ECE174 Introduction to Linear and Nonlinear Optimization

Created by: Aleksandar Jeremic
For: Professor Piya Pal, Christine Lind
December 12th 2023
Fall 2023

0.1 Introduction

In MiniProject 2 we were tasked with solving the following Non-Linear Least Squares Problem (NLLS) in application to a neural network of the form

$$\hat{y} = f_w(\mathbf{x})$$

A neural network is referred to as a "universal approximator" this is because a suitable tuning of the weights can allow this network to approximate a very large class of functions.

Our neural network will be of the form

$$f_w(\mathbf{x}) = w_1\phi(w_2x_1+w_3x_2+w_4x_3+w_5)+w_6\phi(w_7x_1+w_8x_2+w_9x_3+w_{10})+w_{11}\phi(w_{12}x_1+w_{13}x_2+w_{14}x_3+w_{15})+w_{16}$$

where our ϕ is the hyperbolic tangent function defined as follows

$$\phi(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

My code was created to determine the weights (w in the equation) to approximate a function which is given using the 3 input variables (x). We give the neural network 500 data points for a function of the form. with the given constraint

$$y^{(n)} = x_1^{(n)} \cdot x_2^{(n)} + x_3^{(n)}$$
$$\max\left|x_1^{(n)}\right|, \left|x_2^{(n)}\right|, \left|x_3^{(n)}\right| \quad \leq \Gamma = 1$$

We assign 500 random points along this line for our neural network to train on and to minimize the following loss function.

$$l(w) = \sum_{n=1}^{N} r_n^2(w) + \lambda\|w\|_2^2$$

First let us compute the gradient of our neural network in the above form, we will take the gradient with respect to each individual weight w.

$$\begin{bmatrix} \tanh(w_2 \cdot x_1 + w_3 \cdot x_2 + w_4 \cdot x_3 + w_5) \\ w_1 \cdot \tanh'(w_2 \cdot x_1 + w_3 \cdot x_2 + w_4 \cdot x_3 + w_5) \cdot x_1 \\ w_1 \cdot \tanh'(w_2 \cdot x_1 + w_3 \cdot x_2 + w_4 \cdot x_3 + w_5) \cdot x_2 \\ w_1 \cdot \tanh'(w_2 \cdot x_1 + w_3 \cdot x_2 + w_4 \cdot x_3 + w_5) \cdot x_3 \\ w_1 \cdot \tanh'(w_2 \cdot x_1 + w_3 \cdot x_2 + w_4 \cdot x_3 + w_5) \\ \tanh(w_7 \cdot x_1 + w_8 \cdot x_2 + w_9 \cdot x_3 + w_{10}) \\ w_6 \cdot \tanh'(w_7 \cdot x_1 + w_8 \cdot x_2 + w_9 \cdot x_3 + w_{10}) \cdot x_1 \\ w_6 \cdot \tanh'(w_7 \cdot x_1 + w_8 \cdot x_2 + w_9 \cdot x_3 + w_{10}) \cdot x_2 \\ w_6 \cdot \tanh'(w_7 \cdot x_1 + w_8 \cdot x_2 + w_9 \cdot x_3 + w_{10}) \cdot x_3 \\ w_6 \cdot \tanh'(w_7 \cdot x_1 + w_8 \cdot x_2 + w_9 \cdot x_3 + w_{10}) \\ \tanh(w_{12} \cdot x_1 + w_{13} \cdot x_2 + w_{14} \cdot x_3 + w_{15}) \\ w_{11} \cdot \tanh'(w_{12} \cdot x_1 + w_{13} \cdot x_2 + w_{14} \cdot x_3 + w_{15}) \cdot x_1 \\ w_{11} \cdot \tanh'(w_{12} \cdot x_1 + w_{13} \cdot x_2 + w_{14} \cdot x_3 + w_{15}) \cdot x_2 \\ w_{11} \cdot \tanh'(w_{12} \cdot x_1 + w_{13} \cdot x_2 + w_{14} \cdot x_3 + w_{15}) \cdot x_3 \\ w_{11} \cdot \tanh'(w_{12} \cdot x_1 + w_{13} \cdot x_2 + w_{14} \cdot x_3 + w_{15}) \\ 1 \end{bmatrix}^{\mathbf{T}}$$

simply apply the chain rule when the w is inside the hyperbolic tangent function and the gradient is a simple but slightly tedious procedure that requires careful implementation, small errors can cause a butterfly effect compounding tons of error.

