Implementation of a Library for Artificial Neural Networks in C

Jack Breese
TJHSST
Computer Systems Lab 2007-2008

June 10, 2008

1 Abstract

In modern computing, there are several approaches to pattern recognition and object classification. As computational power has increased, artificial neural networks have become ever more popular and prevalent in this regard. Over the course of the year, I implemented a general-purpose library for artificial neural networks in the C programming language. It includes a variety of functions and data structures for use in creating, working with, and training simple perceptron-type neural networks using a backpropagation heuristic. Though it is not fully capable of training a neural network in its current iteration, this library would serve as a useful starting point for further exploration in the field of machine learning.

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 Neural Network Structure

Figure 1 is a diagram of the basic layout of a two-layer perceptron neural network. In a task such as face recognition, a neuron is created in the input layer for each discrete pixel value in the input image. The hidden layer is a layer of neurons used only in calculation. Each neuron adds up the values in the preceding layer, multiplied by the weight of each connection in the weight matrix. This process is then repeated for the output layer, where, after being processed by the sigmoid function, the output will be a value between 0 and 1, with 1 corresponding to 100% certainty that the input image matches the condition for that specific output neuron

2.4 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.

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

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

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

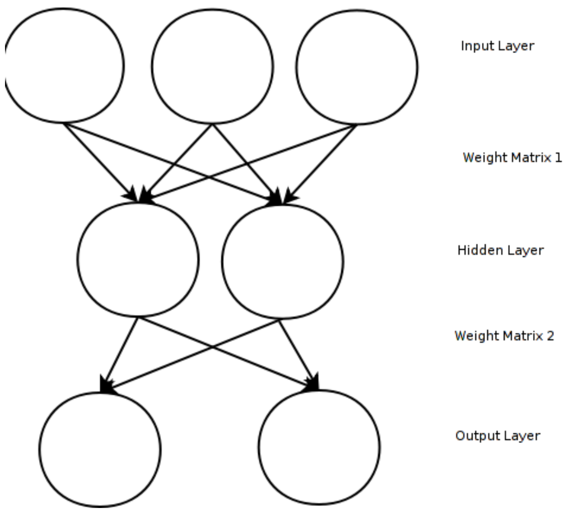

Figure 1: A sample neural network.

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. It has been packaged as a set of header files so that other programs can make use of its methods and data structures. A full method listing is available in appendix A. It includes methods to initialize double-layer perceptron networks of arbitrary size, limited only by the quantity of allocatable memory. It can also calculate the sigmoid activation function and propagate values through the network. In additon, methods exist to adjust

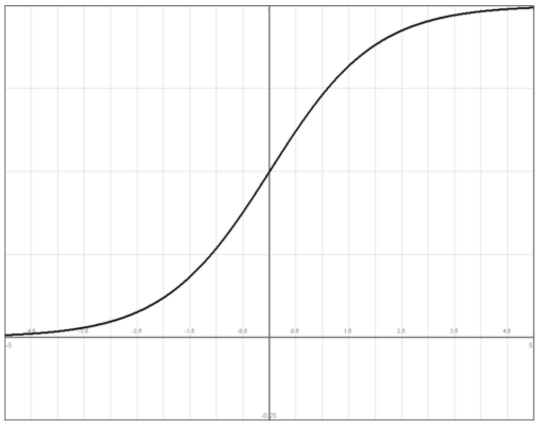

Figure 2: The sigmoid function.

weights in training based on a classical backpropagation algorithm, as well as read in pgm formatted image data to the network. The library also includes several data structures for keeping track of the network and training information, as well as methods to save and load neural network weights and its own file format.

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;
}
```

The library also includes structures which contain PGM formatted image data and attributes for use in training, as well as a structure which keeps track of all available training data. The structures of my network are summarized in appendix A.

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. After the memory is allocated and the structure set up, the program then calls a separate method in order to populate the network with random weights and begin propagating image data through it. A pseudocode implementation of this initialization follows.

