbf{0,0,1,0}) &\xrightarrow{\textbf{yields}} \; w_{53} + b_5 \geq 0 \\
= (\mathbf{0,0,1,1}) &\xrightarrow{\textbf{yields}} \; w_{53} + w_{54} + b_5 \geq 0 \\
= (\mathbf{0,1,0,0}) &\xrightarrow{\textbf{yields}} \; w_{52} + b_5 \geq 0 \\
= (\mathbf{0,1,0,1}) &\xrightarrow{\textbf{yields}} w_{52} + w_{54} + b_5 \geq 0 \\
= (\mathbf{0,1,1,0}) &\xrightarrow{\textbf{yields}} w_{52} + w_{53} + b_5 \geq 0 \\
= (\mathbf{0,1,1,1}) &\xrightarrow{\textbf{yields}} w_{52} + w_{53} + w_{54} + b_5 \geq 0 \\
= (\mathbf{1,0,0,0}) &\xrightarrow{\textbf{yields}} w_{51} + b_5 \geq 0 \\
= (\mathbf{1,0,0,1}) &\xrightarrow{\textbf{yields}} \; w_{51} + w_{54} + b_5 \geq 0
\end{aligned}$$

$$= (\mathbf{1},\mathbf{0},\mathbf{1},\mathbf{0}) \xrightarrow{\textbf{yields}} w_{51} + w_{53} + b_5 \geq 0$$
$$= (\mathbf{1},\mathbf{0},\mathbf{1},\mathbf{1}) \xrightarrow{\textbf{yields}} w_{51} + w_{53} + w_{54} + b_5 \geq 0$$
$$= (\mathbf{1},\mathbf{1},\mathbf{0},\mathbf{0}) \xrightarrow{\textbf{yields}} w_{51} + w_{52} + b_5 \geq 0$$
$$= (\mathbf{1},\mathbf{1},\mathbf{0},\mathbf{1}) \xrightarrow{\textbf{yields}} w_{51} + w_{52} + w_{54} + b_5 \geq 0$$
$$= (\mathbf{1},\mathbf{1},\mathbf{1},\mathbf{0}) \xrightarrow{\textbf{yields}} w_{51} + w_{52} + w_{53} + b_5 \geq 0$$
$$= (\mathbf{1},\mathbf{1},\mathbf{1},\mathbf{1}) \xrightarrow{\textbf{yields}} w_{51} + w_{52} + w_{53} + w_{54} + b_5 \geq 0$$

Part B.

Using the inequalities in part A, I have chosen parameters that would cover all of these inequalities. Here, I have chosen the values one by one. For example, first I have picked a random w_1 then I have picked w_2 then I have chosen w_3 and w_4. Lastly, I have chosen the bias term. The hidden units' weight matrix is given below.

$$\boldsymbol{W_{hidden,1}} = \begin{bmatrix} \mathbf{2} & \mathbf{0} & \mathbf{2} & \mathbf{4} \\ \mathbf{0} & \mathbf{-4} & \mathbf{2} & \mathbf{4} \\ \mathbf{-3} & \mathbf{3} & \mathbf{-2} & \mathbf{0} \\ \mathbf{-2} & \mathbf{4} & \mathbf{0} & \mathbf{-2} \end{bmatrix}, \boldsymbol{Bias} = \begin{bmatrix} \mathbf{-7} \\ \mathbf{-5} \\ \mathbf{-2} \\ \mathbf{-3} \end{bmatrix}$$

$$\boldsymbol{W_{hidden}} = \begin{bmatrix} \mathbf{2} & \mathbf{0} & \mathbf{2} & \mathbf{4} & \mathbf{-7} \\ \mathbf{0} & \mathbf{-4} & \mathbf{2} & \mathbf{4} & \mathbf{-5} \\ \mathbf{-3} & \mathbf{3} & \mathbf{-2} & \mathbf{0} & \mathbf{-2} \\ \mathbf{-2} & \mathbf{4} & \mathbf{0} & \mathbf{-2} & \mathbf{-3} \end{bmatrix}$$

The python script for weight generation is given below.

```
hidden_Weights = np.matrix('2 0 2 4 -7; 0 -4 2 4 -5; -3 3 -2 0 -2; -2 4 0 -2 -3')
print(hidden_Weights)
```

```
[[ 2  0  2  4 -7]
 [ 0 -4  2  4 -5]
 [-3  3 -2  0 -2]
 [-2  4  0 -2 -3]]
```

After generating the weight matrix, to check the accuracy of the system, every possible combination of input values is generated including the bias term. The input matrix is given as follows. Furthermore, the python code is also given below.

$$Input = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

```
inputs = np.zeros([16,5])
i3 = -1
i2 = -1
i1 = -1
i0 = -1

for i in range(16):
    inputs[i,0] = i0
    inputs[i,1] = i1
    inputs[i,2] = i2
    inputs[i,3] = i3

    i3 = -i3
    if(i % 2):
        i2 = -i2
    if(i % 4 == 3):
        i1 = -i1
    if(i % 8 == 7):
        i0 = -i0
inputs[inputs < 0] = 0
inputs[:,4] = 1
print(inputs)
```

```
[[0. 0. 0. 0. 1.]
 [0. 0. 0. 1. 1.]
 [0. 0. 1. 0. 1.]
 [0. 0. 1. 1. 1.]
 [0. 1. 0. 0. 1.]
 [0. 1. 0. 1. 1.]
 [0. 1. 1. 0. 1.]
 [0. 1. 1. 1. 1.]
 [1. 0. 0. 0. 1.]
 [1. 0. 0. 1. 1.]
 [1. 0. 1. 0. 1.]
 [1. 0. 1. 1. 1.]
 [1. 1. 0. 0. 1.]
 [1. 1. 0. 1. 1.]
 [1. 1. 1. 0. 1.]
 [1. 1. 1. 1. 1.]]
```

Now, we will define the activation function (i.e, step function) for our system. The code is given below.

```
#Now we will define the activation function (i.e, step function)
def activation(inputs,weights):
    v=np.matmul(weights,inputs.T)
    v[v >= 0] = 1
    v[v < 0]= 0
    output = v
    return output
```

```
Output=activation(inputs,hidden_Weights)
print(Output)
```

```
[[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 1.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
```

Now, the code for the hidden layer is completed. For accuracy evaluation, we also need to decide the output neuron weights. Using the inequalities in part A, I have decided the parameters the same way I have decided for hidden weights. After assigning the values that cover the inequalities, the weights of the output neuron are as follows.

$$\boldsymbol{W_{Output,1}} = [\mathbf{2} \quad \mathbf{2} \quad \mathbf{3} \quad \mathbf{2}] \; \boldsymbol{Bias_{Output}} = [\mathbf{-1}]$$

$$\boldsymbol{W_{Output}} = [\mathbf{2} \quad \mathbf{2} \quad \mathbf{3} \quad \mathbf{2} \quad \mathbf{-1}]$$

The output of the output neuron is the matrix multiplication of W_{output} and $Output_{hidden}$.Transpose.

