Multi-Layer Perceptron

Karan Sehgal (2016CSB1080)

CSL603: Machine Learning | Lab -3

Introduction- The goal of the assignment is to predict steering angle of the road image for a self-driving car application using multilayer perceptrons. The dataset has been compiled from Udacity's simulator. The original images have been transformed into 32x32 grayscale images.

I. Preprocessing

A. Normalization

Before training, images have been normalized such that the minimum pixel value in an image is 0 and the maximum pixel value is 1.

B. Standardization

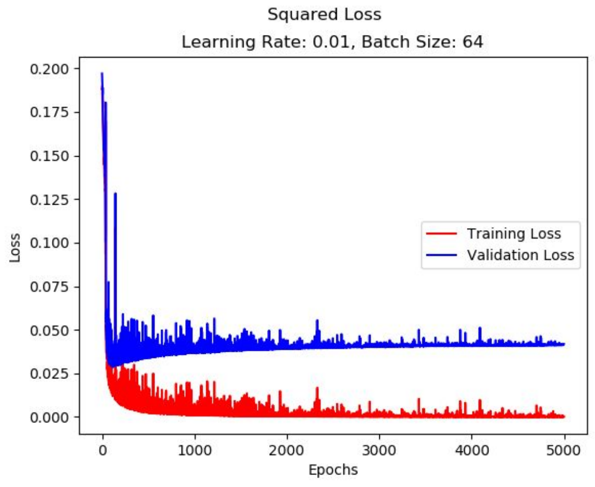
Since in a basic neural network each instance should have some D attributes, 32x32 image has been flatten into a vector of size 1024. Later each attribute has been standardized by subtracting it with the mean and dividing it by the standard deviation of the attribute with respect to the training set.

To compute the training/validation loss, we took the squared mean error/2 value across the entire training/validation set.

Loss: $\frac{1}{2} \Sigma (O-Y)^2 / N$

II. Experiment 1

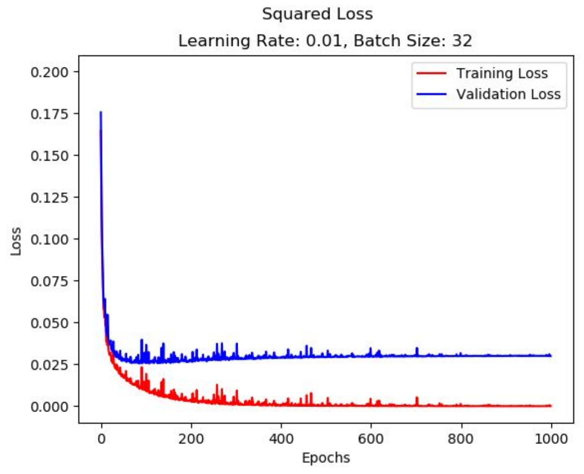
Hyperparameters:
Learning Rate: 0.01 | Batch Size: 64 | Epochs: 5000
Architecture:
Dense Layer: 1024 - 512 (sigmoid)
Dense Layer: 512 - 64 (sigmoid)
Dense Layer: 64 - 1 (no activation)

