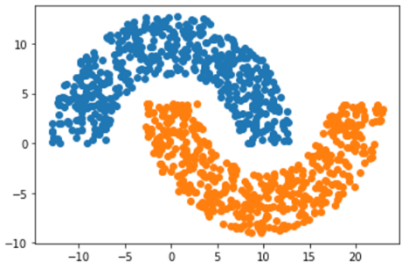
Neural Networks Report #4

Introduction

This makes a great summary of all concepts learned before midterms. That's why many codes is just a duplication of previous homework code with small modifications. Here will be only small comments about the implementation and result will be shown in the pictures. Also I will explain analytical solutions. Again, Jupyter notebook is used and it cells should be run in order to not lose the information

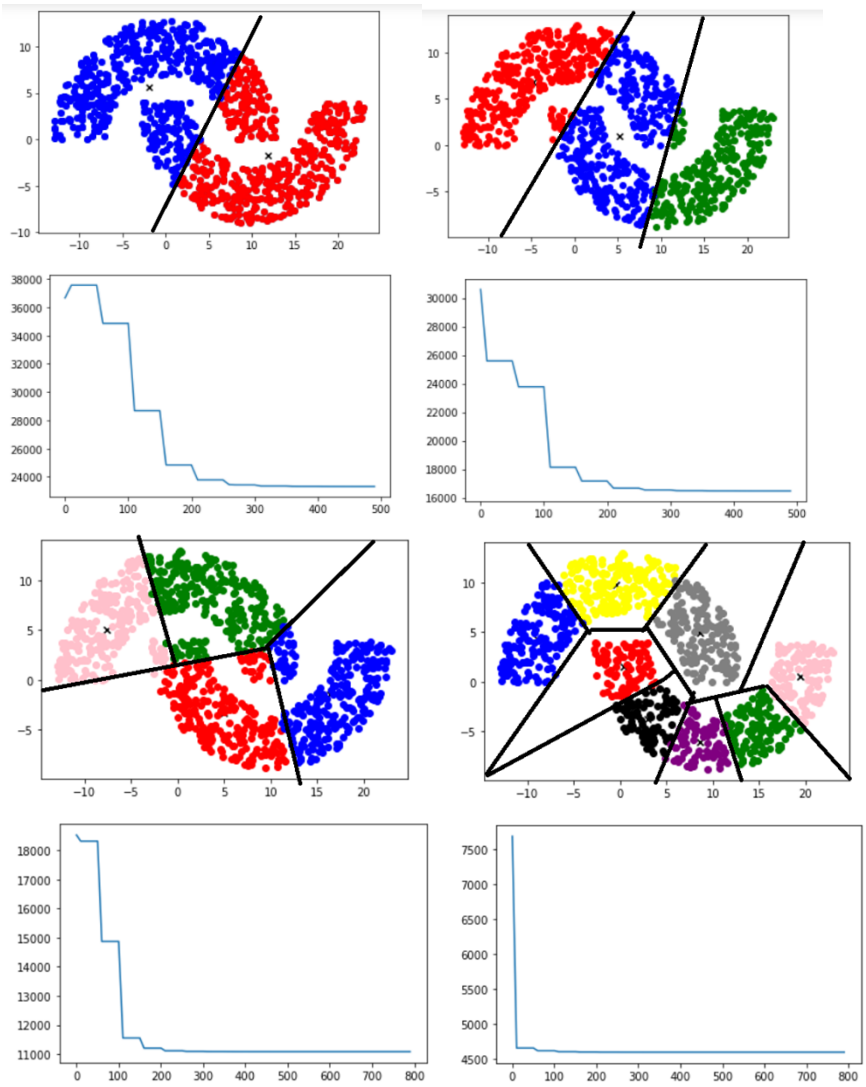
Problem 1

This is direct implementation of homework1, uniform distribution in rectangular region is created and if point is inside the donut region, it is added to data.



