Implementation of a Library for Artificial Neural Networks in C

Jack Breese
TJHSST
Computer Systems Lab 2007-2008

April 7, 2008

1 Abstract

In modern computing, there are several approaches to pattern and object recognition and classification. As computational power has increased, artificial neural networks have become ever more popular and prevalent in this regard. This project intends to implement a library for the creation of general-purpose neural networks in C, and to initialize and train several neural networks on sample data sets including shape classification and optical character recognition.

2 Background

2.1 Defining Neural Networks

An artificial neural network is a computational model which emulates the biological structure of the brain. It is a system of interconnected virtual neurons. In a perceptron, the type of network which is being implemented in this project, the network is organized into a hierarchy of layers. The first layer, or input layer, has one neuron for each input value. Each of these neurons is connected to every neuron in the next layer, the hidden layer. The neurons in this hidden layer are, in turn, connected to each of the neurons in the final, or output layer. The vast computational utility of these

artificial neural networks stems from the adaptability of the system. After initialization, the network can be trained. As these networks are capable of modifying their connections, adapting their responses based on their training, they become increasingly more accurate as they are trained. This modeling of biological networks has been used in an ever-increasing number of fields.

2.2 Uses of Neural Networks

Neural networks have found use in a large number of computational disciplines. In the field of object and pattern recognition, neural networks are commonly used to perform such tasks as optical character recognition and face recognition. Pattern analysis is another field in which neural networks excel. Bayesian neural networks, for example, have become quite commonplace in filtering for e-mail spam, growing more effective every time a message is marked as such. In statistics, neural networks can be used to approximate functions and to interpolate missing data.

2.3 Mathematics and Training of Neural Networks

Perceptron neural networks are broken up into a hierarchy of layers. The first layer, or the input layer, is relatively simple. It contains one neuron for each input value the network is to have. Each of these neurons is then connected to every neuron of the second layer. This second, or hidden, layer allows for the majority of the computational power of these networks. Each neuron is connected to every neuron in the previous layer. These connections are given a weight. The hidden layer neurons take the sum of the values of the connected neurons, multiplied by their respective weights. This sum is then fed into the activation function, in most cases the sigmoid function $f(x) = \frac{1}{1+e^{-x}}$. The value of the neuron is then the result of this function. These values are passed along another series of weighted connections to the network's output layer. The output layer contains one neuron for each condition the network is checking for. These neurons function like those of the hidden layer, and the result of the sigmoid function yields the probability that this condition has been satisfied.

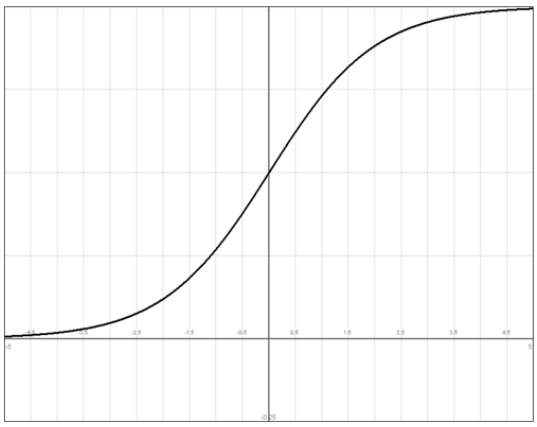

Figure 1: The sigmoid function.

$$sgm(x) = \frac{1}{1 + e^{-x}} \tag{1}$$

$$Value = sgm(\sum (NeuronValue) * (ConnectionWeight)) \tag{2}$$

In training neural networks, a large set of input data is assembled. The network is initialized with random weights at first, and the data is then fed into the network. As each data is tested, the result is checked. The square of the difference between the expected and actual result is calculated, and this data is used to adjust the weights of each connection accordingly. The accuracy of neural networks is mostly a function of the size of their training set rather than their complexity, as shown by a 1964 study. A minimum number of neurons, connections, and layers is required for a perceptron to begin modeling accurately. In this study, the number of neurons in the hidden layer of a perceptron was reduced by $\frac{7}{8}$, yet the network still achieved 80% accuracy after being subjected to the same training. It is interesting to note that, in the implementation of neural networks, the output complexity can have a direct impact on the required training time. In a 2001 study, it was found that, in neural networks for optical character recognition, an array of 26 networks of one output each could achieve the same accuracy as a single network with 26 outputs, but with vastly reduced training time. The implications of this study suggest that it may be advantageous to break

the pattern classification task into smaller subunits in order to decrease the training time required.

3 Project Development

This project is coded entirely in C. Currently, it consists of a framework which uses the libjpeg library to read in jpeg image data, and a framework for the creation of neural nets. My code has methods to initialize double-layer perceptron networks of arbitrary size, limited only by the quantity of allocatable memory. Methods exist to calculate the sigmoid function, take the sum of the weighted connections, and to randomly initialize the weights as a precursor to training. The library is also capable of loading and saving networks using its own file format, without any loss of precision or data.

