Intelligent Systems (IS Fall 2013)

Assignment 1: Neural networks

Student: Fang-Lin He

---

1. (40 points total) Write code to train the single-layer neural network shown in fig. 1.

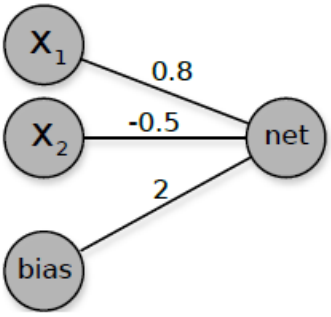


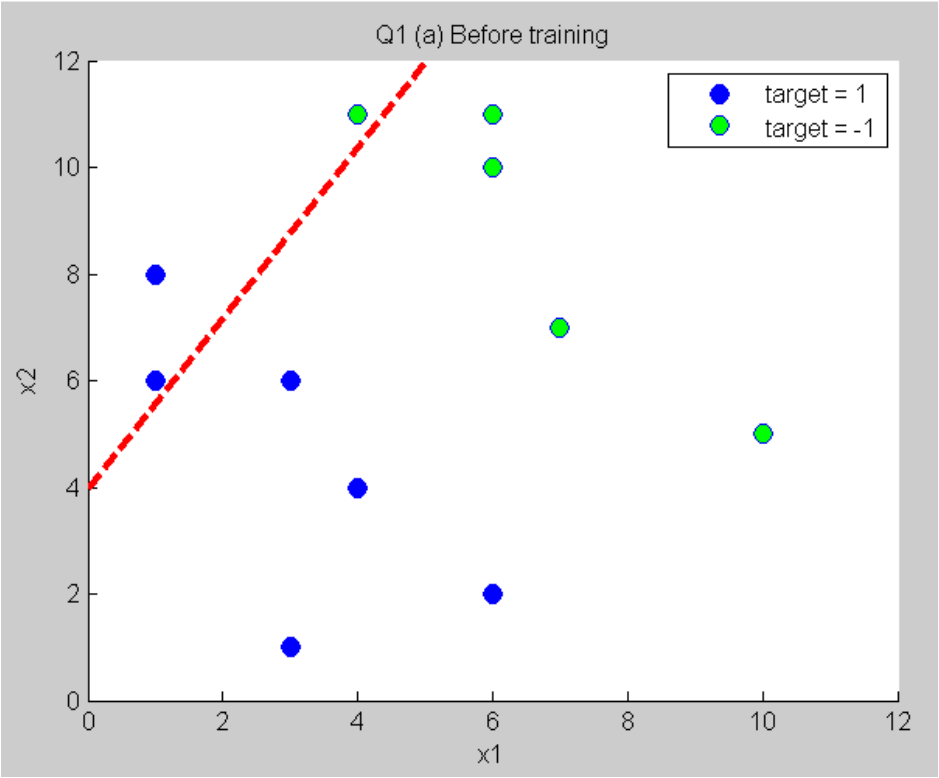
Figure 1: The perceptron network with the initial weight values to be used.

The training set consists of the following points:

- Class 1 (t = 1): (1,8), (6,2), (3,6), (4,4), (3,1), (1,6)
- Class 2 (t = –1): (6,10), (7,7), (6,11), (10,5), (4,11)

(a) (3 points) Plot the training set with the initial separation line where the two classes are displayed in different colours.

>> The input patterns where t = 1 are marked as blue dots; where t = –1 are marked as green dots. The initial separation line is drawn as red dotted line.

