Project 3: Neural Networks
CSC 330

Introduction

In this project, we will consider a number of practical issues in the application of neural networks to learning problems. As in the past, you'll start with some code provided on the I: drive, allowing you to focus on the AI issues.

Every part of this project, even the hand-computation stuff, will be done in your pairs. For each hand-computation, once you've worked it out, feel free to show me your answer, even before the assignment is due. I'll tell you if you're correct or not. That way, you can know if you have the correct numbers to compare to your program output, providing a simple guaranteed way to test your program before turning it in.

As with all other projects, I urge you to start early and spread out your work throughout the assignment period, aiming to finish early. That way you can stop if you get stuck and have plenty of time to ask me questions. This is the single greatest predictor of your success (and learning and happiness) on any AI project!

Part 1: Perceptron Computation and Learning

Due:

- Thursday 4/18 10:00 AM (start of class) – handwritten work on a physical sheet to turn in (1 copy per group)
- Thursday 4/18 11:55 PM – Program, via Moodle

1. Perceptron Computation, By Hand

Before we program a perceptron network, let's practice the basics by running a small example by hand. On paper, draw a network with the following architecture:

Inputs: one bias ($a_0 = -1$) and two regular inputs (a_1 and a_2).
Output layer: one output node (a_3)

Define weights:
$W_{0,3} = 0.3$
$W_{1,3} = 0.1$
$W_{2,3} = -0.2$

Use the sigmoid activation function. Run the network (by hand) on the inputs (-.5, .3) and show the output.

As I said in the introduction above, if you want to check your handwritten answers with me before moving on, please feel free.

2. Perceptron Computation, Programming

Review the provided code. The Example, RandomWeightGenerator, and Sigmoid classes are straightforward. The Perceptron class defines a single Perceptron. Your task for this problem is to fill in the computeOutput() method in the Perceptron class.

Note the provided makeHardcoded() class method, to facilitate setting the weights by hand for easy computation of examples. makeHardcoded() is currently set to create the network for the by-hand problem you just did above. testPerceptronOutput() shows how to run the network on this problem.

So, if computeOutput() is defined correctly, then when you run testPerceptronOutput() you should get the same answers as you got by hand. If your code doesn't work, the first thing I'd double check is that you handled the bias properly. Sometimes it's easy to forget about.

3. Perceptron Learning, By Hand

Next, let's practice perceptron learning by hand. Using the same architecture and initial weights you worked with by hand above, train the network by hand on
(-.5, .3) --> .6
and then on
(.2, -.4) --> .8
That is, update the weights with the first example, and then update those updated weights with the second example. Show all of your work.

Assume a learning rate of 1 (just for now), and use the sigmoid activation function, of course.

As before, please feel free to check your answer with me before moving on, if you'd like. As a quick hint now, I'll tell you that the final weight for $W_{1,3}$ should be .09280.

4. Perceptron Learning, Programming

Now that you've had some good practice with perceptron learning by hand, please define the train1Example() and trainEpochs() methods in the Perceptron class. Both methods mutate the perceptron's weights according to the example(s). Of course, trainEpochs() should call train1Example() repeatedly.

When you're done with train1Example(), the method testPerceptronLearning() should give you the same numbers you got by hand after training only on the first example above. (Didn't work? Check your bias code first!)

When you're done with trainEpochs(), the method testPerceptronLearning2() should give you the same numbers you got by hand after training on the first followed by the second example above.

In addition, once you're done with train1Example() and trainEpochs(), you should be able to run the following methods in the PerceptronNetwork class:

```
learnAnd(1000):
[{7.093398540304798, [4.670637330415617, 4.6668588753181455]}]
Example : [0.0, 0.0] --> [0.0]: [8.298805647569949E-4]
Example : [0.0, 1.0] --> [0.0]: [0.08117117360669923]
```

```
Example : [1.0, 0.0] --> [0.0]: [0.08145342624605958]
Example : [1.0, 1.0] --> [1.0]: [0.9041401944461012]
Average output node error: 0.06482857149285366

learnOr(1000):
[{2.4308353162875385, [5.373356923590509, 5.375578706456158]}]
Example : [0.0, 0.0] --> [0.0]: [0.08085136942728424]
Example : [0.0, 1.0] --> [1.0]: [0.9500144575420262]
Example : [1.0, 0.0] --> [1.0]: [0.9499088462153451]
Example : [1.0, 1.0] --> [1.0]: [0.9997560004905298]
Average output node error: 0.045293016294845785

learnXor(100000):
[{-0.06444631063475031, [-0.1288926212695009, -0.06444631063475056]}]
Example : [0.0, 0.0] --> [0.0]: [0.51610600358629]
Example : [0.0, 1.0] --> [1.0]: [0.5]
Example : [1.0, 0.0] --> [1.0]: [0.4838939964137099]
Example : [1.0, 1.0] --> [0.0]: [0.46782138179306076]
Average output node error: 0.5000083472414103
```

Note that AND and OR are learned with no problem, but even after 100,000 epochs, XOR has not been learned. This is expected, as we'll see in class soon if we haven't already. Your actual numbers won't necessarily match those above, but AND and OR should clearly be learned, and XOR not learned, of course.

