CS380 Final Project:
Implementing a Neural Net in Objective-C

Elana Bell Bogdan

December 7, 2012

1 Overview of the project

Last summer, I began developing a social app for iOS, which centered around the fictional world of "Avatar: the Last Airbender" (henceforth ATLA). Among other features, the program would allow users to complete a personality test indicating in which "elemental tribe" (Earth, Fire, Air, or Water) they most belonged. The output was calculated as a linear combination of 4-tuples, each multiplied by a response scalar. The category corresponding to the highest value in the resulting tuple would then be displayed to the user. However, establishing the correct weights and parameters for each question proved a challenging task, and the "error rate" for the test (frequency at which it produced answers that mystified the user) was unacceptably high.

I concluded that some form of automated calibration would be valuable, but I didn't possess the right knowledge or skills at the time. However, in my more recent perusal of the voluminous text for this course,[1] I've discovered neural nets as an efficient, versatile method for performing precisely this sort of mapping between a finite set of numerical inputs and a handful of binary outputs. Because the rest of my app was coded in Objective-C, and because I had never previously heard neural nets mentioned in the context of iOS, I decided to try implementing one that could run natively on the mobile platform.

2 Approach

In determining how to go about this project, I consulted both the aforementioned text and a variety of online resources, including the StackOverflow forums and an old Usenet FAQ.[2]

[1]Russell & Norvig, *Artificial Intelligence: A Modern Approach*, the Prentice Hall Series in Artificial Intelligence. ISBN 0137903952

[2]http://www.faqs.org/faqs/ai-faq/neural-nets/

I also consulted briefly with students at Swarthmore College, who had applied neural nets (coded in Python) to the task of image recognition. Here I discuss some of the many considerations I encountered.

2.1 Network Topology

The most minimal neural net consists of just two layers, an input and an output, in a "feed-forward" relation. Every node in the input layer connects to every node in the output layer; no node connects back to itself; and it is possible to correctly categorize any set of data – *under the assumption that it is linearly separable.* However, certain, simple Boolean relations (notably XOR) violate this basic assumption, and, given my intention to process real-world, sociological data, I wanted to allow for that additional complexity. I therefore opted to incorporate a third (or "hidden") layer containing more nodes than the output later but fewer than the input; (see discussion below for precise determination of size). I also implemented bias (or "dummy") nodes in both the input and hidden layers, which always activate and enable the system to more effectively adjust its activation thresholds. Given the skew exhibited in response to certain input questions, (e.g. most people side left of center on the issue of whether to trust thoughts or feelings, a relevant personality trait in ATLA), this sort of calibration seemed important. The bias nodes receive weight adjustments from nodes in the next layer up, but they do not relay them to lower layers.

2.2 Weight-Adjustment Algorithm

Following the examples (and pseudo-code) in Russell & Norvig, I implemented a fairly standard, Back-Prop-style algorithm for comparing final predictions against known results, then relaying the differences back through the network. This method does require some basic calculus, but hard-coding the derivative of the activation function was a simple task, since I decided to go with a logistic curve, (the derivative of which, as noted in the text, has the property $g'(z) = g(z)(1 - g(z))$). The parameter that caused me the most trouble was the learning coefficient, since I didn't have a good intuition about what magnitude of values it would be multiplying. Ultimately, I just had to play around with it a lot as I tested the code (see below).

2.3 Training Methodology

Training a neural net typically involves feeding it the same set of inputs and outputs, as a batch (or "epoch"), repeatedly, until it begins to converge and stabilize (hopefully). Figuring out how to generate training data and how to determine specific criteria for termination is wherein lies the rub. For this project, I had two main sources of data: a survey which I managed to get ~ 70 ATLA fans to complete online, and data points generated artificially by my pre-existing app system. The value of the latter was that it constituted a definite, self-consistent pattern on which to train the net, while the former held more empirical weight. I therefore decided first to train within close tolerances on the modeled data, and then to try adjusting for the survey results. "Close tolerances" were defined in one of three ways: a maximum threshold for average error over a given epoch; a minimum-length streak of "correct" (after rounding) predictions; a minimum success rate of correct predictions (again with rounding) per set of epochs. My initial impulse was to reduce average error, but large epochs allow some egregious predictions to slip through the cracks when averaged, and I realized that the actual magnitude of the error doesn't matter very much, as long it's still possible to discern which ALTA category is dominant. I therefore switched over at a pretty early stage to focusing on uninterrupted streaks of overall success.

