CS6886: Systems Engineering For Deep Learning

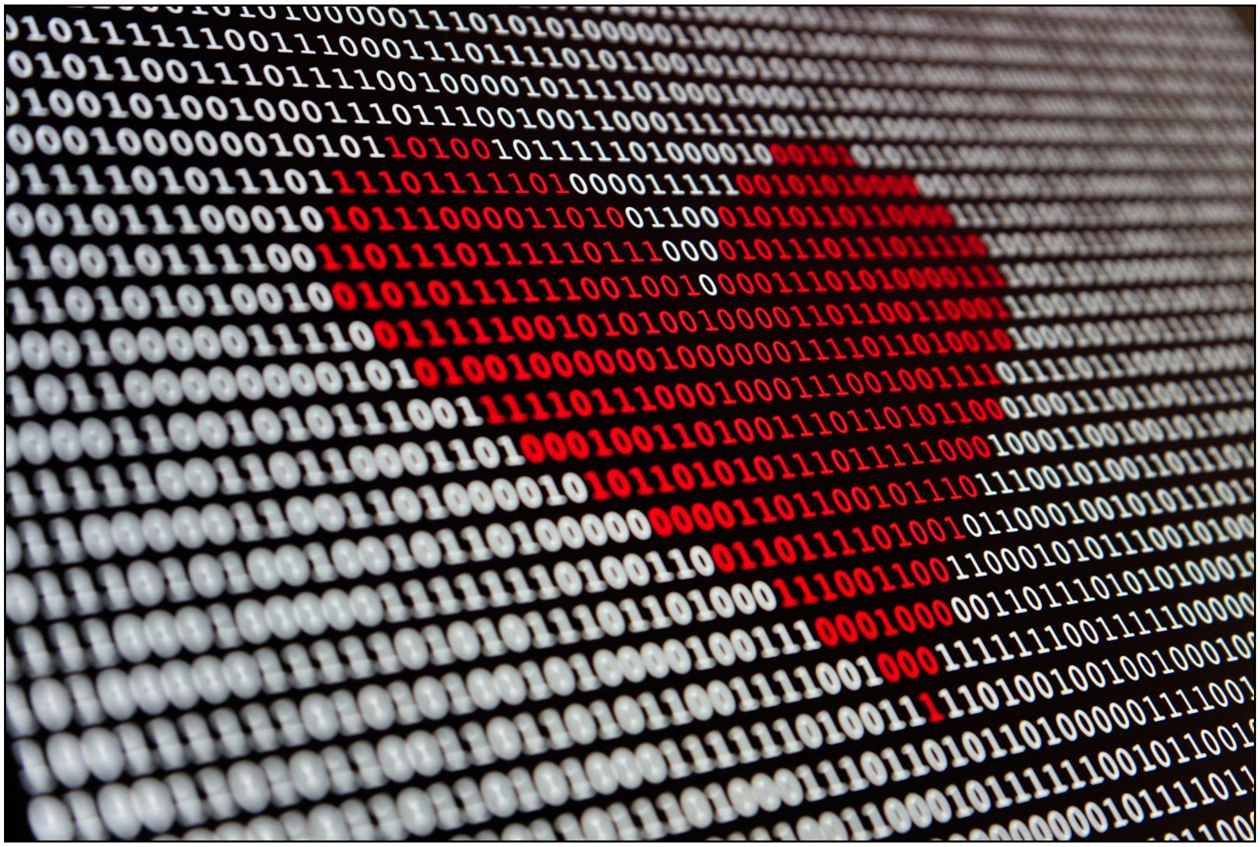
Jose Moti

CE17B118

10/5/2020

Assignment 3

Getting into the Flow of Deep Learning



Introduction

In this assignment, we create a basic DL model with convolutional and fully connected layers. We then tune the model with different hyperparameter orientations and the best one from them is found out with the help of ML-Flow API. **Link** for the detailed problem statement. The Pytorch framework was used for the whole assignment. The whole experiment was done on a single Colab notebook. See the notebook **here** for more details.

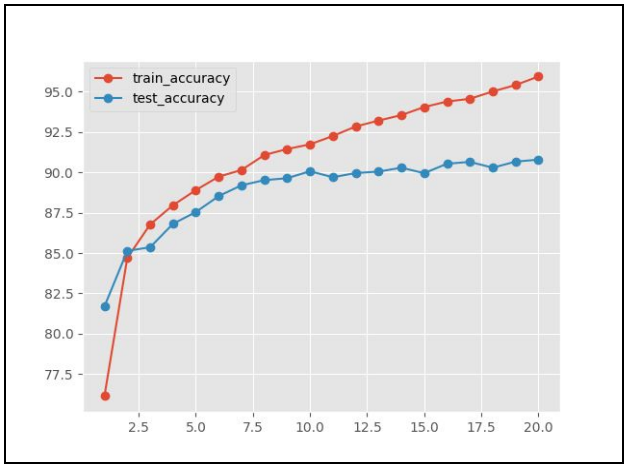
See the last page for the contents in the folder.

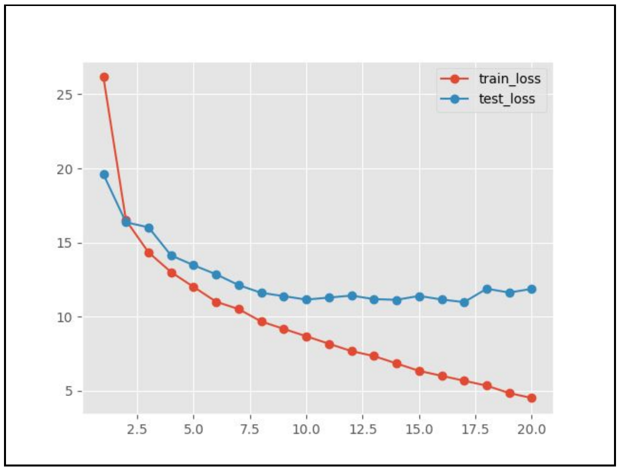
Step-1

A model is initialized and trained with the following parameters and hyperparameters.

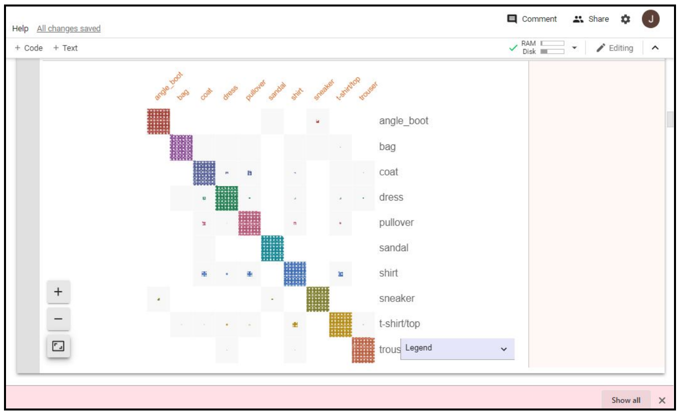
- Convolutional layer 1: 16(3,3) filters
- Convolutional layer 2: 16(3,3) filters
- 2 fully connected layers with 100 and 10 output neurons respectively
- Cross-Entropy loss
- SGD optimizer - learning rate- 0.01 and momentum 0.9
- Batch size - 256
- Xavier initializer for the whole model weights Datasets was imported from torchvision.datasets. 20 epochs were used in training this model to a **train accuracy of 95.9%** and **test accuracy of 90.5%**. The parameters of the final model are saved as a 'step1_model_trained.pth' file. The train and test accuracies as well as the losses are plotted for each epoch and are shown below.