$$\boldsymbol{Output = h(W_{output} Out_{hidden}^{T})}$$

The implemented code is given below.

```
output_Weights= np.matrix('2 2 3 2 -1')
output_bias=np.ones([16,1])
Outp_hidden=Output_hidden.T
Out_hidden=np.concatenate((Outp_hidden,output_bias),axis=1)
Output=activation(Out_hidden,output_Weights)

print("<======Output Matrix======>")
print(Output)
```

```
<======Output Matrix======>
[[0. 0. 0. 1. 1. 1. 1. 0. 0. 0. 0. 1. 0. 0. 0. 1.]]
```

After finding the network output, I have created a logic operation to compare the network output with real output values. By doing that, I will check the accuracy of the network. Thus, I have first defined an XOR function. Its code is given below.

```
def XOR(A,B):
    boolean_arrayA=np.array(A, dtype=bool)
    boolean_arrayB=np.array(B, dtype=bool)
    Part1=np.logical_and(boolean_arrayA,np.logical_not(boolean_arrayB))
    Part2=np.logical_and(np.logical_not(boolean_arrayA),boolean_arrayB)
    Out=np.logical_or(Part1,Part2)
    return Out
```

After creating the XOR function, to compare the output value of the network and the output of the XOR function, I implemented a code to compare these two values. The code is as follows.

```
x1=inputs[:,0]
x2=inputs[:,1]
x3=inputs[:,2]
x4=inputs[:,3]
A=np.logical_or(x1,np.logical_not(x2))
B=np.logical_or(np.logical_not(x3),np.logical_not(x4))
Xor_list=XOR(A,B)
boolean_output=np.array(Output, dtype=bool)
if (boolean_output==Xor_list).all():
    print("100% Accuracy")
```

```
100% Accuracy
```

As we can see, our network's output and the XOR function's output are equal to each other which means our network has 100% accuracy as requested.

Part C.

In part B, we have observed that our network was able to function perfectly under ideal conditions. However, this does not always the case. If we consider the noise for our input samples, our input values might change their states (i.e, from 1 to 0 or from 0 to 1). Thus, I would not consider this network to be robust.

In order to find the most robust decision boundary, I choose the weights as either 1 or -1 for orthogonality and symmetry. By choosing small integers, we minimize the effect of the noise on the network. However, we do not go below 1 or -1 as it would then reduce the importance of undisturbed samples. We can then decide the range of biases of the system using the previous inequalities.

1st Neuron:

As stated above, I am choosing weights as follows.

$$w_{11} = 1\,, w_{13} = 1\,, w_{14} = 1$$

To determine the bias, I use the inequalities found in part A.

$$w_{11} + w_{13} + b_1 < 0$$

$$w_{11} + w_{13} + w_{14} + b_1 \geq 0$$

$$\xrightarrow{yields} w_{11} + w_{13} < -b_1 \leq w_{11} + w_{13} + w_{14}$$

$$\xrightarrow{yields} 2 < -b_1 \leq 3$$

So, I choose -2.5 for b_1.

2nd Neuron:

As stated above, I am choosing weights as follows.

$$w_{22} = -1\,, w_{23} = 1\,, w_{24} = 1$$

To determine the bias, I use the inequalities found in part A.

$$w_{22} + w_{23} + w_{24} + b_2 < 0$$

$$w_{23} + w_{24} + b_2 \geq 0$$

$$\xrightarrow{yields} w_{22} + w_{23} + w_{24} < -b_2 \leq w_{23} + w_{24}$$

$$\xrightarrow{yields} 1 < -b_2 \leq 2$$

So, I choose -1.5 for b_2.

3rd Neuron:

As stated above, I am choosing weights as follows.

$$w_{31} = -1, w_{32} = 1, w_{33} = -1$$

To determine the bias, I use the inequalities found in part A.

$$w_{32} + b_3 \geq 0$$

$$b_3 < 0$$

$$\xrightarrow{yields} 0 < -b_3 \leq w_{32}$$

$$\xrightarrow{yields} 0 < -b_3 \leq 1$$

So, I choose -0.5 for b_3.

4th Neuron:

As stated above, I am choosing weights as follows.

$$w_{41} = -1, w_{42} = 1, w_{44} = -1$$

To determine the bias, I use the inequalities found in part A.

$$w_{42} + b_4 \geq 0$$

$$b_4 < 0$$

$$\xrightarrow{yields} 0 < -b_4 \leq w_{42}$$

$$\xrightarrow{yields} 0 < -b_4 \leq 1$$

So, I choose -0.5 for b_4.

Now that we decided on the hidden weights, we can determine the output neuron's weights.

Output Neuron:

For the output neuron, I implement the logic operation as (Out1 or Out2 or Out3 or Out4). Since each input is positive, I choose the weights of these inputs to be 1. Thus,

$$w_{51} = 1, w_{52} = 1, w_{53} = 1, w_{54} = 1$$

To determine the bias, I use the inequalities found in part A.

$$w_{51} + b_5 \geq 0$$

$$b_5 < 0$$

$$\xrightarrow{yields} 0 < -b_5 \leq w_{51}$$

$$\xrightarrow{yields} 0 < -b_5 \leq 1$$

So, I choose -0.5 for b_5.

The final matrix form of the weights is as follows.

$$\boldsymbol{W_{Hidden,1}} = \begin{bmatrix} \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{1} \\ \mathbf{0} & \mathbf{-1} & \mathbf{1} & \mathbf{1} \\ \mathbf{-1} & \mathbf{1} & \mathbf{-1} & \mathbf{0} \\ \mathbf{-1} & \mathbf{1} & \mathbf{0} & \mathbf{-1} \end{bmatrix}, \boldsymbol{Bias} = \begin{bmatrix} \mathbf{-2.5} \\ \mathbf{-1.5} \\ \mathbf{-0.5} \\ \mathbf{-0.5} \end{bmatrix}$$

$$\boldsymbol{W_{Output,1}} = [\mathbf{1} \quad \mathbf{1} \quad \mathbf{1} \quad \mathbf{1}] \; \boldsymbol{Bias_{Output}} = [\mathbf{-0.5}]$$

Part D.

In this part, we are asked to evaluate both of our NNs to evaluate whether the neural network that we have determined in part C works more robustly in case of noise compared to the neural network that we have chosen in part A. To create the input matrix, we are instructed to generate 100 input samples by first concatenating 25 samples from each input vector. Columns of the matrix represent the inputs and since there are 16 possible combinations at maximum, the final matrix turns out to be a 400x4 matrix. Firstly, I generate the weight matrix for the neural network in Part C.

```
hiddenWeightsRobust = np.matrix('1 0 1 1 -2.5; 0 -1 1 1 -1.5; -1 1 -1 0 -0.5; -1 1 0 -1 -0.5')
print(hiddenWeightsRobust)
```

```
[[ 1.   0.   1.   1.  -2.5]
 [ 0.  -1.   1.   1.  -1.5]
 [-1.   1.  -1.   0.  -0.5]
 [-1.   1.   0.  -1.  -0.5]]
```

After the generation of the weight matrix for part C, I generate the input matrix along with the noise matrix

```
std=0.2
mean=0
Noise=np.random.normal(mean, std, (400,5))
print(Noise.shape)
```

```
(400, 5)
```

```
std=0.2
mean=0
Noise=np.random.normal(mean, std, (400,5))
Noise[:,4]=0 #Bias
print(Noise.shape)
Input_with_Noise=inputsRobust+Noise
```

```
(400, 5)
```

