Berfin Kavşut
21602459
EEE 443
Oct 19, 2020

Assignment #1

Question 1

Maximum a posteriori probability estimate (MAP) algorithm is applied for finding the parameters satisfying maximum conditional probability of parameters under the given inputs.

y is possible inputs coming into our model, and θ is model parameters. Our aim is to find conditional probability distribution of θ under possible inputs, and then find the maximizing θ value for better model prediction.

$$\hat{\theta}_{MAP} = \arg\max_{\theta}\ P(\theta|y)$$

By Bayesian Rule, $P(\theta|y)$ can be written as $\frac{P(y|\theta)P(\theta)}{P(y)}$. Here, $P(\theta)$ is prior probability.

$$\hat{\theta}_{MAP} = \arg\max_{\theta}\ \frac{P(y|\theta)P(\theta)}{P(y)}$$

Since we will find the maximizing θ of $\frac{P(y|\theta)P(\theta)}{P(y)}$, we can get rid of $P(y)$. $P(y)$ is not dependent on θ.

$$= \arg\max_{\theta}\ P(y|\theta)P(\theta)$$

For simplification, we would like to express multiplication as addition and this is possible with logarithmic operation. Maximizing θ value will not change if we apply log operation.

$$\hat{\theta}_{MAP} = \arg\max_{\theta}\ \log\left(P(y|\theta)P(\theta)\right)$$

$$\hat{\theta}_{MAP} = \arg\max_{\theta}\ \log P(y|\theta) + logP(\theta) \qquad (1)$$

In Q1, the optimization problem is L2 Regularization of Least Squares Problem, a.k.a. Tikhonov Regularization.

$$\arg\min_{W} \sum_{n=1}^{N}\left(y^n - h(x^n, W)\right)^2 + \beta\sum_{i=1}^{m} w_i^2 \qquad (2)$$

The problem can also be thought in this fashion:

minimize (Loss(Data|Model) + complexity(Model)) [1]

In the L2 regularization problem, β is the regularization parameter for weights. If we have large regularization parameter, L2 norm of weight vector decreases. If we have small regularization parameter, then weights are affected less as if there was no regularization.

We will see the relation between (1) and (2).

$$\arg\min_{W} \sum_{n=1}^{N} \left(y^n - h(x^n, W)\right)^2 + \beta \sum_{i=1}^{m} w_i^2$$

Minimization problem turns into maximization problem.

$$= \arg\max_{W} -\sum_{n} \left(y^n - h(x^n, W)\right)^2 - \beta \sum_{i} w_i^2$$

We can multiply our equation with a positive constant and result will not change.

$$= \arg\max_{W} -\frac{1}{2\sigma^2} \sum_{n} \left(y^n - h(x^n, W)\right)^2 - \beta \frac{1}{2\sigma^2} \sum_{i} w_i^2$$

$$= \arg\max_{W} -\sum_{n} \frac{\left(y^n - h(x^n, W)\right)^2}{2\sigma^2} - \beta \sum_{i} \frac{w_i^2}{2\sigma^2}$$

$$= \arg\max_{W} \sum_{n} \log e^{-\frac{\left(y^n - h(x^n, W)\right)^2}{2\sigma^2}} + \sum_{i} \log e^{-\beta \frac{w_i^2}{2\sigma^2}}$$

$$= \arg\max_{W} \sum_{n} \log \left(\frac{1}{\sigma\sqrt{2\pi}}\right) e^{-\frac{\left(y^n - h(x^n, W)\right)^2}{2\sigma^2}}$$
$$+ \sum_{i} \log \left(\frac{\sqrt{\beta}}{\sigma\sqrt{2\pi}}\right) e^{-\beta \frac{w_i^2}{2\sigma^2}}$$

$$= \arg\max_{W} \log \prod_{n} \left(\frac{1}{\sigma\sqrt{2\pi}}\right) e^{-\frac{\left(y^n - h(x^n, W)\right)^2}{2\sigma^2}} + \log \prod_{i} \left(\frac{\sqrt{\beta}}{\sigma\sqrt{2\pi}}\right) e^{-\beta \frac{w_i^2}{2\sigma^2}}$$

Then, $f_{w1,w2,w3\ldots wm}(w_1, w_2, \ldots, w_n) = \prod_i \left(\frac{\sqrt{\beta}}{\sigma\sqrt{2\pi}}\right) e^{-\beta \frac{w_i^2}{2\sigma^2}}$ Weights are independent, therefore $f_{wi}(w_i) = \left(\frac{\sqrt{\beta}}{\sigma\sqrt{2\pi}}\right) e^{-\beta \frac{w_i^2}{2\sigma^2}}$ $(for\ i = 1,,,m)$, which is a Gaussian probability distribution with zero mean and standard deviation of $\frac{\sigma}{\sqrt{\beta}}$. When we select our loss function as Mean Squared Error (MSE) in our model and apply L2 regularization for weights, we start with the assumption that weights comes from Gaussian distribution.

Question 2

Part A

We know that XOR problem can be divided into AND and OR problems.

A XOR B = (A AND NOT B) OR (NOT A AND B)

So, we can divide XOR problem in the question as follows.

$$(X1+\overline{X2}) \oplus (\overline{X3}+\overline{X4}) = (X1+\overline{X2}) \cdot \overline{(\overline{X3}+\overline{X4})} + \overline{(X1+\overline{X2})} \cdot (\overline{X3}+\overline{X4})$$

$$= (X1+\overline{X2}) \cdot (X_3 \cdot X_4) + (\overline{X_1} \cdot X_2) \cdot (\overline{X3}+\overline{X4})$$

$$= X1 \cdot X_3 \cdot X_4 + \overline{X2} \cdot X_3 \cdot X_4 + \overline{X_1} \cdot X_2 \cdot \overline{X3} + \overline{X_1} \cdot X_2 \cdot \overline{X4}$$

We will have signum function as activation function. Weights will be selected as 1, and -1 for negated inputs. The reason is that inputs have equal importance, and therefore weights will have equal absolute values.

Signum Function: $f(v) = \begin{cases} +1 & v \geq 0 \\ -1 & v < 0 \end{cases}$

Vectors are represented with boldface.

1) *Hidden Layer*

Neuron #1: $X_1 \cdot X_3 \cdot X_4$

$$o_1 = f(v_1) = f(w_{11}x_1 + w_{12}x_2 + w_{13}x_3 + w_{14}x_4 - \theta_1)$$

$$w_{11} = 1,\ w_{12} = 0,\ w_{13} = 1,\ w_{14} = 1$$

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

$$w_{11}x_1 + w_{12}x_2 + w_{13}x_3 + w_{14}x_4 - \theta_1 \geq 0 \text{ for activation of the neuron}$$

$$w_{11}x_1 + w_{12}x_2 + w_{13}x_3 + w_{14}x_4 - \theta_1 < 0 \text{ for non} - \text{activation of the neuron}$$