Note: Training loss was divided by 6 in order to compare with test loss since the train dataset has 6 times more data points than the test dataset. Also, the bias was initialized to 0. Default Pytorch initialization was used.





<u>Facets</u>



The x axis gives the true labels while y axis gives the predictive labels. A column contains all images of a given label as mentioned on top of that column and a row gives the images that were predicted as the row prediction label. Only training images were taken.

Some inferences from Facets dive plot on the trained model:

- Considering the column of pullover we can see most of the images in it are correctly predicted but some were predicted as shirts/ coats.
- Considering the row of shirts we can see most of the shirt predictions were actually shirts while a good number of the shirt predictions were actually coats/pullovers/t-shirt.
- Most wrong predictions for pullover, shirts, coats, t shirts classes because of their similarity and are difficult to distinguish.
- The least wrong predictions for bags and sandals because they can be easily distinguished.

Step-2

(10a)

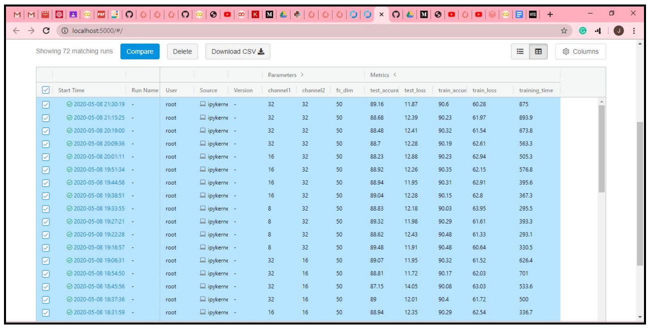
Initially, we change the filter sizes of the convolutional layers, channel sizes of convolutional layers, and the output neurons in the hidden layer. A total of 72 configurations of these hyperparameters are trained and their metrics and params were logged to MLFlow tracking UI. Each of the models was allowed to train until the training accuracy reaches 90%. Different values of each of the hyperparameters used are as given below. The same order of these 5 hyperparameters is used through the rest of this section:

```
conv_layer_1_filter_sizes = [3, 5]

conv_layer_2_filter_sizes = [3, 5]

conv_layer_1_channel_sizes = [8, 16, 32]

conv_layer_2_channel_sizes = [8, 16, 32]

layer_3_sizes = [100, 50]
```

Params1(3,3,8,8,100) gives the required 5 hyperparameters and is taken in the same order as given above.

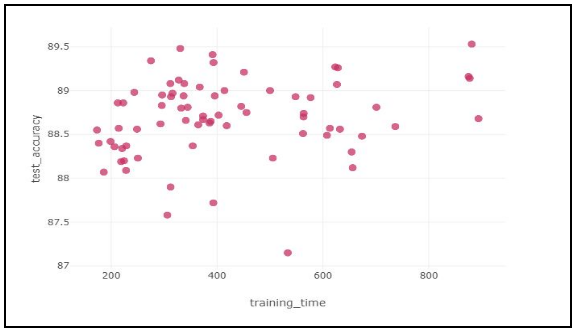
The tracking file for each of these 72 experiments is given under the 'mlruns1' folder. The whole experiment took more than 8 hours. The final mlflow table for all the 72 runs is given below. See the below image for different parameters and metrics column.

For choosing the best model both the test accuracies and the scattered plot of training time vs test accuracy of these 72 configurations taken from mlflow ui. See below.



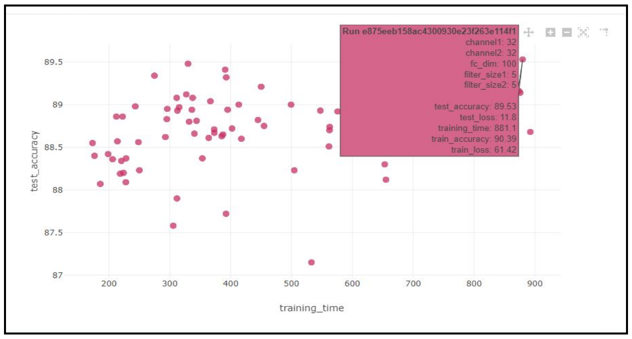
Showing 72 matching runs

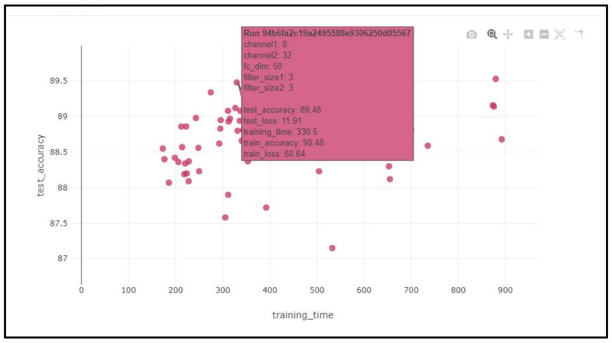
						Parameters			Metrics				
✓	Start Time	Run Name	User	Source	Version	channel1	channel2	fc_dim	test_accura	test_loss	train_accur	train_loss	training_time
✓	2020-05-08 21:30:19	-	root	ipykerne	-	32	32	50	89.16	11.87	90.6	60.28	875
✓	2020-05-08 21:15:25	-	root	ipykerne	-	32	32	50	88.68	12.39	90.23	61.97	893.9
✓	2020-05-08 20:19:00	-	root	ipykerne	-	32	32	50	88.48	12.41	90.32	61.54	673.8
✓	2020-05-08 20:09:36	-	root	ipykerne	-	32	32	50	88.7	12.28	90.19	62.61	563.3
✓	2020-05-08 20:01:11	-	root	ipykerne	-	16	32	50	88.23	12.88	90.23	62.94	505.3
✓	2020-05-08 19:51:34	-	root	ipykerne	-	16	32	50	88.92	12.26	90.35	62.15	576.8
✓	2020-05-08 19:44:58	-	root	ipykerne	-	16	32	50	88.94	11.95	90.31	62.91	395.6
✓	2020-05-08 19:38:51	-	root	ipykerne	-	16	32	50	89.04	12.28	90.15	62.8	367.3
✓	2020-05-08 19:33:55	-	root	ipykerne	-	8	32	50	88.83	12.18	90.03	63.95	295.5
✓	2020-05-08 19:27:21	-	root	ipykerne	-	8	32	50	89.32	11.98	90.29	61.61	393.3
✓	2020-05-08 19:22:28	-	root	ipykerne	-	8	32	50	88.62	12.43	90.48	61.33	293.1
✓	2020-05-08 19:16:57	-	root	ipykerne	-	8	32	50	89.48	11.91	90.48	60.64	330.5
✓	2020-05-08 19:06:31	-	root	ipykerne	-	32	16	50	89.07	11.95	90.32	61.52	626.4
✓	2020-05-08 18:54:50	-	root	ipykerne	-	32	16	50	88.81	11.72	90.17	62.03	701
✓	2020-05-08 18:45:56	-	root	ipykerne	-	32	16	50	87.15	14.05	90.08	63.03	533.6
✓	2020-05-08 18:37:36	-	root	ipykerne	-	32	16	50	89	12.01	90.4	61.72	500
✓	2020-05-08 18:31:59	-	root	ipykerne	-	16	16	50	88.94	12.35	90.29	62.54	336.7



