Report

Assignment-2 DLCV 2023

Q1) Transformer network

a) implemented. In “vit.py” file.

b) Training on CIFAR-10

- A **validation set** is formed from training data with 5000 images.
- These are the model parameters used for training.

```
model = visionTransformer(token_dim=48,
                          patch_size=4,
                          image_size=32,
                          n_attention_layers=4,
                          multihead_attention_head_dim=48,
                          multihead_attention_n_heads=8,
                          multilayer_perceptron_hidden_dim=512,
                          dropout_p=0.3,
                          n_classes=10) #.to(device)

optimizer =optim.Adam(model.parameters(),lr=1e-3)
criterion = nn.CrossEntropyLoss()
```

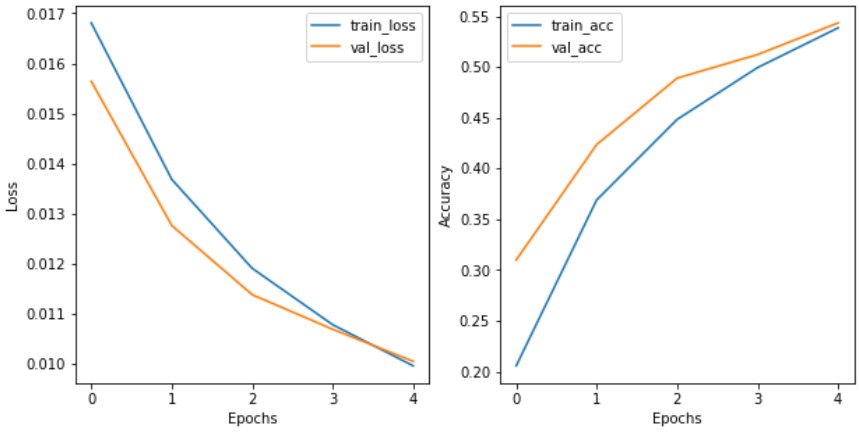
- I trained it on 2 GPUs with dataParallelism. And batch size of 128.
- First when I trained with dropout = 0.3 , and no other regularisation or data augmentation. I observed that validation accuracy became stagnant at 55%. While the model overfitted to training data.
- I used a weight decay of 1e-3. I experimented with several values of weight decay, dropout. It was noticed that in this case, **dropout of 0.65 and weight decay of 5e-4** were giving the best results, with a learning rate of 1e-3.

- **Data augmentation:**
  Random horizontal flip and random rotation up 10 degrees are used.
  Shear did not give good results with 10 degrees angle.

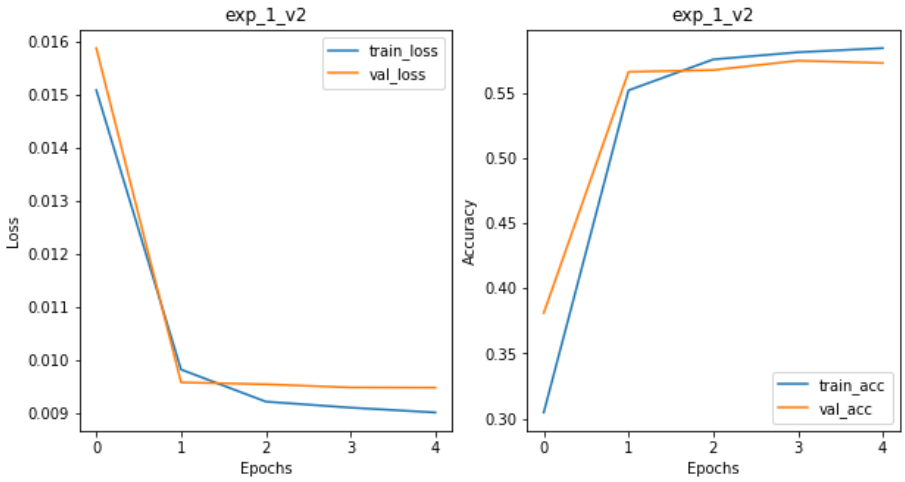
- These are experimentation details:
  (Here i am only showing the best runs that led to final version of model)

Note: loss value is divided by the size of dataset i.e. per datapoint

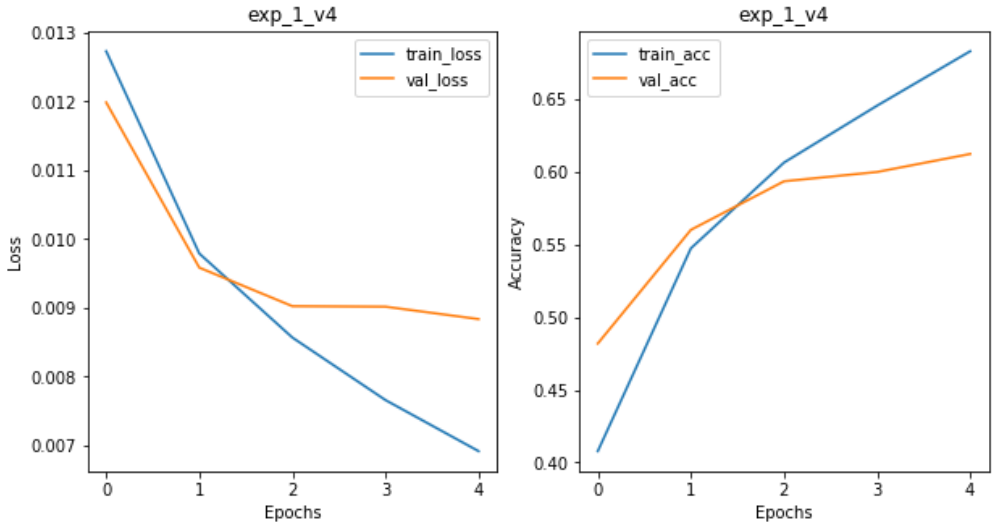
Initial 5 epochs.



- Run 5 more epochs with lr = 0.00001
- It started overfitting



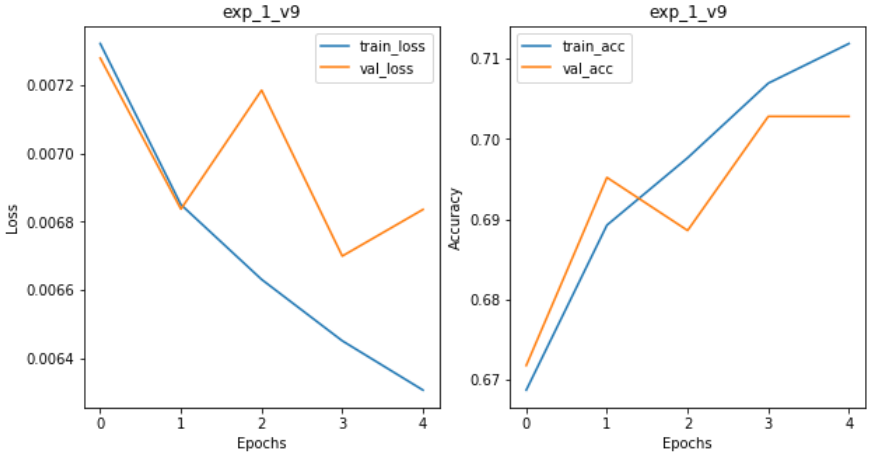
- Here, Added weight decay = 1e-3



- data augmentation is added.

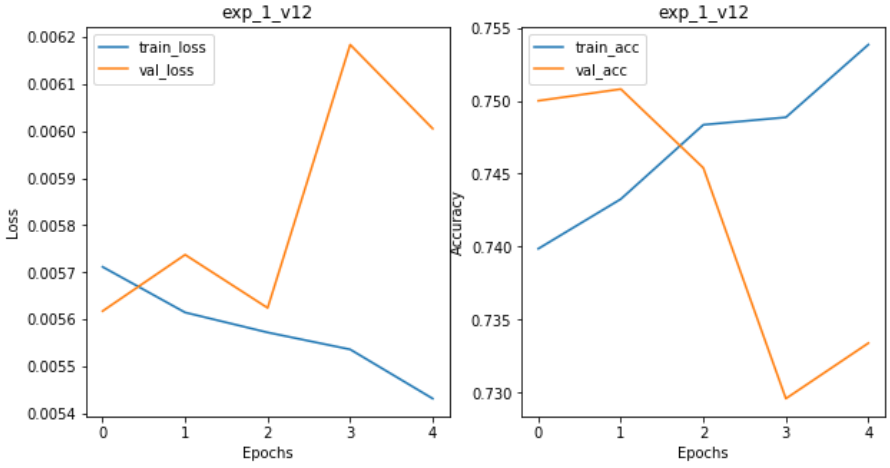
```
transforms.RandomHorizontalFlip(),
transforms.RandomRotation(10),
```

- Improved accuracy,



-

```
# CONTROL PANEL
DROUP_OUT = 0.65
WEIGHT_DECAY = 5e-4
name = 'exp_1_v12'
```

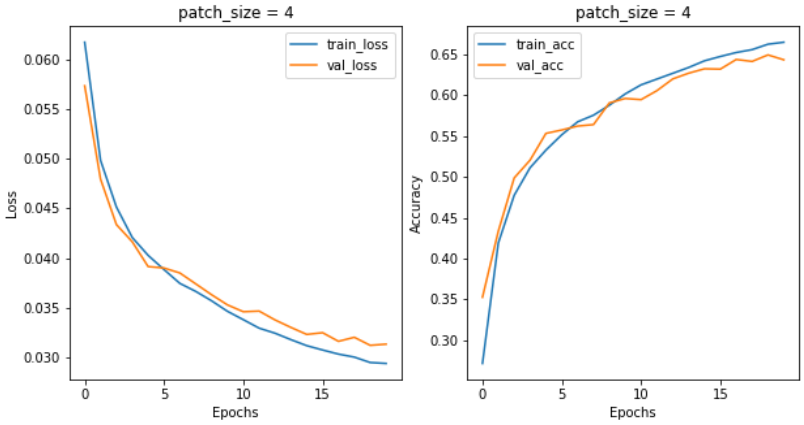


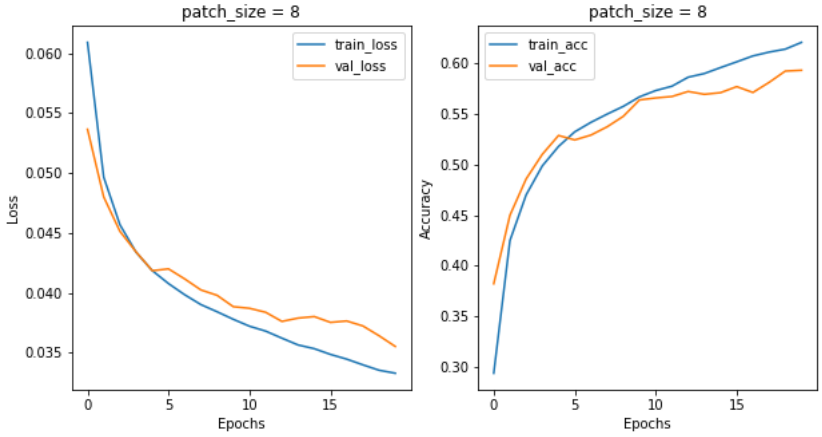
- Here i stopped.
- Tested on test set
  Test loss : 0.00645
  Final Test accuracy : 0.725

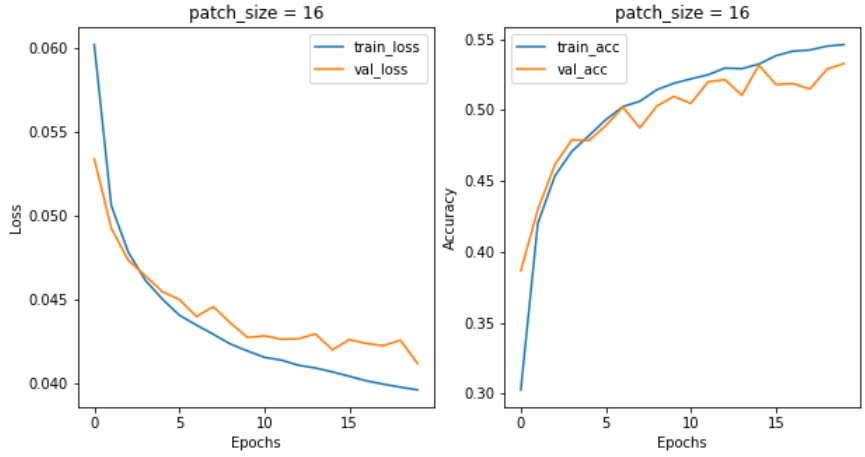
c) Experiment -2

Non-overlapping patches

I trained each for 20 epochs, due to resource constraint i did not train till the optimum.