Neuron should be fired only when $X_1 = 1,\ X_3 = 1,\ X_4 = 1.$

$$1.0 + 1.0 + 1.0 - \theta_1 < 0 \rightarrow \theta_1 > 0$$

$$1.0 + 1.0 + 1.1 - \theta_1 < 0 \rightarrow \theta_1 > 1$$

$$1.0 + 1.1 + 1.0 - \theta_1 < 0 \rightarrow \theta_1 > 1$$

$$1.0 + 1.1 + 1.1 - \theta_1 < 0 \rightarrow \theta_1 > 2$$

$$1.1 + 1.0 + 1.0 - \theta_1 < 0 \rightarrow \theta_1 > 1$$

$$1.1 + 1.0 + 1.1 - \theta_1 < 0 \rightarrow \theta_1 > 2$$

$$1.1 + 1.1 + 1.0 - \theta_1 < 0 \rightarrow \theta_1 > 2$$

$$1.1 + 1.1 + 1.1 - \theta_1 \geq 0 \rightarrow \theta_1 \leq 3$$

$$2 < \theta_1 \leq 3$$

Neuron #2: $\overline{X_2} \cdot X_3 \cdot X_4$

$$o_2 = f(v_2) = f(w_{21}x_1 + w_{22}x_2 + w_{23}x_3 + w_{24}x_4 - \theta_2)$$

$$w_{21} = 0,\ w_{22} = -1,\ w_{23} = 1,\ w_{24} = 1$$

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

$$w_{21}x_1 + w_{22}x_2 + w_{23}x_3 + w_{24}x_4 - \theta_2 \geq 0 \text{ for activation of the neuron}$$

$$w_{21}x_1 + w_{22}x_2 + w_{23}x_3 + w_{24}x_4 - \theta_2 < 0 \text{ for non} - \text{activation of the neuron}$$

Neuron should be fired only when $X_2 = 0,\ X_3 = 1,\ X_4 = 1$.

$$-1.0 + 1.0 + 1.0 - \theta_2 < 0 \rightarrow \theta_2 > 0$$

$$-1.0 + 1.0 + 1.1 - \theta_2 < 0 \rightarrow \theta_2 > 1$$

$$-1.0 + 1.1 + 1.0 - \theta_2 < 0 \rightarrow \theta_2 > 1$$

$$-1.0 + 1.1 + 1.1 - \theta_2 \geq 0 \rightarrow \theta_2 \leq 2$$

$$-1.1 + 1.0 + 1.0 - \theta_2 < 0 \rightarrow \theta_2 > -1$$

$$-1.1 + 1.0 + 1.1 - \theta_2 < 0 \rightarrow \theta_2 > 0$$

$$-1.1 + 1.1 + 1.0 - \theta_2 < 0 \rightarrow \theta_2 > 0$$

$$-1.1 + 1.1 + 1.1 - \theta_2 < 0 \rightarrow \theta_2 > 1$$

$$1 < \theta_2 \leq 2$$

Neuron #3: $\overline{X_1} \cdot X_2 \cdot \overline{X_3}$

$$o_3 = f(v_3) = f(w_{31}x_1 + w_{32}x_2 + w_{33}x_3 + w_{34}x_4 - \theta_3)$$

$$w_{31} = -1,\ w_{32} = 1,\ w_{33} = -1,\ w_{34} = 0$$

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

$$w_{31}x_1 + w_{32}x_2 + w_{33}x_3 + w_{34}x_4 - \theta_3 \geq 0 \text{ for activation of the neuron}$$

$$w_{31}x_1 + w_{32}x_2 + w_{33}x_3 + w_{34}x_4 - \theta_3 < 0 \text{ for non} - \text{activation of the neuron}$$

Neuron should be fired only when $X_1 = 0,\ X_2 = 1,\ X_3 = 0$.

$$-1.0 + 1.0 - 1.0 - \theta_3 < 0 \rightarrow \theta_3 > 0$$

$$-1.0 + 1.0 - 1.1 - \theta_3 < 0 \rightarrow \theta_3 > -1$$

$$-1.0 + 1.1 - 1.0 - \theta_3 \geq 0 \rightarrow \theta_3 \leq 1$$

$$-1.0 + 1.1 - 1.1 - \theta_3 < 0 \rightarrow \theta_3 > 0$$

$$-1.1 + 1.0 - 1.0 - \theta_3 < 0 \rightarrow \theta_3 > -1$$

$$-1.1 + 1.0 - 1.1 - \theta_3 < 0 \rightarrow \theta_3 > -2$$

$$-1.1 + 1.1 - 1.0 - \theta_3 < 0 \rightarrow \theta_3 > 0$$

$$-1.1 + 1.1 - 1.1 - \theta_3 < 0 \rightarrow \theta_3 > -1$$

$$0 < \theta_3 \leq 1$$

Neuron #4: $\overline{X_1} \cdot X_2 \cdot \overline{X4}$

$$o_4 = f(v_4) = f(w_{41}x_1 + w_{42}x_2 + w_{43}x_3 + w_{44}x_4 - \theta_4)$$

$$w_{41} = -1,\ w_{42} = 1,\ w_{43} = 0,\ w_{44} = -1$$

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

$$w_{41}x_1 + w_{42}x_2 + w_{43}x_3 + w_{44}x_4 - \theta_4 \geq 0 \text{ for activation of the neuron}$$

$$w_{41}x_1 + w_{42}x_2 + w_{43}x_3 + w_{44}x_4 - \theta_4 < 0 \text{ for non} - \text{activation of the neuron}$$

Neuron should be fired only when $X_1 = 0,\ X_2 = 1,\ X_4 = 0.$

$$-1.0 + 1.0 - 1.0 - \theta_4 < 0 \rightarrow \theta_4 > 0$$

$$-1.0 + 1.0 - 1.1 - \theta_4 < 0 \rightarrow \theta_4 > -1$$

$$-1.0 + 1.1 - 1.0 - \theta_4 \geq 0 \rightarrow \theta_4 \leq 1$$

$$-1.0 + 1.1 - 1.1 - \theta_4 < 0 \rightarrow \theta_4 > 0$$

$$-1.1 + 1.0 - 1.0 - \theta_4 < 0 \rightarrow \theta_4 > -1$$

$$-1.1 + 1.0 - 1.1 - \theta_4 < 0 \rightarrow \theta_4 > -2$$

$$-1.1 + 1.1 - 1.0 - \theta_4 < 0 \rightarrow \theta_4 > 0$$

$$-1.1 + 1.1 - 1.1 - \theta_4 < 0 \rightarrow \theta_4 > -1$$

$$0 < \theta_4 \leq 1$$

Bias vector of hidden layer: $\boldsymbol{\theta_1} = [\theta_1\ \theta_2\ \theta_3\ \theta_4]^T$

Weight matrix of hidden layer: $W_1 = \left[\boldsymbol{w_1^T}\ \boldsymbol{w_2^T}\ \boldsymbol{w_3^T}\ \boldsymbol{w_4^T}\right]$

Extended weight matrix: $W_{1e} = [W|\,\theta]$

Extended input vector: $\boldsymbol{x_{1e}} = [x_1\ x_2\ x_3\ x_4 - 1]^T$

$$\boldsymbol{v_1} = W_{1e}\boldsymbol{x_{1e}},\quad \boldsymbol{o_1} = \Gamma(v_1) = \Gamma(W_{1e}x_{1e})$$

