CS51 Final Project: Neural Networks

Andrew Sheeler, Evgenia Nitishinskaya, Maxwell Nye

1 Overview

Our original plan was to emulate Detexify, an app which takes in a handwritten symbol and outputs the corresponding LaTeX code. To that end, we constructed a feed-forward neural network with customizable geometry. We researched and implemented a back-propagation algorithm to train the network. Given training data, which consists of input data points and their desired outputs, the algorithm uses the difference between the network's output and the expected output to adjust the network weights via gradient descent. Our current code can automatically train on a large data set (assuming that samples are separated by newlines and the values in each sample by commas).

For the front-end, we used a drawing interface from the Tkinter module (with the help of some code from StackOverflow), and converted the data of a user drawing into a vector (borrowing open source code from Github) which the network could take in. It then outputs any combination of the top result(s) with or without probabilities and a graph showing the probability distribution.

Because networks take thousands of data points to train, we decided to train ours on 1797 handwritten digit samples we found on the UCI Machine Learning Repository.

Demo video can be found here.

2 Planning

Our original plan was to implement the network and algorithm, write a GUI and a ranking system, and apply this program to TeX symbols (possibly several at a time). We achieved most of this, except we restricted our attention to digits. The network would certainly function well as a LaTeX symbol identifier; however we didn't have access to a large database of LaTeX training data. We then revised and refined our plan to focus on the backend and optimization in the first week, and front end and interaction in the second. We actually ended up starting the front end during week one and leaving optimization until the end, because it was best to have the network be functional and usable before we started experimenting with different values.

Although our network cannot recognize TeX symbols, all that is missing is a training data set. Symbol recognition was our original motivation because Detexify is something we use regularly. We couldn't find any freely available sets of handwritten TeX symbols (Detexify is constantly trained by online user inputs), and 2 weeks is not enough time to generate that amount of data (even if we made web app, two weeks wouldn't have generated much). However, we accomplished our central goal and implemented a functional neural network in Python that could be applied to our desired problem (or any other identification task). The jump to classifying TeX symbols is now conceivable, while this task seemed insurmountable several weeks ago.

We mentioned several times that we were planning to implement a log-likelihood error function. We cut this because squared-error was working quite well, and we wanted to focus on making the network usable. We were also going to improve our visualization functions, but didn't bother because we didn't really use them much; Python's interactive mode includes sufficient built-in tools.

3 Design

Our code is written entirely in Python. We first considered MATLAB but decided against it because we wanted more flexibility and modularity, and we thought Python had everything we needed. We used the Spyder IDE which provides a variable explorer, function definitions, and easy visualization.

3.1 The Network

Figuring out how to allow a user to interact with a network was an interesting challenge. At its core, one Neural Network is simply a list of matrices and the associated structure, so we had to figure out a way to make it seem like an interactive object. To do this, we wrote files specifically for interaction independent of any ”network” object. Given a neural network, these files store the weight matrices which define a network, then execute a simple loop for interaction. A drawer file is called, generating a GUI with box that the user draws in. Our GUI saves the user input as a JPEG. Then our JPEG converter (a modification of a converter found on Github) runs, converts the image to a usable vector, and inputs that vector into the network. The result is printed in Python shell.

Our best trained network for digit classification has 2 hidden layers with 30 nodes each (the input was fixed at 64 neurons by the data and the output at 10 by virtue of us using decimal digits). This manages to correctly identify the digit in 1625 of the samples, and ranks the correct digit in the top 3 in 1767 of the cases. It is also quite successful in recognizing our handwriting. However, there is some ’overfitting,’ that is, the network is somewhat specialized to the somewhat limited range of inputs we’ve trained it on. We found that the network identifies certain numbers when drawn one way, and not when drawn another (e.g. 5 with a serif is recognized, but is not without). Our slightly-undertrained network (28 neurons in the hidden layers) ranks the correct digit in the top 3 for 1601 samples, and 1224 in 1st place. Both are included in the code as nNeuronNetwork.py ($n = 28$ or 30) and are user-friendly and interactive.

