EEE-443 Neural Networks Project-1

Name: Ayberk Yarkın

Surname: Yıldız

Id: 21803386

Date: November 5, 2021

Ayberk Yarkın Yıldız
21803386
EEE443 Project-1

Question 1

In this question, it is asked to find the $P(W)$, the prior probability distribution of the network weights by using the fact that MAP estimate for the network weights are obtained by the following optimization problem:

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

I will minimize the expression above to find the network weights. Assume the above expression as:

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

I have m input neurons as given. I will use the probabilistic fact that the posterior distribution is the multiply of the prior distribution and the likelihood for a random variable. Additionally, minimizing a posteriori estimate can be equated as calculating the minimized negative log likelihood. Taking the negative logarithm function does not change the MAP estimate since it is a monotonic function. Applying negative logarithm:

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

The expression becomes:

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

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

Assuming the data is derived from a Gaussian distribution, considering the Bayes' rule for the posterior distribution:

$$P(W|y) = \frac{P(W)P(y|W)}{\int P(W)P(y|W)dW}$$

In this case, the denominator is a constant does not depend on the optimization problem, then it can be ignored.

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

Where $P(W|y)$ is the posterior distribution of W, $P(W)$ is the prior distribution of W and $P(y|W)$ is the likelihood of the dataset.

The term $\sum_n (y^n - h(x^n, W))^2 + \beta$ is simply the error in the training of the neural network, in other words, the likelihood. As stated as the probabilistic fact above, the optimization problem can be separated into its two multipliers: the likelihood and the prior distribution, where L is the likelihood and P is the prior distribution.

$$L = C_1 e^{-\sum_n (y^n - h(x^n, W))^2}$$

$$P = C_2 e^{-\beta \sum_i w_i^2}$$

Where the multiplication of the constants is equal to 1.

$$C_1 C_2 = 1$$

The aim is to evaluate P, in other words, to evaluate C_2. To find C_2, I use the fact that sum of all probability outcomes is equal to 1.

$$\int_{-\infty}^{\infty} C_2 e^{-\beta \sum_i w_i^2} dw_i = 1$$

And we can open the sum of the terms on the exponential by their multiplications in exponential terms as there are m input neurons as given.

$$C_2 \int_{-\infty}^{\infty} e^{-\beta w_1^2} dw_1 \int_{-\infty}^{\infty} e^{-\beta w_2^2} dw_2 \dots \dots \int_{-\infty}^{\infty} e^{-\beta w_m^2} dw_m = 1$$

Calculating the individual integral:

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

The equation is simplified by multiplying the m amount of terms:

$$C_2 \left(\frac{\sqrt{\pi}}{\sqrt{\beta}}\right)^m = 1$$

$$C_2 = \left(\frac{\sqrt{\beta}}{\sqrt{\pi}}\right)^m = \left(\frac{\beta}{\pi}\right)^{\frac{m}{2}}$$

Having found the constant C_2, as a result, the prior probability distribution of the network weights P is found as:

$$P = C_2 e^{-\beta \sum_i w_i^2}$$

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

Ayberk Yarkın Yıldız
21803386
EEE443 Project-1

Question 2

a) In question 2, it is asked to design a neural network with a single hidden layer and four input neurons (with binary inputs), and a single output neuron to implement the following logic function:

$$(X_1 \vee \neg X_2) \oplus (\neg X_3 \vee \neg X_4)$$

XOR gate can be written as:

$$x \oplus y = (x \wedge \neg y) \vee (\neg x \wedge y)$$

Using the above formula and applying De Morgan's Law, the logic function can be written as:

$$((X_1 \vee \neg X_2) \wedge \neg(\neg X_3 \vee \neg X_4)) \vee (\neg(X_1 \vee \neg X_2) \wedge (\neg X_3 \vee \neg X_4))$$

$$((X_1 \vee \neg X_2) \wedge (X_3 \wedge X_4)) \vee ((\neg X_1 \wedge X_2) \wedge (\neg X_3 \vee \neg X_4))$$

Lastly, it can be simplified as:

$$(\neg X_2 \wedge X_3 \wedge X_4) \vee (\neg X_1 \wedge X_2 \wedge \neg X_3) \vee (\neg X_1 \wedge X_2 \wedge \neg X_4) \vee (X_1 \wedge X_3 \wedge X_4)$$

We can implement the total 4 blocks, consists of 3-inputs, in one neuron for each of them with total 4 neurons, and using a 4-input OR gate we can implement the output. Therefore, the hidden layer and output layer expressions are shown below:

$$h_1 = (\neg X_2 \wedge X_3 \wedge X_4)$$

$$h_2 = (\neg X_1 \wedge X_2 \wedge \neg X_3)$$

$$h_3 = (\neg X_1 \wedge X_2 \wedge \neg X_4)$$

$$h_4 = (X_1 \wedge X_3 \wedge X_4)$$

$$out = h_1 \vee h_2 \vee h_3 \vee h_4$$

We can write the weight inequalities for the hidden layer neurons, and then for the output neuron. For each specific output of a neuron, if the output = 1, the corresponding dot product of inputs and their weights become ≥ 0. Reversely, if the output = 0, the corresponding dot product of inputs and their weights become < 0. Additionally, if there is a specific input not included in a neuron, its weight is chosen as 0. In the below expressions, the unipolar activation function is indicated as v.

$$v(x) = \begin{Bmatrix} 1, & if\ x \geq 0 \\ 0, & if\ x < 0 \end{Bmatrix}$$

Hidden Unit 1 (h_1)

Weight vector of h_1 is:

$$w_1{}^T = [w_{11} \;\; w_{12} \;\; w_{13} \;\; w_{14} \;\; \theta_1]$$

For the h_1, there is no X_1 input, its weight is 0 and not included in the truth table.

$$w_{11} = 0$$

Then, output of h_1 neuron is:

$$h_1 = v(w_{12}X_2 + w_{13}X_3 + w_{14}X_4 - \theta_1)$$

X_2	X_3	X_4	h_1	Inequalities
0	0	0	0	$-\theta_1 < 0$
0	0	1	0	$w_{14} - \theta_1 < 0$
0	1	0	0	$w_{13} - \theta_1 < 0$
0	1	1	1	$w_{13} + w_{14} - \theta_1 \geq 0$
1	0	0	0	$w_{12} - \theta_1 < 0$
1	0	1	0	$w_{12} + w_{14} - \theta_1 < 0$
1	1	0	0	$w_{12} + w_{13} - \theta_1 < 0$
1	1	1	0	$w_{12} + w_{13} + w_{14} - \theta_1 < 0$

Table 1: Truth Table and the Inequalities for $\boldsymbol{h_1} = (\neg \boldsymbol{X_2} \wedge \boldsymbol{X_3} \wedge \boldsymbol{X_4})$

Hidden Unit 2 ($\boldsymbol{h_2}$)

Weight vector of h_2 is:

$$w_2{}^T = [w_{21} \;\; w_{22} \;\; w_{23} \;\; w_{24} \;\; \theta_2]$$

For the h_2, there is no X_4 input, its weight is 0 and not included in the truth table.

$$w_{24} = 0$$

Then, output of h_2 neuron is:

$$h_2 = v(w_{21}X_1 + w_{22}X_2 + w_{23}X_3 - \theta_2)$$

X_1	X_2	X_3	h_2	Inequalities
0	0	0	0	$-\theta_2 < 0$
0	0	1	0	$w_{23} - \theta_2 < 0$
0	1	0	1	$w_{22} - \theta_2 \geq 0$
0	1	1	0	$w_{22} + w_{23} - \theta_2 < 0$
1	0	0	0	$w_{21} - \theta_2 < 0$
1	0	1	0	$w_{21} + w_{23} - \theta_2 < 0$
1	1	0	0	$w_{21} + w_{22} - \theta_2 < 0$
1	1	1	0	$w_{21} + w_{22} + w_{23} - \theta_2 < 0$