3.1 Data Structures

In order to implement a neural network effectively, my program makes extensive use of the C programming language's structs. As shown in the code example, there is a neuron struct, which holds the value of each neuron, and a pointer to an array of connection structs. Each connection struct holds a float value for its weight, and a pointer to the neuron from which it originates.

```
typedef struct _connection {
     float weight;
     struct _neuron * from;
} connection;
typedef struct _neuron {
     float d;
     connection * cons;
}neuron;
neuron* mkneuron(int c) {
     neuron* n = malloc(sizeof(neuron));
     n->d = 0;
     connection * a = malloc(c*sizeof(connection));
     n->cons = a;
     return n;
}
```

3.2 Neural Network Initialization

The use of structs in my program greatly simplifies the network initialization process. The initNet method creates an array of pointers to neuron structs of the specified size, and then returns it. Though the current iteration of the program requires the connections to be set up between layers by the method calling the initNet function, initNet will eventually be restructured to recursively initialize the neural network with random weights, preparing it for training.

```
neuron * initNet(int size, int csize) {
     if(size > MAX_NET_SIZE){
          fprintf(stderr, "Error: Exceeded maximum net size");
          exit(1);
     }
     neuron * a = malloc(size*sizeof(neuron));
     int i = 0;
     for(i; i < size; i++)
     {
          a[i] = *mkneuron(csize);
     }
     return a;
}
```

3.3 File Format

In order to accurately save a network, my program uses its own file format. It saves the sizes of each of the layers of the network, and then iterates over the weights of each of the connections of every neuron, one layer at a time. By foregoing markers in the output file, and instead relying on the loading method iterating in the same way, it was possible to drastically cut the required file size and to greatly reduce the time spent on file I/O operations.

3.4 Running Behavior

Currently, my program initializes an array of pointers to an input layer, an array of pointers to a hidden layer, and an array of pointers to an output layer. It then iterates over these arrays, creating the connections between these

neurons and assigning them randomly generated or user-specified weights. After initializing these connections, it then feeds in testing data to the input layer (at this point, all zeros or ones), and iterates over each neuron in the network, summing the values of the weighted connection and calculating the value of the sigmoid function to pass on to the next layer. It then saves the network to a specified file, so that it can be reinitialized in its current state in the future.

3.5 Results of Testing: Problems and Solutions

In the development of my neural network framework, I encountered significant delays due to unexpected segmentation faults. Initially, as I went to calculate the value of my output, my program would add the weighted connections of the first few neurons, and then crash with a segmentation fault. Further analysis revealed that my program was not successfully iterating through the array of connection structs. It was only iterating through the first few, and then receiving a seemingly random memory address. Further analysis of my code revealed that this was a simple error on my part, as I had not correctly allocated the memory for this array. After rewriting a single line of code to correctly allocate a region of memory for this array, my program began to function as expected.

```
//Incorrect Code.
neuron* mkneuron(int c) {
     neuron* n = malloc(sizeof(neuron));
     n->d = 0;
     a = *connection[c];
     n->cons = a;
     return n;
}

//Correct Code.
neuron* mkneuron(int c) {
     neuron* n = malloc(sizeof(neuron));
     n->d = 0;
     connection * a = malloc(c*sizeof(connection));
     n->cons = a;
     return n;
```

```
}
```

3.6 Planned Expansion

Now that code exists to save and load the neural network in its current state, the training process can begin. First, code will be written to adjust the weights based on the accuracy of the network in response to an input data set. Then, a series of small data sets will be created in order to test the network on a variety of different recognition and classification tasks. Presently, several different data sets are planned, ranging from simple shapes and optical character recognition, to more complicated face recognition tasks.

4 Results

Currently, this project provides an example framework to demonstrate the structure of neural nets. Though it is successful in that regard, it has yet to perform actual object recognition or classification. Now that the basic functions exist to create, manage, and use neural networks, it is only necessary to implement training. Once training has been successfully implemented, and the library has been tested on a variety of different data sets, it will be packaged as a set of C header files for use in other programs, and released under a permissive open-source license.

References

[1] Philippe Crochat and Daniel Franklin. Back-propagation neural network tutorial, April 2003.

[2] Herman H Goldstine. Computers and perception. *Proceedings of the American Philosophical Society*, 108:282–290, 1964.

[3] Amit Gupta and Monica S Lam. Estimating missing values using neural networks. *The Journal of the Operational Research Society*, 47:229–238, February 1996.

[4] J. J Hopfield. Learning algorithms and probability distributions in feed-forward and feed-back networks. *Proceedings of the National Academy of Sciences of the United States of America*, 84:8429–8433, December 1987.

[5] John J Hopfield and David W Tank. Computing with neural circuits: A model. *Science*, 233:625–633, 1986.

[6] B. D Ripley. Neural networks and related methods for classification. *Journal of the Royal Statistical Society. Series B (Methodological)*, 56:409–456, 1994.

[7] Matee Serearuno and Tony Holden. A comparison in training time of the single and multiple-output mlp neural networks. *SAC*, 0:32–35, January 2001.

[8] Brad Warner and Manavendra Misra. Understanding neural networks as statistical tools. *The American Statistician*, 50:284–293, November 1996.