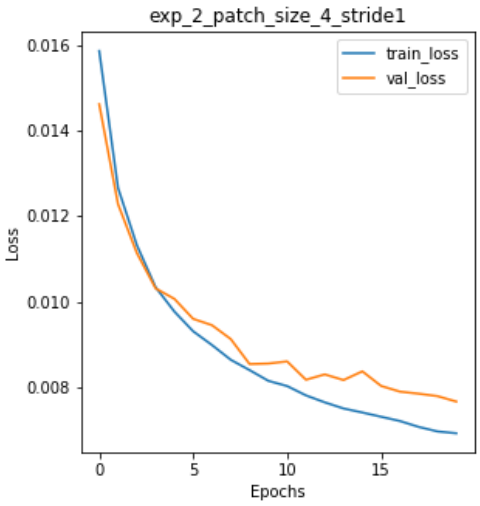


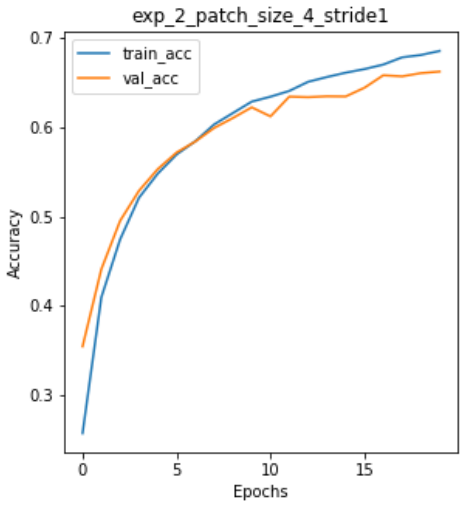
- Here, in the first 20 epochs patch size of 4 gave best results.
- Within these epochs, increased patch size decreased the performance.

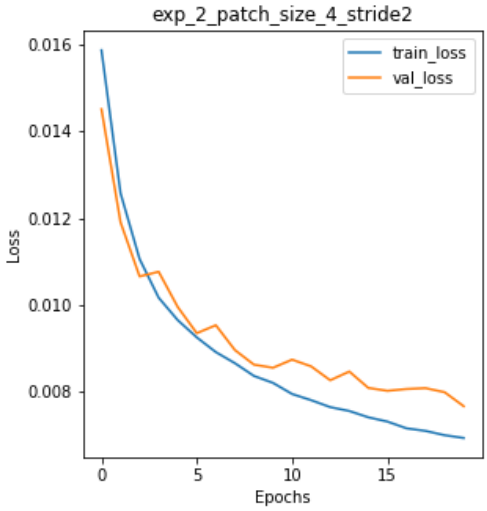
With overlapping patches

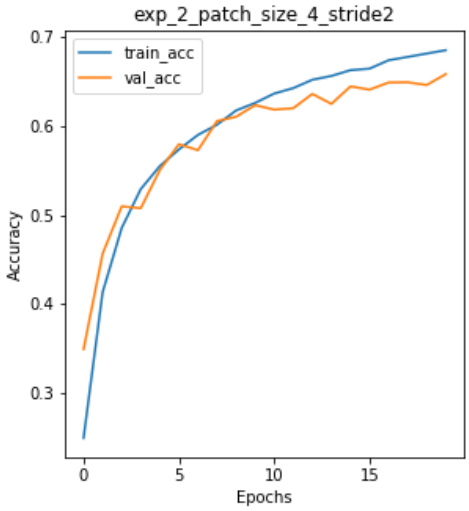
Patch and stride values are written in title of each plot.
(stride value less than patch size causes overlap)

- I tried 2 cases, a) with stride half of patch size b) 1/4th of patch size

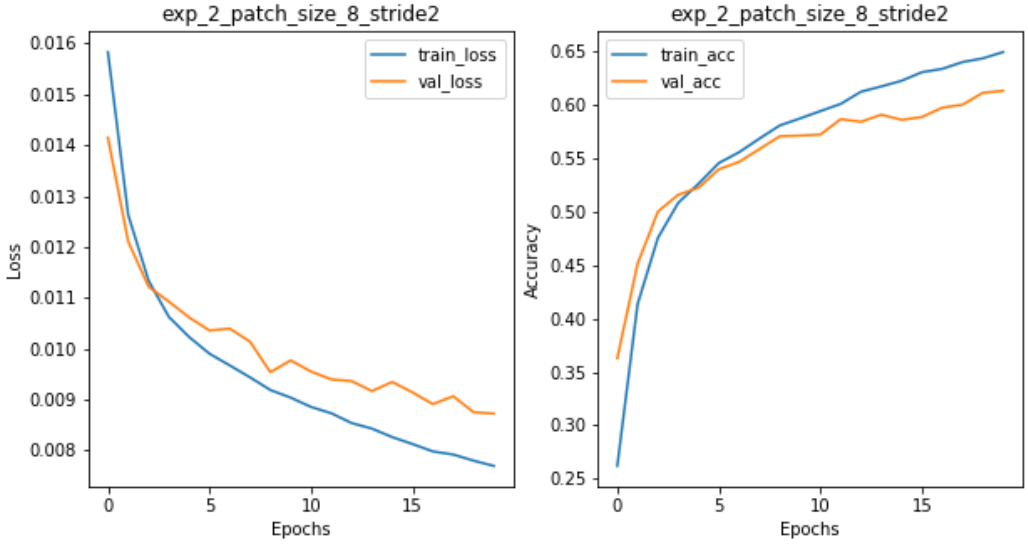


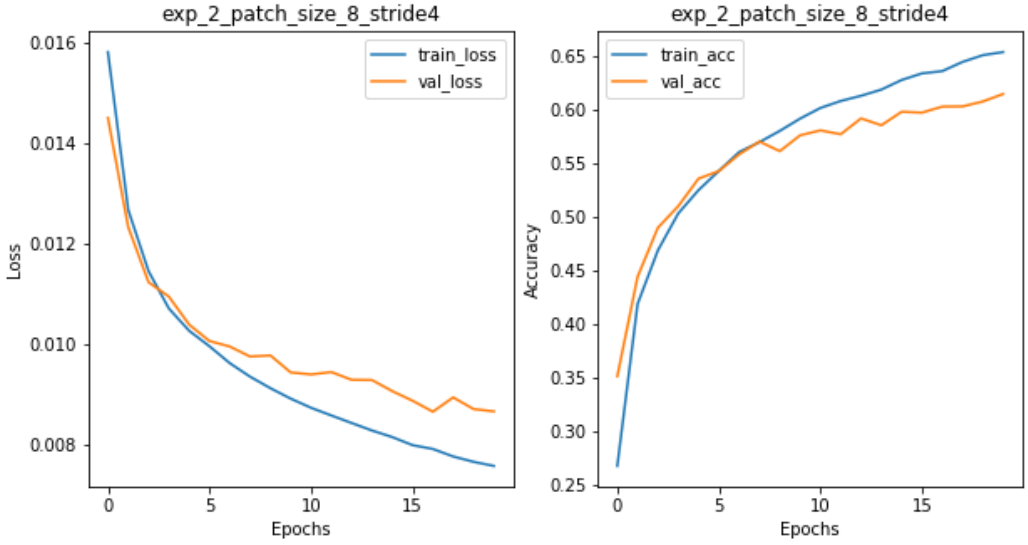




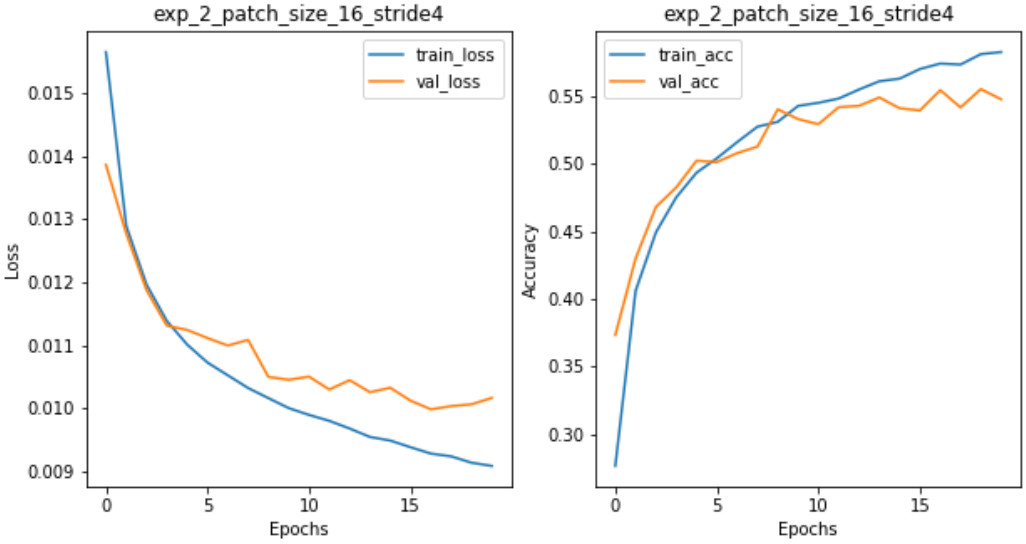


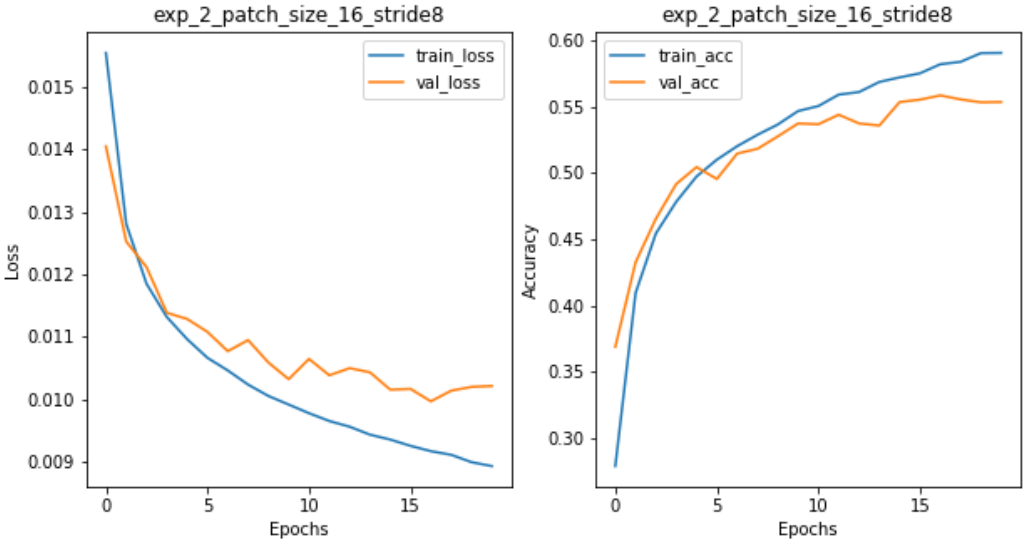
- Above 2, performed better than without overlap case.





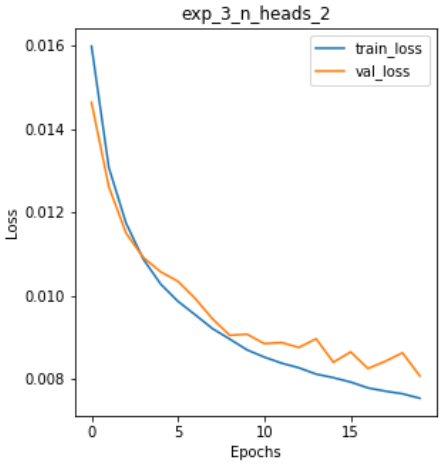
- These also permed better than without overlap case.

