Assignment 1

Beste Aydemir
21703033
EEE 443: Neural Networks
Bilkent University
beste.aydemir@ug.bilkent.edu.tr

October 19, 2020

Question 1.

The MAP estimate for the network weights are found from the following expression:

$$\hat{X}_{MAP} = \underset{W}{\text{argmax}} \frac{P(X, Y \mid W)P(W)}{P(X, Y)} \tag{1}$$

where $P(W)$ is the prior distribution of network weights, $P(X, Y|W)$ is the probability of the realization of input to Y, given W. $P(X, Y)$ indicates the distribution of X and Y, which is not relevant in $argmax$ operator. So the equation reduces to the following. The equation can also be written as the product of each of the N samples x^n, y^n.

$$\hat{X}_{MAP} = \underset{W}{\text{argmax}} P(X, Y \mid W)P(W) \tag{2}$$

$$\hat{X}_{MAP} = \underset{W}{\text{argmax}} \prod_n P(y^n, x^n \mid W)P(W) \tag{3}$$

Since the optimization problem given in the question consists of the summation of the squared error between y^n and $h(x^n, W)$, a similar expression can be obtained by modelling the error between y^n and $h(x^n, W)$ as a Gaussian distribution and then taking its log. The mean of the error is chosen to be 0, since no bias is relevant. Then, the result y^n can be modeled as a shifted version of the error distribution, since $h(x^n, W)$ is not a distribution but a constant. The distribution of the realization of x^n and y^n given W, $P(y^n, x^n \mid W)$, is given by Eq. 6.

$$\varepsilon^n = y^n - h(x^n, W) \sim N(0, \sigma^2) \tag{4}$$

$$y^n \sim N(h(x^n, W), \sigma^2) \tag{5}$$

$$P(y^n, x^n \mid W) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{(-\frac{(y^n - h(x^n, W))^2}{2\sigma^2})} \tag{6}$$

The equation given in the problem statement is:

$$\underset{W}{\text{argmin}} \sum_n (y^n - h(x^n, W))^2 + \beta \sum_i w_i^2 \tag{7}$$

Since there is a similar expression on the exponential of Eq. 6, the log of the Eq. 3 can be taken, while combining Eq. 3. This leads to the following equation. Also the signs are reversed to also give argmin operation.

$$\underset{W}{\text{argmin}} \sum_n \left[-\log\left(\frac{1}{\sqrt{2\pi\sigma^2}}\right) + \frac{\left(y^n - h\left(x^n, w\right)\right)^2}{2\sigma^2} \right] - \log P(W) \tag{8}$$

Next, the data dependent and weight dependent terms will be compared on Eq. 8 and Eq. 7. The data dependent terms give the following. The variance of the error can inferred to be $1/sqrt(2)$.

The argmin part is omitted and the equations are assumed to be equated, in order for MAP estimate to be found with the given equation in the assignment (without any additional terms).

$$\sum_n \left(y^n - h\left(x^n, w\right)\right)^2 = \sum_n \frac{y^n - h\left(x^n, w\right)}{2\sigma^2} \tag{9}$$

$$2\sigma^2 = 1, \quad \sigma = \frac{1}{\sqrt{2}} \tag{10}$$

Similarly, weight dependent terms give:

$$-\log P(w) = \beta \sum_i W_i^2 \tag{11}$$

$$f(W) = Ke^{-\beta \sum w_i^2} \tag{12}$$

where

$$\frac{1}{\beta} = \frac{1}{\sigma\sqrt{2}}, \sigma_1 = \frac{\beta}{\sqrt{2}}, \quad K = \frac{1}{\sigma_1\sqrt{2\pi}} \tag{13}$$

which is multivariate normal with variance $\beta\frac{}{\sqrt{2}.}$

Question 2.

A neural network with four binary input neurons, a hidden layer including four neurons and a single output neuron with a unipolar activation function will be constructed to implement the following logic function:

$$F = (X_1 + \bar{X}_2) \oplus (\bar{X}_3 + \bar{X}_4) \tag{14}$$

X_1	X_2	X_3	X_4	F
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

Table 1: Logic table for F.

a) Since given logic function includes XOR operation of the outputs of $(X_1 + \bar{X}_2)$ and $(\bar{X}_3 + \bar{X}_4)$, it should have more than one decision boundary, leading to more than one hidden layer in implementation. So, in order to implement the logic function with one hidden layer, sum of the products form of the logic function will be used.

$$F = (X_1 + \bar{X}_2)\overline{(\bar{X}_3 + \bar{X}_4)} + \overline{(X_1 + \bar{X}_2)}(\bar{X}_3 + \bar{X}_4) \tag{15}$$
$$= (X_1 + \bar{X}_2)(X_3 X_4) + (\bar{X}_1 X_2)(\bar{X}_3 + \bar{X}_4) \tag{16}$$
$$= (X_1 X_3 X_4) + (\bar{X}_2 X_3 X_4) + (\bar{X}_1 X_2 \bar{X}_3) + (\bar{X}_1 X_2 \bar{X}_4) \tag{17}$$

Each neuron in the hidden layer will be used to implement three input AND operation and their outputs will be directed to OR operation. After the activation function of step function, the output will be equal to the logic function value.

Hidden units will have the following input-output relations.

$$o_1 = f(w_{11}X_1 + w_{13}X_3 + w_{14}X_4 - \theta_1) \tag{18}$$
$$o_2 = f(w_{21}X_1 + w_{23}X_3 + w_{24}X_4 - \theta_2) \tag{19}$$
$$o_3 = f(w_{31}X_1 + w_{33}X_3 + w_{34}X_4 - \theta_3) \tag{20}$$
$$o_4 = f(w_{41}X_1 + w_{43}X_3 + w_{44}X_4 - \theta_4) \tag{21}$$

where f is the step function, w_{ij} is the weight between the j^{th} input and the i^{th} hidden layer neuron, and θ_i are the thresholds.

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

Input and output relations are tabulated below.

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

X_2	X_3	X_4	o_2
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

X_1	X_2	X_3	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

X_1	X_2	X_4	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 2: Logic tables for $(X_1X_3X_4), (\bar{X}_2X_3X_4), (\bar{X}_1X_2\bar{X}_3), (\bar{X}_1X_2\bar{X}_4)$.

o_1	o_2	o_3	o_4	o
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 3: Logic table for $o1 \vee o2 \vee o3 \vee o4$.

According to the logic tables and the step activation function, the inequalities in Table 4 can be found. Every unit in the hidden layer is connected to three input neurons. Not connected neuron is indicated by choosing its weight as 0.The outputs of the hidden units will be directed to an OR operator to give the final output logic.

$$o = f(w_{11}^{(2)} o_1 + w_{12}^{(2)} o_2 + w_{13}^{(2)} o_3 + w_{14}^{(2)} o_4 - \theta_1^{(2)}) \tag{23}$$

The weights are indicated with superscript (2) to show they belong to the weights between the hidden layer and the output. The inequalities for Table 2 and Table 3 can be found on Table 4 and Table 5.

$w_{11}0 + w_{13}0 + w_{14}0 - \theta_1 < 0$		-$\theta_1 < 0$	Choose
$w_{11}0 + w_{13}0 + w_{14}1 - \theta_1 < 0$		$w_{14} < \theta_1$	$w_{11} = 1$,
$w_{11}0 + w_{13}1 + w_{14}0 - \theta_1 < 0$		$w_{13} < \theta_1$	$w_{12} = 0$,
$w_{11}0 + w_{13}1 + w_{14}1 - \theta_1 < 0$	$\Longrightarrow$	$w_{13} + w_{14} < \theta_1$	$w_{13} = 1$,
$w_{11}1 + w_{13}0 + w_{14}0 - \theta_1 < 0$		$w_{11} < \theta_1$	$w_{14} = 1$
$w_{11}1 + w_{13}0 + w_{14}1 - \theta_1 < 0$		$w_{11} + w_{14} < \theta_1$	$n = 3, p = 0$
$w_{11}1 + w_{13}1 + w_{14}0 - \theta_1 < 0$		$w_{11} + w_{13} < \theta_1$	$n - p - 1 < \theta_1 \leq n - p$
$w_{11}1 + w_{13}1 + w_{14}1 - \theta_1 \geq 0$		$w_{11} + w_{13} + w_{14} \geq \theta_1$	$2 < \theta_1 \leq 3$
$w_{22}0 + w_{23}0 + w_{24}0 - \theta_2 < 0$		-$\theta_2 < 0$	Choose
$w_{22}0 + w_{23}0 + w_{24}1 - \theta_2 < 0$		$w_{24} < \theta_2$	$w_{21} = 0$,
$w_{22}0 + w_{23}1 + w_{24}0 - \theta_2 < 0$		$w_{23} < \theta_2$	$w_{22} = -1$,
$w_{22}0 + w_{23}1 + w_{24}1 - \theta_2 \geq 0$	$\Longrightarrow$	$w_{23} + w_{24} \geq \theta_2$	$w_{23} = 1$,
$w_{22}1 + w_{23}0 + w_{24}0 - \theta_2 < 0$		$w_{22} < \theta_2$	$w_{24} = 1$
$w_{22}1 + w_{23}0 + w_{24}1 - \theta_2 < 0$		$w_{22} + w_{24} < \theta_2$	$n = 3, p = 1$
$w_{22}1 + w_{23}1 + w_{24}0 - \theta_2 < 0$		$w_{22} + w_{23} < \theta_2$	$n - p - 1 < \theta_2 \leq n - p$
$w_{22}1 + w_{23}1 + w_{24}1 - \theta_2 < 0$		$w_{22} + w_{23} + w_{24} < \theta_2$	$1 < \theta_2 \leq 2$
$w_{31}0 + w_{32}0 + w_{33}0 - \theta_3 < 0$		-$\theta_3 < 0$	Choose
$w_{31}0 + w_{32}0 + w_{33}1 - \theta_3 < 0$		$w_{33} < \theta_3$	$w_{31} = -1$,
$w_{31}0 + w_{32}1 + w_{33}0 - \theta_3 \geq 0$		$w_{32} < \theta_3$	$w_{32} = 1$,
$w_{31}0 + w_{32}1 + w_{33}1 - \theta_3 < 0$	$\Longrightarrow$	$w_{32} + w_{33} < \theta_3$	$w_{33} = -1$,
$w_{31}1 + w_{32}0 + w_{33}0 - \theta_3 < 0$		$w_{31} < \theta_3$	$w_{34} = 0$
$w_{31}1 + w_{32}0 + w_{33}1 - \theta_3 < 0$		$w_{31} + w_{33} < \theta_3$	$n = 3, p = 2$
$w_{31}1 + w_{32}1 + w_{33}0 - \theta_3 < 0$		$w_{31} + w_{32} < \theta_3$	$n - p - 1 < \theta_3 \leq n - p$
$w_{31}1 + w_{32}1 + w_{33}1 - \theta_3 < 0$		$w_{31} + w_{32} + w_{33} \geq \theta_3$	$0 < \theta_3 \leq 1$
$w_{41}0 + w_{42}0 + w_{44}0 - \theta_4 < 0$		-$\theta_4 < 0$	Choose
$w_{41}0 + w_{42}0 + w_{44}1 - \theta_4 < 0$		$w_{44} < \theta_4$	$w_{41} = -1$,
$w_{41}0 + w_{42}1 + w_{44}0 - \theta_4 \geq 0$		$w_{42} < \theta_4$	$w_{42} = 1$,
$w_{41}0 + w_{42}1 + w_{44}1 - \theta_4 < 0$	$\Longrightarrow$	$w_{42} + w_{44} \geq \theta_4$	$w_{43} = 0$,
$w_{41}1 + w_{42}0 + w_{44}0 - \theta_4 < 0$		$w_{41} < \theta_4$	$w_{44} = -1$
$w_{41}1 + w_{42}0 + w_{44}1 - \theta_4 < 0$		$w_{41} + w_{44} < \theta_4$	$n = 3, p = 2$
$w_{41}1 + w_{42}1 + w_{44}0 - \theta_4 < 0$		$w_{41} + w_{42} < \theta_4$	$n - p - 1 < \theta_4 \leq n - p$
$w_{41}1 + w_{42}1 + w_{44}1 - \theta_4 < 0$		$w_{41} + w_{42} + w_{44} < \theta_4$	$0 < \theta_4 \leq 1$

Table 4: The set of inequalities based on which a set of weights and an activation threshold can be selected.