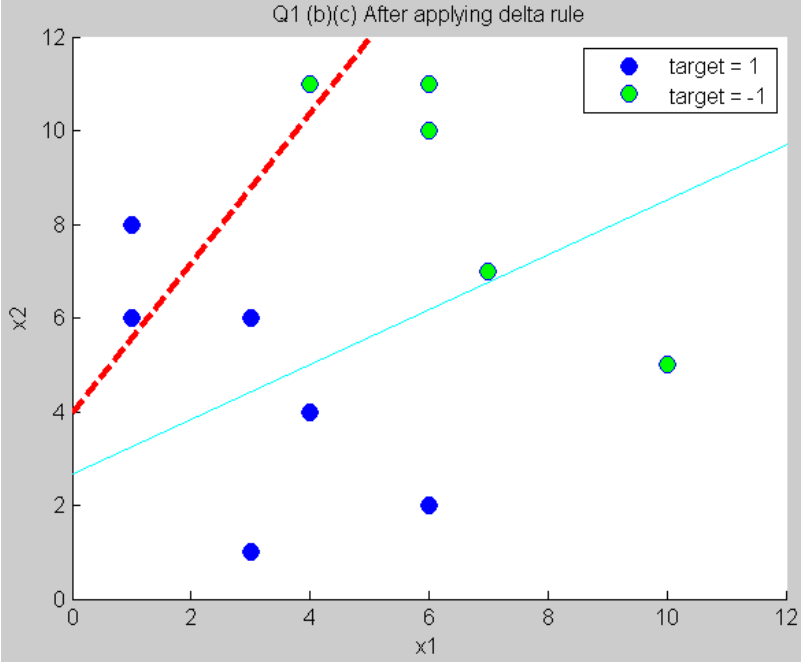


(b) (20 points) Implement the delta rule and apply it for all points from the test set, with a learning rate of $\eta = 1/50$.

>> After applying the delta rule, the weights become 0.4396, -0.7511, and 1.9522 for x1, x2, and bias, respectively.

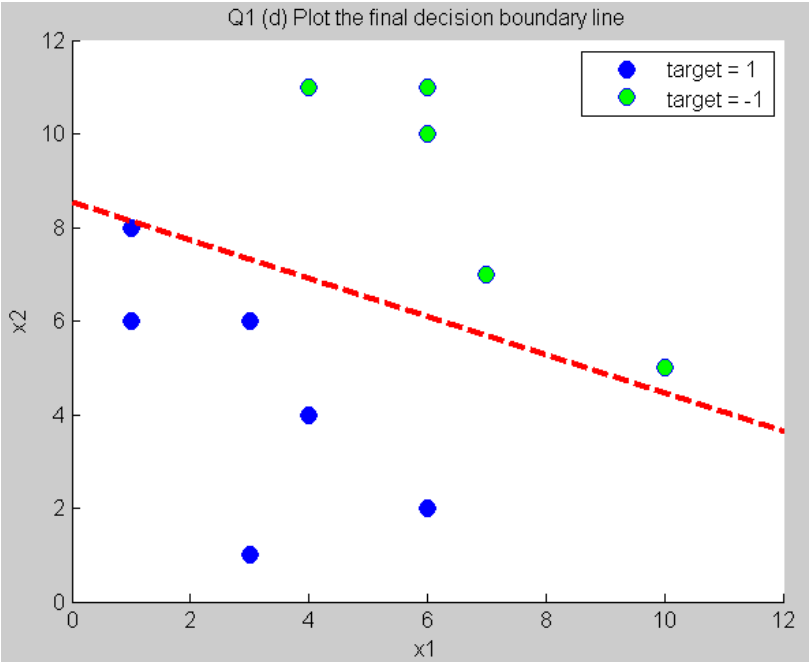
(c) (3 points) Plot the new separation line.

>> The new separation line is drawn as cyan-blue line.



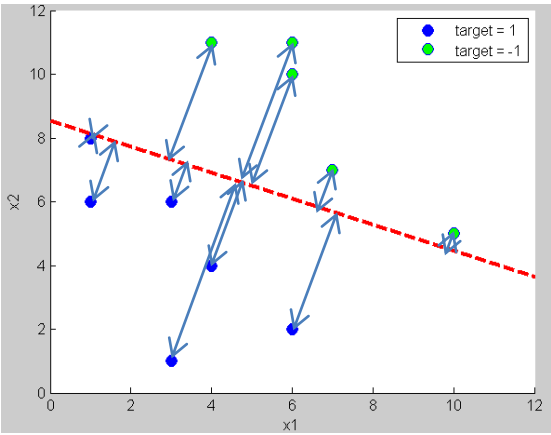
(d) (10 points) Train the perceptron until all the points are correctly classified, and plot the final decision boundary line.

