Neural Network using NumPy

A neural network was developed from scratch using purely NumPy. It was designed in a way to allow the user to be able to specify the neural network's architecture by customizing the number of hidden layers, number of nodes for each layer, and the activation functions used for each layer. The architecture would be specified like the following:

```
nn_architecture = [
    {"input_nodes": 784, "activation": "relu"},
    {"input_nodes": 64, "activation": "relu"},
    {"input_nodes": 10, "activation": "softmax"},
]
```

Here, the first and last layers are the input and output nodes respectively. For the input layer, because our dataset consists of images of fashion items of size 28x28 pixels (784 pixels), the number of nodes for the input layer needs to be 784 to match this. The activation function does not matter in this case.

Our dataset, Fashion MINST, being a multi-class classification problem, means that the last layer needs to have a softmax activation layer as softmax will produce a probability distribution for N classes whereas functions such as Sigmoid or ReLU can only do binary classification. Our dataset has 10 different classes of items, so Softmax is required and the number of nodes in the output layer also must match the number of classes, hence the output layer having 10 nodes. This architecture can have as many hidden layers added to or taken away from it as required, the underlying code is vectorized and so will be able to handle any type of specified architecture.

In addition to this, the neural network uses Stochastic Gradient Descent with mini-batching to train/adjust the weights of the neural network. This has the main benefit of smoothing out fluctuations in the weights as an average is taken instead of just one sample. It also can significantly reduce the time required to train the neural network which was very helpful during the hyperparameter tuning phase of this task. The size of the batches can be specified when calling the train() function.

Other parameters that can be changed include the epochs, learning rate for the gradient descent algorithm, and the seed used for randomization (such as weight initialization and mini-batch shuffling). The labels in the dataset have been one-hot encoded and the pixel information in the feature dataset have been normalized.

At the end of every epoch, the metrics (accuracy, loss) of the neural network is computed by using the current weights to predict values and comparing them with the test dataset. The accuracy used to evaluate parameters for the neural network is computed by getting the mean of all correct and incorrect predictions of the test dataset. The loss function used is Categorical Cross Entropy.

Hyperparameter Tuning

We took the hyperparameter tuning in a few steps. First, we tuned the learning rate to get one that was suitable for our dataset and neural network. Next, we tuned the mini-batch size (using the tuned learning rate). Then we tuned the NN architecture (using the tuned learning rate and mini-

batch size). Both the learning rate and mini-batch size were tuned using the neural network architecture presented at the top of this page.