2) *Output Layer*
   Output Neuron: $o_1 + o_2 + o_3 + o_4$

$$o_5 = f(v_5) = f(w_{51}o_1 + w_{52}o_2 + w_{53}o_3 + w_{54}o_4 - \theta_5)$$

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

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

$$w_{51}x_1 + w_{52}x_2 + w_{53}x_3 + w_{54}x_4 - \theta_5 \geq 0 \text{ for activation of the neuron}$$

$$w_{51}x_1 + w_{52}x_2 + w_{53}x_3 + w_{54}x_4 - \theta_4 < 0 \text{ for non} - \text{activation of the neuron}$$

Neuron should be non-activated only when $o_1 = 0,\ o_2 = 0,\ \ o_3 = 0,\ \ o_4 = 0$.

$$1.0 + 1.0 + 1.0 + 1.0 - \theta_5 < 0$$

$$1.0 + 1.0 + 1.0 + 1.1 - \theta_5 \geq 0$$

$$1.0 + 1.0 + 1.1 + 1.0 - \theta_5 \geq 0$$

$$1.0 + 1.0 + 1.1 + 1.1 - \theta_5 \geq 0$$

$$1.0 + 1.1 + 1.0 + 1.0 - \theta_5 \geq 0$$

$$1.0 + 1.1 + 1.0 + 1.1 - \theta_5 \geq 0$$

$$1.0 + 1.1 + 1.1 + 1.0 - \theta_5 \geq 0$$

$$1.0 + 1.1 + 1.1 + 1.1 - \theta_5 \geq 0$$

$$1.1 + 1.0 + 1.0 + 1.0 - \theta_5 \geq 0$$

$$1.1 + 1.0 + 1.0 + 1.1 - \theta_5 \geq 0$$

$$1.1 + 1.0 + 1.1 + 1.0 - \theta_5 \geq 0$$

$$1.1 + 1.0 + 1.1 + 1.1 - \theta_5 \geq 0$$

$$1.1 + 1.1 + 1.0 + 1.0 - \theta_5 \geq 0$$

$$1.1 + 1.1 + 1.0 + 1.1 - \theta_5 \geq 0$$

$$1.1 + 1.1 + 1.1 + 1.0 - \theta_5 \geq 0$$

$$1.1 + 1.1 + 1.1 + 1.1 - \theta_5 \geq 0$$

$$0 < \theta_5 \leq 1$$

Extended weight matrix of output layer: $W_{2e} = \left[\boldsymbol{w_5^T} \middle| \theta_5\right]$

Extended input vector of output layer(output vector of hidden layer):$\boldsymbol{x_{2e}} = [o_1\ o_2\ o_3\ o_4 - 1]^T$

$$v_5 = W_{2e}\boldsymbol{x_{2e}},\quad \boldsymbol{o_5} = \Gamma(v_5) = \Gamma(W_{2e}x_{2e}) = \Gamma\big(W_{2e}\,\Gamma(W_{1e}x_{1e})\big)$$

Bias terms are selected from inequalities for part a. Any bias term satisfying inequalities work with full accuracy with ideal inputs.

Selected biases: $\theta_1 = 2.1, \theta_2 = 1.1, \theta_3 = 0.1, \theta_4 = 0.1, \theta_5 = 0.1$.

Part B

The whole network works with full accuracy. Neuron #1 is shown to work with full accuracy separately inside MATLAB code.

Output for Part B:

Neural Network works with 100% accuracy!

Neuron #1 works with 100% accuracy!

Part C

Neural network will not work with high accuracy under noisy inputs because biases are selected near to end points. We can improve our model by selecting biases in the middle, so that decision boundaries will be further away from possible noisy inputs. Thereis no need to change weights, because inputs have equal importance, as we have said before.

New selected biases: $\theta_1 = 2.5, \theta_2 = 1.5, \theta_3 = 0.5, \theta_4 = 0.5, \theta_5 = 0.5$.

Part D

Neural network in Part C works with higher accuracy compared to neural network in Part A. Still, it is mostly varying between 88% - 92% accuracy and it is not possible to reach to 100% accuracy level.

Question 3

Part A

In Figure 1, one can see some letters are looking very similar, e.g. Class #1 and Class #17. Even before checking correlation matrix, we can think that it is hard for hand-written alphabet letter classification to work with high accuracy as human eye cannot distinguish classes correctly.

In Figure 2, correlation matrix is shown as an image. Scaling is from black to white as values increase. It can be seen that mostly diagonal entries are high which means there is high correlation or i.e. less within-class variability. And mostly non-diagonal entries are darker due to low correlation or i.e. higher across-class variability. However, this is not always the case. There are some low within-class correlations, and very high across-class correlations. Non-diagonal entries should be lower to be classified well, and diagonal entries should be higher than non-diagonal entries. Still, diagonal entries should not be very high values because we want diverse training data so that our model will have a generalized solution. As a result, I do not expect very high accuracy from our model due to dataset variability.

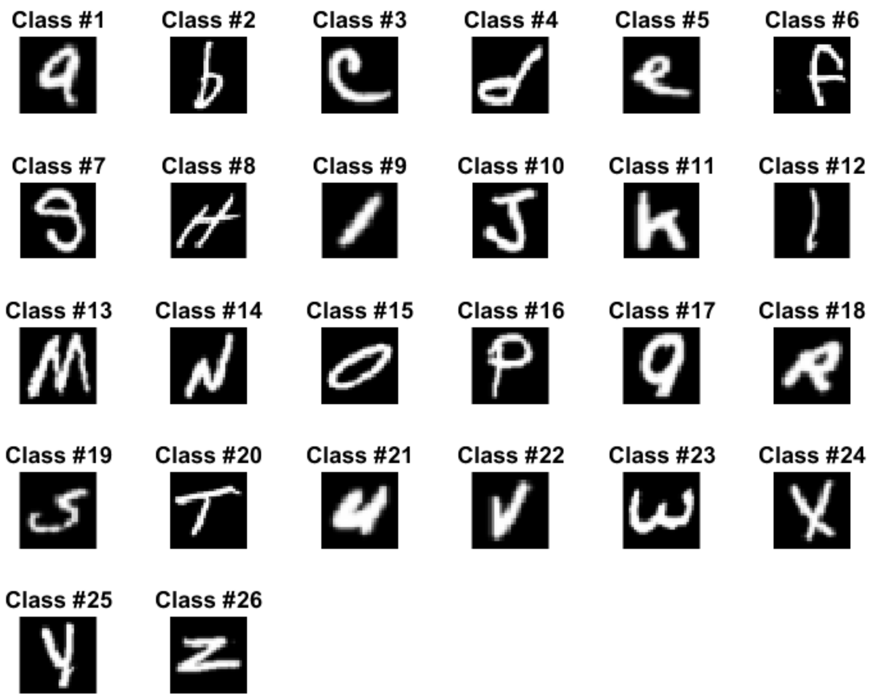


Figure 1. Randomly Sampled Training Images

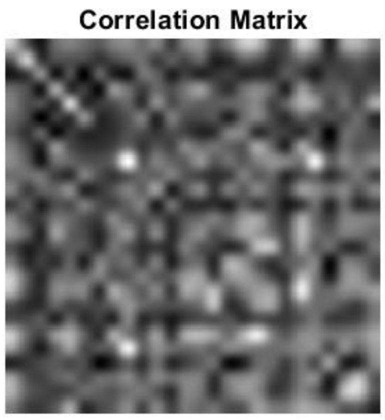


Figure 2. Correlation Matrix

Part B

I chose lambda to be 1, and optimum learning rate to be 0.2. I checked last MSE values for tuning learning rate. Also, I checked the weight images as shown in Figure 3. In Figure 3, it is seen that most of weight images are close to their class letter and good representative of their classes.