Table 2: Truth Table and the Inequalities for $\boldsymbol{h_2} = (\neg \boldsymbol{X_1} \wedge \boldsymbol{X_2} \wedge \neg \boldsymbol{X_3})$

Hidden Unit 3 ($\boldsymbol{h_3}$)

Weight vector of h_3 is:

$$w_3{}^T = [w_{31}\ \ w_{32}\ \ w_{33}\ \ w_{34}\ \ \theta_3]$$

For the h_3, there is no X_3 input, its weight is 0 and not included in the truth table.

$$w_{33} = 0$$

Then, output of h_3 neuron is:

$$h_3 = v(w_{31}X_1 + w_{32}X_2 + w_{34}X_4 - \theta_3)$$

X_1	X_2	X_4	h_3	Inequalities
0	0	0	0	$-\theta_3 < 0$
0	0	1	0	$w_{34} - \theta_3 < 0$
0	1	0	1	$w_{32} - \theta_3 \geq 0$
0	1	1	0	$w_{32} + w_{34} - \theta_3 < 0$
1	0	0	0	$w_{31} - \theta_3 < 0$
1	0	1	0	$w_{31} + w_{34} - \theta_3 < 0$
1	1	0	0	$w_{31} + w_{32} - \theta_3 < 0$
1	1	1	0	$w_{31} + w_{32} + w_{34} - \theta_3 < 0$

Table 3: Truth Table and the Inequalities for $\boldsymbol{h_3} = (\neg \boldsymbol{X_1} \wedge \boldsymbol{X_2} \wedge \neg \boldsymbol{X_4})$

Hidden Unit 4 ($\boldsymbol{h_4}$)

Weight vector of h_4 is:

$$w_4{}^T = [w_{41}\ \ w_{42}\ \ w_{43}\ \ w_{44}\ \ \theta_4]$$

For the h_4, there is no X_2 input, its weight is 0 and not included in the truth table.

$$w_{42} = 0$$

Then, output of h_4 neuron is:

$$h_4 = v(w_{41}X_1 + w_{43}X_3 + w_{44}X_4 - \theta_4)$$

X_1	X_3	X_4	h_4	Inequalities
0	0	0	0	$-\theta_4 < 0$
0	0	1	0	$w_{44} - \theta_4 < 0$
0	1	0	0	$w_{43} - \theta_4 < 0$
0	1	1	0	$w_{43} + w_{44} - \theta_4 < 0$
1	0	0	0	$w_{41} - \theta_4 < 0$
1	0	1	0	$w_{41} + w_{44} - \theta_4 < 0$
1	1	0	0	$w_{41} + w_{43} - \theta_4 < 0$
1	1	1	1	$w_{41} + w_{43} + w_{44} - \theta_4 \geq 0$

Table 4: Truth Table and the Inequalities for $\boldsymbol{h_4} = (\boldsymbol{X_1} \wedge \boldsymbol{X_3} \wedge \boldsymbol{X_4})$

Output Neuron (out)

The outputs of the four hidden layer units are used for inputs for the last output neuron.

Weight vector of out is:

$$w_5^T = [w_{51}\ \ w_{52}\ \ w_{53}\ \ w_{54}\ \ \theta_5]$$

Then, output of out neuron is:

$$out = v(w_{51}h_1 + w_{52}h_2 + w_{53}h_3 + w_{54}h_4 - \theta_5)$$

h_1	h_2	h_3	h_4	out	Inequalities
0	0	0	0	0	$-\theta_5 < 0$
0	0	0	1	1	$w_{54} - \theta_5 \geq 0$
0	0	1	0	1	$w_{53} - \theta_5 \geq 0$
0	0	1	1	1	$w_{53} + w_{54} - \theta_5 \geq 0$
0	1	0	0	1	$w_{52} - \theta_5 \geq 0$
0	1	0	1	1	$w_{52} + w_{54} - \theta_5 \geq 0$
0	1	1	0	1	$w_{52} + w_{53} - \theta_5 \geq 0$
0	1	1	1	1	$w_{52} + w_{53} + w_{54} - \theta_5 \geq 0$
1	0	0	0	1	$w_{51} - \theta_5 \geq 0$
1	0	0	1	1	$w_{51} + w_{54} - \theta_5 \geq 0$
1	0	1	0	1	$w_{51} + w_{53} - \theta_5 \geq 0$
1	0	1	1	1	$w_{51} + w_{53} + w_{54} - \theta_5 \geq 0$
1	1	0	0	1	$w_{51} + w_{52} - \theta_5 \geq 0$
1	1	0	1	1	$w_{51} + w_{52} + w_{54} - \theta_5 \geq 0$
1	1	1	0	1	$w_{51} + w_{52} + w_{53} - \theta_5 \geq 0$
1	1	1	1	1	$w_{51} + w_{52} + w_{53} + w_{54} - \theta_5 \geq 0$

Table 5: Truth Table and the Inequalities for $out = h_1 \vee h_2 \vee h_3 \vee h_4$

In results, the input and output weights are in form:

$$W_{in} = \begin{bmatrix} w_1{}^T \\ w_2{}^T \\ w_3{}^T \\ w_4{}^T \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} & \theta_1 \\ w_{21} & w_{22} & w_{23} & w_{24} & \theta_2 \\ w_{31} & w_{32} & w_{33} & w_{34} & \theta_3 \\ w_{41} & w_{42} & w_{43} & w_{44} & \theta_4 \end{bmatrix}$$

$$W_{out} = w_5^T = [w_{51}\ \ w_{52}\ \ w_{53}\ \ w_{54}\ \ \theta_5]$$

b) In this part, it is asked to determine the network weights that satisfies the %100 performance implementing the logic network. By using the inequalities in part a), I determined the weights as:

$$W_{in} = \begin{bmatrix} {w_1}^T \\ {w_2}^T \\ {w_3}^T \\ {w_4}^T \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} & \theta_1 \\ w_{21} & w_{22} & w_{23} & w_{24} & \theta_2 \\ w_{31} & w_{32} & w_{33} & w_{34} & \theta_3 \\ w_{41} & w_{42} & w_{43} & w_{44} & \theta_4 \end{bmatrix} = \begin{bmatrix} 0 & -0.5 & 0.6 & 0.6 & 1 \\ -0.5 & 1.2 & -0.5 & 0 & 1 \\ -0.5 & 1.2 & 0 & -0.5 & 1 \\ 0.3 & 0 & 0.4 & 0.5 & 1 \end{bmatrix}$$

$$W_{out} = {w_5}^T = [w_{51} \ \ w_{52} \ \ w_{53} \ \ w_{54} \ \ \theta_5] = [1 \ \ 1 \ \ 1 \ \ 1 \ \ 0.5]$$

The overall results of the logic circuit and the neural network output is shown below:

X_1	X_2	X_3	X_4	$(X_1 \lor \neg X_2) \oplus (\neg X_3 \lor \neg X_4)$	out
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	1	1
0	1	0	0	1	1
0	1	0	1	1	1
0	1	1	0	1	1
0	1	1	1	0	0
1	0	0	0	0	0
1	0	0	1	0	0
1	0	1	0	0	0
1	0	1	1	1	1
1	1	0	0	0	0
1	1	0	1	0	0
1	1	1	0	0	0
1	1	1	1	1	1

Table 6: Overall Results

The accuracy is found by code, by comparing the results of the output from the neural network with the determined weights and the output from the logic function. The code can be seen at the end of the report. When I compared the two outputs them, they turn out to be the same. Therefore, the accuracy of the neural network achieves the 100% performance in implementing the logic function.