Take a quick look at the PerceptronNetwork class. Note that it just calls the methods you defined in the Perceptron class, on each perceptron in the network. As we've said in class, each perceptron is just treated individually.

Part 2: Backpropagation Learning

Due:

- Thursday 5/2 10:00 AM (Start of class) – handwritten work on a physical sheet to turn in (1 copy per group)
- Thursday 5/2 11:55 PM – Program, via Moodle

Note that part 2 is **not** dependent on correctly-working code for part 1.

1. Backpropagation Learning, By Hand

Now, let's do the same for a multi-layer network. First, we'll do an example by hand to make sure the computations are clear. Draw a network with the following architecture:

Inputs: one bias (a_0 = -1) and two regular inputs (a_1 and a_2), fully connected to the non-bias nodes of the hidden layer. Initialize all weights in this layer to 0.1.

Hidden layer: one bias (a_3 = -1, with no inputs) and two regular nodes (a_4 and a_5), each connected to the output layer. Initialize all weights in this layer to -0.1.

Output layer: one node.

So you should have weights
$W_{0,4} = W_{0,5} = W_{1,4} = W_{1,5} = W_{2,4} = W_{2,5} = 0.1$
$W_{3,6} = W_{4,6} = W_{5,6} = -0.1$

There are no other weights in the network. Ordinarily, weights are initialized to a small random positive or negative value, but setting them more systematically as above will keep your and my computations consistent and simplify them a little.

Set the learning rate to 1. Usually this is lower, but again, this will simplify our computations. Use the sigmoid activation function. Run the network on the example (1, 0) --> 1, and update the weights using backpropagation. Show your work!

As before, feel free to check your answers with me before moving on. To give you one hint now, the updated $W_{4,6}$ is -0.0375.

2. Backpropagation Learning, Programming

Doing these examples by hand is essential, because in programming this there are many opportunities for mistakes with otherwise unclear consequences. Now that you have a by-hand example clearly annotated, please complete the implementation of the MultilayerNetwork class: computeOutput(), train1Example(), and trainEpochs().

Here's my advice on this: Each ***piece*** of backpropagation is pretty easy to code; each piece is just a loop or two, and working with arrays. What makes coding backpropagation a little harder, though, is that there are so many pieces and so much notation to keep straight. To successfully program this, it is essential that you take the time to use very clear variable names and carefully document (with comments) each step you're doing. I recommend you use the same variable names we've used in class, and that you keep diagrams and formulas in front of you the entire time. If you take your time to write your code very clearly and carefully, and use your partner to help you catch mistakes, it'll just be a matter of time before you're done. But if you write messy code, it'll be very confusing to work with to find that one subtle mistake.

When you're done, you should be able to run the testMultilayerNetwork() method in the MultilayerNetwork class and get the same numbers you got by hand above. (Didn't work? Check your bias code first!)

You should also be able to run the testMultilayerNetwork2() method in the MultilayerNetwork class and get the following numbers:

```
------------------------------
Before training

bias->hidden: [0.9, 0.8]
input->hidden: [  [0.6, 0.5]  [0.4, 0.3]  ]
bias->output: [0.7, 0.6]
hidden->output: [  [0.2, 0.1]  [1.1, 1.2]  ]

output: [0.4651993332181634, 0.4900215505714809]
```

```
------------------------------
After training

bias->hidden: [0.9007307265064192, 0.7956462163843392]
input->hidden:   [      [0.5994154187948646,   0.5034830268925287]
[0.39970770939743233, 0.3017415134462643]  ]
bias->output: [0.7410997626373935, 0.5475262950822211]
hidden->output:  [    [0.18210665493216902,  0.12284514675583785]
[1.0823084519775326, 1.2225875044257566]  ]

output: [0.45167341326274113, 0.5086326269496642]
```

If the testMultilayerNetwork method numbers match your by-hand work, but the testMultilayerNetwork2 method numbers don't match the above, it probably means you have a mistake somewhere in dealing with more than one output unit.

Finally, you should also be able to run learnXorML(10000) and get something like:

```
bias->hidden: [-5.834272434447614, -2.2426522519809806]
input->hidden:   [     [-3.9597874609231285,   -5.8998830716947435]
[-3.9635848727689216, -5.933069464293481]  ]
bias->output: [3.5836177587162252]
hidden->output: [  [7.809597765314744]  [-8.086388385420378]  ]

Example : [0.0, 0.0] --> [0.0]: [0.04281878278294458]
Example : [0.0, 1.0] --> [1.0]: [0.9519724234126947]
Example : [1.0, 0.0] --> [1.0]: [0.9518326434043561]
Example : [1.0, 1.0] --> [0.0]: [0.06158006232881548]
Average output node error: 0.0501484445736773
```

Here, your numbers won't match mine exactly, since we're starting with random initial weights. It's also possible, every once in a while, that your network still won't learn XOR, even after 10,000 epochs. But most of the time it should, as above.

Notice that it's still not particularly easy for the network to learn XOR. When I was preparing this project, my network didn't learn XOR in 1,000 epochs. This depends on the luck of initial weights, and the learning rate too, of course. The reason XOR is kind of hard to learn is because its output is actually very strongly not linearly separable. Some real-life applications aren't as tricky, in fact.