Handwritten Digit Recognizer

Bryan Rodriguez

Sean Goldthwaite

Andrew Fornash

For this project, we wanted to create a neural network that had the ability to take in a handwritten digit it had never seen before and make its best guess on what that digit is. The biggest challenges we faced throughout this project revolve a lot around the fact that this is the first time any of us have worked with neural networks. As such, we all have spent much of this time trying to fully understand concepts such as feedforward and backpropagation, as well as the vast amounts of linear algebra that goes into this kind of project. From our research, we have found that, in addition to the neural network solution, there are numerous types of ways to solve this problem, such as kNN-based solutions and SVMs. However, since this problem of handwritten digit recognition is classed as the 'Hello World' of machine learning, we have decided to tackle it in two of the most common ways: a fully connected neural network, and a convolutional neural network.

For the fully connected neural network, we are using the two main algorithms associated: feedforward and backpropagation. Feedforward is the much simpler of the two algorithms, so we shall begin with that. Given a one-dimensional array of inputs that is of size equivalent to the number of input nodes, this array turns into a matrix of size # of input nodesx1. Next, the dot product of this new input matrix is calculated with the matrix of the randomized weights between the input and hidden nodes. This dot product will then be added to the biases connected to the

same hidden nodes to give us the hidden node's matrix. Finally, this matrix is put through an activation function, in this case, the sigmoid function, and send it on its way to be used in the dot product of the output matrix. This cycle repeats for the output matrix, before finally giving an output of what the computer is thinking about that input.

Backpropagation is given both an input and an answers array, both of which are of the size of their respective node type in the neural network. The input array is put into the feedforward algorithm to get the outputs from it, before the answers are turned into a matrix in a similar way that the inputs were. The error between the output and the real answers is calculated by subtracting the answers matrix with the outputs matrix, before the gradients for the output node are calculated by taking the outputs mapped through the derivative of the sigmoid function and performing a Hadamard product with the output errors. The gradients are then adjusted with a dot product with the neural network's learning rate, before being used to calculate the deltas between the hidden and output nodes. These deltas are added to the weights between the hidden and outputs, before the biases are also added to their deltas, which just so happens to be the gradients calculated originally. This process then repeats as many times as there are layers in the neural network.

Fully connected neural networks are a very primitive form of neural networks, since they were originally studied through the 70s and 80s. As such, they suffer from a lot of issues with time complexity and space complexity. Although the map is very shaky on the exact big O notation for time and space complexity, a well-used estimate is in the $O(2^n)$ range, but this notation is very variable, due to the number of nodes and layers a neural network may have, and the space complexity may be anywhere from O(n) for a very simplistic network to O(n!) and potentially even higher. Although there is no theoretical limitation to fully connected neural

networks, the fact that they have such a gigantic time and space complexity may make them significantly more unwieldy to use for certain projects, so we have also done research into a much more modern approach to neural networks: convolutional neural networks.

Although much of the math is similar to fully connected neural networks, convolutional neural networks end up being much more efficient due to the number one difference between the two: the lack of connections going from every code to every other node. Although this would seem as if it would create a much stupider network, due to the lack of overall neural connections, calculations are much faster. Therefore, it is possible to cram significantly more calculations into a convolutional neural network in the same amount of time to get a significantly better result. This also helps significantly with both the time complexity and the space complexity. Although there is a good case for the general estimate of the time complexity for convolutional neural networks to be $O(2^n)$, the same as a fully connected neural network, in practice the actual time used to compute the training of one is significantly smaller. Its space complexity, however, may be worse than a fully connected neural network, as a cnn needs a lot of a data, perhaps even more than an fcnn. Depending on the amount of data used, it may also become more of a victim to overfitting than an fcnn. Overall, however, provided it has enough data to not suffer from overfitting, it tends to be an upgrade from an fcnn.

We as humans think in a very interesting way: whenever we think about anything, several neurons fire out at once into other neurons, which fire into other neurons. This cycle continues for thousands, if not tens of thousands, of times before something happens, whether that be a random thought appearing in your head or you quickly pulling your hand away from a burning hot stove. Regardless of the situation, some stimulus, an input, causes a chain reaction of neurons to fire, hidden nodes, which eventually leads to an action, an output. After all, neural networks

were directly modeled to be loosely based on the human brain. What is not so obvious, however, is the way that humans learn and how it relates to what a neural network is doing. Imagine a situation where you know absolutely nothing about what a dog or a cat is, and you are asked to classify multiple pictures of ones. You would just guess for each picture before being told what the correct answer is. Now that you know the correct answer, you can attempt to make some sort of mental classification of what to pick on when classifying such images. Eventually, after going through the data an ungodly amount of times, having practically memorized every single detail that you could have from those images and how they relate to a cat vs a dog, you are presented with a new image that you haven't seen before. But you pick up on the flatter face, the longer whiskers, the bigger eyes, and you decide that this image you have never seen before is a cat, and you would be correct. You picked up on how to do the task from having zero prior experience to now practically being a master, outside of some very strange outliers. A similar process happens on a much smaller scale in a lot of workplace environments or learning new skills for a hobby.

Although we have not had time to formally put our neural networks through hours to days of training, so we cannot yet say how the final result will turn out, we can give a small result based on a few tests we did on a much smaller scale:

Percentage of times digit guessed correctly from a 3-layer neural network with 10 minutes of training:

0	0%
1	100%
2	0%
3	0%
4	0%
5	0%
6	20%
7	0%
8	0%
9	0%

Here we can see the network almost completely unsure of itself, as nearly every single guess it made was a 1. Useful if the digit was a 1, but not so useful otherwise. Interestingly, despite the low amount of time, the network has picked up on a few hints on how to determine a 6. Although it is far from perfect, it is very interesting to see that it was able to pick up on something like that so early.

Percentage of times digit guessed correctly from a 3-layer neural network with 40 minutes of training:

0	0%
1	90%
2	0%
3	0%
4	0%
5	0%
6	40%
7	20%
8	0%
9	0%

Within half an hour, we can see the network beginning to make very slow progress away from everything being a 1, as now it is beginning to pick up on some other details with other numbers, including 7, which is a very 1-like number. However, this is coming at the cost of it no

longer believing that every 1 is a 1. It seems to have also gotten slightly more insight into how to tell a 6 apart from any other number. It is surprising, then, that it has yet to pick up on 5, a number that is somewhat similar to 6 shape-wise.

That was as much data as we were able to extract from our short tests, but this is going to be subject to change when we are able to fully train our networks.

The biggest improvement to be made is the fact that the fully connected neural network is only three layers, which may be limiting its usability in the long run and may even cap its success rate in this current task. Furthermore, the way the gradients were calculated for the furry connected neural network is not particularly efficient. Instead, a way of coding in a stochastic gradient descent, a way to calculate gradients in batches, rather than every single time we call for a backpropagation, could potentially allow a significant performance increase. Overall, however, this project has been very interesting to work on, and has taught us a lot in terms of how machines learn. Furthermore, the lessons learnt on just how difficult implementing either of our neural networks has given us new-found respect for those who have figured out how to manage this long before any of us were even born.