c) In this part, it is stated that the input data samples are affected by small random noise. With including the noise, which is selected as $N(0{,}0.1)$, the network in part **b)** is tested, and got the accuracy of 96.5%. Although the weights in part **b)** are not selected to function better under noisy conditions, the accuracy percentage is good. However, for near perfect accuracy, I started to select new set of weights which can adapt the noisy condition situations. I selected the new weights according to the least squares method, and selected the biases as zero at the beginning. Afterwards, I defined the biases by comparing the desired output vector and the found output by the selected weights. A sample evaluation are shown below for the Hidden Unit 1 (h_1):

X_2	X_3	X_4	h_1
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

Weights are selected as according to the least squares method (the bias is 0 at the beginning):

$$w_1{}^T = [w_{11}\ \ w_{12}\ \ w_{13}\ \ w_{14}\ \ \theta_1] = [0\ \ -0.25\ \ 0.25\ \ 0.25\ \ 0]$$

Then, I evaluated the output according to these weights:

$$h_1 = v(w_{12}X_2 + w_{13}X_3 + w_{14}X_4 - \theta_1)$$

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} [-0.25\ \ 0.25\ \ 0.25\ \ 0] = \begin{bmatrix} 0 \\ 0.25 \\ 0.25 \\ 0.5 \\ -0.25 \\ 0 \\ 0 \\ 0.25 \end{bmatrix}$$

The desired output was:

$$h_1 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

The biggest difference for any row between the desired and found output is 1-0.5=0.5 and the smallest is 0.25. Therefore, I should choose the bias between 0.25<θ_1<0.5. For the best case, I got the mean of this interval and determined the bias.

$$\theta_1 = mean(0.25 + 0.5) = 0.375$$

The weight vector for h_1 becomes:

$$w_1{}^T = [w_{11}\ \ w_{12}\ \ w_{13}\ \ w_{14}\ \ \theta_1] = [0\ \ -0.25\ \ 0.25\ \ 0.25\ \ 0.375]$$

The similar approach is done for all hidden layer units. The final weight vectors are shown below. The output weights are remained the same as in part **b)**:

$$W_{in} = \begin{bmatrix} w_1{}^T \\ w_2{}^T \\ w_3{}^T \\ w_4{}^T \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} & \theta_1 \\ w_{21} & w_{22} & w_{23} & w_{24} & \theta_2 \\ w_{31} & w_{32} & w_{33} & w_{34} & \theta_3 \\ w_{41} & w_{42} & w_{43} & w_{44} & \theta_4 \end{bmatrix} = \begin{bmatrix} 0 & -0.25 & 0.25 & 0.25 & 0.375 \\ -0.25 & 0.25 & -0.25 & 0 & 0.125 \\ -0.25 & 0.25 & 0 & -0.25 & 0.125 \\ 0.25 & 0 & 0.25 & 0.25 & 0.625 \end{bmatrix}$$

$$W_{out} = w_5{}^T = [w_{51}\ \ w_{52}\ \ w_{53}\ \ w_{54}\ \ \theta_5] = [1\ \ 1\ \ 1\ \ 1\ \ 0.5]$$

I again tested the accuracy with these new weights, which resulted in an accuracy of %99.75. Therefore, the new neural network with new weights are more resistant to the noisy conditions compared to the weights selected at part **b)**

d) In this part, it is asked to test the two networks determined in part **b)** and **c)** to determine that the network in part **c)** is more consistent in noisy conditions compared to the network in part **b)**

25 samples for each 16 different input combination cases results in 400 input samples. A (400x4) sized input matrix is created. And for the noise, it is selected as a randomly generated Gaussian noise $N(0,0.1)$ with the same size. They are added to have a final input matrix X_n and additionally a column of -1's are added to the last column of this input matrix.

$$X_n = X + N = [x_1 + n_1\ \ \ x_2 + n_2\ \ \ x_3 + n_3\ \ \ x_4 + n_4]_{400\times 4}$$

$$\theta = \begin{bmatrix} -1 \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ -1 \end{bmatrix}_{400\times 1}$$

$$X_f = [X_n\ \ \ \theta]_{400\times 5}$$

Accuracy is computed with the networks in part **b)** and **c)** individually. For the non-robust weighted network in part **b)**, accuracy is found as 86.75%. For the robust weighted network in part **c)**, accuracy is found as 90.0%. Similarly, these results show that the network with the new robust weights is more resistant to the noise. Selecting the weights and biases using least squares method increases the classification performance.

Ayberk Yarkın Yıldız
21803386
EEE443 Project-1

Question 3

a) In this question, the sample images from the data is shown below. The prior data of images were not right oriented. Therefore, I take the transpose of the images, and found the right shapes.

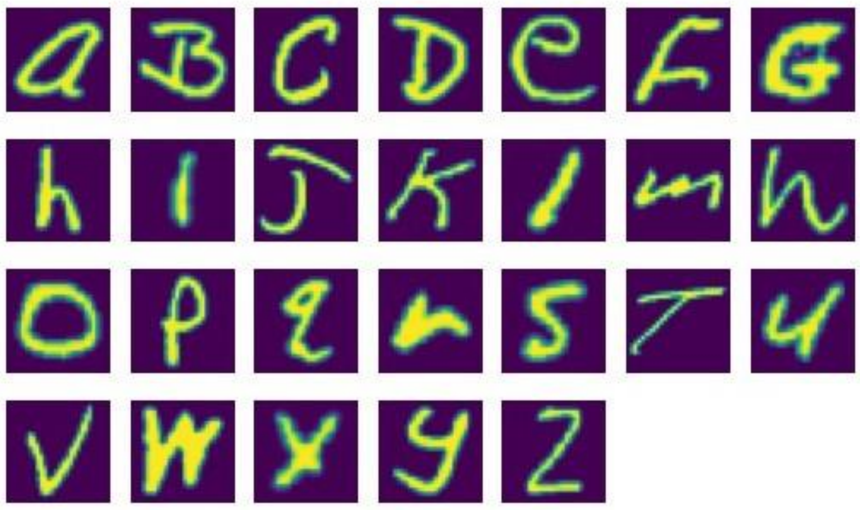


Figure 1: The sample images of each class

Afterwards, the correlation matrix is found for both within-class and across-class. The correlation formula is seen below:

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

The within-class correlation matrix can be seen below:

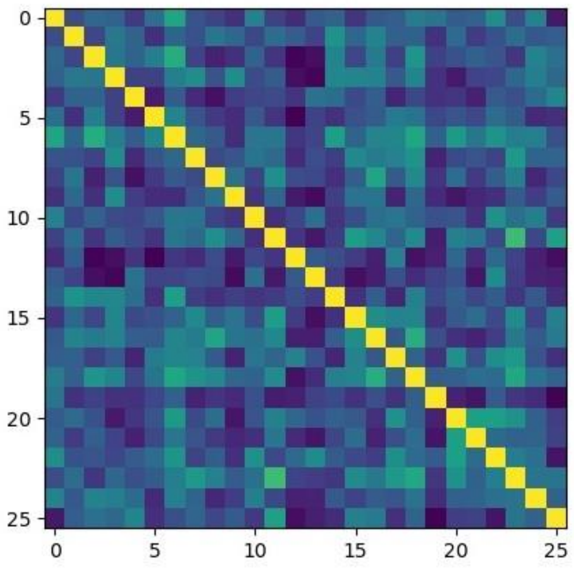


Figure 2: The correlation matrix for within-class

The diagonal corresponds to the same images, therefore has the covariance 1. As the similarity in terms of shape of the letters increase, the color of the matrix become more light and yellow in that location.

