EEE443 - Neural Networks

Homework Assignment - 1

Berkan Ozdamar
21602353

Question 1

In this question, we are given training data x^n and outputs y^n, where n $\in$ [1 N]. Also, a single neuron that receives input from m input neurons with weights w_i, where i $\in$ [1 m]. The neuron predicts a binary output t belongs to *Class A* if t = 1 and *Class B* if t = -1.

We are told that the maximum a posteriori estimate for the network weights are obtained by solving the following problem:

$$L(W) = \underset{W}{\operatorname{argmin}} \sum_n (y^n - h(x^n, W))^2 + \beta \sum_i w_i^2$$

where W is the vector of weights w_i, β is a scalar constant and the h(.) is the output of the neuron. We are asked to derive the prior probability distribution of the network weights.

For understanding the relationship of posterior, prior and likelihood; we should discuss Bayes' Rule which is:

$$P(W|Y) = \frac{P(Y|W)P(W)}{P(Y)}$$

Since the denominator is a constant, we can write the equation as:

$$P(W|Y) \propto P(Y|W)P(W)$$

which corresponds to A posterior distribution of a random variable proportional to the multiplication of its likelihood and prior. Since we are asked to find maximum a posteriori estimate to network weights, the equation turns into;

$$\hat{W}_{MAP} \propto \underset{W}{\operatorname{argmax}} P(Y|W)P(W)$$

The given L function is an argmin function. First, this function can be turned into an argmax function by multiplying it with -1. So the function $\hat{L}$ becomes;

$$\hat{L}(W) = \underset{W}{\operatorname{argmax}} - (\sum_n (y^n - h(x^n, W))^2 + \beta \sum_i w_i^2)$$

Then, this summation function needs to be converted to multiplication of likelihood and prior. This transformation can be done with using the $f(x)$, where:

$$f(x) = e^x$$

Since f(x) function does not change the critical points, and since the $\hat{L}$ is an argmax function, f($\hat{L}$) is inspected.

$$f(\hat{L}(W)) = \underset{W}{\operatorname{argmax}} \quad e^{-(\sum_n (y^n - h(x^n, W))^2 + \beta \sum_i w_i^2)}$$

$$f(\hat{L}(W)) = \underset{W}{\operatorname{argmax}} \quad e^{-\sum_n (y^n - h(x^n, W))^2} e^{-\beta \sum_i w_i^2}$$

Assuming the data is from a Gaussian distribution, the likelihood and the prior can be written in the form:

$$Likelihood(Lh) = \frac{1}{A} e^{\frac{C}{2} \sum_n (y^n - h(x^n, W))^2}$$

$$Prior = Ae^{-\beta \sum_i w_i^2}$$

Moreover, the sum over all probability distribution is 1, so:

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

Then, the summation can divided into m:

$$A\int_{-\infty}^{\infty} e^{-\beta w_1^2} dw_1 \int_{-\infty}^{\infty} e^{-\beta w_1^2} dw_2 .... \int_{-\infty}^{\infty} e^{-\beta w_m^2} dw_m = 1$$

Also the definite integral:

$$\int_{-\infty}^{\infty} e^{-\beta w_k^2} dw_k = \frac{\sqrt{\pi}}{\sqrt{\beta}} \quad for \quad \beta \geq 0, \quad k \in \mathbb{R}$$

Therefore the equation then becomes:

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

$$A = (\frac{\beta}{\pi})^{\frac{m}{2}}$$

Hence, the prior can be written as:

$$Prior = (\frac{\beta}{\pi})^{\frac{m}{2}} e^{-\beta \sum_i w_i^2}$$

Question 2

In Question 2, we are required to design a neural network with a single hidden layer with four inputs and a single output neuron to implement:

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

To implement this logic function, a hidden layer with four inputs and a unipolar activation function(step function) is required.

Part a

In this part, we will derive a set of inequalities for each hidden unit. First, we will work on the logic function. Instead of xor gate, its combination with and gate and or gate will be used. k xor m can be written as:

$$(k \text{ xor } m) = (k \text{ and } \overline{m}) \text{ or } (\overline{k} \text{ and } m)$$

Thus the logic function $(x_1 \text{ or } \overline{x_2})$ xor $(\overline{x_3} \text{ or } \overline{x_4})$ can be updated as:

$$(x_1 + \overline{x_2}) \oplus (\overline{x_3} + \overline{x_4}) = (x_1 + \overline{x_2})\ (x_3 x_4) + (\overline{x_1} x_2)\ (\overline{x_3} + \overline{x_4})$$
$$= (x_1 x_3 x_4) + (\overline{x_2} x_3 x_4) + (\overline{x_1} x_2 \overline{x_3}) + (\overline{x_1} x_2 \overline{x_4})$$

In the question, it asks for us to use a hidden layer with four inputs, and it is viable with the equation we derived. Now, the inequalities for each hidden neuron will be derived individually.

1st Neuron

For the 1st Neuron, inequalities for $(x_1 x_3 x_4)$ will be derived. For that, the truth table for this logic function will be created.

x_1	x_3	x_4	**o_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**

Table 1: Truth Table for 1st Neuron

Also, the output(o_1) will be defined as:

$$o_1 = f(w_{11}x_1 + w_{13}x_3 + w_{14}x_4 + w_{15}), \quad \textit{where}$$

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

and w_{xy} represents the weight of xth Neuron and yth input.
Also, if the y=5 and the weight is w_{x5}, then this is the weight of the bias term of the xth Neuron.

Then, using both the truth table and the output, we can derive the inequalities as:

$$w_{15} < 0$$
$$w_{11} + w_{15} < 0$$
$$w_{13} + w_{15} < 0$$
$$w_{14} + w_{15} < 0$$

$$w_{11} + w_{13} + w_{15} < 0$$
$$w_{11} + w_{14} + w_{15} < 0$$
$$w_{13} + w_{14} + w_{15} < 0$$
$$w_{11} + w_{13} + w_{14} + w_{15} \geq 0$$

2nd Neuron

For the 2nd Neuron, inequalities for $(\overline{x_2}x_3x_4)$ will be derived. For that, the truth table for this logic function will be created.

x2	x3	x4	**o2**
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**

Table 2: Truth Table for 2nd Neuron

Also, the output(o_2) will be defined as:

$$o_2 = f(w_{22}x_2 + w_{23}x_3 + w_{24}x_4 + w_{25})$$

Then, using both the truth table and the output, we can derive the inequalities as:

$$w_{25} < 0$$
$$w_{22} + w_{25} < 0$$
$$w_{23} + w_{25} < 0$$
$$w_{24} + w_{25} < 0$$
$$w_{22} + w_{23} + w_{25} < 0$$
$$w_{22} + w_{24} + w_{25} < 0$$
$$w_{23} + w_{24} + w_{25} \geq 0$$
$$w_{22} + w_{23} + w_{24} + w_{25} < 0$$

3rd Neuron

For the 3rd Neuron, inequalities for $(\overline{x_1}x_2\overline{x_3})$ will be derived. For that, the truth table for this logic function will be created.

Also, the output(o_3) will be defined as:

$$o_3 = f(w_{31}x_1 + w_{32}x_2 + w_{33}x_3 + w_{35})$$

Then, using both the truth table and the output, we can derive the inequalities as:

x_1	x_2	x_3	$\mathbf{o_3}$
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**

Table 3: Truth Table for 3rd Neuron

$$w_{35} < 0$$
$$w_{31} + w_{35} < 0$$
$$w_{32} + w_{35} \geq 0$$
$$w_{33} + w_{35} < 0$$
$$w_{31} + w_{32} + w_{35} < 0$$
$$w_{31} + w_{33} + w_{35} < 0$$
$$w_{32} + w_{33} + w_{35} < 0$$
$$w_{31} + w_{32} + w_{33} + w_{35} < 0$$

4th Neuron

For the 4th Neuron, inequalities for $(\overline{x_1} x_2 \overline{x_4})$ will be derived. For that, the truth table for this logic function will be created.

x_1	x_2	x_4	$\mathbf{o_4}$
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**

Table 4: Truth Table for 4th Neuron

Also, the output(o_4) will be defined as:

$$o_4 = f(w_{41}x_1 + w_{42}x_2 + w_{44}x_4 + w_{45})$$

Then, using both the truth table and the output, we can derive the inequalities as:

$$w_{45} < 0$$
$$w_{41} + w_{45} < 0$$
$$w_{42} + w_{45} \geq 0$$
$$w_{44} + w_{45} < 0$$
$$w_{41} + w_{42} + w_{45} < 0$$
$$w_{41} + w_{44} + w_{45} < 0$$
$$w_{42} + w_{44} + w_{45} < 0$$
$$w_{41} + w_{42} + w_{44} + w_{45} < 0$$

Output Neuron

Output Neuron is the and gate that connects the four hidden Neurons we have constructed. The outputs of those four hidden Neurons will be the inputs of this final output Neuron. Thus, we can create the truth table for the final output as:

o_1	o_2	o_3	o_4	$\mathbf{o_f}$
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**