After the generation of the input matrix, I evaluate the neural network that I determined in part C.

```
output_Weights_Robust= np.matrix('1 1 1 1 -0.5')
output_bias=np.ones([400,1])
Outp_hidden_robust=Output_hidden_robust.T
Out_hidden=np.concatenate((Outp_hidden_robust,output_bias),axis=1)
Output_Robust=activation(Out_hidden,output_Weights_Robust)

print("<======Output_Robust Matrix======>")
print(Output_Robust)
```

```
<======Output_Robust Matrix======>
[[1. 0. 0. 0. 0. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
  1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 1. 0. 0. 1. 0. 0. 0.
  0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0.
  0. 0. 0. 1. 1. 1. 1. 1. 1. 1. 0. 1. 1. 1. 0. 1. 0. 1. 1. 1. 1. 1. 1. 1.
  1. 1. 1. 0. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
  1. 1. 1. 1. 1. 1. 0. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
  1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
  1. 1. 1. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 0. 0. 0. 0.
  0. 0. 0. 0. 0. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
  0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0.
  0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0.
  0. 0. 0. 0. 0. 0. 1. 0. 1. 0. 0. 1. 1. 1. 1. 1. 0. 1. 1. 1. 1. 1. 1. 1.
  1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 0. 1. 0. 0. 0. 1. 0. 0. 0. 0. 0.
  0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 1. 0. 0. 0.
  0. 0. 1. 1. 0. 0. 0. 1. 0. 0. 0. 0. 0. 1. 1. 0. 0. 1. 0. 0. 0. 0. 0. 0.
  0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 1. 0. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
  0. 1. 1. 0. 1. 1. 1. 1. 1. 0. 1. 1. 1. 1. 0. 1.]]
```

```
output_Weights= np.matrix('2 2 3 2 -1')
output_bias=np.ones([400,1])
Outp_hidden=Output_hidden.T
Out_hidden=np.concatenate((Outp_hidden,output_bias),axis=1)
Output=activation(Out_hidden,output_Weights)

print("<======Output Matrix======>")
print(Output)
```

```
<======Output Matrix======>
[[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
  1. 0. 0. 0. 0. 1. 0. 0. 0. 0. 1. 0. 0. 1. 0. 1. 0. 1. 1. 0. 0. 0. 0. 0.
  0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
  0. 0. 0. 1. 1. 1. 1. 1. 1. 0. 1. 0. 1. 0. 1. 0. 1. 1. 1. 1. 1. 0. 1. 1.
  1. 1. 1. 0. 1. 1. 0. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
  1. 1. 1. 1. 1. 1. 0. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
  1. 1. 1. 1. 1. 1. 1. 1. 0. 1. 0. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
  1. 1. 1. 1. 1. 1. 1. 0. 0. 0. 1. 1. 0. 0. 0. 1. 0. 0. 1. 1. 1. 0. 0. 0.
  0. 0. 0. 0. 0. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
  0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 1. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0.
  1. 0. 0. 0. 0. 1. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
  0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1. 1. 1. 0. 1. 1. 1. 0. 1. 1.
  1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0.
  0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0.
  0. 0. 1. 0. 0. 0. 0. 1. 0. 0. 1. 0. 0. 1. 1. 0. 0. 0. 0. 1. 0. 0. 0. 0.
  0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1. 1. 1. 1. 1. 1.
  0. 1. 1. 0. 1. 1. 1. 1. 1. 0. 1. 1. 1. 1. 0. 1.]]
```

After generating the output values, we need to evaluate them in order to check their robustness. Using the XOR function that I have created in Part B, I have evaluated the accuracy of both NNs.

```
x1=inputsRobust[:,0]
x2=inputsRobust[:,1]
x3=inputsRobust[:,2]
x4=inputsRobust[:,3]
A=np.logical_or(x1,np.logical_not(x2))
B=np.logical_or(np.logical_not(x3),np.logical_not(x4))
Xor_list=XOR(A,B)
boolean_output=np.array(Output, dtype=bool)
boolean_output_robust=np.array(Output_Robust, dtype=bool)
```

```
correct_robust=0
correct=0
for i in range(400):
    if(Xor_list[i] == boolean_output[0,i]):
        correct += 1
print('Accuracy of NN in Part B = ' + str(correct/400*100) + "%")
```

```
Accuracy of NN in Part B = 87.5%
```

```
for i in range(400):
    if(Xor_list[i] == boolean_output_robust[0,i]):
        correct_robust += 1
print('Accuracy of NN in Part C = ' + str(correct_robust/400*100) + "%")
```

```
Accuracy of NN in Part C = 91.0%
```

```
print("Accuracy of the NN in Part C is better " + str(correct_robust/400*100) + "% " + "> " + str(correct/400*100) + "
```

```
Accuracy of the NN in Part C is better 91.0% > 87.5%
```

As we can see, the accuracy of the NN in part C is better compared to the first neural network which shows that the weight matrix that we chose for part C provides robustness to the system. So, the robust system performed with 91% accuracy while the non-robust system performed with 87.5% accuracy. The most significant part of this experiment is that we were able to create a significantly better network by selecting weights and biases more carefully. Thus, it shows the importance of planning the weights and biases before initiating the neural network. However, these performances are not absolute. If we rerun the whole code, we might get slightly different accuracies due to a change in the noise matrix.

Question 3.

Part A.

In this part, we are asked to acquire sample images of each class for visualization and find the correlation coefficients between pairs of sample images that we have selected. For this, we first need to read the given hdf5 file and extract the keys.

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import random
import h5py
filename = "assign1_data1.h5"
```

```
file = h5py.File(filename, "r") #Load Data
```

```
# List all groups
print("Keys: %s" % list(file.keys()))
testims = list(file.keys())[0]
testlbls = list(file.keys())[1]
trainims = list(file.keys())[2]
trainlbls = list(file.keys())[3]

# Get the data
test_images = np.array(file[testims]).T
test_labels = np.array(file[testlbls]).T
train_images = np.array(file[trainims]).T
train_labels = np.array(file[trainlbls]).T
```

```
Keys: ['testims', 'testlbls', 'trainims', 'trainlbls']
```

```
train_size = train_labels.shape[0]
print("Number of Train Samples:",train_size) #Shows the number of train samples

print("Train Image Size & Train Label Size:", train_images.shape,train_labels.shape)
```

```
Number of Train Samples: 5200
Train Image Size & Train Label Size: (28, 28, 5200) (5200,)
```

Then, we show the image of each class' first sample.

The code for visualization is given below.

```
#Use the first sample of each class for visualization.
current_letter = 1
sample_ind = list()

row = 5
col = 6
fig = plt.figure()

for i in range(train_size):
    if(current_letter == train_labels[i]):
        ax = plt.subplot(row, col, current_letter)
        plt.imshow(train_images[:,:,i], cmap='gray')
        ax.axis('off')
        ax.autoscale(False)
        sample_ind.append(i)
        current_letter += 1
plt.savefig('Samples.png')
```

The resulting image is as follows.

Figure 6. Visualization of the first sample of each class.

For the correlation matrix, I have used corrcoef provided by NumPy which calculates the correlation coefficient given as

$$corr_{X,Y} = \frac{Cov(X,Y)}{std(X)std(Y)}$$

This formula explains the linear correlation between two random variables.
The code for the generation of the correlation matrix is as follows.