The across-class covariance matrix is shown below:

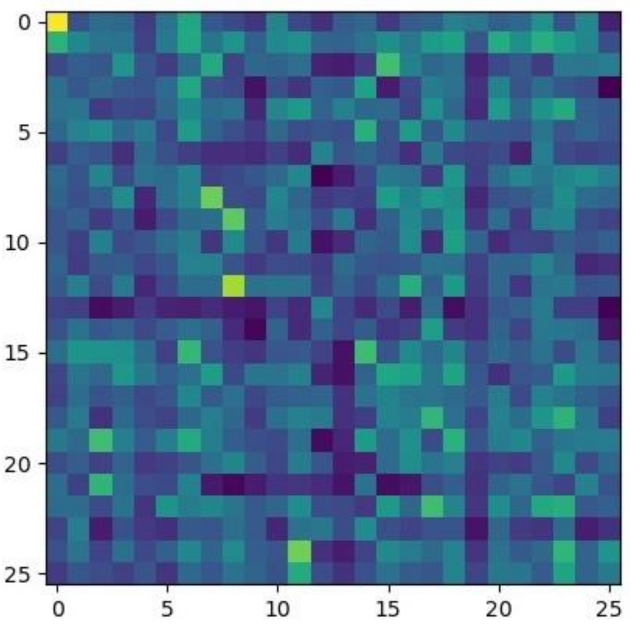


Figure 3: The correlation matrix for across-class

It can be observed that within-class variability is less than across-class variability. The more darker it gets, the less covariance value it has. The correlation matrix gives the information about where our network makes mistakes.

b) A single layer perceptron The optimal learning rate was found $\eta = 0.06$ by trial and error to have the best outcome by measuring the MSE value. The MSE is given by:

$$MSE = \frac{1}{N}\sum_{i=0}^{N}\left||y_i - d|\right|^2 = \frac{1}{N}\sum_{i=0}^{N} J$$

J can be written as:

$$J = \left(Y - \sigma(Wx - B)\right).\left(Y - \sigma(Wx - B)\right)^T$$

The loss function is used. The learning procedure is:

$$Wnew = W - \eta\frac{\partial J}{\partial W}$$

$$Bnew = B - \eta\frac{\partial J}{\partial B}$$

Then, the derivative of J can be found as:

$$\frac{\partial J}{\partial W} = -(Y - \sigma(Wx - B))(\sigma(Wx - B))(1 - \sigma(Wx - B))x^T$$

$$\frac{\partial J}{\partial B} = (Y - \sigma(Wx - B))$$

The code is constructed respectively to the calculations.

The visualization of weights with the learning rate $\eta = 0.06$ is shown below:

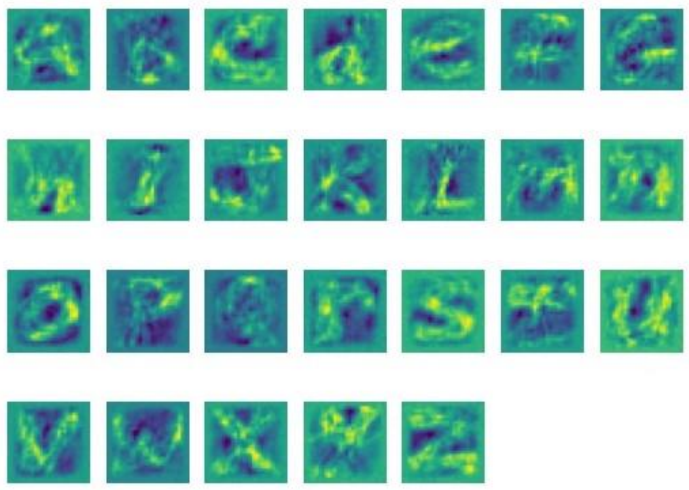


Figure 4: Visualized weight vectors

The network is learning, that the visualized images are slightly similar to the original ones. There were both upper-case and lower-case letters. For the letters whose upper and lower cases are similar, such as “Z” and “z”, learning was more successful. Due to the versions of letters in the dataset, some of the letters learned less relatively, and also due to their different shapes of upper and lower cases, “h” and “H” is an example.

c) The training is continued in this part, with different learning rates. μ_{high} and μ_{low} are the higher and lower learning rates respectively. Since the learning rate is usually in the interval (0,1), I chose the rates as:

$$\eta_{high} = 1$$

$$\eta_{low} = 0.0001$$

MSE plot for different μ values is shown below:

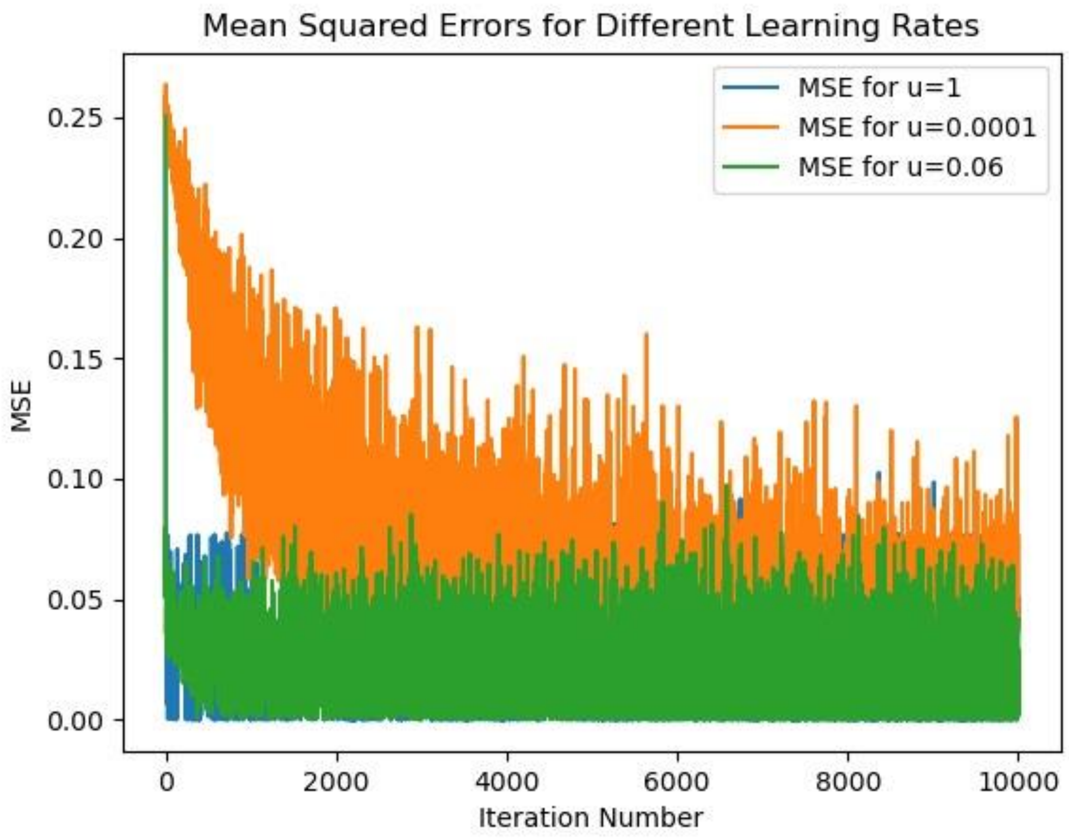


Figure 5: MSE Plots for different learning rates

The low learning rate shows that the learning process is slow when learning rate is low. The MSE value is higher compared to other values. The high learning rate stays the same most of the time due to its high learning rate.

d) In this part, the network encounters a test data that has never seen before. The output is observed by:

$$output = \sigma(Wx' - B)$$

The classification accuracy is calculated for all three different learning rates.

