# Assignment Report

Mitchell Chatterjee
Carleton University (101141206)

The intent of this assignment was to implement backpropagation on a simple network and compare these results to an automated differentiation package. In the second part of the assignment, we explored the effects of using batch normalization and dropout techniques on a Neural Network. We also compared the accuracy and test loss of three different types of networks: CNN, MLP, and Softmax while modulating the aforementioned hyperparameters.

Note that many of the model architectures chosen in Question 2 were inspired by the architectures listed here: http://yann.lecun.com/exdb/mnist/.

The Pytorch package was used in order to complete the second part of this assignment.

## Question 1

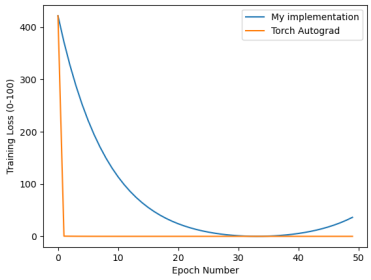


*Figure 0: A comparison of the training loss between my implementation of backpropagation and that of the Torch Autograd library*

In Part 1 we implemented a computational graph library in order to support the computation of the forward and backward passes in the program execution. In order to do so, the program was built up from scratch by implementing a library that built a computational graph using custom implemented computation nodes that computed their output and gradients based on their input values. These nodes were then organized into a graph which controlled the execution of the program. In the forward pass, the input was propagated through the nodes in the graph to produce the output, this was accomplished by computing the postorder traversal of the graph in order to attain the correct execution sequence. While during the backpropagation, the nodes were collected based on a breadth-first search. The gradient at each node was then computed by multiplying the upstream gradient, by the local gradient value.

As we can see from *Figure 0*. My implementation of a computational graph succeeded in computing the same forward pass as the Torch Autograd library. However, my implementation was far inferior in updating the weights of A, B, and C efficiently. Moreover, my implementation

eventually overcorrected and began to diverge from the optimal solution. However, my implementation did successfully achieve updates to the model weights via backpropagation, approaching the optimal value attained by the Torch Autograd library.

*Note: The matrix multiplication method was not implemented completely. I did leave my initial code commented out in order to show my process. However, I ran into too many bugs and was forced to use numpy for this one operation. All other operations are implemented from scratch.*

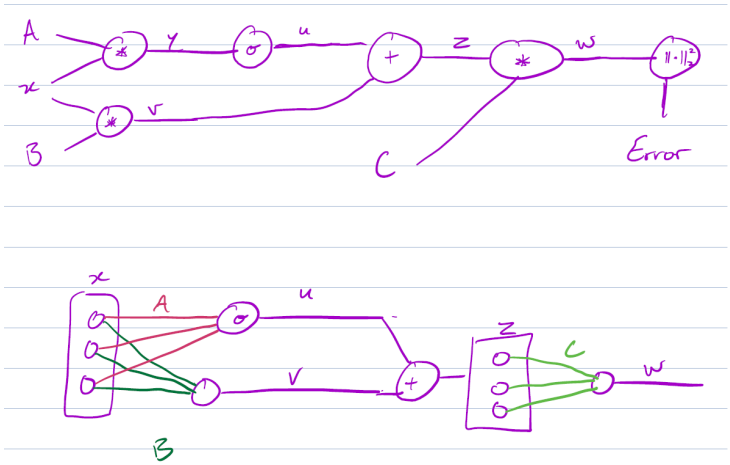


*Figure 1: The computational graph depicting the steps required in order to compute the forward pass.*

## Question 2

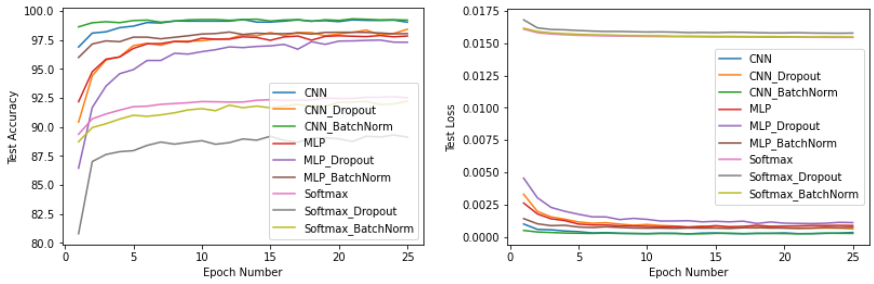


*Figure 2: The output from all networks with and without batch normalization / dropout.*

In question two we compared the results of three different network structures. Namely: CNN, MLP, and Softmax. The structure of these networks can be seen in the following output. Note that the batch normalization can be seen in all networks. Batch normalization was turned on and off through a boolean flag in the forward pass.