```
#Corr Matrix

num_class = 26
corr_matrix = np.zeros(num_class**2).reshape(num_class,num_class)

for i in range(num_class):
    for j in range(num_class):
        corr_matrix[i,j] = np.corrcoef(train_images[:,:,sample_ind[i]].flat, train_images[:,:,sample_ind[j]].flat)[0,1]

corrMap=sn.heatmap(corr_matrix)
plt.savefig("CorrMatrix.png")
```

The resulting correlation matrix is given as the heat map plot in figure 7.

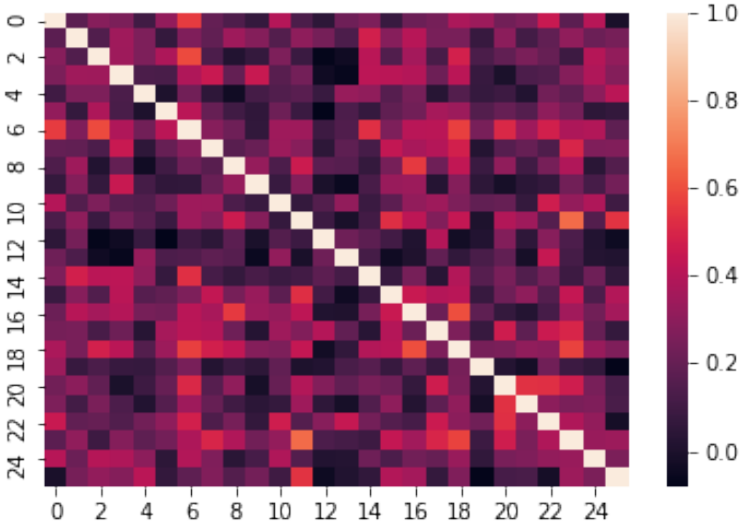


Figure 7. Heat map representation of correlation matrix.

From figure 7, we can observe that diagonal of the matrix is white which indicates that the correlation of the diagonal elements is 1. This is an expected result as diagonal parts represent the correlation of the same inputs.

$$corr_{X,X} = \frac{Cov(X,X)}{std(X)std(X)} = \frac{var(X)}{std(X)^2} = \frac{var(X)}{var(X)} = 1$$

We can define within-class variability as the variance of the members of the class. Also, we can define across-class variability as the variance between the mean of the members of classes. From figure 7, we can get a rough idea of the across-class variability. For different classes, we expect that the correlation between them is near 0. However, we can see that for some classes, this is not the case as some of the correlations is near 0.6. This can be interpreted as the difficulty in the classification of the classes. Furthermore, it can also be interpreted that for some classes, across-class variability is less. Nonetheless, we cannot understand the within-class variability by looking at figure 7. Thus, I have modified the code to understand the within-class variability better. The resulting correlation matrix is given in figure 8.

From figure 8, we can see that diagonal of the correlation matrix is not close to 1. This is because this time, I have taken the correlation between two different samples of the same class for diagonal elements. Thus, it shows that since the correlation between them is small, within-class variability is not so small. This might hinder our network's ability to train efficiently. However, it might also help the network to avoid overfitting.

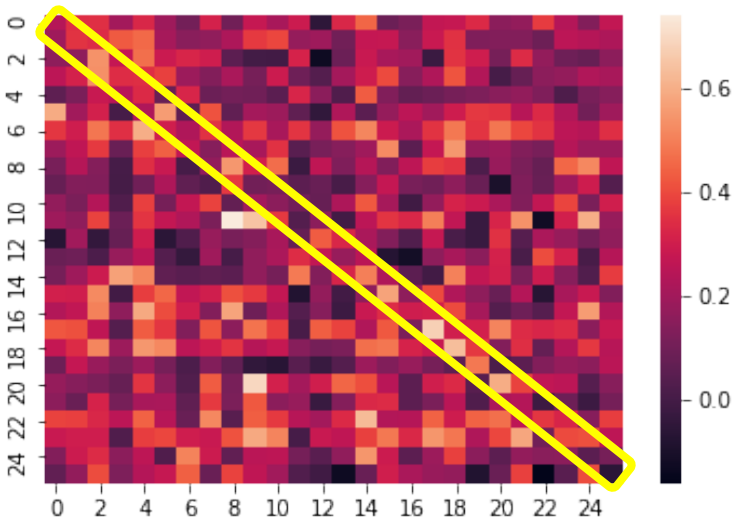


Figure 8. Heat map representation of correlation matrix.

Part B.

We are asked to design a single-layer network that has 26 output neurons using sigmoid activation function. Each neuron is assigned to a different class. Since we are using sigmoid activation function, our output will be between 0 and 1. Thus, to evaluate the output, we need to convert our training label to values between 0 and 1. One-hot encoding can be used for such conversion.

```
#26 Output Neuron, 26 Input Neuron.
#No Hidden Layer.
#Sigmoid Activation Function.

def sigmoid(activation_output):
    transfer=1/(1 + np.exp(-activation_output))
    return transfer

def forward(W, x, b):
    return sigmoid(np.matmul(W,x) - b)
```

```
#Initialization
initialBias=np.random.normal(0, 0.01,(num_class,1))
pixelSize=train_images.shape[0]
initialW=np.random.normal(0, 0.01,(num_class,pixelSize**2))
```

```
#Arrange Labels for Sigmoid Output
OneHoT = np.zeros((train_labels.max().astype(int),train_labels.size))
for i in range(train_labels.size):
    OneHoT[train_labels[i].astype(int)-1,i] = 1
print(OneHoT)
```

```
[[1. 1. 1. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 1. 1. 1.]]
```

After defining the initial weight matrix and the bias vector along with the forward propagation function, I need to define the MSE and stochastic gradient descent rule. We can define mean squared error as follows.

$$MSE = \frac{1}{N}\sum_{i=0}^{N}\|y_{real} - y_{train}\|^2 := \frac{1}{N}\sum_{i=0}^{N} L \ where\ N = sample\ size$$

$$L = \|y_{real} - y_{train}\|^2 = (Y - Y_{train})^T(Y - Y_{train})\ where\ Y_{train} = h(WX - B)$$

By gradient-descent,

$$W' := W - \gamma\nabla_W f(W)\ where\ \nabla_W f(W) = \frac{\partial L}{\partial W}\ and\ \gamma = learning\ rate$$
$$B' := B - \gamma\nabla_B f(B)\ where\ \nabla_B f(B) = \frac{\partial L}{\partial B}\ and\ \gamma = learning\ rate$$

So, taking the derivative of the loss function would give us the gradient for each iteration,

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial Y_{train}} * \frac{\partial Y_{train}}{\partial (WX - B)} * \frac{\partial (WX - B)}{\partial (W)}$$
$$= -2(Y - Y_{train}) * \big(h(WX - B)\big)\big(1 - h(WX - B)\big) * X^T$$

$$\frac{\partial L}{\partial B} = \frac{\partial L}{\partial Y_{train}} * \frac{\partial Y_{train}}{\partial (WX - B)} * \frac{\partial (WX - B)}{\partial (B)}$$
$$= 2(Y - Y_{train}) * \big(h(WX - B)\big)\big(1 - h(WX - B)\big)$$

Now that we have derived the gradient for our network, we can continue with the coding. I have defined a function for updating the weight matrix and the bias vector. I have applied this code loop for 10000 iterations. I have also collected the mean squared errors as requested by the question.

