Problem 3. [70] Back-propagation for Handwritten Digit Recognition

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 multi-class classification problem for recognizing handwritten digits. All inputs to the neural network will be numeric. The neural network has one hidden layer only. The network is also 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 the output 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 ReLU activation function. The initial weights of the network will be randomized. 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 two 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();
}
```

`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).

Dataset

The dataset we will use is called Semeion (https://archive.ics.uci.edu/ml/datasets/Semeion+Handwritten+Digit). It contains 1,593 binary images of size 16 x 16 that each contain one handwritten digit. Your task is to classify each example image as one of the three possible digits: 0, 1 or 2.

Each dataset will begin with a header that describes the dataset: First, there may be several lines starting with "`//`" that provide a description and comments about the dataset. The line starting with "`**`" lists the number of output nodes. The line starting with "`##`" lists the number of attributes, i.e., the number of input values in each instance (in our case, the number of pixels). You can assume that the number of classes will *always* be 3 for this homework because we are only considering 3-class classification problems. The first output node should output a large value when the instance is determined to be in class 1 (here meaning it is digit 0). The second output node should output a large value when the instance is in class 2 (i.e., digit 1) and, similarly, the third output node corresponds to class 3 (i.e., digit 2). Following these header lines, there will be one line for each instance, containing the values of each attribute followed by the target/teacher values for each output node. For example, if the last 3 values for an instance are: 0 0 1 then this means the instance is the digit 2. We have written the dataset loading part for you according to this format, so do NOT change it.

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 `ArrayList<Integer> classValues`. 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 `classValues` is the class (e.g., 1 0 0 for digit 0) for that instance.

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 $x = \sum_{i=1}^{n} w_i x_i$ and x_i are the inputs to the given node, w_i are the corresponding weights, and n is the number of inputs including the bias. When updating weights you'll also use the derivative of the ReLU, defined as $g'(x) = \begin{cases} 0, \text{if } x \leq 0 \\ 1, \text{otherwise} \end{cases}$

`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 back-propagation algorithm given in textbook (Figure 18.24) or in the lecture slides. 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.

Classification

Based on the outputs of the output nodes, `int calculateOutputForInstance(Instance inst)` classifies the instance as the index of the output node with the *maximum* value. For example if one instance has outputs [0.1, 0.2, 0.5], this instance will be classified as digit 2.

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>`
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 at 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 `train1.txt` and `test1.txt`. A sample test command is

```
java HW4 5 0.01 100 train1.txt test1.txt
```

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. As part of our testing process, we will unzip the files you submit to Moodle, remove any classes, call `javac HW4.java` to compile your code, and then call the main method, `HW4`, with parameters of our choosing.

Deliverables

1. Hand in your modified version of the code skeleton that includes your implementation of the back-propagation algorithm. *Do not submit any unmodified .java files*. Do include any additional java class files needed to run your program.

2. Optionally, submit a file called P3.pdf containing any comments about your program that you would like us to know.