Furthermore, we modulated the hyperparameters of these networks to include dropout and batch normalization. Dropout was only applied on the second layer (and thereafter) of every network, except for the Softmax classifier that required it to be placed in the first layer. This was to ensure that we did not lose vital input data from the initial layers.

Batch normalization was applied in a similar way. To all layers beyond the input layer. This ensured that we did not normalize the input data. However, it helped to smooth the gradient allowing the model to learn more efficiently.

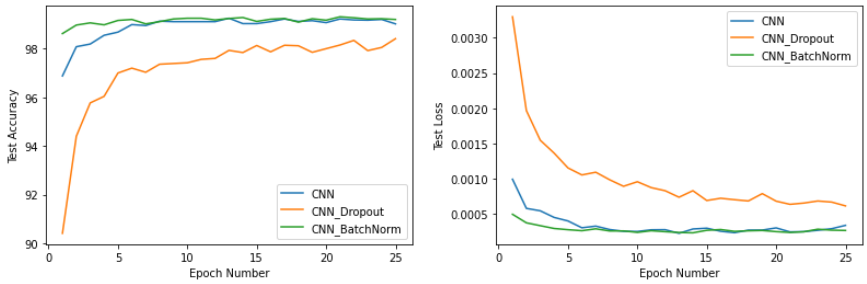


*Figure 3: A closer look at CNNs with and without batch normalization / dropout.*

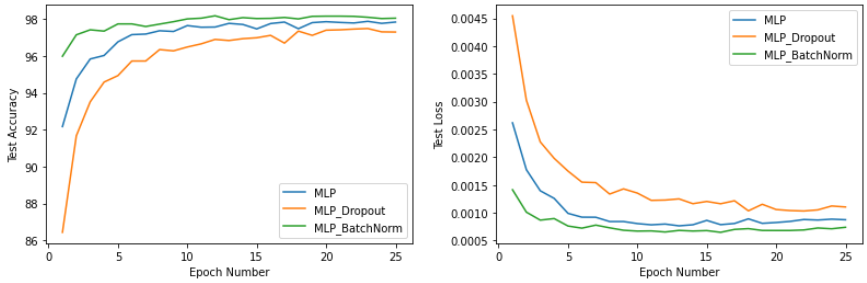


*Figure 4: A closer look at MLPs with and without batch normalization / dropout.*

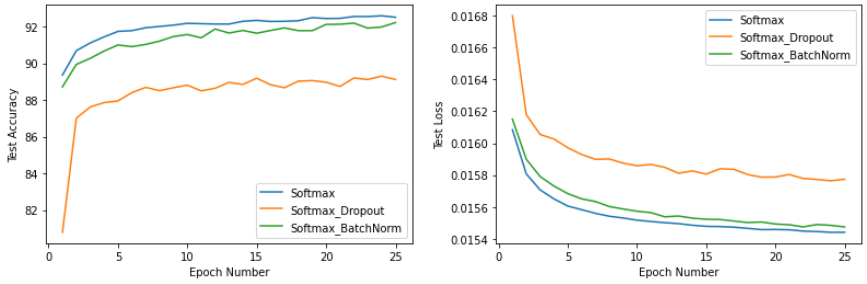


*Figure 5: A closer look at Softmax with and without batch normalization / dropout.*

As we can see from the data, dropout did not improve the performance of any classifier. Both in regard to Test Accuracy and Loss. However, this is likely due to the simplicity of the task and the lack of training epochs. The simplicity of the task allowed most classifiers to quickly converge to a nearly perfect test accuracy while simultaneously approaching a test loss of zero. This can also be attributed to the low number of epochs, as the model did not have a chance to overfit to the data. Instead all models without dropout retained a high generalization accuracy. If dropout were applied to a more complex task, and given greater training time, it is likely that we would see an improvement in the model test accuracy as a result of the regularizing effect of dropout.

Batch normalization did improve the performance of all classifiers when applied to the data. This can be attributed to its smoothing effect on the landscape and gradient, which

allowed the model to train more efficiently[1]. As a result, the models using batch normalization converged to a near perfect accuracy much faster than the other networks.

[1] Santurkar et al, “How Does Batch Normalization Help Optimization?” NIPS 2018