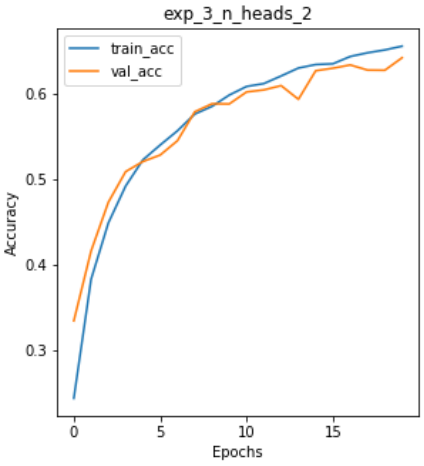


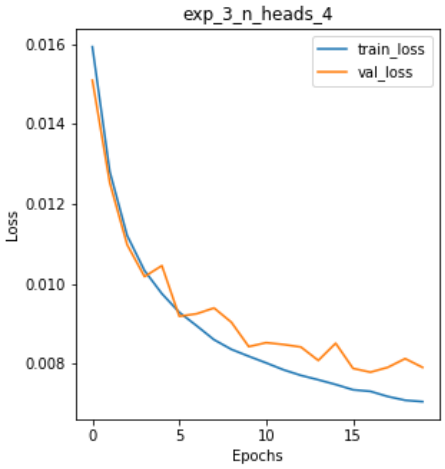


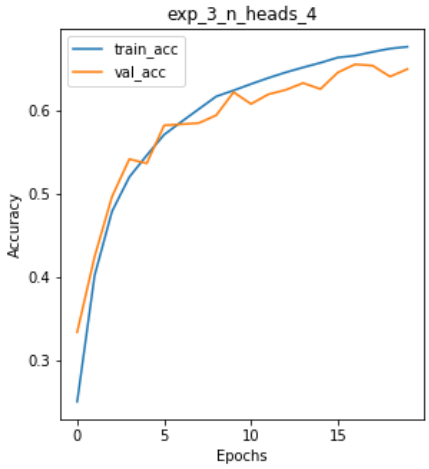
- **Overlapping patches are giving better results than non overlapping patches.**

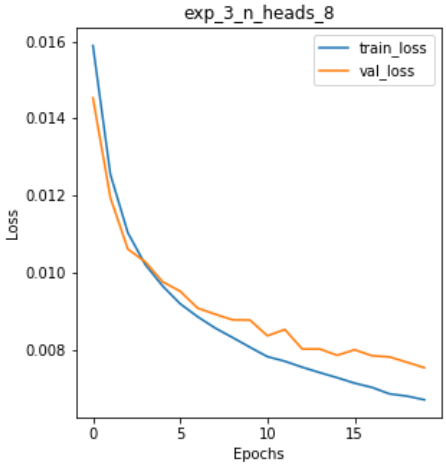
d) number of attention heads

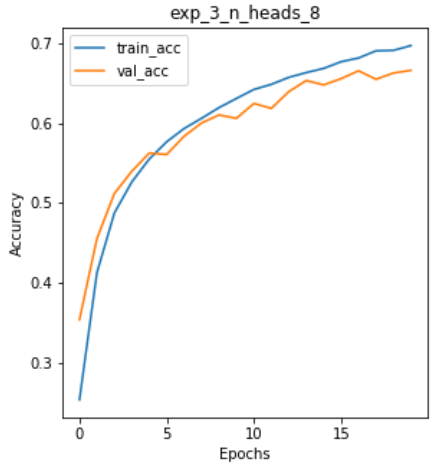








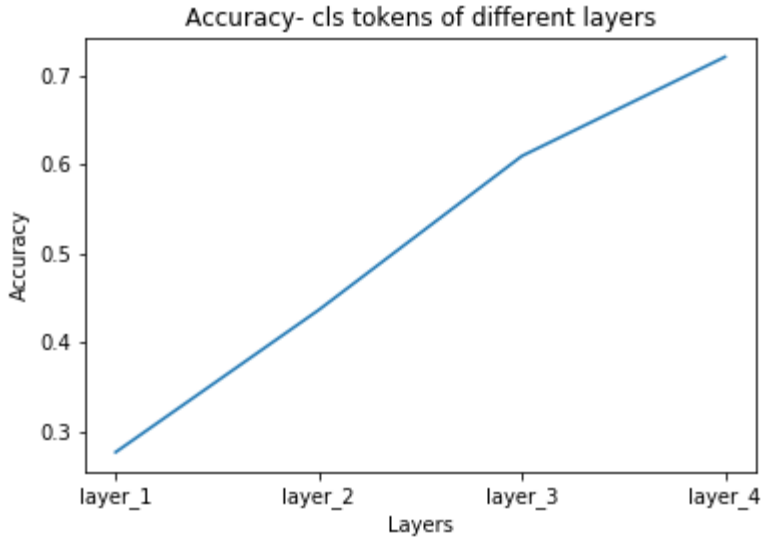


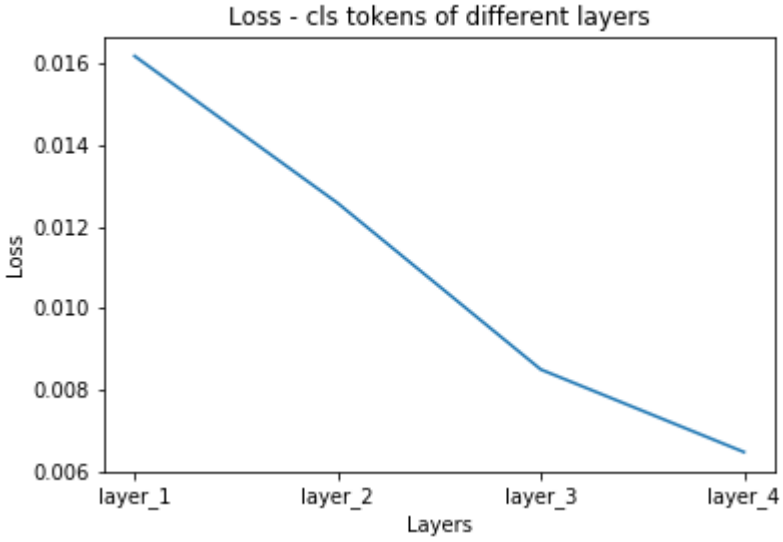


- Here, results got slightly better with an increasing number of attention heads.
- Increasing number of heads, increases model complexity. And it is expected to be able to observe more complex patterns.
- My opinion is that more attention heads can give better results with regularization. But they may overfit quickly without regularization.

e) classification by using the CLS token from different layers

- I used the same classifier head (at the top) for classification.
- These are the results.





    - As the token moves through the layers it will gather more and more information from other tokens corresponding to patches of images.
    - So, it is expected that, cls token from top layers will perform better.
    - But, I felt that using the Classification head specifically trained with top layer cls token is not the best choice to use for the cls tokens taken from middle layers.

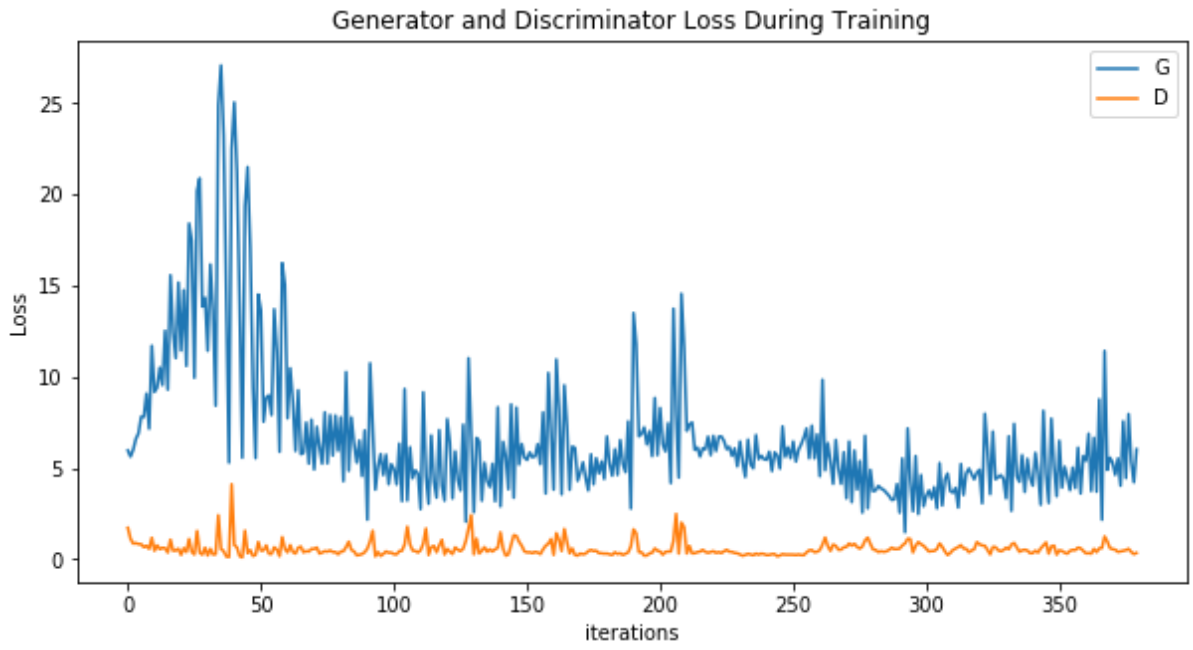
- Training a **separate classification network for cls token at each layer,** by freezing all of the model, can give better results. And it gives the **right indication** of how much information the cls token at each layer holds.
- But because of time constraints, I did not try it.

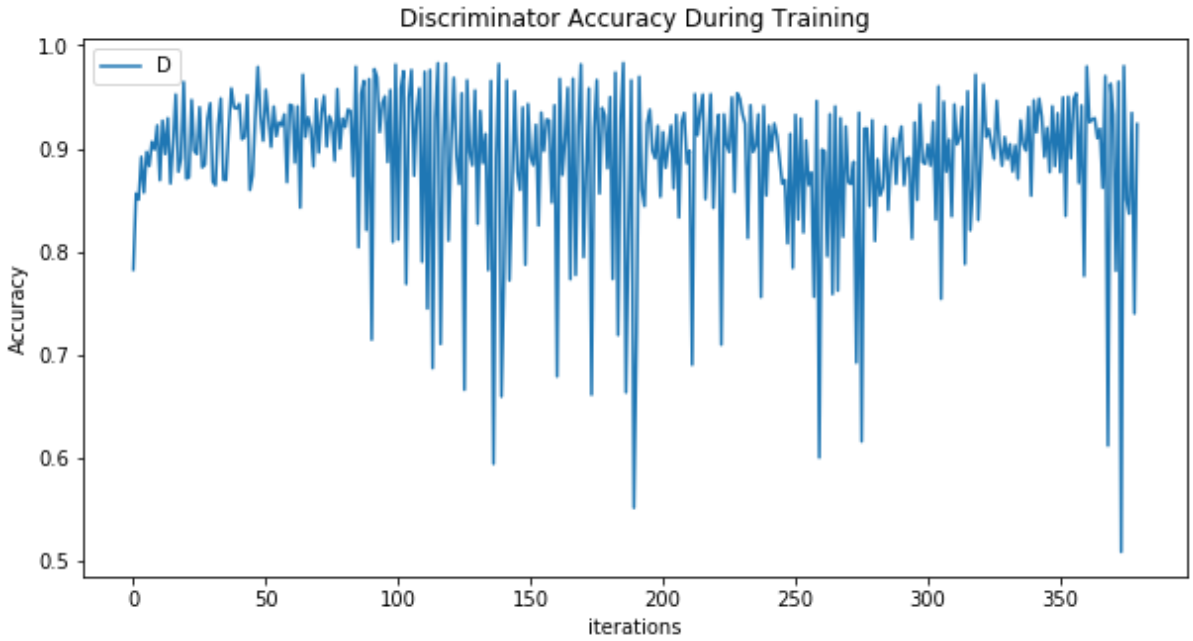
Q2 gan

A)Training

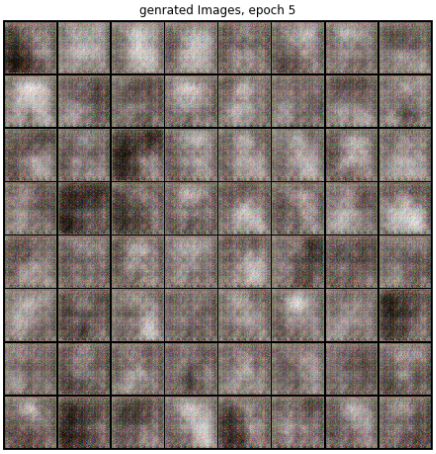
- Adam optimizer is used.
- Model weights are initialised with values from normal distribution with mean 0 , std = 0.02.
- Lr = 0.0002
- Batchsize is set to 256. On 2 GPUs with data parallelism.
- Latent vector size is 100.

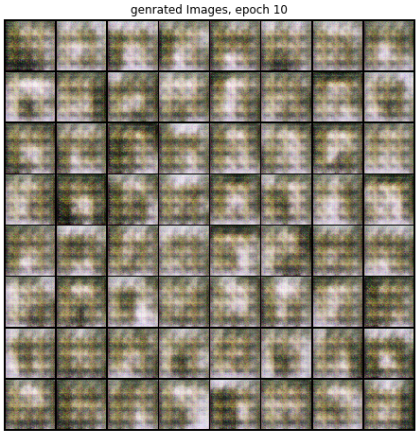
First I run it for 50 epochs. These are the plots.



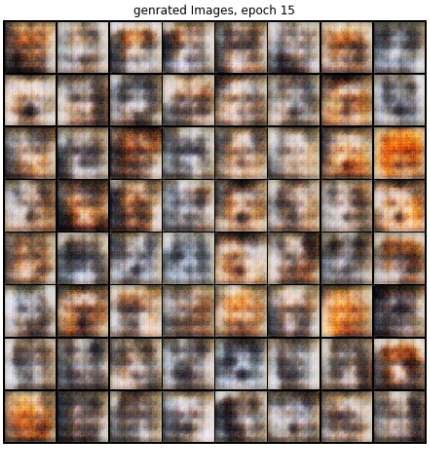


Some samples are taken after every 5 epochs to see the progress.

