Tasks 3 & 4 Report

Programming & Mathematics for AI

Hui Zheng
Emre Dogan
Éamonn Ó Cearnaigh

GitHub Repository https://github.com/EamonnOCearnaigh/NeuralNetworks

Task 3: Neural Network using NumPy

A neural network was developed from scratch using purely NumPy. It was designed to allowthe 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 numberof nodes for the input layer needs to be 784 to match this. The activation function does not matter in this case.

As our dataset, Fashion MINST, is a multi-class classification problem, this means that the last layer needs to have a softmax activation layer. This is because 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. The number of nodes in the output layer must also match the number of classes, hence the output layer has 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 descentalgorithm, 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 usedto 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 networkarchitecture presented at the top of this page.

Learning Rate Tuning

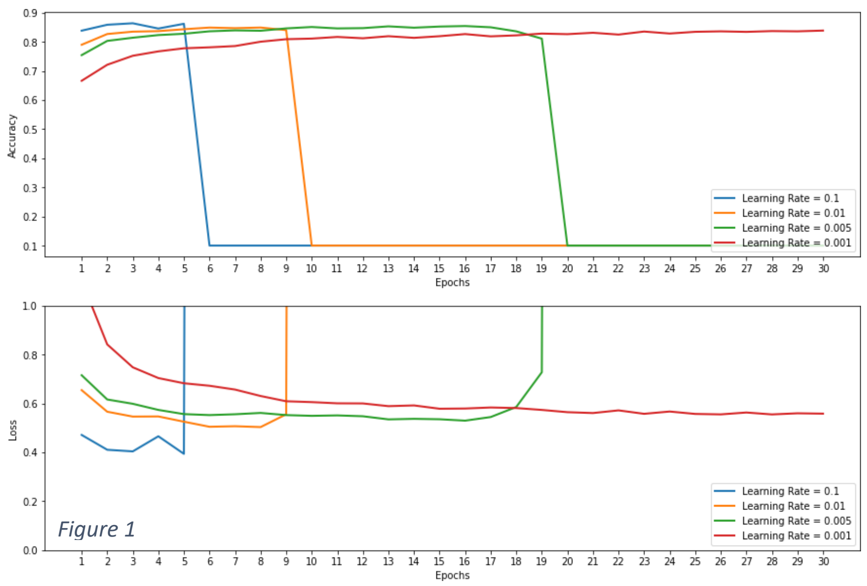


Figure 1

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 that we could implement in the future could be Gradient Clipping. This limits the size of the gradients calculated from the backward 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 we later decided to use the weight initialization instead, which works very well for layers with ReLU activation functions. We used a learning rate of 0.001 but this was not ideal as it took longer to train the model over 30 epochs. However, this is something we hope mini-batching can mitigate in this case. A final insight from Figure 1 is that it seems that after around 20-25 epochs the metrics start to stabilize instead of fluctuating 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, the size of the mini-batch had some effect on the metrics but this effect was not significant. The main point of interest is that as the mini-batch size increases, the stochasticity of the gradient descent decreases and produces less choppy weight updates. (However, in the case of batch size 4096 it had a very negative effect of having the accuracy overshoot and undershoot).

Looking at that alone, it can be seen that a batch size of 2048 would be the most suitable. 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 the model has to be trained over more epochs to achieve the same level of accuracy.

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 minimum if it gets stuck in there; how big of a problem that