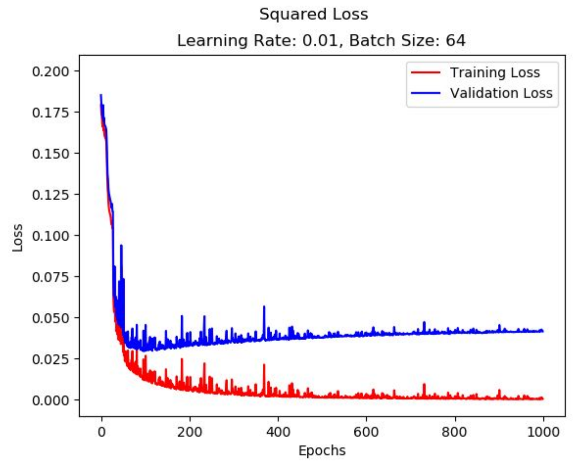


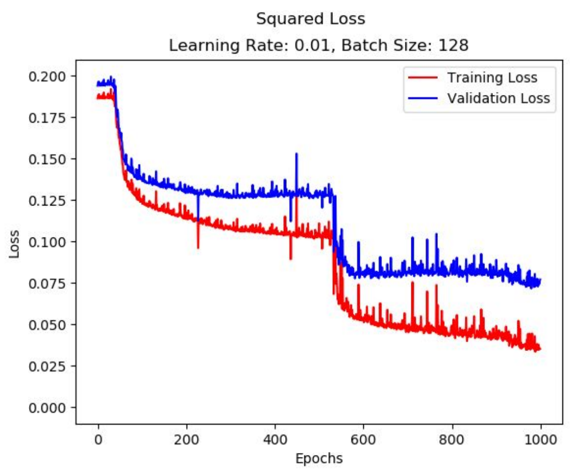
For the first few epochs, on an average, both the training and validation loss were decreasing. After around 120 epochs, the validation loss started increasing while the training loss continued to decrease. The minimum validation loss was around 0.028 while the minimum training loss was around 0.0003. This result tells us that the model has overfitted the data.
High fluctuations can be seen in the graph, which might be because of a high learning rate.

III. Experiment 2

Hyperparameters:
Learning Rate: 0.01 | Batch Size: 32/64/128 | Epochs: 1000
Architecture:
Dense Layer: 1024 - 512 (sigmoid)
Dense Layer: 512 - 64 (sigmoid)
Dense Layer: 64 - 1 (no activation)







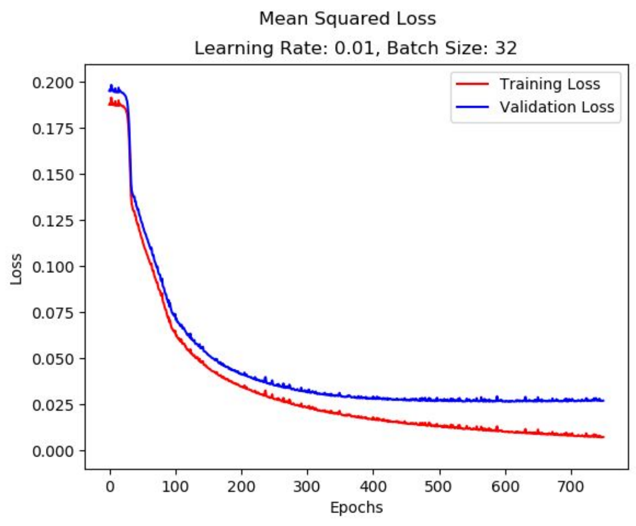
A. Common Characteristics

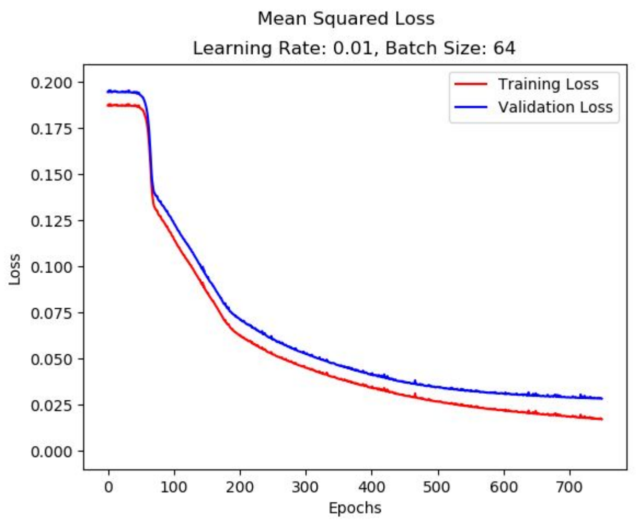
In all the three graphs, we could see the model overfitting the data. Without regularization, neural networks tend to overfit the data.