3 Implementation

I did the majority of my coding in two separate but closely integrated classes: `NNode.h/NNode.m` and `NNet.h/NNet.m`, (both of which I have included, along with inline comments, in this tarball). The latter is where most of the heavy-lifting happens, while the former is mostly a data structure with a few helper functions. These classes, in turn, are called from the main view controller for my app, where they have internal access to my pre-existing classes for question input and processing. It was here that I added several additional training (and parsing) methods, most of which I have extracted for inclusion in the tarball. Here are some brief notes on their uses:

- `-(int)streakTrain:(NNet *)neural withLength:(int)solidRun threshold:(float)maxError andMaxTries:(int)maxTries`

  Continues training until the difference between correct and predicted output has not exceeded `maxError` in at least `solidRun` trials (or `maxTries` has been hit); returns the number of trials at termination; currently configured to learn a sample Boolean set-up

- `-(int)errorTrain:(NNet *)neural withThreshold:(float)maxError andMaxTries:(int)maxTries`

  Averages error over each epoch (i.e. iteration through entire set of input data), attempting to bring below `maxError` threshold within `maxTries` epochs; configured to work with ATLA poll data

- `-(int)modelTrain:(NNet *)neural withEpochSize:(int)eSize errorRate:(float)maxRate andMaxTries:(int)maxTries`

  Runs random input through pre-existing personality model to generate training data in batches of `eSize`; continues running until the number of false predictions per batch is below `maxRate` (or, per usual, it hits the run limit)

4 Results

4.1 Performance with Booleans

Before applying my neural net to anything as complex as the ATLA data, I wanted to ensure that it could learn basic Boolean relations, especially XOR. Fortunately, I built the code to be flexible, so it was easy to generate nets with various numbers of layers and nodes per layer. I tested AND and OR with a {`2, 1`} topology, followed by XOR with both {`2, 2, 1`} and {`2, 3, 1`} configurations. (It may be worth noting that these numbers do not include the bias nodes that got added automatically.) I then randomized initial weights between nodes within the range $[-2, 2]$ and set a very low learning rate.

The fact that I was using a logistic activation function, rather than a simple threshold, may have impeded these tests to some extent, but the net mastered both AND and OR after being trained on around 200 random problems, (which took only a few milliseconds of run time). To my chagrin, XOR, on the other hand, repeatedly got stuck, generating indeterminate outputs of 0.5 to all input. It seemed as if the hidden layer was having no effect, and changing the learning rate offered no quick answers. However, upon thorough review of my code, it became apparent that my `calibrateInput` function, designed to work with the ATLA data, had been scaling my binary input down by a factor of 10. Theoretically, there's no reason why 0 and 0.1 couldn't be valid inputs, but, for whatever arithmetic reason, the downscaling was preventing convergence. As soon as I corrected this issue, the net began learning XOR after around 2000 trials in the {`2, 3, 1`} topology, and closer to 6000 on {`2, 2, 1`}. These figures both had a wide range of variation, which implied to me that the randomized starting conditions played an important role. I therefore found it useful to write a function for re-randomizing all weights within the net, triggered whenever a training loop had been running for too long. This strategy proved surprisingly successful, confirming my suspicion that getting started in the wrong rut (perhaps close to a local maximum) can be prohibitive.