Accuracy for $\eta = 0.06$: 59.769230769230774 %

Accuracy for $\eta = 1$: 18.0 %

Accuracy for $\eta = 0.0001$: 26.846153846153847 %

As it can be seen, the optimal accuracy is between 59%-60% whereas other accuracies are much less than the optimal value.

Ayberk Yarkın Yıldız
21803386
EEE443 Project-1

Question 4

In this question, the Jupyter Notebook is used to learn about the concept of two-layer neural networks. Firstly, the class TwoLayerNet is used to determine the instances of the network. Toy data and a toy model is initialized. Then, it starts the forward pass by using the weights and biases to compute the scores. TwoLayerNet uses the ReLU activation function to evaluate the hidden layers, eventually finding the scores. ReLU is in the form below:

$$ReLU(x) = \max(0, x)$$

It is a non-linear activation function. ReLU provides much faster learning compared to other activation functions. ReLU helped in finding the hidden layer activation, then the scores, and the loss is found. For training, the stochastic gradient descent is used. To increase the accuracy, it is observed that the learning rate, hidden unit number and the regularization strength affect the accuracy. Changing and inserting new values for these, a higher value of accuracy is found. Test accuracy is increased from 48.2% to 50.4%.

Original values:

$$Test\ Accuracy = 48.2\%$$

$$\eta = 0.001$$

$$hidden\ unit\ size = 100$$

$$regularization\ strength = 0.75$$

I changed these values and find a higher accuracy:

$$Test\ Accuracy = 50.4\%$$

$$\eta = 0.001$$

$$hidden\ unit\ size = 120$$

$$regularization\ strength = 0.5$$

New visualization is shown below:

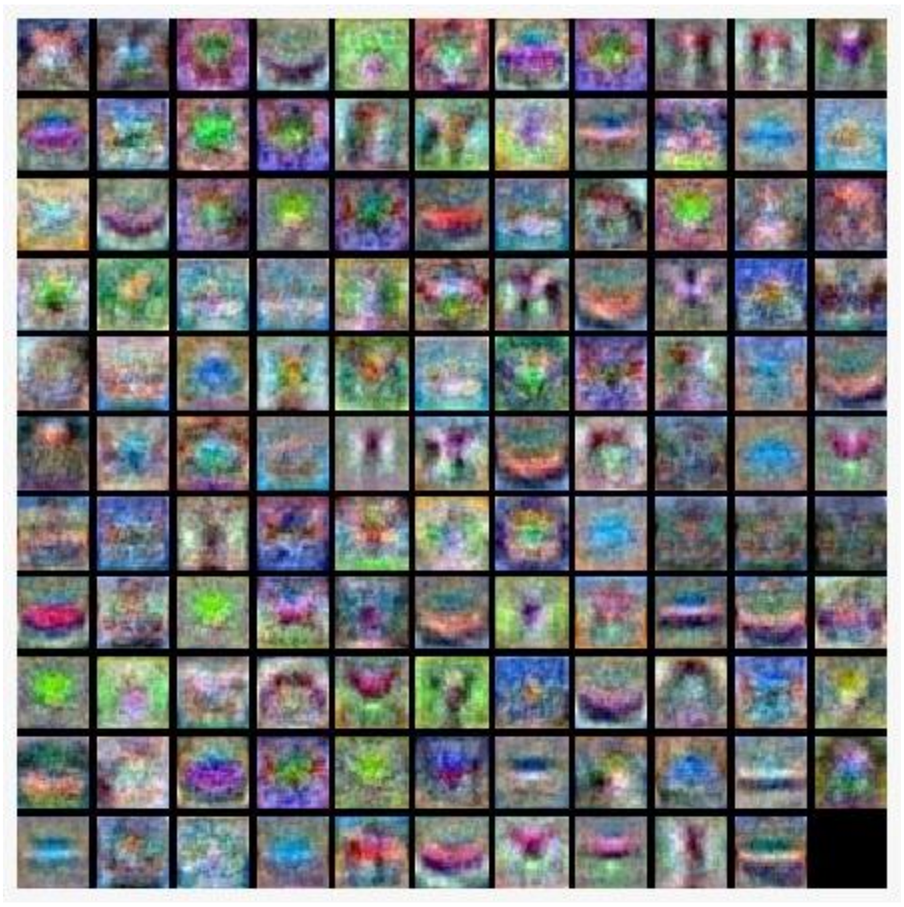


Figure 6: Weights visualization with new parameters

Outputs of the completed demo, including the inline questions, is shown at appendix at the end of the report.

Appendix

Question-2

```
import numpy as np


W_in=np.array([[0,-0.5,0.6,0.6,1], [-0.5,1.2,-0.5,0,1], [-0.5,1.2,0,-0.5,1], [0.3,0,0.4,0.5,1]])


    X = np.array([
                [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]
                ])


    def uStep(x):
        x[x >= 0] = 1
        x[x < 0] = 0
        y = x
        return y

    h_out=uStep(np.matmul(W_in,X.T))

    h_out = np.concatenate((h_out.T,-1*np.ones([16,1])), axis=1)

    W_out=np.array([[1,1,1,1,0.5]])
    out=uStep(np.matmul(W_out,h_out.T))

    def XOR(x,y):
        return (x and (not y)) or ((not x) and y)
    def logicFunction(x):
        return XOR(x[0] or (not x[1]), (not x[2]) or (not x[3]))



    X_c = X[:,0:4]
    out_logic = list()

    for i in range(16):
        out_logic.append(logicFunction(X_c[i,:]))

    print('Question 2 Part B')
    print((out == out_logic).all())



    #part c and d
```

```
W_in_new=np.array([[0,-0.25,0.25,0.25,0.375], [-0.25,0.25,-0.25,0,0.125], [-0.25,0.25,0,-
0.25,0.125], [0.25,0,0.25,0.25,0.625]])

X_n= np.tile(X,(25,1))

std = 0.2

N = np.random.normal(0, std, 2000).reshape(400,5)

N[:,4] = 0


noisy_X_n= X_n + N


X_c_n = X_n[:,0:4]

out_logic_n=list()

for i in range(400):

    out_logic_n.append(logicFunction(X_c_n[i,:]))


W_out_n = W_out




#part c için accuracy

std_s = 0.1

small_noise=np.random.normal(0, std_s, 2000).reshape(400,5)

small_noise[:,4] = 0

small_noisy_X= X_n + small_noise


h_out_accuracy_s=uStep(np.matmul(W_in,small_noisy_X.T))

h_out_accuracy_s=np.concatenate((h_out_accuracy_s.T,-1*np.ones([400,1])), axis=1)

out_accuracy_s=uStep(np.matmul(W_out,h_out_accuracy_s.T))


h_out_accuracy_n=uStep(np.matmul(W_in_new,small_noisy_X.T))

h_out_accuracy_n=np.concatenate((h_out_accuracy_n.T,-1*np.ones([400,1])), axis=1)
```

```
    out_accuracy_n=uStep(np.matmul(W_out_n,h_out_accuracy_n.T))


    count_initial=0

    for i in range(400):

        if(out_logic_n[i] == out_accuracy_s[0,i]):

            count_initial += 1

    print('\nQuestion 2 Part C\nAccuracy for initial weighted network (small noise) = ' +
str(count_initial/400*100) + "%")



    count_after=0

    for i in range(400):

        if(out_logic_n[i] == out_accuracy_n[0,i]):

            count_after += 1

    print('Accuracy for new robust weighted network (small noise) = ' +
str(count_after/400*100) + "%")


    #part d accuracy


    h_out_n = uStep(np.matmul(W_in_new,noisy_X_n.T))

    h_out_n = np.concatenate((h_out_n.T,-1*np.ones([400,1])), axis=1)

    out_n=uStep(np.matmul(W_out_n,h_out_n.T))


    h_out_accuracy=uStep(np.matmul(W_in,noisy_X_n.T))

    h_out_accuracy=np.concatenate((h_out_accuracy.T,-1*np.ones([400,1])), axis=1)

    out_accuracy=uStep(np.matmul(W_out,h_out_accuracy.T))


    count=0

    for i in range(400):
        if(out_logic_n[i] == out_accuracy[0,i]):
            count += 1
    print('\nQuestion 2 Part D\nAccuracy for initial weighted network = ' + str(count/400*100)
+ "%")

    count_n=0
    for i in range(400):
        if(out_logic_n[i] == out_n[0,i]):
            count_n += 1
    print('Accuracy for new robust weighted network = ' + str(count_n/400*100) + "%")
```

