ASSIGNMENT 3

SYSTEMS ENGINEERING FOR DEEP LEARNING

Submitted By: Sooryakiran (ME17B174)

STEP 1

A neural network with the given architecture was trained on the FMNIST dataset. The network weights were initialized with Xavier uniform initialisation. Stochastic Gradient Descend optimizer with momentum was used with a learning rate of 0.001 and with a momentum weightage of 0.9.

Dataset	FMNIST 60000 Training samples 10000 Testing samples
Preprocessing	Normalization
Optimizer	SGD with momentum
Learning Rate	1e-3
Momentum (β)	0.9
Batch Size	32
Network initialization	Weights : Xavier uniform Biases : Zero
Network Architecture	Layer 1 : Conv 3x3, 16 Channels, ReLU Layer 2: Conv 3x3, 16 Channels, ReLU Layer 3: Dense 100 neurons, ReLU Layer 4: Dense 10 neurons, Softmax

Table 1.1: Model and training hyperparameters.

The network was trained in a CPU only instance on Google Colaboratory. With a batch size of 32, I got 90.18 % training accuracy and a test accuracy of 88.33%. The loss vs batches plots and accuracy vs batches plots and the final results are given in the following page.

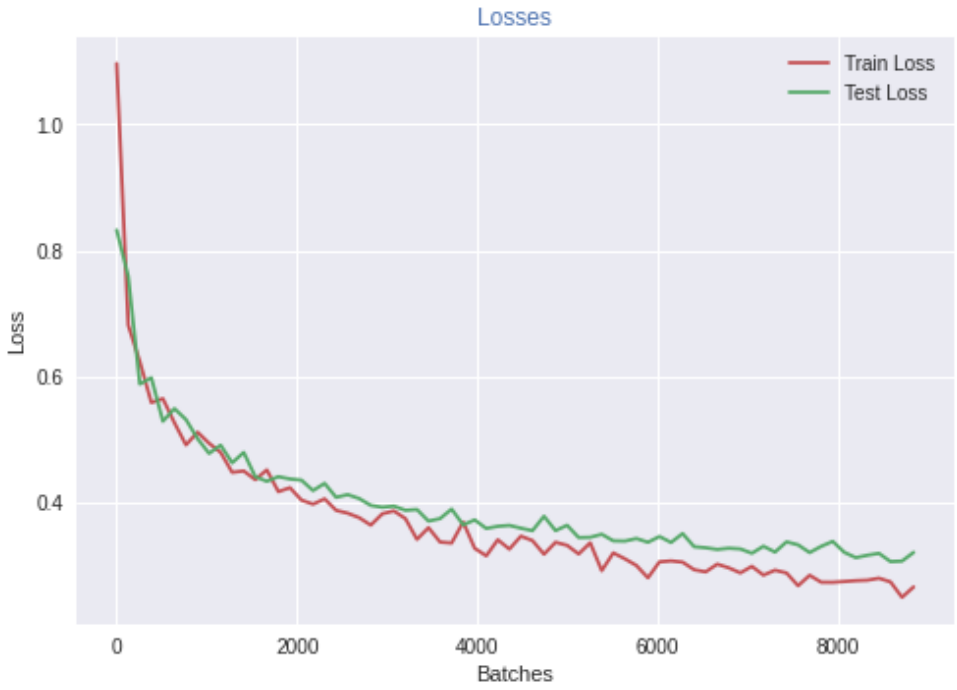


Figure 1.1.a:
Loss vs Batches plot

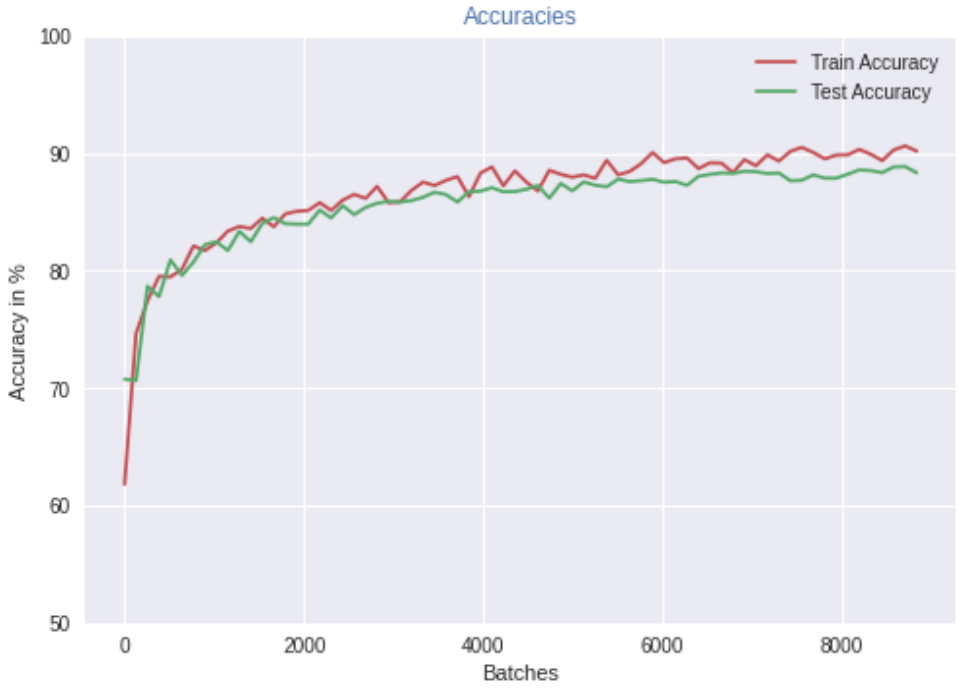


Figure 1.1.b:
Accuracy vs Batches plot

Google Facets was used to analyse the prediction performance of the trained model. Google Facets allow inputs only in CSV format. So the first the model was used to generate predictions for the entire test dataset. Then a CSV file was generated with columns being the True and Predicted labels. This CSV file was uploaded to Google Facets website. Many visualizations like the confusion matrix, distribution of predictions were obtained.

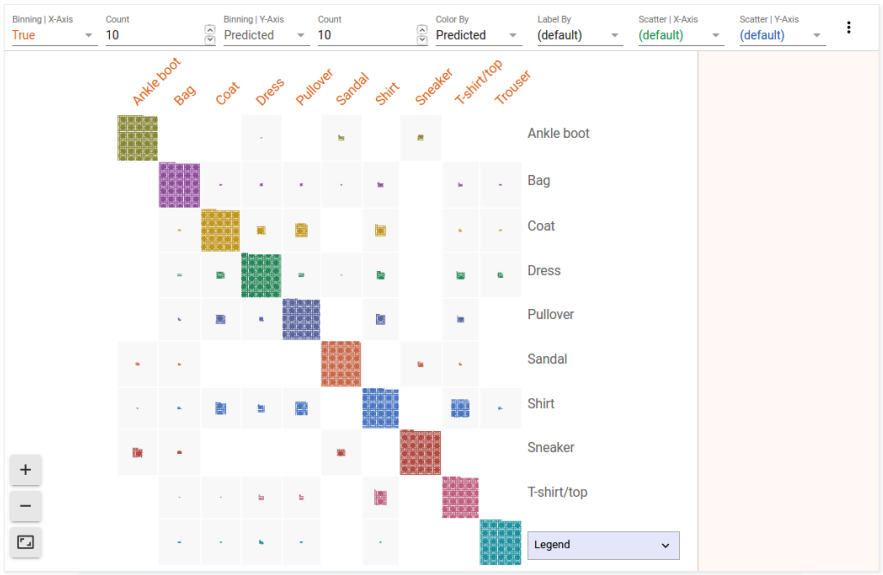


Figure 1.2.a: Confusion Matrix

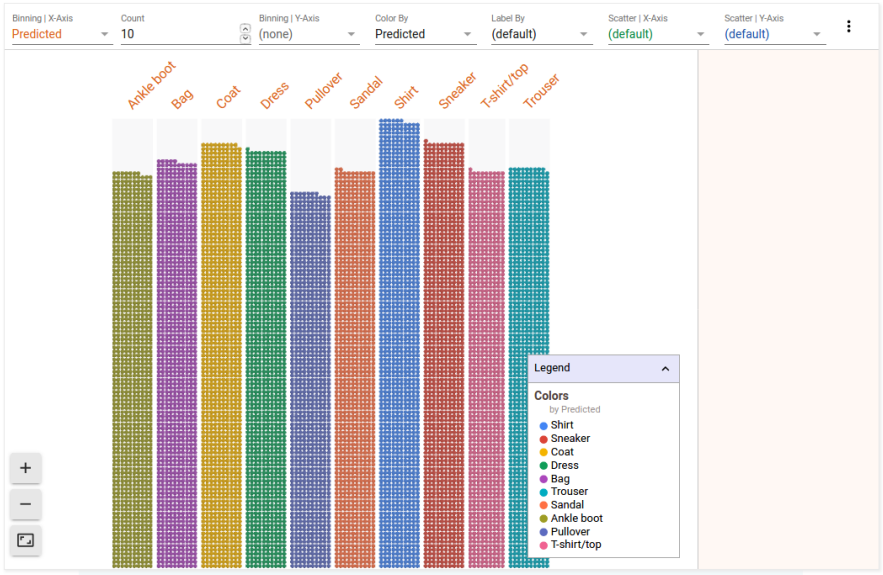


Figure 1.2.b: Distribution of predictions.