>> The new separation line is drawn as red dotted line. It executed totally 12 iterations, and the final weights are -0.0924, -0.2310, and 1.9633 for x1, x2, and bias, respectively.

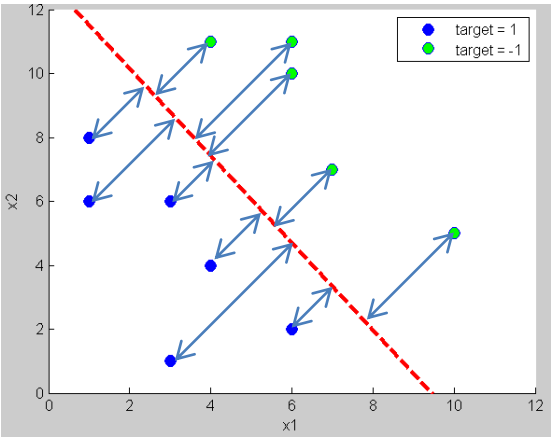


(e) (4 points) Is it a good solution? Discuss potential problems that may arise.

>> Although all the input patterns are classified correctly now, this solution is not good enough. The separation line here is not evenly separating two classes, and this would cause a bad accuracy. If we plot the distances from the input patterns to the separation line, we clearly see that the distribution of distances is not even, that is, some distances are very long and some are very short, as shown in the figure below.

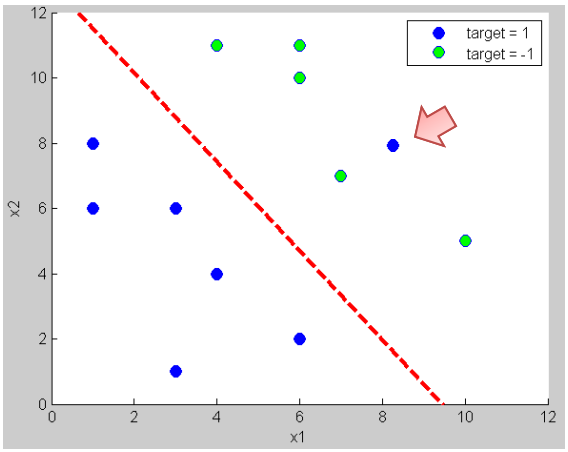


In my opinion, the best solution should be that the separation line separates two classes evenly, like the figure below.



This separation line not only separates two classes evenly, but has the lowest error: $E = \frac{1}{2}(d - net)^2 = \frac{1}{2}(d - \sum_{i=1}^{n} w_i x_i)^2$. So this solution should be better. To obtain this solution, I use the error E as the measurement and stop training until the error does not change dramatically. To be more precise, my solution is to record the errors in the current iteration and the previous one. I set a constant as my stop training threshold; when the difference between these two errors is less than the threshold, it means the solution is good enough, so I can stop training. I set the threshold as 0.0001 in my experiment and get the above result.

To train the perceptron until all the points are correctly classified will also cause another severe potential problem: in some situations, if noise exists, the dots cannot be separated perfectly by a straight line, like the following figure:



Since it is impossible to separate two classes by a separation line, it will get into an endless training loop.

2. (60+5 points total) Write code to train the two-layer feed-forward neural network with the architecture depicted in fig. 2.