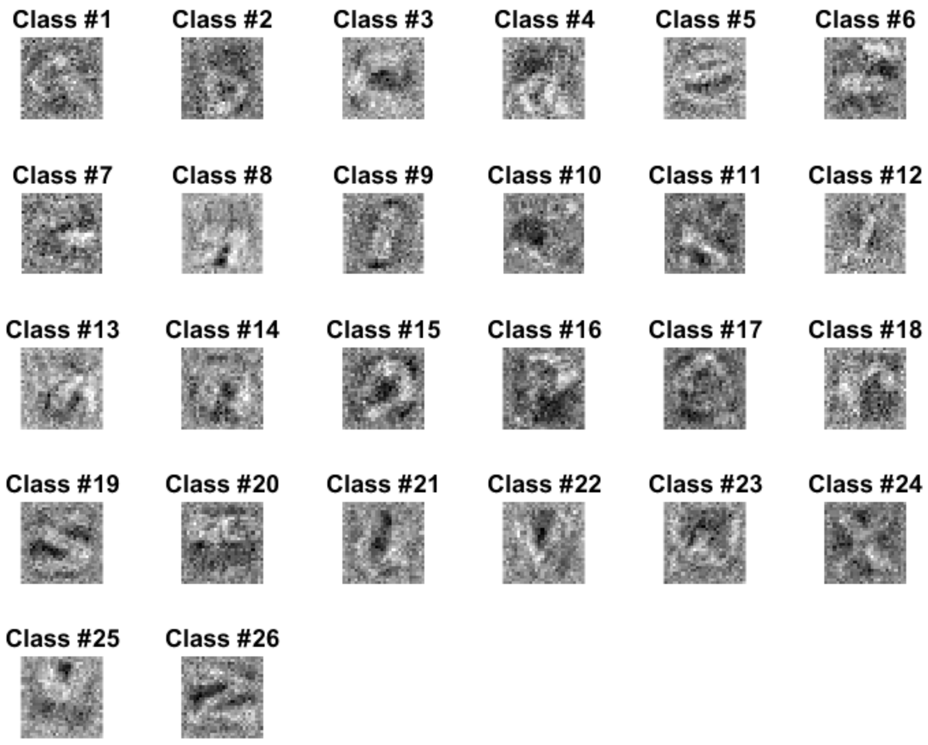


Figure 3. Final Network Weights for Optimum Learning Rate

Part C

Low learning rate is 0.01 multiplied by optimum learning rate, and high learning rate is 100 times of optimum learning rate. MSE curves are displayed in Figure 4. When learning rate is low, MSE drops slower and last reached MSE value is higher than optimum result. We think that learning process is not over yet and there is underfitting. When learning rate is high, minimum point of loss function which is a result of optimum weights are missed and oscillates are higher than optimum case even though MSE dropped faster. Last MSE value is again higher than optimum case.

<u>Results</u>

Learning Rate:0.002

Last Reached MSE Value:0.038448

Learning Rate:0.2

Last Reached MSE Value:0.0097335

Learning Rate:20

Last Reached MSE Value:0.076923

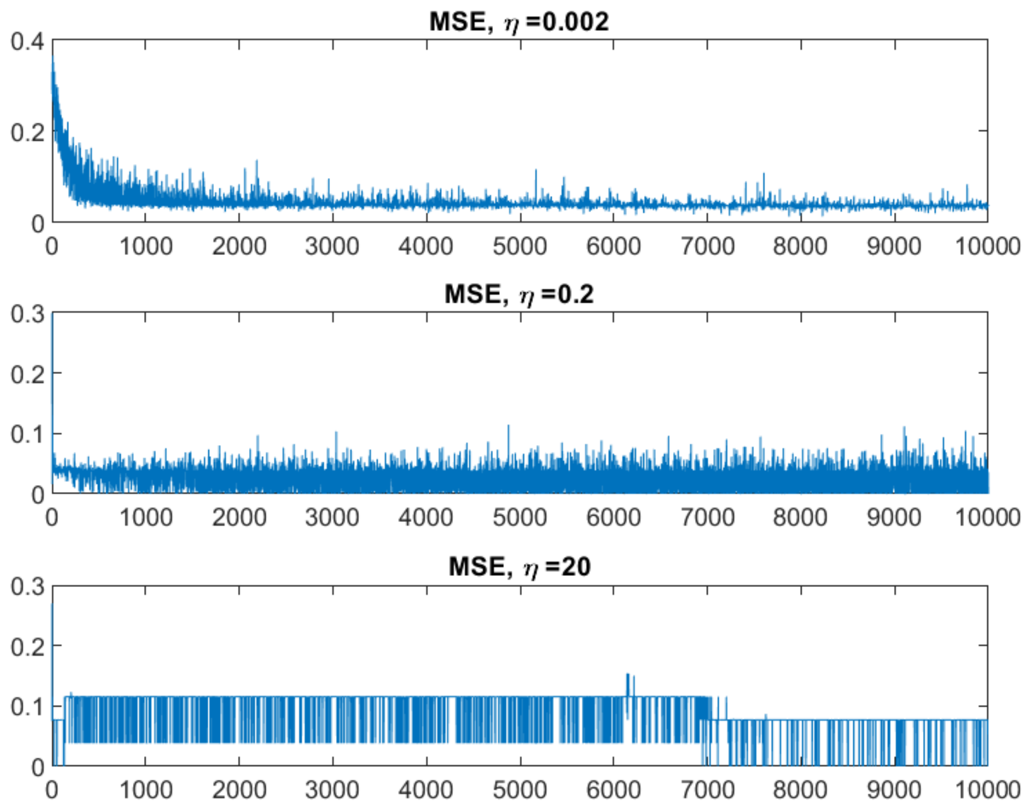


Figure 4. MSE Curves for Low, Optimum, and High Learning Rates

Part D

As it was expected, best test accuracy is coming from the optimum learning rate. Accuracy for low learning rate is less than optimum case, because learning process is not finished yet. High learning rate is the worst case, since there is almost no way for finding minimum point of loss function due to oscillation.

<u>Accuracy percentages with different learning rates</u>

Learning Rate:0.002

Accuracy Percentage:22.9231%

Learning Rate:0.2

Accuracy Percentage:57%

Learning Rate:20

Accuracy Percentage:3.8462%

Question 4

Run code is attached at the end of the report after MATLAB code.

two_layer_net code is studied in order to understand the effect of hyperparameters in a neural network, how to see model could not learn weights and biases, importance of validation and training accuracy, and loss function checks simultaneously, and learn how to debug training process of neural networks in short.

In the 6th kernel, training loss history of toy model is displayed. It is important to see convergence in loss function decay, since it means our model was successful to minimize loss function. If there is no convergence and loss function seems to continue decreasing, then there might be underfitting, which means our model could not learn weights and biases yet. Still, we cannot detect overfitting by looking at loss function since loss function is calculated for training data. We need to check validation and training accuracies simultaneously to see if there is any gap between them. If they are in the same trend, we have the idea that our model is still learning and yields generalization, not overfitting. If training accuracy is high but validation accuracy started to decrease, then there is overfitting.