```
MSE = list()
iteration = 10000
l_rate = 0.06

def Update(W,Bias,Difference,Output,sample):
    w_grd = -2*np.matmul(Difference*(Output)*(1-Output),np.transpose(sample))
    b_grd = 2*Difference*(Output)*(1-Output)

    W -= l_rate*w_grd
    Bias -= l_rate*b_grd
    return W,Bias


for i in range(iteration):
    random_index = random.randint(0,train_size-1)
    sample=train_images[:,:,random_index].reshape(pixelSize**2,1)
    sample=sample/np.amax(sample)
    Real_Output=OneHoT[:,random_index].reshape(num_class,1)
    Output=forward(initialW,sample,initialBias)
    Difference=Real_Output-Output
    #Update
    initialW,initialBias=Update(initialW,initialBias,Difference,Output,sample)

    MSE.append(np.sum(Difference**2)/(Difference.shape[0]))

print("Sum of MSE:", sum(MSE))
```

```
Sum of MSE: 247.35786040301306
```

I have chosen 0.06 for my learning rate through trial and error. Choosing 0.06 gave me the least total MSE for every run. The next part of the question requires us to show the trained network weights as images and visualize them. Here is the Python code and the image grid.

```
row = 5
col = 6
for i in range(num_class):
    plott = plt.subplot(row, col, i+1)
    neuron_weight = initialW[i,:].reshape(pixelSize,pixelSize)
    plt.imshow(neuron_weight, cmap='gray')
    plott.axis('off')
    plott.autoscale(True)

plt.savefig('TrainingWeights.png')
```

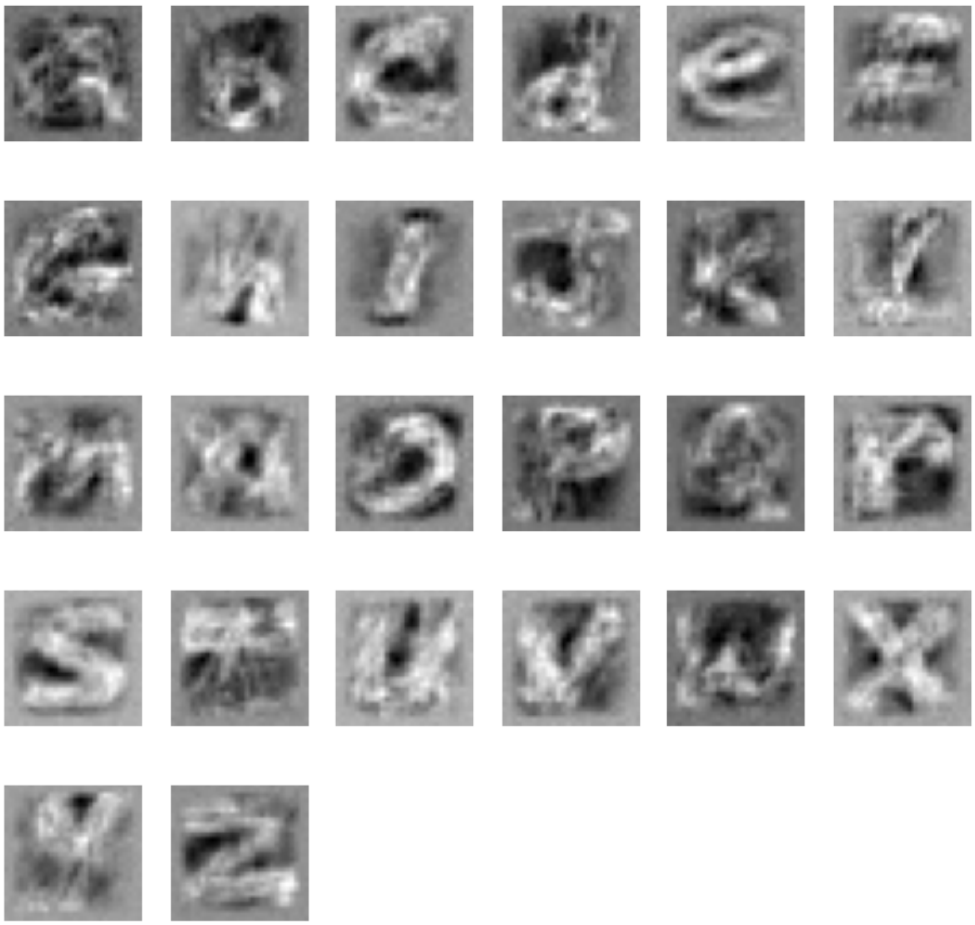


Figure 9. Training weight visualization.

We can see from figure 9 that the network is indeed learning as the images of the weights of each neuron are similar to sample input letters. However, some neuron weight matrices could not learn efficiently due to the various representations of the letter and note that the dataset contains both upper case and lower-case letters under the same label which makes the learning task harder. Furthermore, we know from part A that within-class variability is high which makes the learning task even harder. So, the classes that has less within-class variability was able to be learned much efficiently such as the letter “X” and “x”.

Part C.

This part requires us to repeat the online training process that we have done in the previous part with two different learning rates than what we have used. For the high learning rate, I have chosen 0.8 and for the low learning rate, I have chosen 0.0002. By choosing such significantly different learning rates, we will be able to observe how it affects the gradient-descent algorithm and the MSE of the network.

```
Bias_H=np.random.normal(0, 0.01,(num_class,1))
W_H=np.random.normal(0, 0.01,(num_class,pixelSize**2))

Bias_L=Bias_H
W_L=W_H


l_rate_L = 0.0002
l_rate_H = 0.8


MSE_H = list()
MSE_L = list()

iteration = 10000
```

```
for i in range(iteration):
    random_index = random.randint(0,train_size-1)
    sample=train_images[:,:,random_index].reshape(pixelSize**2,1)
    sample=sample/np.amax(sample)
    Real_Output=OneHoT[:,random_index].reshape(num_class,1)
    Output=forward(W_L,sample,Bias_L)
    Difference=Real_Output-Output
    #Update
    W_L,Bias_L=Update(W_L,Bias_L,Difference,Output,sample,l_rate_L)

    MSE_L.append(np.sum(Difference**2)/(Difference.shape[0]))

for i in range(iteration):
    random_index = random.randint(0,train_size-1)
    sample=train_images[:,:,random_index].reshape(pixelSize**2,1)
    sample=sample/np.amax(sample)
    Real_Output=OneHoT[:,random_index].reshape(num_class,1)
    Output=forward(W_H,sample,Bias_H)
    Difference=Real_Output-Output
    #Update
    W_H,Bias_H=Update(W_H,Bias_H,Difference,Output,sample,l_rate_H)