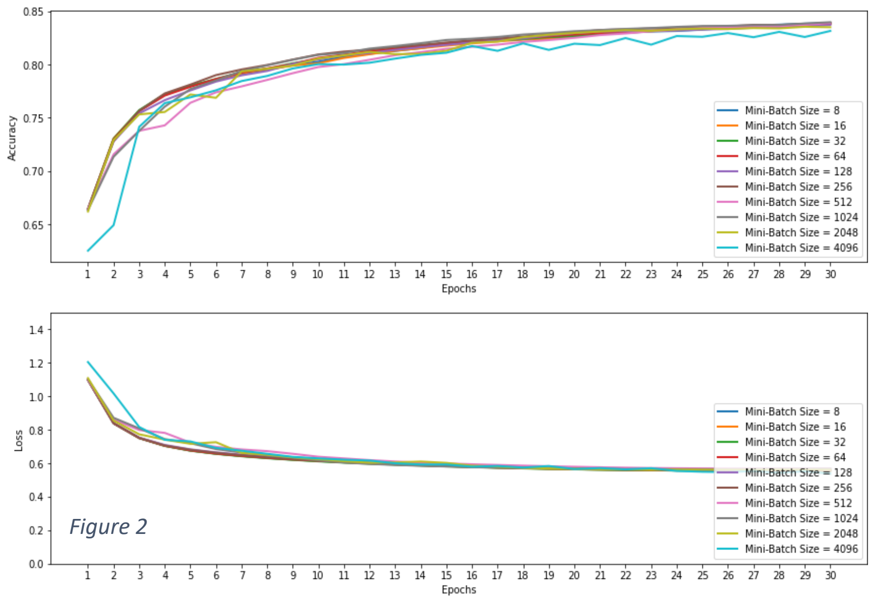


Figure 2

poses, depends on one's 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 researching and examining Figure 2, we 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:

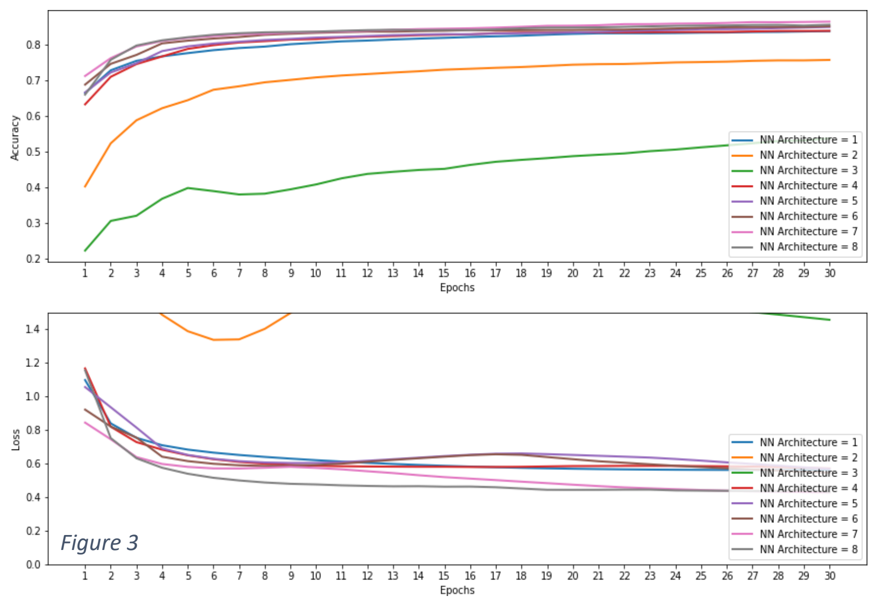


Figure 3

From initial inspections of Figure 3, it is obvious that architectures 2 and 3 produce terrible results. These architectures use sigmoid as their activation functions, 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 otherneural networks online, it seems an accuracy of around 95-97% can be achieved with the use of aConvolutional Neural Network which is better suited to this problem domain (computer vision).

Task 4: Classification using PyTorch

Implementing Neural Network (Baseline)

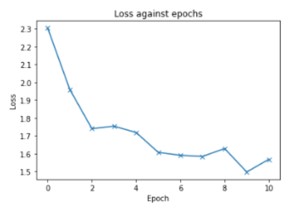
Using PyTorch, NumPy, and Matplotlib, we implemented a neural network to classify images from the CIFAR-10 dataset. The dataset was loaded using torchvision. With an initial training and testing allocation of 50,000 and 10,000 images respectively, we further allocated 5,000 images from the training set to a new validation set. This resulted in 45,000 images for training, 5,000 for validation, and 10,000 for testing.

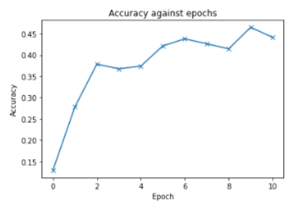
After doing research on common parameters, we established baseline parameters including batch size 128, 10 epochs, learning rate 0.1, and 2 workers for data loading. We used a stochastic gradient descent optimiser. All of these would later be compared against alternative arrangements.

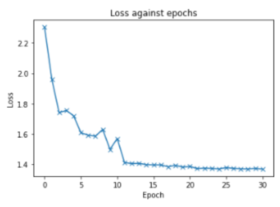
The neural network's functionality was distributed across several Python functions. Primarily, the neural network class used involved functions to train the model on batches of images, validate batches, and then evaluate the results of a given epoch. Output was printed describing the model's progress during runs.

