CS420 Project IV
Experimentation with Artificial Neural Network Architectures

Alexander Saites
3/26/2012
Introduction
In this project, I implemented an artificial neural network that allowed the specification of multiple hidden layers and the number of neurons within each layer. I then tested the implementation with two problems, listed below. The network architecture was varied (both the number of layers and the number of neurons within those layers) and the performance of the network was recorded using root mean square error. The performance of the network was analyzed to determine the optimal network architecture for the problem.

Problems
The following two problems were used to experiment with the network. Note that the training set, testing set, and validation set came from Kristy Vanhorn
(http://web.eecs.utk.edu/~kvanhorn/cs594_bio/project4/backprop.html).

Problem 1
\[ f(x, y) = \frac{1 + \sin \left( \frac{\pi x}{2} \right) \cos \left( \frac{\pi y}{2} \right)}{2}, x \in (-2,2), y \in (-2,2) \]

Problem 2
\[ f(x, y, z) = \left( \frac{3}{13} \right) \left[ \frac{x^2}{2} + \frac{y^2}{2} + \frac{z^2}{4} \right], x \in (-2,2), y \in (-2,2), z \in (-2,2) \]

Graphs and Analysis
To determine the optimal network architecture, I decided to limit my networks to at most three hidden layers, each with at most ten neurons. For this, I set the learning parameter to .1, as I experimentally found this value to be effective in reducing the RMSE.

One way to find the optimal architecture from this space would be to iterate through each all the possible values for neurons, performing a certain number of trials for each set, averaging the RMSE over those trials for that set, and finding the set for which the RMSE is smallest. Unfortunately, there are 1,000 sets, and given the amount of time it takes to test a single set, performing the state search is impractical and out of scope for this project. Further, this method ignores the role of the learning parameter eta.

Despite these challenges, out of curiosity I did perform this state search for two hidden layers (each with 1 to 10 neurons) for the second problem. The resulting surface plot shows that for this space, with a learning parameter of .1, the RMSE decreases for networks with a larger number of neurons. The surface moves downward toward an architecture with 10 neurons in both hidden layers, suggesting that an optimal architecture may require even more neurons.
The surface also shows that the network did a bit better when the first layer had more neurons than the second. This “bottleneck” seems beneficial for the network to extract features from the input.

Additionally, I graphed the RMSE for the testing data after each epoch. For the most part, the results are uninteresting. In most cases, the RMSE steadily declines. In some cases, the error levels off for a bit, then steps or otherwise quickly decreases. In some cases, the error eventually began increasing (probably the result of overlearning). The following graph shows the RMSE for testing data plotted against epochs for the problem 2, using two hidden layers, each with five neurons. Note how noisy the error is and how quickly it decreases.
Figure 2: RMSE vs epoch for problem 2, using two hidden layers
Since it was not prohibitive to vary 1 to 10 neurons for a single hidden layer, I performed this experiment for both problems. For the first problem, the results show that if only using one hidden layer, using four neurons seems to be enough to produce reasonable results. Note that these experiments were performed with an eta of .1 for 3,500 epochs, five trials each averaged together.

![Figure 3: RMSE vs neurons for problem 1](image)

I performed the same experiment with the second problem, and found that if using only one hidden layer, ok results are obtained with four neurons, but the error steadily declines until nine neurons. The small increase at ten neurons may simply be noise.
Since searching this state space directly was prohibitive, I instead selected various values which I believed may produce strong results, tested them, and analyzed those results to decide which values to try next. Based on the above experimentation, I chose sets with more neurons in earlier layers than later, and used at least five neurons in each layer. After selecting several sets which showed strong results, I experimented with changing the learning parameter for the same network architecture.

This graph shows problem 1 with an architecture of [2 9 7 1]. After 5 trials (each with 5000 epochs), the RMSE dropped to 0.0295.
Figure 5: RMSE vs epoch for problem 1

Adding a third layer (with 5 neurons) results in an increase in the RMSE.
It could be that the network just needs longer to train, so I increased the training time to 10,000 epochs.

