CS-540 Homework Assignment #4: Perceptrons and Neural Networks

Assigned: Wednesday, April 6th
Due: Sunday, April 16th

Hand-In Instructions

This assignment includes written problems and programming in Java. Hand in all parts electronically by uploading them in *a single zipped file* to the assignment page on Canvas.

If you choose to work with a partner on the programming portion, you still need to complete and submit the written portion individually. All students must submit the written PDFs + a README.txt (with a partner's name if applicable) to receive full credit. At least one member of each partnership (or team of one) must submit the zipped, completed code in addition to the PDFS and README.

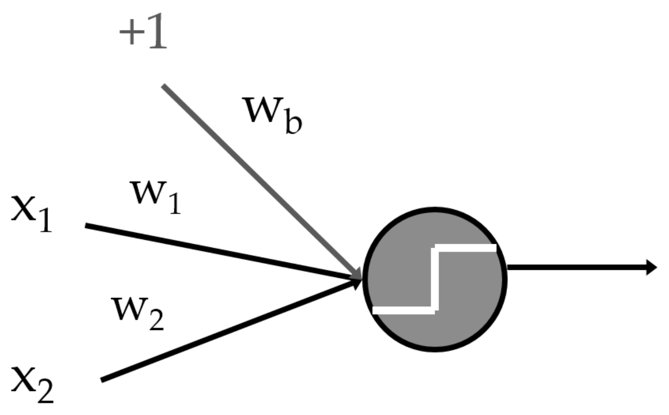
Your answers to each written problem should be turned in as separate pdf files labeled as <wisc NetID>-HW4-P<problem number>.pdf. You can use your program of choice – pencil and paper, word processor, LaTeX – as long as you show your work fluidly and neatly. If you handwrite your solutions to written problems, make sure to scan them in. *No photographed assignments will be accepted.*

For the programming problem, put *all* files needed to run your program, including ones you wrote and ones provided, into a folder called <wisc NetID>-HW4-P3.

Once you are finished, put your programming component folder and your two PDF files into a single directory. Zip it, name it <wisc NetID>-HW4, and upload it to the assignment Canvas page.

Problem 1: Perceptron Learning [15]

You want to teach the perceptron below with the following training instances. Assume that the activation function of the single output unit is the LTU function. That is, it returns 0 if the weighted sum of the incoming values is < 0, and 1 if the sum of the incoming values is >= 0.



x_1	x_2	y
2	2	1
0	0	0
-1	3	1
1	1	1
1	-2	0

The perceptron has been initialized with the following random weights: $w_1 = 0.5$, $w_2 = -1$, w_b (bias unit weight) = 0.5.

a) [7] Perform *one* epoch of the perceptron learning algorithm with an α (learning rate) of 0.2 on the training data. Walk through the training instances sequentially starting with the first instance in the table. Show the Δw_i values, the adjusted weights, and the T (desired output) and O (output) values after each instance.

Now, let's assume you want to design a different perceptron that represents the *majority* function for three binary variables. That is, the perceptron should return 0 if two input variables have the value 0, and it should return 1 if vice-versa.

b) [3] Show all possible instances for boolean variables x_1, x_2, and x_3 and label y in the form of a table, where y is the majority value of x_1, x_2, and x_3.

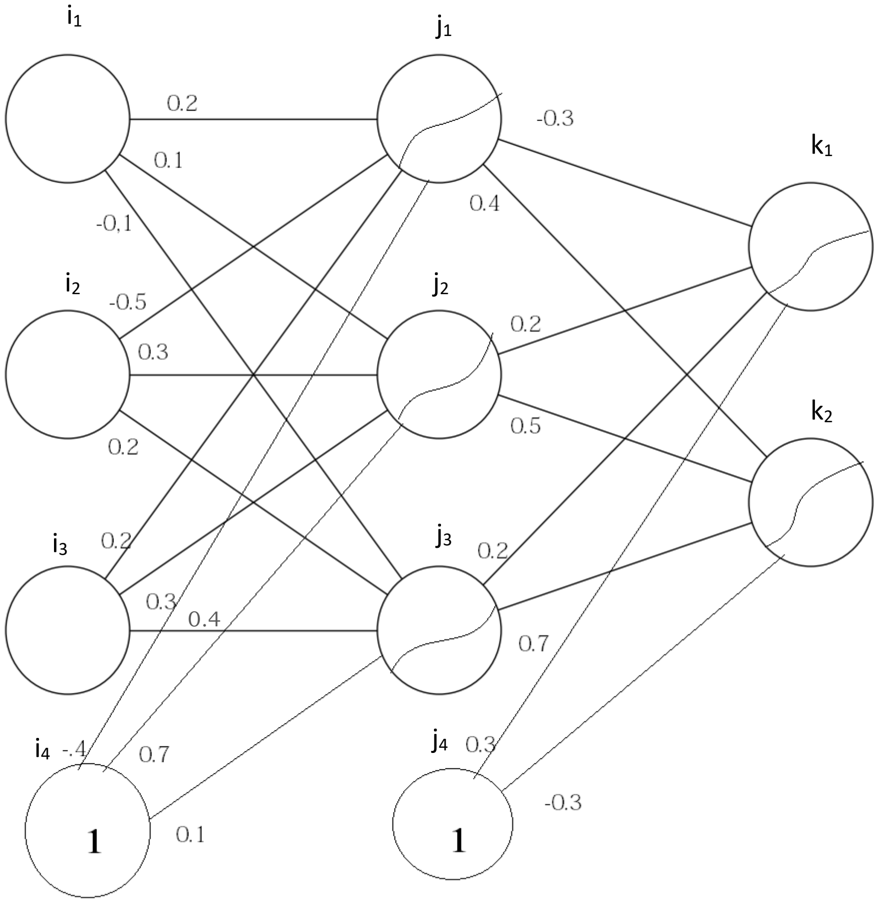
c) [5] Draw a perceptron with **three** input nodes, one bias node, and a single sigmoid output unit that correctly classifies all possible instances for the majority fucntion. Label each edge with weights.