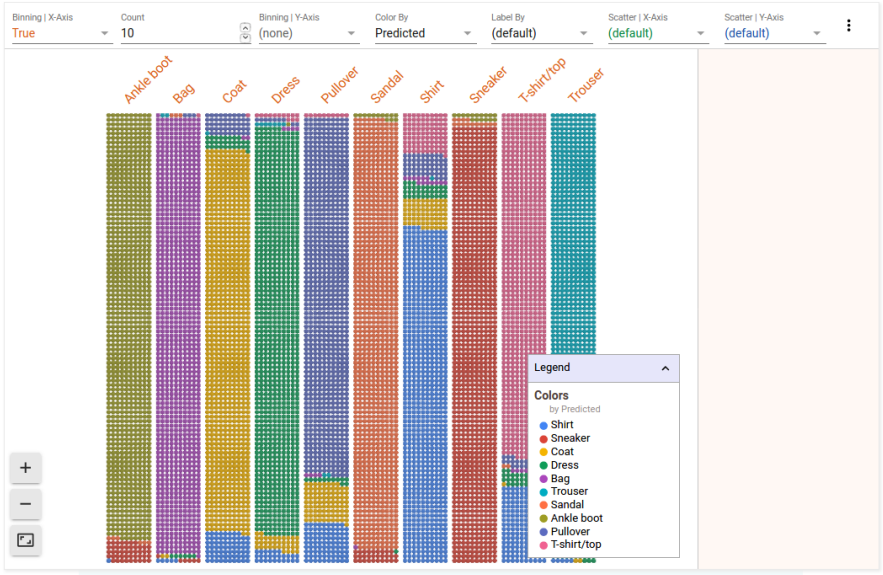


Figure 1.2.c: Predictions for each true label.

These visualizations helps us to understand where out model is failing to perform. For example, in Fig. 1.2.c, we can see that nearly 30% of T-shirts/tops are misclassified as shirts.

¯_(ツ)_/¯	*Loss*	*Accuracy*
Training	0.267	90.186 %
Testing	0.321	88.338 %

Table 1.2: Train and test performances

STEP 2

MLflow was used to conduct grid search to find the optimial hyperparameters for the model. Two experiments were conducted, one to find the right Model Hyperparameters and another one to find the right Training Hyperparameters. These experiments were conducted one by one. The optimal model architecture obtained from the first experiment was used for the second experiment.

The first experiment search space is as follows:

Hyperparameter	*Values*
Conv 1 kernel sizes	3x3, 5x5
Conv 2 kernel sizes	3x3, 5x5
Conv 1 no of filters	8, 16, 32
Conv 2 no of filters	8, 16, 32
Dense 1 no of neurons	64, 100

Table 2.1.a: The search space for experiment 1

Results are as follows.

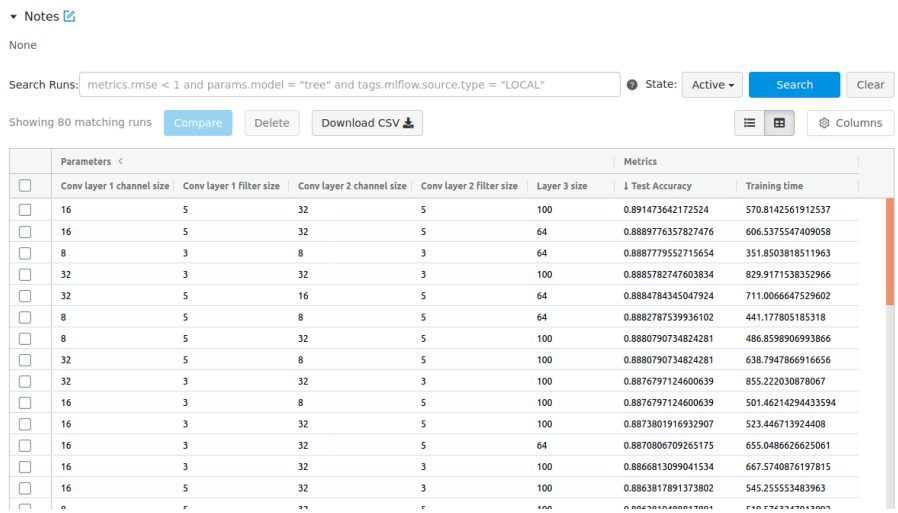


Fig 2.1.a: Experiment results sorted by decreasing accuracy on the test dataset.

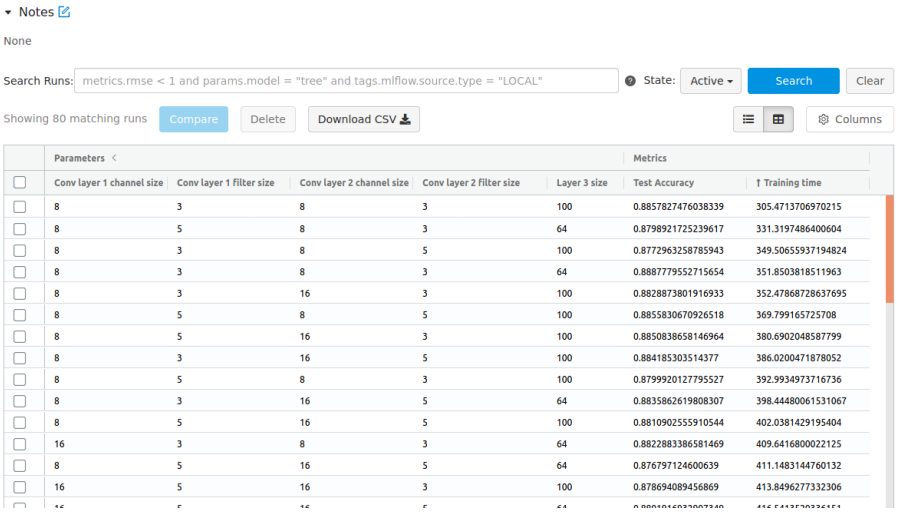


Fig 2.1.b: Experiment results sorted by increasing training time required to reach 90% train accuracy.

Top 3 results are as shown below:

	#1	*#2*	*#3*
Conv 1	5x5, 16 filters	5x5, 16 filters	3x3, 8 filters
Conv 2	5x5, 32 filters	5x5, 32 filters	3x3, 8 filters
Dense 3	100 neurons	64 neurons	64 neurons
Test Accuracy	89.14 %	88.89 %	88.87 %
Training Time	570.81s	606.53s	351.85s

Table 2.1.a: Top 3 results based on test accuracy.

	#1	*#2*	*#3*
Conv 1	3x3, 8 filters	5x5, 8 filters	3x3, 8 filters
Conv 2	3x3, 8 filters	3x3, 8 filters	5x5, 8 filters
Dense 3	100 neurons	64 neurons	100 neurons
Test Accuracy	88.57 %	87.98 %	87.72 %
Training Time	305s	331s	349s

Table 2.1.b: Top 3 results based on training time.

Smaller filter sizes have smaller training time because there are less parameters and less computation involved. But filters with larger receptive field performs better as they have higher representative power. This effect is more profound for shallow networks like the one we experimented with. Also, netoworks with higher number of filters can represent more features and are likely to have higher performance but compromises on training time. When the number of neurons on the dense layer is incresed, it improves the performance but increases the training time.

