When deciding how to design an artificial neural network there are several decisions one could make when creating the implementation. Some choices which have been already made for this network are:

* The type of ANN
* Subject to learn
* How clean the data is
* Number of inputs
* Number of outputs

One potential decision that has already been made is the type of ANN. In this case we are to implement a simple back propagation network. Another is what the neural network will be made to learn about. In this case we have been provided two functions that the network will need to learn to approximate. Another decision (which may not necessarily be a realistic choice in a real world scenario) is how clean the learning data is. In this case we will implement one network that has ideal data as an input, and another that has noise incorporated into the learning. When designing a network, one aspect that needs to be defined for implementation is picking the number of inputs and outputs. Depending on what you are attempting to model there are many choices that could potentially be made. Such as whether or not to include certain aspects of a system as inputs to the network. In this case we were provided an already defined function with a given number of inputs and outputs, so there are no decisions to be made in this regard.

Even with these portions of the network already decided before beginning, there is still more to decide on to complete the network, such as:

* Network Depth (number of hidden layers)
* Network Width (number of neurons in each layer)
* Bias Neuron
* Momentum
* Threshold
* Maximum Iterations

One is to decide how deep the network is. Could it translate directly from inputs to outputs? Since the given functions are not linear then the answer is no. Instead the network will need one or more hidden layers. Another decision is how wide the network is, or how many neurons to have in each network. Adding more will allow the network to learn better, but it will slow down the process with added computation time. Another choice is whether or not to include a bias neuron for each layer, as this will allow the decision boundary for a given neuron to deviate from the origin. Another is whether or not to include a momentum parameter, and if so then deciding on a value for it. Then there is the value of the Threshold, or at what point the error is sufficiently small as to stop training the network. In a similar vein there's the value of the maximum number of iterations to complete if the threshold value is never met.

Tuning these value will come down to experimentation because it can be hard to predict exactly what effect they will have. I will choose moderate default values to start with for all the networks, and then tweak them to try to get a better performing network.

Network Depth: I have chosen one hidden layer for the one input function, and two hidden layers for the two input function. No particular reason, just seems like a good place to start.

Network Width: I have chosen 25 neurons for the first hidden layer. This seems like a lot compared to the typical examples I’ve seen, and I chose this since the network is implemented in C++ and runs pretty fast, so this number is computed reasonably quickly, and it seems like it should help it learn better.

Bias Neuron:

Momentum:

Threshold:

I chose to include all of these since they seem relatively standard for neural networks and I just decided to keep the values that were used in the sample implementation code I have. Bias +1, momentum -1, and threshold 0.0005f.

Maximum iterations:

Hopefully the network will work so well that this will be unnecessary, but otherwise I will just adjust this to finish after 10 or 20 seconds if it hasn’t met the threshold yet. I’ll start this out at 10000 or so but it’ll be dependent on the actual results I’m seeing.

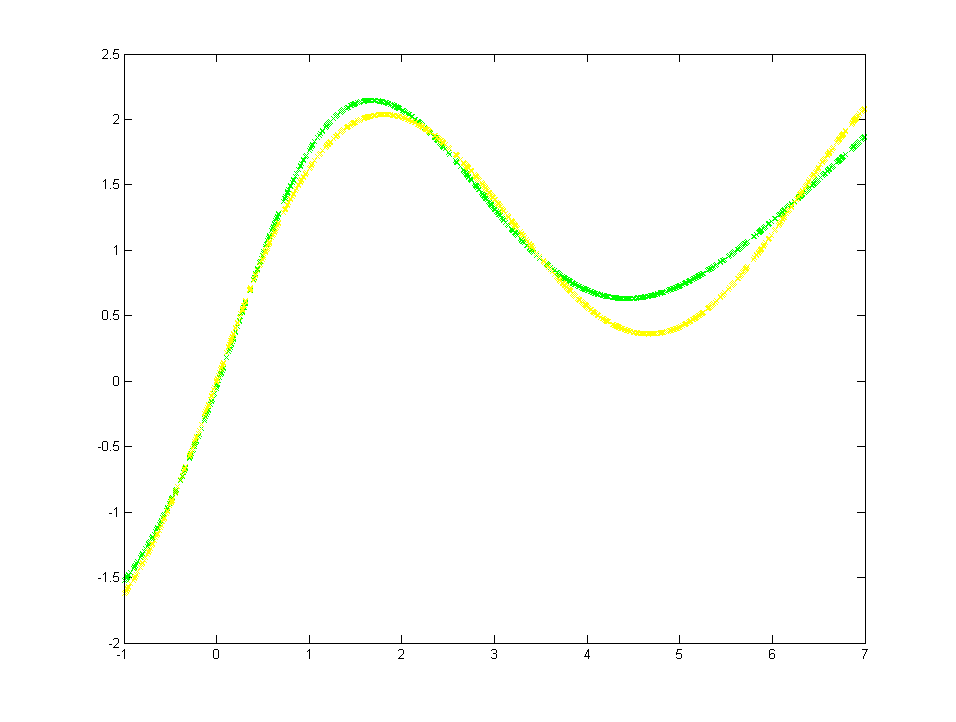
Decisions that were not made until actual implementation include:

* Method of generating the data
* Method of verifying the network is learning
* Method of visualizing the learning network
* Realizing the need to compare the test data to the output data to the ideal data

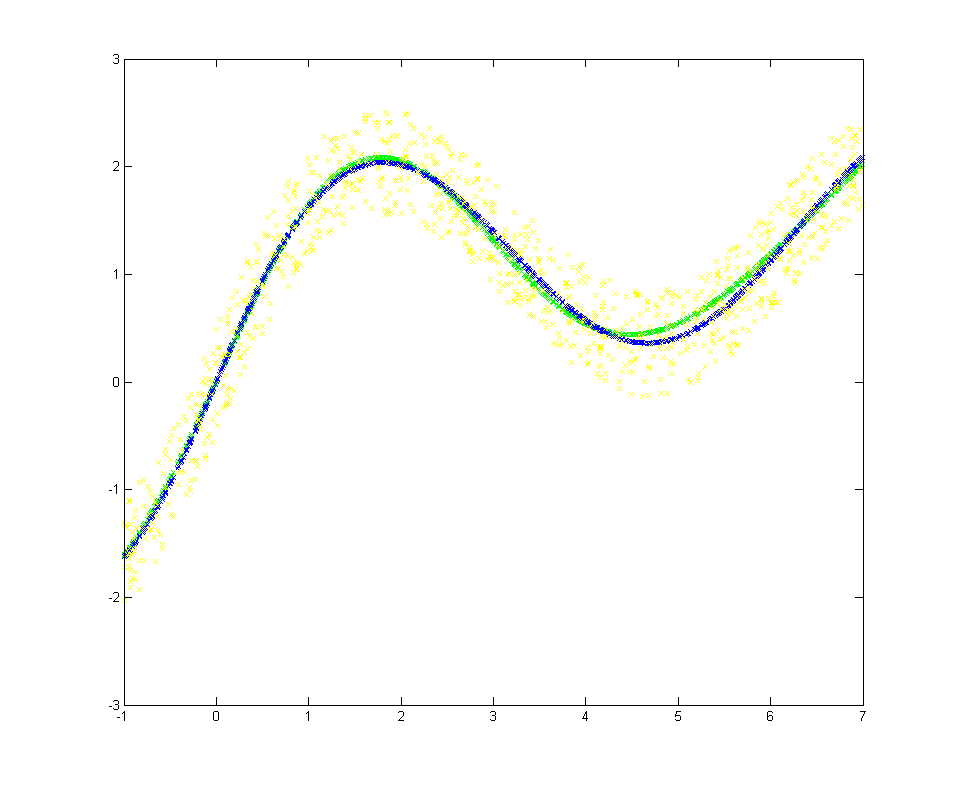
Once I began the implementation I realized I had to have some way of generating the test data. I chose to ignore values outside of the range since they were not defined according to the problem. I chose to use a random number generate to pick random X’s and then compute their Y value according to the function. Then if this was an noisy function then I would add a random value from -.5 to .5.

I then need to be verifying the network was properly learning. I chose to use the Mean Square Error value (or more accurately, the mean of the mse over one run of the test data). On every iteration of the test data I computed the MSE, and if that value was below a threshold value then the network was considered trained.

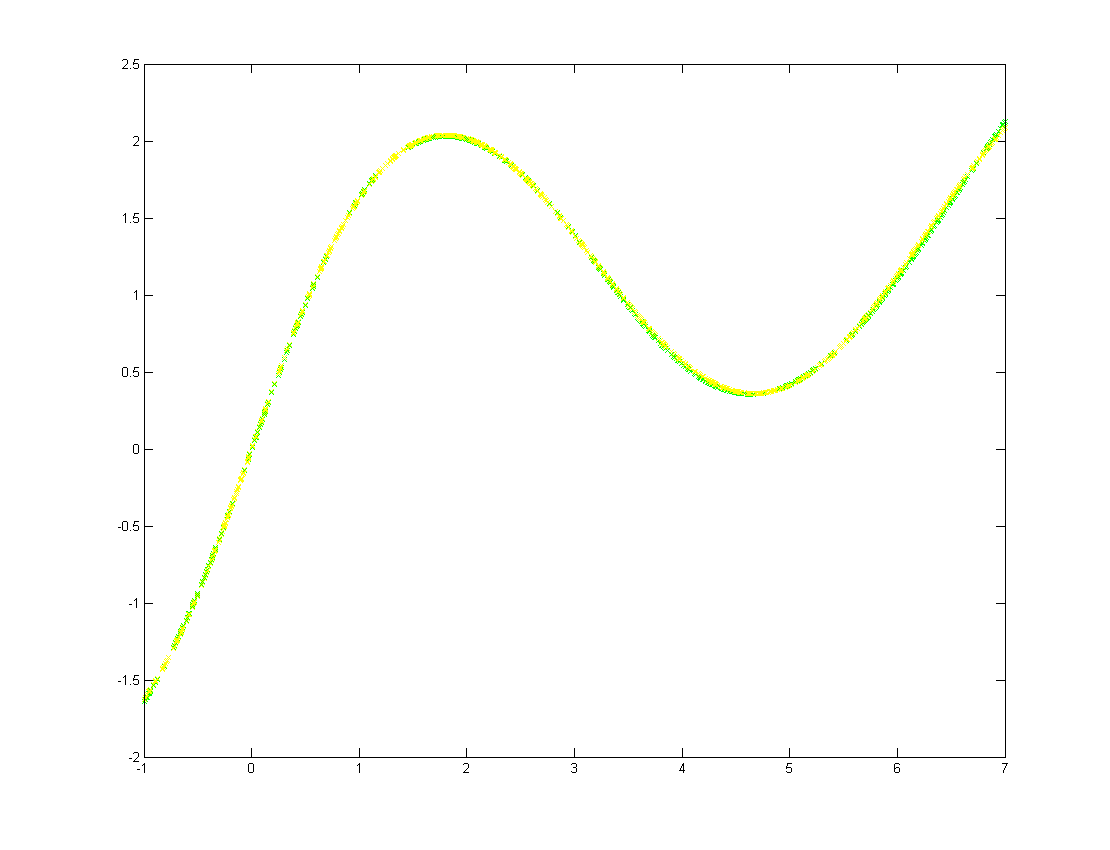
At this point, although I could have a “trained” network, I still didn’t have any idea how close it was to the original function. This was solved by saving out the data out in csv format and using matlab to plot the network output versus the ideal output. Then the issue was I couldn’t see any difference in the noise vs without noise graphs, so I then needed to add a csv of the test data so the noise could be visualized.

Fortunately my initial values seemed to give reasonable results. The main parameter I had to adjust significantly was the threshold value. When watching printouts of the MSE, it would quickly drop and then level off, after which it would drop very slowly. This curved dropoff was different for each function, so I had to tweak it depending on which function was being run. I settled on 0.0001 for function 1 and 0.005 for function two. This seemed to give reasonable results and all the graphs I was seeing looked pretty good. Except there was one weird result.

This graph was for the first function without noise. The yellow is the test data, the blue would be the ideal data, except the yellow has no noise so the blue is covered completely by the yellow, and the green is the network output. At first glance this looks pretty good, in that the output is quite close to the function output. That is until you look at the graph with noise.

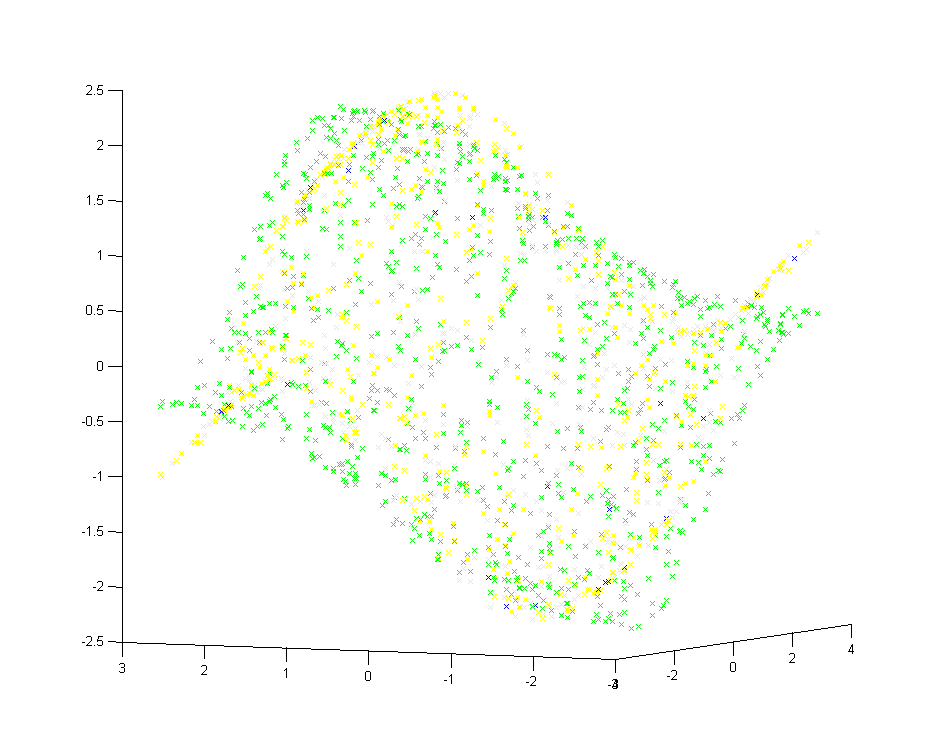


The colors are all the same, and you can see the yellow test data is very noisy. Yet despite the noise, the green network output values are much closer to the blue ideal data. The network is identical, the parameters are the same, and for some reason it was able to train on the noisy data better than the ideal data. I do not understand why this is the case. I went back to the noiseless data and bumped the threshold down to 0.0000001 and this is the resulting graph.