Now we will compute the Jacobian of our entire set of data points, this involves multiply our data matrix with our gradient function, this creates a matrix of size 500,16 needless to say I will not be showing the matrix in my report

$$\begin{bmatrix} \frac{\partial f_w(x^1)}{\partial w_1} & \frac{\partial f_w(x^1)}{\partial w_2} \cdots & \cdots & \frac{\partial f_w(x^1)}{\partial w_{16}} \\ \frac{\partial f_w(x^2)}{\partial w_1} & \frac{\partial f_w(x^2)}{\partial w_2} & \cdots & \frac{\partial f_w(x^2)}{\partial w_{16}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_w(x^{500})}{\partial w_1} & \frac{\partial f_w(x^N)}{\partial w_2} & \cdots & \frac{\partial f_w(x^N)}{\partial w_{16}} \end{bmatrix}$$

our y function in our r(w) is a constant with respect to all of the w's so it becomes 0 in the jacobian
where our r is defined as

$$r(w) = \sum_{n=1}^{N} (f_w(x^{(n)}) - y^{(n)})$$

Now we need to minimize the loss function using NLLS, we are going to use the Levenberg Marquadt algorithm to approximately solve the NLLS above.

0.2 Levenberg Marquadt

The Levenberg-Marquardt algorithm involves solving a NLLS using the following steps

Initialization: Pick an initial point $w^{(0)}$, an initial trust parameter $\lambda^{(0)} > 0$, and constants $0 < \alpha < 1$, $\beta > 1$.

For $k = 0, 1, 2, \ldots, k^{\max}$:

$$\text{First-order approximation of } r \text{ at } w^{(k)} : \tilde{r}(w; w^{(k)}) = r(w^{(k)}) + Dr(w^{(k)})(w - w^{(k)}). \tag{1}$$

$$\text{Solve the linear least squares problem: } w^* = \arg\min_w \left\| \tilde{r}(w; w^{(k)}) \right\|_2^2 + \rho^{(k)} \left\| w - w^{(k)} \right\|_2^2. \tag{2}$$

Check and update:

$$\text{If } \|r(w^*)\|_2^2 \leq \left\| r(w^{(k)}) \right\|_2^2, \text{ set } \rho^{(k+1)} = \alpha \rho^{(k)} \text{ and } w^{(k+1)} = w^*. \tag{3}$$

$$\text{Otherwise, set } \rho^{(k+1)} = \beta\rho^{(k)} \text{ and } w^{(k+1)} = w^{(k)}. \tag{4}$$

I set my alpha value to 0.8 following the book and my Beta to 2 also from the book, I used different values but in general, I didn't find a big difference. My stopping criterion involved a few features, if the error was below a small value, I used .1 as my stopping value, I also stopped at an iteration limit of 300, I found increasing the iteration limit past this point is usually pointless, either it has reached the error limit or it would take a very long time to run much further. I finally stopped my code if rho ever increased above 100, I did this because I found more often than not if rho is increasing above 100 there is an issue with either initialization or it getting stuck and it would make the rest of my data almost unreadable if it could go above 100.

0.2.1 Different Initializations and Standard Deviations

Below is the training loss of the algorithm for different values of lambda (the regularization parameter) and different initializations where I change the standard deviation of my initial weights.

We also found on average our final training error would be on the scale of 10^{-1} very rarely it would jump to 10, this is caused by bad initialization or not enough iterations in the case the weights need extra time to reach the final value.