$$
\begin{array}{|c|}
\hline
\mathrm{w}_{11}^{(2)}0 + w_{12}^{(2)}0 + w_{13}^{(2)}0 + w_{14}^{(2)}0 - \theta_1^{(2)} < 0 \\
\mathrm{w}_{11}^{(2)}0 + w_{12}^{(2)}0 + w_{13}^{(2)}0 + w_{14}^{(2)}1 - \theta_1^{(2)} \geq 0 \\
\mathrm{w}_{11}^{(2)}0 + w_{12}^{(2)}0 + w_{13}^{(2)}1 + w_{14}^{(2)}0 - \theta_1^{(2)} \geq 0 \\
\mathrm{w}_{11}^{(2)}0 + w_{12}^{(2)}0 + w_{13}^{(2)}1 + w_{14}^{(2)}1 - \theta_1^{(2)} \geq 0 \\
\mathrm{w}_{11}^{(2)}0 + w_{12}^{(2)}1 + w_{13}^{(2)}0 + w_{14}^{(2)}0 - \theta_1^{(2)} \geq 0 \\
\mathrm{w}_{11}^{(2)}0 + w_{12}^{(2)}1 + w_{13}^{(2)}0 + w_{14}^{(2)}1 - \theta_1^{(2)} \geq 0 \\
\mathrm{w}_{11}^{(2)}0 + w_{12}^{(2)}1 + w_{13}^{(2)}1 + w_{14}^{(2)}0 - \theta_1^{(2)} \geq 0 \\
\mathrm{w}_{11}^{(2)}0 + w_{12}^{(2)}1 + w_{13}^{(2)}1 + w_{14}^{(2)}1 - \theta_1^{(2)} \geq 0 \\
\mathrm{w}_{11}^{(2)}1 + w_{12}^{(2)}0 + w_{13}^{(2)}0 + w_{14}^{(2)}0 - \theta_1^{(2)} \geq 0 \\
\mathrm{w}_{11}^{(2)}1 + w_{12}^{(2)}0 + w_{13}^{(2)}0 + w_{14}^{(2)}1 - \theta_1^{(2)} \geq 0 \\
\mathrm{w}_{11}^{(2)}1 + w_{12}^{(2)}0 + w_{13}^{(2)}1 + w_{14}^{(2)}0 - \theta_1^{(2)} \geq 0 \\
\mathrm{w}_{11}^{(2)}1 + w_{12}^{(2)}0 + w_{13}^{(2)}1 + w_{14}^{(2)}1 - \theta_1^{(2)} \geq 0 \\
\mathrm{w}_{11}^{(2)}1 + w_{12}^{(2)}1 + w_{13}^{(2)}0 + w_{14}^{(2)}0 - \theta_1^{(2)} \geq 0 \\
\mathrm{w}_{11}^{(2)}1 + w_{12}^{(2)}1 + w_{13}^{(2)}0 + w_{14}^{(2)}1 - \theta_1^{(2)} \geq 0 \\
\mathrm{w}_{11}^{(2)}1 + w_{12}^{(2)}1 + w_{13}^{(2)}1 + w_{14}^{(2)}0 - \theta_1^{(2)} \geq 0 \\
\mathrm{w}_{11}^{(2)}1 + w_{12}^{(2)}1 + w_{13}^{(2)}1 + w_{14}^{(2)}1 - \theta_1^{(2)} \geq 0 \\
\hline
\end{array}
\quad \Longrightarrow \quad
\begin{array}{|c|}
\hline
\text{-}\theta_1^{(2)} < 0 \\
\mathrm{w}_{14}^{(2)} \geq \theta_1^{(2)} \\
\mathrm{w}_{13}^{(2)} \geq \theta_1^{(2)} \\
\mathrm{w}_{13}^{(2)} + w_{14}^{(2)} \geq \theta_1^{(2)} \\
\mathrm{w}_{12}^{(2)} \geq \theta_1^{(2)} \\
\mathrm{w}_{12}^{(2)} + w_{14}^{(2)} \geq \theta_1^{(2)} \\
\mathrm{w}_{12}^{(2)} + w_{13}^{(2)} \geq \theta_1^{(2)} \\
\mathrm{w}_{12}^{(2)} + w_{13}^{(2)} + w_{14}^{(2)} \geq \theta_1^{(2)} \\
\mathrm{w}_{11}^{(2)} + w_{14}^{(2)} \geq \theta_1^{(2)} \\
\mathrm{w}_{11}^{(2)} + w_{13}^{(2)} \geq \theta_1^{(2)} \\
\mathrm{w}_{11}^{(2)} + w_{13}^{(2)} + w_{14}^{(2)} \geq \theta_1^{(2)} \\
\mathrm{w}_{11}^{(2)} + w_{12}^{(2)} \geq \theta_1^{(2)} \\
\mathrm{w}_{11}^{(2)} + w_{12}^{(2)} + w_{14}^{(2)} \geq \theta_1^{(2)} \\
\mathrm{w}_{11}^{(2)} + w_{12}^{(2)} + w_{13}^{(2)} \geq \theta_1^{(2)} \\
\mathrm{w}_{11}^{(2)} + w_{12}^{(2)} + w_{13}^{(2)} + w_{14}^{(2)} \geq \theta_1^{(2)} \\
\hline
\end{array}
$$

Choose

$$
\begin{gathered}
w_{11}^{(2)} = 1, \\
w_{12}^{(2)} = 1, \\
w_{13}^{(2)} = 1, \\
w_{14}^{(2)} = 1 \\
p = 0 \\
-p < \theta_1^{(2)} \leq 1 - p \\
0 < \theta_1^{(2)} \leq 1
\end{gathered}
$$

Table 5: Inequalities for second layer weight selection.

b) As explained in part a, the weight vectors are chosen in the following manner. Thetas are chosen arbitrarily between found intervals.

$$w_1 = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \end{bmatrix} w_2 = \begin{bmatrix} 0 \\ -1 \\ 1 \\ 1 \end{bmatrix} w_3 = \begin{bmatrix} -1 \\ 1 \\ -1 \\ 0 \end{bmatrix} w_4 = \begin{bmatrix} -1 \\ 1 \\ 0 \\ -1 \end{bmatrix} w_1^{(2)} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \tag{24}$$

$$\theta_1 = 2.2, \theta_2 = 1.2, \theta_3 = 0.2, \theta_4 = 0.2, \theta_1^{(2)} = 0.2 \tag{25}$$

This network is shown to achieve 100% accuracy in the code.

c) In order to find the most robust decision boundary, the thetas can be chosen to be in the middle of the intervals.

$$\theta_1 = 2.5, \theta_2 = 1.5, \theta_3 = 0.5, \theta_4 = 0.5, \theta_1^{(2)} = 0.5 \tag{26}$$

If thetas are chosen to be closer to the interval endpoints, fluctuations on the inputs can land on the other side of the decision boundary, causing errors.

d) The validation performance was 74.25% for the network designed in Part a. Choosing the middle of the intervals gave 90.25% performance. The best performance was 91.0% for theta values close to the middle of the intervals (0.52).

Question 3.

a) One sample image from each class and the correlation coefficients between those images are shown in a 26x26 matrix. The matrix is symmetric and the main diagonal is composed of 1's, a letter's correlation with itself. Higher values indicate the similarity between the letters, like 0.56 between letter A and G $(X_{1,7})$. Lower values, like -0.01 between A and Z $(X_{1,26})$, show the difference in shapes. It can be expected the network to classify the letters with high correlation as each other. Also, since the data set includes both uppercase and lowercase of letters, the letters having different forms, such as G, will have higher within class variability compared to letters like O and X.

Figure 1: Sample images of 26 classes.

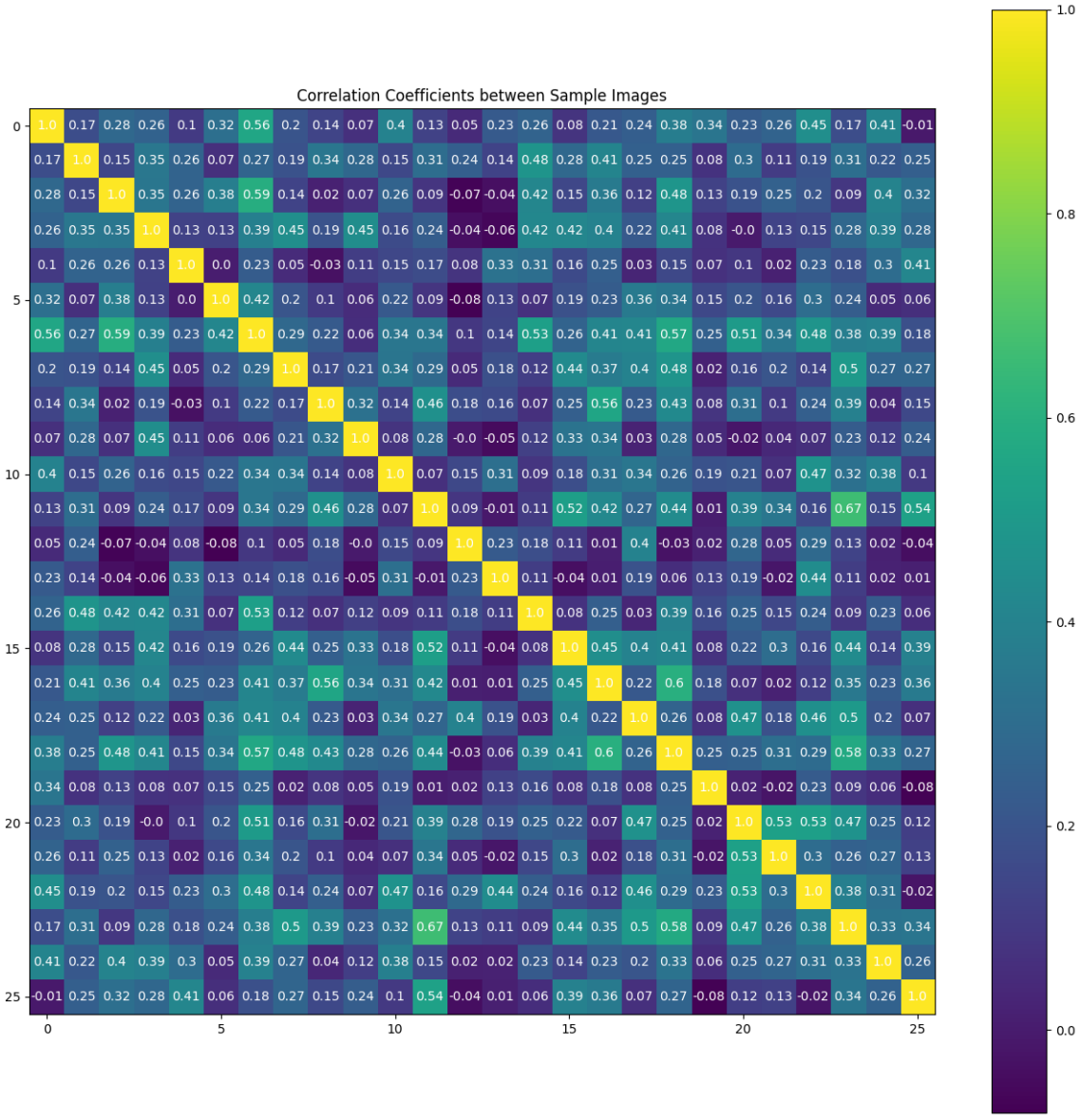


Figure 2: Correlation coefficient matrix between each pair of 26 sample images.

b) After tuning, η^* was chosen to be 0.325 and λ to be 1, with the final MSE value of $3.42e-04$. The network weights are visualized in Figure 3. Hyperparameter tuning part can be found in the code. The weights corresponding to the letters with similar uppercase and lowercase versions seem to have the most clear weights, such as S,U,Z,X. However, for letters like E and H are not very clear. Besides from the within-class variability, the lack of across class variability might have hindered the weight clarity. The letters I and L might be unclear due to this reason.