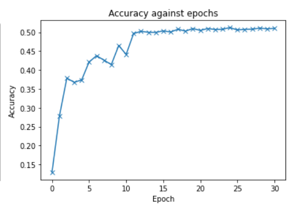
The baseline configuration involved a neural network with an input layer, one hidden layer, and an output layer. A feed-forward function flattened images into vectors, and processed the images through the layers and their associated activation functions. We used Rectified Linear Units activation functions as they are said to perform better than sigmoid and other popular choices. Relu learns nonlinear functions and rectifies the vanishing gradient problem.

We produced results after 10 epochs at a learning rate of 0.1 (left), but also played around with decreasing this further within the same history. We also tried running 30 epochs by applying 10-epoch runs with decrementing learning rates (0.1, 0.01, 0.001). The latter gradient is relatively unstable. Accuracy, in general, is low, with final accuracy after 30 epochs being only 50%. Results plateau from epoch 10.







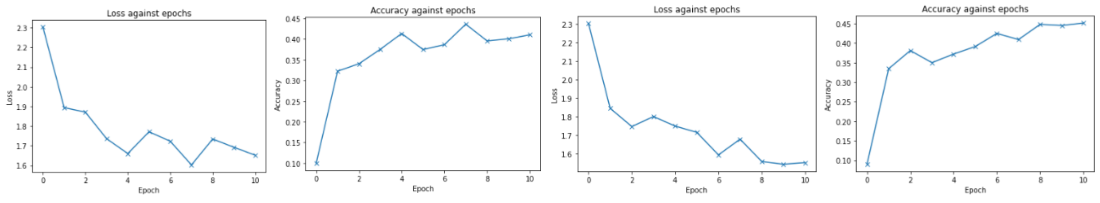


Proposing Improvements

We used an additional dropout layer with two different variations in dropout probability to explore the effects on the baseline. We also implemented L2 regularisation via the weight_decay parameter in the optimiser function.

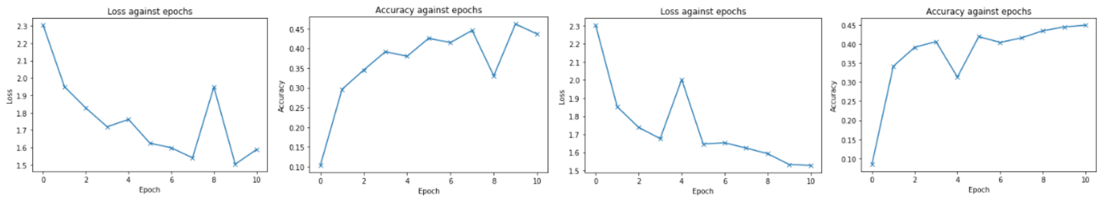
Dropout Layer

A dropout layer helps to reduce overfitting by randomly selecting a percentage of input units to set to 0 during training. The dropout layer (dropout probability 25% left, 15% right) was added to the baseline model with the following results. We found the latter to produce a slightly more optimised result. Final accuracy was higher and the gradient slightly less volatile.



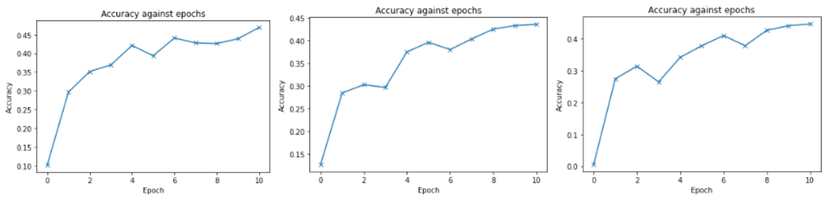
L2 Regularisation

L2 regularisation reduces overfitting by shrinking parameter estimates, which simplifies the model. This can be done via the weight_decay parameter in the optimiser function. This parameter was set to 1e-5 (left) and then reset to 1e-4 (right) in two tests.