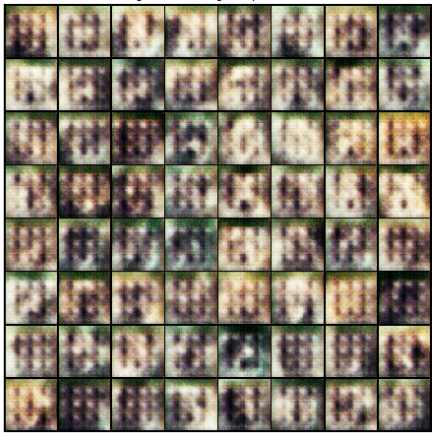




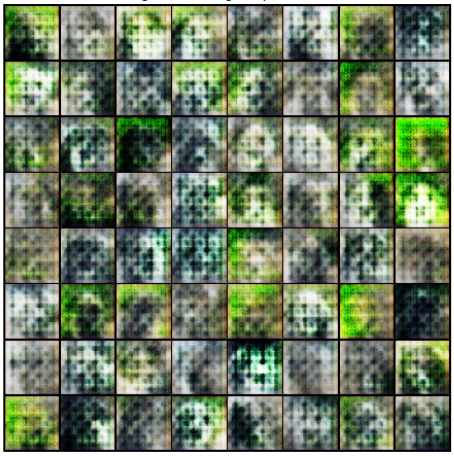
genrated Images, epoch 15



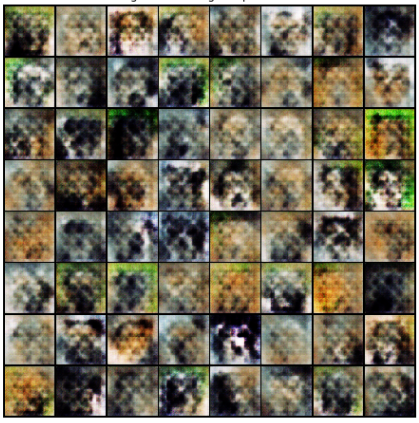
genrated Images, epoch 20



genrated Images, epoch 25

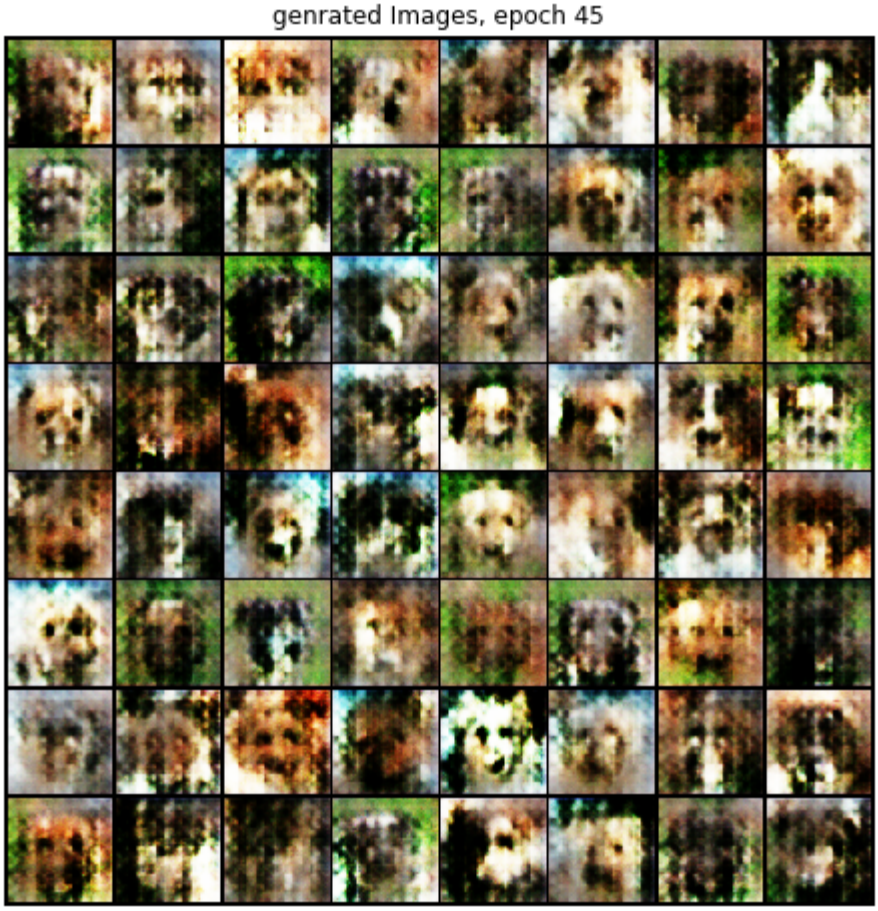


genrated Images, epoch 30

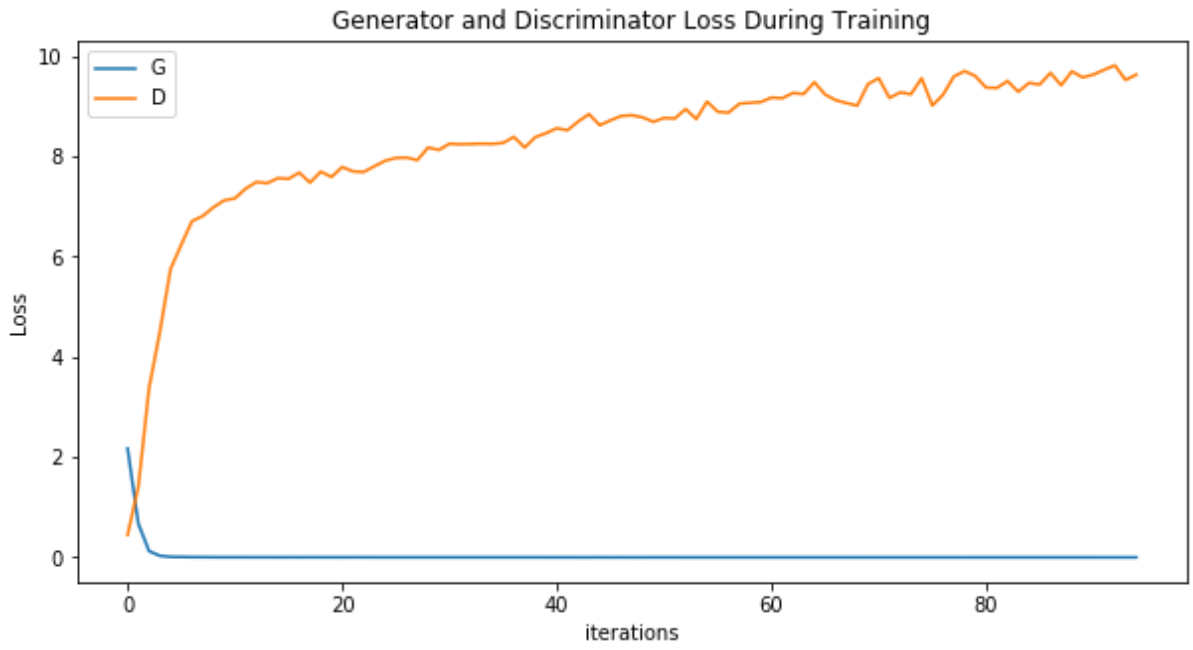


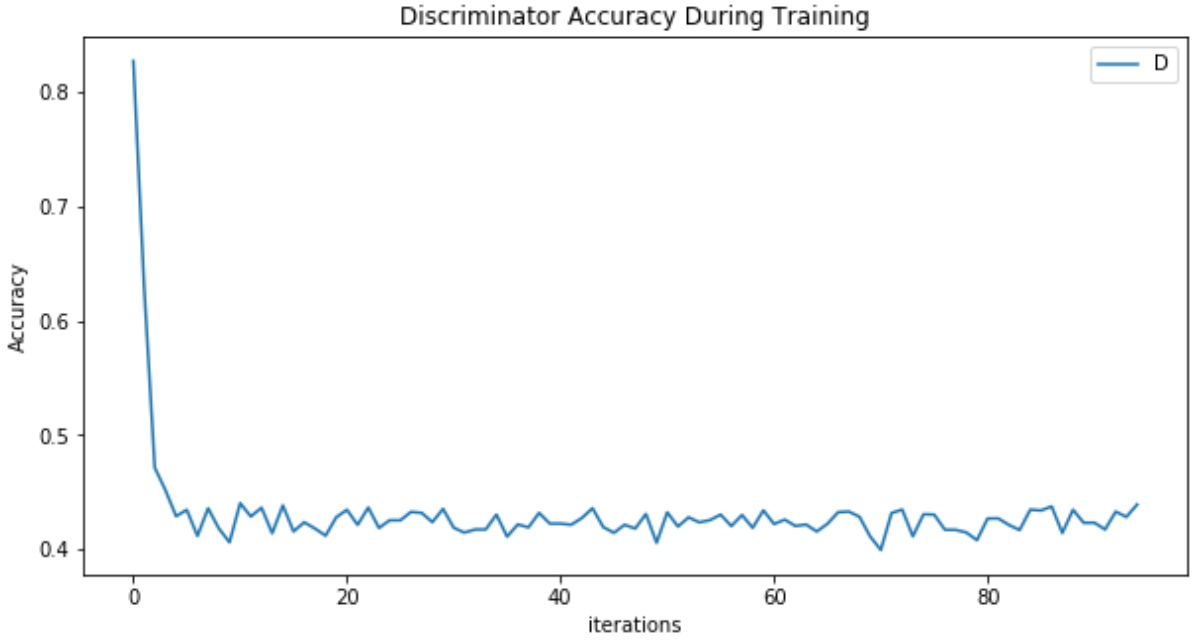
genrated Images, epoch 35

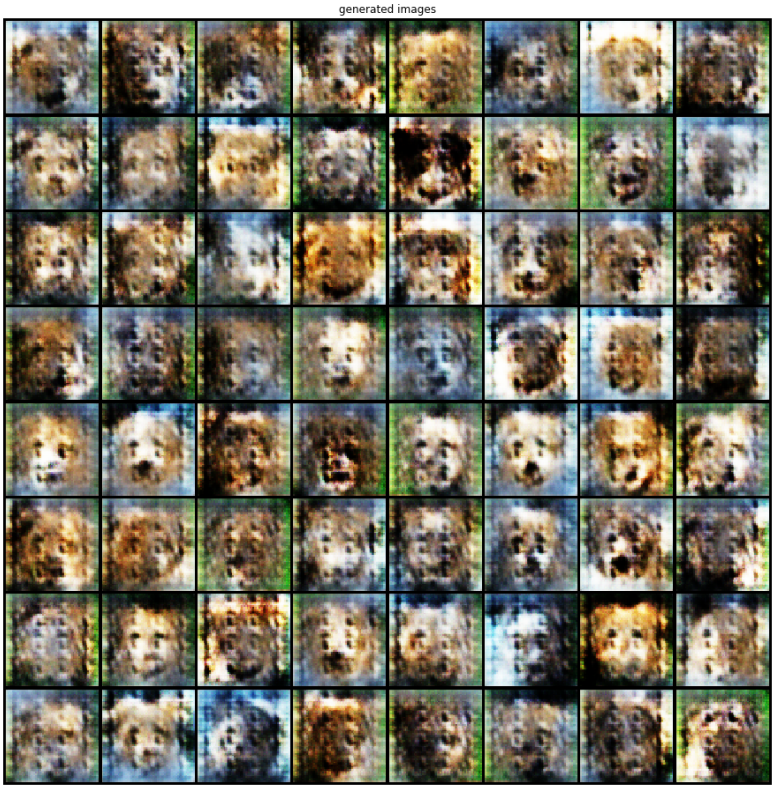
genrated Images, epoch 40



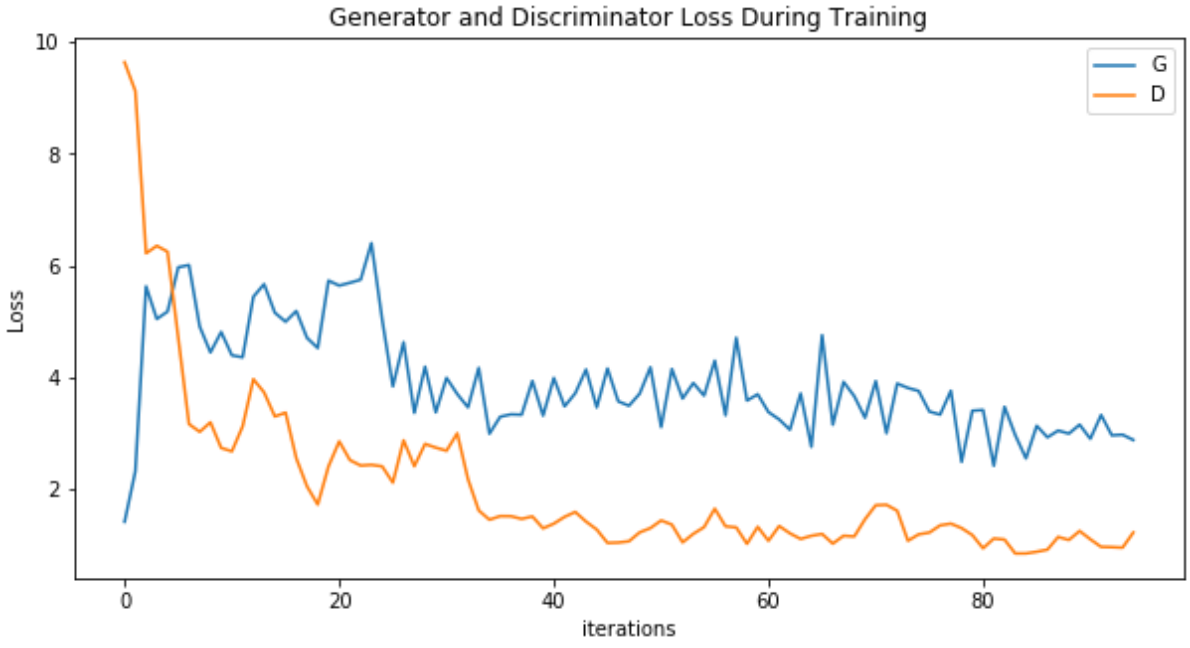
- I noticed that the discriminator is becoming very strong. So I kept it frozen for 5 epochs and only trained the generator.


