Skylar Wurster
Neural Networks
Professor Morris
February 15th, 2018

Lab 1 Report

Summary: A matrix based approach to the network was taken, where each web of weights or biases formed a matrix to be used for computation. This results in a cleaner implementation with less for loops than the iterative coding solution, and also supports batch updating with multiple inputs at once. For example, instead of a single input of `double[4][1]`, we can have `double[4][n]` with n inputs. Similarly, double[n][1] expected results.

Inside of the `NeuralNets.pde` file holds the main method (setup), where the controls for the GUI are located. A number of inputs, outputs, hidden layers, and nodes per hidden layer must be specified before clicking initialize. To choose nodes per layer, separate desired sizes with a space. For instance, if your goal is for two hidden layers of size 4 and 3, set the "number of hidden layers" field to "2", and the "nodes per hidden layer" field to "4 3". If no hidden layers are desired, set the "hidden layers" field to "0", and the "nodes per hidden layer" field can be ignored.

Since this network is hard-coded to call the functions that generate 4 inputs and 1 output (see `project1InputSeeded(int)` and `project1Output(double[])`), your input field must say 4 and the output must be 1. Feel free to play with the number of hidden nodes and layers. I've been unable to get a hidden layer with size 2 to converge, but any single layer with >2 layers converges. Support for multiple hidden layers is implemented and works and successfully converged with hidden layers of size 4 and 3 (see Figure 4).

The GUI shows connections, weights, biases, inputs, and desired outputs. The speed of training can be adjusted with the slider on the bottom part of the screen. The integer it is set to is the time in milliseconds between each update. Increase the delay to see the network's current cost, input, and output. Once "Begin" is clicked, the initialized network will train until all errors for an epoch are less than 0.05.

`Networks.pde` holds the neural network information. Here, you'll find the SigmoidNetwork being used, and the MPNetwork used for homework 1. Both extend from the parent class Network.

The network must first be initialized through `Initialize()`. This is used to create the network architecture and randomize weights and biases between -1 and 1. From here, the main method will create a thread to train the network using the `Train(double[], double[])` function. This takes the input and desired output and performs weight updates based on feedforward and backpropagation.

Inside of `Train(double[], double[])` are a number of fields that store important information. Since backpropagation relies on the values of the nodes before activation (and before an update), the values passed in to the nodes, $v_{k,j}$ in the notes, where k is the layer and j is the node, need to be saved. Similarly, $y_{k,j}$ - the output of the node, is also saved. The pre-activation values are used in `populateDeltas(Matrix actual, Matrix, ArrayList<Matrix>)` since $\varphi'(v_{k,j})$ needs to be calculated. The activated values are used

when calculating $\frac{\partial E}{\partial W}$, because $y_{k,j}$ is multiplied with $\delta_{k,j}$ (see `costFunctionPrime(ArrayList<Matrix>, ArrayList<Matrix>)`).

Learning rate is hard coded as a public variable in the Network class, and is where it was adjusted for tests seen in the Appendix.

Testing went well, but would have been more enlightening if I used seeded random variables for my weights. Because of the randomness, it's difficult to say where convergence would happen the quickest. Following trend lines from Figure 3, it appears to show that larger learning rates cause quicker convergence in general. However, from the data that I've gathered, there is no statistical significance from using momentum or not.

I had expected convergence to be quicker, but began to notice a pattern while training. Inputs 0000 through 1110 would have decreasing errors, while 1111 would increase. The cost function for every input except 1111 would be <0.05, while 1111 would be above 0.3. This would happen after some number of epochs, and then the cost for 1111 would slowly begin declining until convergence. This happens with each learning rate chosen.

I've also observed that the network converges faster with a larger hidden layer. With a hidden layer of 16 nodes, my network took 3183, 1695, 2908, 8941, and 2686 epochs with the first 5 trials and a learning rate of 0.35. On average, this took less that 5 seconds each (and was more fun to watch).

After some point, the network takes too long to converge. With 64 nodes in my hidden layer, the cost remained at 0.499999 for most inputs. A similar result happens with deeper networks; expanding to 2 or 3 hidden layers makes the cost sit at around 0.125.

Appendix

η	Number of Epochs
0.05	1102527
0.1	10925
0.15	7204
0.2	43385
0.25	133532
0.3	120385
0.35	24681
0.4	108960
0.45	24244
0.5	9685

Figure 1: learning rate affecting convergence without momentum, data sampled on Feb 11, 2018

η	Number of Epochs
0.05	601854
0.1	552704
0.15	119068
0.2	21634
0.25	16688
0.3	51789
0.35	1651
0.4	104500
0.45	27634
0.5	67451

Figure 2: learning rate affecting convergence with momentum and $\alpha = 0.9$, data sampled on Feb 11, 2018