In the 7th kernel, model is trained for CIFAR10 dataset. Additional to Q3, we have regularization parameter here. Regularization problem is the same with Q1. Loss function and L2 norm of weights are minimized together. Regularization of weights means that there is a suppressive effect on weights if weights are trying to fit to noise, and therefore regularization is applied for generating more general models rather than overfitting models.

In the part "Tune your hypermeters", we look at weight images. Weight images seem very close to each other, which makes us think that our model could not learn classification problem yet. Hidden layer size, learning rate, regularization rate and number of epochs can be changed for better performance.

Number of epochs are increased, so weights will be processed more with training data, which is good for learning. If number of epochs is too high, there might be overfitting though. Optimum number should be captured. Learning rate and regularization strength is increased. Learning rate determines how much we will walk with gradient to reach minimum point of loss function. If it is too low, we cannot reach minimum point and we need more and more epochs. If it is too high, found optimum point, or loss function value, starts to oscillate like it did in Q3. Regularization strength is also increased for a better generalization. The improvement in model can seen on increase in validation accuracy and weight images becoming better representatives of their own classes.

Answer for Inline Question: 1,3

My explanation: When I changed number of hidden units, I observed that training accuracy and test accuracy drop together or increase together, but there is no increasing gap between them. I also checked with different iteration numbers. Even in extreme case like having 5 hidden units, 5000 iterations, there is no gap between test accuracy and training accuracy. Overfitting can happen because of having too many layers or hidden nodes, but I do not think having more hidden units affects the gap between training accuracy and test accuracy in our example due to having only one hidden layer and output layer.

When there is a gap between training accuracy and test accuracy, I would suspect that our model could not generalize enough and we should give more diverse data to train it for generalization. For example, if there is only 10 images, it is very hard for our model to generalize and predict a lot altered images in same classes.

Similarly, when regularization strength is too low, our model will fit to noise easily and generalization will be avoided. This will result in a gap between training accuracy and test accuracy. But, if it was higher than it should be, training accuracy and test accuracy would fall together.

References

[1] Machine Learning Crash Course, ‘Regularization for Simplicity: L_2 Regularization'. [Online]. Available: https://developers.google.com/machine-learning/crash-course/regularization-for-simplicity/l2-regularization. [Accessed: 10- Oct- 2020].