Generator and Discriminator Loss During Training
G
D
Loss
iterations
0
2
4
6
8
10
0
20
40
60
80

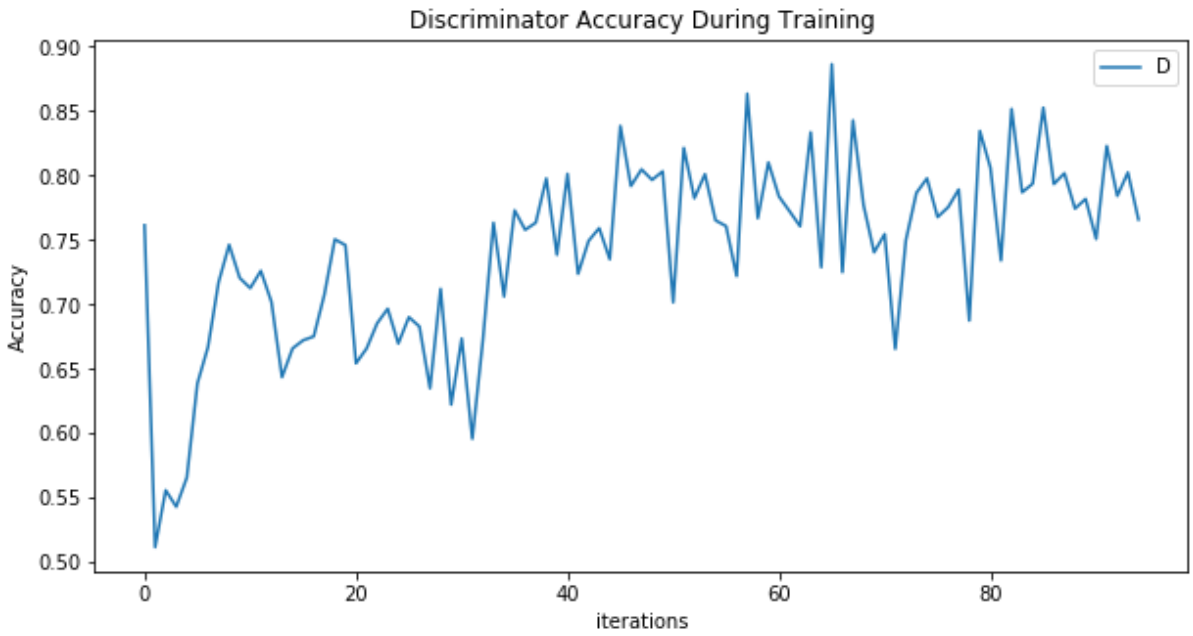

Discriminator Accuracy During Training
D
Accuracy
iterations
0.4
0.5
0.6
0.7
0.8
0
20
40
60
80



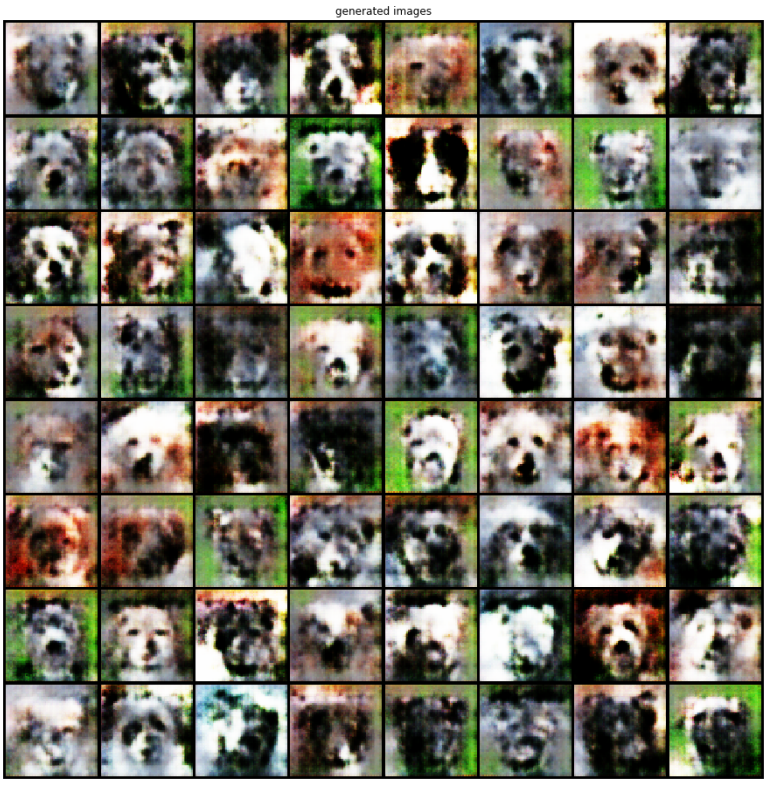
- Continued normal training of both descriminator and generator for 5 epochs



-


Discriminator Accuracy During Training
D
Accuracy
iterations
0.90
0.85
0.80
0.75
0.70
0.65
0.60
0.55
0.50
0
20
40
60
80


generated images



v101:

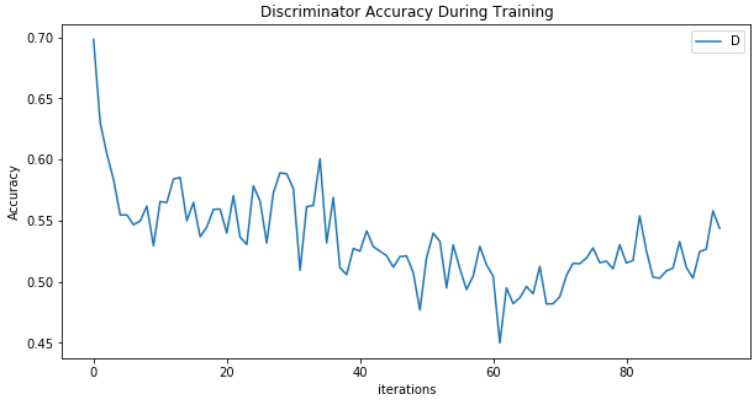
model_zoo/netG_101.pth

model_zoo/netD_101.pth

- I thought freezing the discriminator for every alternative epoch may give better results.
- I felt , i can get the same effect by lowering the learning rate of the discriminator. So, I lowered the discriminator learning rate.
-

Changed the lr of optimizerD to lr/6, now it gave better results.

- Now , discriminator accuracy is not going to high values; it is around the ideal value of 0.5.
- Results also got better.


Discriminator Accuracy During Training
D
Accuracy
iterations
0.70
0.65
0.60
0.55
0.50
0.45
0
20
40
60
80


Generator and Discriminator Loss During Training
G
D
Loss
iterations
2.2
2.0
1.8
1.6
1.4
1.2
1.0
0
20
40
60
80


generated images

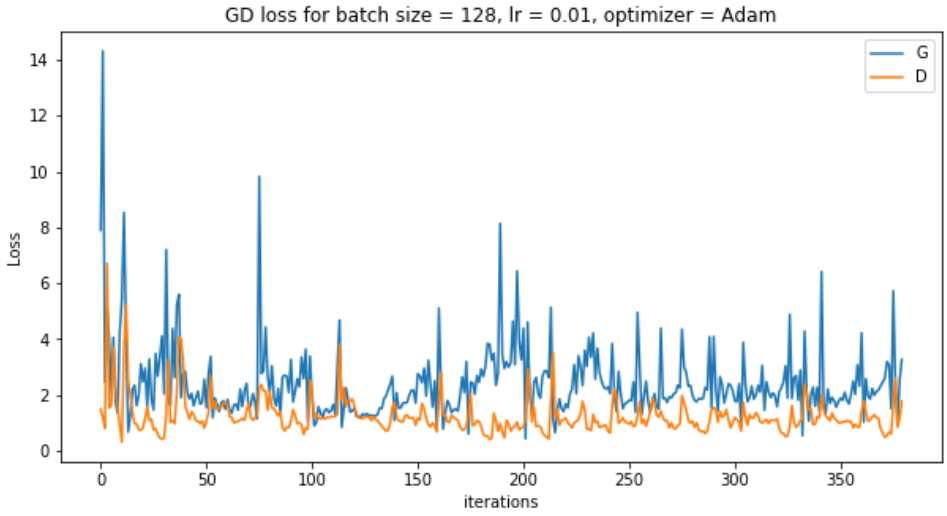
B) Hyperparameters

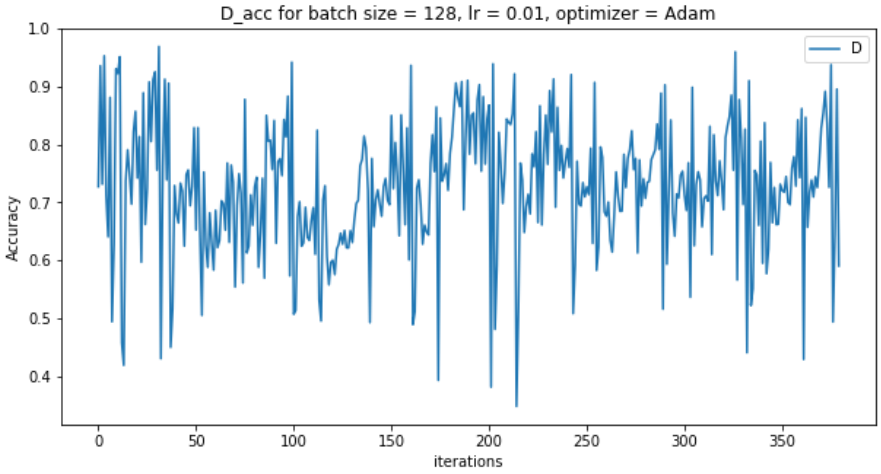
I tried with these values

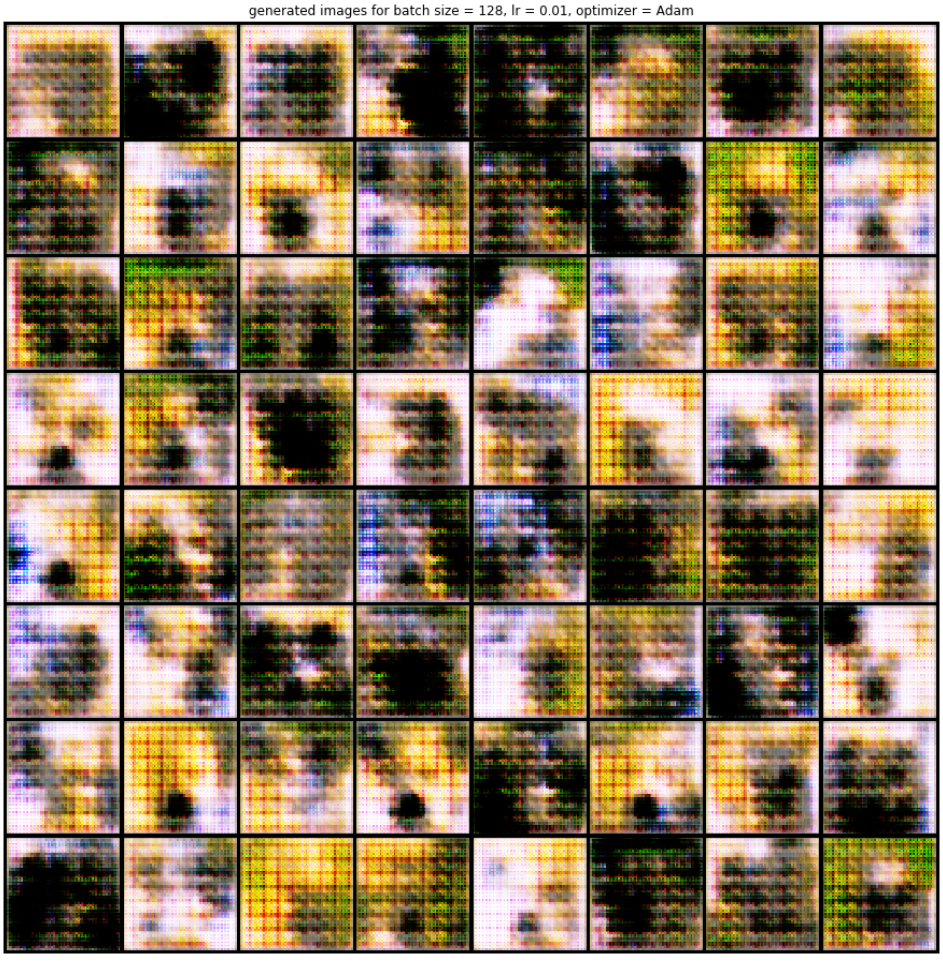
```
Batch_sizes = [128,256] # removed 64
Lr = [1e-2,1e-3,5e-4]
optimizer = [optim.Adam,optim.SGD,optim.RMSprop]
optim_names = ["Adam","SGD","RMSprop"]
```