```
Function initNet:

     For each layer in the net:

          Allocate enough memory for each neuron in the layer

          For each neuron in the layer:
               Connect to each neuron in the previous layer, if any.

               For each connection to the previous layer:
                    Set a random weight.
```

3.3 Network Algorithms

The following are pseudocode versions of other algorithms used in my library, in order to better illustrate the functions of the network.

```
Calculating Neuron Values

     For each neuron in the previous layer:
          Sum += neuron_weight*neuron_value
     neuron_value = activation_function(sum)

Training

     For each neuron in the network:
          For each connection to the neuron:
               weight = random_value()
     Until desired accuracy is reached:
```

```
For each example in the training data:
     actual_out = run_network(example)
     exp_out = calculate_expected(example)
     error = exp_out  actual out
For each neuron in the network:
     calculate_delta_weights(error)
     adjust_weights(neuron)
```

3.4 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.

3.5 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. Once saved, it then reads in image data, feeds it into the network, and propagates it through. It then waits for user input, in the form of a list of filenames for use in training.

3.6 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;
}
```

3.7 Training Issues

Though my library has been successfully implemented, I was unable to create a testing program which trained it to recognize faces. I created and sanatized a training data set based on one released by the Carnegie Mellon University CS department, but encountered problems with null termination, and was unable to create an array of filenames which correctly corresponded with expected network output values. The library is, however, functional in every other regard.

4 Results

This library currently serves as a full implementation of all methods necessary for the creation, management, and training of arbitrarily sized two-layer perceptron neural networks. Though it includes the methods and data structures needed in this regard, is has not yet been used to successfully train a neural network, due to the limitations of the C programming language mentioned earlier. Though actually training the network is an exercise left up to the end-user, this library still serves as an excellent starting point for future exploration in the field of neural networks and machine learning. It has been packaged as a set of C header files and released under a permissive open-source license, so that it may prove useful to others working in a similar field.

Appendix A

Methods

```
neuron* mkNeuron(int c)
```

This method allocates memory for a neuron. C is the number of neurons in the previous layer that will connect to it.

```
Neuron* initLayer(int size, int plsize)
```

This method instantiates a layer of size size, with a weight matrix stored for the layer before it, of size plsize.

```
void calcValue(neuron* n, int s)
```

This method calculates the value for a neuron based on the weight matrix it stores and the values of the previous layer. It should only be called by the calcNet method, never on its own.

```
void calcNet(neuron* i, neuron* h, neuron* o)
```

This method propagates the calculations through the entire neural network, calling the calcValue method on each neuron.

```
void readTrain(char* filename, trainInfo * tin, char* zval)
```

This method reads training data from a structured file, filename, and reads it into tin. Zval is used in calculating the expected outputs for a given training data, and is the type of input file that should produce an output of zero.

```
void readPGM(char* filename, pImg* pgm)
```

This method reads in any valid pgm image, filename, to a pImg struct (which must first be allocated.) It is used in reading in image data to train and test the neural network.

```
void backProp(neuron* i, neuron* h, neuron* o, double r, double e)
```

This method performs backpropagation in order to adjust the weights on the neural network during the training process. R is the rate at which training should be performed, and e is the error from what is expected.

void saveWeights(neuron* i, neuron* h, neuron* o)

This method saves the weight matrices of each layer in the
neural network, so that it can be loaded and used again after training.

double act(double x)

This method is the sigmoid activation function pictured in the report.

Data Structures

connection

This struct holds a double weight, and a neuron pointer.
Each neuron holds an array of connections pointing to the
layer preceding it, in order to store the weight matrix.
Neuron

This struct holds a double, d, which stores the neurons
value,a double err, which stores the calculated error for
training use, and an array of connections.

pImg

This struct stores a valid pgm formatted image file, including
all attributes, as well as raw ASCII formatted image data.

TrainInfo

This struct stores an array of training image files, as well
as an array of doubles which holds the expected values for
each input file.

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.