Appendix

```
function berfin_kavsut_21602459_hw1(question)
clc
close all

switch question
    case '1'
    disp('1')
    disp('Answer is on the report.')

    case '2'
    disp('2')
    disp('Neural Network with Logic Gates')

    disp('Part A')
    disp('Answer is on the report.')

    disp('Part B')
    disp('Bias terms and extended weight matrix of hidden 
layer:')
    %set weights for each of four hidden units
    %weights and biases are taken from part a
    w1 = [1 0 1 1]; b1 = 2.1
    w2 = [0 -1 1 1]; b2 = 1.1
    w3 = [-1 1 -1 0]; b3 = 0.1
    w4 = [-1 1 0 -1]; b4 = 0.1

    %single hidden layer for AND implementations
    %weight matrix with bias terms
    W1e = [[w1 b1];[w2 b2];[w3 b3];[w4 b4]] %4x5

    disp('Bias term and extended weight vector of output 
layer:')

    %output layer(neuron) for OR implementation
    %weight vector with bias term
    w5 = [1 1 1 1]; b5 = 0.2
    W2e = [w5 b5]

    %all possible inputs with -1 input (bias input)
    input_no = 16;
    X1 =[0 0 0 0;
         0 0 0 1;
         0 0 1 0;
         0 0 1 1;
         0 1 0 0;
         0 1 0 1;
         0 1 1 0;
         0 1 1 1;
         1 0 0 0;
```

```
          1  0  0  1;
          1  0  1  0;
          1  0  1  1;
          1  1  0  0;
          1  1  0  1;
          1  1  1  0;
          1  1  1  1]';
    B = -1*ones(1,input_no);
    X1e = [X1;B]; %5x16

    %output vector of first hidden layer
    V1 = W1e*X1e; %4x16
    O1 = step_activation(V1,0); %4x16

    %output of neural network, output from output neuron
    V2 = W2e*[O1;-1*ones(1,input_no)];
    O2 = step_activation(V2,0);
    o = O2;

    %desired output
    d = logic_gates(X1e);

    accuracy = sum(d==o)/input_no*100;
    %disp(strcat('Accuracy percentage:',num2str(accuracy),'%
'))

    if(accuracy == 100)
        disp('Neural Network works with 100% accuracy!')
    else
        disp('Neural Network does not work with full
accuracy!')
    end

    w1e = [w1 b1];
    v1 = w1e*X1e;
    o1 = step_activation(v1,0);

    %desired output of neuron #1
    d = logic_gates_neuron(X1e);

    accuracy = sum(d==o1)/input_no*100;
    %disp(strcat('Accuracy percentage:',num2str(accuracy),'%
'))

    if(accuracy == 100)
        disp('Neuron #1 works with 100% accuracy!')
    else
        disp('Neuron #1 does not work with full accuracy!')
    end

    disp('Part C')
    disp('Bias terms are revised for robustness!');
    %set weights for each of four hidden units
    %weights and biases are chosen for part c
    %biases are not selected at the edge of inequalities like
part a,
    %they are in the middle of their min and max values for
robustness

    %single hidden layer for AND implementations
    %weight matrix with bias terms
    b1_new = 2.5
    b2_new = 1.5
    b3_new = 0.5
    b4_new = 0.5
    W1e_new = [[w1 b1_new];[w2 b2_new];[w3 b3_new];[w4
b4_new]]

    %output layer(neuron) for OR implementation
    %weight vector with bias term
    b5_new = 0.5
    W2e_new = [w5 b5_new]

    disp('Part D')

    %create 25 replicas for each input vector
    X1e_old = X1e;
    X1e=zeros(5,400);
    for i=1:16
        %25 replicas of each input vector
        X1e(:,(i-1)*25+1:(i*25)) = repmat(X1e_old(:,i),1,25);
    end

    N1e = 0 + 0.2*randn(4,400); %mean=0, std=0.2
    N1e = [N1e; zeros(1,400)]; %5x400

    %linear combination of random input and noise
    X1e_noise = X1e + N1e;

    %part a
    V1 = W1e*X1e_noise;
    O1 = step_activation(V1,0);
    V2 = W2e*[O1;-1*ones(1,400)];
    O2 = step_activation(V2,0);
    O = O2; %1x400

    %part c
    V1 = W1e_new*X1e_noise;
    O1 = step_activation(V1,0);
    V2 = W2e_new*[O1;-1*ones(1,400)];
    O2 = step_activation(V2,0);
    O_new = O2; %1x400
    %desired output
    V1 = W1e*X1e;
    O1 = step_activation(V1,0);
    V2 = W2e*[O1;-1*ones(1,400)];
    O2 = step_activation(V2,0);
    D = O2; %1x400

    disp(strcat('Accuracy of neural network in part a:
',num2str(sum(O==D)/400*100),'%'));
    disp(strcat('Accuracy of neural network in part c:
',num2str(sum(O_new==D)/400*100),'%'));


    case '3'
    disp('3')
    disp('Perceptron of Alphabet Letters')

    disp('Part A')

    %read dataset
    dataset = h5readData();
    train_images = im2double(dataset.trainims);
    test_images = im2double(dataset.testims);
    test_labels = dataset.testlbls;
    train_labels =  dataset.trainlbls;

    %take size of images, training image no, test image no
    [m,l,train_no] = size(train_images);
    [~,~,test_no] = size(test_images);
    class_no = 26; %class_no = unique(train_labels);

    %visualize a sample image for each classs
    sample_train = zeros(28,28,26);
    sample_train2 = zeros(28,28,26); %to be used in
correlation matrix calculation
    sample_test = zeros(28,28,26);

    disp('Sample images from each class are displayed in
figure.')
    figure;
    for i = 1:class_no

        %take indices vector of ith class images
        index = find(i == train_labels);

        %take one random index from ith class, take the train
image
        rand_ind = floor((length(index)-1)*rand()+1);
        sample_train(:,:,i) =
train_images(:,:,index(rand_ind));
```

```
        %take another random index from ith class, take the
train image
        rand_ind = floor((length(index)-1)*rand()+1);
        sample_train2(:,:,i) =
train_images(:,:,index(rand_ind));

        %take one random index from ith class, take the test
image
        index = find(i == test_labels);
        rand_ind = floor((length(index)-1)*rand()+1);
        sample_test(:,:,i) = test_images(:,:,index(rand_ind));

        %display sample images
        subplot(5,6,i);
        imshow(squeeze(sample_train(:,:,i)),[]);
        title(strcat('Class #',num2str(i)));

    end
    saveas(gcf,'Q3_Sampled_Images.png');

    %correlation matrix
    %size of (class no x class no)
    %diagonal entries are for within-class correlation
coefficients
    %non-diagnoal entries are for across-class correlation
coefficients
    %p = cor[X,Y] = cov[X,Y]/sqrt((var[X]*var[Y]))
    cor_matrix = zeros(26,26);
    for i=1:class_no
        for j = 1:class_no

            %take one sample images
            X = squeeze(sample_train(:,:,i));
            Y = squeeze(sample_train(:,:,j));%for across-class

            if (i == j)
                Y =  squeeze(sample_train2(:,:,j)); %for
within-class
            end

            %find correlation coefficients for X and Y,
            %which are turned into column vectors
            R = corrcoef(X,Y);
            cor_matrix(i,j) = R(1,2); %take non-diagonal entry

        end
    end

    disp('Correlation Matrix:')
    cor_matrix
    disp('Correlation Matrix is also displayed as an 26x26
image in figure.')
    corr_flag = 1;
    if(corr_flag)
        figure;
        %imshow(cor_matrix,[]);
        imshow(imresize(cor_matrix,[260,260]),[]);
        title('Correlation Matrix');
        saveas(gcf,'Q3_Correlation_Matrix_resize.png');
%
%          figure;
%          imshow(cor_matrix,[]);
%          title('Correlation Matrix');
%          saveas(gcf,'Q3_Correlation_Matrix.png');
    end

    %we have neurons as much as class number
    neuron_no = class_no;
    input_size = m*l; %length of vectorized images

    lambda = 1; %sigmoid function constant
    n_opt = 0.2; %learning rate

    %take learning rates a lot smaller and a lot bigger than
optimum learning rate
    n = [0.01 1 100]*n_opt;

    for k=1:3

        %random weights and bias terms from gaussian
distribution
        %with zero mean, 0.01 variance
        mean = 0; std = sqrt(0.01);
        W = mean + std*randn(neuron_no,input_size);
        b = mean + std*randn(neuron_no,1);
        We = [W b]; %extended weight matrix

        %activation function is sigmoid function
        %sigmoid_activation(v,lambda,T)

        %start iteration for training model
        MSE = 0;
        iter_no = 10000;

        for i =1:iter_no

            %take random train image
            rand_ind = floor((train_no-1)*rand()+1);
            x = train_images(:,:,rand_ind);
            x = reshape(x,input_size,1);
            x = x./(max(x(:))); %rescale image
            xe = [x;-1]; %vectorize image

            %linear activation potential
            v = We*xe;
            %it is followed by activation function
            o = sigmoid_activation(v,lambda,0);
            d =  zeros(class_no,1);
            d(train_labels(rand_ind))= 1;

            %gradient descent update
            delta_W = n(k)*lambda*(d-o).*(o.*(1-o))*xe';
            We = We + delta_W;

            MSE = (1/class_no)*sum((d-o).^2);
            %objective = (1/2)*norm(real_output-output,2);
            history(k).MSE(i) = MSE;
            history(k).W(:,:,i) = We(:,1:end-1);
            history(k).b(:,i) = We(:,end);
            %history.objective_function(i) = objective;

        end

    end

%     for k =1:3
%         figure;
%         for i = 1:class_no
%             subplot(5,6,i);
%             imshow(reshape(history(k).W(i,:,end),m,l),[]);
%             title(strcat('Class #',num2str(i)));
%         end
%         saveas(gcf,strcat('Q3_Trained
Weights_',num2str(n(k)),'.png'));
%     end

    disp('Part B')
    disp('Weights for optimum learning rate are displayed in
figure.')
    disp('Optimum learning rate is 0.2.')

    %for optimum learning rate, display weights
    figure;
    for i = 1:class_no
        subplot(5,6,i);
        imshow(reshape(history(2).W(i,:,end),m,l),[]);
        title(strcat('Class #',num2str(i)));
    end
    saveas(gcf,strcat('Q3_Trained Weights_last_0.2.png'));

    disp('Part C')
    disp('MSE curves are displayed in figure.')
    figure;
    for k=1:3
        subplot(3,1,k);
        plot(1:iter_no,history(k).MSE);
        title(strcat('MSE, {\eta} = ',num2str(n(k))));

        disp(strcat('Learning Rate: ',num2str(n(k))));
        disp(strcat('Last Reached MSE Value:',
num2str(history(k).MSE(end))));

    end
    saveas(gcf,strcat('Q3_MSE_learning rate_0.2.png'));

    disp('Part D')
    disp('Accuracy percentages with different learning
rates:')

    for k=1:3
        We = [history(k).W(:,:,end) history(k).b(:,end)];
        X = reshape(test_images,input_size,test_no);
        X = X./max(X); %rescale image
        Xe = [X;-1*ones(1,test_no)];
        V = We*Xe;
        O = sigmoid_activation(V,lambda,0);
        [~,output] = max(O);
        D = test_labels';
        accuracy = sum(output==D)/test_no*100;

        disp(strcat('Learning Rate: ',num2str(n(k))));
        disp(strcat('Accuracy Percentage: ',
num2str(accuracy),'%'));
    end

    case '4'
    disp('4')
    disp('Answer is on the report.')

end

end

function output = logic_gates(X)
    %implmeentation of logic gates
    %output = (X1 OR NOT X2) XOR (NOT X3 OR NOT X4,
    %which is quivalent to ((X1 OR NOT X2) AND NOT (NOT X3 OR
NOT X4)) ...
    %OR (NOT (X1 OR NOT X2) AND (NOT X3 OR NOT X4))

    %input
    %X: each column vector is one input vector,
    %contains concatenated input vector

    %implement logic gates for each column vector inside for-
loop
    [row,col] = size(X);
    output = zeros(1,col);
    for i=1:col
        x1 = X(1,i);
        x2 = X(2,i);
        x3 = X(3,i);
        x4 = X(4,i);
        output(i) = ((x1||~x2)&~(~x3||~x4)) ||
(~(x1||~x2)&(~x3||~x4));
    end

end
function output = logic_gates_neuron(X)
    %implmeentation of logic gates
    %output = (X1 OR NOT X2) XOR (NOT X3 OR NOT X4,
    %which is quivalent to ((X1 OR NOT X2) AND NOT (NOT X3 OR
NOT X4)) ...
    %OR (NOT (X1 OR NOT X2) AND (NOT X3 OR NOT X4))

    %input
    %X: each column vector is one input vector,
    %contains concatenated input vector

    %implement logic gates for each column vector inside for-
loop
    [row,col] = size(X);
    output = zeros(1,col);
    for i=1:col
        x1 = X(1,i);
        x3 = X(3,i);
        x4 = X(4,i);
        output(i) = (x1&x3&x4);
    end

end

function result = step_activation(v,T)
    %unipolar step activation function
    %inputs
    %x: input, T: threshold
    result = (v >= T);
end

function result = sigmoid_activation(v,lambda,T)
    %unipolar sigmoid activation function
    %results are between 0-1
    %inputs
    %v: inputs, lambda: parameter for sigmoid, T: threshold
    if(lambda > 0)
        result = 1./(1+exp(-lambda*(v-T)));
    else
        result = nan;
        disp('Lambda should be a positive value!');
    end

end

function dataset = h5readData()
    %read dataset from hdf5 file

    %h5disp(filename);
    %h5info(filename);
    %h5read(filename);
    dataset.testims = h5read('assign1_data1.h5','/testims');
    dataset.testlbls = h5read('assign1_data1.h5','/testlbls');
    dataset.trainims = h5read('assign1_data1.h5','/trainims');
    dataset.trainlbls =
h5read('assign1_data1.h5','/trainlbls');
end
```