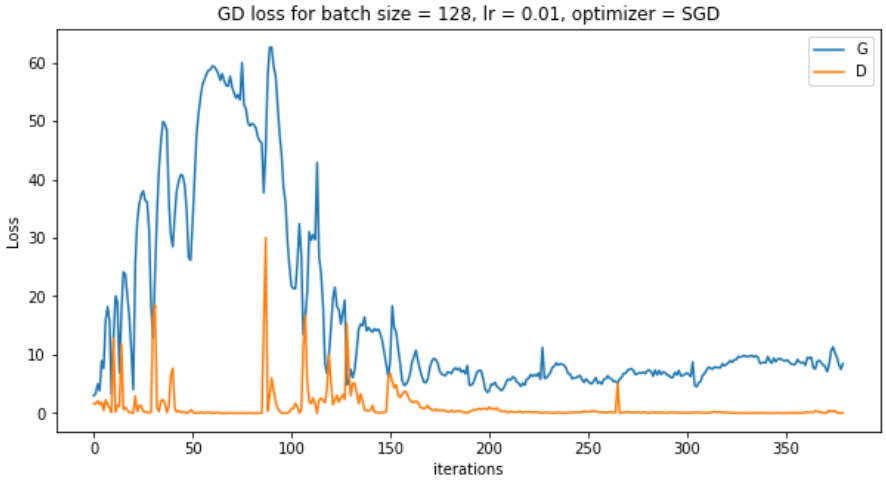
Each combination is run for 10 epochs, due to constraints.

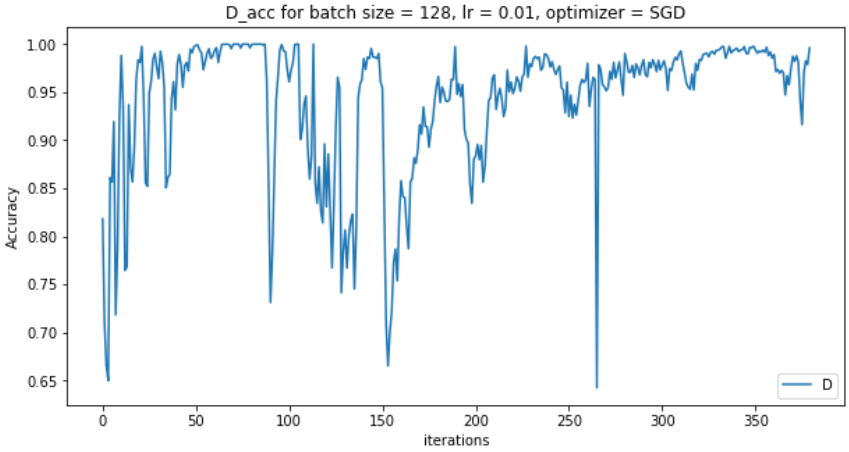
- Each combination is mentioned in the title

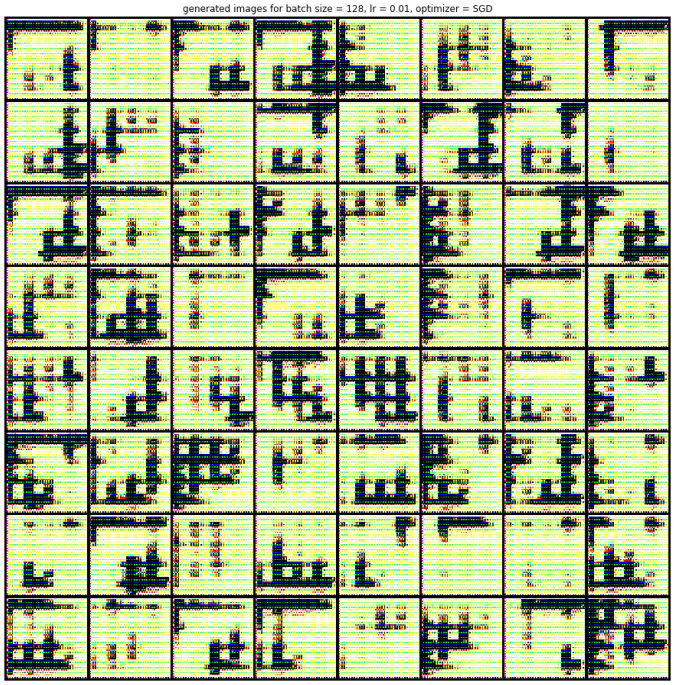


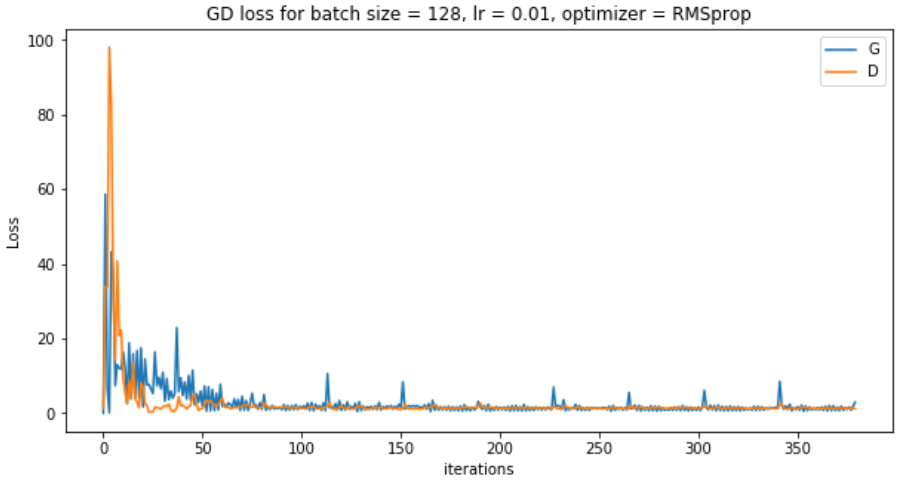

D_acc for batch size = 128, lr = 0.01, optimizer = Adam
D
Accuracy
iterations
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0
50
100
150
200
250
300
350

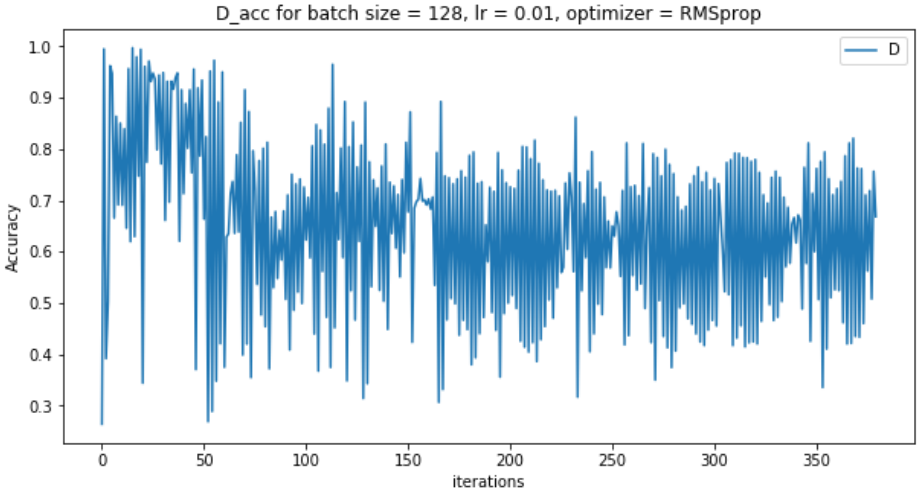

generated images for batch size = 128, lr = 0.01, optimizer = Adam


GD loss for batch size = 128, lr = 0.01, optimizer = SGD
G
D
Loss
iterations
0
10
20
30
40
50
60
0
50
100
150
200
250
300
350

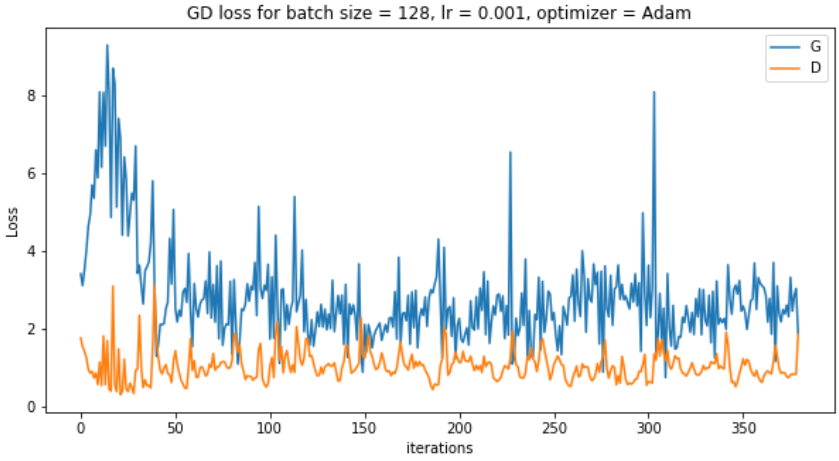

D_acc for batch size = 128, lr = 0.01, optimizer = SGD
D
Accuracy
iterations
0.65
0.70
0.75
0.80
0.85
0.90
0.95
1.00
0
50
100
150
200
250
300
350

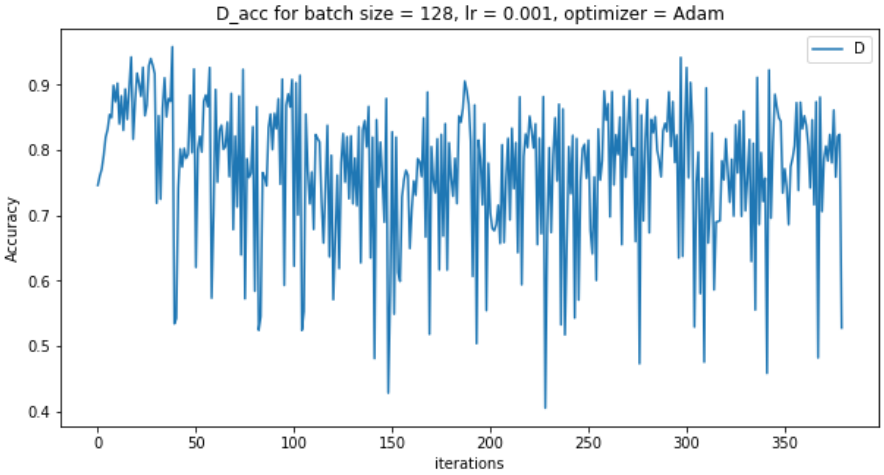

generated images for batch size = 128, lr = 0.01, optimizer = SGD

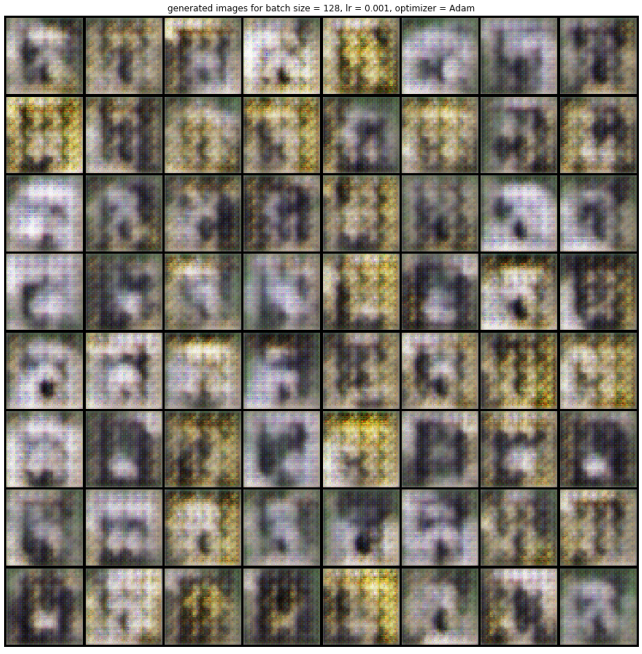

GD loss for batch size = 128, lr = 0.01, optimizer = RMSprop
G
D
Loss
iterations
0
20
40
60
80
100
0
50
100
150
200
250
300
350

