IQS2 **Spring 2013**

Programming Project 2: Artificial Neural Network

Assigned: Friday, February 22 *Due: Friday, April 5, 5:00 pm*

1 Introduction

In this project, you will write a Java application that implements an artificial neural network, as discussed in lecture. I will provide you with a good deal of code, including code that reads and writes matrices respectively from and to files. The network that you will implement will have three layers: a 256 unit input layer, a 256 unit hidden layer, and a 10 unit output layer. The purpose of the network is to classify hand drawn (by the drawing code provided) digits. Your neural network code will employ the back propagation algorithm along with simple gradient descent to update weight matrix values. Your code must also employ gradient checking to ensure that your back propagation code has been properly implemented.

You will work in groups of three to complete this. Choose wisely, since all group members will receive the same grade, and all group members will be responsible for understanding all of the code the group implements.

Since we have been discussing the concepts of neural networks in lecture, and since the lecture notes provide the relevant equations, I will not be including them here. I am, however, happy to discuss any aspect of the slides should you have questions.

2 Minimum Required Functionality

As part of your code, you must complete implementation of each of the following methods (method prototypes provided below and in the starter code). Descriptions of the methods are in the documentation in the starter code. **MODIFY THE CODE IN ANY METHOD OTHER THAN THOSE LISTED BELOW AT YOUR OWN RISK!** The other methods work. If you want to some print statements to help you debug, fine. But if you mess with the functionality of other methods, I won't help you fix your mess! This being said, there is nothing mysterious about any of the code. If you have questions about any aspect of the methods I provided, I'll be happy to discuss them.

- `private int getMatrix(Matrix m)`
- `private Matrix[] performBackPropagation()`
- `private static Matrix vectorizeY(String yValue)`
- `private static Matrix inputStringToMatrix(String input)`
- `private Matrix createInitialTheta(int rows, int cols)`
- `private double logisticFunction(double x)`
- `private Matrix logisticFunction(Matrix x)`
- `private Matrix computeHypothesis(Matrix input, Matrix theta1, Matrix theta2)`
- `private Matrix[] gradientCheck(Matrix[] trainingData, Matrix[] outputData, Matrix[] thetaValues, double lambdaValue)`
- `private double jTheta(Matrix[] trainingData, Matrix[] outputData, Matrix[] thetaValues, double lambdaValue)`
- `private double sumSquaredMatrixEntries(Matrix m)`

In addition, your application must satisfy the following

- It must be flexible in terms of the number of input units, hidden units, and output units. That is, though we will be using 256 input units, 256 hidden units (in a single hidden layer), and 10 output units, the sizes of each of these layers should be configurable — if you change the values of the constants in the program, your code should still work fine.
- Your code must employ the JAMA `Matrix` class and its associated methods for working with matrices and performing matrix calculations. See the class CS web page for the JAMA `Matrix` API.
- The names of all group members must be present in the documentation (comments) at the beginning of the `ClassifierWindow.java` file.
- You may NOT change the names of the files. That is, your program should contain exactly three classes, with the same names as the classes in the starter code!

2.1 Instrumenting Your Code For Debugging

The usual rules for adding debugging code apply. That is, you can add as much code as you want to help you see what is going on with your program (and in fact some debugging code will be required). But that output should not be present when your code is turned in. One way to do this is to have a `boolean` instance variable in your code that tells whether debugging output should be displayed. You write your code so that the program generates debugging output if and only if this variable is set to `true`. This means, basically, having a lot of `if` statements along the lines of

```
if (debug) {
    // show some debugging output
}
```

3 A Back Propagation Technicality

This section describes a technicality that you'll have to deal with when implementing back propagation. I didn't mention it in lecture, since I wanted you to understand the algorithm, not necessarily worry about a small coding detail. To see the problem, let's review a bit what you'll have to compute when running back propagation.

