AI – Back propagation lab 2

Squaring the circle

In the prior lab, you used back propagation to find a neural network that would implement a specific, discrete function (a logic gate). The function that was meant to be implemented was completely specified, and as long as the NN got the indicated input / output pairs, we had no interest in what the NN did with any other input values.

The above type of usage is not typical for NNs. A more usual scenario is that one has an infinite number of possible input values where the expected output is known for a significant subset (perhaps all) of the input values. There is no way to train this network on all possible inputs. However, the great hope is that the NN will implement a function that has smooth transitions between the known points so that the NN will produce reasonable values for inputs that it is not trained on.

That NNs can do this is not immediately obvious. If we consider the example of fitting a polynomial of degree $N-1$ to N points, then as N goes up, the fitted polynomial will tend to oscillate violently between the fixed points. If you ask what the polynomial's value is between two adjacent fixed points, it may easily diverge substantially from the values at those two fixed points. It turns out that a well-trained neural network can converge nicely to a smooth function based on the input / output pairs that it is trained with.

For our second back propagation lab, we will explore this: you will find a neural network to detect whether a point in the plane falls into a specified circle centered about the origin. In particular, your script will be given a single argument – an inequality – in the form of

```
x*x+y*y>=1.0987
```

The number on the right, the square of the radius, will be in the range [.8, 1.2]. The inequality may be any of `>`, `>=`, `<`, or `<=`

Your lab is: **Use your own hand implemented back propagation code to derive and output a NN that detects whether a random point within the side length 3 square centered at the origin satisfies the inequality given to the script.**

If your NN outputs more than .5, it is construed that your network is averring that the inequality is satisfied; otherwise not. The way the grader will test this is that it will construct the network that your script specifies, throw 100,000 random points at it falling within the 3x3 square, and check to see whether the network makes the inequality determination correctly. You are looking to get a score of 99,000 or better out of 100,000.

The grader gives your script 100 seconds to construct the NN that you output. It will take the final NN that your script outputs, and timing out is OK. The points that it tests with are guaranteed to not be on the circle corresponding to the inequality (ie. you never have to worry about the equality aspect).

Tips and Comments

1) For both of the back propagation labs, there is no cause for rounding. We round when reporting values to other humans because usually there is no need for extended precision, but when only python is involved, it's just as happy with one real as another. Unless there is a specific reason for rounding, keep the weights and values as reals and don't round.

2) It was possible (though certainly not recommended) to implement NNFF and NNBP1 by hard-coding the neural network, meaning that each cell was being calculated explicitly. If this describes what you did, it becomes increasingly problematic to do this as the network size increases (which it will for this and future labs). It's especially a problem with this lab because in this lab it is likely that you will vary the network architecture to improve convergence, and that should be a really easy thing to do involving changing no more than a single line of code.

3) In the first back propagation lab, your dot product code for back propagation was not exercised (since the NN was too shallow). In this lab it will be exercised, and it is probably the single most likely place for a technical error. So, it's in your best interest to check that part of the code thoroughly if your network is not converging.

4) For updating the weights, you can do them after each run of back propagation, or you can also do them in batches of 100 (as suggested near the end of the 3rd video of 3 Blue 1 Brown's videos on NNs/back propagation – 19, 21, 14, 10 minutes, respectively):

https://www.youtube.com/watch?v=aircAruvnKk	What is a neural network?
https://www.youtube.com/watch?v=IHZwWFHWa-w	Gradient descent overview
https://www.youtube.com/watch?v=Ilg3gGewQ5U	Back propagation overview
https://www.youtube.com/watch?v=tIeHLnjs5U8	Back propagation calculus

Note: when watching this sequence, there are some terms that are different from ours:

- His σ is our logistic function
- MNIST is a large database of images of digits that have been labelled
- His Cost is our Error function
- RELU is another logistic function
- In the final video, his letters differ from ours:
  - His a is our x, his y is our t, and his z is our y.
  - And his weight indices are swapped from ours

5) While an adaptive alpha was not necessary for the first back propagation lab, you might find it useful here. The most straightforward thing is to make alpha proportional to the error, but you might find it more useful to relate it to a log or square root. At the beginning, as you are starting to experiment, you could see where your back propagation bogs down for your given alpha and then reduce it to some other fixed value at that point to see whether it makes a difference.

6) In the first back propagation lab, the total error was reasonably well defined. There were a certain number of input / output pairs, and you could take each one of them and compute the errors for each and then average. However, for this lab, you'll need to come up with your own measure of total error. Clearly, you can't just use a single in/out pair for the error, because your NN may be doing really great in one region, but not in a different one. One possibility is to have a fixed number of points (for example, on a grid, or perhaps spaced around the given circle) that you examine.

In my code, I generate test cases randomly, and I keep track of the error that I got for the last 100. My net error is the sum of the most recent 100 errors (note, I could average, by dividing by 100, but this just means a scaling of the termination criteria). Each time I try a new point and do the back propagation, I subtract off the oldest error in the list, (conceptually) pop that error, add in the new error, and append it to the list. This is fast and avoids 99 error calculations on each pass. Of course, these errors are not precise because I don't recalculate them based on the updated weight values, but I cross my fingers as hard as I can, squeeze my eyes shut, and hope very muchly that the error deviations will be small and that their errors will cancel each other out. This has proved to be sufficient. If I wanted to be more careful, I could every so often (say once for every 5000 values) completely redo my error list just in case some bias is affecting me.

7) Finally, consider a poor (wo)man's approach to the lab. They might consider that the circle given will be approximately a unit circle, whose area is π, while the area of the square that it's enclosed within is 9. Thus, the chance that a random point lands outside the circle is about 2/3. So the poor (wo)man will simply output a network that always outputs a 1 (for greater than inequalities) and get around 66%. This is akin to the weatherman in PDX saying every single day, "I predict it's going to rain." Your script should do significantly better this before you start to be excited.