3.2 The Algorithms

3.2.1 Backpropagation

We were almost certain from the beginning that we wanted to use back-propagation. Reading online confirmed that this was a common choice for classification problems like ours. We ran into a small hiccup when our algorithm didn’t seem to be working as expected, but this turned out to be a result of a design decision which did not agree with the algorithm’s expectations. Namely, we put in weights before the first neurons (sigmoid functions). This meant that some of the activities the network used could be 1 or 0, something the algorithm assumes does not happen. Removing this first layer of weights solved the problem.

While writing the code, we experimented with the network manually. For example, to test back-propagation, we trained a network to output 0 when fed $(1, 0, 0)$. After we were satisfied that it worked as desired, we set up a tester that counted how much of our data set we could classify correctly, and also wrote a function that ran several networks with different parameters to select the optimal one. In doing so we assumed that the network’s behavior with, say, 32 nodes in a layer would be similar when there were 30, to reduce the number of cases we had to test.

An important part of a training procedure is the speed, i.e. how much the network changes in response to errors. We found that it was best to have a large speed (close to 1). If the same network is to be trained multiple times, it helped to steadily decrease the speed with each pass.

3.2.2 Image Conversion and Compression

Our GUI saves user input as a JPEG image. In order to input this into the network, we had to convert it into a vector. Because the conversion process is quite complex, we used open-source code from Github. Originally, this code was meant to take a JPEG and break it into a matrix of color codes (an $n \times n$) array of 4-tuples, corresponding to color intensity). We modified this code as follows: First, we converted the color-codes back into numbers. Then we eliminated all but the first element of the 4-tuple (because we're drawing in all black, the rest are redundant). Following the invariant of the training data we found online, we ran a small compression algorithm: we divided the array into 4 by 4 chunks, and created a new array with each element corresponding to one "chunk," with value equal to the sum of the values in the chunk. This serves as both a compression and a blurring, which aids the network's speed and accuracy.

4 Reflection

Neural networks are fickle things. In order to train correctly, they need lots of testing data, which can only realistically be obtained from the internet. However, unless the programmer takes specific precautions, a network will take almost every aspect of the data into account. In other words, the network creates its own hidden expectations of what the data will look like, and will behave poorly if these standards are not followed. This makes preprocessing very important, and quite tricky.

We encountered a lot of problems where the network would get stuck, or not update correctly, or the performance was subpar. After closely examining the input data, we would realize we weren't following a hidden invariant, and have to rewrite some code. We ended up editing our normalization, vector orientation, compression algorithm, and marker width (we realized by inspecting the numbers that the training data had really fat numbers), to name a few. We also realized Python saved some data across runs which we needed to clear manually. At one point we made an extremely overfitted network by training it too many times, and had to discard it.

From the research we've done, it seems that the neural network can't get any better without being a lot more complicated. It does a whole lot better than random, so we're happy with it.

Although we knew about neural networks before, we didn't really understand them until we had to implement them. We spent many hours discussing the algorithm and the design before starting the project. We also learned a lot of Python. We found Python's flexibility and loose rules (especially with types for function inputs/outputs) useful for the occasional need for special case handling, but the careful programming we learned from using OCaml helped us avoid the nastier errors Python users might encounter from lazy programming.

5 Advice for Future Students

Stay in school. Don't do drugs, unless it's for science.

It's important to find a problem you care about. Think of something you really want to exist, or think of something you've always wanted to learn about. The 'tedious' parts of your project (swimming through cryptic errors, write ups for your TFs ;)) will be worth it. In our case, combined interests in neuroscience, NLP, Python, and TeX gave rise to our idea.

Also, plan extensively before you ever start coding. Especially in our group, as we had little practical neural network knowledge, it was very important to discuss design and implementation. Because there are so many ways to implement a network, it was important to agree early on how we wanted to go about it. We also needed to understand the theory well before writing any code.