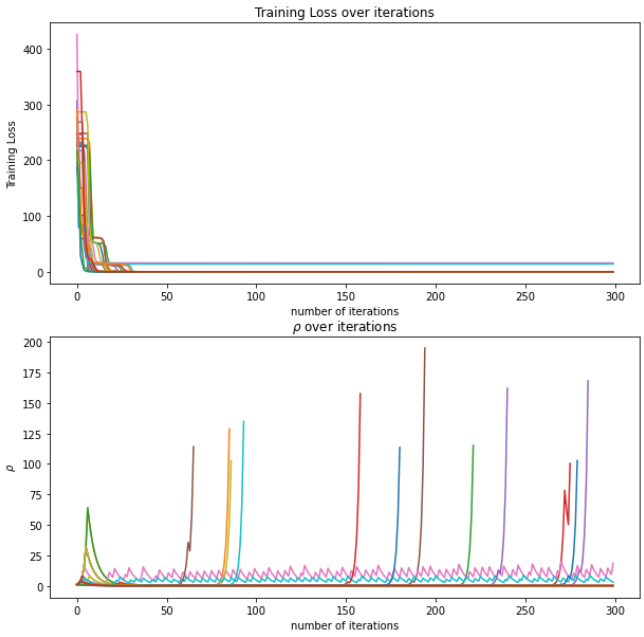

Figure 1: Training Loss and ρ Value by Iterations

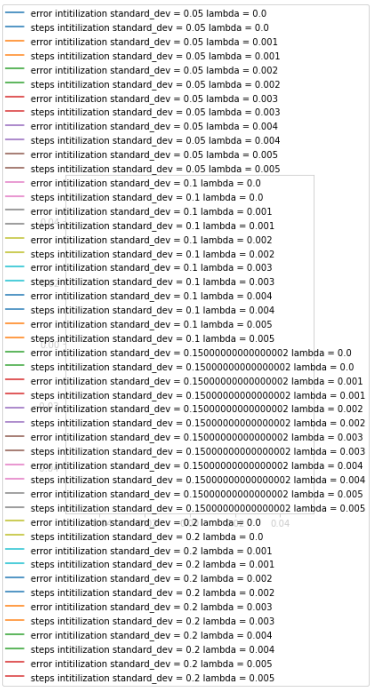

Figure 2: Legend

In general all of the loss functions took longer to reach a low error value regardless of how we initialized on the except of too high of a standar deviation, occassionally we would end up in a position in which our weights didn't converge very well and our ρ value would approach infinity, to combat this we didnt let ρ past about 100-1000 to make sure we can see how the other iterations and initializations actually moved (if one ρ reaches 10^{30} we wouldnt be able to see our other ρ values which are sitting below 10 on average.

0.2.2 Different λ and Γ Terms Testing

Next, We need to test our code on a testing set that is related to our training set only because it has the same function, we need to make sure we are not reusing our previous training set data, so we reinitialize with 100 data

points on the same function. We run our algorithm to get the weights for the training set, and then we use those weights and directly plug them into our training loss function to determine the errors. I do this once again for a varying λ regularization parameter and different Γ values where Γ is our parameter being used to initialize all our points within a certain domain of our nonlinear function given by the function. In all future cases where I change λ and Γ I hold my weights initialization to a random normal distribution with a standard deviation of 0.1.

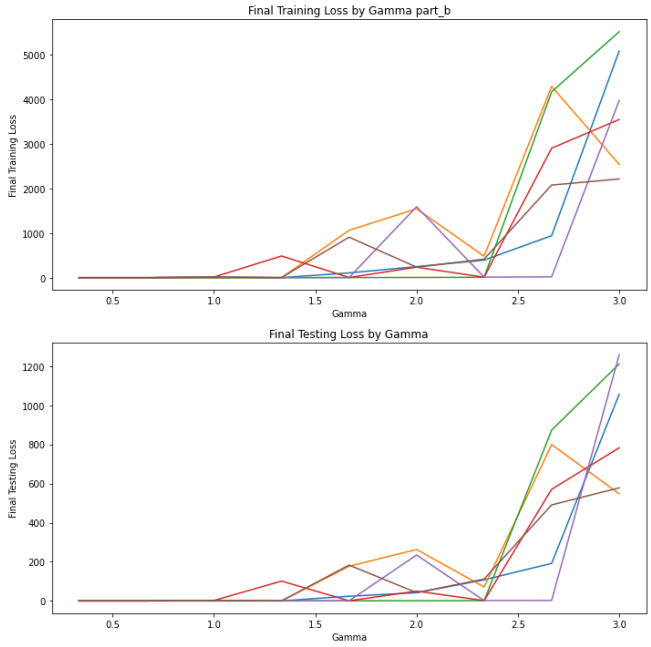

Figure 3: Training/Testing Loss by Γ and λ

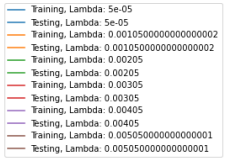

Figure 4: Legend

We see a massive increase in training loss as we increase gamma, this is expected because as we increase the cube which our function must approximate under we DO NOT increase the number of points it can train on. When we double the length of a side of the cube we have increased the area by r^3 this means when we reach $\Gamma = 3$ We are approximating 27 times the area with the same number of points, this obviously will lead to serious approximation issues. I expect to see this trend among all cases of us increasing the Γ value.