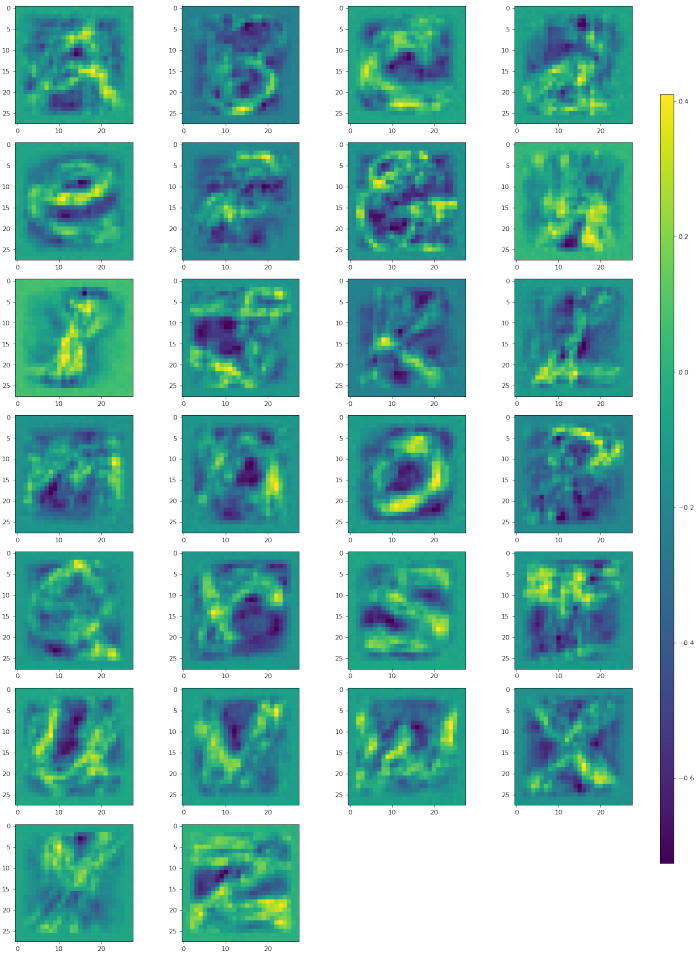


Figure 3: Weight images from the best learning rate.

c) The plots of MSE at each iteration are given below. The plots are shown on seperate figures due to overlapping. All plots are not smooth and show noisy behavior, since the MSE is not calculated for batches but rather one image. Also, learned weights and MSE are highly dependent on the data and the way it is sampled in training.

For $\eta_{low} = 0.0001$, the learning was too slow for the network to progress and decrease overall

MSE, which measures the deviation of each image's output from its true label.

For $\eta^* = 0.325$, the network learned the weights effectively and the average MSE was lower than other two results, as shown in the code. This can also be seen from the testing accuracy being the highest among three.

For $\eta^* = 20$,the learning rate was to high to gradually move towards the lower loss region for W. Even though the loss seems to be on the same level as the best learning rate, the average MSE is higher, as shown on the code.

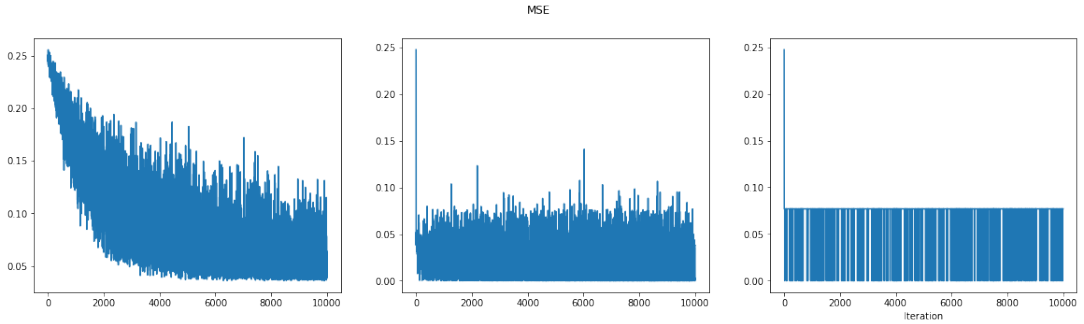


Figure 4: Weight images from the best learning rate.

d) Average MSE over testing samples was $0.055, 0.026, 0.073$ for low, best and high learning rates. The test accuracy was $19.46, 54.84, 3.84$ respectively. It's clear that the best learning rate performs better than the other two. Low learning rate fails to achieve acceptable performance but it still performs better than very high learning rate, which seems to have paralyzed the network.

Question 4.

A two-layer network containing D inputs, H hidden layer neurons, and C classes is implemented.

The scores, which will be directed to softmax function, are calculated for the randomly constructed toy data. Toy data consists of N (5) inputs of size D (4). The hidden layer activations are found by the following.

$$H_1 = ReLU(XW_1 + b_1) \tag{27}$$

ReLU activation function does not have the saturation problem in sigmoid activation functions. ReLu function is simpler to calculate and can lead to sparse representation with the possibility to accelerate learning [2].

Then, the scores are found by:

$$f_y = H_1 W_2 + b_2 \tag{28}$$

The scores for C classes are directed to a Softmax function, in which the probabilities are calculated. Then, cross-entropy loss, which is composed of data loss and regression loss is calculated [1]. The data loss is calculated by finding the probability of correct classes for each data instance and getting its minus log. Regularization loss is found by multiplying the sum of the square of each weight value with a regularization strength constant, $\lambda = 0.05$ in this case.

For one data instance data loss L is [1]:

$$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right) \tag{29}$$

$$L = \frac{1}{N}\sum_i L_i + \frac{1}{2}\lambda \sum_k \sum_l W_{k,l}^2 \tag{30}$$

In the Backward Pass part, the gradient of the loss with respect to W_1, W_2, b_1 and b_2 are calculated from the previously written backward pass, which uses the chain rule, and a numeric approach in $eval\_numerical\_gradient$.

In the numeric approach, a small h is added and subtracted from the parameter of interest and the loss is found for those values. From that, the gradient is calculated. The relative error between the two gradients is found for all parameters. The relative error between the two was less than 1.79e-13.

The network is trained for CIFAR-10 data set using SGD. The network works for inputs of size D = 3072, with a hidden layer of H = 50 to C = 10 classes. 200 randomly chosen training examples (batch size) are used per step in updating the parameters. This is repeated for 1000 iterations. Also after each epoch (every data instance gets a chance to update the parameter) containing multiple iterations, the learning rate is multiplied by a decay rate. The validation accuracy averaged over epochs is 0.287.

To debug the training, loss in each iteration and classification accuracy in each epoch for training and validation sets are plotted. The loss is decreasing somewhat linearly, which indicates a low learning rate, according to the notebook. Also, the first layer's weight visualizations are not reminiscent of patterns.

In order to tune the hyperparameters (hidden layer size, learning rate, number of training epochs, regularization strength, and learning rate decay) validation accuracy is investigated. Also, training accuracy is calculated to give information about overfitting.

Since it was stated that the training and validation accuracy are close to each other and therefore no overfitting, the model can be expanded by increasing the hidden layer neuron number H.

The gap between low testing accuracy and higher training accuracy, can be caused by overfitting or the training data failing to capture the complexity of the phenomena, leading to underfitting. This problem can be solved by training on a larger dataset and increasing the regularization strength.

Training on a larger dataset can reduce the bias in the dataset, if any. So, it can work to generalize the model by introducing more variation in training.

Adding more hidden units makes the model more complicated, causing overfitting. As mentioned in hyperparameter tuning part, increasing H effectively increases training accuracy while decreasing the validation accuracy.

Increasing the regularization strength in the case of overfitting can solve this problem. Since, regularization pulls the weights down, which reduces the ReLU activation function's effect in training.

References

[1] "CS231n Convolutional Neural Networks for Visual Recognition" [Online]. Available: https://cs231n.github.io/neural-networks-case-study/#grad. [Accessed: 17 Oct. 2020]

[2] "A Gentle Introduction to the Rectified Linear Unit (ReLU)" *Machine Learning Mastery*. [Online]. Available: https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/. [Accessed: 17 Oct. 2020]

```
import sys
import numpy as np
import h5py
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import random
question = sys.argv[1]


def beste_aydemir_21703033_hw1(question):
   if question == '1':
      ##Question 1
      print(question)


   elif question == '2':
      #Part B
      #Generate binary inputs
      nums = np.array(range(16))
      bin_nums = ((nums.reshape(-1,1) & (2**np.arange(4))) != 0).astype(
         int)
      X = bin_nums[:,::-1].T
      print("Binary inputs:")
      print(X)

      #True output vector for all possible inputs
      true = np.array([[0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0,
         1]])
      print("True output vector:")
      print(true)

      #Found weight vector
      W1 = np.array([[1, 0, 1, 1], [0, -1, 1, 1], [-1,1,-1,0],
         [-1,1,0,-1]])
      t1 = np.array([[2],[1], [0], [0]])
      W2 = np.array([[1,1,1,1]])
      t2 = 0

      place = 0.1 #To add to t1 and t2 and define their place in the
         interval

      #Evaluating performance for a chosen network
      theta1 = t1 + place
      theta2 = t2 + place
      out = output(X,W1,W2,theta1,theta2)
      print("Network output and accuracy for theta1 = [%f, %f, %f, %f]
         and theta2 = %f:" %(theta1[0],theta1[1],theta1[2],theta1[3],
         theta2))
```

```
print(out)
print(accu(out,true))

#Part c
#Most robust decision boundary
place = 0.5
theta1 = t1 + place
theta2 = t2 + place
out = output(X,W1,W2,theta1,theta2)
print(true.shape)
print("Network output and accuracy for theta1 = [%f, %f, %f, %f]
    and theta2 = %f:" %(theta1[0],theta1[1],theta1[2],theta1[3],
    theta2))
print(out)
print(accu(out,true))

#Part d
np.random.seed(0)
#Generating 400 input samples by concating each vector 25 times
    next to each other
X_ext = np.repeat(X, [25], axis=1)

true_ext = np.repeat(true, [25], axis=1)
noise = np.random.normal(0, 0.2, size=(4, 25*16))
X_noise = X_ext + noise

macc = 0
for i in range (100):
  place = i/100
  theta1 = t1 + place
  theta2 = t2 + place
  out = output(X_noise,W1,W2,theta1,theta2)
  acc = accu(out,true_ext)
  if (acc > macc):
    mtheta1 = theta1
    mtheta2 = theta2
    macc = acc
  if (i==50 or i == 20):
    print("Network output and accuracy for theta1 = [%f, %f, %f, %f
        ] and theta2 = %f:" %(theta1[0],theta1[1],theta1[2],theta1
        [3], theta2))
    print(acc)

print("Best network output and accuracy: theta1 = [%f, %f, %f, %f]
     and theta2 = %f:" %(mtheta1[0],mtheta1[1],mtheta1[2],mtheta1
    [3], mtheta2))
print(macc)
```

```
elif question == '3':
   print(question)
   plt.close('all')

   # Getting the data from assign1_data1 file
   filename = "assign1_data1.h5"
   with h5py.File(filename, "r") as f:
      trainims = np.array(f['trainims'])
      trainlbls = np.array(f['trainlbls'])
      testims = np.array(f['testims'])
      testlbls = np.array(f['testlbls'])


   # Part A
   # Plotting sample images from each letter class
   n = 26 #Number of classes
   m = 28*28 #Size of images
   axes = []
   fig = plt.figure(figsize=(14, 20), dpi=80)

   for a in range(n):
      axes.append(fig.add_subplot(7, 4, a + 1))
      im = plt.imshow(np.transpose(trainims[200*a,:,:])) #The image
         is transposed

   cb_ax = fig.add_axes([1, 0.1, 0.02, 0.8])
   cbar = fig.colorbar(im, cax=cb_ax)
   #plt.savefig('samples.png', dpi=100)
   plt.tight_layout()
   plt.show()

   # Displaying correlations in matrix format
   sample_ims = np.zeros((n,m)) #Add the sample images to the rows of
       a matrix
   for i in range(n):
     sample_ims[i,:] = np.reshape(trainims[200*i,:,:], (1,m))

   corr = np.corrcoef(sample_ims)

   fig, ax = plt.subplots(figsize=(16, 16), dpi=80)
   im = ax.imshow(corr)

   # Loop over data dimensions and create text annotations
   for i in range(n):
      for j in range(n):
         text = ax.text(j, i, float(np.round(corr[i,j], 2)),
                    ha="center", va="center", color="w")
```

```
ax.set_title("Correlation Coefficients between Sample Images")
plt.colorbar(im);
#plt.savefig('corr.png', dpi=100)
plt.show()


# Part B
random.seed(1)

# Hyperparameter tuning
lambda1 = [1, 1.05]
eta = list(range(40))
W_array = []
mse_min = 1
mse_best = []
l_best = 0
e_best = 0

# for l in lambda1:
#  for e in eta:
#    print("Lambda", l)
#    print("eta", e/80)
#    W, mse, W_array = train(trainims, trainlbls, l, e/80)


#    print(mse_min)
#    print(mse[-1])
#    #Best MSE
#    if (mse_min > mse[-1]):
#     mse_min = mse[-1]
#     mse_best = mse
#     l_best = l
#     e_best = e/80

l_best = 1
e_best = 0.325
W_best, mse_best, W_array = train(trainims, trainlbls, l_best,
   e_best)
print("Lambda best" , l_best)
print("Eta best" , e_best)


# Showing W
axes = []
fig = plt.figure(figsize=(14, 20), dpi=80)
# fig.suptitle('Sample Images from Letter Classes')
```

```
    for a in range(n):
       axes.append(fig.add_subplot(7, 4, a + 1))
       imag = W_best[a,0:m]
       imag = np.reshape(imag,(28,28)).T
       im = plt.imshow(imag) #The image is transposed

    cb_ax = fig.add_axes([1, 0.1, 0.02, 0.8])
    cbar = fig.colorbar(im, cax=cb_ax)
    # plt.savefig('test3png.png', dpi=1000)
    plt.tight_layout()
    plt.show()

    # Part c
    # Plotting MSE for eta_low and eta_high
    W_low, mse_low, W_array = train(trainims, trainlbls, lambda1 = 1,
       eta = 0.0001)
    W_high, mse_high, W_array = train(trainims, trainlbls, lambda1 =
       1, eta = 20)

    fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(20, 5))
    axes[0].plot( mse_low)
    axes[1].plot( mse_best)
    axes[2].plot( mse_high)
    plt.xlabel('Iteration')
    plt.suptitle('MSE')
    plt.show()
    fig.tight_layout()

    #Part d
    print('Test performance')
    mse_low, accs_low = classification_acc(testims,testlbls,W_low,
       lambda1 = 1)
    mse_best, accs_best = classification_acc(testims,testlbls,W_best,
       lambda1 = 1)
    mse_high, accs_high = classification_acc(testims,testlbls,W_high,
       lambda1 = 1)
    print("Average mse (over samples) for eta_low:", mse_low)
    print("Test accuracy for eta_low:", accs_low)
    print("Average mse (over samples) for eta:", mse_best)
    print("Test accuracy for eta:", accs_best)
    print("Average mse (over samples) for eta_high:", mse_high)
    print("Test accuracy for eta_high:", accs_high)



#For Question 3
```

```
#This function finds a network's output for an imageset and
def classification_acc(ims,lbls,W,lambda1):
  #Find the output of the network
  size = ims.shape[0]
  m = 784
  X = np.reshape(ims.transpose(1,2,0), (m,size))
  X = X.astype('double')
  #X *= 1.0/np.linalg.norm(X, axis=0)
  X *= 1.0/np.max(X, axis=0)
  X_wo = np.append(X, -np.ones((1,size)) ,axis = 0)

  V = np.matmul(W,X_wo)
  O = 1 / ( 1 + np.exp( -V*lambda1 ))

  D = np.zeros((size,26))
  a = np.array(lbls).astype('int')
  a -= 1
  D[np.arange(a.size),a] = 1

  mseavg = (np.sum((D.T-O)*(D.T-O), axis = 0)).mean() /26

  classno = np.argmax(O, axis=0)
  acc = ((lbls-1) == classno).mean()*100

  return mseavg, acc

# This function trains the network according to lambda1 and eta
def train(trainims, trainlbls, lambda1, eta):
  # # Generating weight matrix(784,26) including the beta vector(26,)
  m = 28*28
  n = 26
  sigma = 0.01
  mu = 0
  W = np.random.normal(mu, sigma, (m+1,n))
  W_T = np.transpose(W)

  # Online training with 10000 iterations
  iter_no = 10000
  mse_arr = np.zeros((iter_no,))
  W_arr = np.zeros((iter_no,n,m+1))

  for a in range(iter_no):
    im_index = random.randint(0, 5200-1) #randomly selecting x
    im_x = np.reshape(trainims[im_index,:,:], (m,)) #image (784,)
    im_x = im_x.astype('double')
    im_x *= 1.0/np.max(im_x, axis=0)
    #im_x *= 1.0/np.linalg.norm(im_x)
    x = np.append(im_x, -1) #append -1 for beta
```

```
    V = np.matmul(W_T,x) #(26,785)
    V2 = V*lambda1
    O = 1 / ( 1 + np.exp( -V2 ))

    O = np.reshape(O, (n,))

    D = np.zeros((n,))
    indexlbl = int(trainlbls[im_index]-1)
    D[indexlbl] = 1

    miss = D - O
    mse = (sum(miss*miss))/n
    mse_arr[a] = mse
    f_prime = (lambda1/2) * (1-O*O)
    f_prime = O*(1-O)*lambda1

    learning_sig = np.multiply(miss,f_prime)

    x = np.reshape(x, (1,m+1))

    learning_sig_exp = np.tile(learning_sig, (m+1, 1)).T
    x_exp = np.tile(x, (n, 1))

    delta_W = eta * np.multiply(learning_sig_exp,x_exp)
    W_T = W_T + delta_W
    W_arr[a,:,:] = W_T

  return W_T, mse_arr, W_arr

#For Question 2
#This function gives the logic output for X(4,samples) given W1, W2, t1,
    t2
def output(X, W1, W2, t1, t2):
  v1 = np.matmul(W1,X)
  o1 = ((v1 - t1) >= 0).astype('uint8')
  v2 = np.matmul(W2,o1)
  o2 = ((v2 - t2) >= 0).astype('uint8')
  return o2

#This function returns the accuracy between two arrays as percent
def accu(o1,true):
  return np.sum((o1 == true).astype('uint8'))/(true.shape[1])*100


beste_aydemir_21703033_hw1(question)
```