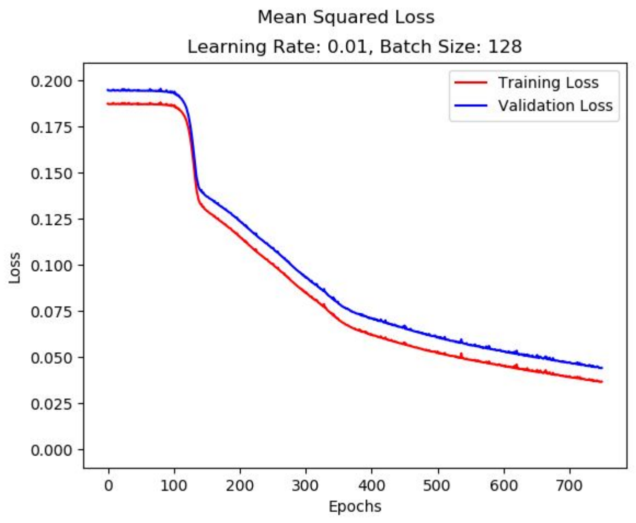
B. Effect of Batch Size

As the batch size increases, the rate of convergence decreases. This is because as we increase our batch size, the model considers more training instances at once hence is more likely to go in the right direction but take large steps towards it, since the gradient strength is large (since it is not averaged by the batch size). The number of steps per epoch decreases with increase in batch size, since we run the backpropagation lesser number of times. Which might make the model less generalized. We can see this in the third graph, where the model oscillates between epochs 50 to 500 because of a higher batch size and takes more time to converge as compared to the models with lower batch size.

It was also seen that if the gradients are averaged by the batch size, larger batch size results in a smoother graph, since the step size is smaller. Those graphs are shown below for 750 epochs.



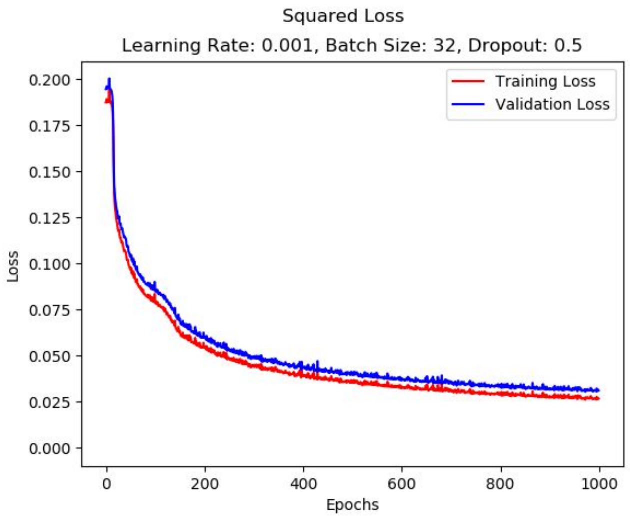




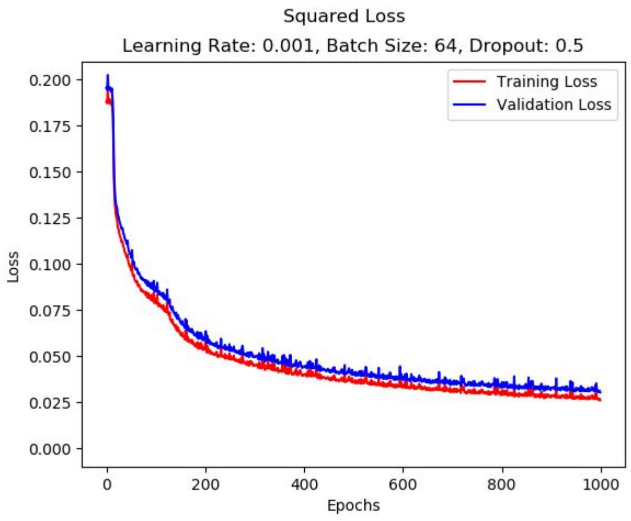
IV. Experiment 3

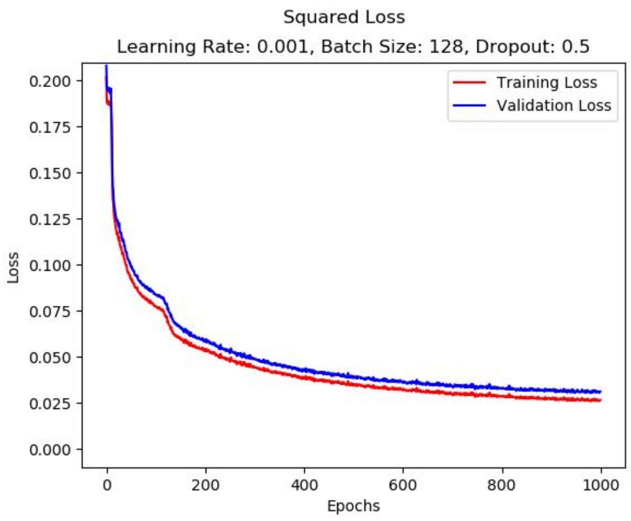
Hyperparameters:
Learning Rate: 0.001 | Batch Size: 32 | Epochs: 1000
Architecture:
Dropout(keep_prob=0.5)
Dense Layer: 1024 - 512 (sigmoid)
Dropout(keep_prob=0.5)
Dense Layer: 512 - 64 (sigmoid)
Dropout(keep_prob=0.5)
Dense Layer: 64 - 1 (no activation)