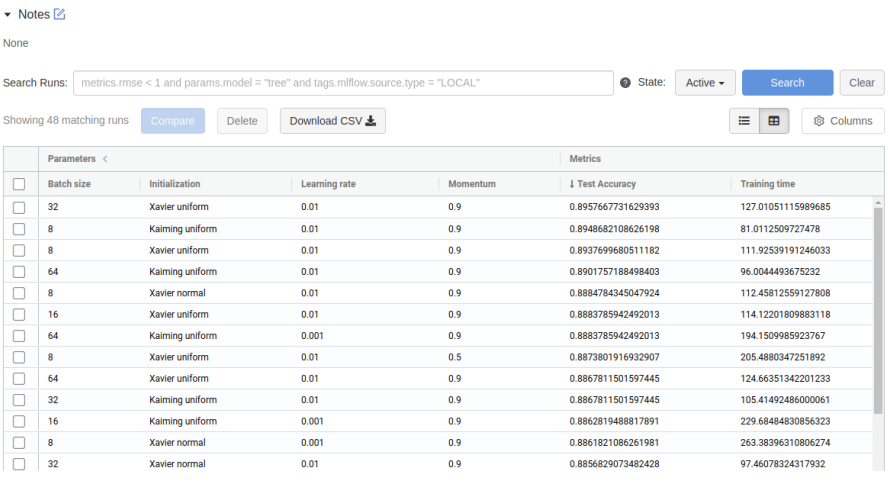
I decided to choose the model with lowest training time for the next experiment because it is acceptable to sacrifice 0.64% of accuracy for 46% reduction in training time.

Experiment 2 was conducted on the model that had the smallest training time.

Hyperparameter	*Values*
Initialisation	Xavier Uniform, Xavier Normal, Kaiming Uniform
Batch sizes	8, 16, 32, 64
Optimizer params	1. Learning rate : 0.001 Momentum : 0.9 2. Learning rate : 0.01 Momentum : 0.9 3. Learning rate : 0.001 Momentum : 0.5 4. Learning rate : 0.01 Momentum : 0.5

Table 2.2:
The search space for experiment 2

Results are as follows.


Notes

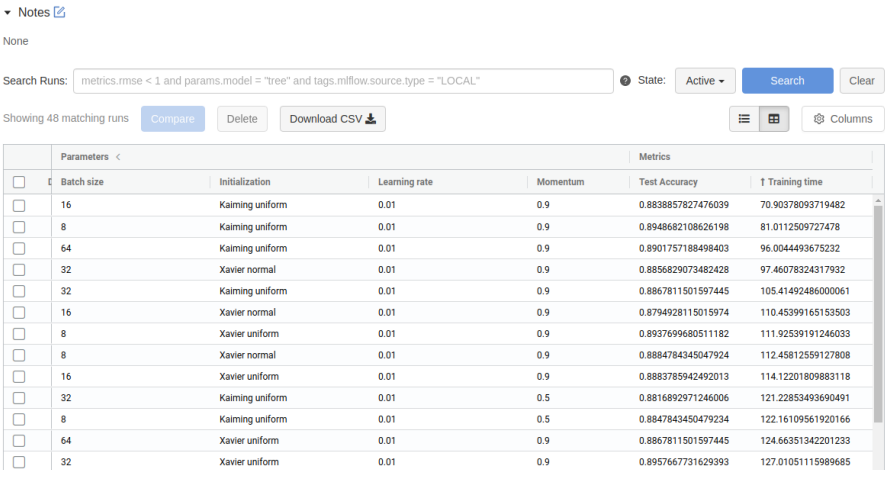
None

Search Runs: metrics.rmse < 1 and params.model = "tree" and tags.mlflow.source.type = "LOCAL" State: Active Search Clear

Showing 48 matching runs Compare Delete Download CSV Columns

	Parameters			Metrics	
Batch size	Initialization	Learning rate	Momentum	↓ Test Accuracy	Training time
32	Xavier uniform	0.01	0.9	0.8957667731629393	127.01051115989685
8	Kaiming uniform	0.01	0.9	0.8948682108626198	81.0112509727478
8	Xavier uniform	0.01	0.9	0.8937699680511182	111.92539191246033
64	Kaiming uniform	0.01	0.9	0.8901757188498403	96.0044493675232
8	Xavier normal	0.01	0.9	0.8884784345047924	112.45812559127808
16	Xavier uniform	0.01	0.9	0.8883785942492013	114.12201809883118
64	Kaiming uniform	0.001	0.9	0.8883785942492013	194.1509985923767
8	Xavier uniform	0.01	0.5	0.8873801916932907	205.4880347251892
64	Xavier uniform	0.01	0.9	0.8867811501597445	124.66351342201233
32	Kaiming uniform	0.01	0.9	0.8867811501597445	105.41492486000061
16	Kaiming uniform	0.001	0.9	0.8862819488817891	229.68484830856323
8	Xavier normal	0.001	0.9	0.8861821086261981	263.38396310806274
32	Xavier normal	0.01	0.9	0.8856829073482428	97.46078324317932

Fig 2.2.a:
Experiment 2 results sorted by decreasing accuracy on the test dataset.


Notes

None

Search Runs: metrics.rmse < 1 and params.model = "tree" and tags.mlflow.source.type = "LOCAL" State: Active Search Clear

Showing 48 matching runs Compare Delete Download CSV Columns

	Parameters			Metrics	
Batch size	Initialization	Learning rate	Momentum	Test Accuracy	↑ Training time
16	Kaiming uniform	0.01	0.9	0.8838857827476039	70.90378093719482
8	Kaiming uniform	0.01	0.9	0.8948682108626198	81.0112509727478
64	Kaiming uniform	0.01	0.9	0.8901757188498403	96.0044493675232
32	Xavier normal	0.01	0.9	0.8856829073482428	97.46078324317932
32	Kaiming uniform	0.01	0.9	0.8867811501597445	105.41492486000061
16	Xavier normal	0.01	0.9	0.8794928115015974	110.45399165153503
8	Xavier uniform	0.01	0.9	0.8937699680511182	111.92539191246033
8	Xavier normal	0.01	0.9	0.8884784345047924	112.45812559127808
16	Xavier uniform	0.01	0.9	0.8883785942492013	114.12201809883118
32	Kaiming uniform	0.01	0.5	0.8816892971246006	121.22853493690491
8	Kaiming uniform	0.01	0.5	0.8847843450479234	122.16109561920166
64	Xavier uniform	0.01	0.9	0.8867811501597445	124.66351342201233
32	Xavier uniform	0.01	0.9	0.8957667731629393	127.01051115989685

Fig 2.2.b:
Experiment 2 results sorted by increasing training time required to reach 90% train accuracy.

Top 3 results are as shown below:

	#1	*#2*	*#3*
Batch size	32	8	8
Initialization	Xavier Uni.	Kaiming Uni.	Xavier Uni.
Learning rate	0.01	0.01	0.01
Momentum	0.9	0.9	0.9
Test Accuracy	89.57 %	89.48 %	89.37 %
Training Time	127s	81s	111s

Table 2.2.a:
Top 3 results based on test accuracy.

	#1	*#2*	*#3*
Batch size	16	8	64
Initialization	Kaiming Uni.	Kaiming Uni.	Kaiming Uni.
Learning rate	0.01	0.01	0.01
Momentum	0.9	0.9	0.9
Test Accuracy	88.38 %	89.48 %	89.01 %
Training Time	70s	81s	96s

Table 2.2.b:
Top 3 results based on training time.

Learning rate of 0.01 performed well in most cases. It converges faster and results in better performance on the test dataset. Also models with momentum of 0.9 outperformed models with momentum of 0.5 because it is better in escaping local minima traps. Kaiming initialization is found to converge faster. A smaller batch size give rise to gradients that are more randomized comparitively. This might help in escaping the local minima traps.

Here are some scatter plots that can help us choose even better models. However, not every one of these plots are conclusive. Plots like the one for learning rate, momentum, and mumber of filters can help us generalise how different models behave according to different hyperparameters. The MLflow logs are converted to CSV files and uploaded to Google Facets.