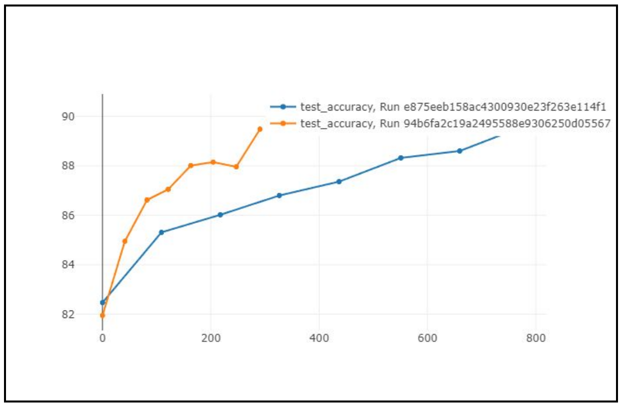
<u>**Inferences**</u>

- The best test accuracy is given by **Params1(5,5,32,32,100)** of 89.53% in 881seconds.
- The second best test accuracy is given by **Params1(3,3,8,32,50)** of 89.48% in 330seconds. See the below images.





- Even though there is a 0.05% difference in accuracy, there is a difference of 9 minutes between the best and second-best model in terms of accuracy. Due to this considerable decrease in time to train and not much trade-off in test accuracy, we will be using **Params1(3,3,8,32,50)** for PART-b. The test accuracy wrt training time of these 2 models given here.



- 68 out of 72 models have test accuracies greater than 88% showing there is not much overfitting occurring in the models.
- No other specific hyperparameter relations found out between the params and metrics. See the colab notebook for all the graphs.

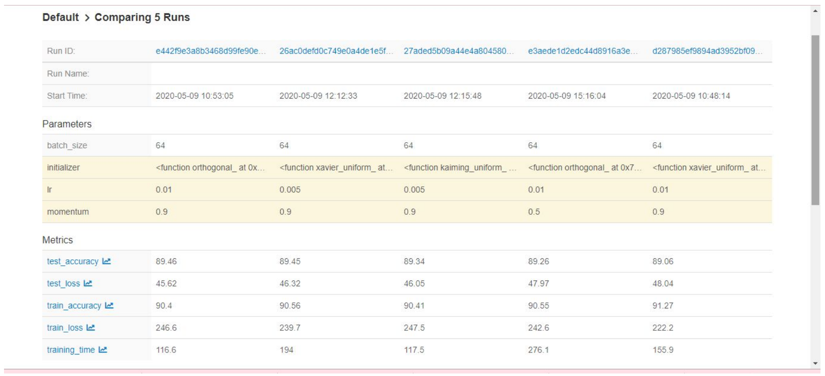
10b

For the second step bias was initialized to 0 for all experiments. Also, a maximum of 50 epochs was used. Ie, if the training will stop after 50 epochs even if the train accuracy doesn't reach 90%. It took about 9 hours to try out the 48 hyperparameter configurations.

```
initializers =
[torch.nn.init.xavier_uniform_,torch.nn.init.kaiming_uniform_,torch.nn.ini
t.orthogonal_]

batch_sizes = [1028,512,256,64]

lrs = [0.01,0.005]

momentums = [0.9,0.5]
```

Params2(torch.nn.init.xavier_uniform_,1028,0.01,0.9) - an example class that is used is used. 1028 was taken instead of 1024 by mistake (doesn't make much difference I think). The tracking models in this experiment given under 'mlruns2_50'.

The screenshot of the best 5 models in terms of test accuracy from mlflow is given below.

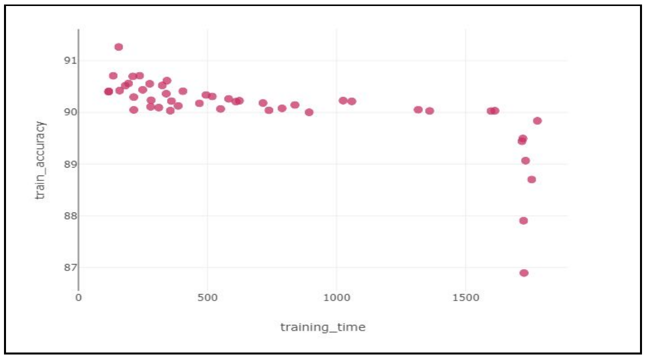


Default > Comparing 5 Runs

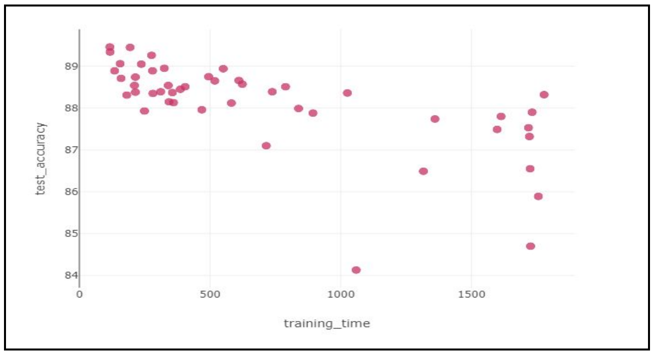
Run ID:	e442f9e3a8b3468d99fe90e...	26ac0defd0c749e0a4de1e5f...	27aded5b09a44e4a804580...	e3aede1d2edc44d8916a3e...	d287985ef9894ad3952bf09...
Run Name:					
Start Time:	2020-05-09 10:53:05	2020-05-09 12:12:33	2020-05-09 12:15:48	2020-05-09 15:16:04	2020-05-09 10:48:14
Parameters					
batch_size	64	64	64	64	64
initializer	<function orthogonal_ at 0x...	<function xavier_uniform_ at...	<function kaiming_uniform_ ...	<function orthogonal_ at 0x7...	<function xavier_uniform_ at...
lr	0.01	0.005	0.005	0.01	0.01
momentum	0.9	0.9	0.9	0.5	0.9
Metrics					
test_accuracy	89.46	89.45	89.34	89.26	89.06
test_loss	45.62	46.32	46.05	47.97	48.04
train_accuracy	90.4	90.56	90.41	90.55	91.27
train_loss	246.6	239.7	247.5	242.6	222.2
training_time	116.6	194	117.5	276.1	155.9

Inferences

- Out of 48 experiments, 7 didn't reach 90% train accuracy by the 50th epoch. See plot.