    MSE_H.append(np.sum(Difference**2)/(Difference.shape[0]))
```

```
plt.plot(MSE_H)
plt.plot(MSE_L)
plt.plot(MSE)
plt.legend(["MSE for u="+str(l_rate_H), "MSE for u="+str(l_rate_L), "MSE for u="+str(l_rate)])
plt.title("Mean Squared Errors for Different Learning Rates")
plt.xlabel("Iteration Number")
plt.ylabel("MSE")
plt.show()
plt.savefig('MSE_Changes.png')
```

The output plot for the code above is given in figure 10.

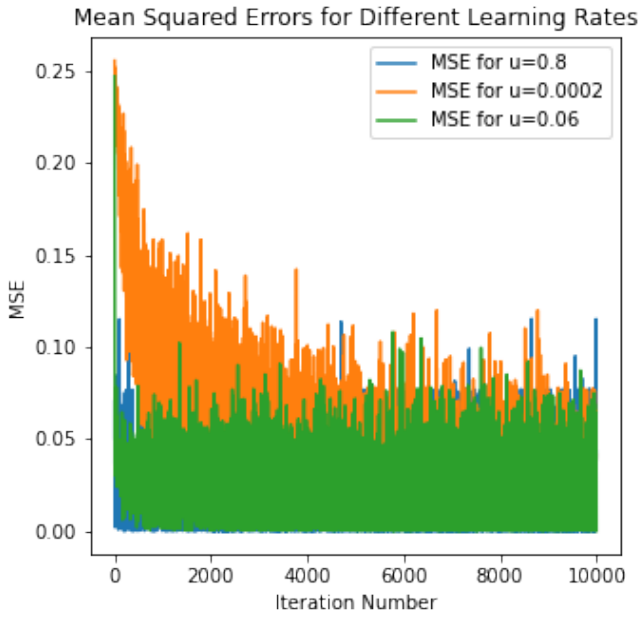


Figure 10. MSE plot for different learning rates.

From figure 10, we can see that the learning rate of 0.0002 slows down the learning process significantly. Moreover, we see that for the learning rate of 0.8, there is a sudden jump in MSE. However, this value does decrease any further since it had stuck in a local minimum.

Part D.

This final part required us to test the network on the test data that the algorithm has never seen. Similar to previous parts, I have used the functions that I have defined in Part A and Part B. I have also converted the labels to one-hot encoding to be able to compare them with the outputs of the network as before.

Part D

```
print("Test Image Size & Test Label Size:", test_images.shape,test_labels.shape)

test_images = test_images.reshape(pixelSize**2,test_labels.shape[0])
test_size = test_labels.shape[0]
bias_matrix = np.zeros((num_class, test_size))

for i in range(test_size):
    bias_matrix[:,i] = initialBias.flatten()
print("bias_matrix size:" ,bias_matrix.shape)

test_images = test_images/np.amax(test_images)
Output = forward(initialW,test_images,bias_matrix)
print("Output Matrix Size:",Output.shape)
```

```
Test Image Size & Test Label Size: (784, 1300) (1300,)
bias_matrix size: (26, 1300)
Output Matrix Size: (26, 1300)
```

```
#For learning rate = 0.06
Output_indices = np.zeros(Output.shape[1])
for i in range(Output.shape[1]):
    Output_indices[i] = np.argmax(Output[:,i])+1 #Returns the index of the maximum element, add 1 since index starts f

true_count = 0
for i in range(Output_indices.shape[0]):
    if(Output_indices[i] == test_labels[i]):
        true_count += 1;

print('Accuracy for Learning rate = 0.06: ', round(true_count/test_labels.shape[0]*100,2),"%")
```

```
Accuracy for Learning rate = 0.06:  59.23 %
```

```
#For learning rate = 0.8
bias_matrix_H= np.zeros((num_class, test_size))

for i in range(test_size):
    bias_matrix_H[:,i] = Bias_H.flatten()
print("bias_matrix size:" ,bias_matrix_H.shape)

Output_H = forward(W_H,test_images,bias_matrix_H)

Output_indices = np.zeros(Output_H.shape[1])

for i in range(Output_H.shape[1]):
    Output_indices[i] = np.argmax(Output_H[:,i])+1 #Returns the index of the maximum element, add 1 since index starts

true_count = 0
for i in range(Output_indices.shape[0]):
    if(Output_indices[i] == test_labels[i]):
        true_count += 1;

print('Accuracy for Learning rate = 0.8: ', round(true_count/test_labels.shape[0]*100,2),"%")
```

```
bias_matrix size: (26, 1300)
Accuracy for Learning rate = 0.8:  20.92 %
```

```
#For learning rate = 0.0002
bias_matrix_L= np.zeros((num_class, test_size))

for i in range(test_size):
    bias_matrix_L[:,i] = Bias_L.flatten()
print("bias_matrix size:" ,bias_matrix_L.shape)

Output_L = forward(W_L,test_images,bias_matrix_L)

Output_indices = np.zeros(Output_L.shape[1])
for i in range(Output_L.shape[1]):
    Output_indices[i] = np.argmax(Output_L[:,i])+1 #Returns the index of the maximum element, add 1 since index starts

true_count = 0
for i in range(Output_indices.shape[0]):
    if(Output_indices[i] == test_labels[i]):
        true_count += 1;

print('Accuracy for Learning rate = 0.0002: ', round(true_count/test_labels.shape[0]*100,2),"%")
```

```
bias_matrix size: (26, 1300)
Accuracy for Learning rate = 0.0002:  29.85 %
```

As it can be observed from the outputs, the optimal learning rate (i.e, 0.06) reaches around the range of 59-60% accuracy. The non-optimal learning rates (i.e, 0.0002 and 0.8) could only reach around 21% and 30% accuracy respectively.

Question 4.

In this question, we are asked to experiment on simple two-layer neural networks and become familiar with them. Answers to this question are written at the end of the code. Some changes applied to cs231n.data_utils as scipy.misc does not support imread anymore. Thus, I have used cv2 instead of scipy.misc. Discussion of the results is also given as comment at the end of the code.

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 [40]:
```python
# A bit of setup

import numpy as np
import matplotlib.pyplot as plt

from cs231n.classifiers.neural_net import TwoLayerNet

from __future__ import print_function

%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))))
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

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 [41]:
```python
# 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()
```

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 [42]:
```python
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
```

Forward pass: compute loss

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

In [43]:
```python
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
```

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 [44]:
```python
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[p
```

```
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
```

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.