Implementing a Neural Network

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

In [1]:
```
# A bit of setup
from __future__ import print_function
import numpy as np
import matplotlib.pyplot as plt

from cs231n.classifiers.neural_net import TwoLayerNet



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

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

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

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:
3.6802720745909845e-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 [6]:
```
from cs231n.gradient_check import eval_numerical_gradient

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

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

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

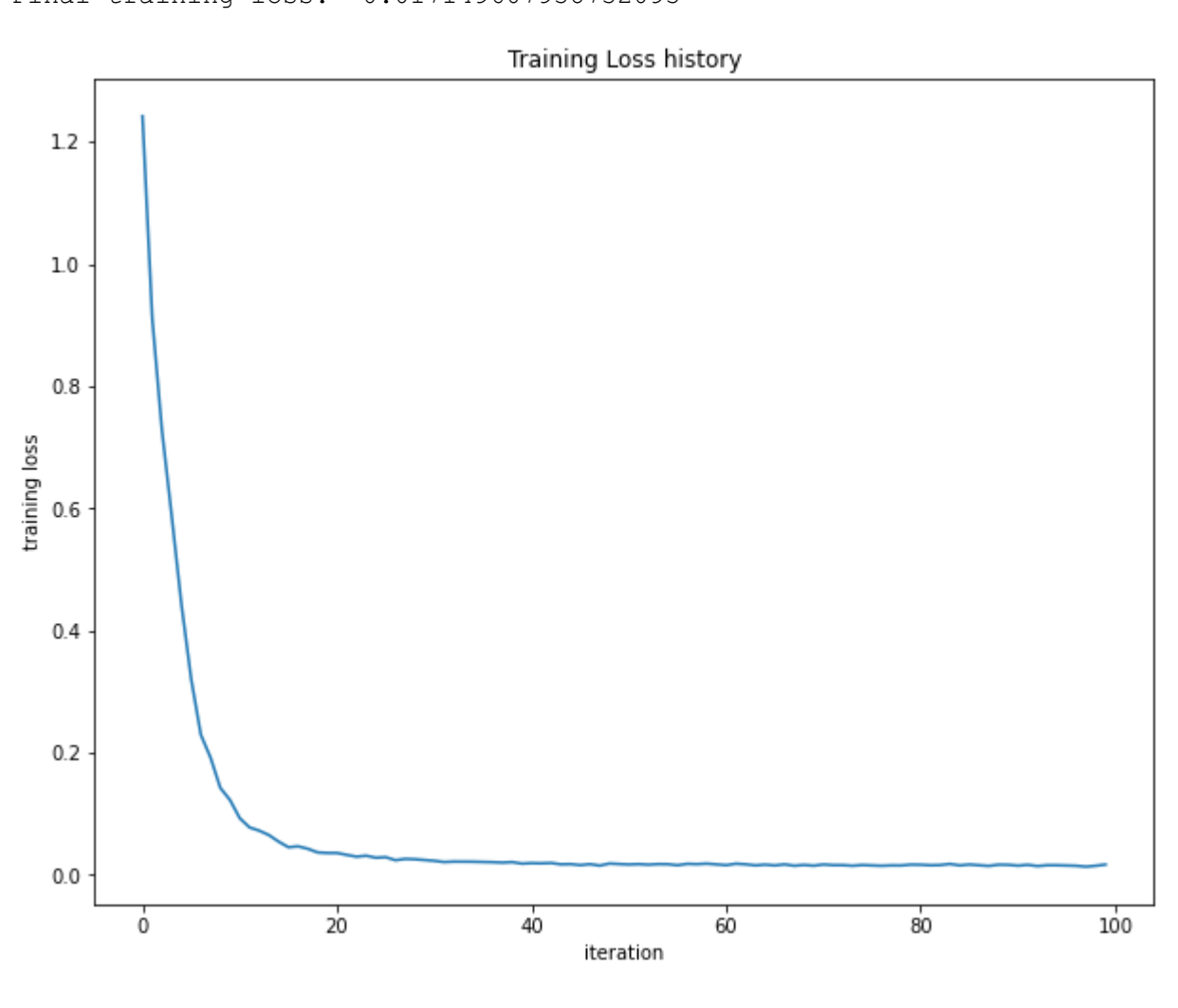
Train the network

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

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

In [7]:
```
net = init_toy_model()
stats = net.train(X, y, X, y,
            learning_rate=1e-1, reg=5e-6,
            num_iters=100, verbose=False)

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

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