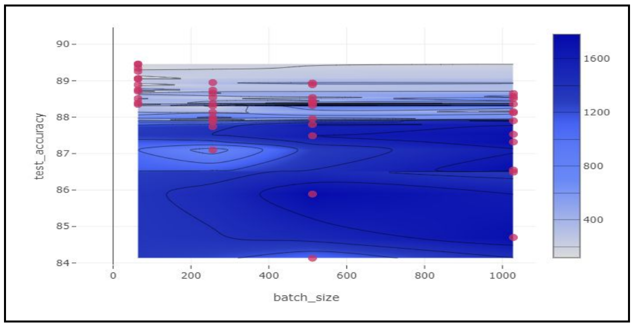


- Final *test accuracy vs total train time* plot given.



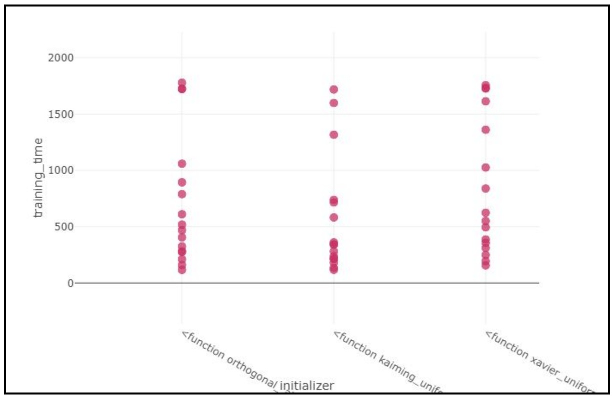
Best hyperparameter configuration **Params2(orthogonal initializer, 64, 0.01, 0.9).**

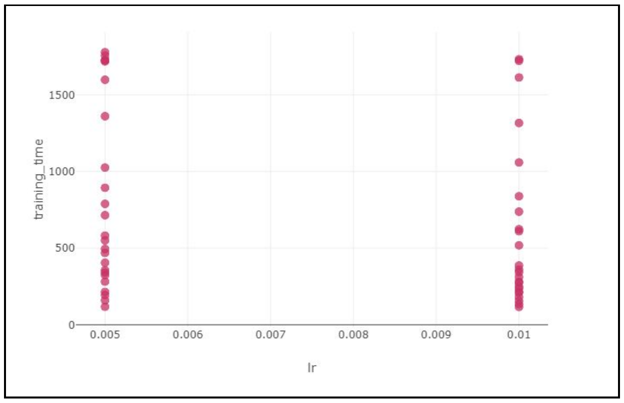
- The contour plot of **batch size** with the final test accuracy on the y-axis and total training time on the z-axis is given below.



Best accuracy, as well as lower training time, is given when the batch size of 64 is used compared to that of 256, 512, and 1028. This is because since there are only 10 classes, taking 64 images in each batch helps to capture the features as a whole almost the same as taking higher batch size values. Also, this would make the model reach 90% accuracy in much lesser epochs since more batches can be included in an epoch.

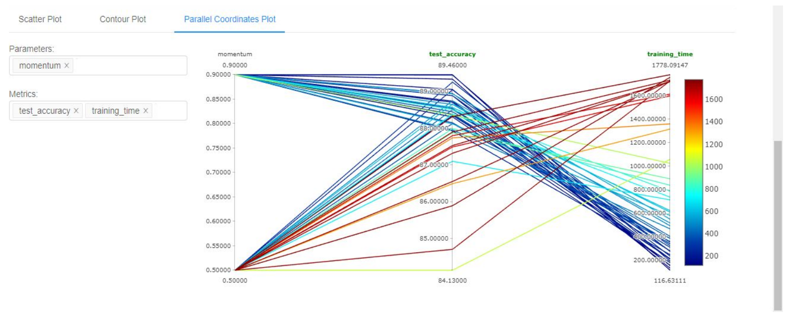
- Similar results were observed for the different **initializers**. Also, similar results were observed for the two different **learning rates** used.



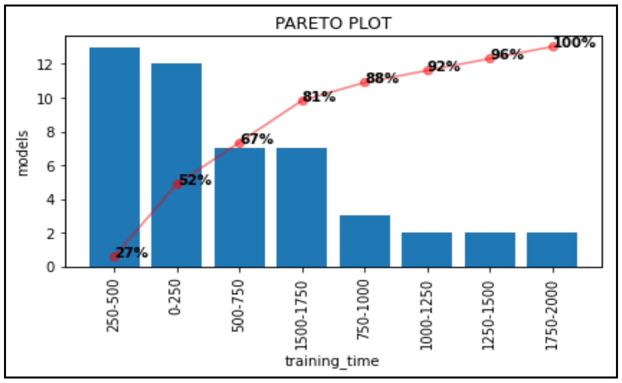


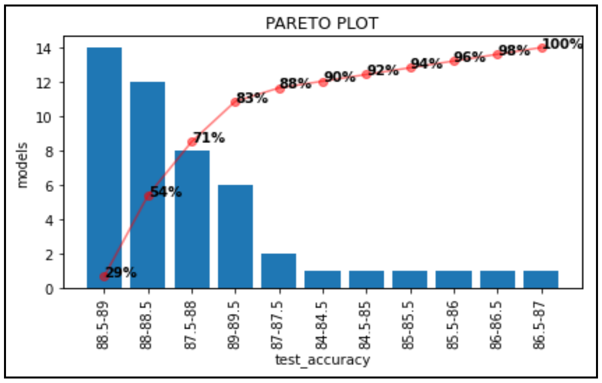
This may be because we have just tried two learning rates and these learning rates might be closer than we thought for our models thus giving similar results.

- We can see a clear gap while using a higher as well as a lower **momentum**.This is because momentum helps the learning curve to have extra momentum along the direction of learning which makes the training process much faster and efficient.



Pareto plot with number of models on y axis is given below:



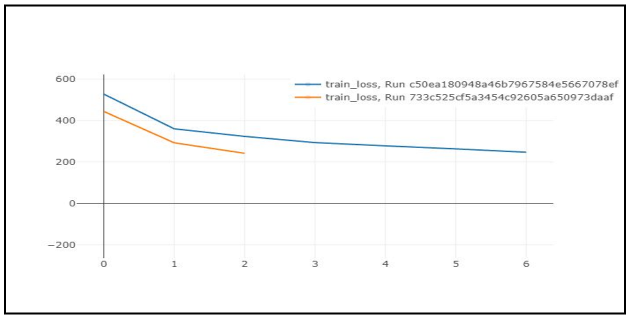


12d

Even though we did about 120 different configurations of hyperparameters some configurations such as running with high and low learning rates are not tried out. So in this section, we will do the extra configurations which are given in the 12d section and train till 90% reached. Otherwise, if the model is found to be not learning it is terminated.

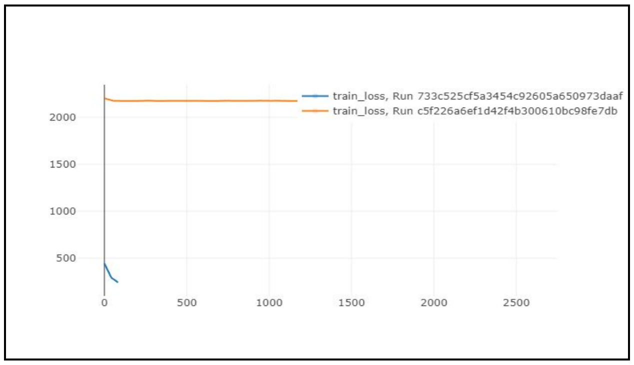
Learning Rate is too High

The plots for train_loss vs epochs for two learning rates are given here.