Table 5: Truth Table for Output Neuron

Also, the output(o_f) will be defined as:

$$o_f = f(w_{51}o_1 + w_{52}o_2 + w_{53}o_3 + w_{54}o_4 + w_{55})$$

Then, using both the truth table and the output, we can derive the inequalities as:

$$w_{55} < 0$$

$$w_{51} + w_{55} \geq 0$$

$$w_{52} + w_{55} \geq 0$$

$$w_{53} + w_{55} \geq 0$$

$$w_{54} + w_{55} \geq 0$$

$$w_{51} + w_{52} + w_{55} \geq 0$$

$$w_{51} + w_{53} + w_{55} \geq 0$$

$$w_{51} + w_{54} + w_{55} \geq 0$$

$$w_{52} + w_{53} + w_{55} \geq 0$$

$$w_{52} + w_{54} + w_{55} \geq 0$$

$$w_{53} + w_{54} + w_{55} \geq 0$$

$$w_{51} + w_{52} + w_{53} + w_{55} \geq 0$$

$$w_{51} + w_{52} + w_{54} + w_{55} \geq 0$$

$$w_{51} + w_{53} + w_{54} + w_{55} \geq 0$$

$$w_{52} + w_{53} + w_{54} + w_{55} \geq 0$$

$$w_{51} + w_{52} + w_{53} + w_{54} + w_{55} \geq 0$$

Part b

In this part, we are asked to choose a particular weight and bias matrices and run the network with those matrices to show that network achieves 100% performance in implementing the logic function.

First, to try all possibilities, I have declared the inputs vector which is a size of (16,5) vector. The (16,0:4) part of the input vector represents the every possible input combination. And the 5th column is all 1, which represents the bias vector. The implementation of the inputs vector in Python is as:

[1]:
```
#Part B

import numpy as np


inputs = np.zeros((16,5))

# For generating the all possible input values, concatenated with the bias vector.␣
 ↪5th column is all 1, which
# represents the bias vector.
def generateInputVector(inputVector):
    for i in range(np.shape(inputVector)[0]):
        temp = bin(i).split('b')
        temp = temp[-1]
        rowVector = [int(x) for x in str(temp)]
        rowVector.append(1)
        sizeRowVector = len(rowVector)
        inputVector[i, (np.shape(inputVector)[1]) - sizeRowVector:] = rowVector[:]
    return inputVector

inputs = generateInputVector(inputs)
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.]]
```

Then, the step function is declared as requested in the question.

[2]:
```
# The activation function which the the unit step function.
def unitStepFunction(k):
    k = np.where(k<0, 0,1)
    return k
```

I have designed the weight and bias matrix for hidden layer, which fits the inequalities described in **Part a**. The strategy was to pick random integers that fits the inequalities. Thus the weights and bias matrices for hidden layer are as follows:

$$weights_{hidden} = \begin{pmatrix} 5 & 0 & 5 & 5 \\ 0 & -4 & 5 & 5 \\ -6 & 11 & -5 & 0 \\ -5 & 13 & 0 & -6 \end{pmatrix} \quad , \quad bias_{hidden} = \begin{pmatrix} -13 \\ -9 \\ -10 \\ -11 \end{pmatrix}$$

Then, instead of using two different matrices, weights and bias matrices are concatenated in one matrix called hiddenweights.

$$hiddenweights = W_h = \begin{pmatrix} 5 & 0 & 5 & 5 & -13 \\ 0 & -4 & 5 & 5 & -9 \\ -6 & 11 & -5 & 0 & -10 \\ -5 & 13 & 0 & -6 & -11 \end{pmatrix}$$

For the output neuron, the weights and bias is also calculated by picking random integers that fits the inequalities. The output neuron's weights and bias is as:

$$weights_{output} = \begin{pmatrix} 2 & 3 & 5 & 7 \end{pmatrix} \quad , \quad bias_{output} = \begin{pmatrix} -1 \end{pmatrix}$$

Then, instead of using two different matrices, weights and bias matrices are concatenated in one matrix called outputweights.

$$outputweights = W_o = \begin{pmatrix} 2 & 3 & 5 & 7 & -1 \end{pmatrix}$$

[3]:
```
# Hand calculated weights for hidden layer and output layer.

hiddenweights = np.array([[5, 0, 5, 5, -13],
                          [0, -4, 5, 5, -9],
                          [-6, 11, -5, 0, -10],
                          [-5, 13, 0, -6, -11]])

outputweights = np.array([2, 3, 5, 7, -1])
```

Then the network is tested with the given weights and inputs vector created above. The equation for network is as:

$$o_{hidden} = f(W_h.inputs^{\mathrm{T}})$$

Then add a bias term to o_{hidden} to create the W_o and calculate the final output of the network as:

$$o_{final} = f(o_{hidden}.{W_o}^{\mathrm{T}}), \quad where$$

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

Python implementation is below:

[4]:
```
# Testing the network with given weights.

hiddenOutsTemp = unitStepFunction(inputs.dot(hiddenweights.T))

# Again, adding the bias vector before going into output neuron.
bias = np.ones((16,1))
hiddenOuts = np.concatenate((hiddenOutsTemp, bias),axis =1)
print(hiddenOuts)
```

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

[5]:
```
# Final output for the network with given weights.

outputs = unitStepFunction(hiddenOuts.dot(outputweights.T))
print(outputs)
```

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

The outputs vector we have acquired is the network's final output. To check whether the network performs 100%, we will implement the logic function directly and test the outcomes. The implementation of the logic function and the outcome is as:

[6]:
```
# Implementation of the logic function. Therefore, we can test the performance of␣
 ↪the network above.

def xor_gate(a, b):
    result = (a and (not b)) or ((not a) and b)
    return result

def logicFunction(inputVector):
    result = list()
    for i in range(np.shape(inputVector)[0]):
        temp = xor_gate(inputVector[i,0] or (not inputVector[i,1]), (not␣
 ↪inputVector[i,2]) or (not inputVector[i,3]))
        result.append(temp)
    return result
```

[7]:
```
# The outputs from the logic function. We will check this output with the ones we␣
 ↪have found above.
outputsCheck = logicFunction(inputs[:,0:4])
print(outputsCheck)
```

```
[False, False, False, True, True, True, True, False, False, False, False, True,
False, False, False, True]
```

The outputsCheck vector is the outcome of the logic function. Here, False represents 0 and True represents 1. This way we can check the outcomes from network and the logic function. They accuracy calculation function is accuracyCalc is defined below. Basically, what it does is to check how many elements are matching and converts it to percentage value.

[8]:
```
# For calculating the accuracy of the network, accuracyCalc function is defined.

def accuracyCalc(x, y):
    result = 0
    count = 0
    size = np.shape(x)[0]
    sentence = 'The accuracy of the model is: '
    for i in range(size):
        if (x[i] == y[i]):
            count = count +1
    result = (count / size) * 100

    return result
```

[9]:
```
# The accuracy result between the network and the logic function itself.

accuracy = accuracyCalc(outputs, outputsCheck)
print('The accuracy of the model is: ' + str(accuracy))
```

```
The accuracy of the model is: 100.0
```

Here, we can see that the network's performance is 100% with the given weights.

Part c

In this part, we are asked to find update the network so that it can perform more robustly under noisy conditions. The weights we have found in the **Part b** are random integers that satisfies the inequalities. Since there was no noise in the inputs, the network performed perfectly. But that system may not work so good in a noisy environment.

In order to crate the most robust system, weights should be selected as either 1 or -1 for symmetry and orthogonality. That way, noise would affect the network less and system would become more robust. Since we have selected the -1 for inverted inputs, 0 for inputs that are not necessary for the current neuron and 1 for input itself, biases should be updated according to inequalities. Thus, we will inspect inequalities one more time for each neuron to update bias accordingly.