Problem 2

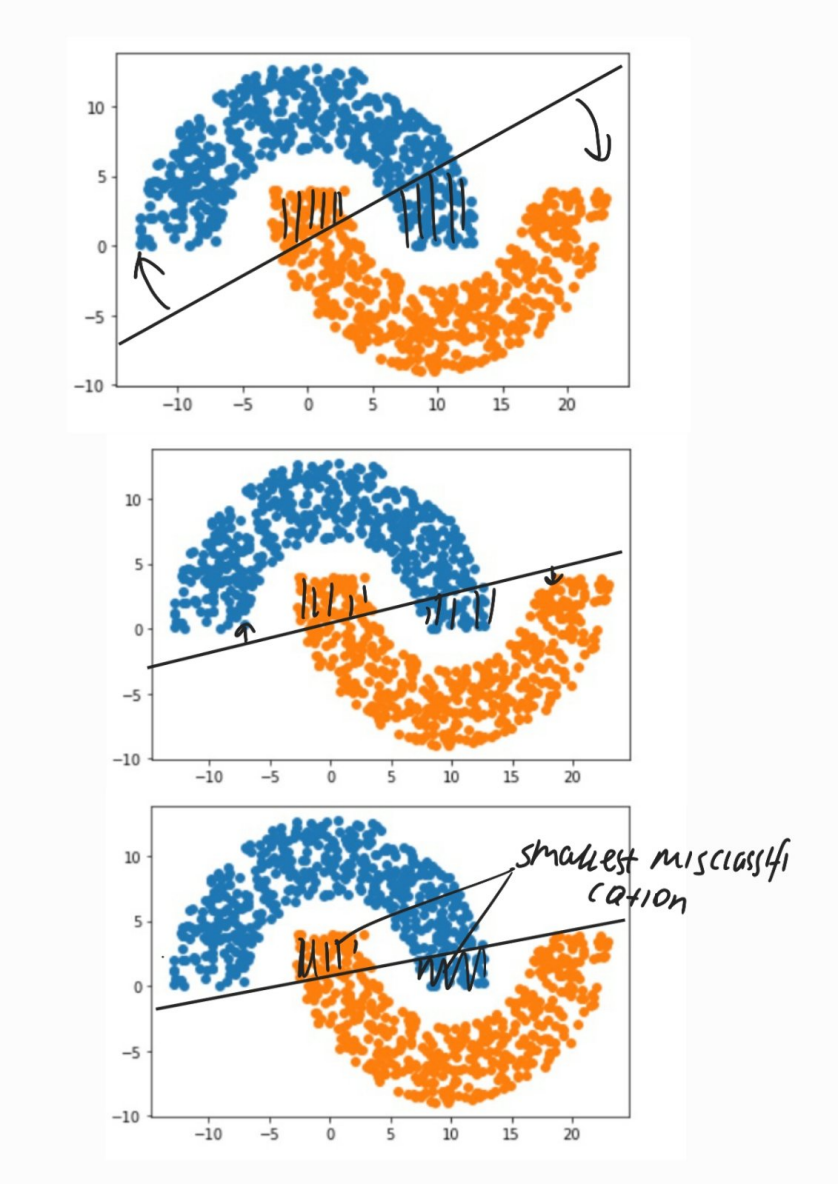
The functions are copied from homework1 competitive learning, only for making 8 clusters is created. The results are below. Because programming the boundary line from the clusters is another machine learning challenge, it was done by myself, the code itself divided clusters by different colors. The cluster centers are shown by x. For good analysis, I also added loss function.



All function loss converges to minimum well and without noise. Interesting that the more clusters, the faster the function converges, may be because of complicated data.