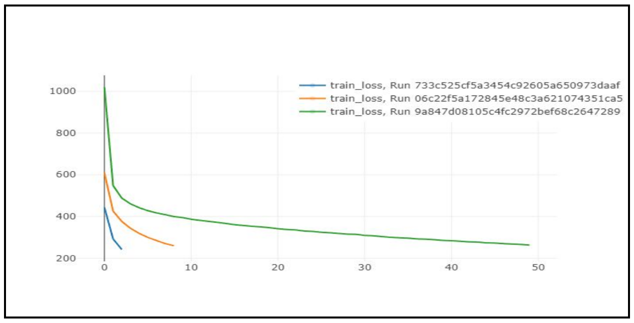
Orange is with lr=0.1 in the first image
Orange is with lr=0.5 in the second image
Blue is with lr=0.01

The model finds the solution faster with a higher learning rate. But increasing the learning rate above a certain value can cause a problem of exploding gradients which makes the model unable to learn from its weights thus giving a higher loss value. This happens while we use a learning rate of 0.5 in our model. The higher loss value shows that the model is not learning. Also, 0.01 helps to find better solutions towards the end by taking smaller steps compared to 0.1.

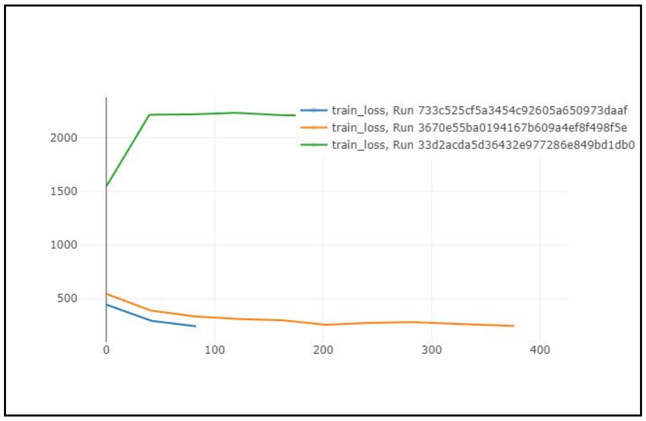


Learning Rate is too low

lr=0.0001(green) compared with lr=0.001(orange) and lr=0.01(blue). Using lower learning rates makes training time much higher from less than 10 epochs to 50 epochs.

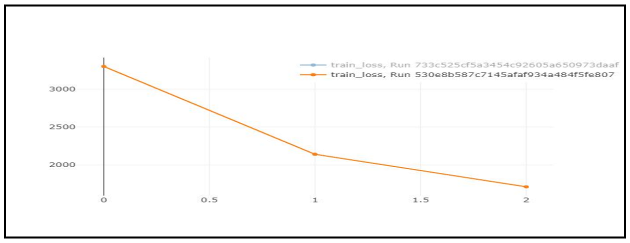


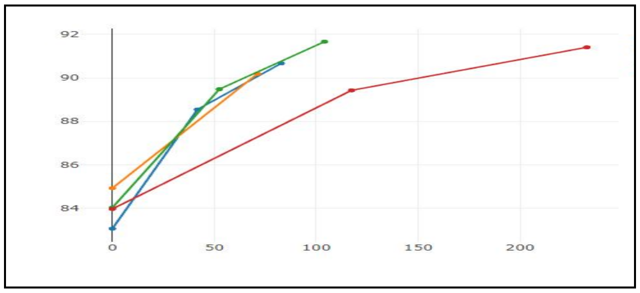
Momentum is too high



Using momentum=0.999, momentum=0.99 and momentum=0.9 models are trained and compared. 0.99 puts additional momentum helping the model to train faster in much lesser epochs but keeping a much higher momentum of 0.999 makes the model unable to learn maybe because gradients are exploded in the direction momentum was imparted.

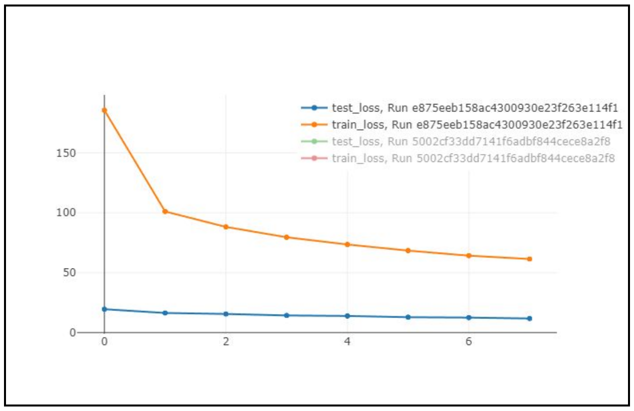
Batch size is too small





Training loss vs epoch after using a batch size of 8 is given above. The model reaches a training accuracy of 90% in just 2 epochs. Here we are not comparing loss between different batch sizes because they were not normalized and losses come out in different ranges. So train accuracies vs time plot are also shown below for different batch sizes. Batch sizes of 64, 32, 16, 8 are used.

Overfitting due to large model



No such specific overfitting model is found. The maximum difference between the test and train accuracy of a model was 3% and it was with Params1(5,3,32,16,50) and the train and test loss vs epochs graph is given here. There were denser models than this that gave better results so this cannot be considered explicitly as a case of overfitting.

Step-3

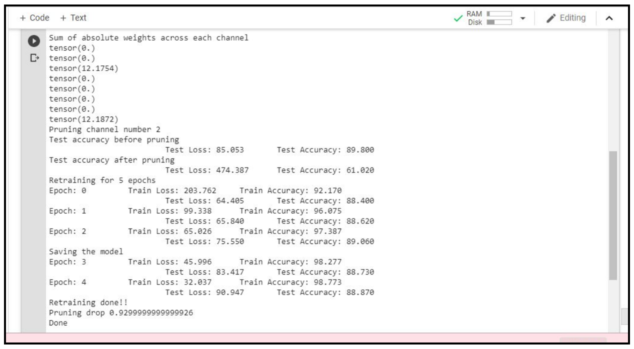
Heuristic

Pruning of the trained CNN is done in this step. Pruning is done by a simple heuristic of pruning filters with lower L1 norm. Paper Link. Here, the sum of the absolute values of the weights are taken and compared across different channels in a layer. The channel with lowest sum is pruned. Pruning is emulated by making the weights and biases 0 and setting the grad values 0 at each training step. A new train function 'train_prune' was used for this. See code in colab - step3.