1st Neuron

For the 1st Neuron, inequalities for $(x_1x_3x_4)$ will be inspected again. But this time, the weights will be $w_{11} = 1$, $w_{13} = 1$, and $w_{14} = 1$ since they are all positive inputs to the 1st Neuron.

$$
\begin{gathered}
w_{15} < 0 \\
w_{11} + w_{15} < 0 \\
w_{13} + w_{15} < 0 \\
w_{14} + w_{15} < 0 \\
w_{11} + w_{13} + w_{15} < 0 \\
w_{11} + w_{14} + w_{15} < 0 \\
w_{13} + w_{14} + w_{15} < 0 \\
w_{11} + w_{13} + w_{14} + w_{15} \geq 0
\end{gathered}
$$

Substituting $w_{11} = 1$, $w_{13} = 1$, and $w_{14} = 1$ into the inequalities above, we get:

$$
\begin{gathered}
w_{15} < 0 \\
w_{15} < -1 \\
w_{15} < -2 \\
w_{15} \geq -3
\end{gathered}
$$

So, the final equation for bias term w_{15} is:

$$-2 > w_{15} \geq -3$$

Thus, middle point -2.5 is selected for the bias term of 1st Neuron.

$$w_{15} = -2.5$$

2nd Neuron

For the 2nd Neuron, inequalities for $(\overline{x_2}x_3x_4)$ will be inspected again. The values of input weights are:

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

The inequalities are:

$$w_{25} < 0$$

$$w_{22} + w_{25} < 0$$
$$w_{23} + w_{25} < 0$$
$$w_{24} + w_{25} < 0$$
$$w_{22} + w_{23} + w_{25} < 0$$
$$w_{22} + w_{24} + w_{25} < 0$$
$$w_{23} + w_{24} + w_{25} \geq 0$$
$$w_{22} + w_{23} + w_{24} + w_{25} < 0$$

Substituting w_{22} = -1, w_{23} = 1, and w_{24} = 1 into the inequalities above, we get:

$$w_{25} < 0$$

$$w_{25} < 1$$
$$w_{25} < -1$$
$$w_{25} \geq -2$$

So, the final equation for bias term w_{25} is:

$$-1 > w_{25} \geq -2$$

Thus, middle point -1.5 is selected for the bias term of 2nd Neuron.

$$w_{25} = -1.5$$

3rd Neuron

For the 3rd Neuron, inequalities for $(\overline{x_1}x_2\overline{x_3})$ will be inspected again. The values of input weights are:

$$w_{31} = -1, \quad w_{32} = 1, \quad and \quad w_{33} = -1.$$

The inequalities are:

$$\begin{gathered} w_{35} < 0 \\ w_{31} + w_{35} < 0 \\ w_{32} + w_{35} \geq 0 \\ w_{33} + w_{35} < 0 \\ w_{31} + w_{32} + w_{35} < 0 \\ w_{31} + w_{33} + w_{35} < 0 \\ w_{32} + w_{33} + w_{35} < 0 \\ w_{31} + w_{32} + w_{33} + w_{35} < 0 \end{gathered}$$

Substituting w_{31} = -1, w_{32} = 1, and w_{33} = -1 into the inequalities above, we get:

$$\begin{gathered} w_{35} < 0 \\ w_{35} < 1 \\ w_{35} < 2 \\ w_{35} \geq -1 \end{gathered}$$

So, the final equation for bias term w_{35} is:

$$0 > w_{35} \geq -1$$

Thus, middle point -0.5 is selected for the bias term of 3rd Neuron.

$$w_{35} = -0.5$$

4th Neuron

For the 4th Neuron, inequalities for $(\overline{x_1}x_2\overline{x_4})$ will be inspected again. The values of input weights are:

$$w_{41} = -1, \quad w_{42} = 1, \quad and \quad w_{44} = -1.$$

The inequalities are:

$$\begin{gathered} w_{45} < 0 \\ w_{41} + w_{45} < 0 \\ w_{42} + w_{45} \geq 0 \\ w_{44} + w_{45} < 0 \\ w_{41} + w_{42} + w_{45} < 0 \\ w_{41} + w_{44} + w_{45} < 0 \\ w_{42} + w_{44} + w_{45} < 0 \\ w_{41} + w_{42} + w_{44} + w_{45} < 0 \end{gathered}$$

Substituting w_{41} = -1, w_{42} = 1, and w_{44} = -1 into the inequalities above, we get:

$$w_{45} < 0$$
$$w_{45} < 1$$
$$w_{45} < 2$$
$$w_{45} \geq -1$$

So, the final equation for bias term w_{45} is:

$$0 > w_{45} \geq -1$$

Thus, middle point -0.5 is selected for the bias term of 4th Neuron.

$$w_{45} = -0.5$$

Output Neuron

For output Neuron, as we have already discussed in **Part a**, we have found the inputs from the outputs of four hidden layer Neurons, which are o_1, o_2, o_3 and o_4. Their corresponding weights will be:

$$w_{51} = 1, \quad w_{52} = 1, \quad w_{53} = 1, \quad and \quad w_{54} = 1.$$

The inequalities for output neuron is:

$$w_{55} < 0$$
$$w_{51} + w_{55} \geq 0$$
$$w_{52} + w_{55} \geq 0$$
$$w_{53} + w_{55} \geq 0$$
$$w_{54} + w_{55} \geq 0$$
$$w_{51} + w_{52} + w_{55} \geq 0$$
$$w_{51} + w_{53} + w_{55} \geq 0$$
$$w_{51} + w_{54} + w_{55} \geq 0$$
$$w_{52} + w_{53} + w_{55} \geq 0$$
$$w_{52} + w_{54} + w_{55} \geq 0$$
$$w_{53} + w_{54} + w_{55} \geq 0$$
$$w_{51} + w_{52} + w_{53} + w_{55} \geq 0$$
$$w_{51} + w_{52} + w_{54} + w_{55} \geq 0$$
$$w_{51} + w_{53} + w_{54} + w_{55} \geq 0$$
$$w_{52} + w_{53} + w_{54} + w_{55} \geq 0$$
$$w_{51} + w_{52} + w_{53} + w_{54} + w_{55} \geq 0$$

Substituting w_{51} = 1, w_{52} = 1, w_{53} = 1, and w_{54} = 1 into the inequalities above, we get:

$$w_{55} < 0$$

$$w_{55} \geq -1$$
$$w_{55} \geq -2$$
$$w_{55} \geq -3$$
$$w_{55} \geq -4$$

So, the final equation for bias term w_{55} is:

$$0 > w_{55} \geq -1$$

Thus, middle point -0.5 is selected for the bias term of output Neuron.

$$w_{55} = -0.5$$

So, the robust weights and biases are as:

$$robustWeights_{hidden} = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & -1 & 1 & 1 \\ -1 & 1 & -1 & 0 \\ -1 & 1 & 0 & -1 \end{pmatrix} \quad , \quad robustBias_{hidden} = \begin{pmatrix} -2.5 \\ -1.5 \\ -0.5 \\ -0.5 \end{pmatrix}$$

Then, instead of using two different matrices, weights and bias matrices are concatenated in one matrix called robustHiddenweights.

$$robustHiddenweights = W_{rh} = \begin{pmatrix} 1 & 0 & 1 & 1 & -2.5 \\ 0 & -1 & 1 & 1 & -1.5 \\ -1 & 1 & -1 & 0 & -0.5 \\ -1 & 1 & 0 & -1 & -0.5 \end{pmatrix}$$

For the output neuron, the weights and bias is also calculated by picking random integers that fits the inequalities. The output neuron's weights and bias is as:

$$robustWeights_{output} = \begin{pmatrix} 1 & 1 & 1 & 1 \end{pmatrix} \quad , \quad robustBias_{output} = \begin{pmatrix} -0.5 \end{pmatrix}$$

Then, instead of using two different matrices, weights and bias matrices are concatenated in one matrix called robustOutputweights.

$$robustOutputweights = W_{ro} = \begin{pmatrix} 1 & 1 & 1 & 1 & -0.5 \end{pmatrix}$$

The implementation of robust weights matrices are:

[10]:
```
# Part C

# Robust Weights
# w_rh = robust weights for hidden layer
# w_ro = robust weights for output layer

w_rh = np.array([[1, 0, 1, 1, -2.5],
                           [0, -1, 1, 1, -1.5],
                           [-1, 1, -1, 0, -0.5],
                           [-1, 1, 0, -1, -0.5]])
```

```
w_ro = np.array([1, 1, 1, 1, -0.5])
```

Part d

In this part, we are asked to check whether the robust weights perform better than the weights from **Part b** under noisy conditions. First, 400 input samples will be generated from concatenating 25 samples from each input vector. Then, a noise vector will be created from a Gaussian with 0 mean and 0.2 standard deviation. Before adding the Gaussian noise to the inputs, the output will be gathered from the 400 samples to check the accuracy later. And then Gaussian noise has added to 400 input samples by linearly combining the inputs and noise matrices. The Python implementation is below:

[11]:
```
# Part D

# Generate 400 input samples by concatenating 25 samples from each input.
inputsWithNoise = np.zeros((400,5))
for k in range(25):
    for i in range(np.shape(inputs)[0]):
        inputsWithNoise[(k*16)+i,:] = inputs[i,:]

# Then check the outputs of the inputsWithNoise.
outputsCheck_D = logicFunction(inputsWithNoise[:,0:4])

# Create a gaussian noise with 0 mean and 0.2 std. Then add this noise to the␣
 ↪inputsWithNoise array.
# np.random.seed(7) is for getting the same output each run.
np.random.seed(7)
gaussianNoise = np.random.normal(loc=0, scale=0.2, size=1600).reshape(400, 4)
inputsWithNoise[:, 0:4] += gaussianNoise
```

Then the noise added inputs are tested with the random weights from **Part b** and the robust weights from **Part c**. The implementation is as below:

[12]:
```
# Then test the inputsWithNoise with the random weights network.

outputTemp = unitStepFunction(inputsWithNoise.dot(hiddenweights.T))
bias = np.ones((400,1))
hiddenOutsTemp = np.concatenate((outputTemp, bias),axis =1)

random_network_output = unitStepFunction(hiddenOutsTemp.dot(outputweights.T))
```

[13]:
```
# Then test the inputsWithNoise with the robust weights network.

robustOutputTemp = unitStepFunction(inputsWithNoise.dot(w_rh.T))
bias = np.ones((400,1))
robustHiddenOuts = np.concatenate((robustOutputTemp, bias),axis =1)

robust_network_output = unitStepFunction(robustHiddenOuts.dot(w_ro.T))
```

Then the accuracies for both the random weighted network and robust network is checked. The accuracy values for both network is given below:

[14]:
```
# Accuracy of the random weight network.

accuracy = accuracyCalc(random_network_output, outputsCheck_D)
print('The accuracy of the random weighted network is: ' + str(accuracy))
```

```
The accuracy of the random weighted network is: 86.75
```

[15]:
```
# Accuracy of the robust network.

accuracy = accuracyCalc(robust_network_output, outputsCheck_D)
print('The accuracy of the robust network is: ' + str(accuracy))
```

```
The accuracy of the robust network is: 90.0
```

The accuracy of the robust network is 90.0% whereas the accuracy of the non-robust network is the 86.75%. The accuracy of the network is improved significantly by selecting the weights more robustly. But for even the robust network, the accuracy is not 100% since there is a Guassian noise with 0.2 standard deviation. If the standard deviation is would be smaller, then the accuracy of the network would be higher than 90%.

Question 3

In this question, we would like to process images of alphabet letters with a perceptron. The images are already divided into train images, test images, train labels and test labels.

Part a

In this part, a sample image from each class is visualized. And correlation coefficients between pairs of sampled images are displayed.

First, the training images, testing images, training labels and testing labels are acquired.

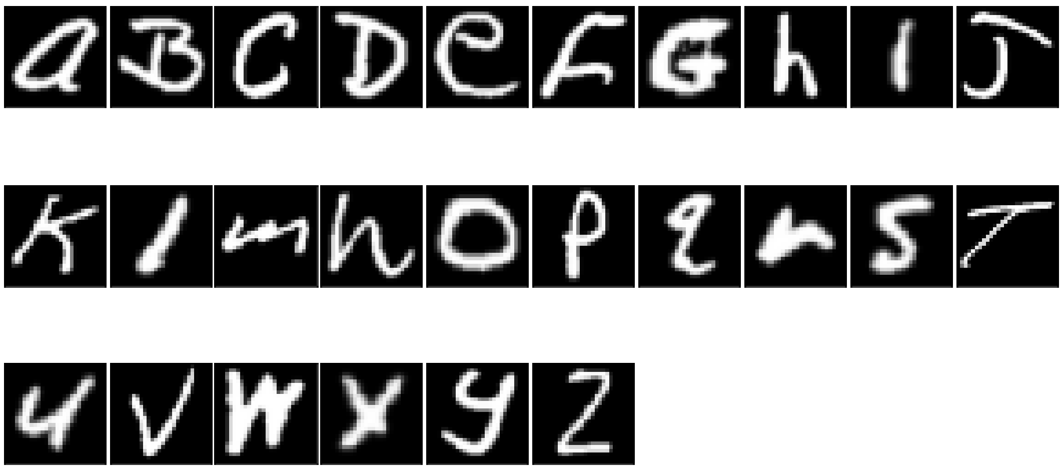
[1]:
```
import numpy as np
import h5py
import matplotlib.pyplot as plt
```

[2]:
```
# Part A

f = h5py.File('assign1_data1.h5', 'r')
dataKeys = list(f.keys())
print('The data keys are:' + str(dataKeys))

# Gathering the  train images, test images, train labels and test labels.
testims = f['testims']
testlbls = f['testlbls']
trainims = f['trainims']
trainlbls = f['trainlbls']
print('The size of testims is: ' + str(np.shape(testims)))
print('The size of testlbls is: ' + str(np.shape(testlbls)))
print('The size of trainims is: ' + str(np.shape(trainims)))
print('The size of trainlbls is: ' + str(np.shape(trainlbls)))
```

```
The data keys are:['testims', 'testlbls', 'trainims', 'trainlbls']
The size of testims is: (1300, 28, 28)
The size of testlbls is: (1300,)
The size of trainims is: (5200, 28, 28)
The size of trainlbls is: (5200,)
```

For visualization of the sample image from each class, uniqueClass array is defined and stored first sample from each class. For the correlation matrix, uniqueClass2 array is also created and stored the second sample from each class. The python implementation of the visualization is given below:

[3]:
```
figureNo = 0
plt.figure(figureNo, figsize=(8,8))
label = 1

# Create 2 arrays for gathering 1 sample from each unique class.
# But these 2 arrays will gather different sample from each class.
uniqueClass = list()
uniqueClass2 = list()

# For visualizing a sample from each class.
# Also, store desired samples to uniqueClassX arrays.
for i, value in enumerate(trainlbls):
    if (label == value):
        uniqueClass.append(trainims[i])
        uniqueClass2.append(trainims[i+1])
        label += 1
        ax1 = plt.subplot(3, 10, label-1)
        ax1.imshow(trainims[i].T, cmap = 'gray')
        ax1.set_yticks([])
        ax1.set_xticks([])

plt.subplots_adjust(wspace=0.05, hspace=0.05, left=0, right=1, bottom=0, top=0.5)
plt.show()
```



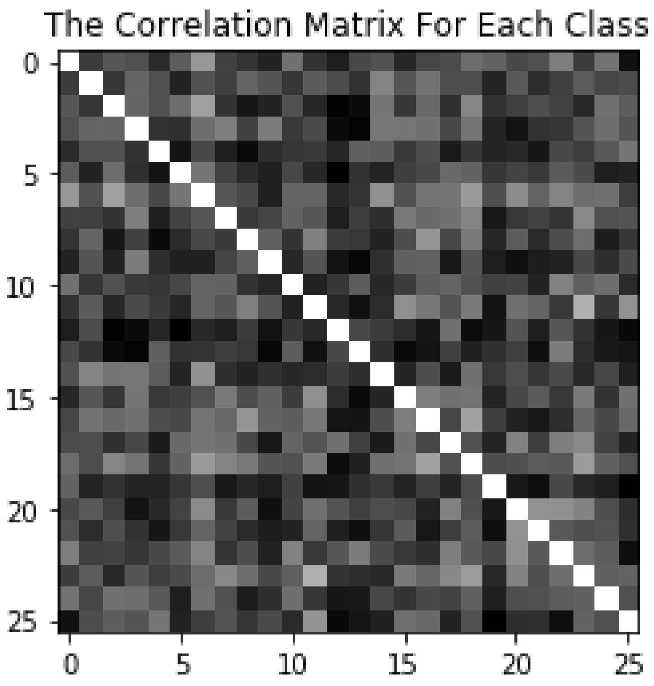
For correlation matrix, Numpy's corrcoef function is used. This function returns the Pearson correlation coefficients which is:

$$\rho = \frac{\mathrm{cov}(X,Y)}{\sigma_x \sigma_y}$$

First, correlation between samples from each class is calculated only using the uniqueClass array we have discussed earlier. The python implementation is below:

[4]:
```
# Create the Correlation Matrix for same sample from each class

uniqueClass = np.asarray(uniqueClass)
uniqueClass = uniqueClass.reshape(np.shape(uniqueClass)[0], 28*28)
correlations = list()
for i in range(np.shape(uniqueClass)[0]):
    for j in range(np.shape(uniqueClass)[0]):
        correlations.append(np.corrcoef(uniqueClass[i], uniqueClass[j])[1,0])
```

[5]:
```
# The visualization of the Correlation Matrix.

correlations = np.asarray(correlations)
correlations = correlations.reshape(26,26)
figureNo += 1
plt.figure(figureNo)
plt.title('The Correlation Matrix For Each Class')
plt.imshow(correlations, cmap = 'gray')
plt.show()
```



The coefficients of the correlation vary between 0 and 1. From the figure, it can be seen that the diagonal is white which corresponds to 1. This result is expected since the diagonal corresponds the same image from the same class. Thus, we can express the Pearson correlation coefficients for the diagonal as:

$$\rho = \frac{\text{cov}(X,Y)}{\sigma_x \sigma_y}, \quad \textit{where} \quad Y = X$$

$$\rho = \frac{\text{cov}(X,X)}{\sigma_x \sigma_x} = \frac{\text{var}(X,X)}{\sigma_x \sigma_x} = \frac{\text{var}(X,X)}{\text{var}(X,X)} = 1$$