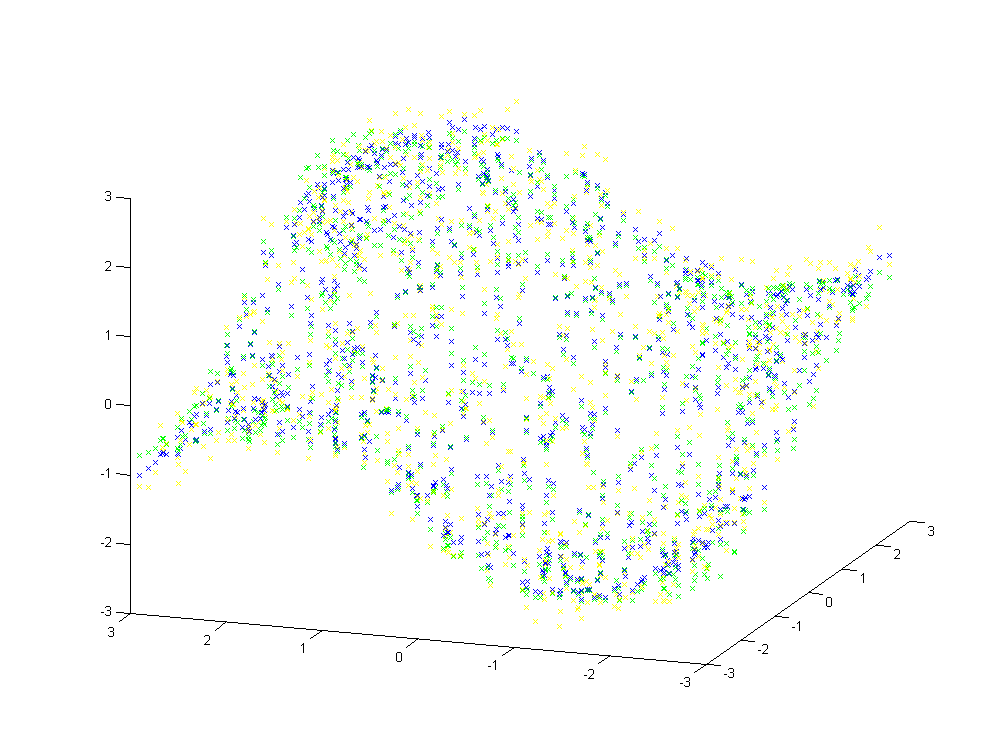


With the much higher precision it became very very close to the ideal yellow line. It isn’t very surprising that running the test data more times allowed it to create a better approximation. What was surprising to me is that the same network was able to train with the noisy data better than ideal data for whatever reason.

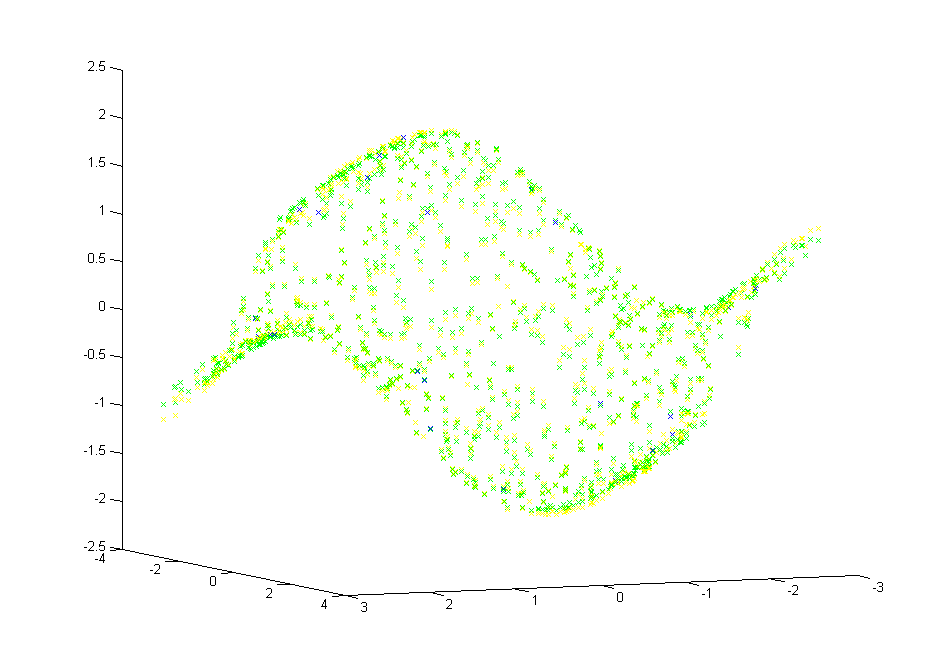
The 2D function gave similar results.



There are very few blue dots, which shows the test data was very nearly identical to the ideal data, but the green dots are not quite lined up with the yellow dots, showing that it did a pretty good job of reaching an approximation of the function, but it’s not quite replacement quality.



And again with noise you can see that the yellow dots have diverged from the blue ideal dots, yet the green values are looking closer to ideal than they were with the noiseless data. I still am not sure how to account for this oddity. So I took the same approach as I did to the first one and changed the threshold to 0.00001. The resulting graph gave results consistent with the first function.



Once again the blue dots have been mostly hidden by the near ideal yellow test data, but the yellow dots themselves are getting close to being covered up by the very close to idea green data. With the tighter threshold value and the larger number of iterations, the network was trained to be very close to the desired function.

Software base on code from http://www.codeproject.com/Articles/13582/Back-propagation-Neural-Net