For our neural network, we have 256 input units, a single hidden layer with 256 units, and an output layer with 10 units. That means that $\Theta^{(1)}$ is 256 x 257 (the extra 1 comes from the bias unit) and $\Theta^{(2)}$ is 10 x 257.

Also $\Delta^{(1)}$ is 256 x 257 and $\Delta^{(2)}$ is 10 x 257, since the sizes of these should match the sizes of the corresponding Θ matrices.

You'll also need to compute $\delta^{(3)}$ and $\delta^{(2)}$. These and the Δ matrices will be updated (or computed) according to the following formulas.

$$\delta^{(3)} = a^{(3)} - y^{(i)}$$
$$\delta^{(2)} = \left(\Theta^{(2)}\right)^T \delta^{(3)} .* a^{(2)} .* (1 - a^{(2)})$$

$$\Delta^{(2)} := \Delta^{(2)} + \delta^{(3)}\left(a^{(2)}\right)^T$$

$$\Delta^{(1)} := \Delta^{(1)} + \delta^{(2)}\left(a^{(1)}\right)^T$$

Now, let's consider each equation in turn. For

$$\delta^{(3)} = a^{(3)} - y^{(i)},$$

both $a^{(3)}$ and $y^{(i)}$ are 10 x 1, so the difference is 10 x 1, and $\delta^{(3)}$ is thus 10 x 1.

For

$$\delta^{(2)} = \left(\Theta^{(2)}\right)^T \delta^{(3)} .* a^{(2)} .* (1 - a^{(2)}),$$

$\left(\Theta^{(2)}\right)^T$ is 257 x 10 and $\delta^{(3)}$ is 10 x 1, so the product is 257 x 1. Similarly, $a^{(2)}$ (with the bias unit included) is 257 x 1, as is $a^{(2)} .* (1 - a^{(2)})$, so $\delta^{(2)}$ is 257 x 1.

Now let's consider the last two update equations.

With

$$\Delta^{(2)} := \Delta^{(2)} + \delta^{(3)}\left(a^{(2)}\right)^T,$$

the last multiplication on the right involves a 10 x 1 multiplied by a 1 x 257 (if the bias unit in $a^{(2)}$ is included). This results in a 10 x 257 matrix, which works fine, because $\Delta^{(2)}$ is 10 x 257.

The problem, however, occurs with the last update equation above:

$$\Delta^{(1)} := \Delta^{(1)} + \delta^{(2)}\left(a^{(1)}\right)^T$$

The last multiplication on the right is a 257 x 1 matrix multiplying a 1 x 257 matrix (if we include the bias unit in $a^{(1)}$). Thus the product is 257 x 257. But $\Delta^{(1)}$ is 256 x 257. If nothing is done, the Matrix class will throw a run time exception, because you're trying to add matrices with different sizes!

The problem is those pesky bias units. And the solution is relatively simple. Before using $\delta^{(2)}$ in this last update equation (the one for $\Delta^{(1)}$), remove the first element of $\delta^{(2)}$, changing it from a 257 x 1 matrix into a 256 x 1 matrix. That makes $\delta^{(2)}\left(a^{(1)}\right)^T$ a 256 x 257 matrix, and we no longer have the size mismatch.

In case you're concerned that this is somehow throwing out information (it is), don't worry. Roughly speaking, the δ values are capturing the errors in our inputs to the following level of the neural network. But the bias unit inputs are always 1, so there really are no errors associated with those inputs (any “error” would be captured by the associated θ values). By throwing away that first entry of $\delta^{(2)}$, you’re throwing away something we don’t want to account for anyway.

4 Deliverables

You should provide me with a zipped BlueJ project called `XXXNeuralNetworkCharacterClassifier.zip`, where `XXX` represents **the first three letters of one of your group member’s last name**.

Your project should be submitted to me by emailing it to me. Please do not drop it in my netfiles inbox, since BlueJ, Netfiles, and Mac do not always play well together.