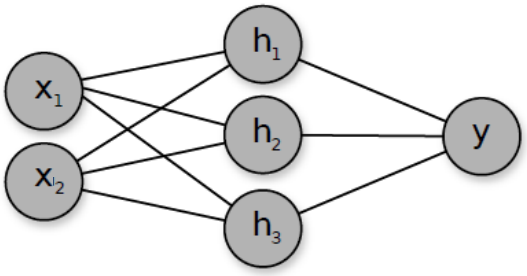


Figure 2: The perceptron network with the initial weight values to be used.

The training set consists of the following points:

- Class 1 ($t = 1$): (4,2), (4,4), (5,3), (5,1), (7,2)
- Class 2 ($t = -1$): (1,2), (2,1), (3,1), (6,5), (3,6), (6,7), (4,6), (7,6)

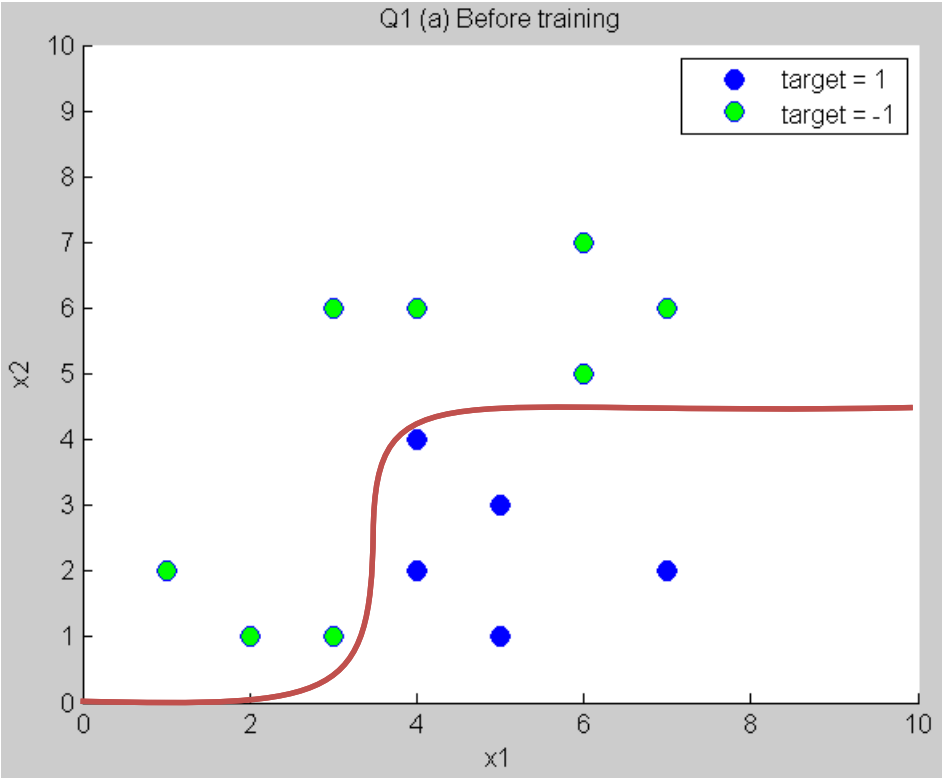
The accompanying test set is:

- Class 1 ($t = 1$): (4,1), (5,2), (3,4), (5,4), (6,1), (7,1)
- Class 2 ($t = -1$): (3,2), (8,7), (4,7), (7,5), (2,3), (2,5)

This network uses sigmoidal activation function (i.e. the logistic function) for the hidden layer and linear activation for the output layer. Also, don't forget the biases (even though they are not shown in the image). The weights (and biases) should be randomly initialized from a uniform distribution in the range [–0.1, 0.1].

(a) (5 points) Plot the data. Would the network be able to solve this task if it had linear neurons only? Explain why.

>> From the figure below, we can clearly see that, if use only linear neurons, this network cannot be solved, since we cannot find a single separation line to perfectly classify these two classes. One of the possible solutions is to use a sigmoid function to separate these two classes, like the red line plots.