Question-3

```
import numpy as np
import matplotlib.pyplot as plt
import random
import h5py

filename = 'assign1_data1.h5'
    f1 = h5py.File(filename,'r+')

    testims = np.array(f1["testims"])
    testlbls = np.array(f1["testlbls"])
    trainims = np.array(f1["trainims"])
    trainlbls = np.array(f1["trainlbls"])

    trainims = trainims.T
    testims = testims.T
    trainsize = trainlbls.size
    counting = 1
```

```
fig = plt.figure()
for i in range(trainsize):
    if(counting == trainlbls[i]):
          ax = plt.subplot(5, 7, counting)
          plt.imshow(trainims[:,:,i])
          ax.axis('off')
          counting += 1

co_matrix = np.zeros([26,26])
for a in range(26):
    for b in range(26):
        corrcoef = np.corrcoef(trainims[:,:,a*200].flat, trainims[:,:,b*200].flat)
        co_matrix[a,b]=corrcoef[1,0]

fig2 = plt.figure()
plt.imshow(co_matrix)

co_matrix_2 = np.zeros([26,26])
for a in range(26):
    for b in range(26):
        corrcoef_2 = np.corrcoef(trainims[:,:,a*199].flat, trainims[:,:,b*200].flat)
        co_matrix_2[a,b]=corrcoef_2[1,0]

fig3 = plt.figure()
plt.imshow(co_matrix_2)

"""
partb
"""
```

```
one_encoder = np.zeros([26,5200])

for i in range(trainsize):
    one_encoder[int(trainlbls[i])-1,i] = 1

learning_rate = 0.06
learning_rate_up = 1
learning_rate_down = 0.0001
mean_3=0
std_3=0.01

def sigmoid(x):
    return 1/(1+np.exp(-x))

def norm(x):
    return x/np.max(x)

def RandWB(m,s):
    b = np.random.normal(m,s,26).reshape(26,1)
    w = np.random.normal(m,s,26*(28**2)).reshape(26,(28**2))
    return w,b

def ins(ims,onehot):

    rax = random.randint(0,5199)
    rag = trainims[:,:,rax].reshape(28**2,1)
    rag = norm(rag)
    ray = one_encoder[:,rax].reshape(26,1)
    return rax,rag,ray


def outa(w,b,i):

    return sigmoid(np.matmul(w,i)-b)


def mse(k):

    return np.sum((k)**2/(k.shape[0]))


def error(m,n):

    return m-n


def sigderiv(a,b):

    return (a*b*(1-b))


def updates(l,i):

    change = l*i

    return change


mseList = list()

weight_c,bias_c = RandWB(mean_3,std_3)


for i in range(10000):


    rax,rag,ray = ins(trainims,one_encoder)


    out_ne = outa(weight_c,bias_c,rag)


    erro=error(ray,out_ne)
    wupdate = -2*np.matmul(sigderiv(erro,out_ne),rag.T)

    bupdate = 2*(sigderiv(erro,out_ne))

    weight_c -= updates(learning_rate,wupdate)
    bias_c -= updates(learning_rate,bupdate)

    mseList.append(mse(error(ray,out_ne)))

figlet = plt.figure()
for i in range(26):
    ax2 = plt.subplot(4, 7, i+1)
    plt.imshow(weight_c[i,:].reshape(28,28))
    ax2.axis('off')

"""part c """

weightsUp,biasUp = RandWB(mean_3,std_3)
weightsDown,biasDown = RandWB(mean_3,std_3)

mseListHi = list()
mseListLow = list()

for i in range(10000):
    rax,rag,ray = ins(trainims,one_encoder)
    out_ne = outa(weightsUp,biasUp,rag)
    erro = error(ray,out_ne)
    wupdate = -2*np.matmul(sigderiv(erro,out_ne),rag.T)
    bupdate = 2*(sigderiv(erro,out_ne))
```

```
        weightsUp -= updates(learning_rate_up,wupdate)
        biasUp -= updates(learning_rate_up,bupdate)

        mseListHi.append(mse(error(ray,out_ne)))

    for i in range(10000):
        rax,rag,ray = ins(trainims,one_encoder)
        out_ne = outa(weightsDown,biasDown,rag)
        erro = error(ray,out_ne)

        wupdate = -2*np.matmul(sigderiv(erro,out_ne),rag.T)
        bupdate = 2*(sigderiv(erro,out_ne))

        weightsDown -= updates(learning_rate_down,wupdate)
        biasDown -= updates(learning_rate_down,bupdate)

        mseListLow.append(mse(error(ray,out_ne)))

    fig4 = plt.figure()
    plt.plot(mseListHi)
    plt.plot(mseListLow)
    plt.plot(mseList)
    plt.legend(["MSE for u="+str(learning_rate_up), "MSE for u="+str(learning_rate_down),
"MSE for u="+str(learning_rate)])
    plt.title("Mean Squared Errors for Different Learning Rates")
    plt.xlabel("Iteration Number")
    plt.ylabel("MSE")
    plt.show()
```

```
""" part d """
testsize = testlbls.shape[0]
testims = testims.reshape(28**2,testsize)
testnorm = norm(testims)


def accuracies(w,b):
    bias_d = np.zeros([26,testsize])


    for i in range (testsize):
        bias_d[:,i] = b.flatten()
    guessino = outa(w,bias_d,testnorm)
    guessino_i = np.zeros(guessino.shape[1])


    for i in range (guessino.shape[1]):
        guessino_i[i] = np.argmax(guessino[:,i])+1


    counters = 0
    for i in range (guessino_i.shape[0]):
        if (guessino_i[i] == testlbls[i]):
            counters += 1


    accuracy = counters/testlbls.shape[0]*100
    return accuracy


print('Accuracy for learning rate =',learning_rate,':',accuracies(weight_c,bias_c),'%')
print('Accuracy for learning rate =',learning_rate_up,':',accuracies(weightsUp,biasUp),'%')
print('Accuracy for learning rate
=',learning_rate_down,':',accuracies(weightsDown,biasDown),'%')
```

Question-4

Implementing a Neural Network

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

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

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))))
```

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

In [2]:
```
# Create a small net and some toy data to check your implementations.
# Note that we set the random seed for repeatable experiments.