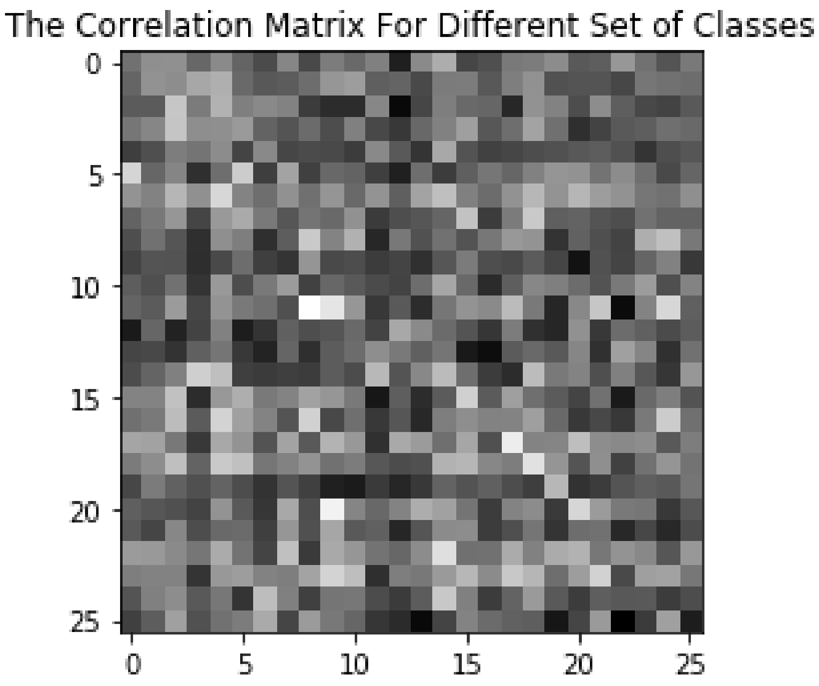
Thus, it can be seen that the within class variability is lower than the across class variability. Moreover, some across classes have low correlation between them(since the pixel is dark) but some classes have some correlation(since the pixel is gray). Those gray spots may make the classification harder since it means correlation between them. Those classes which have some correlation between them can be the letters that look alike such as i and l.

Then the correlation between different samples from each class is calculated with the uniqueClass and uniqueClass2 matrices. This way, the correlation between the different samples from the same class can be observed. The python implementation and the visualization is also below:

[6]:
```
# Create the Correlation Matrix for different sample from each class

uniqueClass2 = np.asarray(uniqueClass2)
uniqueClass2 = uniqueClass2.reshape(np.shape(uniqueClass2)[0], 28*28)
corrs_differentSet = list()
for i in range(np.shape(uniqueClass)[0]):
    for j in range(np.shape(uniqueClass)[0]):
        corrs_differentSet.append(np.corrcoef(uniqueClass[i], uniqueClass2[j])[1,0])
```

[7]:
```
# The visualization of the Correlation Matrix.

corrs_differentSet = np.asarray(corrs_differentSet)
corrs_differentSet = corrs_differentSet.reshape(26,26)
```

```
figureNo += 1
plt.figure(figureNo)
plt.title('The Correlation Matrix For Different Set of Classes')
plt.imshow(corrs_differentSet, cmap = 'gray')
plt.show()
```



Here, we see that the diagonal has a darker tone. This corresponds that the different samples from the same class has some lower correlation than the same samples which is expected. However, applying a classification considering the correlations will not be efficient.

Part b

In this part, we are asked to implement a single layer perceptron with an output neuron for each digit. Also, initial network weights and bias terms will be randomly drawn from Guassian distribution $\mathcal{N}(0, 0.01)$. The activation function is the sigmoid function.

Since an image is the size of (28,28) matrix, each input has $28x28 = 784$ pixel values. Also, there are 26 classes, so the weights matrix has the shape (26,784) and the bias matrix has the shape (26,1). All values are set from Guassian distribution $\mathcal{N}(0, 0.01)$.

Then, the labels for each class is converted to one-hot encoding. That way, we can represent the output with the size of the classes(26) and each neuron can represent a different letter. In other words, the label of a class is represented with 26 digits which each digit represents different class. Thus, the label class for training data is converted to a matrix size of (5200,26) from (5200,1) since there are 5200 training data. The Python implementation is below:

```
[8]: # Part B

# For getting the same result each run.
np.random.seed(8)

# Since each image has dimensions of 28*28 and there are 26 classes, weights vector␣
 ↪will be of size [26, 28*28]
```

```
# And there is a bias for each class so the shape of bias vector is [26,1]
# Values for both weights and bias vectors are random numbers from a Gaussian␣
 ↪Distribution with 0 mean and 0.01 std.
weights = np.random.normal( loc=0, scale=0.01, size=(26,28*28))
bias = np.random.normal( loc=0, scale=0.01, size=(26,1))

# The labels vector's values are from 1 to 26. This could cause errors so, they␣
 ↪will be reshaped into one-hot encoding.
trainlabels = np.zeros((26, np.shape(trainlbls)[0]))
testlabels  = np.zeros((26, np.shape(testlbls)[0]))

for i, values in enumerate(trainlbls):
    trainlabels[26 - int(values),i] = 1

for i, values in enumerate(testlbls):
    testlabels[26 - int(values),i] = 1

# For rows I wanted to keep each sample. For both testlabels and trainlabels, I␣
 ↪have taken the transpose
# to keep the arrays (1300,26) and (5200,26)
trainlabels = trainlabels.T
testlabels = testlabels.T

# Also, make the train and test images a numpy array for eaze.
trainimages = np.asarray(trainims)
testimages = np.asarray(testims)

# Example of class 1(A,a) and class 26(Z,z) representations
print('Class 1 (A,a) is: ' + str(trainlabels[0]))
print('Class 26 (Z,z) is: ' + str(trainlabels[5199]))
```

```
Class 1 (A,a) is: [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.]
Class 26 (Z,z) is: [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.]
```

Here, representation of the class1(A,a) and class26(Z,z) with one-hot encoding is shown.

Then the sigmoid function is defined as the activation function of the network. Sigmoid function can be expressed as:

$$s(x) = \frac{1}{1+e^{-x}}$$

The Python implementation is as below:

[9]:
```
# Sigmoid function will be used in training the perceptron model.
def sigmoid(x):
    result = 1 / (1 + np.exp(-x))
    return result
```

Network is updated according to gradient-descent learning rule, and the Mean-Squared Error(MSE) of the each iteration is recorded. MSE function can be expressed as:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(Y_i - \hat{Y}_i)^2 = \frac{1}{N}\sum_{i=1}^{N} E$$

Python implementation of the MSE is as below:

[10]:
```
# Mean Square Error as the loss function
def MSE(error):
    result = np.mean(error**2)
    return result
```

In the MSE function, Y is the label and $\hat{Y}$ is the prediction. Since, sigmoid is used in the network, the prediction is:

$$\hat{Y} = s(WX - B)$$

where s is the sigmoid function. Thus, the E can be written as:

$$E = (Y - s(WX - B))(Y - s(WX - B))^T$$

Then, for updating weights and bias, gradient descent update rule can be written as:

$$\hat{W} = W - \mu \frac{\partial E}{\partial W}, \qquad \hat{B} = B - \mu \frac{\partial E}{\partial B}$$

where μ is the learning rate. If we take the partial derivative of the loss function(E) with respect to W and B respectively,

$$\frac{\partial E}{\partial W} = -2(Y - s(WX - B))(s(WX - B))(1 - s(WX - B))X^T$$

$$\frac{\partial E}{\partial B} = 2(Y - s(WX - B))(s(WX - B))(1 - s(WX - B))$$

The weights and bias is updated with the update rule shown above. Moreover, the a random image is selected from training data and flattened so that the selected image has size of (1,28*28) which is (1,784). For each iteration, weights and bias is updated and MSE is stored. The Python implementation of the perceptronTrain is as below:

[11]:
```
def perceptronTrain(train_data, labels, weights, bias, iterations, learningRate):

    MSE_loss = list()

    for i in range(iterations):

        # Selecting a random image and its label.
        random = np.random.randint(0,5200,1)
        selectedImage = train_data[random,:]
        selectedLabel = labels[random,:]

        # Normalization (For setting each pixel between 0 and 1. )
        selectedImage = selectedImage / np.amax(selectedImage)

        # Calculation of the weighted sum and its sigmoid.
        weightedSum = weights.dot(selectedImage.T) - bias
        y_ = sigmoid(weightedSum)
        y_difference = selectedLabel.T - y_

        # Update Rules

        update = y_difference*y_*(1-y_)
        weight_update = -2*learningRate * (update).dot(selectedImage)
        bias_update = 2*learningRate * update
        weights = weights - weight_update
        bias = bias - bias_update

        # Loss Function
        MSE_loss.append(MSE(y_difference))

    return weights, bias, MSE_loss
```