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:
3.6802720745909845e-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 [5]:

```
from cs231n.gradient_check import eval_numerical_gradient
```

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

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

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

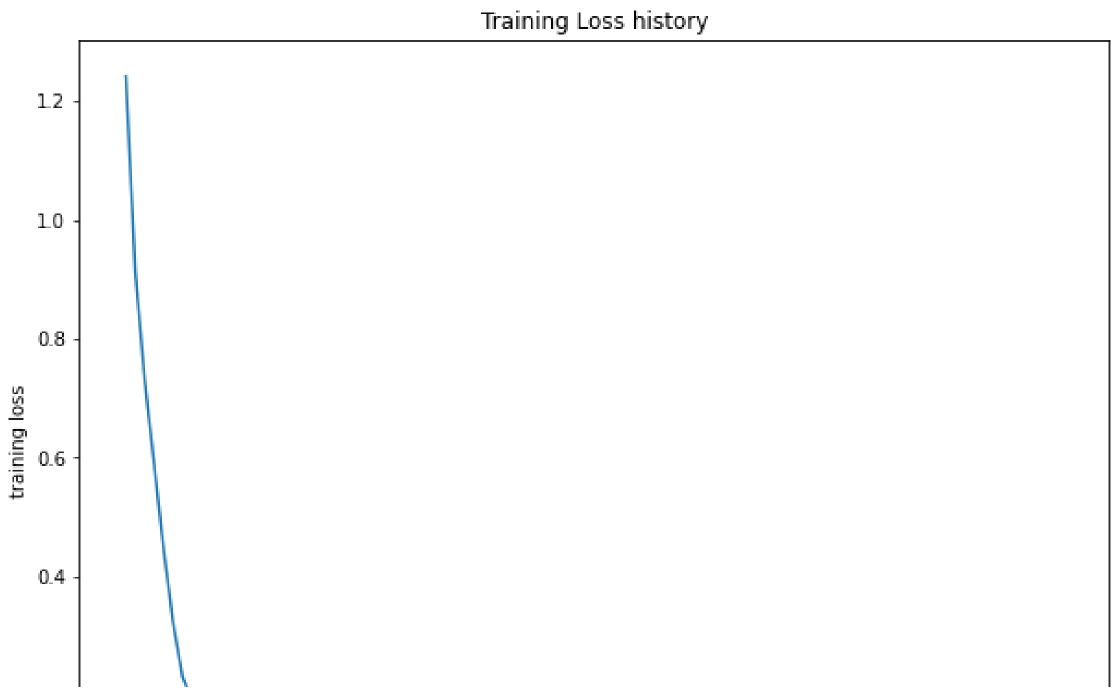
To train the network we will use stochastic gradient descent (SGD), similar to the SVM and Softmax classifiers. Look at the function `TwoLayerNet.train` and fill in the missing sections to implement the training procedure. This should be very similar to the training procedure you used for the SVM and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

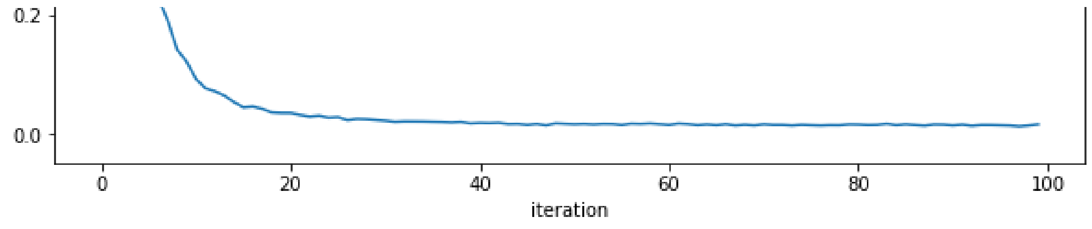
Once you have implemented the method, run the code below to train a two-layer network on toy data. You should achieve a training loss less than 0.2.

In [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 memory is
sue)
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