(b) (30 points) Implement and apply backpropagation (with $\eta = 1/30$) until all examples are correctly classified. (This might take a few thousand epochs)
>> One of the execution results:
Initial weights for input & hidden layer:

	h_1	h_2	h_3
bias	0.0564	-0.0525	-0.0189
x_1	-0.0799	0.0062	-0.0790
x_2	-0.0412	-0.0817	-0.0775

Initial weights for hidden layer & output layer:

bias	h1	h2	h3
0.056886	-0.041686	0.020707	0.092885

After 9130 iterations, all the training data is classified correctly.

Final weights for input & hidden layer:

	h_1	h_2	h_3
bias	-1.2730	-3.7171	0.4353
x_1	-0.1232	0.6537	-0.0960
x_2	0.5780	1.5090	-0.2439

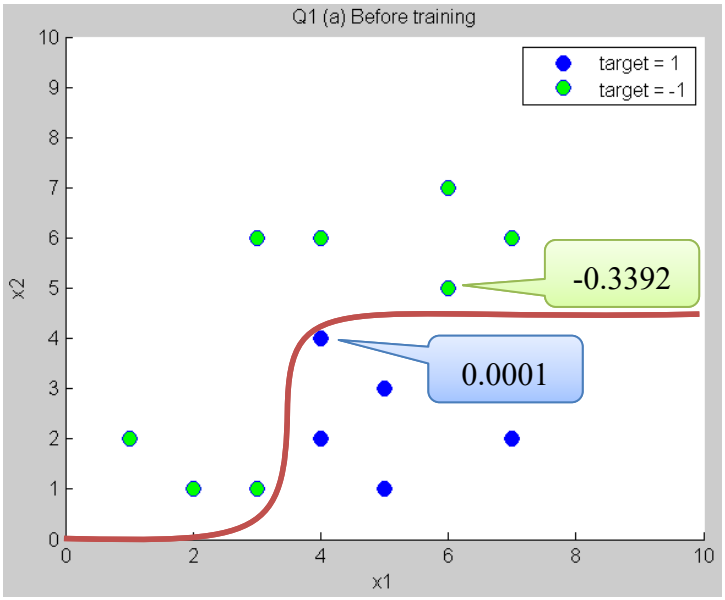
Initial weights for hidden layer & output layer:

bias	h1	h2	h3
-1.830345	-3.466441	3.658388	1.385403

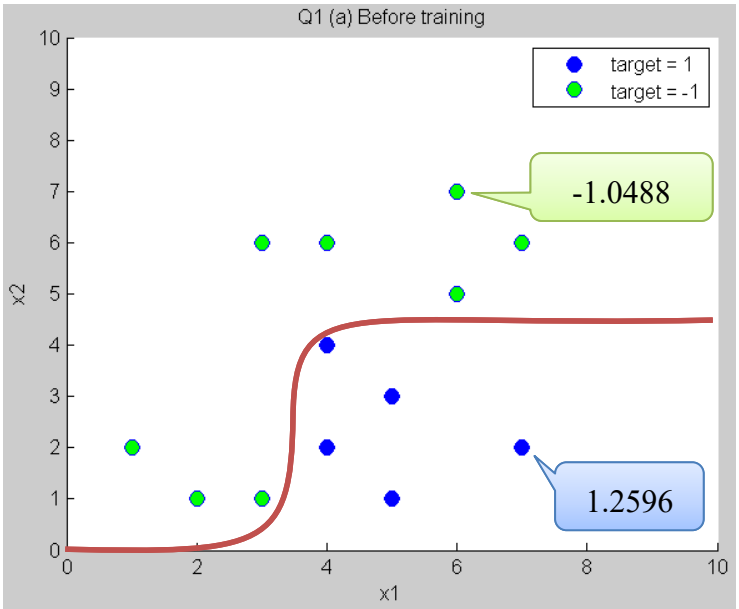
Final y values:

- Class 1 ($t = 1$): 0.6817, 0.0001, 0.6043, 0.7444, 1.2596
- Class 2 ($t = -1$): -0.9273, -1.0536, -0.4554, -0.3392, -0.8669, -1.0488, -0.8339, -0.7010