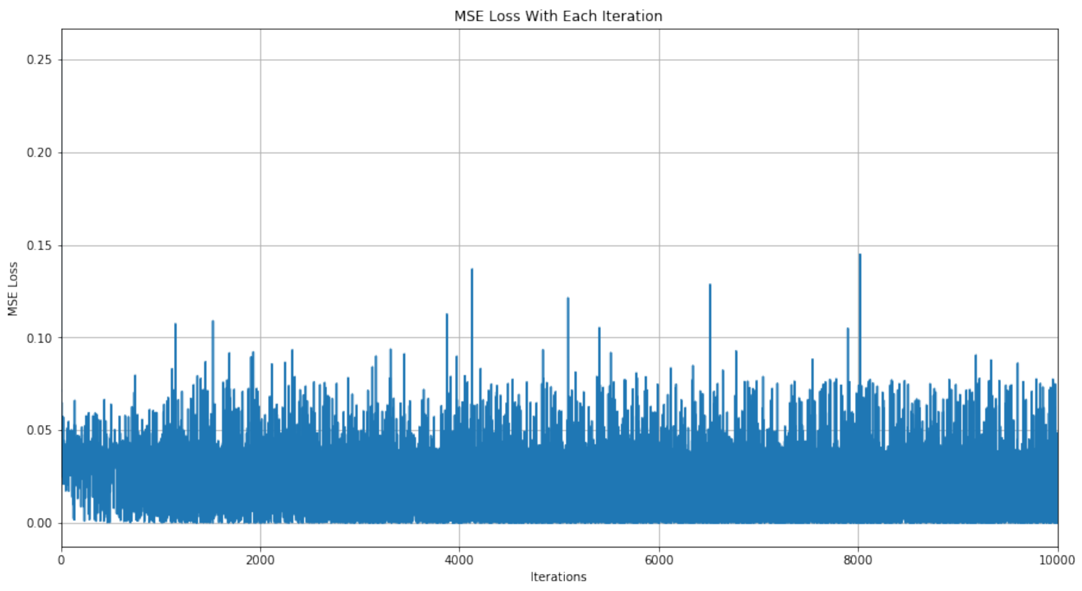
For finding the optimal learning rate, a grid search is done. Then, the learning rate which minimizes the final value of the MSE is selected as the optimal learning rate. The Python implementation is as:

[12]:
```
iterations = 10000

# Created the learningRateArray to select the optimal learning rate.
learningRateArray = np.arange(0.01, 0.2, 0.01)

# Flatten the image so that we can represent a neuron for every pixel.
trainimages_flat = trainimages.reshape(5200,28*28)

# Selecting the best learning rate amongs the learningRateArray.

# For each learning rate, selection is made by comparing the mean of the last item
# of the MSE loss.

# NOTE : Takes about a minute to run.
best = np.inf
for learningRate in learningRateArray:
    np.random.seed(8)
    weights = np.random.normal( loc=0, scale=0.01, size=(26,28*28))
    bias = np.random.normal( loc=0, scale=0.01, size=(26,1))
    weights_best, bias_best, MSE_loss_best = perceptronTrain(trainimages_flat,␣
 ↪trainlabels, weights, bias, iterations, learningRate)
    if ( np.mean(MSE_loss_best[-1]) < best):
        best = np.mean(MSE_loss_best[-1])
        learningRateBest = learningRate

print("The optimal learning rate is %.2f" % learningRateBest)


# Training with the optimal learning rate.
np.random.seed(8)
weights = np.random.normal( loc=0, scale=0.01, size=(26,28*28))
bias = np.random.normal( loc=0, scale=0.01, size=(26,1))
weights_best, bias_best, MSE_loss_best = perceptronTrain(trainimages_flat,␣
 ↪trainlabels, weights, bias, iterations, learningRateBest)
```

```
The optimal learning rate is 0.16
```

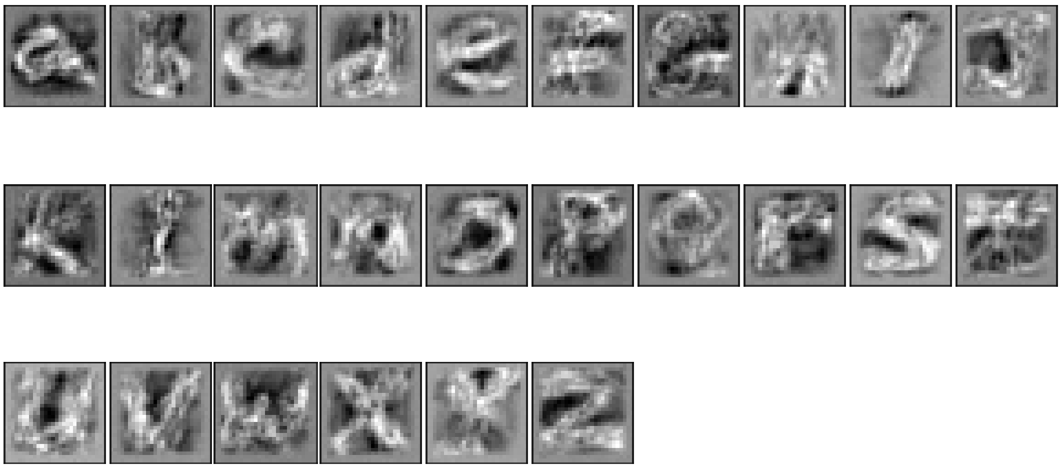
Then the final weights and bias is stored. Also, MSE is collected with each iteration for 10000 iterations. The visualization of the MSE loss for 10000 iterations with learning rate(μ) = 0.16 is as below:

[13]:
```
# Visualization of the MSE loss for 10000 iteration

figureNo += 1
plt.figure(figureNo, figsize=(15,8))
plt.plot(MSE_loss_best)
plt.xlim(0, 10000)
plt.xlabel('Iterations')
plt.ylabel('MSE Loss')
plt.title('MSE Loss With Each Iteration')
plt.grid()
```



Then the trained network's weights matrix which has a size of (26,28*28) is reshaped into (26,28,28) to visualize the weights for each class as a seperate image. The visualization is as below:

[14]:
```
# Reshaping the weights matrix to its original image size
weights_best = weights_best.reshape(26,28,28)

# Visualization of the weights matrix
figureNo += 1
plt.figure(figureNo, figsize=(8,8))
for i, value in enumerate(weights_best):
    ax1 = plt.subplot(3, 10, i+1)
    ax1.imshow(weights_best[25- i].T, cmap = 'gray')
    ax1.set_yticks([])
    ax1.set_xticks([])

plt.subplots_adjust(wspace=0.05, hspace=0.05, left=0, right=1, bottom=0, top=0.5)
plt.show()
```



From the visualization above, it can be seen that the network is learning since the weights for letters roughly represents the desired class. However, there are some letters that are not clearly visualized. The classes that are not clearly visualized are mostly the classes which their upper-case and lower-case representations differ. And the letters that their upper-case and lower-case representations are similar like s, u, v, x are trained better.

Part c

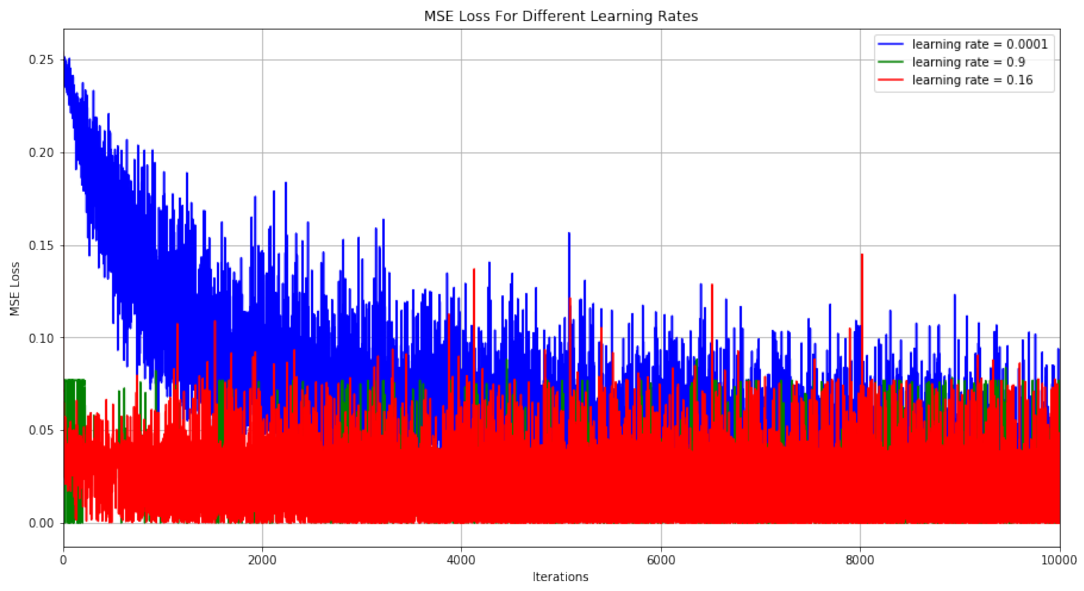
In this part, the network training process is done two more time with two different learning rates. One of the training is done with a learning rate which is substantially higher than the optimal learning rate we have found, and the other training is done with a learning rate which is substantially lower. Then, it is checked whether these different learning rates makes the learning algorithm stuck in a local minimum. The substantially higher learning rate is selected as 0.9 and the substantially lower learning rate is selected as 0.0001. Then the same network is iterated for 10000 times with the new learning rates and the corresponding weights and biases are stored separately. Also, the MSE for both network is collected. The Python implementation is as below:

[15]:
```
# Part C

# For getting the same result each run.
np.random.seed(8)

# Since each image has dimensions of 28*28 and there are 26 classes, weights vector␣
 ↪will be of size [26, 28*28]
# And there is a bias for each class so the shape of bias vector is [26,1]
# Values for both weights and bias vectors are random numbers from a Gaussian␣
 ↪Distribution with 0 mean and 0.01 std.
weightsHigh = np.random.normal( loc=0, scale=0.01, size=(26,28*28))
biasHigh = np.random.normal( loc=0, scale=0.01, size=(26,1))

weightsLow = np.random.normal( loc=0, scale=0.01, size=(26,28*28))
biasLow = np.random.normal( loc=0, scale=0.01, size=(26,1))

iterations = 10000
learningRateHigh = 0.9
learningRateLow = 0.0001

weights_high, bias_high, MSE_loss_high = perceptronTrain(trainimages_flat,␣
 ↪trainlabels, weightsHigh, biasHigh, iterations, learningRateHigh)
```

```
weights_low, bias_low, MSE_loss_low = perceptronTrain(trainimages_flat,␣
    ↪trainlabels, weightsLow, biasLow, iterations, learningRateLow)
```

The MSE's for optimal learning rate, substantially higher learning rate and substantially lower learning rate are drawn in the same plot:

[16]:
```
figureNo += 1
plt.figure(figureNo, figsize=(15,8))
plt.plot(MSE_loss_low, 'b')
plt.plot(MSE_loss_high, 'g')
plt.plot(MSE_loss_best, 'r')

plt.xlim(0, 10000)
plt.xlabel('Iterations')
plt.ylabel('MSE Loss')
plt.title('MSE Loss For Different Learning Rates')
plt.legend(["learning rate = " + str(learningRateLow), "learning rate = " +␣
    ↪str(learningRateHigh), "learning rate = %.2f" %(learningRateBest)])
plt.grid()
```



The MSE's for the three learning rates are shown above. From the plot, it can be seen that when the learning rate is substantially lower(0.0001), the MSE is decreasing slower but in a steady rate. However, when the learning rate is substantially higher(0.9), made MSE got stuck at a local minimum and the MSE could not decrease any further. Therefore, the learning rate with the optimal value(μ) has a faster decrease than the μ_{low} and it is not stuck in a local minimum like μ_{high}.

Part d

In this part, the network is tested with the test data. For that, a predict function is written. It picks the test image and calculate the weighted sum with the weights and bias that we have got after training the network. Then the output of the weighted sum is a size of (26,1) matrix. The location of the number that has the highest number defines the class. So, the output is in a form of one hot encoding. However, for simplicity, I have changed the result of the prediction from one hot encoding to the integer. The Python implementation of the predict class is:

```
[17]: # Part D

      # predict function takes the testimages and predicts the output
      # with the calculated weights and bias.
      # returns the class value as integer.
      def predict(testimages, weights, bias):
          result = list()
          testimages = testimages.reshape(1300,28*28)
          weights = weights.reshape(26,28*28)
          for i in range(np.shape(testimages)[0]):
              weightedSum = weights.dot(testimages[i].T).reshape(-1,1) - bias
              y_ = sigmoid(weightedSum)
              position = np.argmax(y_)
              result.append(26 - position)
          return result
```

```
[18]: # Predictions for optimal, high and low learningrates.

      test_best = predict(testimages, weights_best, bias_best)
      test_high = predict(testimages, weights_high, bias_high)
      test_low = predict(testimages, weights_low, bias_low)
```

This function is the same function that I have written in the **Question 2**. This function calculates the accuracy of the predictions that the network makes. Below, the accuracies for the networks trained with 3 different learning rates are calculated.

```
[19]: # For calculating the accuracy, accuracyCalc function is defined.

      def accuracyCalc(x, y):
          result = 0
          count = 0
          size = np.shape(x)[0]
          sentence = 'The accuracy of the model is: '
          for i in range(size):
              if (x[i] == y[i]):
                  count = count +1
          result = (count / size) * 100

          return result
```

```
[20]: accuracy_best = accuracyCalc(test_best, testlbls)
      accuracy_high = accuracyCalc(test_high, testlbls)
      accuracy_low = accuracyCalc(test_low, testlbls)
      print('Accuracy for the learning rate = %.2f is : %s'␣
       ↪%(learningRateBest,accuracy_best))
      print('Accuracy for the learning rate = ' +str(learningRateHigh) +'  is: '␣
       ↪+str(accuracy_high))
      print('Accuracy for the learning rate = ' +str(learningRateLow) +'  is: '␣
       ↪+str(accuracy_low))
```

```
Accuracy for the learning rate = 0.16 is : 49.15384615384615
Accuracy for the learning rate = 0.9  is: 13.615384615384615
Accuracy for the learning rate = 0.0001  is: 24.0
```

As it can be seen from the results, the learning rate $\mu = 0.16$ performs much better than the μ_{low} and μ_{high}. This result is expected since we have considered the optimal value of the learning rate is $\mu = 0.16$. Also, it can be seen that the μ_{low} performed a little better than the μ_{high}. The reason may be the that the network trained with μ_{high} is stuck at a local minimum at the beginning of the iterations.

Question 4

1 Implementing a Neural Network

In this question, a demo run of two layer neural network will be executed to get familiar with multi layered networks. Also, tuning of number of epochs, learning rate and regularization strength will be done. In the cell below, it can be seen that the relative error is used for error function.

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.

[1]:
```
# A bit of setup


from __future__ import print_function
from cs231n.classifiers.neural_net import TwoLayerNet
import matplotlib.pyplot as plt
import numpy as np

%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))))
```

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 the cell below, input size , hidden unit size, number of classes and number of inputs are defined. Then, using the **init_toy_data** function, a random sample from standart normal distribution is created with given number of inputs and input size. Also, labels of the inputs is defined. **init_toy_model** creates the two layer network with given number of input size, hidden unit size, number of classes and standard deviation.

[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()
```

2 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.

The `TwoLayerNet.loss` function computes the loss and the gradients for a two layer NN. The loss consists of the data loss and the regularization loss which are Softmax classifier loss and L2 regularization. The gradients are the gradients of the parameters with respect to loss function.

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

[3]:
```
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:
3.6802720745909845e-08
```

The scores is calculated using the weights and bias of the network. Now, compute the loss using the data loss and the regularization loss below:

3 Forward pass: compute loss

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

```
[4]: 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
```

Using the data loss and L2 regularization loss, we get a difference between loss and correct loss 1.7985612998927536e-13 which is much smaller to previously found 3.6802720745909845e-08. This result is expected since regularization reduces the variance hence avoids overfitting. Thus, error reduces. L2 regularization can be expressed as:

$$L2 = \lambda \sum_{1}^{n} \theta_i^2$$

4 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:

```
[5]: 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[param_name])))
```