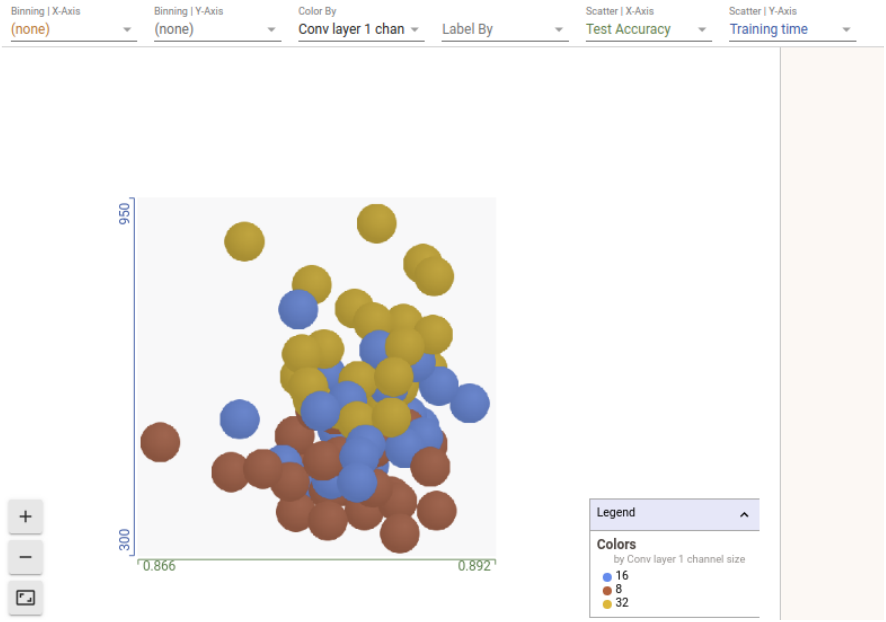


Fig 2.3.a:
Conv 1 no of filters
x-axis : Test accuracy
y-axis: Training time

Fig 2.3.b:
Conv 1 kernel size
x-axis : Test accuracy
y-axis: Training time

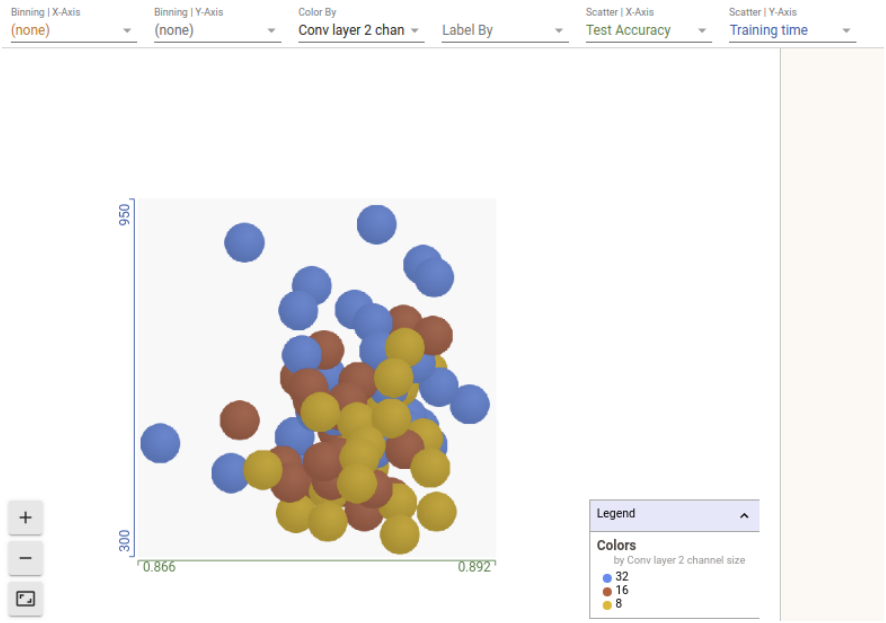


Fig 2.3.c:
Conv 2 no of filters
x-axis : Test accuracy
y-axis: Training time

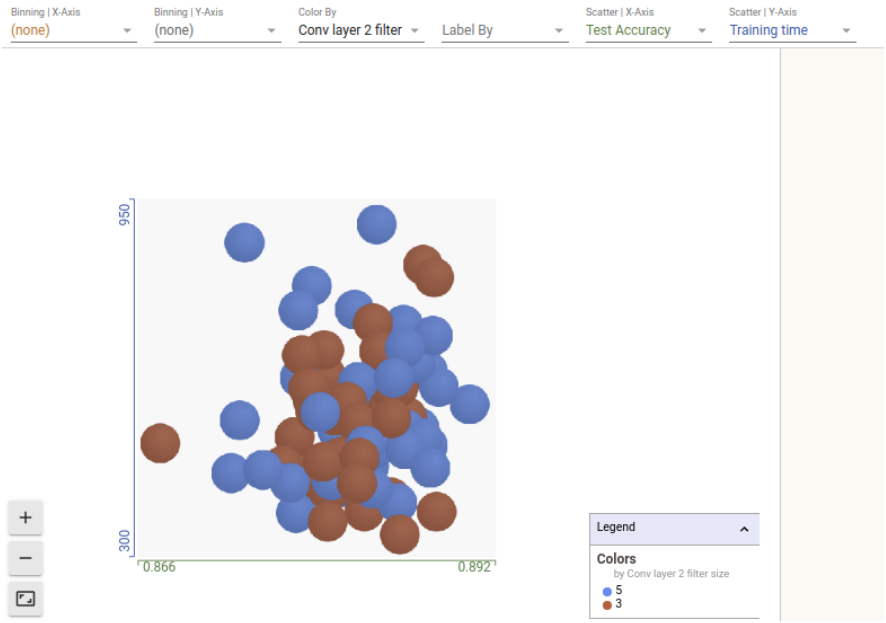


Fig 2.3.d:
Conv 2 kernel size
x-axis : Test accuracy
y-axis: Training time

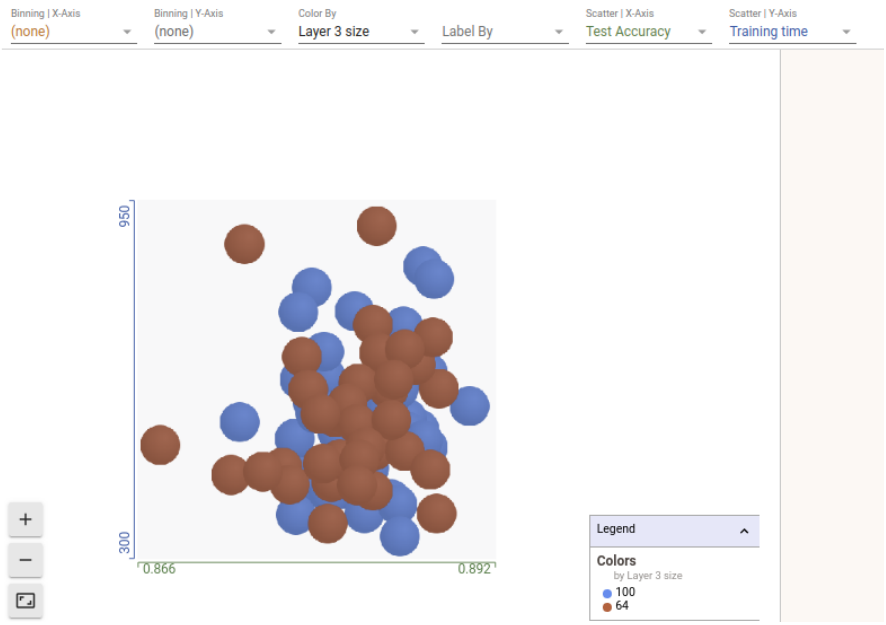


Fig 2.3.e:
Number of neurons in layer 3
x-axis : Test accuracy
y-axis: Training time