0.3 New Nonlinear Function

Now I have concocted my own non linear function, I wanted to make this one strange, so I implemented

$$y = x_1^2 \cdot \max(x_2, 1) + \max(x_3, 1) \tag{5}$$

It's a weird function that implements a strange ReLU looking function added along with a squared term multiplied by another ReLU lookalike. I reuse all the same code from the previous part with exception to the fact that I now must create a new function to generate the new data with.

0.3.1 Different Initializations and Standard Deviations

first, let us run the training data and try different initializations and different lambda functions

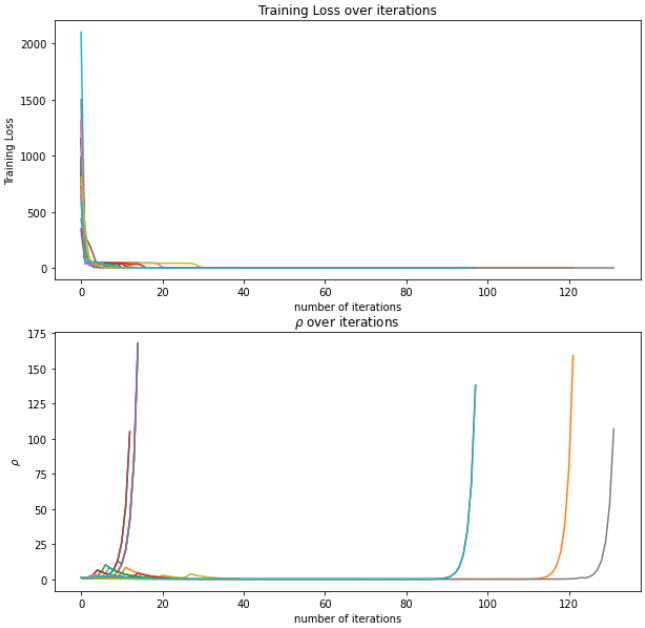

Figure 5: Training Loss and ρ Value by Iterations

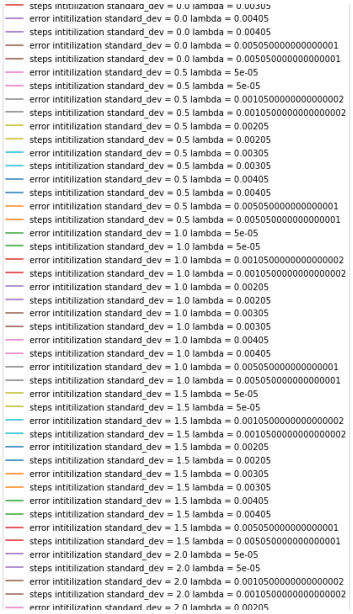

Figure 6: Legend

We actually see our training loss function reaches the minimum error threshold quite quickly and very few initialization and λ make it past 100 iterations before either reaching the ρ limit and none reached the iterations limit. I believe this is probably because our function actually might be quite easy to approximate because of the ReLU looking functionality, for instance, $x_2 \leq 1$ we have an x^2 plus a line or just x^2 so I believe this function actually behaves well enough in certain regions which makes it easier for the neural network to approximate. Also as expected our final training error in general would have similar if not slightly lower error than our original function once again I believe this is because our function is very easy to generalize for certain regions and similar in difficulty to generalize on in most other regions of our cube

0.3.2 Different λ and Γ Terms Testing

Now we will test our code on a testing set similar to our previous example on our first nonlinear function. We will also run using different Γ and different λ values while holding our standard deviation of weight initializations to a constant 0.1