As we can see from the graph, our overfitting problem has been resolved by using dropouts as our regularization technique. However, our convergence rate has also decreased.



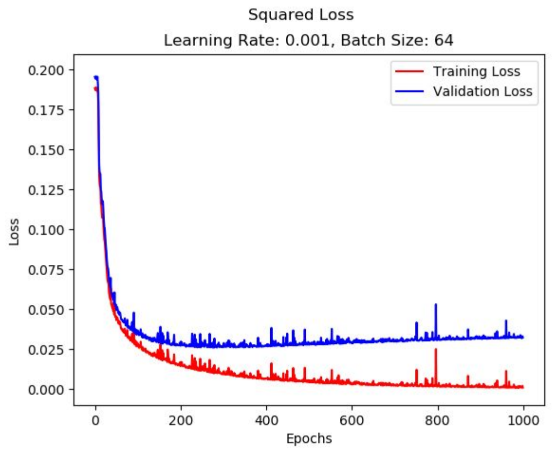
.

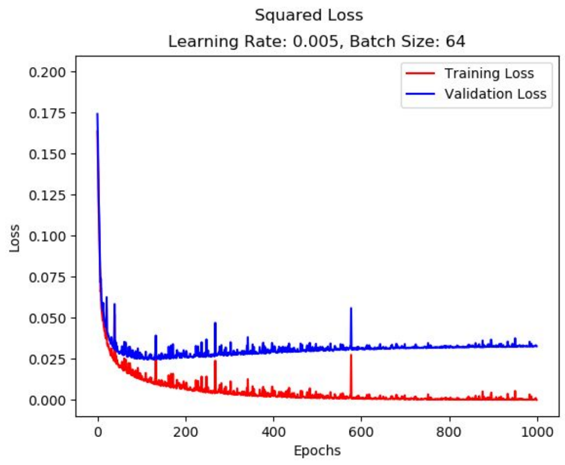


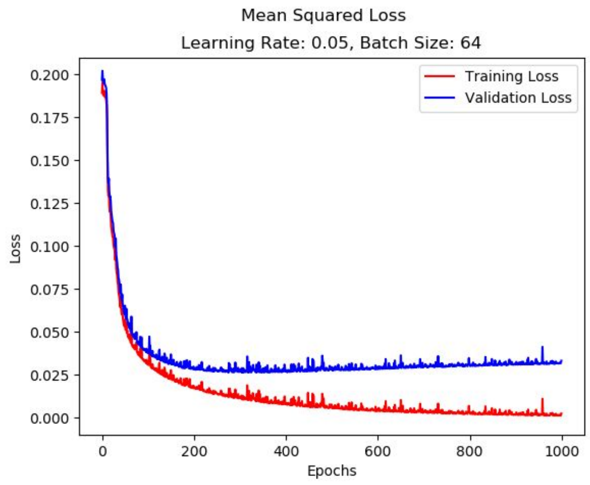


V. Experiment 4

Hyperparameters:
Learning Rate: 0.001/0.005/0.05 | Batch Size: 64 | Epochs: 1000
Architecture:
Dense Layer: 1024 - 512 (sigmoid)
Dense Layer: 512 - 64 (sigmoid)
Dense Layer: 64 - 1 (no activation)







In the first two graphs, gradients are not averaged by the batch size, whereas in the third graph gradients are averaged by the batch size. This is because otherwise, the learning rate of 0.05 was too high for the model, the loss went to infinity and eventually NaN.

