Fall 2015 COSC 3P71 Introduction to AI: Assignment 3

Instructor: B. Ombuki-Berman
TA: Justin Maltese

Due date: Monday, November 30th 2015, 10:00 AM
Lates: No lates
Goal: Feed-forward Neural Networks implementation

Languages: Any programming language available in our labs!

Hand in: (i) A listing of your source code (**ii**) **Example output** demonstrating the functionality of your program (**iii**) See further submission details at the end of this assignment. Include department cover page. **(iiii)** Electronic submission of all your code and data. Use "submit3p71" to submit your code. Run this at the top-level folder of your assignment directory. Be sure to name your top-level folder "Windows" or "Linux", to indicate what platform your executable is compiled for! The marker will use this to try your program.

Task: Use Backpropagation to learn a system for Parity Checking
Parity checking is a rudimentary method for detecting simple, single-bit errors in a memory system.

Every byte of data that is stored in a system memory contains 8 bits of real data, where each bit is a zero or a one. It is possible to count up the number of zeros or ones in a byte. For example, the byte 11011010 has 5 ones and 3 zeros. The byte 00101000 has 2 ones and 6 zeros. Hence, some bytes will have an even number of ones, and some will have an odd number. When parity checking is enabled, each time a byte is written into memory, a *parity generator/checker* (a.k.a logic circuit) examines the byte and determines whether the data byte had an even or odd number of ones. If it had an even number of ones, the ninth (parity) bit is set to a one, otherwise it is set to a zero. The outcome is that no matter how many ones there were in the original eight data bits, there are an odd number of ones when all the 9 bits are looked at all together. This is called *odd parity*. On the other hand, if the byte had an odd number of ones, and the parity bit is set to a one, the system would be checking for *even parity.* (Note: the standard in PC memory is odd parity)

Backpropagation Implementation

In this assignment, you are going to train a multi-layer feed forward neural network using the backpropagation learning algorithm seen in class to memorizethe parity memory. The architecture of your network will be a basic three layer (meaning only one hidden layer) feed-forward neural network. The backpropagation algorithm performs the input to output mapping by minimizing a cost function.

The network will implement a 4-bit system (1/2 a byte, For example, 0101), generating an even parity bit on the output. The input data will thus consist of 4-bit input patterns with an associated 1-bit output pattern (parity bit) for each. You will need to generate your own training data as described above. Below is a sample of the training data.

Sample Data Bits	Number of Ones in Data Bits	Parity Bit	Number of Ones including Parity Bit
0000	0	0	0
1000	1	1	2
1100	2	0	2
1111	4	0	4

since we are dealing with a 4-bit system, we know that there are 2^4 = 16 distinct bit patterns. The network once trained on all 16 patterns will be able to predict the corresponding parity bit when given a 4-bit pattern. Thus, if we were to apply any of the input patterns to the output layer, the output node should produce the correct parity.

It is helpful to recognize that all we are trying to do is find a set of weights that best allows our network to predict the correct parity bit when given a 4-bit pattern. The back-propagation algorithm is one such tool to accomplish this. Essentially, training a network using back-propagation requires the following steps:

Initialize the weights of the network randomly in the range [-1,1] or [-0.5,0.5]
while** all examples not classified correctly and not at maximum epoch **do
    for** each example e in the training set **do
        ActualOutput = getNeuralNetOutput(e) ; //forward pass
        ExpectedOutput = expected (correct) output for e
        Calculate error (ExpectedOutput - ActualOutput) at the output units
        Propagate error backwards until all nodes have an error contribution
        Update the weights in the network
    end
end

An overview of all required back-propagation equations, are provided. In the back-propagation algorithm, weight adjustments are controlled by a scaling factor called the *learning rate*. We need to scale the adjustments because large adjustments to the weights and threshold values would cause oscillation thus preventing convergence of the system. Convergence is achieved when all input patterns are correctly associated with their expected output patterns. Simply put, we must nudge all the weights and threshold values in the correct direction, bringing the entire network a little closer to convergence. Too much of a nudge and we over shoot our mark. Too little and we will wait forever to achieve convergence, if at all.

Essentially, a learning rate, which is too low initially, may get the system stuck in a local minimum. However, if the learning rate is too high then you may never find the global minimum because of over shooting the convergence point. Many presentations of the training set are required to achieve convergence. A large learning rate does speed things up but can cause the problems outlined above.

To speed up the training process we can introduce the concept of momentum. The concept of momentum is simple - if we make an adjustment to a weight then the probability is high that the next adjustment to that same weight will be in the same direction. Thus we introduce a momentum factor β, which applies a percentage of the previous weight update value to the current weight update value for each individual weight.
For the 4-bit even parity problem, you will need to implement a BP-ANN with 4 input nodes, 4

hidden nodes and 1 output node. You are required to add momentum to the system to aid in training. The 4-bit parity problem is relatively easy to conquer and thus the network should converge with little difficulty. Typical values of 0.5 for the learning rate and 0.9 for momentum seem to work consistently, but feel free to experiment with different values if time allows.

Hint:

Consider using a weighted adjacency matrix to represent connections of the network.

Further submission details:

A listing with each training pattern applied to the network along with the corresponding output that the network produces. Additional information such as the topology of the net, learning rates used and number of epochs needed should be included.