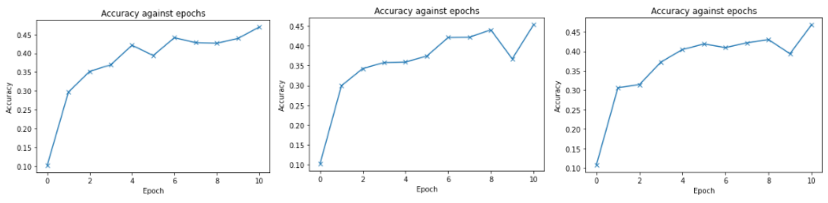
The observed result appeared to disrupt accuracy quite significantly, while the overall gradients improved.

Number of Hidden Layers: 1 (baseline), 2, 3



This determines the number of layers the data passes through while traversing the network. We examined the effect of adding one and then two layers to the baseline model. The baseline appears to perform best, with fewer drops in accuracy than the more complex alternatives.

Number of nodes in Hidden Layer: 256 (baseline), 128, 512

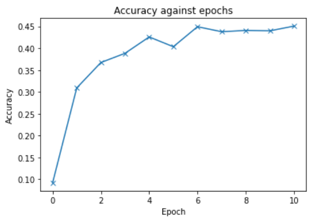


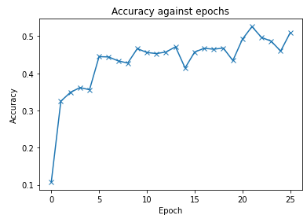
Although we expected the models with more nodes to produce better results, the baseline outperformed the alternatives.

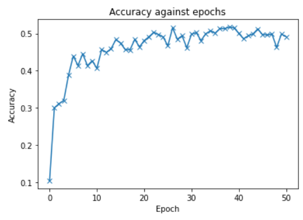
Evaluating Parameters

All parameters listed here compared the baseline mode, using accuracy as the primary metric. See notebook for full details. The main goals were to aim for higher accuracy and avoid overfitting. **For each parameter, we selected the best result relative to the baseline for the final models (see conclusion below).**

Number of Epochs: 10 (baseline), 25, 50

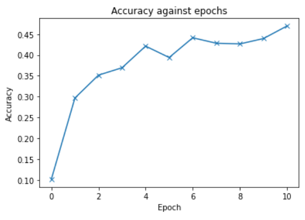


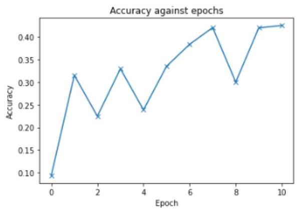


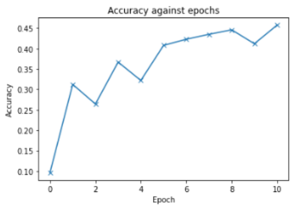


The more epochs, the longer the model's training stage. Increasing the number of epochs has the potential to increase accuracy but risks overtraining the model. 25 epochs shows an increased accuracy, but the unstable nature of the gradient shows signs of overfitting. 50 epochs definitely displays overfitting due to the volatile gradient observed.

Batch Size: 128 (baseline), 64, 256

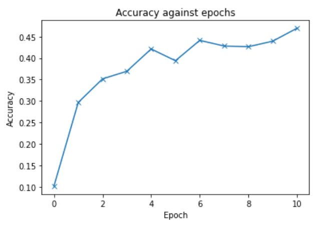


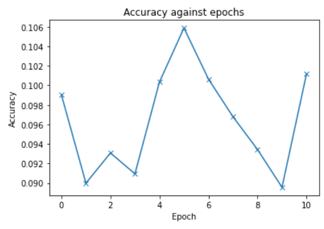




The batch size determines the number of images loaded into the network at once. The baseline outperforms both lower and higher batch sizes with a much smoother gradient. Decreasing and increasing the batch size both made the gradient less stable and likely reduced generalisation. It is likely that the models became too sensitive to local minima.

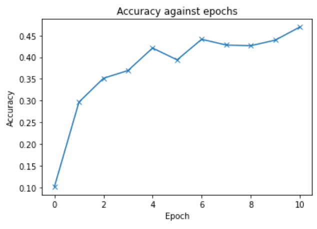
Optimisation Function: Stochastic Gradient Descent (baseline), Adam

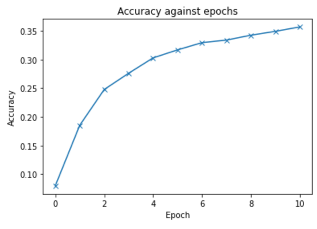


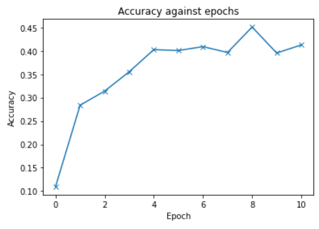


SGD performs much better than Adam, which suffered a significant and unstable drop in accuracy. This is likely due to SGD generalising more effectively than Adam.