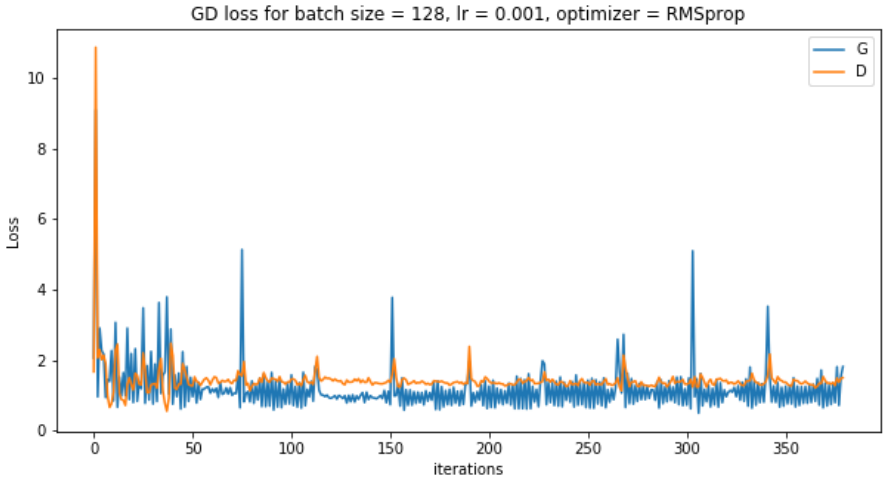

D_acc for batch size = 128, lr = 0.01, optimizer = RMSprop
D
Accuracy
iterations
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
0
50
100
150
200
250
300
350

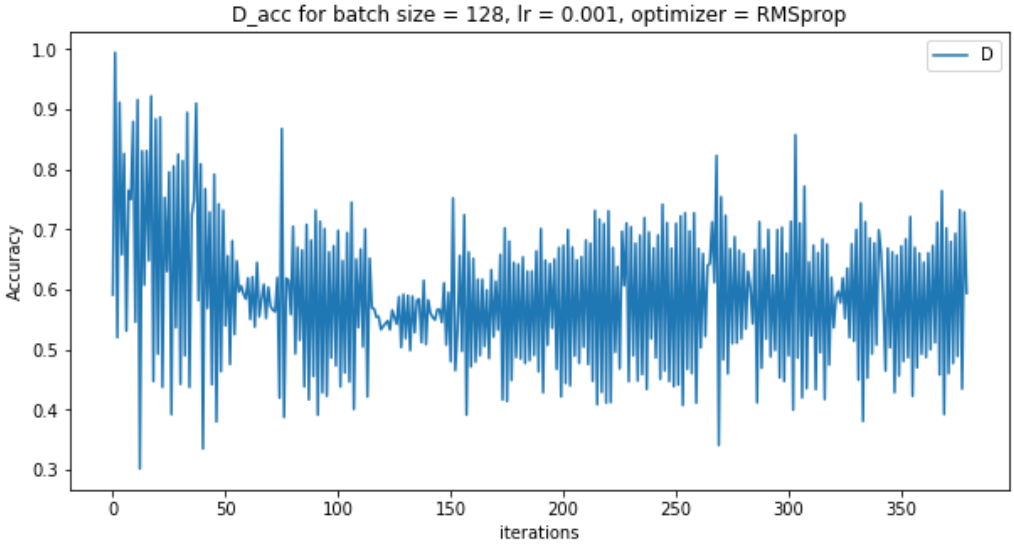

generated images for batch size = 128, lr = 0.01, optimizer = RMSprop

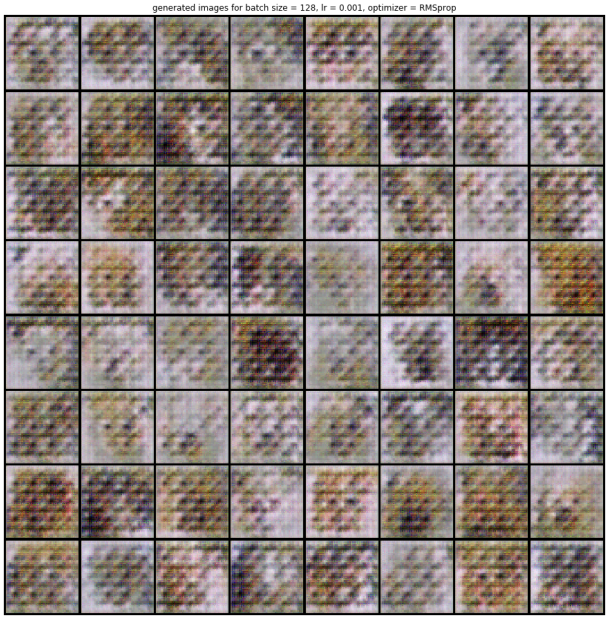

GD loss for batch size = 128, lr = 0.001, optimizer = Adam
G
D
Loss
iterations
0
2
4
6
8
0
50
100
150
200
250
300
350

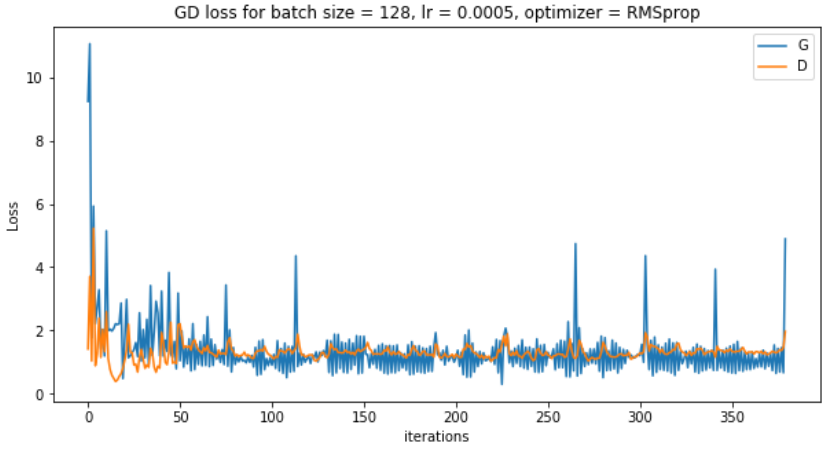

D_acc for batch size = 128, lr = 0.001, optimizer = Adam
D
Accuracy
iterations
0.4
0.5
0.6
0.7
0.8
0.9
0
50
100
150
200
250
300
350

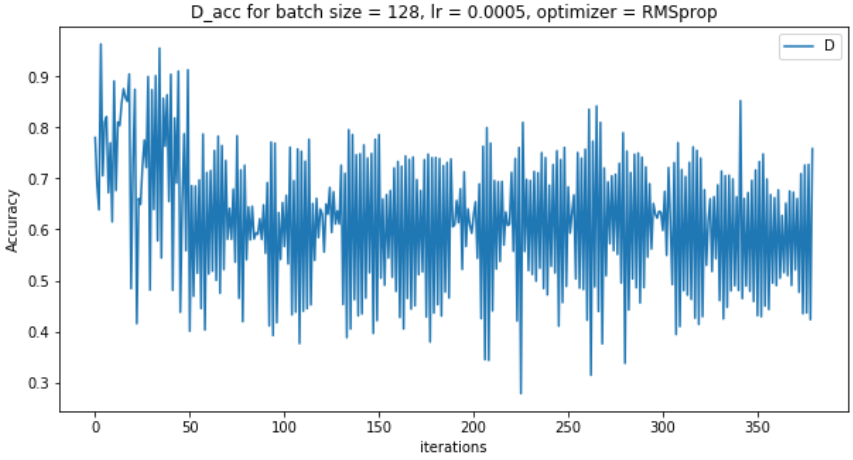

generated images for batch size = 128, lr = 0.001, optimizer = Adam

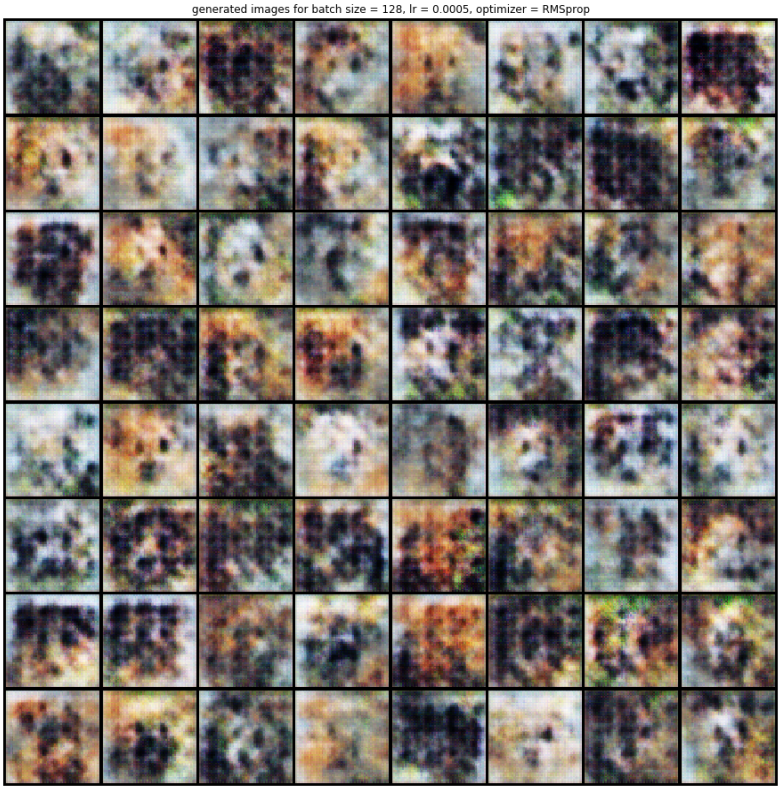

GD loss for batch size = 128, lr = 0.001, optimizer = RMSprop
G
D
Loss
iterations
0
2
4
6
8
10
0
50
100
150
200
250
300
350

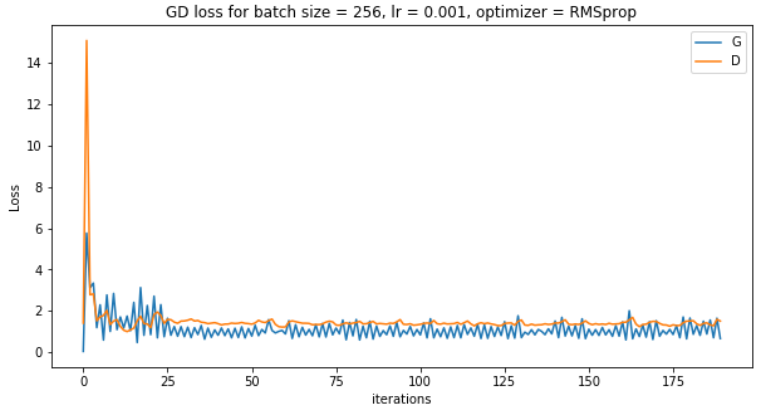

D_acc for batch size = 128, lr = 0.001, optimizer = RMSprop
D
Accuracy
iterations
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
0
50
100
150
200
250
300
350