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

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

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

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

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 [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:
3.6802720496109664e-08
```

Forward pass: compute loss

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

In [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
```

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 [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=Fals
    print('%s max relative error: %e' % (param_name, rel_error(param_grad_num, grads
```

```
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

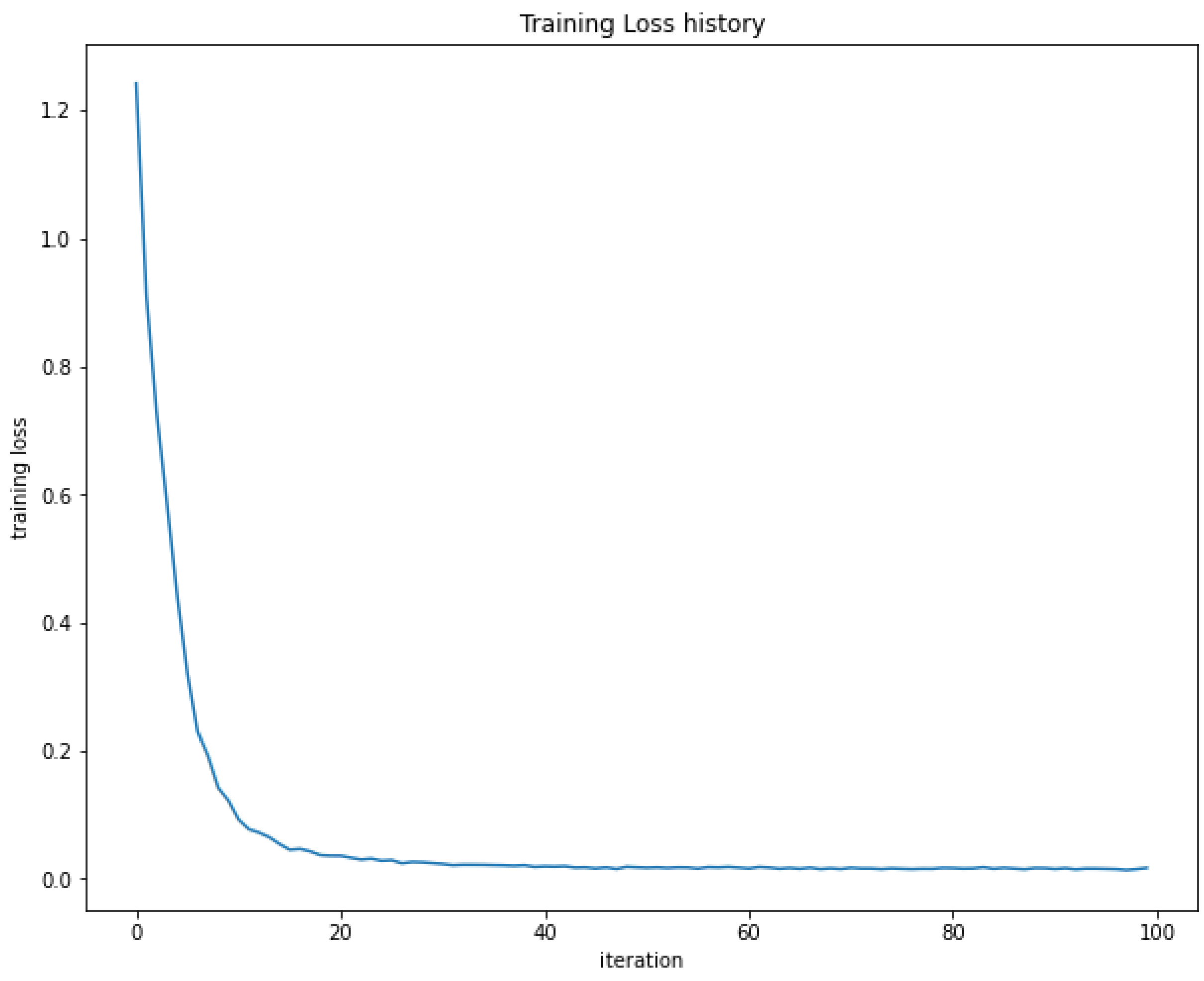
To train the network we will use stochastic gradient descent (SGD), similar to the SVM and Softmax classifiers. Look at the function `TwoLayerNet.train` and fill in the missing sections to implement the training procedure. This should be very similar to the training procedure you used for the SVM and 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 [6]:

```
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 [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 memo
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,)
```