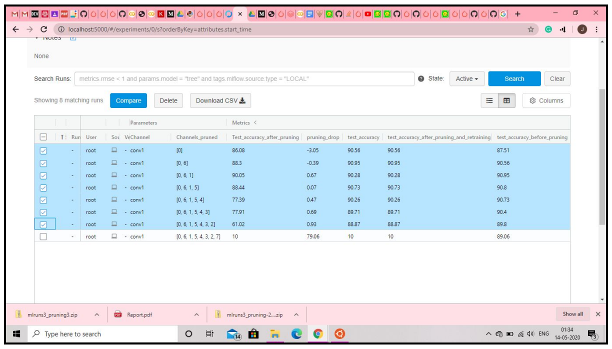
Each of the 4 layers are pruned - one channel at a time for conv layers and 1 neuron at a time for fc layers. *Params1(3,3,8,32,50)* and *Params2(orthogonal initializer, 64, 0.01,0.9)* were used to create the model. The layerwise pruning results are given below. For each channel/neuron pruned, *test accuracy before pruning*, *test accuracy after pruning*, *test accuracy after pruning and retraining* and *pruning drop* is logged as metrics. Pruning was done till there was a pruning drop of more than 1%.

Conv1

The results were astonishing. There were a total of 8 channels out of which 7 channels were pruned with a final test accuracy of **89.060 %.** This model was saved for pruning on further layers. The output sample as well as the mlflow table is given below.



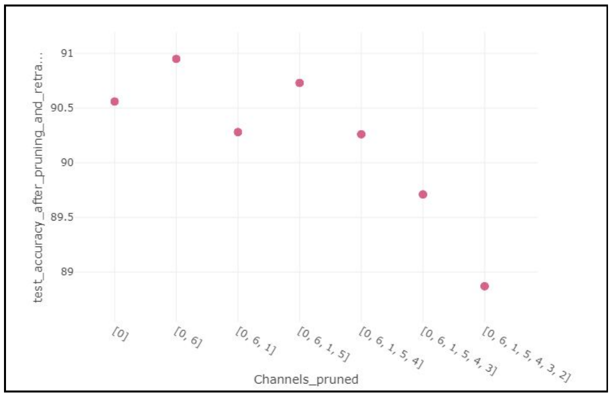
```
Sum of absolute weights across each channel
tensor(0.)
tensor(0.)
tensor(12.1754)
tensor(0.)
tensor(0.)
tensor(0.)
tensor(0.)
tensor(12.1872)
Pruning channel number 2
Test accuracy before pruning
                         Test Loss: 85.053       Test Accuracy: 89.800
Test accuracy after pruning
                         Test Loss: 474.387      Test Accuracy: 61.020
Retraining for 5 epochs
Epoch: 0         Train Loss: 203.762     Train Accuracy: 92.170
                         Test Loss: 64.405       Test Accuracy: 88.400
Epoch: 1         Train Loss: 99.338      Train Accuracy: 96.075
                         Test Loss: 65.840       Test Accuracy: 88.620
Epoch: 2         Train Loss: 65.026      Train Accuracy: 97.387
                         Test Loss: 75.550       Test Accuracy: 89.060
Saving the model
Epoch: 3         Train Loss: 45.996      Train Accuracy: 98.277
                         Test Loss: 83.417       Test Accuracy: 88.730
Epoch: 4         Train Loss: 32.037      Train Accuracy: 98.773
                         Test Loss: 90.947       Test Accuracy: 88.870
Retraining done!!
Pruning drop 0.9299999999999926
Done
```

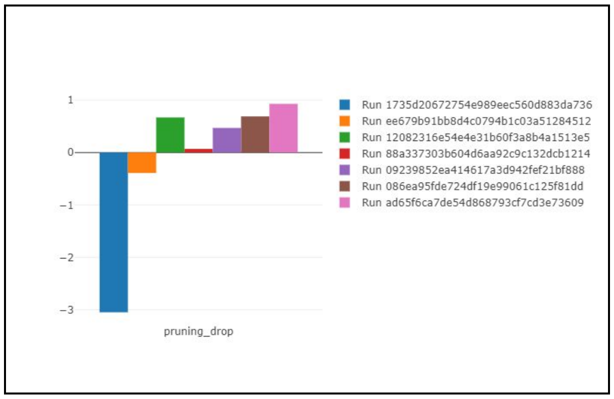


				Parameters		Metrics				
	Run	User	Sou	VeChannel	Channels_pruned	Test_accuracy_after_pruning	pruning_drop	test_accuracy	test_accuracy_after_pruning_and_retraining	test_accuracy_before_pruning
✓	-	root		- conv1	[0]	86.08	-3.05	90.56	90.56	87.51
✓	-	root		- conv1	[0, 6]	88.3	-0.39	90.95	90.95	90.56
✓	-	root		- conv1	[0, 6, 1]	90.05	0.67	90.28	90.28	90.95
✓	-	root		- conv1	[0, 6, 1, 5]	88.44	0.07	90.73	90.73	90.8
✓	-	root		- conv1	[0, 6, 1, 5, 4]	77.39	0.47	90.26	90.26	90.73
✓	-	root		- conv1	[0, 6, 1, 5, 4, 3]	77.91	0.69	89.71	89.71	90.4
✓	-	root		- conv1	[0, 6, 1, 5, 4, 3, 2]	61.02	0.93	88.87	88.87	89.8
	-	root		- conv1	[0, 6, 1, 5, 4, 3, 2, 7]	10	79.06	10	10	89.06

The final layer is when the 8th channel (out of 8) itself is pruned, destroying the model and thus it is not used for comparisons. The order of pruning of 7 channels are [0,6,1,5,4,3,2].

The test accuracies after pruning and retraining as well as pruning drop in percentage for each channel pruned is given below.



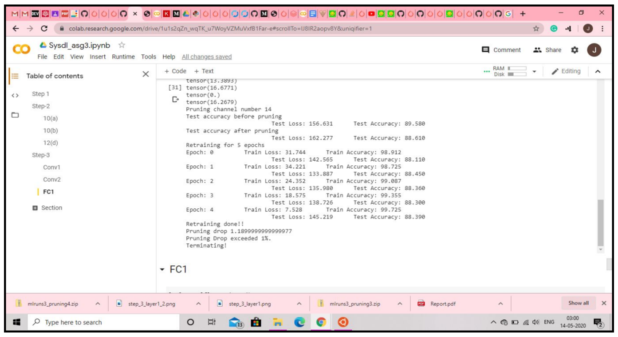


Inferences:

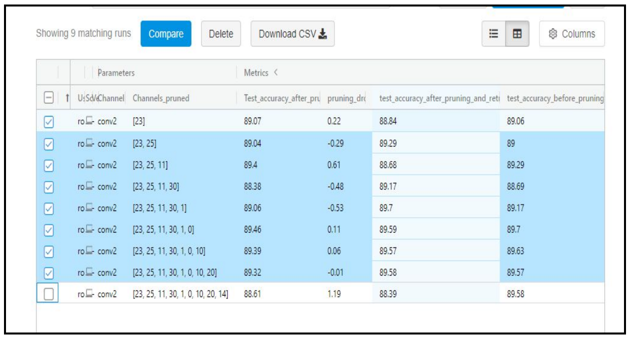
- Starting with a pretrained model with test accuracy of 87.510 % we were able to reach **89.060 %** after pruning 7 out of 8 channels in the first layer, retraining in each session.
- Shows how overfitting the parameters of the model are. We were using 8 channels when we needed only 1.
- By pruning we reduce the FLOPs by a large value, 7 in this case making the inference much faster.