Learning Rate: 0.1 (baseline), 0.01, 0.001





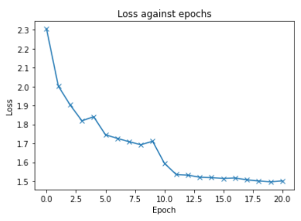


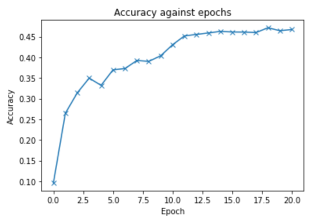
Smaller learning rates than the baseline seem to improve generalisation but at the expense of accuracy, as observed. Learning rate 0.01 has the smoothest gradient, but lower accuracy than 0.1.

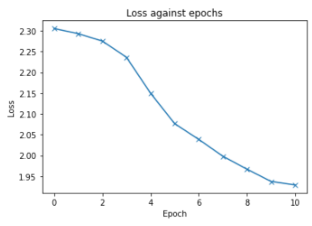
Conclusion: Final Models

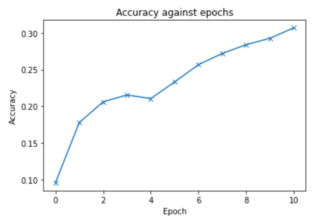
A dropout layer with dropout probability 0.15 was added to the baseline model in order to produce the final neural network. This was because 0.15 performed relatively better than 0.25 during the above tests. An additional hidden layer was added, bringing the total to two. Nodes in the hidden layers were set to 256 based on previous tests.

Based on performance relative to the baseline, we decided to select the following parameters: 10 epochs (in order to minimise overfitting risk), batch size 128 (significantly stabler gradient), and SGD rather than Adam optimisation function (significantly better performance). Learning rates of 0.1 and 0.01 were initially run in series (left), totalling 20 epochs as this proved to have a seemingly positive effect on the results in the baseline. We also provided an alternative version (right) using only 0.01 as it previously resulted in a significantly smoother gradient than other models.









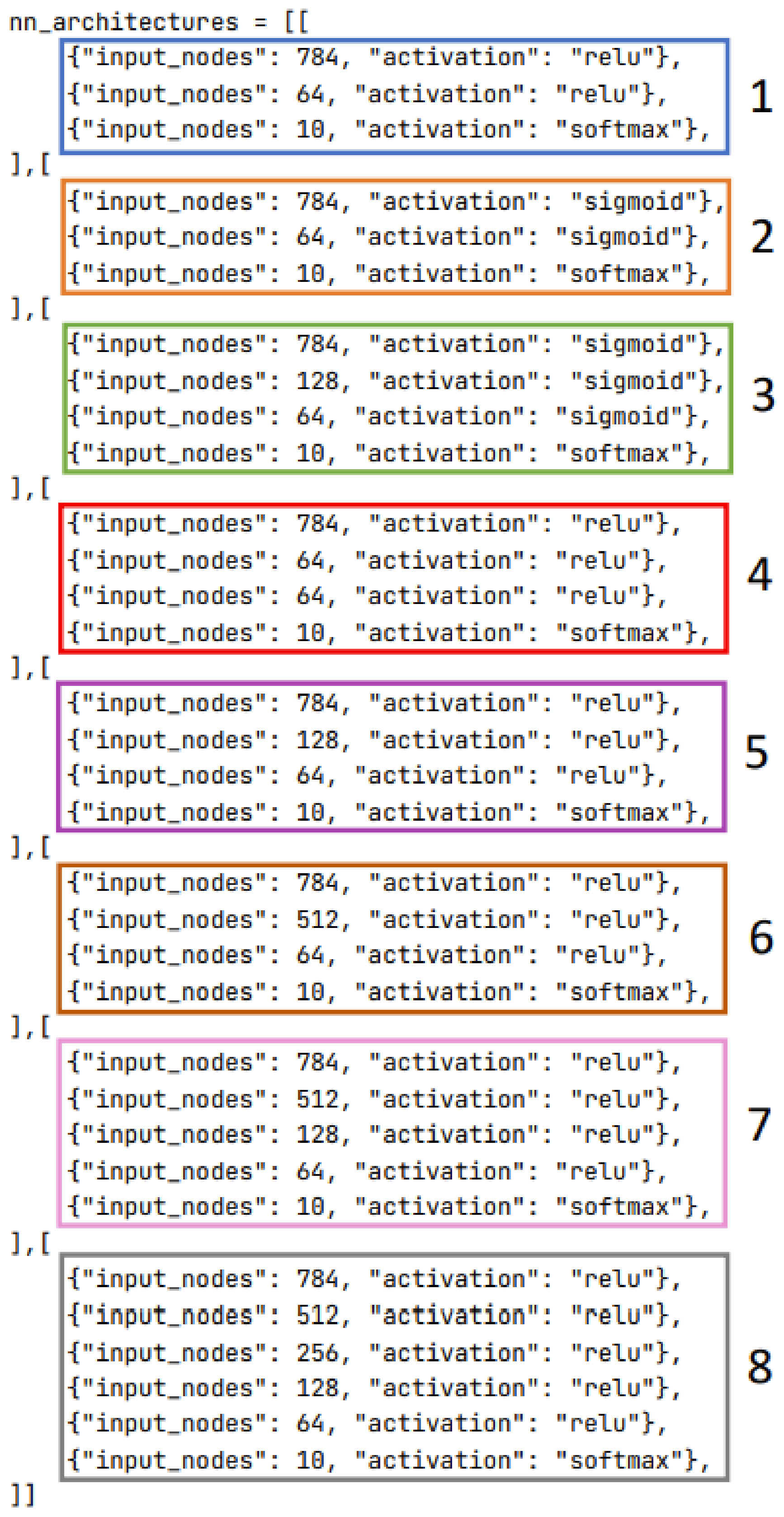
Effects of the combined additions and changes to the baseline model were unfortunately minimal regarding accuracy but the gradient was significantly less volatile than previous models.