4.2 Analysis of the ATLA Poll

While converting my data to CSV for ease of parsing, I noticed some initial areas of concern. Most visibly, certain categories of output (Air) were sorely underrepresented, while others (Water, and, to a lesser extent, Earth) dominated the sample. I was immediately worried that this disparity might cause the net to favor the better trained data disproportionately, and so I created a second table containing only subsets from the larger output groups. As an additional precaution, I then rearranged the row ordering in each table, so as to prevent clumping among output groups, (which I've read can have a negative impact of its own). Unfortunately, having to accommodate the weakest link, as it were, resulted in a quite significantly smaller table, which meant a correspondingly weaker statistical sample.

Of course, the statistical validity of the poll was already unclear, in light of the apparent similarity among the input distributions of different output groups. Mean and standard deviation of response to each question did not vary significantly by elemental identification, and it struck me that perhaps that ATLA was too broadly interpretable to produce consistent data... Nonetheless, neural nets often catch patterns that standard statistical analysis doesn't have a good way to handle, and so I proceeded.

The neural net definitely did achieve a certain degree of convergence initially, but, again and again, its rate of progress would start an asymptotic approach to zero very early on. The average error at this stable state proved highly dependent on both the number of nodes in the middle layer and the learning rate. After much trial and error, I determined their optimal values to be 9 (for an overall topology of {`15, 9, 4`}) and $\alpha = 20$ respectively, (although it's interesting to note that learning rates exceeding 20 tended to be highly ineffective). Regrettably, even after tens of thousands of epochs, the neural net was still effectively making random guesses. Nonetheless, this finding is interesting in and of itself, and it most likely does reflect on the inconsistent, idiosyncratic ways in which different people relate to media like ATLA.

4.3 Approximation of the Pre-existing Model

In marked contrast to its performance on the empirical data, my neural net proved extraordinarily proficient at learning the parameters and inequalities inherent to the personality test I'd written over the summer. Although it occasionally required up to 100,000 trials (or around 100 epochs of 1000 auto-generated samples), the algorithm managed to approximate the intended output within an arbitrary degree of accuracy. (I tested it down to under 20 mistakes per epoch, a success rate of 99.8%.) In this case, the optimal parameters turned out to be an $\{$`11, 7, 4`$\}$[3] topology at a learning rate of $\alpha = 2.5$. Although effectively converting this imperfect model into a neural net had not been my original goal, it was gratifying to accomplish all the same, and I've come to believe that it may actually indicate the best direction for my app as a whole.

5 Conclusion

This project yielded a bunch of unexpected conclusions about neural nets, along with a few more expected ones about my data set... Although a properly implemented, multi-layer neural net is mathematically guaranteed to converge over time *in theory*, its dimensions, parameters, and starting conditions can make a huge difference in practice. Choosing the right training methodology as well can help to establish reasonable expectations and useful interpretations for output. For example, depending on the activation function employed, demanding perfectly binary output may prolong training unnecessarily and possibly even lead to overfitting.

While overfitting is not inherently harmful when the fit is to an accurate model, my old personality test has never suited everyone, and I envision future users of my app wanting to adjust the test to suit their own purposes and notions. Hence my ultimate epiphany: evidently, ATLA *means* different things to different people, so why not write an app that can *be* different things to different people? Instead of just using the net to determine parameters for a large fan base that evades classification, why not put the net into the hands of individual fans, so they can interact with it as they use my app?

Preliminary tests indicate that, after learning the old model, my neural net can adjust to accommodate specific, contradictory feedback after being given that feedback between 2 and 2000 times. Depending on the extent of the overhaul required, such an update may decrease correspondence with the original model by anywhere from 40 to 210 matches per 1000 trials. Although the upper limits of these ranges may seem high, it's important to note that they reflect *arbitrary* feedback, which can include something as extreme as changing identification from Water to Fire. Tests on *likely* modifications (such as correcting Water to Earth) are almost exclusively near the lowermost limits of both, and occasionally happen instantaneously.

Thus, while training the entire net on the old model can take a minute or two, that step can be performed ahead of time, before my app even hits the AppStore, and these small modifications by end-users should be both snappy and effective, even on a mobile device.

[3] There were only 11 input nodes as a result of my having written the last four questions specifically for the recent poll, without time to incorporate them into the original model.