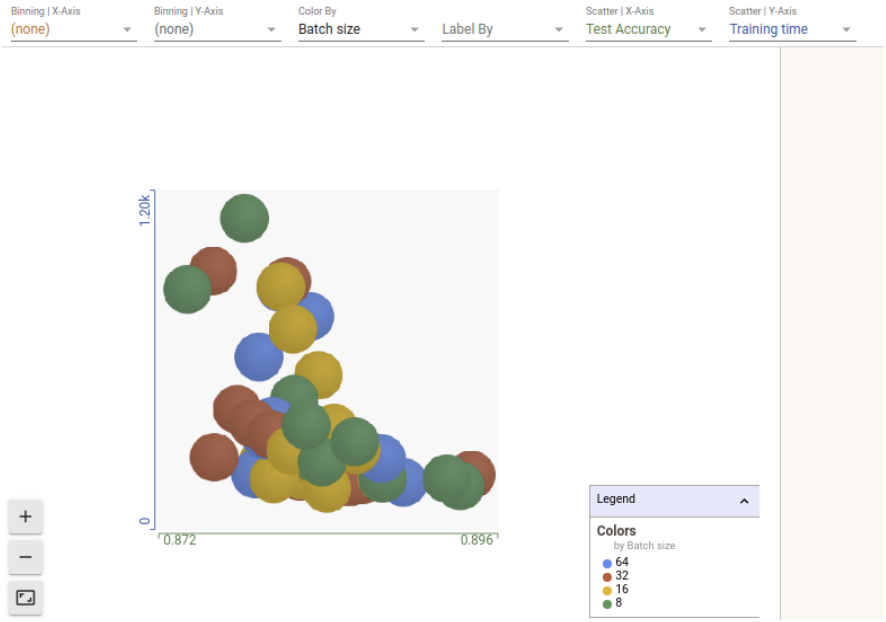


Fig 2.3.f:
Batch size
x-axis : Test accuracy
y-axis: Training time

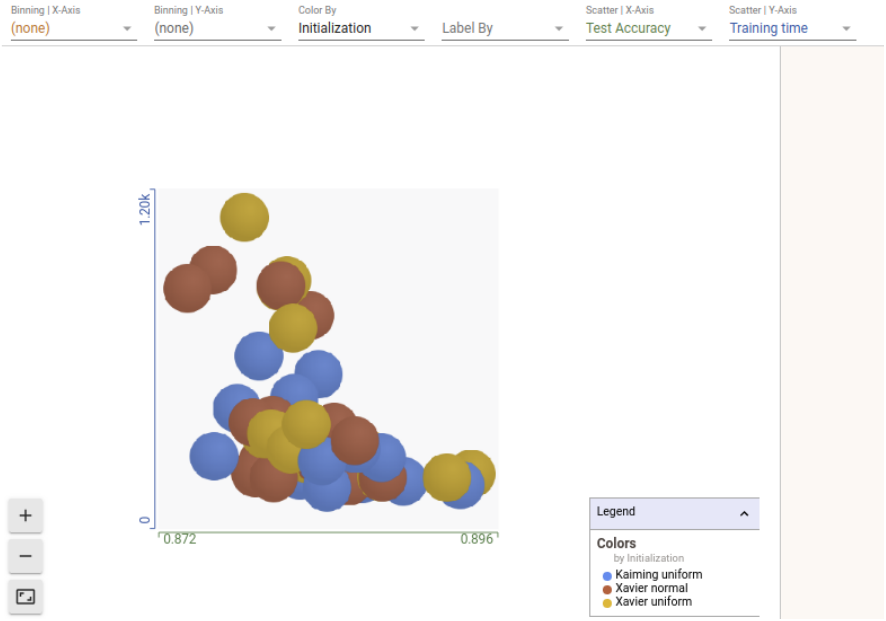


Fig 2.3.g:
Initialization
x-axis : Test accuracy
y-axis: Training time

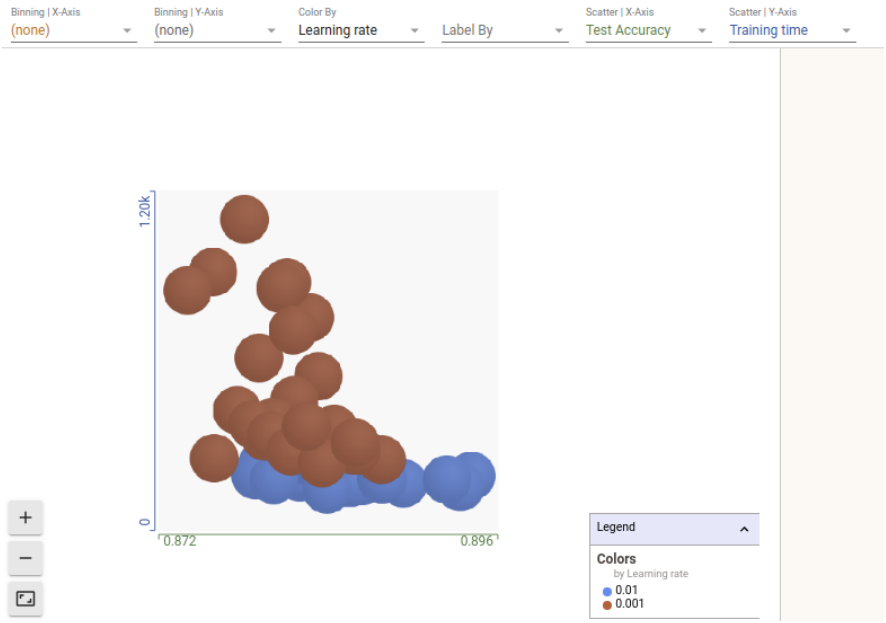


Fig 2.3.h:
Learning rate
x-axis : Test accuracy
y-axis: Training time

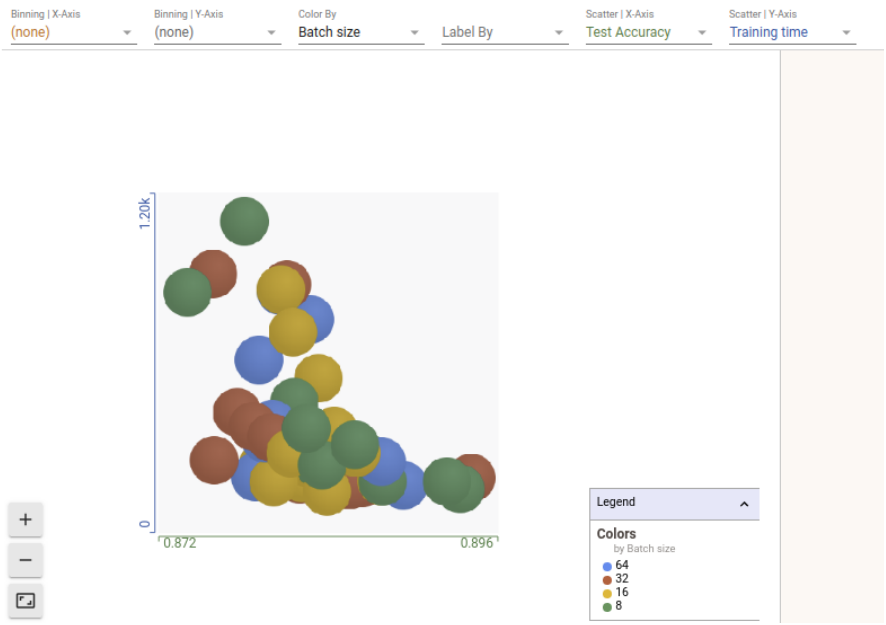


Fig 2.3.i:
Momentum
x-axis : Test accuracy
y-axis: Training time

The pareto plots of all models with respect to testing accuracy and training time are as given below.

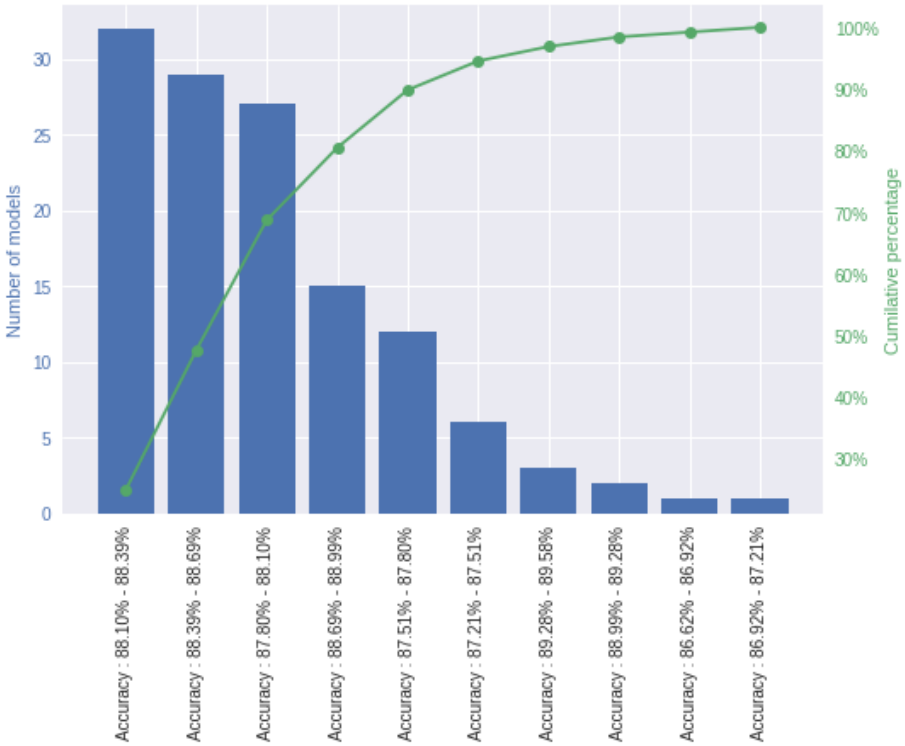


Fig 2.4.a:
Pareto plot of all models with respect to testing accuracy.