```
W1 max relative error: 3.561318e-09
b1 max relative error: 2.738421e-09
W2 max relative error: 3.440708e-09
b2 max relative error: 4.447625e-11
```

All the maximum differences of the gradients of W1,b1,W2 and b2 is below 1e-8 which shows that the backward pass is working correctly and its error is very small which is neglectable.

5 Train the network

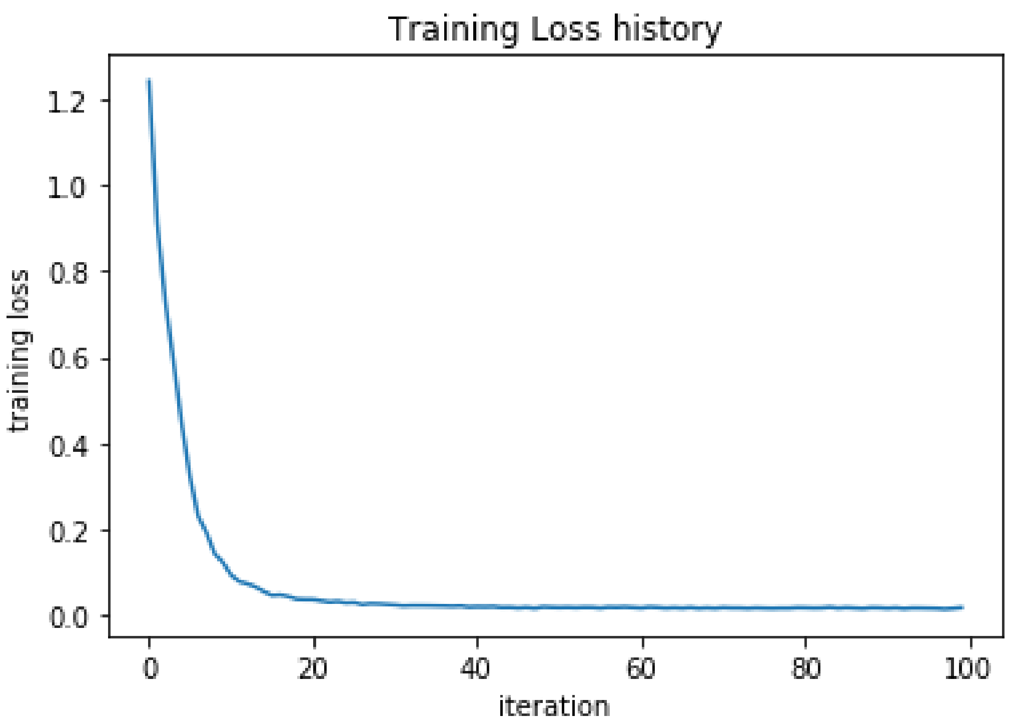
To train the network we will use stochastic gradient descent (SGD), similar to the SVM and Softmax classifiers. Look at the function `TwoLayerNet.train` and fill in the missing sections to implement the training procedure. This should be very similar to the training procedure you used for the SVM and

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.

[6]:
```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
```



The network is trained with learning rate = 1e-1 and regularization strength = 5e-6. Over 100 iterations, the training loss is visualized and its final value is 0.017.

6 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.

```
[7]: 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,)
```

Data is seperated to train, validation and test sets.

7 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.

[8]:
```
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
```

With the learning rate = 1e-4, learning rate decay = 0.95 and regularization strength = 0.25, the validation accuracy is not good. Also, loss of the network is decreasing too slowly. Thus, we will visualize the weights and the loss function to debug.

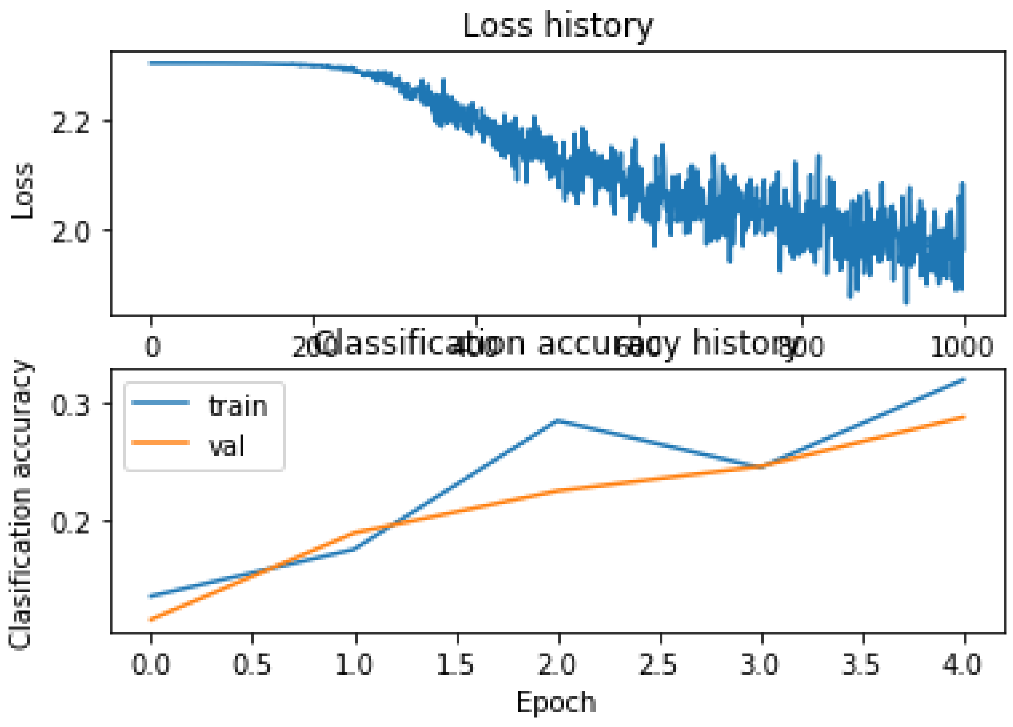
8 Debug the training

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.

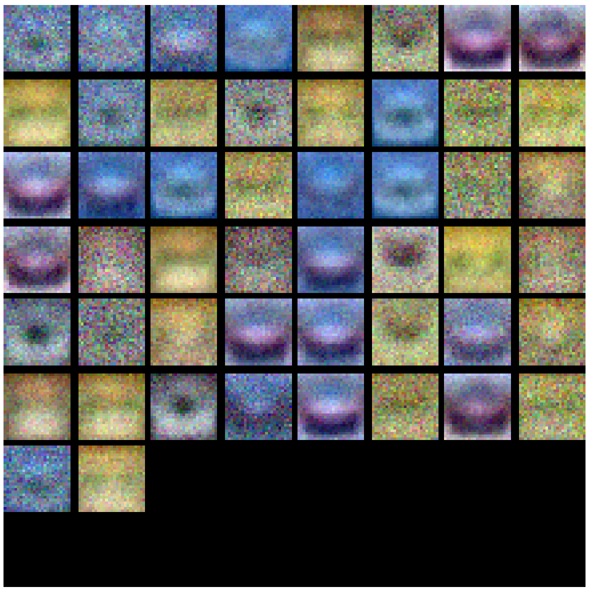
Another strategy is to visualize the weights that were learned in the first layer of the network. In most neural networks trained on visual data, the first layer weights typically show some visible structure when visualized.

[9]:
```
# 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()
```



From the Loss history, it can be seen that the loss is decreasing too slowly over 1000 iterations. It almost decreasing linearly. This informs that the learning rate may be too low. Also the training accuracy and the validation accuracy is too close, which may occur when the model does not have enough data. However, while increasing the data size, overfitting should be considered.

[10]:
```
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)
```



Also, from the visualization of the weights of the network, it can be seen that the weights does not clearly distinctive and the network did not train well.

9 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.).

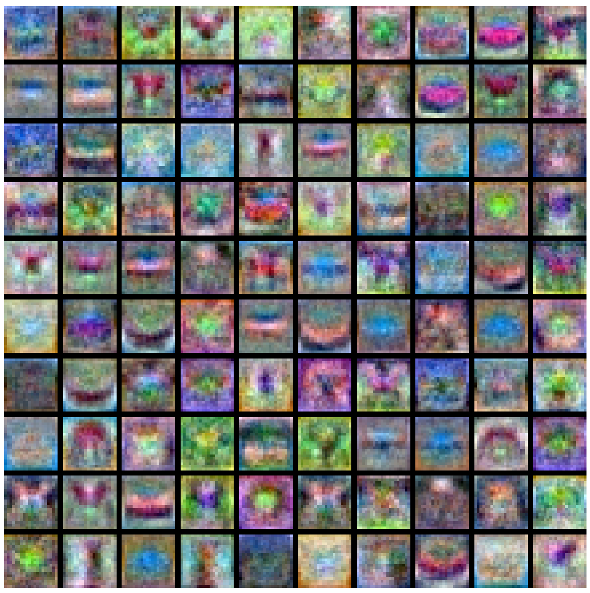
[11]:
```
best_net = None # store the best model into this

#################################################################################
# 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          #
```

```python
# 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                                #
#################################################################################
```

```
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
```

After tuning the hidden unit size, learning rates and the regularization strengths; the validation accuracy becomes 0.49 which is so much greater than the previous findings.

[12]:
```
# visualize the weights of the best network
show_net_weights(best_net)
```



Also, from the visualization of the weights, it can be seen that the each weights is nor more distinctive compared to previous ones. Thus, we can see that the network is trained better with the tuning of the hyperparameters.

10 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%.

[13]:
```
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 and 3**

Your explanation:

1. When the dataset is small, the network can not train well since it is more prune to pverfitting. Also, when the dataset is small, the effect of the outlier is huge on the network. Thus, training on a larger dataset can help to network train better and decrease the gap between testing and training accuracy.

2. Adding more hidden unit makes the network more complex. A network which is not complex enough for the given dataset may fail at training well which leads to underfitting. Then, adding more hidden units makes the network more complex and may overcome the underfitting. However, adding too many hidden unit may cause overfitting so the values should be tuned.

3. Regularization reduces the variance hence by penalizing the flexibility of the model. This approach avoids overfitting which makes the error decrease. However, finding a good regularization strength is also important since if the regularization strength increases too much, then the penalty grows and coefficients will approach zero. Or if the regularizaton strength is too low, then the penalty is shrinks and may even have small to no effect on the network.