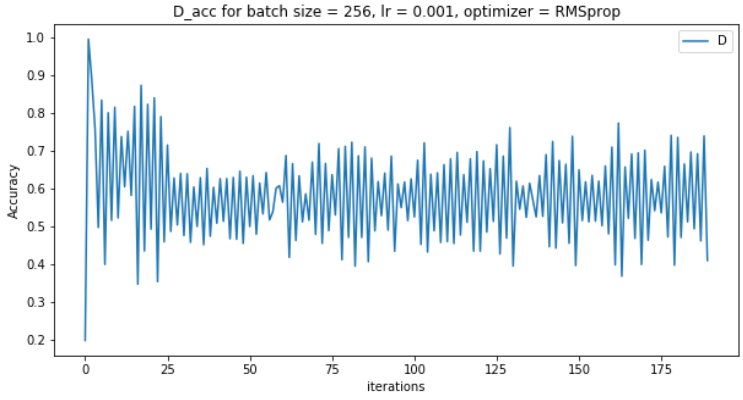

generated images for batch size = 128, lr = 0.001, optimizer = RMSprop

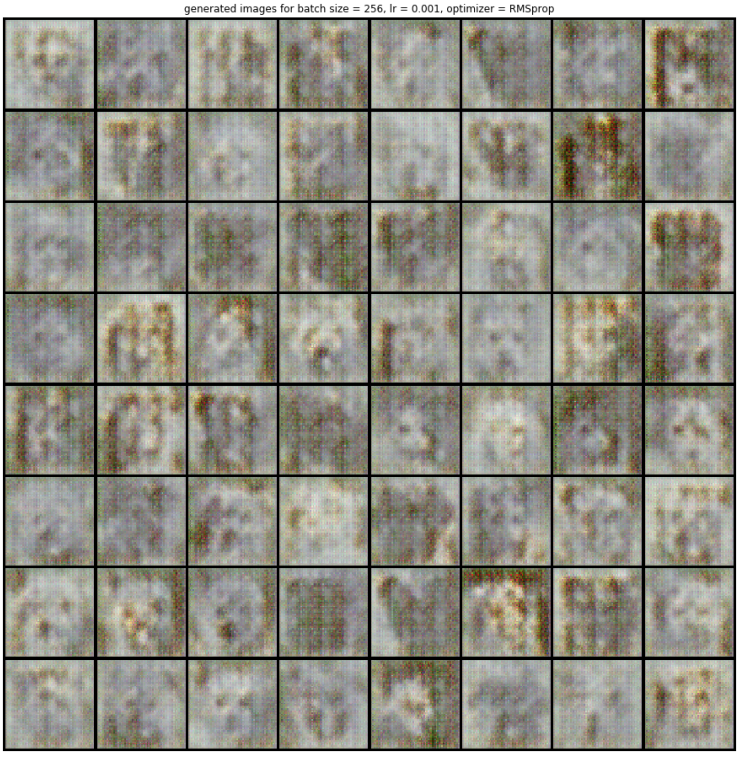

GD loss for batch size = 128, lr = 0.0005, optimizer = RMSprop
G
D
Loss
iterations
0
2
4
6
8
10
0
50
100
150
200
250
300
350

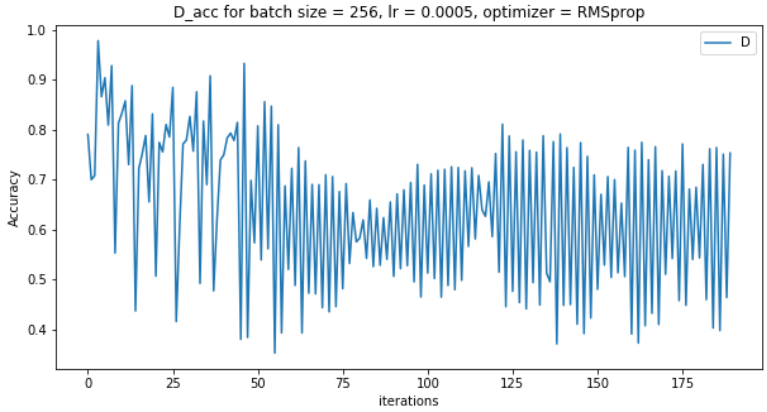

D_acc for batch size = 128, lr = 0.0005, optimizer = RMSprop
D
Accuracy
iterations
0.3
0.4
0.5
0.6
0.7
0.8
0.9
0
50
100
150
200
250
300
350

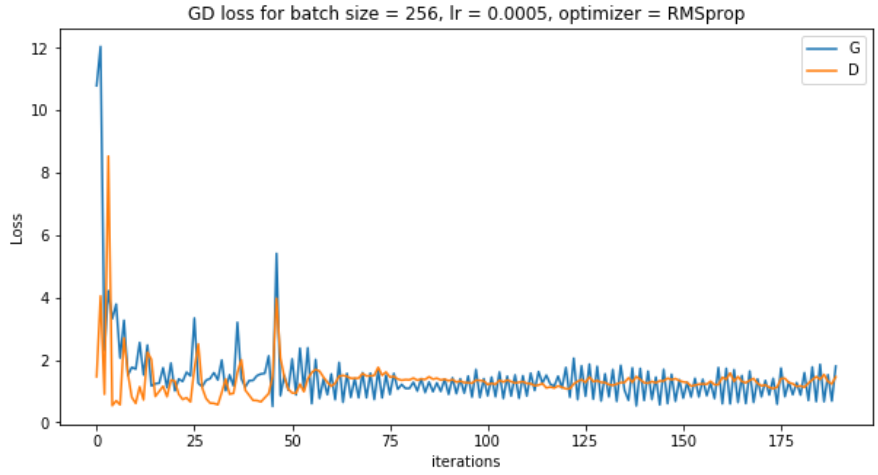

generated images for batch size = 128, lr = 0.0005, optimizer = RMSprop

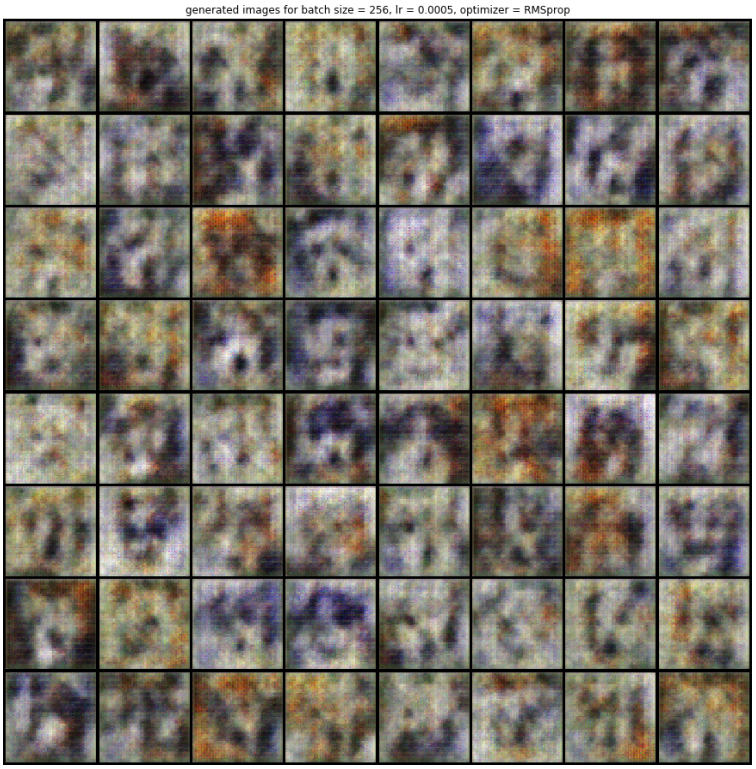

GD loss for batch size = 256, lr = 0.001, optimizer = RMSprop
G
D
Loss
iterations
0
2
4
6
8
10
12
14
0
25
50
75
100
125
150
175


D_acc for batch size = 256, lr = 0.001, optimizer = RMSprop
D
Accuracy
iterations
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
0
25
50
75
100
125
150
175


generated images for batch size = 256, lr = 0.001, optimizer = RMSprop


D_acc for batch size = 256, lr = 0.0005, optimizer = RMSprop
D
Accuracy
iterations


GD loss for batch size = 256, lr = 0.0005, optimizer = RMSprop
G
D
Loss
iterations


generated images for batch size = 256, lr = 0.0005, optimizer = RMSprop

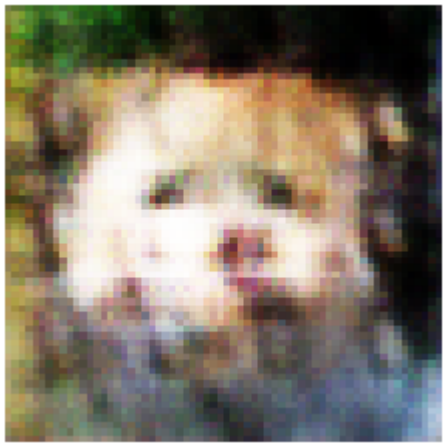
(here, some plots were not included, which did not gave good results.)

- All the plots are in “ plots/b/process” folder.

- RMSprop is clearly giving better results than SGD and Adam for most of the cases. And loss and accuracy plots are less volatile compared to others.
- Bigger batch size is giving better results (256 better than 64 and 128) . (plots for batch size 64 are also there in the plots folder)
- Here learning rate 5e-4 gave better results.

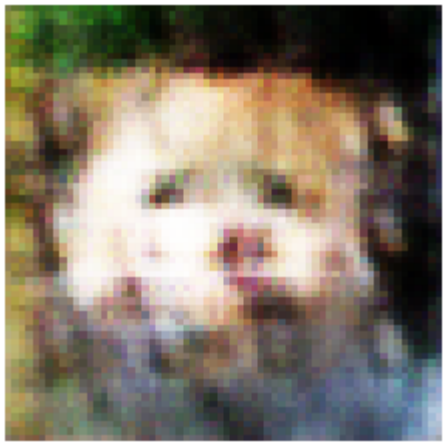
D) interpolation in Z - space

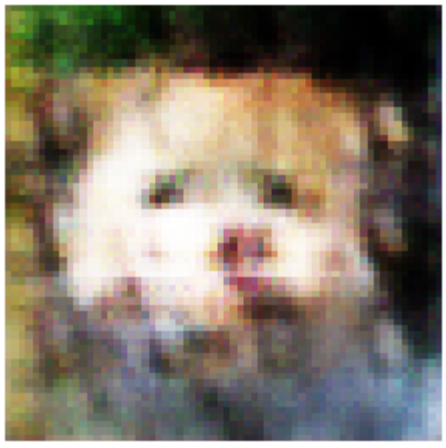
Img 1

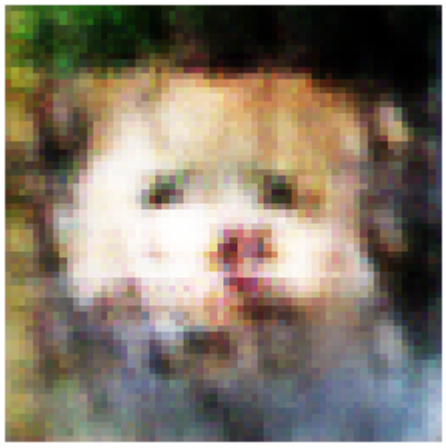
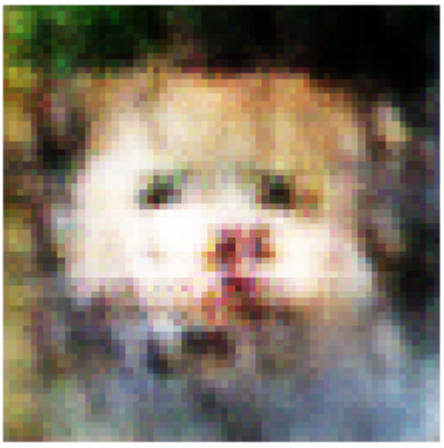
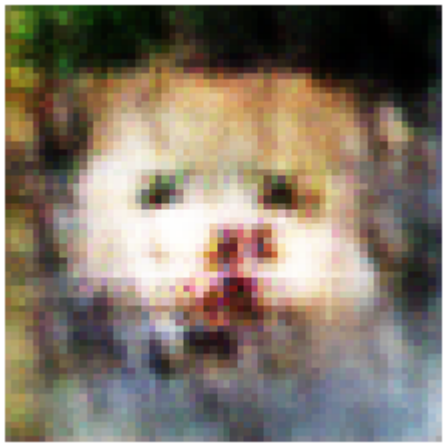
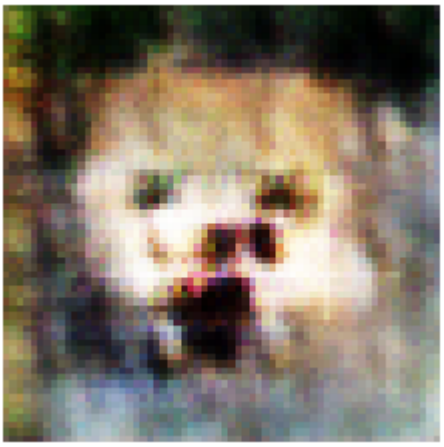


img 2

Interpolation : 10 points are taken as a convex combination.





C) implemented in AdaIN_Generator class.

e) to improve diversity of the image adding extra random vectors in each stage of the generator might help.