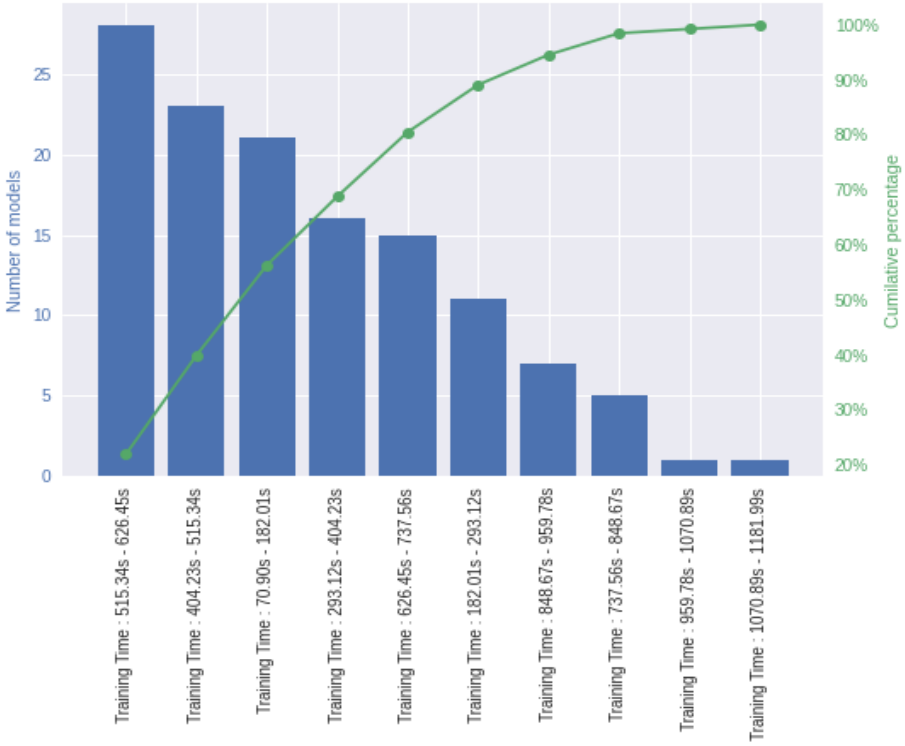


Fig 2.4.b:
Pareto plot of all models with respect to training time.

Loss vs epoch/batches plots for some special cases asked in the question are given below.

Learning rate too high: The largest learning rate used for the experiment was 0.01. This is not a high learning rate. Therfore to produce the plot, I trained a model separately with learning rate. We can see that the loss plot has got many irregular jumps.

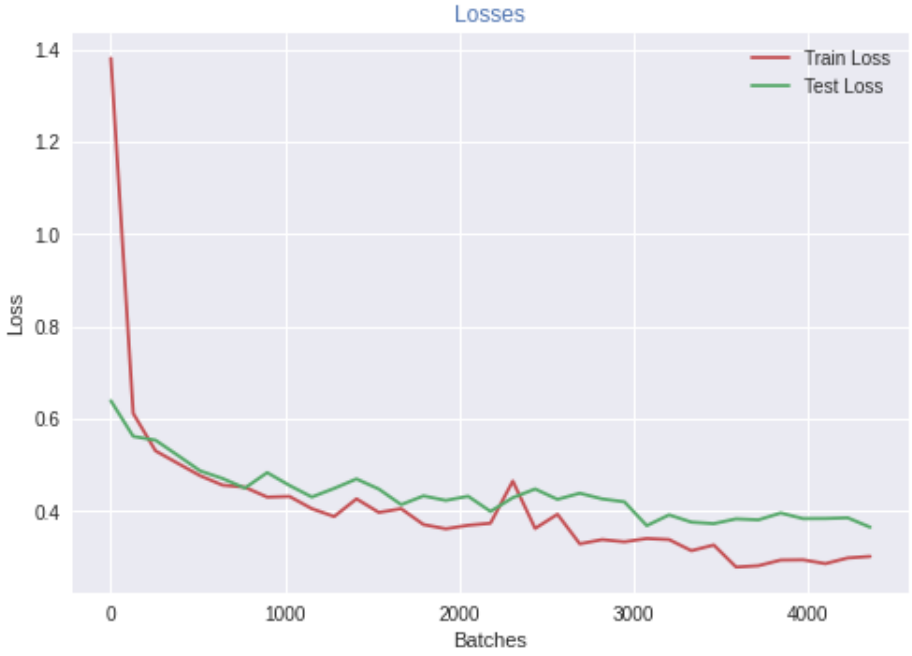


Fig 2.5.a: High learning rate (0.05). Loss plot has irregular jumps.

Learning rate too low: The MLflow experiment did not contain any examples for low learning rates. So I trained a separate model to demonstrate. The network is taking longer to converge.

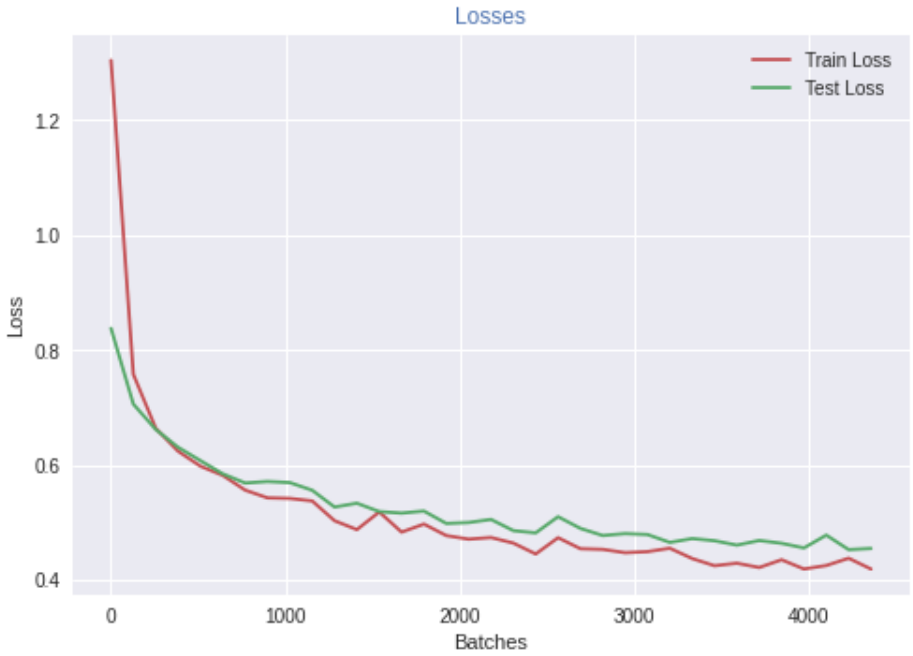


Fig 2.5.b: Low learning rate (0.0001). Loss plot takes very longer to converge.

Momentum too high: This plot is also generated by running an experement separately. We can see that the plot has got an oscillatory nature. The model oscillates around local minimas. Sometimes the network is able to escape local minimas, but sometimes it keeps on oscillating.

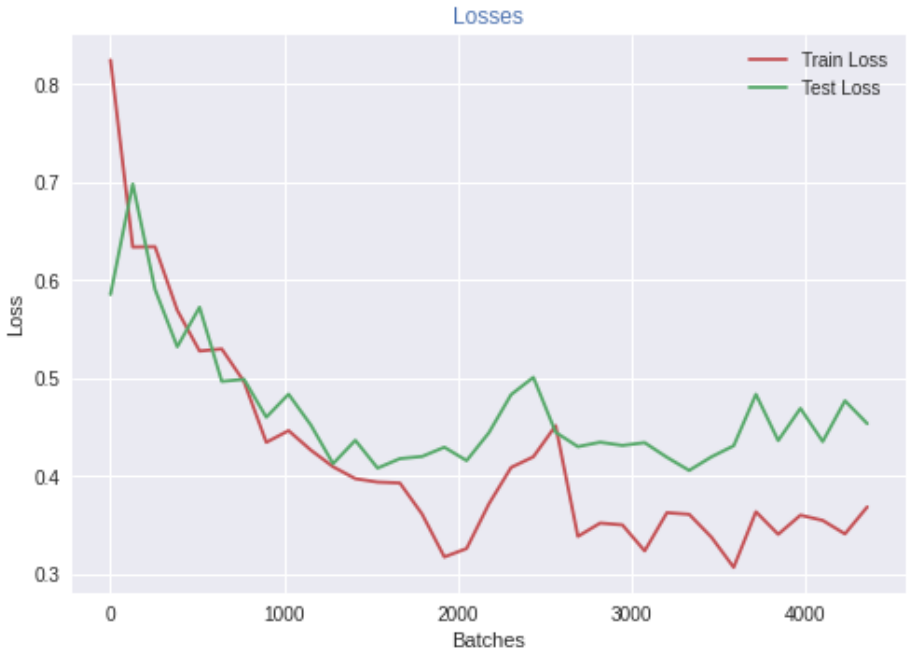


Fig 2.5.c:
High momentum (0.99, lr: 0.001). The loss oscillates too much.