Learning Rate Tuning

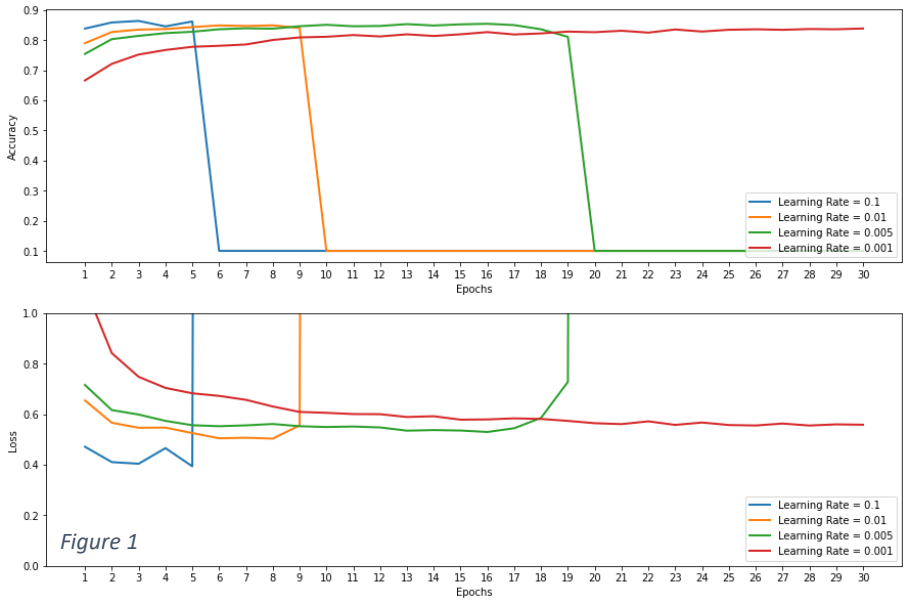

Figure 1

The information in the plot in Figure 1 shows the accuracy and loss value of the neural network model over 30 epochs using different learning rates. Interestingly, any learning rate tested above 0.001 will eventually lead to vanishing/exploding gradients. After doing some research as to why this might happen, we have concluded that this is a very common issue with deep neural networks. Some solutions to this problem we could implement in the future could be Gradient Clipping, which limits the size of the gradients calculated from the backwards propagation stage, better weight initialization, and weight regularization. In the remaining time that we had, we decided to implement better weight initialization. Initially, we were just creating the neural network with random weights but decided to use He weight initialization instead which works very well for layers with ReLU activation functions. Using a learning rate of 0.001 is not ideal as it took longer to train the model over 30 epochs, however, this is something we hope mini-batching can mitigate in my case. A final insight from Figure 1 is that it seems after around 20-25 epochs the metrics start to stabilize and instead fluctuate back and forth. It will be best to implement an early stopping criterion at this point to prevent overfitting of the training dataset.

Mini Batch Tuning

After tuning the learning rate, we decided to use a rate of 0.001 as that was the only reasonable performer from Figure 1.

It can be seen from Figure 2 that the size of the mini-batch had some effect on the metrics but not significantly. The main point of interest is that as the mini-batch size increases, stochasticity of the gradient descent decreases and produces less choppy weight updates (however in the case of batch size 4096 it has had a very negative effect of having the accuracy overshoot and undershoot). If looking at that alone it can be seen that a batch size of 2048 would be the most suitable and it would also significantly reduce the training time and make optimal use of the GPU parallelism that can be provided by GPU accelerated NumPy libraries such as CuPy and Numba. However, a very large batch size will also mean that you will have to train the model over more epochs to achieve the same level of accuracy.

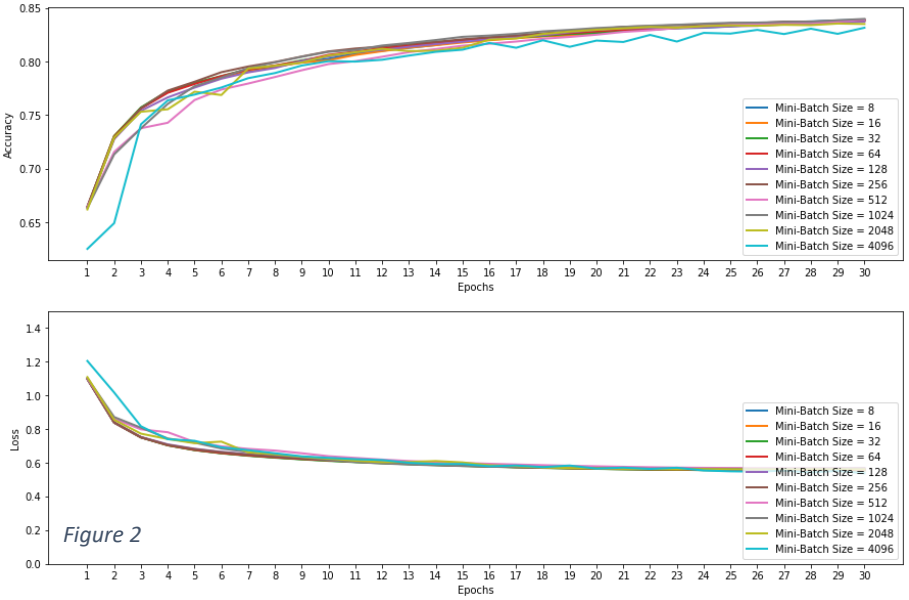

Figure 2

Whether this would cancel out the time gained from GPU parallelism depends on a case-by-case basis and is hard to justify. Along with that, a larger batch size can make it harder to "jump" out of a local minima if it gets stuck in there; how big of a problem that depends on your dataset and problem domain. Finally, a smaller batch size can cause the model to settle in flatter minima which can generally mean better generalization. Overall, after examining this plot and doing some research, we have decided to use a batch size of 128.