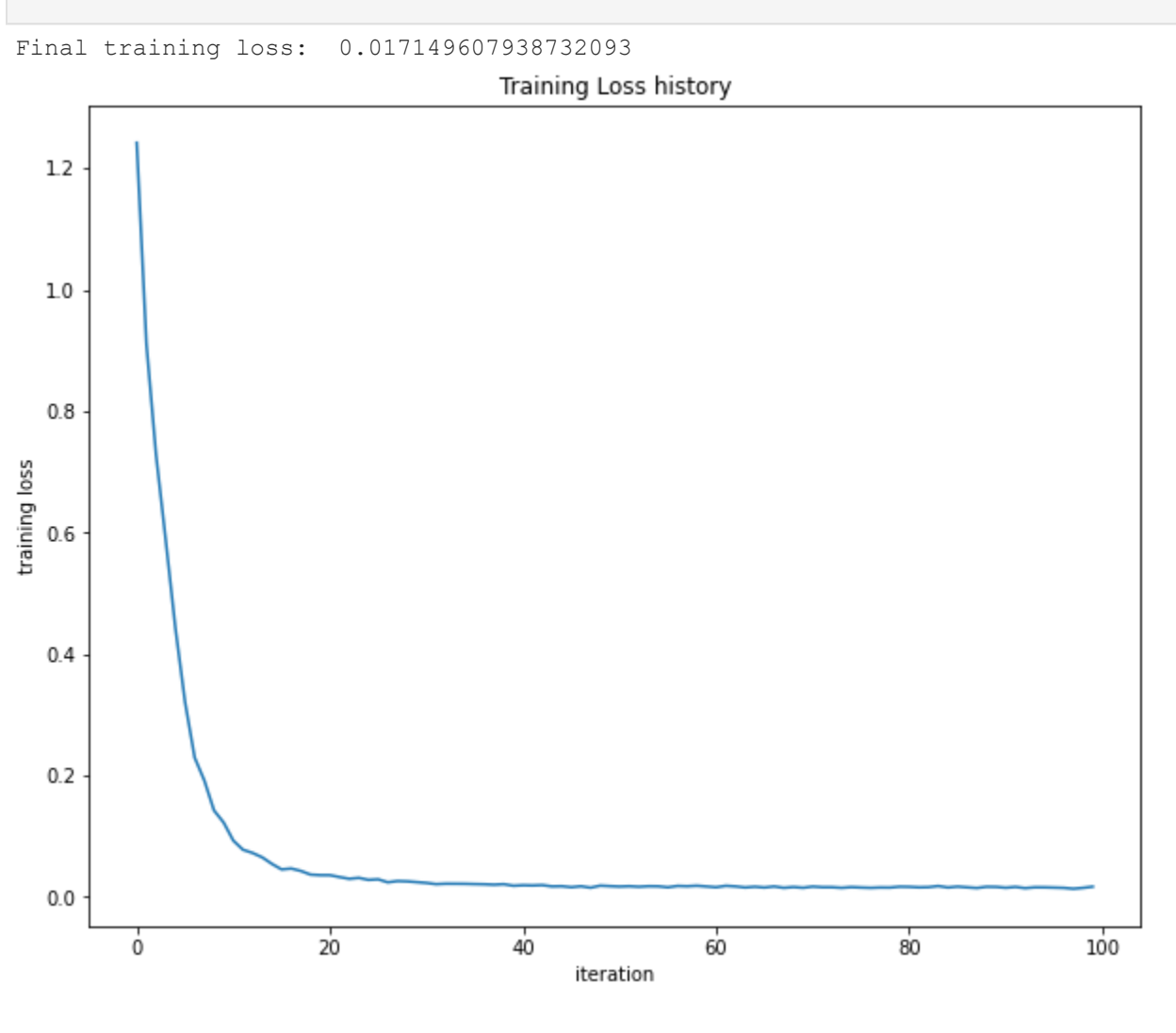
Once you have implemented the method, run the code below to train a two-layer network on toy data. You should achieve a training loss less than 0.2.

In [45]:
```python
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
```



Load the data

Now that you have implemented a two-layer network that passes gradient checks and works on toy data, it's time to load up our favorite CIFAR-10 data so we can use it to train a classifier on a real dataset.

In [46]:
```python
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 memor
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)
```

```
Clear previously loaded data.
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,)
```

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 [47]:
```python
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 2.302954
iteration 100 / 1000: loss 2.302550
iteration 200 / 1000: loss 2.297648
iteration 300 / 1000: loss 2.259602
iteration 400 / 1000: loss 2.204170
iteration 500 / 1000: loss 2.118565
iteration 600 / 1000: loss 2.051535
iteration 700 / 1000: loss 1.988466
iteration 800 / 1000: loss 2.006591
iteration 900 / 1000: loss 1.951473
Validation accuracy:  0.287
```

Debug the training

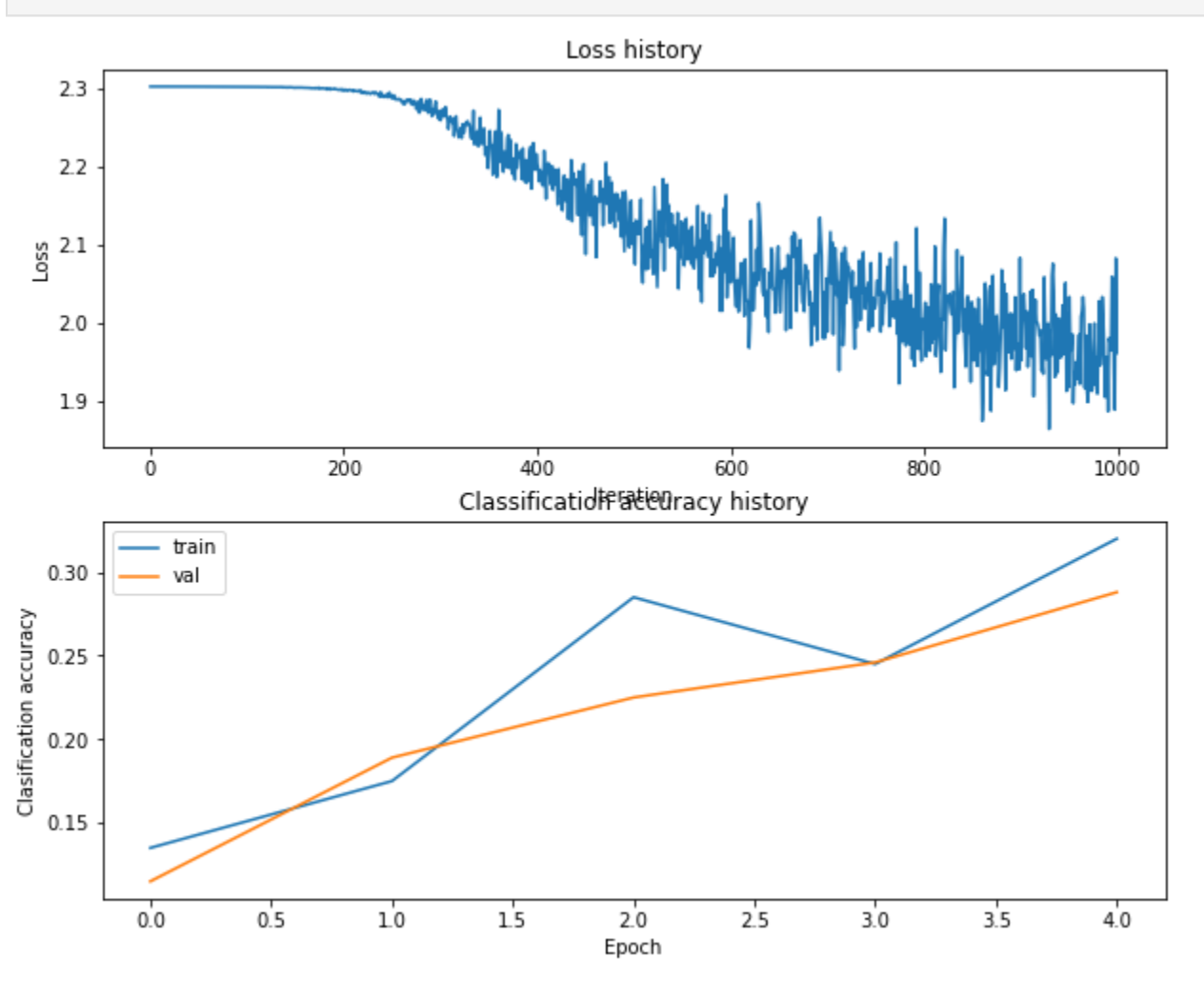
With the default parameters we provided above, you should get a validation accuracy of about 0.29 on the validation set. This isn't very good.