Conv2

There were a total of 32 channels out of which 8 channels were pruned (25 %) with a final test accuracy of **89.580 %.** This model was saved for pruning on further layers. Sample output and Mlflow table given below.

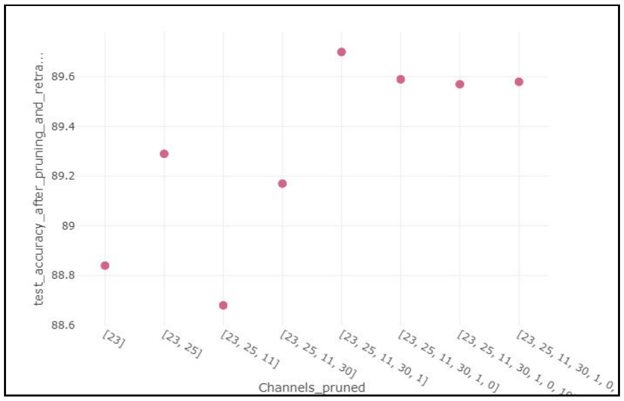


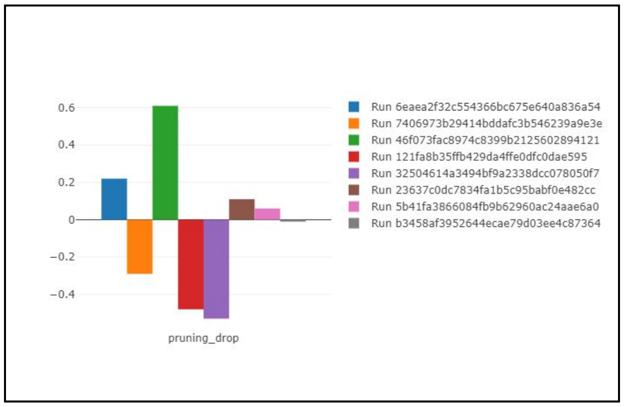
```
tensor(13.3893)
[31] tensor(16.6771)
tensor(0.)
tensor(16.2679)
Pruning channel number 14
Test accuracy before pruning
                         Test Loss: 156.631      Test Accuracy: 89.580
Test accuracy after pruning
                         Test Loss: 162.277      Test Accuracy: 88.610
Retraining for 5 epochs
Epoch: 0         Train Loss: 31.744      Train Accuracy: 98.912
                         Test Loss: 142.565      Test Accuracy: 88.110
Epoch: 1         Train Loss: 34.221      Train Accuracy: 98.725
                         Test Loss: 133.887      Test Accuracy: 88.450
Epoch: 2         Train Loss: 24.352      Train Accuracy: 99.087
                         Test Loss: 135.980      Test Accuracy: 88.360
Epoch: 3         Train Loss: 18.575      Train Accuracy: 99.355
                         Test Loss: 138.726      Test Accuracy: 88.300
Epoch: 4         Train Loss: 7.528       Train Accuracy: 99.725
                         Test Loss: 145.219      Test Accuracy: 88.390
Retraining done!!
Pruning drop 1.1899999999999977
Pruning Drop exceeded 1%.
Terminating!
```



		Parameters		Metrics			
	U:S:V	Channel	Channels_pruned	Test_accuracy_after_pru	pruning_drc	test_accuracy_after_pruning_and_reti	test_accuracy_before_pruning
✓	ro	conv2	[23]	89.07	0.22	88.84	89.06
✓	ro	conv2	[23, 25]	89.04	-0.29	89.29	89
✓	ro	conv2	[23, 25, 11]	89.4	0.61	88.68	89.29
✓	ro	conv2	[23, 25, 11, 30]	88.38	-0.48	89.17	88.69
✓	ro	conv2	[23, 25, 11, 30, 1]	89.06	-0.53	89.7	89.17
✓	ro	conv2	[23, 25, 11, 30, 1, 0]	89.46	0.11	89.59	89.7
✓	ro	conv2	[23, 25, 11, 30, 1, 0, 10]	89.39	0.06	89.57	89.63
✓	ro	conv2	[23, 25, 11, 30, 1, 0, 10, 20]	89.32	-0.01	89.58	89.57
☐	ro	conv2	[23, 25, 11, 30, 1, 0, 10, 20, 14]	88.61	1.19	88.39	89.58

The test accuracies after pruning and retraining as well as pruning drop in percentage for each channel pruned is given below.

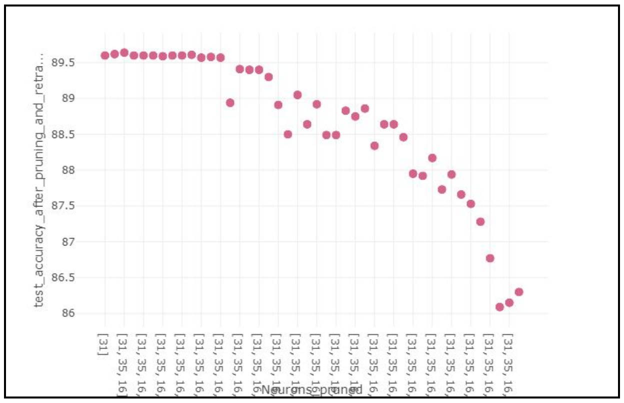


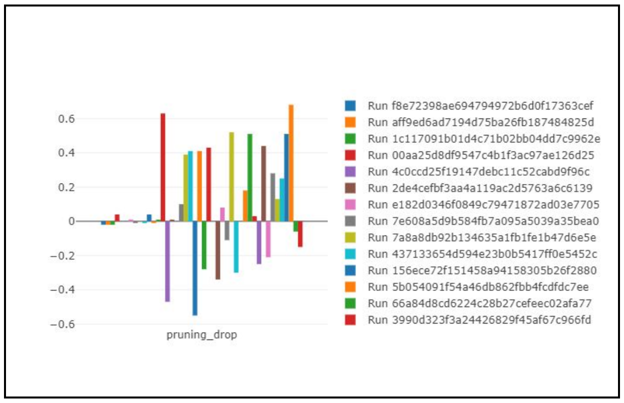


FC1

Starting with a test accuracy of **89.590 %**, 44 out of the 50 neurons in the hidden layer were pruned by making the respective biases and weights 0. The final test accuracy was found to be **86.300 %** which is arguably better considering 88 percent of parameters in that layer were removed. The weight was of dimension [50,18432] and bias of dimension [50]. Pruning is done rowwise for the weights which is equivalent to removal of neurons in the hidden layer one by one.

The test accuracy after pruning and retraining in each of these 44 pruning cases is given below. Also the pruning drops in each of these cases is given.





- The test accuracy goes below 87 % after the 41st row pruning. Thus 40/50 rows were pruned by maintaining a test accuracy of 87%.

