CS 540-2: Introduction to Artificial Intelligence

Homework Assignment #4

Assigned: Tuesday, March 28
Due: Sunday, April 9

Hand-In Instructions

This assignment includes written problems and programming in Java. **All problems must be done individually.** Hand in all parts electronically by copying them to the Moodle dropbox called "HW4 Hand-In". Your answers to each written problem should be turned in as separate pdf files called <wisc NetID>-HW4-P1.pdf and <wisc NetID>-HW4-P2.pdf. If you write out your answers by hand, you'll have to scan your answer and convert the file to pdf. Put your name at the top of the first page in each pdf file.

For the programming problem, put all the java files needed to run your program, including ones you wrote, modified or were given and are unchanged, into a folder called <wisc NetID>-HW4 Put your Confusion Matrix in a pdf file called <Wisc NetID>-HW4-P3.pdf and put it into this folder as well. Compress this folder to create <wisc NetID>-HW4-P3.zip and copy this file to the Moodle dropbox. Make sure your program compiles and runs on CSL machines. Your program will be tested using several test cases of different sizes. Make sure all files are submitted to the Moodle dropbox.

Late Policy

All assignments are due **at 11:59 p.m.** on the due date. One (1) day late, defined as a 24-hour period from the deadline (weekday or weekend), will result in 10% of the total points for the assignment deducted. So, for example, if a 100-point assignment is due on a Wednesday and it is handed in between any time on Thursday, 10 points will be deducted. Two (2) days late, 25% off; three (3) days late, 50% off. No homework can be turned in more than three (3) days late. Written questions and program submission have the same deadline. A total of three (3) free late days may be used throughout the semester without penalty. Assignment grading questions must be raised with the instructor within one week after the assignment is returned.

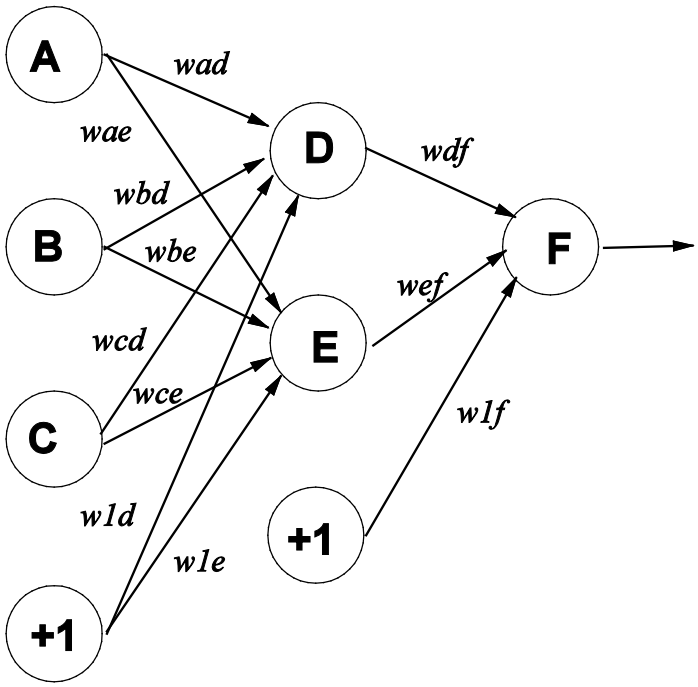
Problem 1. [10] Support Vector Machines

Do Exercise 18.17 in the textbook: Construct a support vector machine that computes the XOR function. Use values of +1 and -1 (instead of 1 and 0) for both inputs and outputs, so that an example looks like ([-1, 1], 1) or ([-1, -1], -1). Map the input $[x_1, x_2]$ into a space consisting of x_1 and x_1x_2 . Draw the four input points in this space, and the maximal margin separator. What is the margin? Now draw the separating line back in the original Euclidean input space.

Problem 2. [20] Neural Networks and Back-Propagation

(a) [5] Define a neural network with two input units and one output unit that computes the Boolean function $(A \vee \neg B) \wedge (\neg A \vee B)$. If it is possible to construct a Perceptron that implements this function, define a set of weights that will work. If it is not possible to implement this function using a Perceptron, define a two-layer feed-forward network (i.e., containing one hidden layer) that implements it, using as few hidden units as possible. Use the definitions in Figure 7.8 in the textbook for the logical connectives $\neg$, $\wedge$ and $\vee$. Let 1 correspond to True and 0 False. Use a step function (LTU) as the activation function. Draw a figure that shows your network topology and the weights and bias values used.