Batch size too low: When a network is trained using stochastic gradient descend on a very small batch size. The updates become more randomized. The plot below shows a network trained with batch size of 4. All other settings were similar to that of the above given networks.

Fig 2.5.d:
Low batch size. All the updates become more approximate and randomized, thus leading to a rugged loss plot.

Overfitting due to large model: The experiments conducted did not have any good exaples of overfitting. So a different larger model was trained to demonstrate the effect. This example was trained longer than others to make the effect more profound. We can see the test loss actually diverging.

Fig 2.5.e: Overfitting on a large model.(Conv 5x5, 64; Conv 5x5, 64; 100 neurons)

STEP 3

The best model from the previous experiment was choosen. Training time was used as the criterion for seelecting the model. It is because there was huge decrease of 43% in training time with just 0.6% drop in test accuracy. The architecture of the selected model is shown below.

Layers	*Description*
1	Conv, 3x3, 8 filters, ReLU
2	Conv, 3x3, 8 filters, ReLU
3	Dense, 100 neurons, ReLU
4	Dense, 10 neurons, Softmax

Table 3.1: Selected model.

Channel pruning for convolutional layers and neuron pruning for dense fully connected layers are implimented in pytorch. In channel pruning we turn of an entrire filter from the convolutional layers and in neuron pruning, one entire neuron is removed. This can be achieved by turning off the corresponding weights to zero and its gradients to zero.

But pruning by turning the weights to zero does not speed up inference as the number of computations remain the same. So I implimented pruning that modifies the entire model. The pseudo code for pruning a conv layer is given in *Algorithm 3.1.a* and the pseudo code for pruning a dense layer is given in *Algorithm 3.1.b.* with suitable abstractions.

```
Algorithm 1
===========

Pruning convolutional channels
------------------------------

new_model = ConvNet() // A new model of the computed architecture
                      // created

for all layers in the base_base model:
    if layer != layer_to_be_pruned && layer != next_layer:
        // Copy parameters for all other layers
        new_model[layer].params = base_model[layer].params

    else if layer == layer_to_be_pruned:
        new_param = base_model[layer].param
        if new_param.type = "weight":
            // Modify the weight param
            new_param = remove_channel_weights(new_param)
        else:
            // Modify the bias param
            new_param = remove_channel_bias(new_param)
        new_model[layer].params = new_param

    else:
        // Next layer
        new_param = base_model[layer].param
        if layer.type = "conv" && new_param.type = "weight":
            // Next layer is conv
            new_param = remove_conv_input_channel(new_param)
        else if layer.type = "fc" && new_param.type = "weight":
            // Next layer is fc
            // We have to remove selected rows from the weight
            // matrix
            new_param = remove_fc_input_channel(new_param)
        new_model[layer].params = new_param
```

Algorithm 3.1.a: Pruning a convolutional layer.

Unlike the other method which constraints the parameters of the pruned channel/neuron to be zero, this model creates a new model with the required architeture and copies the required parameters to the new model. The weights and biases of the current model will have to be modified. Furthermore, the weights of the next layer will also has to modified.

The full implimentation can be found in the iPython Colaboratory notebook.

```
Algorithm 2
===========

Pruning dense neurons
-----------------------------

new_model = ConvNet() // A new model of the computed architecture
                      // created

for all layers in the base_base model:
    if layer != layer_to_be_pruned && layer != next_layer:
        // Copy parameters for all other layers
        new_model[layer].params = base_model[layer].params

    else if layer == layer_to_be_pruned:
        new_param = base_model[layer].param
        if new_param.type = "weight":
            // Modify the weight param
            new_param = remove_neuron_weights(new_param)
        else:
            // Modify the bias param
            new_param = remove_neuron_bias(new_param)
        new_model[layer].params = new_param

    else:
        // Next layer
        // For next layer, we care only about weights, not bias
        new_param = base_model[layer].param
        if new_param.type = "weight":
            new_param = remove_fc_input_channel(new_param)
        new_model[layer].params = new_param
```

Algorithm 3.1.b: Pruning a fully dense connected layer.

A small heuristic is used to predict the layer to be pruned. The influence of weights to the final prediction is approximated by the squared sum of all parameters, weights and biases in a channel/neuron. The plot below shows the effectiveness of the heuristic.

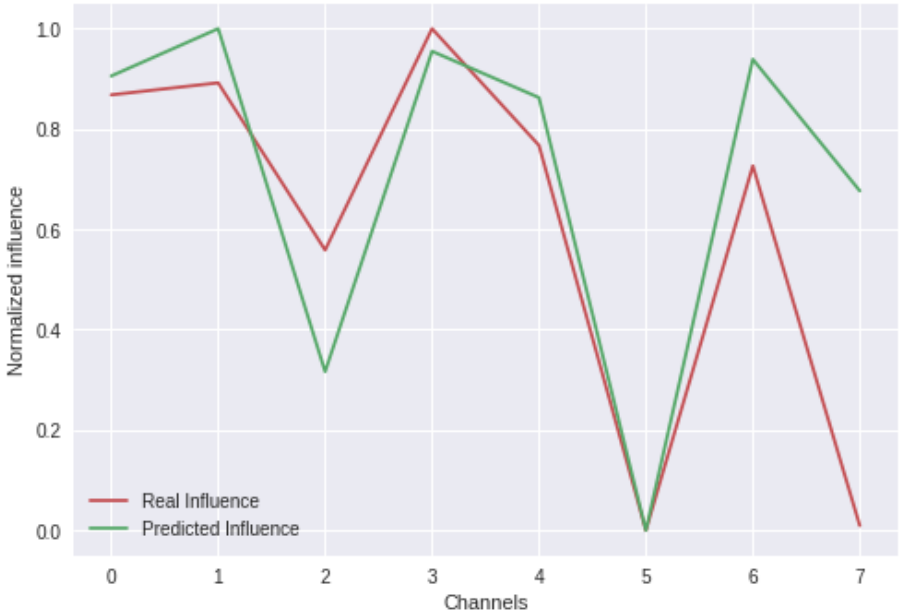


Fig 3.1: Predicted influence of each channel from the first layer compare to the real influence

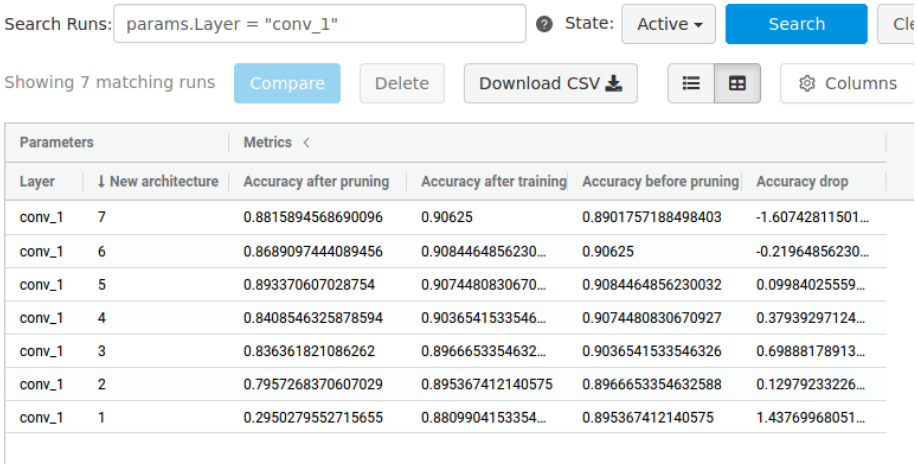
The predicted influenze is the squared sum of parameters normalized to lie beteen zero and one. The real influence is the normalized drop in test accuracy when the channel is removed.

The predited channel from the first convolutional layer was removed. This new network was tested to find the drop in accuracy. Then the new network was trained for five epochs and was tested again. The results are summarised as below.

	Before pruning	*After pruning*	*Fine tuned*
Accuracy	89.01 %	88.15 %	90.6 %

Table 3.2: Accuracy triplet after removing a channel from conv 1 layer.

This process was repeated multiple times for the first convolutional layer untill the drop in accuracy becomes greater than 1% or there are no layers left. All the experiments were logged using Mlflow, the results are given below.

