SIMON D. LEVY

Computer Science Department, Washington and Lee University

CSCI 315 Assignment #3

Due 11:59PM Wednesday 15 February, via Github

Goal

The goal of this assignment is to use back-propagation on the problems we tackled in the previous assignment: Boolean functions and digit recognition. So you should be able to reuse a significant amount of code from that assignment.

Part 1

Copy / paste / modify your `perceptron.py` module into a new module `backprop.py`. This module should provide a class that you instantiate by specifying one extra parameter, h, the number of hidden units. Your `train` method should take an extra parameter, η (eta), specifying the learning rate, which you can default to 0.5. Use the algorithm at the end of the lecture slides to flesh out the `train` and `test` methods.

Once you've set up your backprop code, it should be straightforward to copy/paste/modify your `part1.py` from the previous assignment. Since the point of backprop is to learn functions like XOR, modify your code to train on this one function and report the results. Since we're using a squashing function rather than a hard threshold, you can simply report the floating-point value of the output (instead of True / False). A good result is one where you get no more than 0.2 for the False values and no less than 0.8 for the True. I was usually able to get results like this using three hidden units, η=0.5, and 10,000 iterations.

Once you've got your XOR solver working, add two methods to your backprop class: `save`, to save the current weights, and `load`, to load in a new set of weights. This will be essential when training larger, slower-learning networks like the ones in the rest of the assignment. You are free to implement these methods however you like, but I suggest using the Python

pickling tools you learned about in CSCI 111. (If you're rusty, take a look at slides 12-20 of Prof. Lambert's presentation of this topic.)

Part 2: 2-not-2 revisited

Now redo your Part 2 from last time: a 196-input, one-output backprop network that learns to distinguish between 2 and not-2. To get the misses and false positives, you can use a threshold. Ideally, you could consider an output below 0.5 as 0 and above 0.5 as 1. But I found this threshold too high, missing many of the 2's.

Of course, you'll have to experiment with a different number of hidden units (and possibly learning rate η) to get something you're happy with. Unlike the previous part, where you are almost certain to get good results on XOR with enough iterations, the goal here is not to "solve" the classification, but rather to *explore the behavior of back-prop on an interesting problem and report your results in a concise and understandable way.*

Once you're satisfied with your results on this part, use your `save` method to save the trained weights, and add some code at the end to `load` them, run your tests, and report your results. Once you've got this whole `part2.py` script working, comment-out the training part, so that the script simply loads the weights, tests with them, and reports the results. This is how I will test your script.

Part 3: Backprop as full digit classifier

Here we'll go for the "Full Monty" and try to classify all 10 digits correctly. Use your new backprop class to instantiate, train, and test a 196-input, 10-output network on a one-in-N ("one-hot") code for the digits. (This is the code at the bottom of each pattern, though it is easy to build yourself if you didn't read it from the data file.) For testing, you might simply pick the largest of the ten output units as the "winner".

Before you start training for lots of iterations here, I'd get your testing part of your `part3.py` code working: just train for one iteration, then run the tests and produce a 10×10 table (confusion matrix) showing each digit (row) and how many times it was classified as each digit (column). (A perfect solution would have all 250s on the diagonal of this table, but that is an extremely unlikely result.) Again, there's no "correct" number of hidden units, iterations, or the like. At some point you'll have to stick with something that works reasonably, and produce a nice table to report your results with it.

If you think about the number of weights you're now training ($197*h+(h+1)*10$ $197*h+(h+1)*10$), you can see why it will be crucial to *get your setup and report working nicely before you spend hours training*. As with Part 2, you'll save the weights once you're satisfied, then add code to load and test with them, and finally comment-out the training part.

Part 4 (Extra Credit)

For extra credit, see how creative you can get with visualizing and/or improving the training and testing of your network. Here are some suggestions. For the visualizations, I suggest Matplotlib.

1. Try using the momentum concept we discussed to improve training. If you get this to work, code up a little example that uses two different momentum values (a zero and a nonzero value) to demonstrate.
2. On each training iteration, compute the RMS error over your output unit(s), and display its progress at the end of the training run. For a neural net, this error is computed by squaring the Tj−OjTj−Oj vector, summing over the resulting vector, and accumulating this sum over the pp patterns. At the end of each iteration, you divide this sum by $p*m$ $p*m$ and take the square root of the resulting quotient. This value gives you the overall average of how poorly the network did on each part of each pattern.
3. Make a nice 3D visual presentation of the confusion matrix from Part 3. Think about what a perfect (no-confusion) solution would look like, and see how close your results are.
4. Using your XOR network from Part 1, create a 2D or 3D plot of the error surface based on the weights. You'll have to pick one or two weights to work with, and then produce some data representing the error (distance from the correct output) for various values of those weights.
5. Although we've been focusing on the weights, the values of the hidden units are often the key to understanding how a backprop network solves a given problem. Pick one of the three problems above on which your network has done a good job. Then report, visualize, and/or describe how the values of the hidden units help classify each pattern into a different category.

[*] Based on https://www.cs.colorado.edu/~mozer/Teaching/syllabi/DeepLearning2015/assignments/assignment3.html