Neural Network Architecture Tuning

The final piece of hyperparameter tuning is finding the optimal neural network architecture. The different architectures we have decided to test are listed in Appendix A:

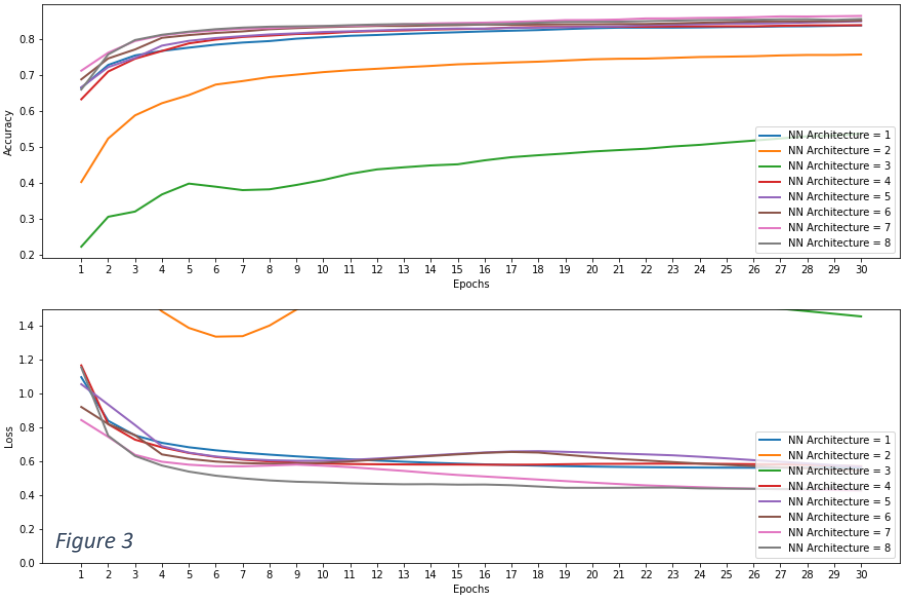

Figure 3

From first inspections of Figure 3, it is obvious architecture 2 and 3 produce terrible results. These architectures use sigmoid as their activation functions and so it can be deduced that for our dataset and problem domain, sigmoid is not a suitable activation function to use. So, all the other neural network architectures are using ReLU as their activation functions. From those, we can observe that generally the deeper and more nodes per layer, the better the performance. It seems from the plot that architecture 8 produces the lowest loss metric and highest accuracy of 85%.

In conclusion, for a simple ANN, a final accuracy in the range of 85% is very good. Looking at other neural networks online, it seems an accuracy of around 95-97% can be achieved with the use of a Convolutional Neural Network which is better suited to this problem domain (computer vision).

Appendix A: Supplementary Figure

```
nn_architectures = [[
    {"input_nodes": 784, "activation": "relu"},
    {"input_nodes": 64, "activation": "relu"},
    {"input_nodes": 10, "activation": "softmax"},
],[
    {"input_nodes": 784, "activation": "sigmoid"},
    {"input_nodes": 64, "activation": "sigmoid"},
    {"input_nodes": 10, "activation": "softmax"},
],[
    {"input_nodes": 784, "activation": "sigmoid"},
    {"input_nodes": 128, "activation": "sigmoid"},
    {"input_nodes": 64, "activation": "sigmoid"},
    {"input_nodes": 10, "activation": "softmax"},
],[
    {"input_nodes": 784, "activation": "relu"},
    {"input_nodes": 64, "activation": "relu"},
    {"input_nodes": 64, "activation": "relu"},
    {"input_nodes": 10, "activation": "softmax"},
],[
    {"input_nodes": 784, "activation": "relu"},
    {"input_nodes": 128, "activation": "relu"},
    {"input_nodes": 64, "activation": "relu"},
    {"input_nodes": 10, "activation": "softmax"},
],[
    {"input_nodes": 784, "activation": "relu"},
    {"input_nodes": 512, "activation": "relu"},
    {"input_nodes": 64, "activation": "relu"},
    {"input_nodes": 10, "activation": "softmax"},
],[
    {"input_nodes": 784, "activation": "relu"},
    {"input_nodes": 512, "activation": "relu"},
    {"input_nodes": 128, "activation": "relu"},
    {"input_nodes": 64, "activation": "relu"},
    {"input_nodes": 10, "activation": "softmax"},
],[
    {"input_nodes": 784, "activation": "relu"},
    {"input_nodes": 512, "activation": "relu"},
    {"input_nodes": 256, "activation": "relu"},
    {"input_nodes": 128, "activation": "relu"},
    {"input_nodes": 64, "activation": "relu"},
    {"input_nodes": 10, "activation": "softmax"},
]]
```