```
Final training loss:  0.017149607938732093
```



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 [8]:
```
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,)
```

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 [9]:
```
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
```

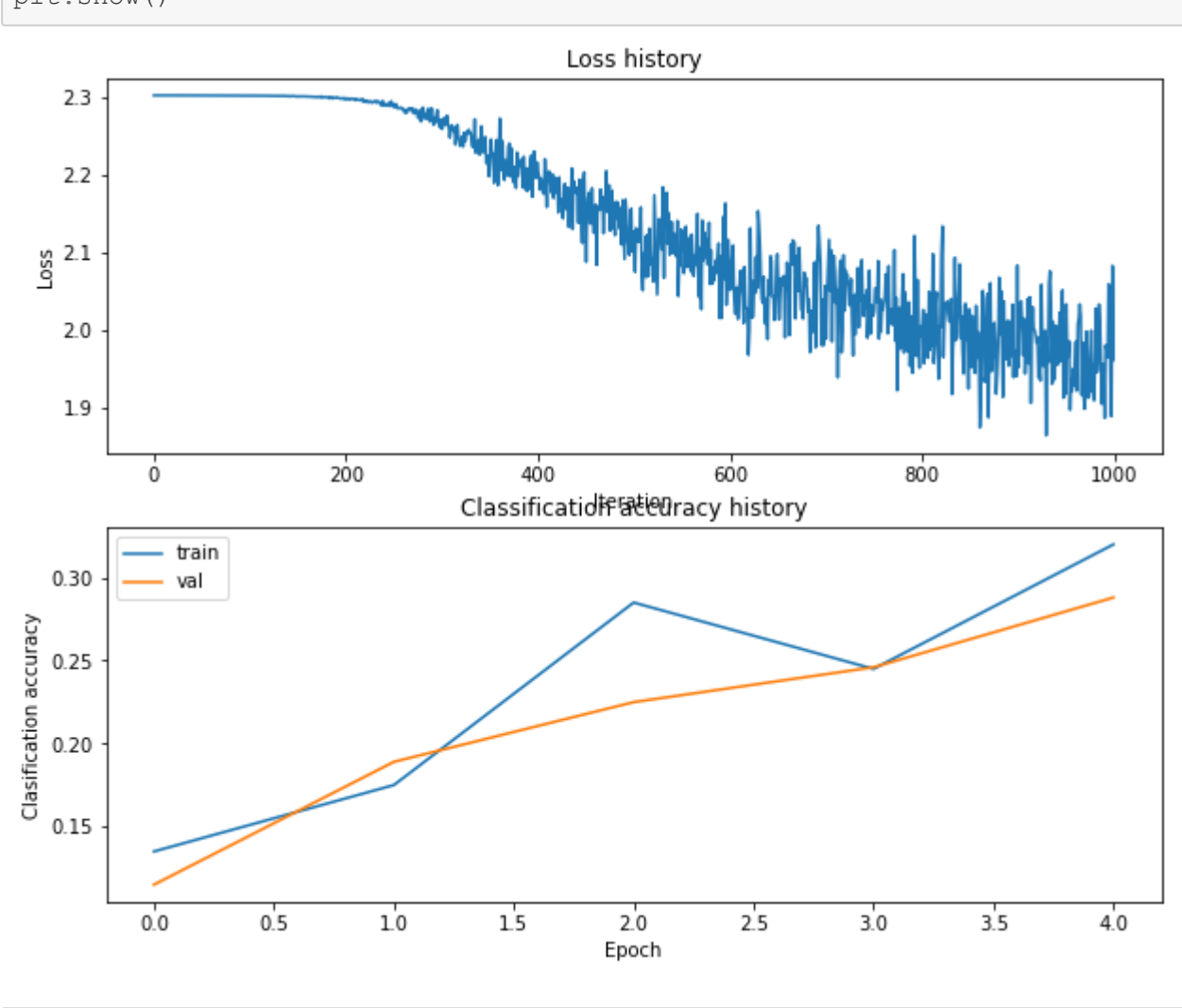
Debug the training

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.

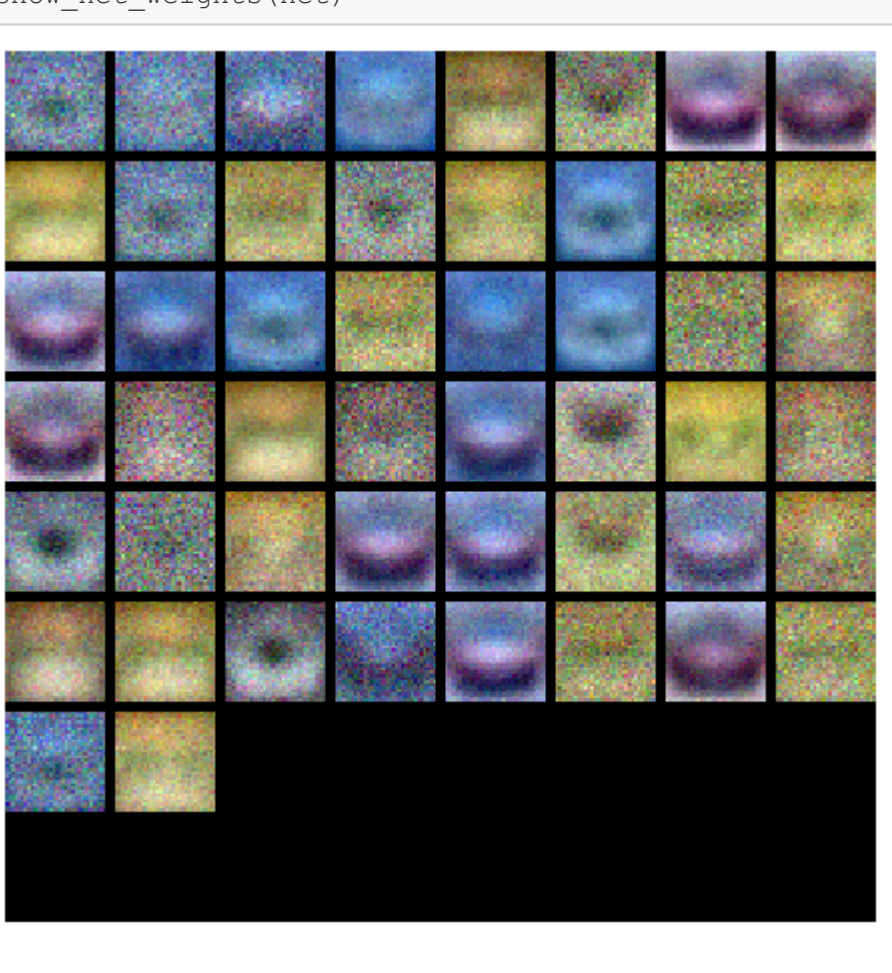
One strategy for getting insight into what's wrong is to plot the loss function and the accuracies on the training and validation sets during optimization.

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 [10]:
```
# 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 [11]:
```
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.

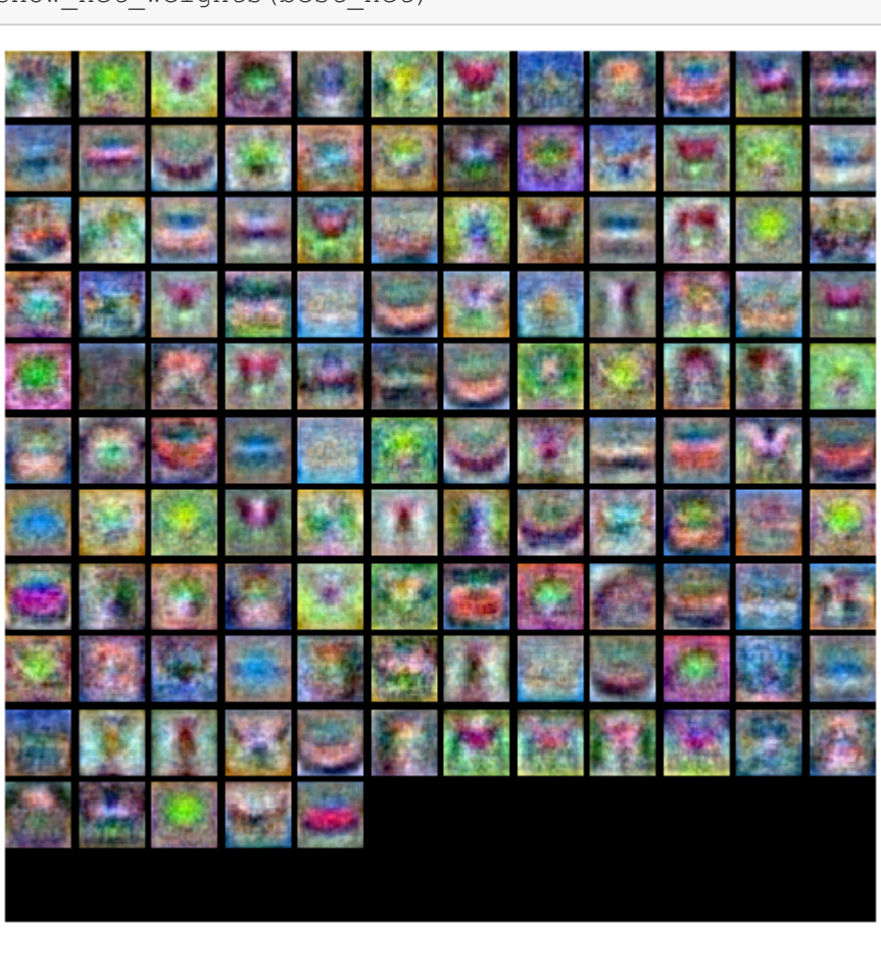
Approximate results. You should be aim to achieve a classification accuracy of greater than 48% on the validation set. Our best network gets over 52% on the validation set.

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 [17]:
```
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_sizes = [ 100, 110, 125, 150]
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, 1.1, 1.25]
lrds = [0.95]
best_val = -1
results_val = {}
results_train = {}

for hidden_size in hidden_sizes:
    for lr in learning_rates:
        for lrd in lrds:
            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, learning_rate_decay = lrd,
reg=reg,
                         num_iters=1500)

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

                y_train_pred = net.predict(X_train)
                train_acc = np.mean(y_train_pred == y_train)

                #print('lr: %f, reg: %f, val_acc: %f' % (lr, reg, val_acc))
                print('hidden_size: %f, lr: %f, lrd: %f, reg: %f, val_acc: %f, train acc: %f' % (hidden
_size, lr, lrd, reg, val_acc, train_acc))

                if val_acc > best_val:
                    best_val = val_acc
                    best_net = net

                results_val[(hidden_size,lr, lrd, reg)] = val_acc
                results_train[(hidden_size,lr, lrd, reg)] = train_acc
print('Best validation accuracy: %f' % best_val)
#################################################################################
#                                END OF YOUR CODE                               #
#################################################################################
```