When using the sigmoid output unit, assume the model returns the class (0 or 1) closest to the activation value of the function. For example, output 1 if the sigmoid function outputs 0.51, output 0 if the sigmoid function outputs 0.49.

Problem 2: [25] Back-Propagation

Let's say you've been given the feed forward multi-layer neural network below. For hidden and output nodes, assume the use of the Sigmoid activation function. Note the bias nodes at the input and hidden layers; their weights should also be adjusted.

The network weights have been randomly initialized as labeled in the diagram.



a) [5] What would the activation values at each of the two output nodes be if the training instance [0.3, 0.2, 0.1] was fed into the network input nodes? Show your work.

b) [20] Perform one epoch of the back-propagation algorithm with a learning rate of 0.2 on the same network using the following training set. Use the lecture slides, explanation in the textbook, and the tutorial posted in the Readings section on Canvas to guide you. Assume stochastic gradient descent: the network must be updated after each training instance. Show all deltas, activation values, and intermediate work to receive credit.

$\mathbf{x_1}$	$\mathbf{x_2}$	$\mathbf{x_3}$	$\mathbf{y_1}$	$\mathbf{y_2}$
1	0	1	0	1
0.5	1	0	1	0

Problem 3: [60] Feed-Forward Neural Network Implementation

Network reach is a measure of how many unique users see a post as it percolates through a social network. This is particularly relevant to businesses that advertise through Facebook posts, as the more unique users see a post, the more money the business earns.

Imagine that you run a business on Facebook, which means you want your posts to reach as many unique users as possible. View count alone isn't expressive enough, as one person viewing a post 100 times is less valuable than 100 people viewing the same post once. You would like to train a feed-forward neural network to predict the social reach of your post based on other attributes, like the number of comments, the number of total views, whether the post was a paid promotion, and the type of post (we're pretending that Facebook doesn't already track this information).

In this problem you are to write a program that builds a 2-layer, feed-forward neural network and trains it using the back-propagation algorithm. The problem that the neural network will handle is a **regression** problem that predicts post reach, or number of users who see a Facebook post, given numeric training data gathered from real-world Facebook posts. That is, the network will have exactly one output node, from which you will return the (log) predicted post reach as an output value.

The data has been standardized, scaled and cleaned so as not to saturate nodes in the network. If you're interested, I've included original figures and documentation in a zip file.

Like the previous assignment, this dataset is only part of the test suite with which graders will test your code, so be careful not to hardcode. We will test your network on other datasets.

All inputs to the neural network will be numeric. The neural network has one hidden layer. The network is fully connected between consecutive layers, meaning each unit, which we'll call a node, in the input layer is connected to all nodes in the hidden layer, and each node in the hidden layer is connected to all nodes in the output layer. Each node in the hidden layer and the output layer will also have an extra input from a "bias node" that has constant value +1. So, we can consider both the input layer and the hidden layer as containing one additional node called a bias node.

All nodes in the hidden and output layers (except for the bias nodes) should use the Rectified Linear Unit or ReLu activation function. The initial weights of the network will be randomized using an integer random seed passed in as an argument.

Assuming that input examples (called instances in the code) have m attributes (hence there are m input nodes, not counting the bias node) and we want h nodes (not counting the bias node) in the hidden layer, and o nodes in the output layer, then the total number of weights in the network is (m+1)h between the input and hidden layers, and (h+1)o connecting the hidden and output layers. The number of nodes to be used in the hidden layer will be given as input.