Here, I found that the results are very reasonable. Since the most "difficult", says, the samples which close to the red line the most, like the below shows:

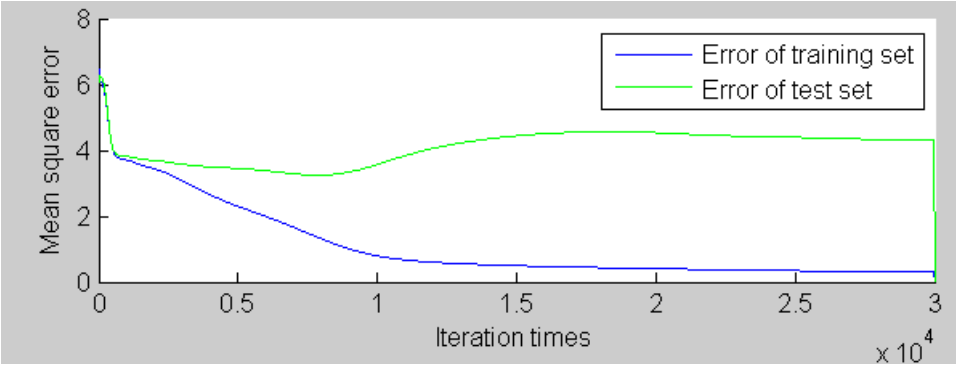


The absolute values are the smallest in all y values. On the other hand, the two furthest data have the largest values, like the below shows:



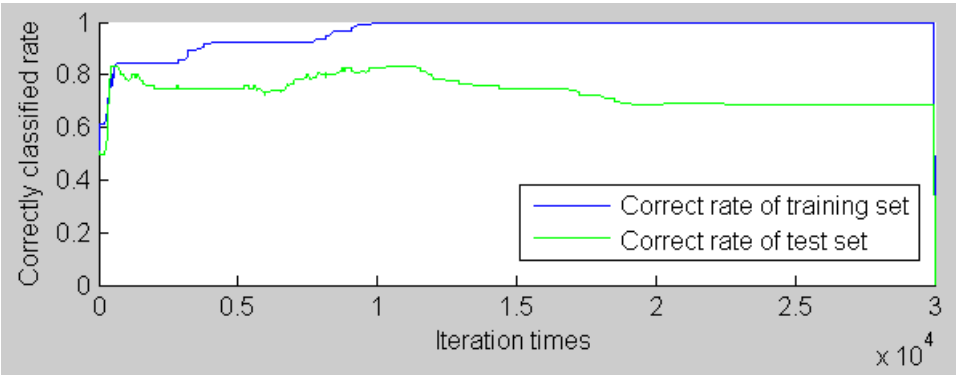
(c) (10 points) Perform 10 runs with different random initializations of the weights, and plot the training and test error for each epoch averaged across the 10 runs. How many epochs does it take, on average, to correctly classify all points. What do you notice?

>> The figure below is the average error of training set and test set.



In my experiment, I run 30,000 iterations to observe the changes of the error. The average number of epochs in average to classify all training points correctly is 8,624 times. From the experimental results, I observed two things. First, after around 9,000 iterations, the error of training set still goes down slowly, but the error of test set starts going up. As professor mentioned in class, we should do "early stopping" when we observe that the error of the test set (validation set) increases rapidly.