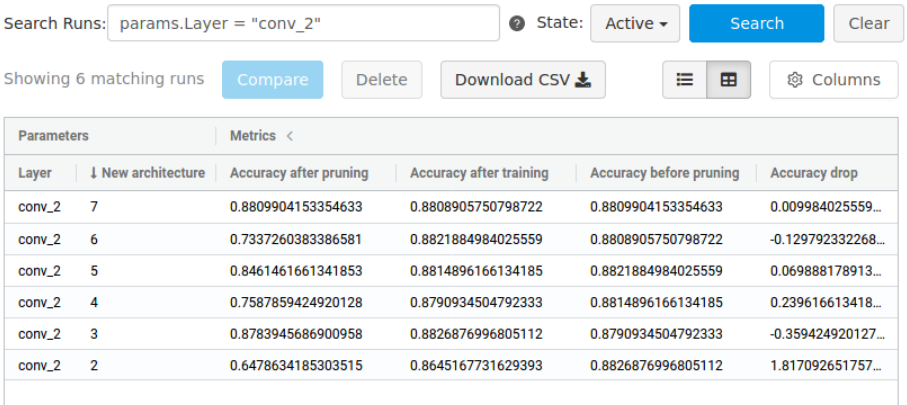


Parameters		Metrics			
Layer	New architecture	Accuracy after pruning	Accuracy after training	Accuracy before pruning	Accuracy drop
conv_1	7	0.8815894568690096	0.90625	0.8901757188498403	-1.60742811501...
conv_1	6	0.8689097444089456	0.9084464856230...	0.90625	-0.21964856230...
conv_1	5	0.893370607028754	0.9074480830670...	0.9084464856230032	0.09984025559...
conv_1	4	0.8408546325878594	0.9036541533546...	0.9074480830670927	0.37939297124...
conv_1	3	0.836361821086262	0.8966653354632...	0.9036541533546326	0.69888178913...
conv_1	2	0.7957268370607029	0.895367412140575	0.8966653354632588	0.12979233226...
conv_1	1	0.2950279552715655	0.8809904153354...	0.895367412140575	1.43769968051...

Table 3.3.a: Accuracy triplets after removing channels one by one following the heuristic from conv 1 layer

It is interesting to see that even 1 channel was sufficient for FMNIST dataset.

The same experiment was done for conv 2 and the dense layers.



Parameters		Metrics			
Layer	New architecture	Accuracy after pruning	Accuracy after training	Accuracy before pruning	Accuracy drop
conv_2	7	0.8809904153354633	0.8808905750798722	0.8809904153354633	0.009984025559...
conv_2	6	0.7337260383386581	0.8821884984025559	0.8808905750798722	-0.129792332268...
conv_2	5	0.8461461661341853	0.8814896166134185	0.8821884984025559	0.069888178913...
conv_2	4	0.7587859424920128	0.8790934504792333	0.8814896166134185	0.239616613418...
conv_2	3	0.8783945686900958	0.8826876996805112	0.8790934504792333	-0.359424920127...
conv_2	2	0.6478634185303515	0.8645167731629393	0.8826876996805112	1.817092651757...

Table 3.3.b: Accuracy triplets after removing channels one by one following the heuristic from conv 2 layer

Search Runs: params.Layer = "fc_1" State: Active Search Clear

Showing 3 matching runs Compare Delete Download CSV Columns

Parameters		Metrics			
Layer	↓ New architecture	Accuracy after pruning	Accuracy after training	Accuracy before pruning	Accuracy drop
fc_1	99	0.10063897763578275	0.8562300319488818	0.8645167731629393	0.828674121405...
fc_1	98	0.0878594249201278	0.8750998402555911	0.8562300319488818	-1.886980830670...
fc_1	97	0.12889376996805113	0.8610223642172524	0.8750998402555911	1.407747603833...

Table 3.3.c:
Accuracy triplets after removing channels one by one following the heuristic from fc 1 layer.

STEP 4

The analytical formula that computes the FLOPs of a layer is as given below:

$$FLOPS : Convolution$$
$$input = [n, H, W]$$
$$kernel = f \times k \times k$$
$$Padding = P_x, P_y$$
$$Strides = S_x, S_y$$
$$FLOPS = outputSize * elementWiseFlops$$
$$OutputSize = f.(\frac{W-K+2P_x}{S_x} + 1).(\frac{H-K+2P_y}{S_y} + 1)$$
$$elementWiseFlops = flopMultiply + flopsAdd + activation$$
$$elementWiseFlops = nk^2 + (nk^2 - 1) + a = 2nk^2 - 1 + a$$
$$FLOPS = (2nk^2 - 1 + a).f.(\frac{W-K+2P_x}{S_x} + 1).(\frac{H-K+2P_y}{S_y} + 1)$$

Fig. 4.1.a:
FLOPs of a conv layer.

$$FLOPS : Dense$$
$$input = [n]$$
$$weight = m \times n$$
$$output = [m]$$
$$FLOPS = outputSize * elementWiseFlops$$
$$OutputSize = m$$
$$elementWiseFlops = flopMultiply + flopsAdd + activation$$
$$elementWiseFlops = n + (n - 1) + a = 2n - 1 + a$$
$$FLOPS = m(2n - 1 + a)$$

Fig. 4.1.b:
FLOPs of a dense layer.

A pareto chart was plotted with the testing accuracy against the number of FLOPs for all the models from the pruning experiment.

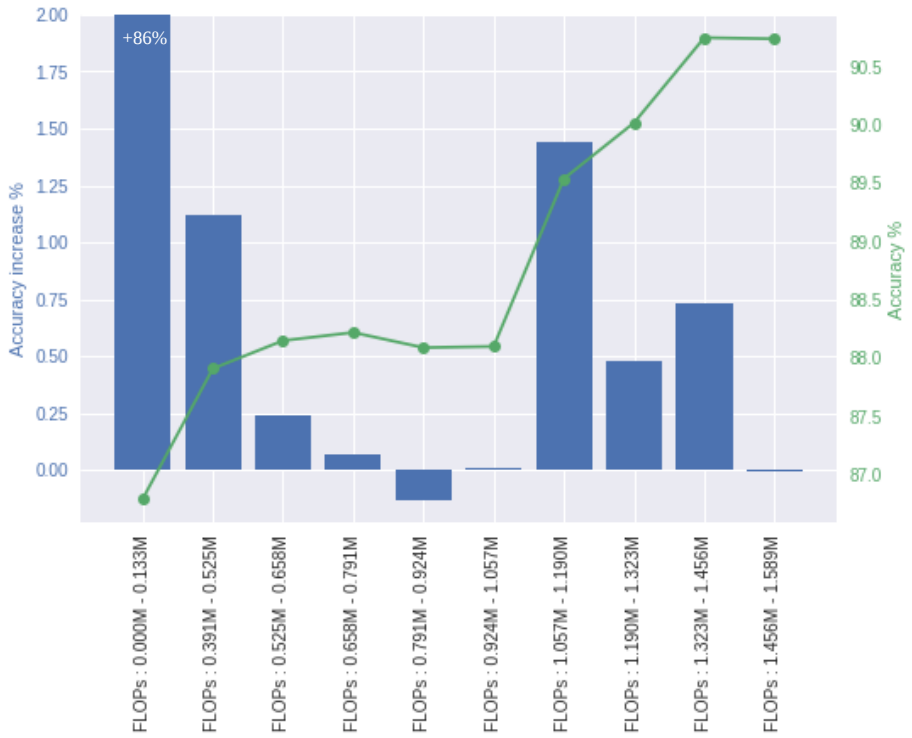


Fig. 4.2:
Pareto plot of average test accuracy vs model FLOPs

Within the limits of experimental error on errors due to less data points, we can see that the testing accuracy increases with increase in the number of flops. However, this increase begins to saturate after a limit.

The model with lowest FLOP with test accuracy higher than 87% is taken. Its architecture is as given below.

Layers	*Description*
1	Conv, 3x3, 1 filter, ReLU
2	Conv, 3x3, 2 filters, ReLU
3	Dense, 98 neurons, ReLU
4	Dense, 10 neurons, Softmax

Table 4.1:
Selected model with lowest FLOP and testing accuracy greater than 87%.

The model was written using matrix multiplication in Pytorch, i.e without using the inbuilt *nn.Conv2d* and *nn.Linear* module. Convolution was implimented by converting it to a GEMM. This was done using *torch.unfold(. . .)* function. However, this is not implimented as a diffenretiable function. So this custom 2D convolution can be used only for inference.

Time for inference using Conv2d and Linear	*Time for inference WITHOUT using Conv2d and Linear*
1.62s	2.99s

Table 4.2: Inference times for the test dataset with batch_size 32

The custom layers performed worse. It had nearly 84% increase in testing time. Furthermore, it is not trainable.