FC2

- In FC1, the neurons from the hidden layer (50 neurons) are made 0 in each step. This is because the input layer contains 18432 neurons and trying to make these neurons 0 one-by-one is not possible.
- So in FC2, I made the weights corresponding to the removed hidden layer neurons 0. I also checked whether any additional neurons could be pruned from this hidden layer which gave a negative result. So only a max of 44 neurons could be removed out of 50 from the hidden layer.
- Also the output neurons could not be pruned as all 10 of them are needed for predicting the output.

Overall

- 7/8 channels removed from conv1.
- 8/32 channels from conv2
- 44/50 neurons from hidden layers.

The final models saved from each of the steps has following parameters.

- **Step1 - Params1(3,3,16,16,100), Params2(orthogonal initializer, 64, 0.01, 0.9)**
- **Step 2 - Params 1(3,3,8,32,50), Params2(orthogonal initializer, 64, 0.01, 0.9)**
- **Step 3 - Params1(3,3,1,24,6), Params2(orthogonal initializer, 64, 0.01, 0.9)**

Step-4

Considering one image,

Input - (C, H,W), Filter1 - (M_1,C,R_1,S_1), Filter2 - (M_2,M_1,R_2,S_2)

Hidden layer neurons - F_1 , Output layer neurons - F_2

Addition and multiplications taken as 1 FLOP each. Also for Relu operation 1 is considered as 1 FLOP.

For conv1,

Multiplication - $CR_1S_1(H-R_1+1)(W-S_1+1)M_1$

Bias - $(H-R_1+1)(W-S_1+1)M_1$

RELU - $(H-R_1+1)(W-S_1+1)M_1$

FLOPS - $(CR_1S_1+2)(H-R_1+1)(W-S_1+1)M_1$

For conv2,

FLOPS - $(M_1R_2S_2+2)(H-R_1-R_2+2)(W-S_1-S_2+2)M_2$

For fc1,

FLOPS - $(H-R_1-R_2+2)(W-S_1-S_2+2)M_2F_1 + F_1 + F_1$ (for weight, bias and RELU each)

For fc2,

FLOPS - $(F_1F_2 + F_2)$ (no RELU after final layer)

Put C=1, (R_1,R_2,S_1,S_2)=3, (H,W)=28, F_2=10 , for the complete network

TOTAL FLOPS - $[7436M_1] + [576(9M_1+2)M_2] + [576M_2F_1+2F_1] + [10F_1+10]$

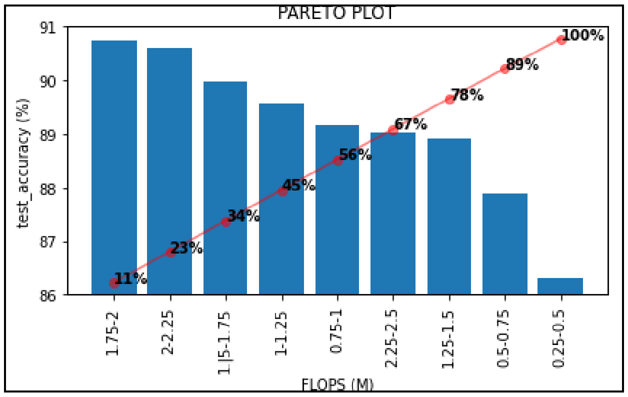
Where M_1 is number of channels in conv1, M_2 number of channels in conv2 and F_1 is number of neurons in the hidden layer.

The total number of flops of the final models after each step with the given M1, M2, F1 values are given as table below. Code is given as flops.py.

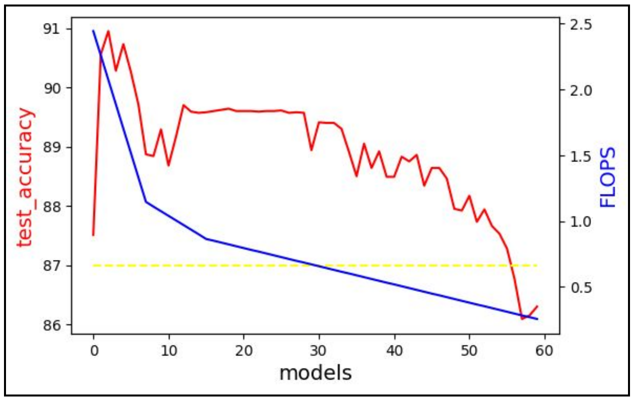
	M_1	M_2	F_1	conv1	conv2	fc1	fc2	Total(~)
Step 1	16	16	100	118976	1345536	921800	1010	2.387 M
Step 2	8	32	50	59488	1363968	921700	510	2.346 M
Step 3	1	24	6	7436	152064	82956	70	0.243 M

The **Pareto Plot** of flops and test accuracy for each of the models is given here.

Average of test accuracy taken in y axis. Using flops 0.75M - 1M would be the best.



Apart from the last 4 models remaining all have an accuracy greater than 87%. This model has **1 channel in conv1, 24 channels in conv2 and 10 hidden layer neurons**. This new model can be created by Params1(3,3,1,24,10). The flops table for this model is given below.



Best model	1	24	10	7436	152064	138620	110	0.298 M

Inferences

- The drop in flops is sharper when removing channels from the initial layer compared to others.
- The inference flops decrease about 10 times compared to only a 4% decrease in test accuracy. This can make a huge difference while inferring a large number of images or inferring from real time data.

For the next part I have taken F1=6 instead of 10 because I have saved only the final model. It gives an accuracy of 86.3% slightly lower than 87%.

Inference time

- The weights of the pruned model are loaded and the inputs are unfolded for convolution operation using 'torch.nn.functional.unfold'.
- Then the forward pass of the whole model was done with matrix multiplications and Relu operations only.
- Also another conv model was created with new pruned dimensions and its weights were updated with the learned weights.
- They are compared to the model in Step1 for the complete testloader. See Colab TIME Compare in Step -4.

	Pruned model GEMM	Step1 model	Step 3 model	Model with pruned dimensions
Inference time(s)	2.168 s	2.614s	2.457 s	1.397 s

- Pruned models do evaluation in less time due to lesser flops.
- Writing the weights into conv and fc weights and evaluating takes lesser time than doing GEMM with pruned weights. This may be because Pytorch does convolution in a much more optimized way compared to unfold and matrix multiply.

Inference on GPU - (Bonus)

Inference time on GPUs	0.953s	0.844s	0.840s	0.780s

- The optimized code for GEMM took more time compared to other models. This is because much better optimizations could be done on conv operations such as dataflows which is more efficient than optimizing the matmul and unfold operations. The pytorch takes care of dataflow optimizations for conv layers for GPUs.

Catch-22 Paradox -(Bonus)

Autonomous cars need more and more real world experience in order to become efficient. But they will only get access to the real world once they are highly efficient. For example, these autonomous cars need to learn by itself how to drive on public roads (real world would have more noisy data than simulations considering different environments). But these autonomous cars can be used on public roads once they are proven efficient. Talking fully autonomous here, not as in Tesla co-pilot. This can be considered paradoxical.

Folder arrangement

- Whole code written in **full_code.ipynb.**
- **complete** folder is recommended as models are used from other steps too.
- 3 mlflow folders for step2 and 1 combined folder for step3.
- All models and folders named appropriately to understand.

References

- Pytorch documentation
- Stack overflow
- Medium

***********THANK YOU***********