You are required to implement the following three methods from the class NNImpl and one method for the class Node:

```
public class Node{
public void calculateOutput()
}

public class NNImpl{
public int calculateOutputForInstance(Instance inst);
public void train();
public double getMeanSquaredError(List<Instance> dataset);
}
```

void calculateOutput() calculates the output at a particular node and stores that value in a member variable called outputValue.
int calculateOutputForInstance (Instance inst) calculates the output (i.e., the index of the class) for the neural network for a given example (aka instance).
void train() trains the neural network using a training set, fixed learning rate, and number of epochs (provided as input to the program).

double getMeanSquaredError(List<Instance> dataset) computes the mean squared error of a dataset, or (sum of (T-0)squared)/number of instances in the set

Implementation Details

We have created four classes to assist your coding, called Instance, Node, NeuralNetworkImpl and NodeWeightPair. Their data members and methods are commented in the skeleton code. We also give an overview of these classes next.

The Instance class has two data members: ArrayList<Double> attributes and double output. It is used to represent one instance (aka example) as the name suggests. attributes is a list of all the attributes (in our case binary pixel values) of that instance (all of them are double) and output is the observed output value for that training instance. The label (y) is a real number and not categorical.

The most important data member of the Node class is int type. It can take the values 0, 1, 2, 3 or 4. Each value represents a particular type of node. The meanings of these values are:

0: an input node
1: a bias node that is connected to all hidden layer nodes
2: a hidden layer node
3: a bias node that is connected to all output layer nodes
4: an output layer node

Node also has a data member double inputValue that is only relevant if the type is 0. Its value can be updated using the method void setInput(double inputValue). It also has a data member ArrayList<NodeWeightPair> parents, which is a list of all the nodes that are connected to this node from the previous layer (along with the weight connecting these two nodes).

This data member is relevant only for types 2 and 4. The neural network is fully connected, which means that all nodes in the input layer (including the bias node) are connected to each node in the hidden layer and, similarly, all nodes in the hidden layer (including the bias node) are connected to the node in the output layer. The output of each node in the output layer is stored in double outputValue. You can access this value using the method double getOutput(), which is already implemented. You only need to complete the method void calculateOutput(). This method should calculate the output activation value at the node if its type is 2 or 4. The calculated output should be stored in outputValue. The formula for calculating this value is determined by the definition of the ReLU activation function, which is defined as g(x)=max(0, x)

where x is the weighed sum of incoming edges times their corresponding activation vlaues. When updating weights you'll also use the derivative of the ReLU function, defined as

```
g'(x)=if x > 0: 1
        else: 0
```

NodeWeightPair has two data members, Node and weight. These should be self explanatory. NNImpl is the class that maintains the whole neural network. It maintains lists of all the input nodes (ArrayList<Node> inputNodes) and the hidden layer nodes (ArrayList<Node> hiddenNodes), and the output layer nodes (ArrayList<Node> outputNodes). The last node in both the input layer and the hidden layer is the bias node for that layer. Its constructor creates the whole network and maintains the links. So, you do not have to worry about that. As mentioned before, you have to implement two methods here. The data members ArrayList<Instance> trainingSet, double learningRate and int maxEpoch will be required for this.

To train the network, implement the stochastic gradient descent variation of the back-propagation algorithm given in the lecture slides. *Be careful to implement using ReLU, not the sigmoid function.* You can implement it by updating all the weights in the network after each instance (as is done in the algorithm in the textbook). This is the extreme case of Stochastic Gradient Descent. Finally, remember to change the input values of each input layer node (except the bias node) when using each new training instance to train the network.

Testing

We will test your program on multiple training and testing sets, and the format of testing commands will be:

```
java HW4 <numHidden> <learnRate> <maxEpoch> <trainFile> <testFile>
<randomSeed>
```

where trainFile, and testFile are the names of training and testing datasets, respectively. numHidden specifies the number of nodes in the hidden layer (excluding the bias node in the hidden layer). learnRate and maxEpoch are the learning rate and the number of epochs that the network will be trained, respectively. In order to facilitate debugging, we are providing you with sample training data and testing data in the files facebook_train.txt and facebook_test.txt A sample test command is

```
java HW4 5 0.01 100 facebook_train.txt facebook_test.txt 1
```

You are NOT responsible for any input or console output. We have written the class HW4 for you, which will load the data and pass it to the method you are implementing. **Do NOT modify any IO code.**

When finished, move your code (helper functions included) into the same directory as your two written PDFs, zip them up as <yourname>-HW4.zip and submit to Canvas. Don't forget a README if you worked with a partner.