(b) [15] Consider a learning task where there are three real-valued input units, A, B and C, and one output unit, F. You decide to use a 2-layer feed-forward neural network with two hidden units, D and E, defining the hidden layer, as shown in the figure below. The activation function used at nodes D, E and F is the *sigmoid* function defined in Figure 18.17b in the textbook. Each input unit is connected to every hidden unit, and each hidden unit is connected to the output unit. The initial weights and biases are given as: *wad* = 0.3, *wae* = -0.1, *wbd* = 0.3, *wbe* = -0.1, *wcd* = 0.3, *wce* = -0.1, *w1d* = 0.2, *w1e* = 0.2, *wdf* = 0.3, *wef* = -0.1, *w1f* = 0.2.



(i) [5] What is the output of nodes D, E and F given a training example with values A = 0.3, B = 0.8, and C = 0.1? Assume that D, E, and F all output real values as computed by their associated *sigmoid* function.

(ii) [10] Compute *one* (1) step of the Back-propagation algorithm (see Figure 18.24 in the textbook) using the same inputs given in (i) and teacher output F = 1. The error should be computed as the difference between the integer-valued teacher output and the real-valued output of unit F. Using a learning rate of α = 0.2. Give your answer as the 11 new weights and biases. Show your work.

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 5-class classification problem for recognizing five handwritten digits: 1, 4, 7, 8 and 9. 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 *Sigmoid* 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 five possible digits: 1, 4, 7, 8 and 9. If desired, you can view an image using the supplied python code called `view.py` Usage is described at the top of this file.

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 5 for this homework because we are only considering 5-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 1). The second output node should output a large value when the instance is in class 2 (i.e., digit 4), and similarly, the third output node corresponds to class 3 (i.e., digit 7), the fourth output node corresponds to class 4 (i.e., digit 8) and the last output node corresponds to class 5 (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. If the last 5 values for an instance are 1 0 0 0 0 this means the instance is the digit 1. Similarly, '0 1 0 0 0' means digit 4, '0 0 1 0 0' means digit 7, '0 0 0 1 0' means digit 8, and '0 0 0 0 1' means 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`, `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 0 0 for digit 1) 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 Sigmoid activation function, which is defined as

$$g(x) = 1/(1 + e^{-x})$$

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 Sigmoid, defined as

$$g'(x) = g(x)\,(1 - g(x))$$

`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 (round values to 1 decimal place). For example, if one instance has outputs [0.1, 0.2, 0.5, 0.3, 0.1], this instance will be classified as digit 7. If there is tie between multiple values, the instance will be classified as the largest digit. For example, if outputs are [0.1, 0.4, 0.2, 0.4, 0.1], it should be classified as 8.

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 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 `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 *.java` to compile your code, and then call the main method, `HW4`, with parameters of our choosing.

Confusion Matrix

A Confusion Matrix depicts the number of matches and mismatches by the prediction model over instances of all possible classes. In our convention of the Confusion Matrix, a row represents an actual class and each cell of the row shows the number of instances that were classified by the prediction model as the class represented by the column header. The name stems from the fact that it makes it easy to see if the system is confusing pairs of classes (e.g., digit 1 being confused with digit 9).

Example: Say there are 40 instances of digit 1 in a test set and 35 instances of the total are correctly classified as digit 1, but 3 instances are misclassified as digit 9, 1 instance is misclassified as digit 4, and 1 instance as digit 7. Then the row for digit 1 will be filled as shown in the figure below. Similarly, rows for the other digits will get filled.

		Predicted Class				
		Digit 1	**Digit 4**	**Digit 7**	**Digit 8**	**Digit 9**
Actual Class	**Digit 1**	35	1	1	0	3
	Digit 4					
	Digit 7					
	Digit 8					
	Digit 9					

You are required to build a Confusion Matrix using the input parameters given below by calculating the actual and predicted classes for the test set `test1.txt` after training using the training set `train1.txt` The provided skeleton code prints actual and predicted classes for each test instance, which you can use to determine the value to *manually* enter in each cell of the matrix. Put your Confusion Matrix in a *pdf* file called `<Wisc NetID>-HW4-P3.pdf`

Input parameters for creating your Confusion Matrix:
Number of nodes in the hidden layer: 5
Learning rate: 0.02
Number of epochs: 500

So, you should run the following to generate the data used to construct your Confusion Matrix:

```
java HW4 5 0.02 500 train1.txt test1.txt
```

Deliverables

Submit *all* your java files. You should *not* include any package path or external libraries in your program. Copy all source files into a folder named `<Wisc NetID>-HW4` and also put into this folder the file called `<Wisc NetID>-HW4-P3.pdf` that contains your Confusion Matrix. Then compress this folder into a zip file, called `<Wisc NetID>-HW4-P3.zip` and submit this `.zip` file to Moodle.

Extra Credit

Perform various experimental evaluations such as using *k*-fold cross-validation to find the best number of hidden units, computing a training curve (see Figure 18.25(a)), computing a learning curve (see Figure 18.25(b)), and comparing training time and accuracy using different types of activation functions such as sigmoid versus ReLU. Describe what you did and report your results.