Problem 3. Back-Propagation for Handwritten Digit Recognition [65 points]

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 images of handwritten digits. 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 layer (except for the bias node) should use the **ReLU** activation function, while all the nodes in the output layer should use the **Softmax** activation function. The initial weights of the network will be set randomly (already implemented in the skeleton code). 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 only required to implement the following methods in the classes `NNImpl` and `Node`:

```
public class Node{
      public void calculateOutput()
      public void calculateDelta()
      public void updateWeight(double learningRate)
}

public class NNImpl{
      public int predict(Instance inst);
      public void train();
      private double loss(Instance inst);
}
```

`void calculateOutput()`:
calculates the output at the current node and stores that value in a member variable called `outputValue`

`void calculateDelta()`:
calculates the delta value, Δ, at the current node and stores that value in a member variable called `delta`

`void updateWeight(double learningRate)`:
updates the weights between parent nodes and the current node using the provided learning rate

`int predict (Instance inst)`:
calculates the output (i.e., the index of the class) from the neural network for a given example

`void train():`
trains the neural network using a training set, fixed learning rate, and number of epochs (provided as input to the program). This function also prints the total Cross-Entropy loss on *all* the training examples after each epoch.

`double loss(Instance inst):`
calculates the Cross-Entropy loss from the neural network for a single instance. This function will be used by `train()`

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: 6, 8 or 9. If desired, you can view an image using the supplied python code called `view.py` Usage is described at the beginning of this file (this is entirely optional if you want to view what an image in the dataset looks like. In other words, it has nothing to do with your implementation in Java).

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 digits. 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 6). The second output node should output a large value when the instance is in class 2 (i.e., digit 8) and, similarly, the third output node corresponds to class 3 (i.e., digit 9). 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 9. 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`, `NNImpl` and `NodeWeightPair`. Their data members and methods are commented in the skeleton code. An overview of these classes is given 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 6) 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 it's of type 2 or 4. The calculated output should be stored in `outputValue`. The value is determined by the definition of the activation function (ReLU or Softmax), which depends on the type of node (type 2 means ReLU, type 4 means Softmax). Definitions of the ReLU and Softmax activation functions are given at https://github.com/Kulbear/deep-learning-nano-foundation/wiki/ReLU-and-Softmax-Activation-Functions as well as in the Notes section below.

`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 input nodes (`ArrayList<Node> inputNodes`), hidden layer nodes (`ArrayList<Node> hiddenNodes`), and 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 it's already implemented in the skeleton code. As mentioned before, you must implement these three 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 the textbook (Figure 18.24) or in the lecture slides. Implement it by updating all the weights in the network after *each* instance (as is done in the algorithm in the textbook), so you will be doing a form of Stochastic Gradient Descent for training. Use Cross-Entropy loss as the loss function at the output nodes for the back-propagation algorithm. Details about Cross-Entropy loss are available at http://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html 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 predict(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, 0.3, 0.6], this instance will be classified as digit 9. If there are multiple maximum values, the smallest digit will be predicted. For example, if the outputs are [0.4 0.4 0.2], then the instance should be classified as 6.

Testing

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

```
java DigitClassifier <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. To facilitate debugging, we are providing you with sample training data and testing data in the files `train1.txt` and `test1.txt`. We are also providing you the file `testcases.txt` which contains three example test cases. The outputs for these three test cases are also provided. A sample test command is

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

You are *not* responsible for handling parsing of the training and testing datasets, creating the network, or printing test dataset results on console. We have written the class `DigitClassifier` 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 java files you submit to Canvas, call `javac DigitClassifier.java` to compile your code, and then call it with above command, with parameters of our choice.

Deliverables

Hand in your modified versions of `NNImpl.java` and `Node.java` that include your implementation of the back-propagation algorithm. *Also submit any additional helper `.java` files required to run your program* ***(including the provided ones in the skeleton)***. Do *not* submit any other files including the data files (no `.class` or `.txt` files). Also, all the `.java` files should be zipped as `<wiscNetID>-HW4-P3.zip` for submission to Canvas.

Notes

- Use the `Collections.shuffle(list,random)` function on the training data with the random object available in the class only *once* before every epoch while implementing the `void train()` function. For the purpose of this homework, we fixed the `random` object to a certain value. Therefore, you should not worry about what `random` is. **You only have to make sure that the shuffling given above happens *once before every epoch*.**

- After *each* epoch of training and weight updating, print the cumulative Cross-Entropy loss on the *entire* training set in the format shown in the sample output. This should be done in the `train()` function. Use 3 decimal double precision using `%.3e` to get the required formatting of the sample output.

- Do ***not*** include package statements in your final submission (i.e., do *not* include statements in the java files that start with the keyword **package**).

- You only need to implement the above-mentioned methods and you must *not* change any other existing function implementations. However, you are free to add any helper methods you'd like to implement these methods.

- Make sure your implementation can run under 30 seconds for a `maxEpoch` of 100 and 10 hidden units, otherwise it will timeout by the grading script.

- To better understand how backpropagation and weight updating work in detail, you can look at the following tutorial. **Note, however, that the example in this tutorial uses a *different* loss function; therefore, the computations made in the tutorial will not be**

exactly the same as in this homework. https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/

- The **ReLU** function is defined as

$$g(z) = \max\ (0, z)$$

where $z = \sum_{i=1}^{n} w_i\, x_i$ 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 need to use the derivative of the ReLU, defined as

$$g'(z) = \begin{cases} 0, & z \leq 0 \\ 1, & otherwise \end{cases}$$

- The **Softmax** function is defined as

$$g(z_j) = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

where z_j is the weighted sum of its inputs at the j^{th} output node, and K is the number of output nodes.

- The **Cross-Entropy loss** function for a single example $\boldsymbol{x}^{(i)}$ is defined as

$$L^{(i)} = -\sum_{k=1}^{K} y_k^{(i)} \ln g(z_k)$$

where $\boldsymbol{y}^{(i)}$ is the target class value of the i^{th} example $\boldsymbol{x}^{(i)}$. For example, if the target digit is 6, then the target class value $= [1, 0, 0]$. The total loss is defined as the average loss across the entire training set:

$$L = \frac{1}{|D|} \sum_{i=1}^{|D|} L^{(i)}$$

where $|D|$ is the number of examples in the training set. When updating weights, it's easier to compute the gradients of the Softmax activation function and Cross-Entropy loss together, which is defined as

$$\frac{\partial L^{(i)}}{\partial z_j} = g(z_j) - y_j^{(i)}$$

More details can be found at https://deepnotes.io/softmax-crossentropy

- Based on the above information, if we set the learning rate to α, the weight update rule for this neural network is

$$\Delta w_{ij} = \alpha a_i\, \Delta_j$$

where a_i is the output of node i, and

$$\Delta_j = \begin{cases} y_j - g(z_j), & if\ j\ is\ an\ output\ unit \\ g'(z_j) \sum_k w_{jk}\, \Delta_k, & if\ j\ is\ a\ hidden\ unit \end{cases}$$