On the other hand, I also plot the correctness rate which is calculated by 'the number of correctly classified data points in training / test set' over 'the number of data points in training / test set'. From this figure below,

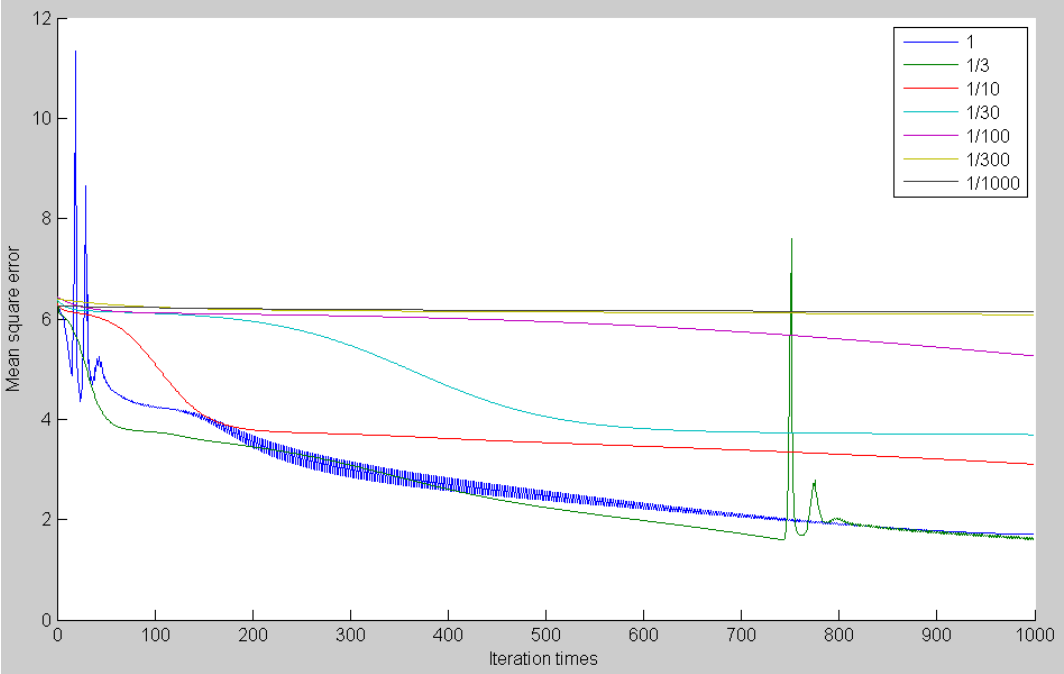


I found that, after around 10,000 iterations, all points in the training set are correctly classified. On the other hand, points in the test set are never classified correctly. Besides, the correctness rate in test set even starts decreasing after around 12,500 iterations. In summary, from the observation of the error and correctness rate, although the error of training set keeps decreasing time after time, we should stop training at around 10,000 iterations.

(d) (15 points) Try different learning rates η = [1, 1/3, 1/10, 1/30, 1/100, 1/300, 1/1000] and plot the training error over 1000 epochs. What is the effect of

varying the learning rate?