Problem 3

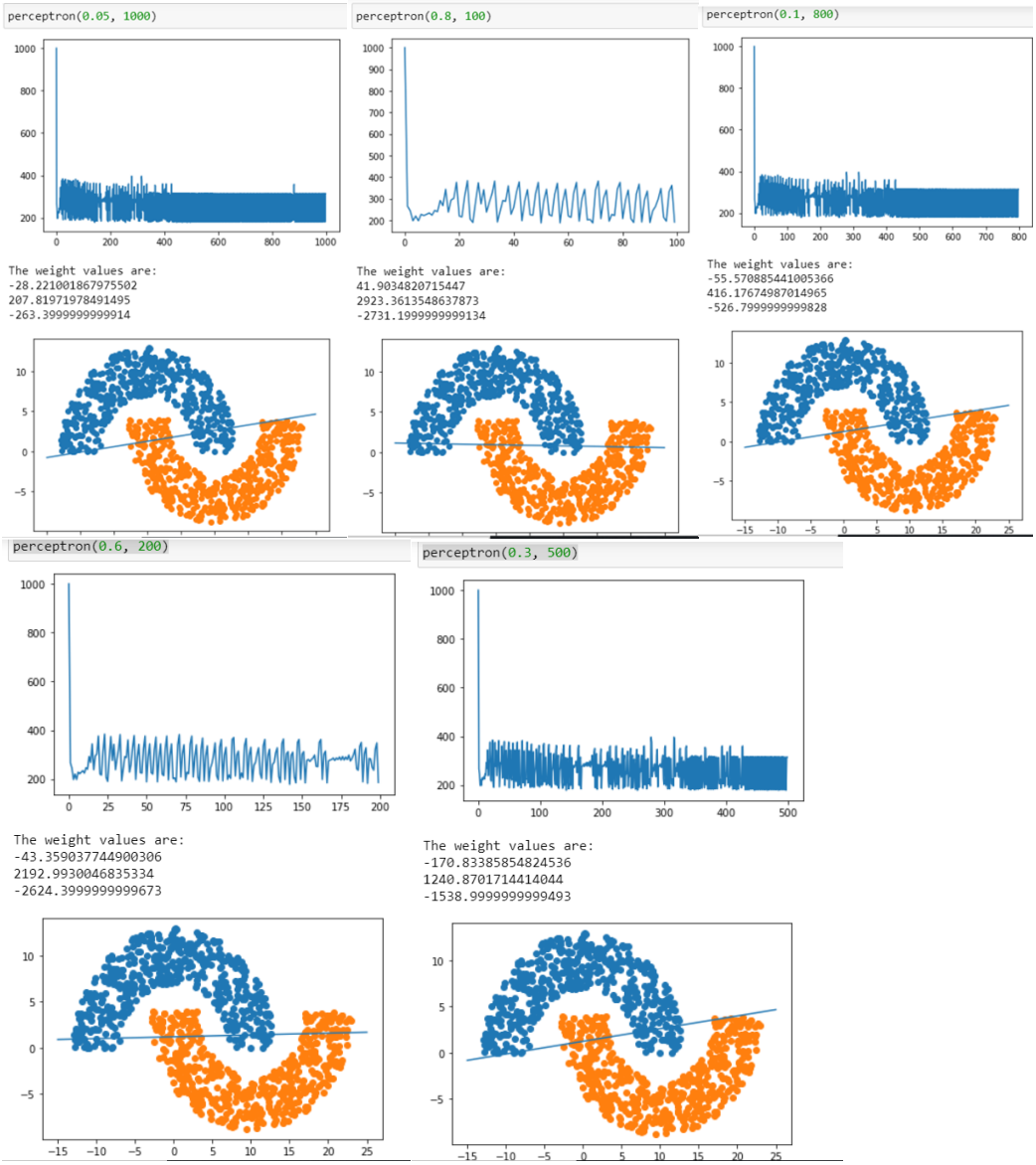
This part is done by the strategy of Professor that he used in explaining the problem 3. We make a boundary by hand such that the misclassification is minimized, if it's not, we just move in the direction of minimization, below it is shown step by step



We first create a border by random and see the misclassification, from there we conclude that there is a way to change the border and reduce error, we go to the second picture and still see that there is some free space until we can rotate the line so misclassification is reducing and rotate it. Finally, wen the line touches the other part of data we can conclude that the misclassification is minimized as rotating it in any directions more will produce more errors, same as translating. This is not mathematical solution, but instead analytical based on Bayesian model

Problem 4

Again, code of perceptron is copied from homework3, function is created and used for different learning rates and epoch. Also note that I added the bias for a better performance. Experimenting with parameters we have



In all cases the loss function is fluctuating, means it is oscillation about its minimum and every time skips it. Even with very small learning rate it is not ideal. The reason again hides in the shape of data, with is vary balanced between two sides and makes the perceptron confused in every iteration. For large learning rate I used small number of epoch since it should converge faster by theory, as step is big. And the result with highest lr is the least precise one. On the other hand, the border that agrees with analytical solution is when learning rate is 0.05 and epoch number is 1000. It is very certain, as it goes slowly to its minimum, therefore should not overstep it. But unfortunately, it is least computational efficient one.

