CS 3445 – Fall, 2019
Project 4

Neural Networks for Digit Recognition

Due: November 21 (recommended), but no later than November 24

For this assignment, I want you to design and implement a program that uses a perceptron to recognize discretized images of handwritten digits, and then use your program to explore input and output representation issues. A neural network with a hidden layer would probably work better, but I want you to focus on representation issues, and perceptrons are much simpler to implement.

Requirements

Your code should allow you to:

1. create a fully-connected perceptron with a given number of input nodes and a given number of output nodes,
2. train that perceptron for a given number of epochs with a given learning rate on a given file of training examples, and
3. test the perceptron on a given file of test examples, calculating its accuracy.

I would like you to use your code to explore the impact of the input and output representations on the performance of the perceptron as a function of the number of training epochs.

Input Representations

I have provided you with files that contain training and test input/output pairs for the digit recognition problem. There are two sets of files, corresponding to two different ways of representing the input:

1. A 32 x 32 bitmap of 0s and 1s representing the image of the digit (see an example on the next page),
2. An 8 x 8 down-sampled image, which was created by dividing the 32 x 32 bitmap into 64 non-overlapping 4 x 4 blocks and creating a new 8 x 8 matrix whose elements are integers in the range 0-16 and represent the number of 1s in the corresponding 4 x 4 blocks in the original.

```
00000000000000001110000000000000
00000000000000001110000000000000
00000000000000001110000000000000
00000000000000011100000000000000
00000000000000011100000000000000
00000000000000111100000000000000
00000000000000111100000000000000
00000000000001111100000000000000
00000000000001111100000000000000
00000000000001111100000000000000
00000000000011111100001111000000
00000000000011111000001111000000
00000000000011111000001111000000
00000000000111111000001111000000
00000000001111110000011111000000
00000001111111100000111111000000
00000011111111000000111110000000
00000011111111100000111110000000
00000111111111100001111110000000
00000011111111111111111100000000
00000011111111111111111100000000
00000000111111111111111100000000
00000000000000111111111000000000
00000000000000011111111000000000
00000000000000001111110000000000
00000000000000001111110000000000
00000000000000001111110000000000
00000000000000001111110000000000
00000000000000001111110000000000
00000000000000001111110000000000      A sample bitmap (for a handwritten "4")
```

There are input/output pairs for all of these representations in the `digit-recognition-examples` folder in `assign2-distributed.zip`, available on Blackboard in the Assignments section. In all cases there are 3,823 input/output pairs on which to train your perceptron and 1,797 input/output pairs on which to test your perceptron.

The 32 x 32 examples are in the `32x32` folder and are in the following files:

- `optdigits-32x32-info`
- `optdigits-32x32.tra`
- `optdigits-32x32.tes`

The file `optdigits-32x32-info` contains some information about the data. The file `optdigits-32x32.tra` contains the training examples. The file `optdigits-32x32.tes` contains the testing examples. An input is represented as 32 lines of 32 0s and 1s, and the digit that this is an image of is a single digit on the next line.

The 8 x 8 examples in which the matrix elements are integers in the range 0–16 are in the `8x8-integer-inputs` folder and are in the following files:

- `optdigits-8x8-int-info`
- `optdigits-8x8-int.tra`
- `optdigits-8x8-int.tes`

The file `optdigits-8x8-int-info` contains some information about the data. The file `optdigits-8x8-int.tra` contains the training examples. The file `optdigits-8x8-int.tes` contains the testing examples. An input is represented as a line of 65 comma separated integers, where the first 64 represent the input and the last one is the digit that this is an image of.

Output Representations

I would like you to also explore two ways of representing the output of your perceptron:

1. a single output node, whose output value is multiplied by 10 and rounded to get the perceptron's classification (assuming your output node activation function is a sigmoid), and
2. ten nodes, each representing a digit, where the classification is taken as the digit that the node with the maximum value is representing.

In some cases, the output format of the example files corresponds closely to one of these output representations. You will need to translate for the situations where this is not the case, e.g. if the file specifies the correct digit as an integer, you will need to translate it into a vector of 0s and 1s for the second output representations.

Tests

I would like you to test the learning abilities of your perceptron for each of the four possible combinations of input and output representations. For each pair of input and output representations, how does the performance of the network change as the network is trained?

Here are two reasonable ways to measure performance: 1) the mean squared error (the average squared error over all the input/output pairs), and 2) the percentage of inputs classified correctly. For this assignment, you only have to do the second one.

You should provide graphs for this measure showing how it changes as the network is being trained, i.e. how is the error on the training set changing as the network is being trained. You should also report the performance of the final network on the test set.

The topology of the perceptron is fully specified by the input and output representation (which is why I wanted you to use perceptrons instead of 3-layer networks), so the only parameters that can vary are the learning rate and maximum number of epochs. I would like you to run your tests for a range of learning rates from 0.1 to 1.0 (perhaps 0.1, 0.5, and 1.0). If it looks like it's necessary to use a learning rate smaller than 0.1 in some cases or beneficial to use a learning greater than 1.0, try some outside of that interval as well. A learning rate of 1.0 is much larger than the rule of thumb I mentioned in class (0.01), but it seems to work well here. My network, using the second input representation and the second output representation and a learning rate of 1.0, was able to classify 82% of the test set correctly. With a learning rate of 0.1, it was able to classify only 72% of the test set correctly. If you have time, it would be worth exploring even more extreme learning rates.

You will need to specify a maximum number of epochs. It should be large enough that the performance in each of your tests levels off, but small enough that training takes a reasonable amount of time. Past experience indicates that 50 epochs should be sufficient, but train for longer if performance has not leveled off at that point.

So, you need to test the four pairs of input/output representations on (at least) the three learning rates specified above and, for each of these 12 cases:

1. create a graph of the performance measure during training, and
2. report the performance measure for the network's performance on the test set.

Note that since a trained network will be different depending on the initial weights, you should train and test networks for the four pairs of input/output representations multiple times with different initial weights in order to get a good idea of their performance. **Note, also, that since I want you to have more time to work on your final projects, I'm not going to ask you to do this.** One network for each of the four pairs is enough.

Notes

Initial Weights

Although the initial weights in the neural network should be random, I found that the range of those weights had a significant impact on the trainability of the network. I got good results initializing the weights to random doubles between -1.0 and 1.0 (but, see the next note about the activation function).

The Activation Function

If the initial weights are "too large," the inputs to the activation function will be so large that the outputs will all be 1.0, which makes it pretty hard to learn. We need the activation function to be in the "right" place, and this can be accomplished either by playing with the initial weights to shift the input to the activation function, or by playing with the constant in the exponent of the sigmoid to shift the ramp. A previous student who did this project found that restricting the initial weight magnitudes to no more than 0.15 worked well. I got good results shifting the ramp by adding 0.5 to the exponent:

$$g(x) = \frac{1.0}{1 + e^{(-x+0.5)}}$$

Bias Nodes

You will probably need to have a *bias node* in the input layer. The bias node is an extra node whose activation level is always 1. It should be connected to every output node and the weights on those connections should be trained just like the weights on connections between the other input nodes and output nodes.

Grading

Since I have prescribed the tests you should run, the grading will be split between your code and your report:

50%: the correctness and clarity of your code,

50%: the content, organization, and clarity of your report.

NOTE: There will be no project reviews for this assignment, so it is even more important than usual that your code be clear and well-documented.