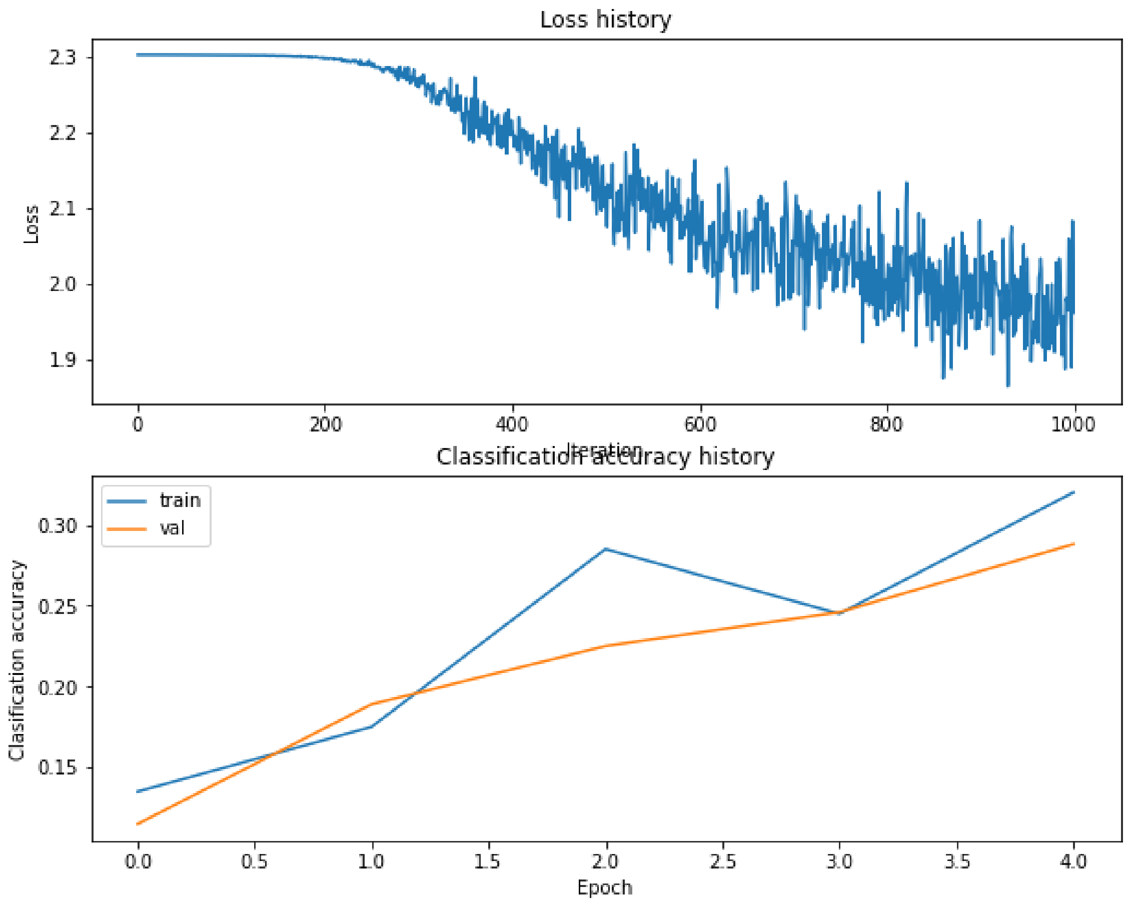
With the default parameters we provided above, you should get a validation accuracy of about 0.29 on the validation set. This isn't very good.

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.

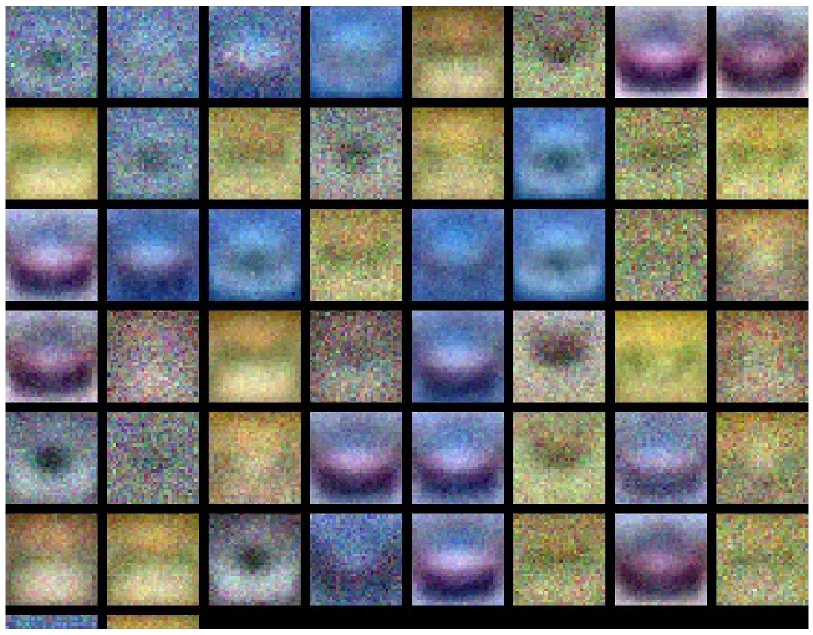
In [9]:

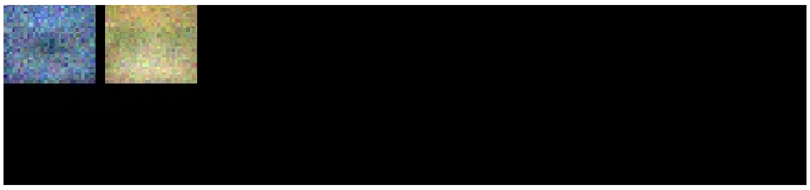
```
# Plot the loss function and train / validation accuracies
plt.subplot(2, 1, 1)
plt.plot(stats['loss_history'])
plt.title('Loss history')
plt.xlabel('Iteration')
plt.ylabel('Loss')

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]:

```
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.).

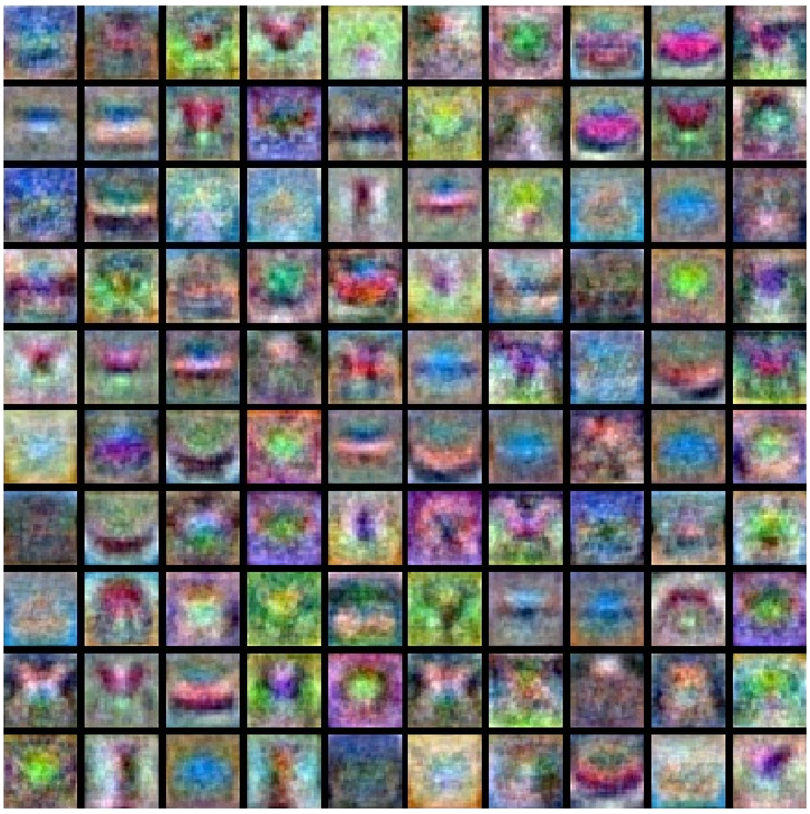
In [11]:

```
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 = 100
output_size = 10

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

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

best_val = -1

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

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

        if val_acc > best_val:
            best_val = val_acc
            best_net = net

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

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

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:  0.482
```

Inline Question

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

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

***Your answer*:**

Your explanation: