CMSC 675 Neural Networks, Project 1

Charles Lohr

October 19, 2009

Introduction

I chose to implement Project 1 using C++ and making my generic nodes into C++ Classes. I titled each node as being a "Node." I decided to provide it with the following publicly accessible members:

- std::vector<Node *> **Inputs**; A mapping of all of the nodes this one takes input from. Think of this as this node's synapse list.
- std::vector<float> **InputWeights**; A listing of the weight applied to each node, note this is one bigger than *Inputs*, since the first value is the bias.
- std::vector<vector Node *> **Outputs**; A mapping of all the nodes that take input from this node.
- float **Output**; The output value from this node, as determined by *Function*.
- float **Delta**; The derivative value from this node, as determined by *Function*.
- float **LastSum**; The value that is the unadulterated sum of all the nodes, times their inputs. This is passed into *Function*.
- bool **Fixed**; If set, the node's Output value will not be changed. You can set it's Output value to be whatever you chose. This mimics the input layer of an ANN.
- void (***Function**)(float x, float & val, float & derivative); The function pointer to the function that could be linear or sigmoid.

I also decided to implement it with the following functions:

- **constructor**: Node(void (*fn)(float input, float & val, float & derivative)); It requires the type of function that it is to simulate, i.e. Linear, or Sigmoid.
- **AttachOutput**(Node * n); Attaches node n to *this* node's input. It then informs the other node of the recriprocal.
- **CalculateOutputs**(); Calculate *this* node's outputs from its inputs.

- **Backprop**(float wrongness, float training = 0.05); Use back propagation to help train the network. This is recursive.

In each example my network is constructed in a different way. In both examples, the networks are two-layer. They topographically can be easily manipulated in the main portion of whatever program is being run. The sigmoid function I use in all cases was tanh(x) = (1-e^{-x})/(1+e^{x}). I chose to randomly assign the initial weights between -1 and 1. I did not take any variations on BP learning, since I had plenty of computing time and I did each test did not exceed 2.5 seconds. The output stages were always linear, the hidden stages were always sigmoid.

Classification

For the classification problem, I modified my network to contain two input nodes, the first the X location, the second, the Y. This feeds into a first hidden layer with **seven** nodes. Then, it feeds into a second hidden layer with **two** nodes. All of the nodes in my network (Except for the output nodes) use a sigmoid. I chose to use a linear output for my output so I could closely observe what was going on.

My results were fairly poor, while my network coincidentally produced good output the first time, and has very high training accuracy... The test accuracy is lucky at best. The following network was trained in 500 epochs with a learning rate of 0.02; For reference, the seed used was 93.

Coordinate	Actual Network Output	Expected Network Output
0.70,0.70	-1.00	-1.00
1.10,1.10	1.00	1.00
0.50,-0.50	-0.99	-1.00
0.90,-0.90	1.00	1.00
-0.40,0.40	-1.00	-1.00
-1.20,1.20	1.00	1.00
-0.30,-0.30	-1.01	-1.00
-1.50,-1.50	1.00	1.00
0.80,0.00	-1.25	-1.00
0.10,-0.70	-0.71	-1.00
-1.00,0.30	0.17	1.00
1.50,1.00	1.60	1.00
MSE (Actual)	29.823%	
MSE (Learning)	0.004%	

Many other seeds produced data that in fact crossed the threshold and produced the incorrect data altogether. This leads me to believe the correct answers are more on luck - in particular the third answer. None of my neural network setups were able to definitively describe where the third element should go.

I believe a major problem with this isn't so much overlearning as much as it is overgeneralization. My neural net could not learn to be as general as it was being asked to be.