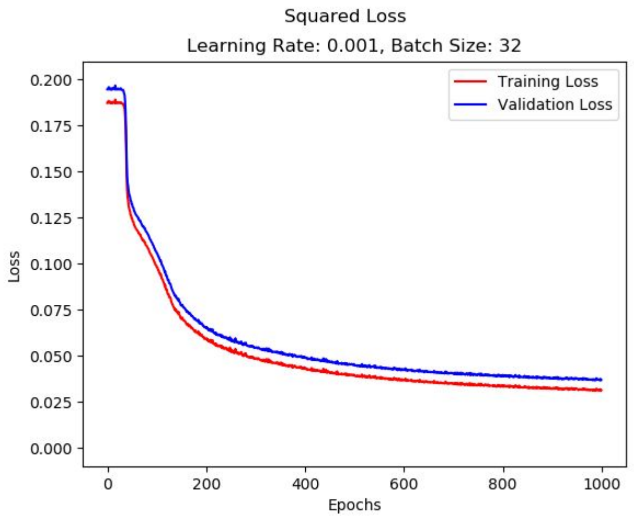
As we can see a lower learning rate results in comparatively lesser amount of fluctuations because step size is small, whereas with a higher learning learning rate, the convergence is faster. In the graph we can see that the model with learning rate 0.005 reaches a lower validation loss faster than a model with learning rate 0.001.

VI. Bonus

For the bonus section, I experimented with a deeper network, ADAM optimizer, different activation functions and CNNs.

A. Adding More Layers

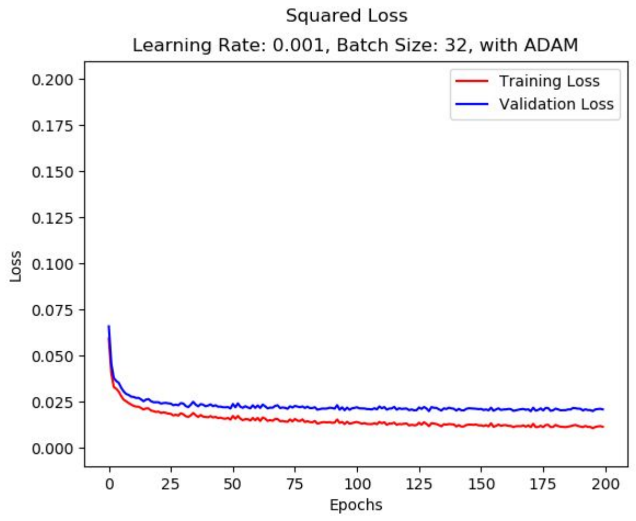
Hyperparameters:
Learning Rate: 0.001 | Batch Size: 32 | Epochs: 1000
Architecture:
Dropout(0.5)
Dense Layer: 1024 - 512 (sigmoid)
Dropout(0.5)
Dense Layer: 512 - 256 (sigmoid)
Dropout(0.5)
Dense Layer: 256 - 32 (sigmoid)
Dropout(0.5)
Dense Layer: 32 - 1 (no activation)



Not much difference was observed when compared to its shallower counterpart.

B. More Layers with ADAM

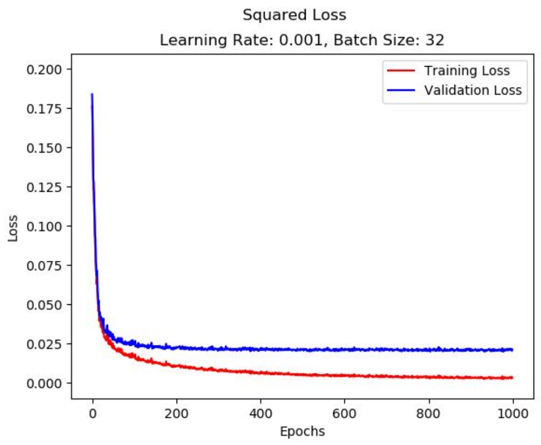
Hyperparameters:
Learning Rate: 0.001 | Batch Size: 32 | Epochs: 200
Optimizer: ADAM
Architecture:
Dropout(0.5)
Dense Layer: 1024 - 512 (sigmoid)
Dropout(0.5)
Dense Layer: 512 - 256 (sigmoid)
Dropout(0.5)
Dense Layer: 256 - 32 (sigmoid)
Dropout(0.5)
Dense Layer: 32 - 1 (no activation)



Using ADAM optimizer, the convergence rate increased drastically when compared to its non-optimized counterpart.