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 [48]:
```python
# 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()
```

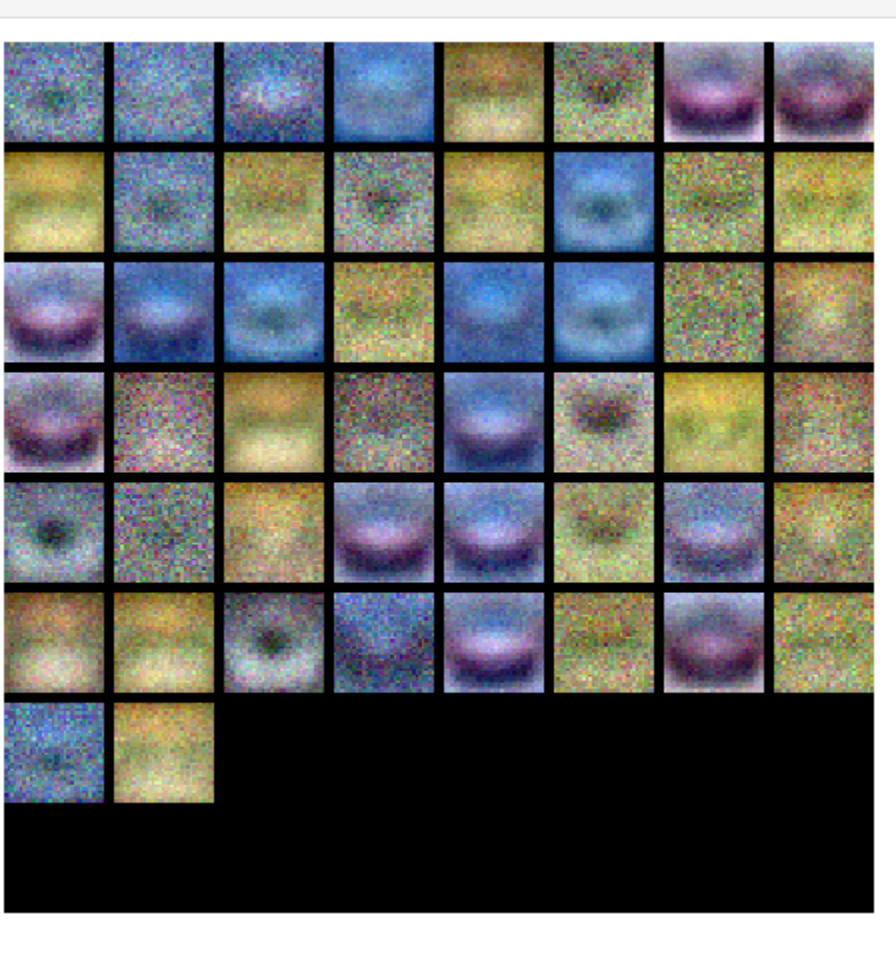


In [49]:
```python
from cs231n.vis_utils import visualize_grid

# 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)
```



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 [50]:
```python
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_laye
# :(
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                                #
#################################################################################
```

```
lr: 0.000700, reg: 0.750000, val_acc: 0.477000
lr: 0.000700, reg: 1.000000, val_acc: 0.477000
lr: 0.000700, reg: 1.250000, val_acc: 0.478000
lr: 0.000800, reg: 0.750000, val_acc: 0.471000
lr: 0.000800, reg: 1.000000, val_acc: 0.464000
lr: 0.000800, reg: 1.250000, val_acc: 0.470000
lr: 0.000900, reg: 0.750000, val_acc: 0.489000
lr: 0.000900, reg: 1.000000, val_acc: 0.474000
lr: 0.000900, reg: 1.250000, val_acc: 0.473000
lr: 0.001000, reg: 0.750000, val_acc: 0.479000
lr: 0.001000, reg: 1.000000, val_acc: 0.477000
lr: 0.001000, reg: 1.250000, val_acc: 0.467000
lr: 0.001100, reg: 0.750000, val_acc: 0.474000
lr: 0.001100, reg: 1.000000, val_acc: 0.491000
lr: 0.001100, reg: 1.250000, val_acc: 0.477000
Best validation accuracy: 0.491000
```

In [51]:
```python
# visualize the weights of the best network
show_net_weights(best_net)
```

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 [52]:
```python
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: **1, 2, 3**

Your explanation:

1: We can train our network on larger dataset so that we can tune the weights in detail. However, this has risk of over-fitting. Thus, it might not always be the answer. However, if the network is under-fitted, increasing the training dataset might stablize the network and increase the test accuracy.

2: Adding more hidden units would also add more depth to the weights. However, it also has the same risk as in 1. More layers = more parameters. Thus, there is a risk of over-fitting. Furthermore, naively adding more units can lead to bad gradient propagation such as vanishing gradient problem. It also increases time to train the network and network becomes more susceptible to noise and error.

3: In the case of over-fitting, our test accuracy might perform worse since the network becomes over specilized on the training dataset and it perform worse on generalization. Thus, to avoid over-fitting, we might increase the regularization strength to penalizes the weight matrices. Nonetheless, increasing the regularization strength more than needed might result our network to be under-fitted which might result in even worse performance.

Conclusion: In the end, three of these methods can be implemented together or alone for better performance. However, their usage might also negatively impact the neuron. Thus, caution and experimentation is needed for the most efficient utilization.

Comments: All in all, in this excerise, we have observed a simple two-layer network. Here, we have observed how to evaluate our network's performance such as plotting loss function and train / validation accuracies for debugging and got a glimpse of how to implement it on python. Moreover, it also indroduce us to concepts such as regularization for neural networks and under-fitting/over-fitting along with validation. At first, we have observed that the initial validation accuracy was pretty low. Therefore, we needed to tune the parameters (learning rate and regularization strength). By trial and error, we were able to find the best validation accuracy around **49.1%**. Visualizing the weight matrix also gave us insight on the network as weights matrix usually resembles the input/output images which was also the case in Question 3. Our test accuracy turned out to be **48.2%**. This is smaller than the training accuracy since our network weights were generated using the training dataset and not the test dataset. However, for better generalization, multiple methods can be applied individually or together. We can train on larger dataset for better training accuracy and in turn, it might lead to better test accuracy if our current network is under-fitted. We can add more hidden units for better optimizaition of the weight matrix. However, it migth lead to over-fitting. Finally, we can increase the regularization strength which would help our network to avoid over-fitting be might lead to under-fitting. We can combine these methods for negating their disadvantages and better test accuracy.

In the end, this exercise set the groundwork for me to implement multi-layer neural network in the future.