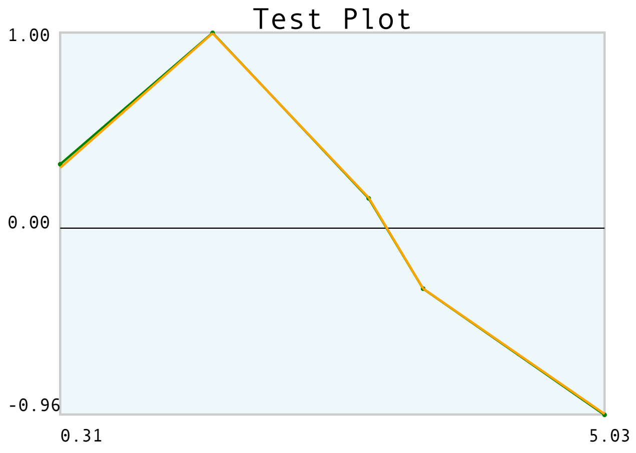
The code to generate this output is found in **classification.cpp**.

Function Approximation

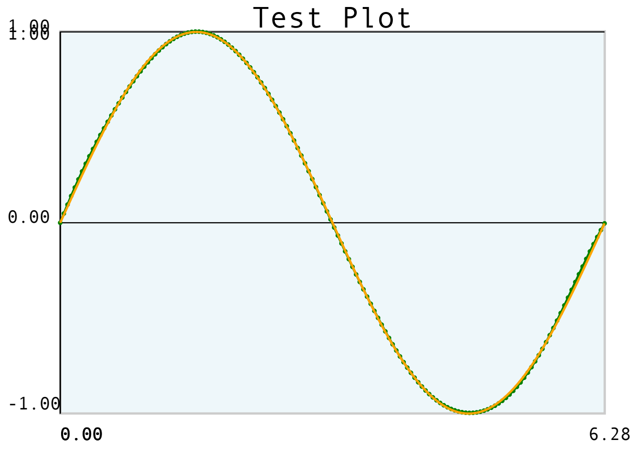
For the second portion of the project, I decided to use a slightly larger neural network. I found that while my neural network could approximate sign with fewer function, using a network with **nine** first-hidden-layer nodes and **two** second-hidden-layer nodes could quickly learn how to approximate *sin* very quickly. My learning rate was 0.04, I performed 30,000 epochs. It took only two seconds to program my entire network. For reference purposes, it uses a random seed of 95.

Input	Actual Network Output	Expected Network Output
0.31	0.33	0.31
1.63	1.00	1.00
2.98	0.15	0.16
3.46	-0.31	-0.31
5.03	-0.96	-0.95
MSE (Actual)	0.008%	
MSE (Learning)	0.001%	

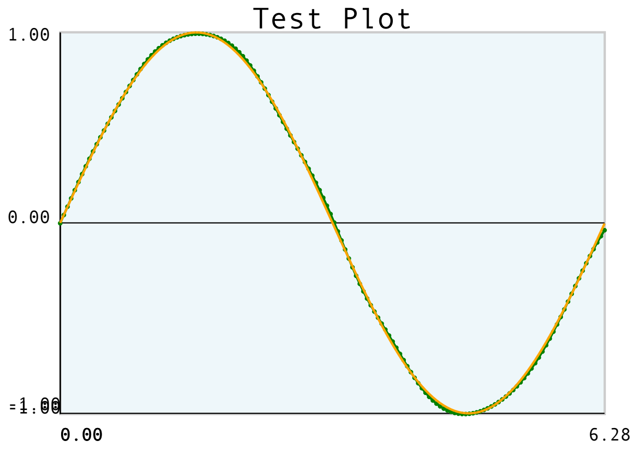
You can see below a graph of the actual output.



I think a large problem with this network as well was it's overtraining. When I decided to see how close it could get with the range of inputs over the interval of 2π, it got still performed well. The MSE was 0.006% even with only a few training points.



When I trained the same network using points randomly assigned to the number line, with 300,000 samples, strangely enough the MSE actually increased. The MSE was 0.011%. The graph is below.



The output of the first part of this program was generated by **sin.cpp**.