C. RELU Activation

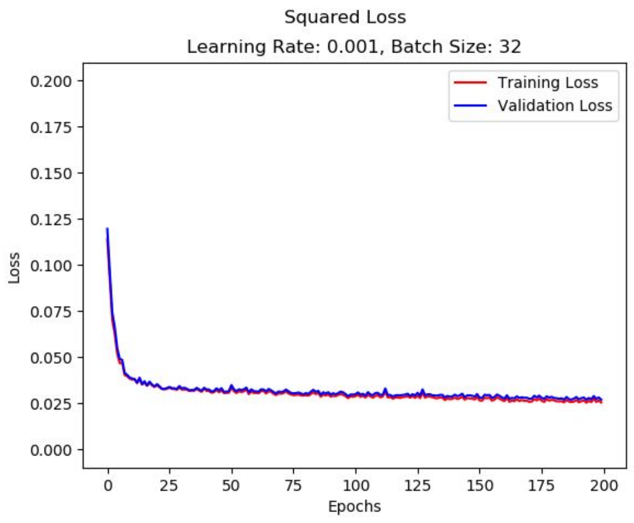
Hyperparameters:
Learning Rate: 0.001 | Batch Size: 32 | Epochs: 1000
Architecture:
Dropout(0.5)
Dense Layer: 1024 - 512 (relu)
Dropout(0.5)
Dense Layer: 512 - 64 (relu)
Dropout(0.5)
Dense Layer: 64 - 1 (no activation)



As compared to its sigmoid counterpart, using RELU activation helped the network to converge faster, however it also was more prone to overfitting.

D. Convolutional Neural Network

Hyperparameters:
Learning Rate: 0.001 | Batch Size: 32 | Epochs: 200
Architecture:
Conv2D(n2, k3x3, s1, padding:same) (tanh)
Maxpool(k2x2, s2)
Dropout(0.5)
Conv2D(n4, k3x3, s1, padding:same) (tanh)
Maxpool(k2x2, s2)
Dropout(0.5)
Flatten()
Dense Layer: 256 - 32 (tanh)
Dense Layer: 32 - 1 (no activation)

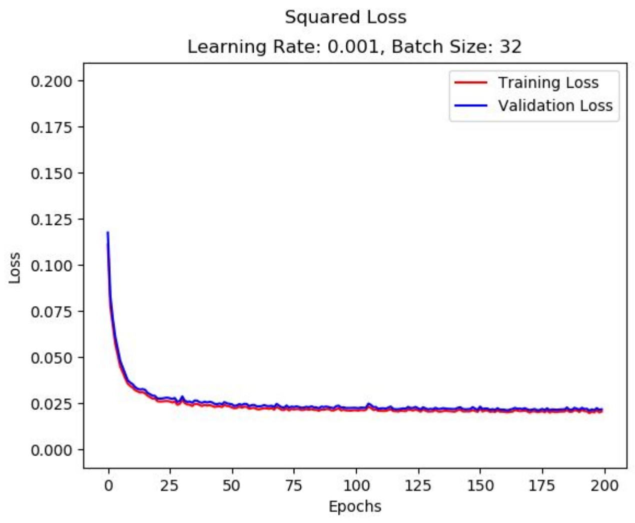


To implement CNN, I had taken help from Stanford's CS231n course and used their im2col function for implementing a faster CNN.

As compared to a basic neural network, CNN converged faster for this image dataset. The input were batch_size x 32 x 32 tensor.

E. CNN with ADAM

Hyperparameters:
Learning Rate: 0.001 | Batch Size: 32 | Epochs: 200
Optimizer: ADAM
Architecture:
Conv2D(n2, k3x3, s1, padding:same) (tanh)
Maxpool(k2x2, s2)
Dropout(0.5)
Conv2D(n4, k3x3, s1, padding:same) (tanh)
Maxpool(k2x2, s2)
Dropout(0.5)
Flatten()
Dense Layer: 256 - 32 (tanh)
Dense Layer: 32 - 1 (no activation)



The difference between the losses by CNN and CNN with ADAM were of the order of 10^{-3}.