```
hidden_size: 100.000000, lr: 0.000700, lrd: 0.950000, reg: 0.750000, val_acc: 0.473000, train acc: 0.
493163
hidden_size: 100.000000, lr: 0.000700, lrd: 0.950000, reg: 1.000000, val_acc: 0.464000, train acc: 0.
482163
hidden_size: 100.000000, lr: 0.000700, lrd: 0.950000, reg: 1.100000, val_acc: 0.467000, train acc: 0.
484980
hidden_size: 100.000000, lr: 0.000700, lrd: 0.950000, reg: 1.250000, val_acc: 0.474000, train acc: 0.
477571
hidden_size: 100.000000, lr: 0.000800, lrd: 0.950000, reg: 0.750000, val_acc: 0.471000, train acc: 0.
500143
hidden_size: 100.000000, lr: 0.000800, lrd: 0.950000, reg: 1.000000, val_acc: 0.477000, train acc: 0.
486673
hidden_size: 100.000000, lr: 0.000800, lrd: 0.950000, reg: 1.100000, val_acc: 0.461000, train acc: 0.
482122
hidden_size: 100.000000, lr: 0.000800, lrd: 0.950000, reg: 1.250000, val_acc: 0.468000, train acc: 0.
477449
hidden_size: 100.000000, lr: 0.000900, lrd: 0.950000, reg: 0.750000, val_acc: 0.487000, train acc: 0.
505020
hidden_size: 100.000000, lr: 0.000900, lrd: 0.950000, reg: 1.000000, val_acc: 0.477000, train acc: 0.
492898
hidden_size: 100.000000, lr: 0.000900, lrd: 0.950000, reg: 1.100000, val_acc: 0.478000, train acc: 0.
486776
hidden_size: 100.000000, lr: 0.000900, lrd: 0.950000, reg: 1.250000, val_acc: 0.463000, train acc: 0.
476673
hidden_size: 100.000000, lr: 0.001000, lrd: 0.950000, reg: 0.750000, val_acc: 0.476000, train acc: 0.
493184
hidden_size: 100.000000, lr: 0.001000, lrd: 0.950000, reg: 1.000000, val_acc: 0.466000, train acc: 0.
490408
hidden_size: 100.000000, lr: 0.001000, lrd: 0.950000, reg: 1.100000, val_acc: 0.474000, train acc: 0.
486633
hidden_size: 100.000000, lr: 0.001000, lrd: 0.950000, reg: 1.250000, val_acc: 0.468000, train acc: 0.
480184
hidden_size: 100.000000, lr: 0.001100, lrd: 0.950000, reg: 0.750000, val_acc: 0.476000, train acc: 0.
508612
hidden_size: 100.000000, lr: 0.001100, lrd: 0.950000, reg: 1.000000, val_acc: 0.475000, train acc: 0.
492959
hidden_size: 100.000000, lr: 0.001100, lrd: 0.950000, reg: 1.100000, val_acc: 0.461000, train acc: 0.
490592
hidden_size: 100.000000, lr: 0.001100, lrd: 0.950000, reg: 1.250000, val_acc: 0.458000, train acc: 0.
487408
hidden_size: 110.000000, lr: 0.000700, lrd: 0.950000, reg: 0.750000, val_acc: 0.475000, train acc: 0.
490224
hidden_size: 110.000000, lr: 0.000700, lrd: 0.950000, reg: 1.000000, val_acc: 0.470000, train acc: 0.
480980
hidden_size: 110.000000, lr: 0.000700, lrd: 0.950000, reg: 1.100000, val_acc: 0.473000, train acc: 0.
480061
hidden_size: 110.000000, lr: 0.000700, lrd: 0.950000, reg: 1.250000, val_acc: 0.463000, train acc: 0.
470796
hidden_size: 110.000000, lr: 0.000800, lrd: 0.950000, reg: 0.750000, val_acc: 0.467000, train acc: 0.
497776
hidden_size: 110.000000, lr: 0.000800, lrd: 0.950000, reg: 1.000000, val_acc: 0.480000, train acc: 0.
488857
hidden_size: 110.000000, lr: 0.000800, lrd: 0.950000, reg: 1.100000, val_acc: 0.477000, train acc: 0.
487592
hidden_size: 110.000000, lr: 0.000800, lrd: 0.950000, reg: 1.250000, val_acc: 0.474000, train acc: 0.
481735
hidden_size: 110.000000, lr: 0.000900, lrd: 0.950000, reg: 0.750000, val_acc: 0.492000, train acc: 0.
505959
hidden_size: 110.000000, lr: 0.000900, lrd: 0.950000, reg: 1.000000, val_acc: 0.465000, train acc: 0.
493286
hidden_size: 110.000000, lr: 0.000900, lrd: 0.950000, reg: 1.100000, val_acc: 0.482000, train acc: 0.
488857
hidden_size: 110.000000, lr: 0.000900, lrd: 0.950000, reg: 1.250000, val_acc: 0.448000, train acc: 0.
466959
hidden_size: 110.000000, lr: 0.001000, lrd: 0.950000, reg: 0.750000, val_acc: 0.480000, train acc: 0.
505041
hidden_size: 110.000000, lr: 0.001000, lrd: 0.950000, reg: 1.000000, val_acc: 0.484000, train acc: 0.
495980
hidden_size: 110.000000, lr: 0.001000, lrd: 0.950000, reg: 1.100000, val_acc: 0.485000, train acc: 0.
492367
hidden_size: 110.000000, lr: 0.001000, lrd: 0.950000, reg: 1.250000, val_acc: 0.468000, train acc: 0.
487592
hidden_size: 110.000000, lr: 0.001100, lrd: 0.950000, reg: 0.750000, val_acc: 0.485000, train acc: 0.
503224
hidden_size: 110.000000, lr: 0.001100, lrd: 0.950000, reg: 1.000000, val_acc: 0.465000, train acc: 0.
482224
hidden_size: 110.000000, lr: 0.001100, lrd: 0.950000, reg: 1.100000, val_acc: 0.475000, train acc: 0.
488327
hidden_size: 110.000000, lr: 0.001100, lrd: 0.950000, reg: 1.250000, val_acc: 0.467000, train acc: 0.
487224
hidden_size: 125.000000, lr: 0.000700, lrd: 0.950000, reg: 0.750000, val_acc: 0.475000, train acc: 0.
491673
hidden_size: 125.000000, lr: 0.000700, lrd: 0.950000, reg: 1.000000, val_acc: 0.474000, train acc: 0.
482204
hidden_size: 125.000000, lr: 0.000700, lrd: 0.950000, reg: 1.100000, val_acc: 0.471000, train acc: 0.
482429
hidden_size: 125.000000, lr: 0.000700, lrd: 0.950000, reg: 1.250000, val_acc: 0.461000, train acc: 0.
476878
hidden_size: 125.000000, lr: 0.000800, lrd: 0.950000, reg: 0.750000, val_acc: 0.470000, train acc: 0.
500143
hidden_size: 125.000000, lr: 0.000800, lrd: 0.950000, reg: 1.000000, val_acc: 0.477000, train acc: 0.
489673
hidden_size: 125.000000, lr: 0.000800, lrd: 0.950000, reg: 1.100000, val_acc: 0.481000, train acc: 0.
484714
hidden_size: 125.000000, lr: 0.000800, lrd: 0.950000, reg: 1.250000, val_acc: 0.466000, train acc: 0.
480286
hidden_size: 125.000000, lr: 0.000900, lrd: 0.950000, reg: 0.750000, val_acc: 0.484000, train acc: 0.
500184
hidden_size: 125.000000, lr: 0.000900, lrd: 0.950000, reg: 1.000000, val_acc: 0.467000, train acc: 0.
494837
hidden_size: 125.000000, lr: 0.000900, lrd: 0.950000, reg: 1.100000, val_acc: 0.483000, train acc: 0.
493673
hidden_size: 125.000000, lr: 0.000900, lrd: 0.950000, reg: 1.250000, val_acc: 0.473000, train acc: 0.
481327
hidden_size: 125.000000, lr: 0.001000, lrd: 0.950000, reg: 0.750000, val_acc: 0.490000, train acc: 0.
503204
hidden_size: 125.000000, lr: 0.001000, lrd: 0.950000, reg: 1.000000, val_acc: 0.468000, train acc: 0.
491673
hidden_size: 125.000000, lr: 0.001000, lrd: 0.950000, reg: 1.100000, val_acc: 0.463000, train acc: 0.
489857
hidden_size: 125.000000, lr: 0.001000, lrd: 0.950000, reg: 1.250000, val_acc: 0.472000, train acc: 0.
486755
hidden_size: 125.000000, lr: 0.001100, lrd: 0.950000, reg: 0.750000, val_acc: 0.485000, train acc: 0.
507429
hidden_size: 125.000000, lr: 0.001100, lrd: 0.950000, reg: 1.000000, val_acc: 0.498000, train acc: 0.
503327
hidden_size: 125.000000, lr: 0.001100, lrd: 0.950000, reg: 1.100000, val_acc: 0.491000, train acc: 0.
495959
hidden_size: 125.000000, lr: 0.001100, lrd: 0.950000, reg: 1.250000, val_acc: 0.473000, train acc: 0.
481837
hidden_size: 150.000000, lr: 0.000700, lrd: 0.950000, reg: 0.750000, val_acc: 0.475000, train acc: 0.
496918
hidden_size: 150.000000, lr: 0.000700, lrd: 0.950000, reg: 1.000000, val_acc: 0.468000, train acc: 0.
485571
hidden_size: 150.000000, lr: 0.000700, lrd: 0.950000, reg: 1.100000, val_acc: 0.465000, train acc: 0.
479388
hidden_size: 150.000000, lr: 0.000700, lrd: 0.950000, reg: 1.250000, val_acc: 0.454000, train acc: 0.
476735
hidden_size: 150.000000, lr: 0.000800, lrd: 0.950000, reg: 0.750000, val_acc: 0.473000, train acc: 0.
499265
hidden_size: 150.000000, lr: 0.000800, lrd: 0.950000, reg: 1.000000, val_acc: 0.474000, train acc: 0.
488653
hidden_size: 150.000000, lr: 0.000800, lrd: 0.950000, reg: 1.100000, val_acc: 0.479000, train acc: 0.
486041
hidden_size: 150.000000, lr: 0.000800, lrd: 0.950000, reg: 1.250000, val_acc: 0.466000, train acc: 0.
484102
hidden_size: 150.000000, lr: 0.000900, lrd: 0.950000, reg: 0.750000, val_acc: 0.494000, train acc: 0.
495551
hidden_size: 150.000000, lr: 0.000900, lrd: 0.950000, reg: 1.000000, val_acc: 0.479000, train acc: 0.
490041
hidden_size: 150.000000, lr: 0.000900, lrd: 0.950000, reg: 1.100000, val_acc: 0.477000, train acc: 0.
487082
hidden_size: 150.000000, lr: 0.000900, lrd: 0.950000, reg: 1.250000, val_acc: 0.455000, train acc: 0.
483531
hidden_size: 150.000000, lr: 0.001000, lrd: 0.950000, reg: 0.750000, val_acc: 0.486000, train acc: 0.
509184
hidden_size: 150.000000, lr: 0.001000, lrd: 0.950000, reg: 1.000000, val_acc: 0.490000, train acc: 0.
499429
hidden_size: 150.000000, lr: 0.001000, lrd: 0.950000, reg: 1.100000, val_acc: 0.462000, train acc: 0.
493571
hidden_size: 150.000000, lr: 0.001000, lrd: 0.950000, reg: 1.250000, val_acc: 0.474000, train acc: 0.
484694
hidden_size: 150.000000, lr: 0.001100, lrd: 0.950000, reg: 0.750000, val_acc: 0.497000, train acc: 0.
516102
hidden_size: 150.000000, lr: 0.001100, lrd: 0.950000, reg: 1.000000, val_acc: 0.479000, train acc: 0.
498429
hidden_size: 150.000000, lr: 0.001100, lrd: 0.950000, reg: 1.100000, val_acc: 0.480000, train acc: 0.
494653
hidden_size: 150.000000, lr: 0.001100, lrd: 0.950000, reg: 1.250000, val_acc: 0.464000, train acc: 0.
485694
Best validation accuracy: 0.498000
```

In [18]:
```
# 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 [19]:
```
test_acc = (best_net.predict(X_test) == y_test).mean()
print('Test accuracy: ', test_acc)
```

```
Test accuracy:  0.492
```

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: 1,3.

*Your explanation:The gap between low testing accuracy and higher training accuracy, can be caused by overfitting or the training data failing to capture the complexity of the phenomena, leading to underfitting. This problem can be solved by training on a larger dataset and increasing the regularization strength.

Training on a larger dataset can reduce the bias in the dataset, if any. So, it can work to generalize the model by introducing more variation in training.

Adding more hidden units makes the model more complicated, causing overfitting. As mentioned in hyperparameter tuning part, increasing H effectively increases training accuracy while decreasing the validation accuracy.

Increasing the regularization strength in the case of overfitting can solve this problem due to bias variance trade-off. Also, regularization pulls the weights down, which reduces the ReLU activation function's effect in training.

two_layer_net

October 19, 2020

1 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.

[1]:
```
# A bit of setup
from __future__ import print_function
import numpy as np
import matplotlib.pyplot as plt

from cs231n.classifiers.neural_net import TwoLayerNet



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

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

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

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.

[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.

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
```

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
```

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:

[6]:
```
from cs231n.gradient_check import eval_numerical_gradient

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

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

# these should all be less than 1e-8 or so
for param_name in grads:
    f = lambda W: net.loss(X, y, reg=0.05)[0]
```

```
    param_grad_num = eval_numerical_gradient(f, net.params[param_name],␣
↪verbose=False)
    print('%s max relative error: %e' % (param_name, rel_error(param_grad_num,␣
↪grads[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
```

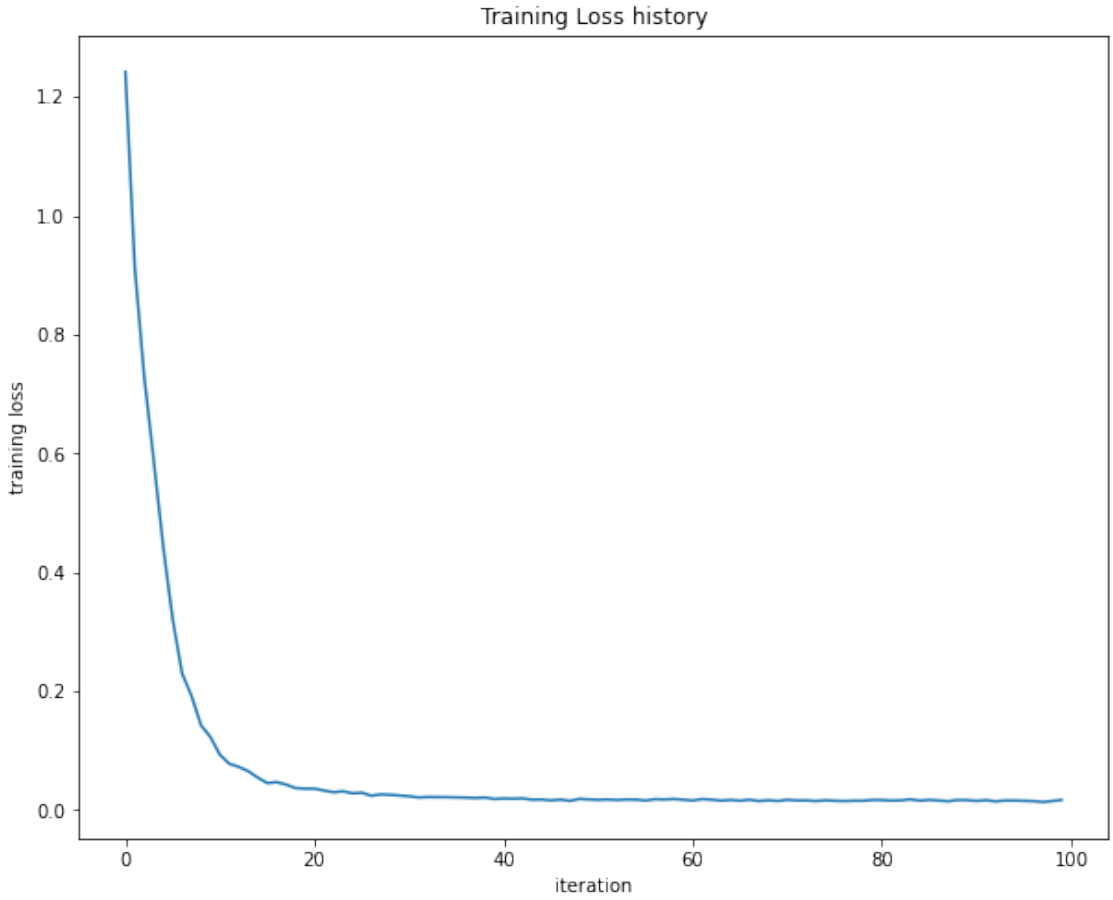
5 Train the network

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.

[7]:
```
net = init_toy_model()
stats = net.train(X, y, X, y,
            learning_rate=1e-1, reg=5e-6,
            num_iters=100, verbose=False)

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

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

```
Final training loss:  0.017149607938732093
```



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.

[8]:
```
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,)
```

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.

