# Project 3 Writeup

For this project, I implemented a neural network from scratch, following closely the Python/NumPy code found here: https://www.kaggle.com/code/wwsalmon/simple-mnist-nn-from-scratch-numpy-no-tf-keras/notebook. The problem our neural network is trying to learn is to classify handwritten digits from the MNIST dataset (which you can download here: https://yann.lecun.com/exdb/mnist/). We use a package (in mnist.go) to load this data in. Each image is 28x28 pixels, but is represented as a 784 element long array. Our network contains 3 layers - one input, one hidden, and one output. We write a set of matrix operations (mimicking operations found in NumPy, albeit far less optimized), and operations for gradient descent, forward propagation, and back propagation - the tools that ultimately allow our network to learn a set of weights and biases that can make accurate inferences on unseen test data.

An early disclaimer: I regretfully misallocated my time a bit on this project, focusing too much on the sequential version of the network, leaving insufficient time for the parallelism. I'm happy to say that the neural network does converge successfully with high accuracy (80%+) after ~100 epochs with the sequential version. However, even though my code compiles for both parallel versions, and I'm satisfied with the logic I've implemented for both the work stealing and work balancing programs, the final accuracy for both is no better than guessing (~10%). I believe there are two main possibilities for why this is: 1) the way in which I'm attempting to parallelize training the network is theoretically wrong or 2) I have a bug where I'm combining the results of the individual threads.

Option 2 is likely (see the above disclaimer!), but Option 1 is more interesting, and allows me to discuss how I parallelized the task. The technique I employ is a form of ensemble learning: I'm training many small models, and at the end, averaging the results of each to form a final set of weights and biases. In this case, each "task" our executors create is one of these small models which only contains a subset of the data (in our case, 1000 of the 60000 total images). With a smaller sample size per model, we have more variance in accuracy, but our hope is that averaging the outputs of all 60 models minimizes this variance to converge to something like we'd expect out of our sequential model, i.e. when we train our model on all 60000 images at once. Why did our parallelism fail? It's possible that this assumption that 1000 images is enough to train our model is false - our very simple 3 layer network might not have been able to pick up on anything more than noise with only 1000 data points. Or, taking the averages of a set of weights and biases doesn't really make sense…

A large part of my difficulties in debugging this network were due to the fact that there is not really a deterministic output: the weights and biases are initialized randomly, and the closest thing to determinism that we can get is seeing a steady increase in accuracy regardless of the random initialization. Thus, even though parallelizing the task in the fashion I did shouldn't theoretically be extremely difficult since all I'm doing is creating smaller, independent models, and then averaging their results, figuring out exactly what was wrong with the network is extremely difficult. For example, I spent 3 whole days wondering why my accuracy wasn't converging, debugging line by line, before simply realizing that I had never normalized my inputs!

I also considered other forms of parallelization, but determine they would be too difficult given the time constraint. For example, I could have parallelized individual matrix operations, perhaps in a divide and conquer fashion, similarly to how (I believe) NumPy executes matrix operations. These matrix operations are a particularly large hotspot - each one is a double for loop at the minimum, and could be parallelized similar to how we discussed in Lecture 7. Additionally, I could have implemented "one weird trick for parallelizing convolutional neural networks" (https://arxiv.org/abs/1404.5997), in which there are interactions between the different workers (i.e. "each worker must send the gradient for each example to the worker which generated that example in the forward pass" (page 3)). Parallelizing neural networks, I learned, is hard! Implementing all of the above would have certainly resulted in a more optimal implementation.

Two important bottlenecks - apart from the matrix operations - are 1) where we load in the data, and 2) where we aggregate the results at the end. Loading the data isn't particularly slow (takes a few seconds). We use a 3rd party package to do this, as mentioned above. Aggregating the results of our different models is definitely something that can be parallelized, but is not, currently.

However, our form of parallelism is, theoretically, extremely effective in reducing our largest bottle neck - the matrix operations. The time complexity of a matrix multiplication, for example, is cut down by a linear factor. Multiplying two matrices with 1000 datapoints is 60x faster than multiply two with 60000 datapoints.

Our work stealing and work balancing algorithms both did not show considerable speed up in initial testing. I did write a speedup.py file but do not have time to wait for it to finish processing before it generates the graphs! You can run it the typical way using the slurm file in the benchmark folder.

Theoretically, however, I would not expect our work stealing algorithm to perform much differently than our work balancing algorithm in terms of speed up. This is because our tasks are all of the same size and each thread is assigned the same number of tasks initially, so we expect each thread to finish at approximately the same time, giving little opportunity to steal or balance.

I ran testing on my MacBook Air with an Apple M2 chip and 8 GB memory.