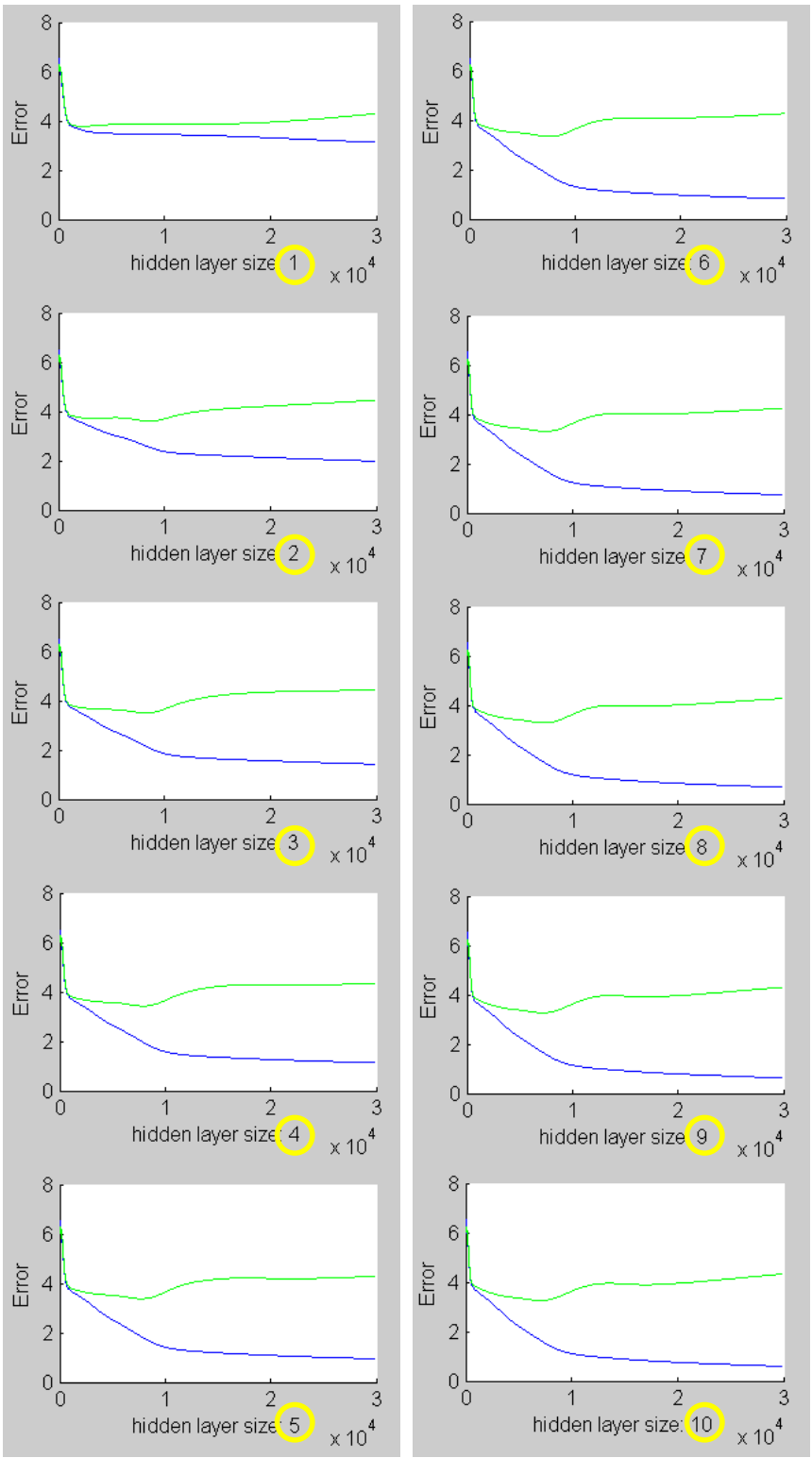
>> I plot the training error over 100 epochs like the below shows:



From this figure, I observed that, if the learning rate is too large, says 1 which is drawn in blue line, the error changes rapidly and is not stable, so it is more difficult to find a minimum. Similarly, from the green line with learning rate = 1/3, we can also see that the error suddenly increase at around 750 iterations. If the learning rate is too small, says 1/300 and 1/1000, the error changes slightly after iteration, so we need more iteration times to reach the local minimum, which means we need longer time to find a good solution. In this plot, I think the 1/10 and 1/30 are the best learning rate, since they do not cause the sudden unexpected change but converge in a reasonable rate.

(e) (Bonus: 5 points) Vary the number of hidden units (from 1 to 10) and run each network with 10 different random initializations. Plot the average train and test errors. Explain the effect of varying the number of hidden units.

>> The execution result is shown as below. Each figure shows the results of varying number of hidden units. The blue line is the error of training set, and the green line is of test set.



From these plots, I figure out that, after 30,000 iterations, the larger number of hidden units, the lower error we get. The trained weights may seem quite ideal for the test set. However, on the aspect of test set, we get the lowest error at around 9000 iteration times; after that, even more iteration times, we get fixed error of around 4.5 or higher. From this observation, we can conclude that, with higher number of hidden units, the trained weights are over-fitting and do not fit the test set better. On the other hand, for with only one hidden unit, even the

error of training set does not converge, which means the model is under-fitting.