Abstract

This code is put forward as an inspirational example of a basic artificial neural network (ANN). Specifically, the “multilayer perceptron”. It has a solely feedforward architecture, and is trained via backpropagation. This code demonstrates advantages and disadvantages with thoroughly commented MATLAB code. It may be useful for research, though results are generally unpredictable and do not always converge. (An inherent drawback to these networks.)

The code was written as required for the course EEL6819 - Artificial Neural Networks at Florida Atlantic University, taught by Dr. P.S. Neelakanta. It uses none of the built-in MATLAB NN-Toolbox functions, and therefore has many editable parameters.

If you’re trying to write your own program, this may be a good read for ideas. I don’t guarantee my code in any way, other than its awesomeness.

1 Introduction

Documentation on artificial neural networks is extensive. Personally, I found the first few chapters of Haykin’s Neural Networks and Learning Machines to be sufficient to complete this project. There may be more complete texts; it’s up to the reader to choose.

Basically, this paper is not an introduction the concept of perceptrons, it’s an example of their implementation. I’m going to try to explain without any fancy equations; find those in the books.

1.1 Cell Matrices

The overall network is defined by layers. These are constructed using MATLAB’s “cell matrices”. The most useful feature of cell matrices for this project is that they may contain different sized matrices within each cell. This is a convenient way to store the information from each neural layer in the network into a single array, since each layer is a different size.

**Anytime I use curly brackets to address a variable, you can assume it’s a matrix. This is MATLAB’s cell notation.**

Consult the MATLAB help file for more information on cell matrices and their syntaxes.

2 Architecture

2.1 Network Attributes & Notation

There are other adjustable parameters, but the defining attributes of the network are:

- **Number of layers (size).** This can be thought of as the “length” of the network.
- **Number of neurons in each layer (structure).** Each layer in the network may have as many neurons as desired.
- **Connection weights.** These matrices establish weighted connections between all neurons in adjacent layers.
Figure 1: Forward propagation structure through the network. Each vertical bar represents a vector of values.

Once the size and structure of the network are established, the main purpose of the program is to adjust the weight values to obtain the desired response from a given input.

Each individual layer has attributes that we track throughout the training phase. These cell matrices are used for backpropagation calculations.

Figure 2: The matrices associated with neuronal layer \( j \). The local field is not present in the input layer, and there is no squashing function at the output layer.

There are a few variable conventions that I try to follow throughout the code:

- Layer numbers are labeled to stay consistent with most textbook equations: \( i \) is the previous layer, \( j \) is the current layer, \( k \) is the next layer.
- Values at current layer are \( y\{j\} \). Therefore, \( y\{1\} \) is the input vector.
- Local field in current layer \( v\{j\} \)
- Weights between layers \( w\{j\} \)

We end up with a model of any given layer that looks like Figure 2.

### 2.2 Weight Matrices

The matrix composition of the weights is the key to getting this algorithm to work. This is probably the most important part of this project. For a demonstration of this concept, see Figure 3.

\[
\begin{align*}
    \mathbf{w}\{2\} &= \begin{bmatrix} w_{d,0} \\
                                 w_{e,0} \end{bmatrix} \\
    \mathbf{w}\{1\} &= \begin{bmatrix} w_{a,d} & w_{a,e} \\
                                  w_{b,d} & w_{b,e} \\
                                  w_{c,d} & w_{c,e} \end{bmatrix}
\end{align*}
\]

Figure 3: A 3-layer perceptron with weights represented in matrix form.

In essence, rows of the weight vector represent the connections coming out of a single neuron, and columns represent connections going into a single neuron. This can be scaled up to any size interconnection.

The main reason for creating these matrices is optimization. A simple matrix multiplication now performs the tedious calculations for us.

\[
y\{k\} = \text{squash}(y\{j\} \ast w\{j\})
\]

In words, the values at the next layer \( k \) are the previous values multiplied by the connection weights from layer \( j \) to layer \( k \). A squashing function is used everywhere except the output layer (this helps the algorithm to converge more often).

### 2.3 Program Flow

On the whole, the phases of the program are as follows:

1. Define the input vector and desired response (teacher value)
2. Randomly assign weight values for a non-zero starting point
3. Forward propagate through the network
4. Find the error at the output
5. Backpropagation from the output back towards the input, calculating local gradient and making adjustments to the weight vector.

The local gradient calculation can be found in textbooks, and the implementation can be found in my extremely well-commented code.

This process is usually repeated until the error converges to zero. (Though I wouldn’t recommend it - many architectures do not converge. Ever.)

3 Convergence

Convergence depends on a many factors. Important, one of these factors is the initial weight values. Since these are randomly generated, convergence is largely random.

The network is trained in “epochs,” where each epoch is a forward and backward propagation. If we train the network, say, 500 times with a given pattern, we can study the convergence of the error energy. If it isn’t converging, we have some options:

- Continue training the network with fingers crossed, perhaps with different training patterns
- Re-initiate the weight vectors to another randomly distributed set of values (this may need to be done many times)
- Abandon the specified structure, adding or subtracting layers and changing their sizes

Running the code a few times will give a feel for which architectures are more convergent. These networks are very sensitive to initial conditions, so the weight vectors may need to be re-generated many times to get the right starting values.

4 Current/Future Work

The current algorithm recognizes a sin wave pattern buried in noise. The next step is to apply this algorithm to recognize some speech patterns. This report will be updated when it works/fails.

The .m-files accompanying this report should function standalone. They were written in MATLAB 7.9.0 (R2009b). There are detailed instructions on how to use the script in the comments of the main file.

The current algorithm allows the output to be a vector. This allows the program to make more complex decisions, rather than “yes” or “no”. e.g. a two-value output can choose one of four combinations at the output. (00,01,10,11)

If a Monte-Carlo-type teaching script was implemented, I might trust my computer to drive my car.