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 [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
```

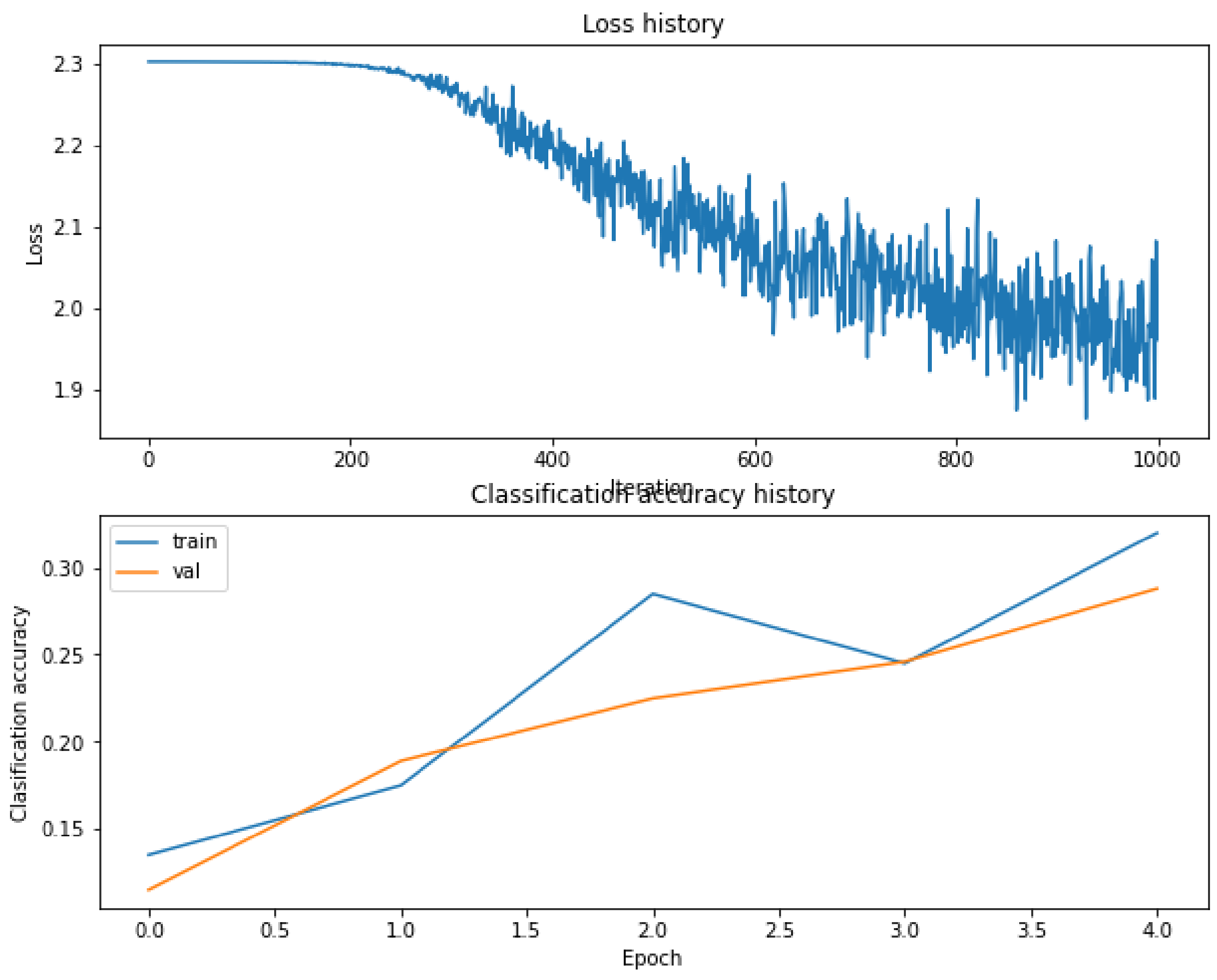
Debug the training

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.

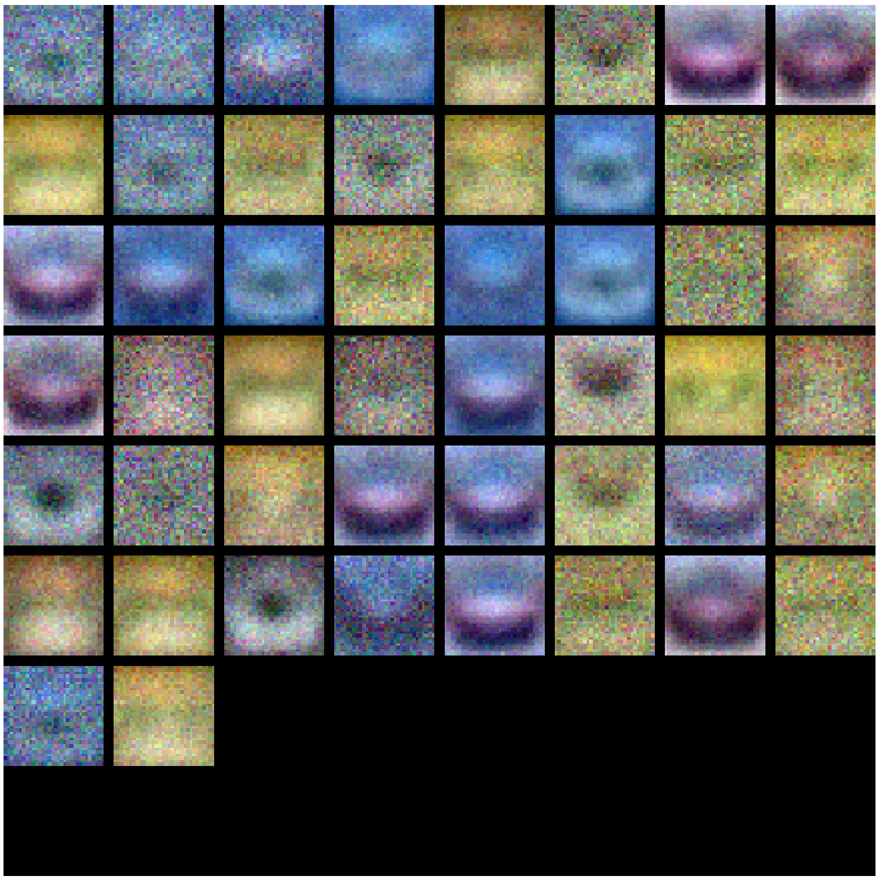
One strategy for getting insight into what's wrong is to plot the loss function and the accuracies on the training and validation sets during optimization.

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 [9]:
```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 [10]:
```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.

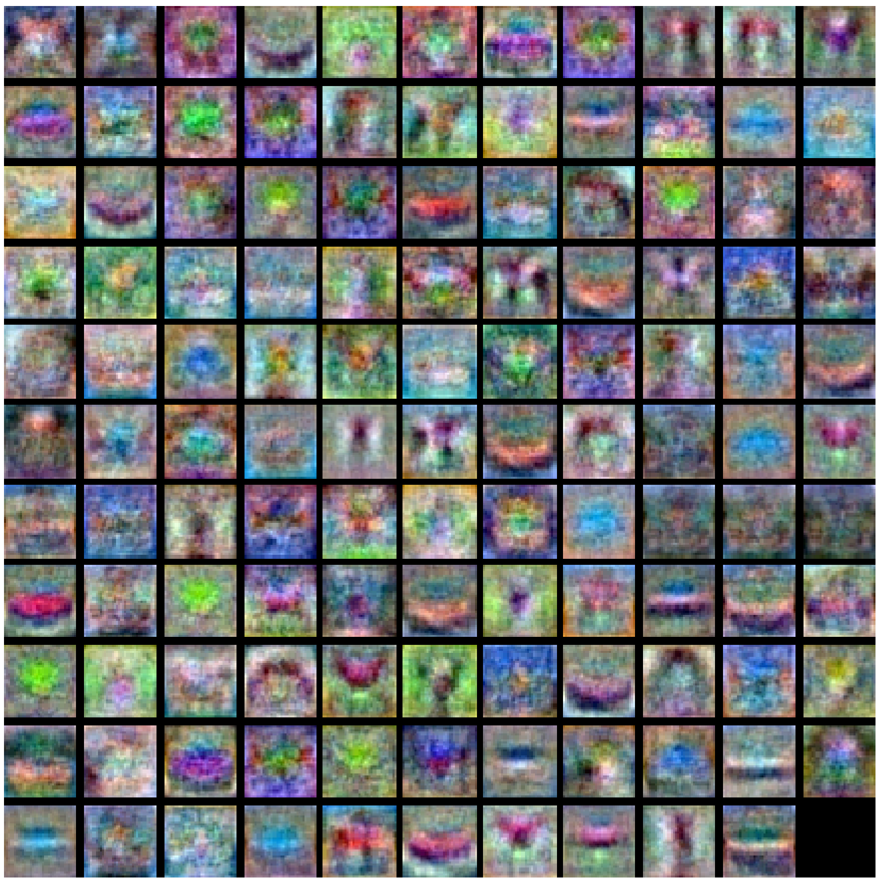
Experiment: You goal in this exercise is to get as good of a result on CIFAR-10 as you can, with a fully-connected Neural Network. Feel free implement your own techniques (e.g. PCA to reduce dimensionality, or adding dropout, or adding features to the solver, etc.).

In [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         #
# automatically like we did on the previous exercises.                         #
#################################################################################

input_size = X_train.shape[1]
hidden_size = 120
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_la
# :(
learning_rates = np.array([0.7, 0.8, 0.9, 1, 1.1])*1e-3
regularization_strengths = [0.5, 0.6, 0.7]

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: 0.500000, val_acc: 0.477000
lr: 0.000700, reg: 0.600000, val_acc: 0.469000
lr: 0.000700, reg: 0.700000, val_acc: 0.473000
lr: 0.000800, reg: 0.500000, val_acc: 0.485000
lr: 0.000800, reg: 0.600000, val_acc: 0.471000
lr: 0.000800, reg: 0.700000, val_acc: 0.471000
lr: 0.000900, reg: 0.500000, val_acc: 0.482000
lr: 0.000900, reg: 0.600000, val_acc: 0.474000
lr: 0.000900, reg: 0.700000, val_acc: 0.486000
lr: 0.001000, reg: 0.500000, val_acc: 0.496000
lr: 0.001000, reg: 0.600000, val_acc: 0.474000
lr: 0.001000, reg: 0.700000, val_acc: 0.474000
lr: 0.001100, reg: 0.500000, val_acc: 0.503000
lr: 0.001100, reg: 0.600000, val_acc: 0.472000
lr: 0.001100, reg: 0.700000, val_acc: 0.505000
Best validation accuracy: 0.505000
```

In [12]:

```
# 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 [13]:
```
test_acc = (best_net.predict(X_test) == y_test).mean()
print('Test accuracy: ', test_acc)
```

```
Test accuracy:  0.504
```

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 can decrease the gap.

Your explanation: The large dataset provide more distinct information for the network, that helps to increase the accuracy. The network becomes more adaptable to changes in the data. However, if the dataset increases it may cause overfitting. Increasing the hidden units also increases the accuracy. It will have more weights to cover the large amount of inputs. It may also cause overfitting similar to the larger dataset case. The regularization strength's aim is to avoid the overfitting case, therefore it may help to increase the accuracy. If there is overfitting in the network, regularization strength make network to work properly. However, increasing the regularization strength highly causes a slow in the learning of the network, and underfitting.