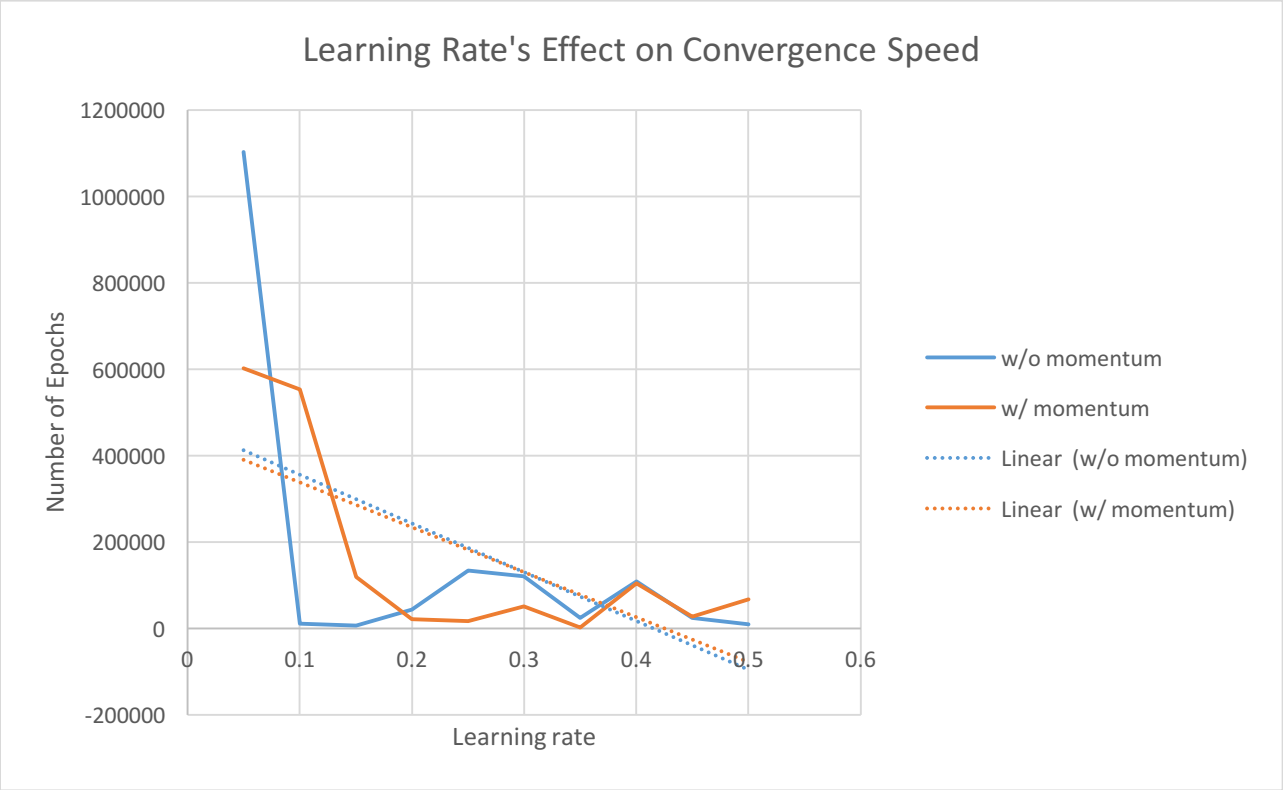

Figure 3: a graph showing how learning rate and momentum affects convergence speed

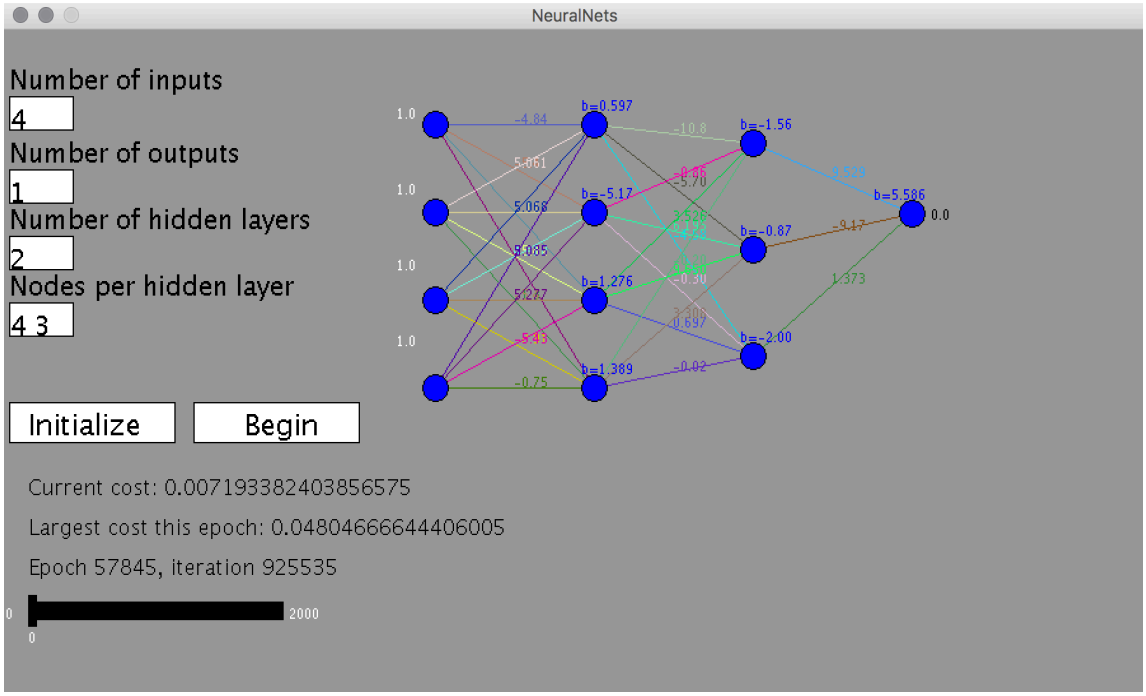

Figure 4: Example of the NeuralNets applet GUI after a deep network was trained on the project function. This converged after 57845 epochs.