[9]:
```
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
```

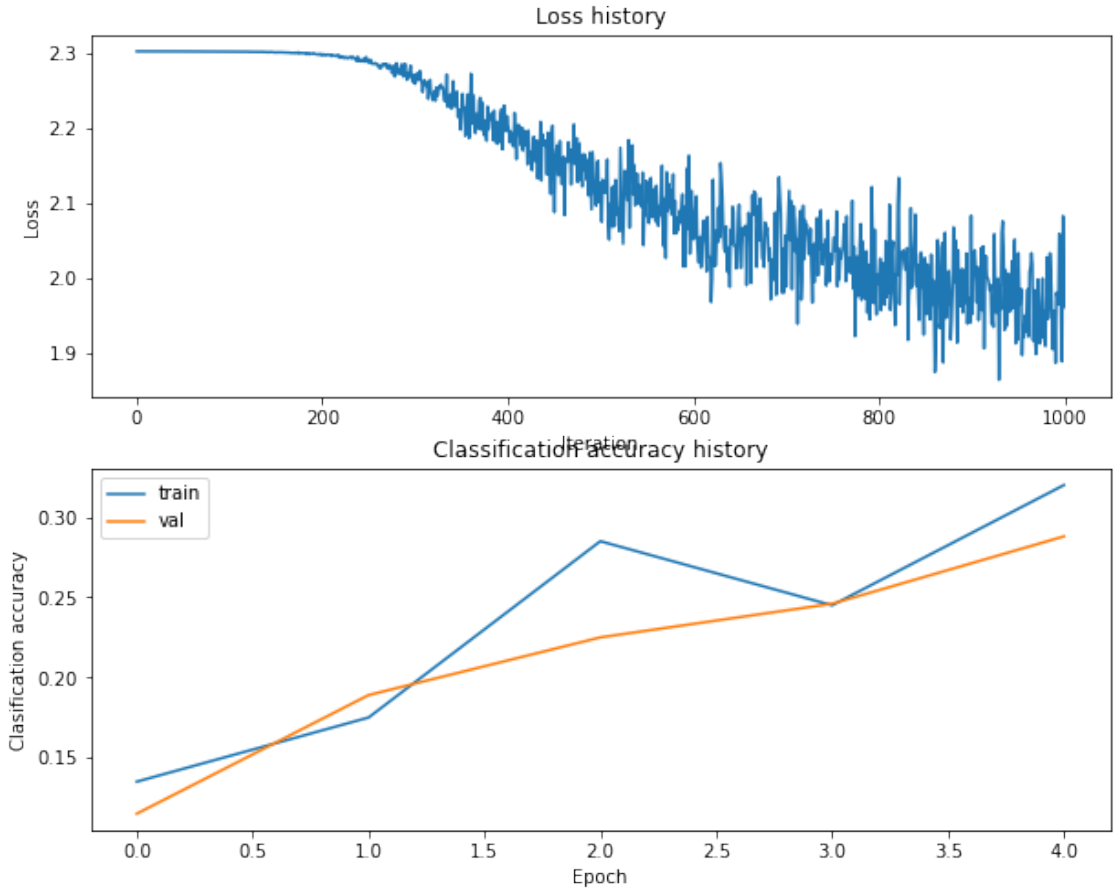
8 Debug the training

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.

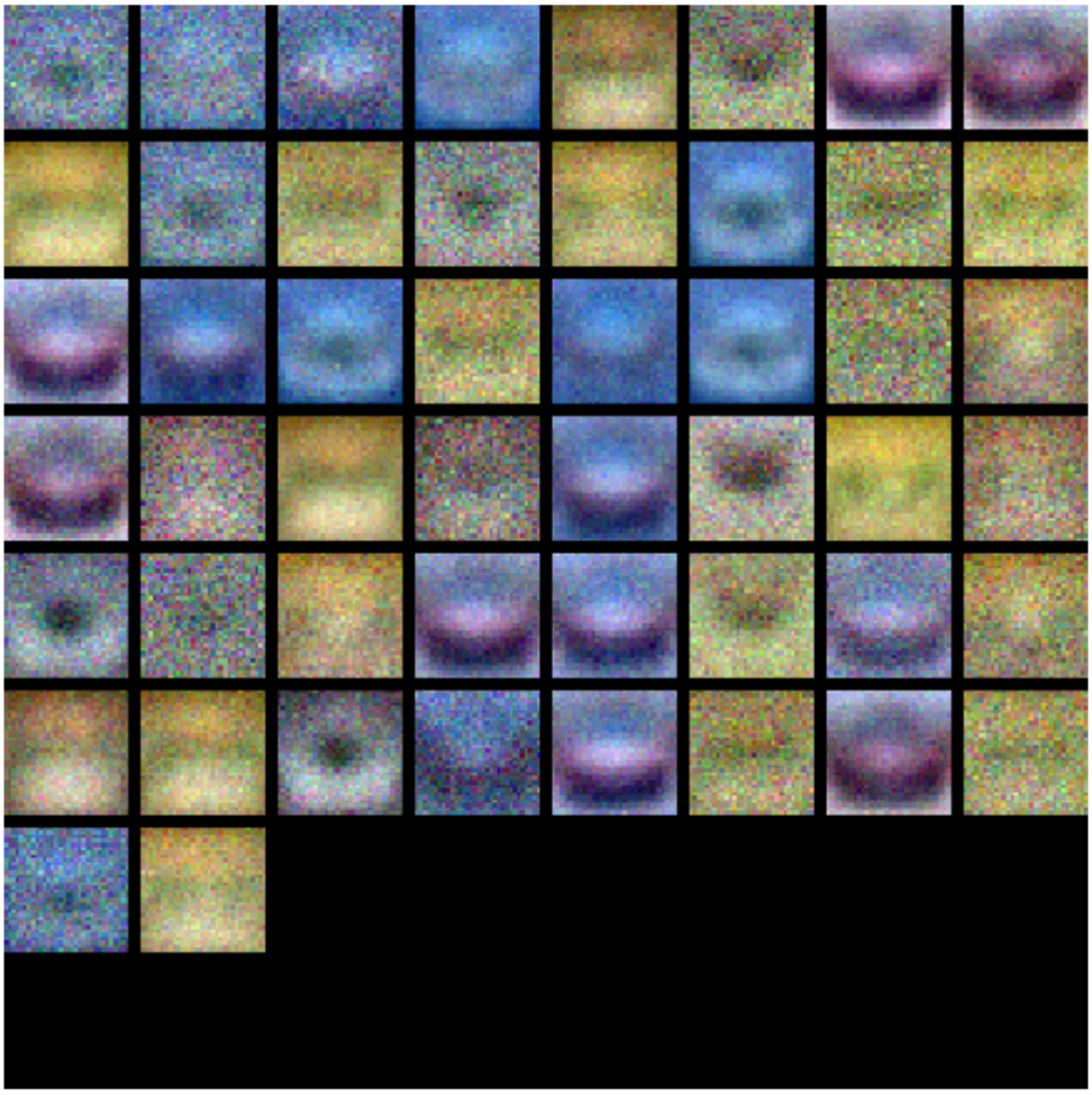
Another strategy is to visualize the weights that were learned in the first layer of the network. In most neural networks trained on visual data, the first layer weights typically show some visible structure when visualized.

[10]:
```
# 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()
```



[11]:
```
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)
```



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.).

[17]:
```
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_sizes = [ 100, 110, 125, 150]
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, 1.1, 1.25]
lrds = [0.95]
best_val = -1
results_val = {}
results_train = {}

for hidden_size in hidden_sizes:
    for lr in learning_rates:
        for lrd in lrds:
            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,␣
 ↪learning_rate_decay = lrd, reg=reg,
                          num_iters=1500)

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

                y_train_pred = net.predict(X_train)
                train_acc = np.mean(y_train_pred == y_train)

                #print('lr: %f, reg: %f, val_acc: %f' % (lr, reg, val_acc))
                print('hidden_size: %f, lr: %f, lrd: %f, reg: %f, val_acc: %f,␣
 ↪train acc: %f' % (hidden_size, lr, lrd, reg, val_acc, train_acc))

                if val_acc > best_val:
                    best_val = val_acc
                    best_net = net

                results_val[(hidden_size,lr, lrd, reg)] = val_acc
                results_train[(hidden_size,lr, lrd, reg)] = train_acc
print('Best validation accuracy: %f' % best_val)
################################################################################
#                              END OF YOUR CODE                                ␣
 ↪#
```