![Figure 6: Problem 1 for 10,000 epochs](image)

Although these results are better, they are not as good as the two-layer network. For this problem, a two layers design is probably best.

Next, I experimented with the value of eta. I used the design with two hidden layers containing nine and seven neurons, respectively. First, an eta of .01:
With this smaller eta, there is much less noise, but the overall RMSE is a bit higher than before. Next, I tried a larger eta of .3:
Unsurprisingly, the results are super noisy. The larger eta value causes “corrections” to constantly overshoot the target values. The value .1 I had been using seems a good balance of correcting quickly without overshooting too much.

Next, I attempted to find a good architecture for Problem 2. Since I have seen all possibilities for two hidden layers with fewer than 10 neurons, I decided to experiment with three hidden layers. First, I tried [3 10 9 8 1].
These results were not especially impressive, so I decided to try [3 10 10 10 1].

Figure 9: Problem 2 with three hidden layers

Figure 10: Terrible
Using 10 neurons in all three layers turned out terribly, so I decided to go back to a proper bottleneck.

This improved the results, but still are not as good as the two-layer solution. As a result, I decided to go back to the two-layer solution with a greater number of neurons.
These results were far more promising, so I began experimenting with eta.

As before, the smaller eta significantly reduces noise, but takes longer to decrease the error. As a result, I suspected increasing eta to .3 would result in a very noisy picture:
As expected, the result is just pure noise.

These results show that for the most part, the problems are not extremely sensitive to the network architecture. Given enough time, it seems most reasonably sized networks will drop the error. The surface plot for Problem 2 best shows this fact. They are sensitive to changes in eta, but that is also no surprise: too large results in overcorrecting and too small results in lengthy convergence times.

One final important note is that bottlenecks do seem to really help the network, but they do not seem to benefit much from a large number of layers.

**Code**

I wrote my neural network in Java, but based it off of a neural network implementation I had written in C last semester. The C implementation did not allow multiple hidden layers, so modifications were made to the Java program to allow this feature. I then wrote a driver program in Java to accept command line arguments for the total number of layers; the learning parameter eta; the training, testing, and validation files; and the network architecture itself. To facilitate easier analysis of the network, I wrote a Matlab wrapper around the Java driver and used it to experiment and graph data. Although the Matlab wrapper could be written around the Java ANN, I wrote it around the Java driver so I could more easily use Java’s IO and error checking features. The code (including the Matlab wrapper) is included in the jar file submitted with this project.

To use the Java driver, you must specify the total number of layers (including the input and output layers), a learning parameter (eta), the file containing the training data, the file containing the testing data, and the file containing the validation data, followed by the network architecture. The architecture should be specified as the number of neurons for each layer in the network (starting with the input layer). Thus, the program can be instantiated in the following way:

```java
java NetDriver 3 .1 prob1.txt prob1Test.txt prob1Valid.txt 2 5 1
```

That command would run NetDriver on problem 1 with a network with a learning parameter of .1, 2 input neurons, 5 hidden neurons in the first layer, and 1 output neuron.

The files containing numerical data for the network to train and test on should contain inputs followed by outputs, separated by spaces, each trial on its own line. Although the program should successfully skip blank line, you may run into trouble if you have an extra new lines, so it is best to avoid them.

The Matlab driver can also be used to run the Java driver. To do so, first add the directory containing the class files to your class path (use `javaaddpath('path')`), then import `NetDriver`. To use the driver, create a java String array for the arguments, then create a new `NetDriver` object using

```java
nd = NetDriver(args,false);
```

The second parameter determines whether or not `NetDriver` prints the RMSE for each test after each epoch.
You can get access to the RMSE for the validation by using `nd.validationRMSE`. Additionally, the RMSE values after during each test are saved in `nd.rmseValues`. Since it is an array of doubles, you can plot it directly in Matlab.