Problem 5

From previous problem I choice lr = 0.05, and epoch = 1000. I made a function that initialized the network with random weight values in range of -5 to 5. This region is chosen since by intuition the slope and bias can not be outside this high range in the given data. Then, I created the same function

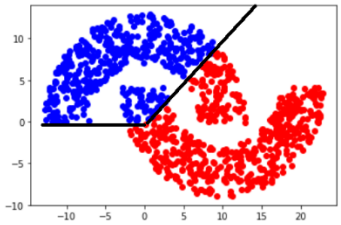
random_perceptron that only uses random initial weights and ran it 5 times. Interestingly the result was different each time and result never converge to theoretical optimal value. Reason may be because the parameters were chosen from the case when everything was 0 initially, and those parameters are not good for the random case, so perceptron is very sensitive to the lr and epoch number

Problem 6

One general function for single hidden layer perceptron is made.

For simplicity, I removed the bias part. My sigmoid was giving an overflow so I added a boundary in +-100 where it becomes 1/0. Also to make the performance less sensible to slight changes, the range of the random weights was decreased.

It uses the matrix w1 and vector w2 as weight storage and its output is single neuron which gives the probability of point to be in class “A”. So, if it is less than 0.5, it will be automatically in class “B”. Tuning this network was a tough process as it is very sensible to the changes. The result of classification is then



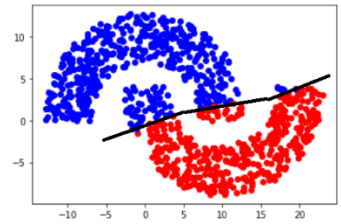
Which has a not small misclassification error, may be 2 neurons is not enough. It is hard to draw the boundary in this case as extracting equations from weights is impossible. I could not achieve it by code, so made it by hand. Loss decreases very slightly, so does not give any efficient information, I ignore it and added just for reference.

Problem 7

Done by for loop and we see that code is very sensitive to random weight and sometimes it makes only one class classification. Thus, the parameters should be tuned very carefully.

Problem 8

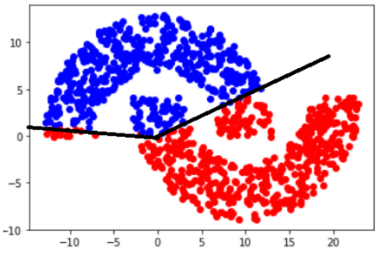
RelU gives a new challenge as previously sigmoid output was taken as probability of being in any class, how should we interpret RelU? The solution can be so if it is positive, it is in class “A”, if it is 0 in class “B”. We get



Which is fine, but with 5 different random weights it also sometimes gives a bad result. Thus, the random initialization makes the approach more complex.

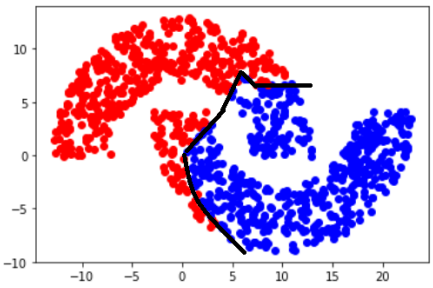
Even with weak results in both, RelU function seem to perform better as it requires less epoch number for training and gives strange results less often. Additionally, it did not create any problems with overflow and its code was a little bit simpler to implement. Probably, that's the reason why it is more popular.

Problem 9



The result is better than two neuron case. It is really hard to tune this network. The loop also gives very unstable results as it is very sensible to parameters

Problem 10



This one is also hard to tune. Interestingly, there is small interval when it converges, this is really small one. If your epoch is lower than some region, everything is red, if slightly larger, everything is blue

Problem 11

Code is similar to the single hidden layer perceptron, one additional is just added. Tuning is again very tough task. We get