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

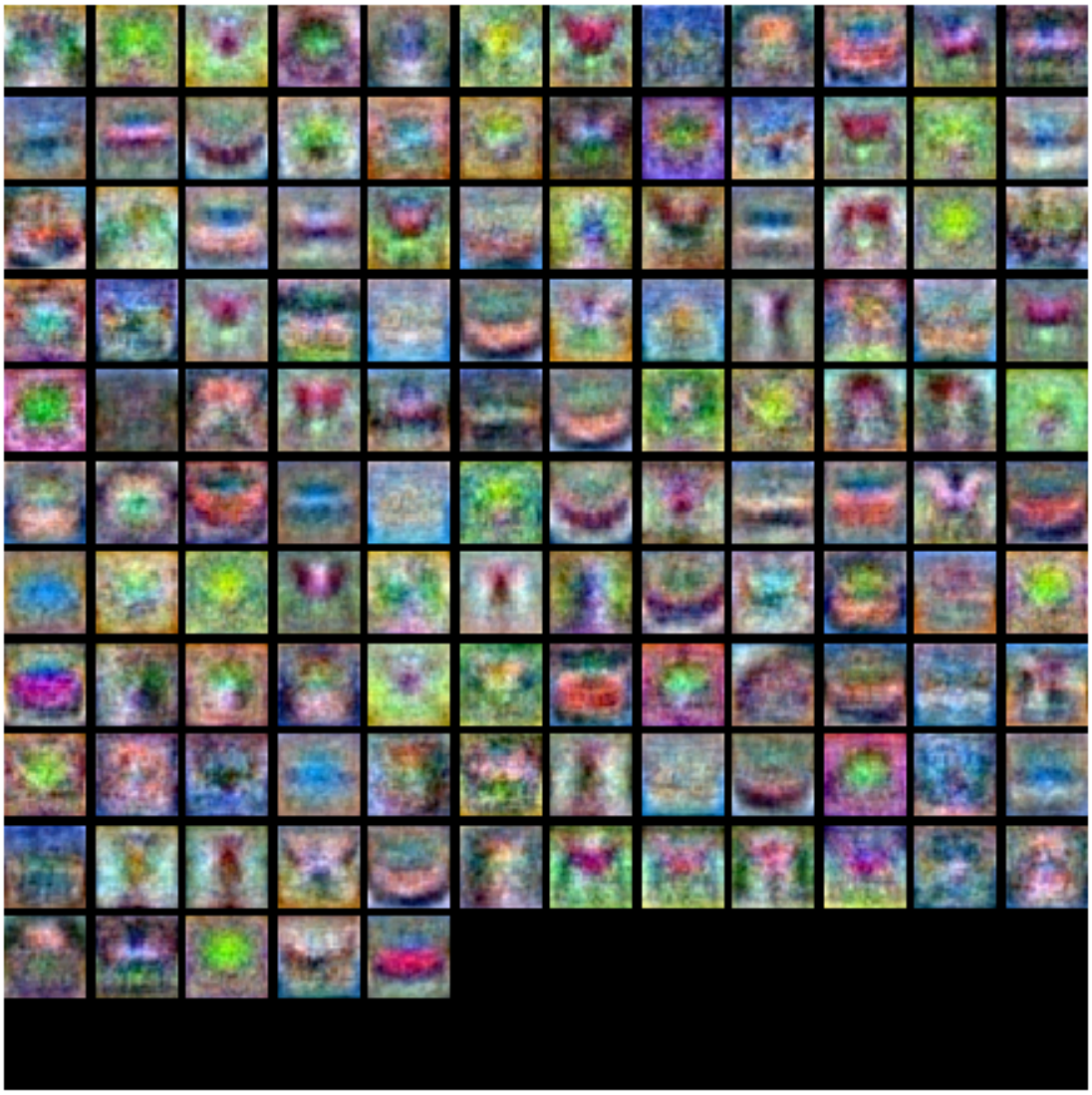
```
hidden_size: 100.000000, lr: 0.000700, lrd: 0.950000, reg: 0.750000, val_acc:
0.473000, train acc: 0.493163
hidden_size: 100.000000, lr: 0.000700, lrd: 0.950000, reg: 1.000000, val_acc:
0.464000, train acc: 0.482163
hidden_size: 100.000000, lr: 0.000700, lrd: 0.950000, reg: 1.100000, val_acc:
0.467000, train acc: 0.484980
hidden_size: 100.000000, lr: 0.000700, lrd: 0.950000, reg: 1.250000, val_acc:
0.474000, train acc: 0.477571
hidden_size: 100.000000, lr: 0.000800, lrd: 0.950000, reg: 0.750000, val_acc:
0.471000, train acc: 0.500143
hidden_size: 100.000000, lr: 0.000800, lrd: 0.950000, reg: 1.000000, val_acc:
0.477000, train acc: 0.486673
hidden_size: 100.000000, lr: 0.000800, lrd: 0.950000, reg: 1.100000, val_acc:
0.461000, train acc: 0.482122
hidden_size: 100.000000, lr: 0.000800, lrd: 0.950000, reg: 1.250000, val_acc:
0.468000, train acc: 0.477449
hidden_size: 100.000000, lr: 0.000900, lrd: 0.950000, reg: 0.750000, val_acc:
0.487000, train acc: 0.505020
hidden_size: 100.000000, lr: 0.000900, lrd: 0.950000, reg: 1.000000, val_acc:
0.477000, train acc: 0.492898
hidden_size: 100.000000, lr: 0.000900, lrd: 0.950000, reg: 1.100000, val_acc:
0.478000, train acc: 0.486776
hidden_size: 100.000000, lr: 0.000900, lrd: 0.950000, reg: 1.250000, val_acc:
0.463000, train acc: 0.476673
hidden_size: 100.000000, lr: 0.001000, lrd: 0.950000, reg: 0.750000, val_acc:
0.476000, train acc: 0.493184
hidden_size: 100.000000, lr: 0.001000, lrd: 0.950000, reg: 1.000000, val_acc:
0.466000, train acc: 0.490408
hidden_size: 100.000000, lr: 0.001000, lrd: 0.950000, reg: 1.100000, val_acc:
0.474000, train acc: 0.486633
hidden_size: 100.000000, lr: 0.001000, lrd: 0.950000, reg: 1.250000, val_acc:
0.468000, train acc: 0.480184
hidden_size: 100.000000, lr: 0.001100, lrd: 0.950000, reg: 0.750000, val_acc:
0.476000, train acc: 0.508612
hidden_size: 100.000000, lr: 0.001100, lrd: 0.950000, reg: 1.000000, val_acc:
0.475000, train acc: 0.492959
hidden_size: 100.000000, lr: 0.001100, lrd: 0.950000, reg: 1.100000, val_acc:
0.461000, train acc: 0.490592
hidden_size: 100.000000, lr: 0.001100, lrd: 0.950000, reg: 1.250000, val_acc:
0.458000, train acc: 0.487408
hidden_size: 110.000000, lr: 0.000700, lrd: 0.950000, reg: 0.750000, val_acc:
0.475000, train acc: 0.490224
hidden_size: 110.000000, lr: 0.000700, lrd: 0.950000, reg: 1.000000, val_acc:
0.470000, train acc: 0.480980
hidden_size: 110.000000, lr: 0.000700, lrd: 0.950000, reg: 1.100000, val_acc:
```

```
0.473000, train acc: 0.480061
hidden_size: 110.000000, lr: 0.000700, lrd: 0.950000, reg: 1.250000, val_acc:
0.463000, train acc: 0.470796
hidden_size: 110.000000, lr: 0.000800, lrd: 0.950000, reg: 0.750000, val_acc:
0.467000, train acc: 0.497776
hidden_size: 110.000000, lr: 0.000800, lrd: 0.950000, reg: 1.000000, val_acc:
0.480000, train acc: 0.488857
hidden_size: 110.000000, lr: 0.000800, lrd: 0.950000, reg: 1.100000, val_acc:
0.477000, train acc: 0.487592
hidden_size: 110.000000, lr: 0.000800, lrd: 0.950000, reg: 1.250000, val_acc:
0.474000, train acc: 0.481735
hidden_size: 110.000000, lr: 0.000900, lrd: 0.950000, reg: 0.750000, val_acc:
0.492000, train acc: 0.505959
hidden_size: 110.000000, lr: 0.000900, lrd: 0.950000, reg: 1.000000, val_acc:
0.465000, train acc: 0.493286
hidden_size: 110.000000, lr: 0.000900, lrd: 0.950000, reg: 1.100000, val_acc:
0.482000, train acc: 0.488857
hidden_size: 110.000000, lr: 0.000900, lrd: 0.950000, reg: 1.250000, val_acc:
0.448000, train acc: 0.466959
hidden_size: 110.000000, lr: 0.001000, lrd: 0.950000, reg: 0.750000, val_acc:
0.480000, train acc: 0.505041
hidden_size: 110.000000, lr: 0.001000, lrd: 0.950000, reg: 1.000000, val_acc:
0.484000, train acc: 0.495980
hidden_size: 110.000000, lr: 0.001000, lrd: 0.950000, reg: 1.100000, val_acc:
0.485000, train acc: 0.492367
hidden_size: 110.000000, lr: 0.001000, lrd: 0.950000, reg: 1.250000, val_acc:
0.468000, train acc: 0.487592
hidden_size: 110.000000, lr: 0.001100, lrd: 0.950000, reg: 0.750000, val_acc:
0.485000, train acc: 0.503224
hidden_size: 110.000000, lr: 0.001100, lrd: 0.950000, reg: 1.000000, val_acc:
0.465000, train acc: 0.482224
hidden_size: 110.000000, lr: 0.001100, lrd: 0.950000, reg: 1.100000, val_acc:
0.475000, train acc: 0.488327
hidden_size: 110.000000, lr: 0.001100, lrd: 0.950000, reg: 1.250000, val_acc:
0.467000, train acc: 0.487224
hidden_size: 125.000000, lr: 0.000700, lrd: 0.950000, reg: 0.750000, val_acc:
0.475000, train acc: 0.491673
hidden_size: 125.000000, lr: 0.000700, lrd: 0.950000, reg: 1.000000, val_acc:
0.474000, train acc: 0.482204
hidden_size: 125.000000, lr: 0.000700, lrd: 0.950000, reg: 1.100000, val_acc:
0.471000, train acc: 0.482429
hidden_size: 125.000000, lr: 0.000700, lrd: 0.950000, reg: 1.250000, val_acc:
0.461000, train acc: 0.476878
hidden_size: 125.000000, lr: 0.000800, lrd: 0.950000, reg: 0.750000, val_acc:
0.470000, train acc: 0.500143
hidden_size: 125.000000, lr: 0.000800, lrd: 0.950000, reg: 1.000000, val_acc:
0.477000, train acc: 0.489673
hidden_size: 125.000000, lr: 0.000800, lrd: 0.950000, reg: 1.100000, val_acc:
```

```
0.481000, train acc: 0.484714
hidden_size: 125.000000, lr: 0.000800, lrd: 0.950000, reg: 1.250000, val_acc:
0.466000, train acc: 0.480286
hidden_size: 125.000000, lr: 0.000900, lrd: 0.950000, reg: 0.750000, val_acc:
0.484000, train acc: 0.500184
hidden_size: 125.000000, lr: 0.000900, lrd: 0.950000, reg: 1.000000, val_acc:
0.467000, train acc: 0.494837
hidden_size: 125.000000, lr: 0.000900, lrd: 0.950000, reg: 1.100000, val_acc:
0.483000, train acc: 0.493673
hidden_size: 125.000000, lr: 0.000900, lrd: 0.950000, reg: 1.250000, val_acc:
0.473000, train acc: 0.481327
hidden_size: 125.000000, lr: 0.001000, lrd: 0.950000, reg: 0.750000, val_acc:
0.490000, train acc: 0.503204
hidden_size: 125.000000, lr: 0.001000, lrd: 0.950000, reg: 1.000000, val_acc:
0.468000, train acc: 0.491673
hidden_size: 125.000000, lr: 0.001000, lrd: 0.950000, reg: 1.100000, val_acc:
0.463000, train acc: 0.489857
hidden_size: 125.000000, lr: 0.001000, lrd: 0.950000, reg: 1.250000, val_acc:
0.472000, train acc: 0.486755
hidden_size: 125.000000, lr: 0.001100, lrd: 0.950000, reg: 0.750000, val_acc:
0.485000, train acc: 0.507429
hidden_size: 125.000000, lr: 0.001100, lrd: 0.950000, reg: 1.000000, val_acc:
0.498000, train acc: 0.503327
hidden_size: 125.000000, lr: 0.001100, lrd: 0.950000, reg: 1.100000, val_acc:
0.491000, train acc: 0.495959
hidden_size: 125.000000, lr: 0.001100, lrd: 0.950000, reg: 1.250000, val_acc:
0.473000, train acc: 0.481837
hidden_size: 150.000000, lr: 0.000700, lrd: 0.950000, reg: 0.750000, val_acc:
0.475000, train acc: 0.496918
hidden_size: 150.000000, lr: 0.000700, lrd: 0.950000, reg: 1.000000, val_acc:
0.468000, train acc: 0.485571
hidden_size: 150.000000, lr: 0.000700, lrd: 0.950000, reg: 1.100000, val_acc:
0.465000, train acc: 0.479388
hidden_size: 150.000000, lr: 0.000700, lrd: 0.950000, reg: 1.250000, val_acc:
0.454000, train acc: 0.476735
hidden_size: 150.000000, lr: 0.000800, lrd: 0.950000, reg: 0.750000, val_acc:
0.473000, train acc: 0.499265
hidden_size: 150.000000, lr: 0.000800, lrd: 0.950000, reg: 1.000000, val_acc:
0.474000, train acc: 0.488653
hidden_size: 150.000000, lr: 0.000800, lrd: 0.950000, reg: 1.100000, val_acc:
0.479000, train acc: 0.486041
hidden_size: 150.000000, lr: 0.000800, lrd: 0.950000, reg: 1.250000, val_acc:
0.466000, train acc: 0.484102
hidden_size: 150.000000, lr: 0.000900, lrd: 0.950000, reg: 0.750000, val_acc:
0.494000, train acc: 0.495551
hidden_size: 150.000000, lr: 0.000900, lrd: 0.950000, reg: 1.000000, val_acc:
0.479000, train acc: 0.490041
hidden_size: 150.000000, lr: 0.000900, lrd: 0.950000, reg: 1.100000, val_acc:
```

```
0.477000, train acc: 0.487082
hidden_size: 150.000000, lr: 0.000900, lrd: 0.950000, reg: 1.250000, val_acc:
0.455000, train acc: 0.483531
hidden_size: 150.000000, lr: 0.001000, lrd: 0.950000, reg: 0.750000, val_acc:
0.486000, train acc: 0.509184
hidden_size: 150.000000, lr: 0.001000, lrd: 0.950000, reg: 1.000000, val_acc:
0.490000, train acc: 0.499429
hidden_size: 150.000000, lr: 0.001000, lrd: 0.950000, reg: 1.100000, val_acc:
0.462000, train acc: 0.493571
hidden_size: 150.000000, lr: 0.001000, lrd: 0.950000, reg: 1.250000, val_acc:
0.474000, train acc: 0.484694
hidden_size: 150.000000, lr: 0.001100, lrd: 0.950000, reg: 0.750000, val_acc:
0.497000, train acc: 0.516102
hidden_size: 150.000000, lr: 0.001100, lrd: 0.950000, reg: 1.000000, val_acc:
0.479000, train acc: 0.498429
hidden_size: 150.000000, lr: 0.001100, lrd: 0.950000, reg: 1.100000, val_acc:
0.480000, train acc: 0.494653
hidden_size: 150.000000, lr: 0.001100, lrd: 0.950000, reg: 1.250000, val_acc:
0.464000, train acc: 0.485694
Best validation accuracy: 0.498000
```

[18]:
```
# visualize the weights of the best network
show_net_weights(best_net)
```



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%.

[19]:
```
test_acc = (best_net.predict(X_test) == y_test).mean()
print('Test accuracy: ', test_acc)
```

```
Test accuracy:  0.492
```

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: 1,3.

*Your explanation:The gap between low testing accuracy and higher training accuracy, can be caused by overfitting or the training data failing to capture the complexity of the phenomena, leading to underfitting. This problem can be solved by training on a larger dataset and increasing the regularization strength.

Training on a larger dataset can reduce the bias in the dataset, if any. So, it can work to generalize the model by introducing more variation in training.

Adding more hidden units makes the model more complicated, causing overfitting. As mentioned in hyperparameter tuning part, increasing H effectively increases training accuracy while decreasing the validation accuracy.

Increasing the regularization strength in the case of overfitting can solve this problem due to bias variance trade-off. Also, regularization pulls the weights down, which reduces the ReLU activation function's effect in training.