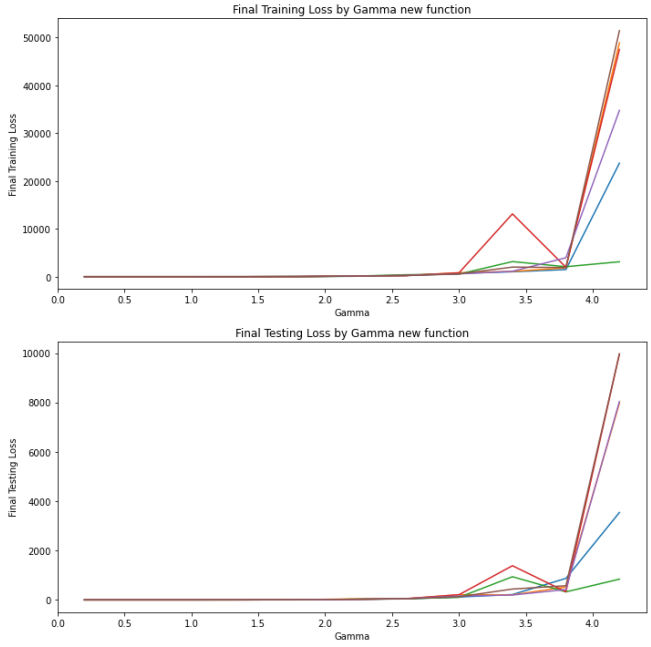

Figure 7: Training/Testing Loss by Γ and λ

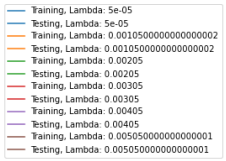

Figure 8: Legend

We can see low testing loss for a Γ below about 2 we start to see alittle bit of movement which on this scale is a large amount of error and then it exponentially increases as we increase the totaly number of possible points which our neural network must approximate while maintaining a low number of training points. This makes sense that we would have such High error for such a large Γ

0.4 Noise Filled Data

First we are implementing a new function to generate noisy data, then we rerun the first 2 experiments exactly the same, first we are changing the initialization and the lambda, then the lambda and the Gamma, then we run our test with testing data and we are able to truly determine if the neural network is overfitting on noise and perhaps this new testing set will yield much higher error for that reason.

0.4.1 Different Initializations and Standard Deviations

Finally we will implement noise in our system. I use the original function but I include an epsilon of varying magnitudes to determine how noise would affect my networks ability to approximate my function.

We can see our Training Loss begins much higher than any other section so far, We see for noise values of around 0.006 randomly seeded in our distribution we have a final training error of about 200 depending on initialization this can also stay very low, this is hundreds of times worse than originally, however this makes sense because this noise level is quite high! When initializing well we can still see a low error of around a tenth but this means that we have fitted around the noise which is not a good thing. the original noise levels are fairly low so even if it fits on the noise the error isnt too egregious. We can actually see where the noise level was too high there is a short asymptote at around 200, those are noise levels from .006 to about .01. We can also see not all of these asymptote

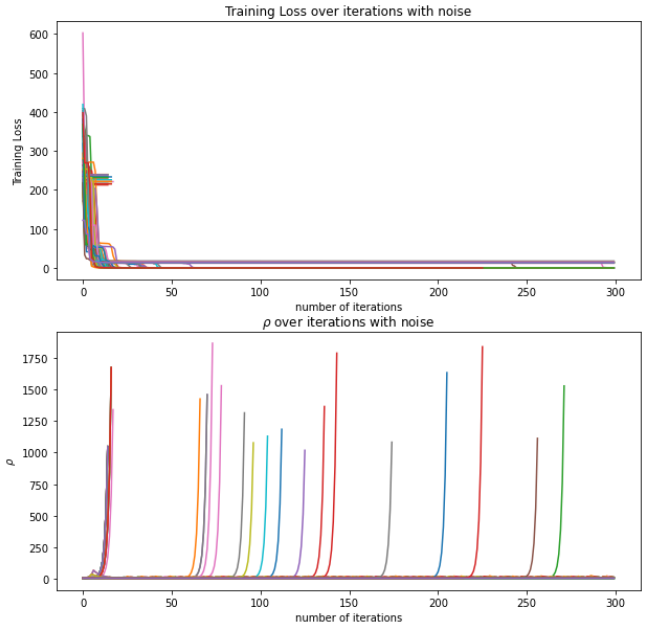

Figure 9: Training/Testing Loss by Γ and λ

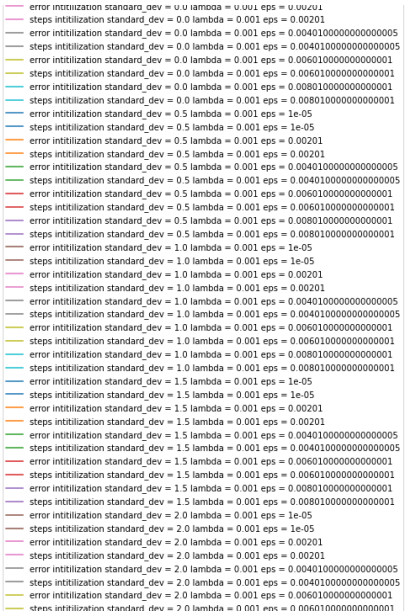

Figure 10: Legend

to the same value, but in general the same noise level almost always asymptotes at the same value. Later we will see how this is affected with changing Gamma and changing noise levels.