For future work, we would re-attempt this project using a convolutional neural network to improve accuracy. We would also provide a greater focus on performance metrics rather than just loss and accuracy, in order to examine different aspects of the models. We could also examine more optimiser functions like AdaGrad and RMSProp. More variation and different intervals (0.05 for example) in learning rate could provide more insight into its effect on accuracy. We underestimated the complexity of the problem dataset, as we briefly read of many sources struggling to raise accuracy much further than our final models.

Reflections

The members in this team managed the project so that from the beginning, the work was allocated to each person. Initially, each person worked on his/her respective tasks individually, but we soon realized that we could achieve better results if we worked together at the same time. This led to weekly meetings, either physically at the University, or through Teams. We maintained a Google Doc throughout the project, as well as several GitHub repositories in order to view each other's work. These weekly meetings allowed us to brainstorm ideas and exchange thoughts, so that as we worked our way through the tasks, each person provided inputs and contributions to all the tasks. This merging of all our efforts made the final output a product of collaborative teamwork and synergy.

Appendix A: Task 3 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"},
]]
```

References

~20% of code for both tasks 3 and 4 is directly inspired by online resources. The rest we attribute to lectures, labs, and existing machine learning knowledge.

PyTorch (2021) Available at: https://pytorch.org/ (Accessed: December 2021)

Upgrad (2021) Available at: https://www.upgrad.com/blog/basic-cnn-architecture/ (Accessed: December 2021)

Github (2021) Convolutional Neural Networks. Available at: https://github.com/martinoywa/cifar10-cnn-exercise/blob/master/cifar10_cnn_exercise.ipynb (Accessed: December 2021)

Towards Data Science (2021) About Train, Validation and Test Sets in Machine Learning. Available at: https://towardsdatascience.com/train-validation-and-test-sets-72cb40cba9e7 (Accessed: December 2021)

Towards Data Science (2021) Neural Network using NumPy. Available at: https://towardsdatascience.com/lets-code-a-neural-network-in-plain-numpy-ae7e74410795 (Accessed: December 2021)

ML from scratch (2021) Available at: https://mlfromscratch.com/neural-network-tutorial/#/ (Accessed: December 2021)

Kaggle (2021) Available at: https://www.kaggle.com/accepteddoge/fashion-mnist-with-numpy-neural-networks (Accessed: December 2021)

ML Cheatsheet (2021) Activation functions https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html#relu (Accessed December 2021)