0.4.2 Different λ and Γ Terms Testing with changing ϵ values

Here we see how the noise affects our function when increasing Gamma, here we see extremely large error for almost all cases. The error compounds from the noise as well as from the increase in Γ the function is borderline unapproximatable on 500 points, we can say this because at our worst case we have training error of over 20000 dividing this by the 500 points we get a training loss of about 40 per points, this is 400 times more error than some of our $\Gamma = 1, \epsilon = 0$ cases. Also the lower testing error is very misleading because we are only training on 100 points, so we could assume if we tested on 500 points our testing error would on average be about 5 times greater, this would mean worse error than our testing set which is something we would just about expect.

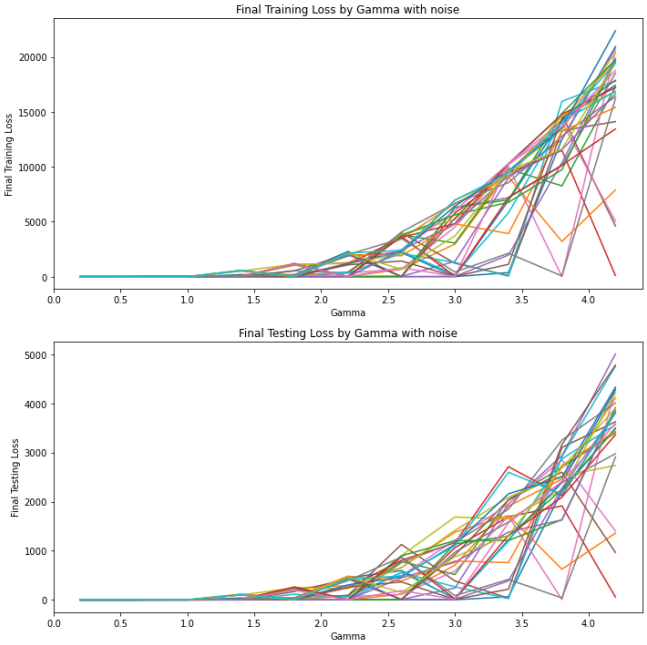

Figure 11: Training/Testing Loss by Γ and λ

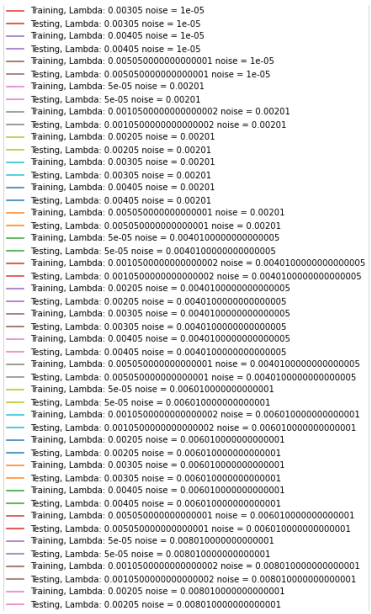

Figure 12: Legend

0.5 Conclusion

In this mini project I created my first ever neural network which I used to approximate 2 different nonlinear functions, I used a technique for solving a NLLS problem such as approximating a non linear function. I specifically used the Levenberg-Marquadt algorithm to minimize my objective (loss) function. I linearized my neural network and used Levenberg-Marquadt to minimize the loss function by changing the weights. From there I wrote a code which would compute all of this for me. During the process I also ran the code for varying λ, Γ and ϵ parameters testing my weights on a new testing set. I also implemented a new non linear function to approximate and I testing my neural network on this new function with minimal code changing showing just how versatile this code is at approximating a wide variety of functions. I would also like to finalize by saying, everytime I ran a simulation with a new set, I reran the weight calculation, this is why sometimes the data might actually appear to decrease sometimes before it jumps back up, the general trend is more important and sometimes in a few hundred runs of the same algorithm the weights wont be the best or theyll be amazing compared to previous runs. I think its important to say this because it shows how variable data is and that its better to rely on the general trend than a specific line because of how randomness when generating these weights can play a role in the final response. And I dont want to hide any of the cases where the algorithm didnt work as well as it could have because it shows the learning procedure of intializations and how not everything will always work.