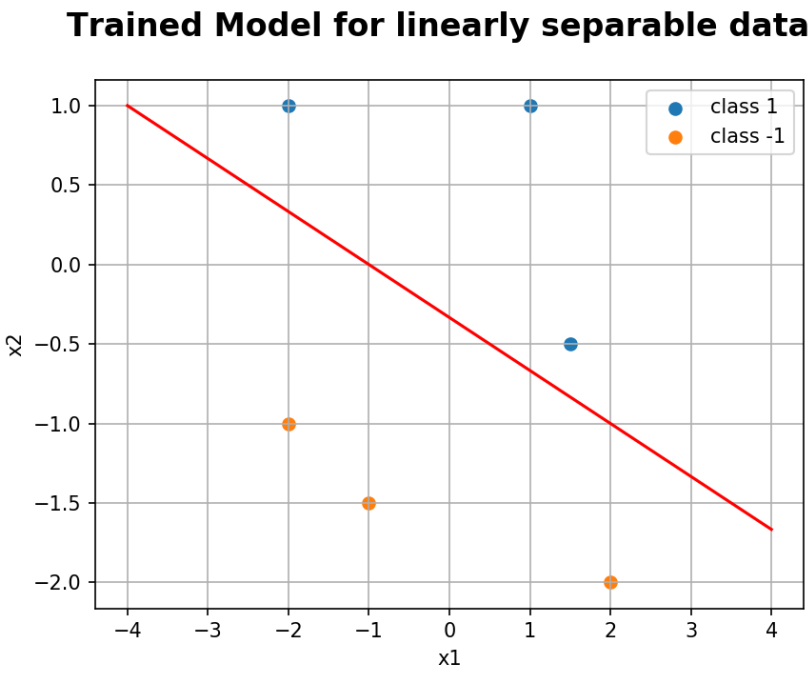
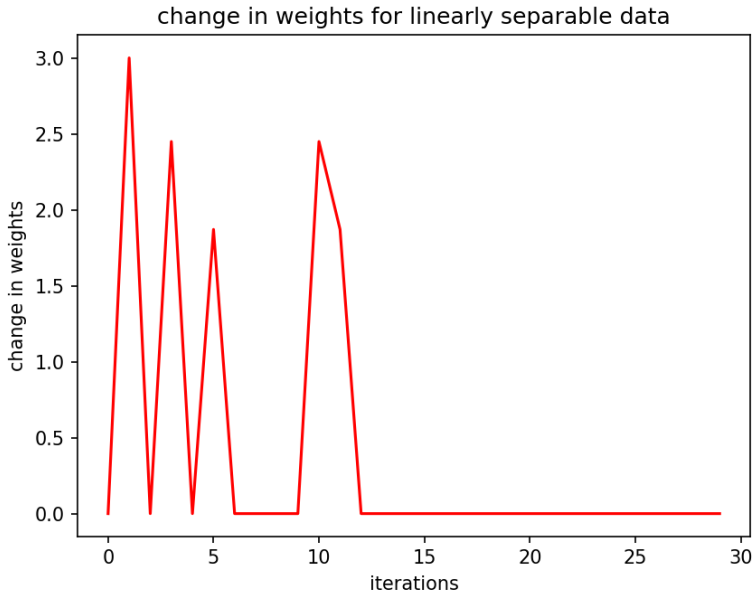
Problem 1.2:

- The code can be found in the folder.
- Upon 0 initialization of all the weights and bias, the algorithm converged in 2 epochs
- The final weights were the same as in the written problems, and the final decision boundary is plotted in the figure as shown below.



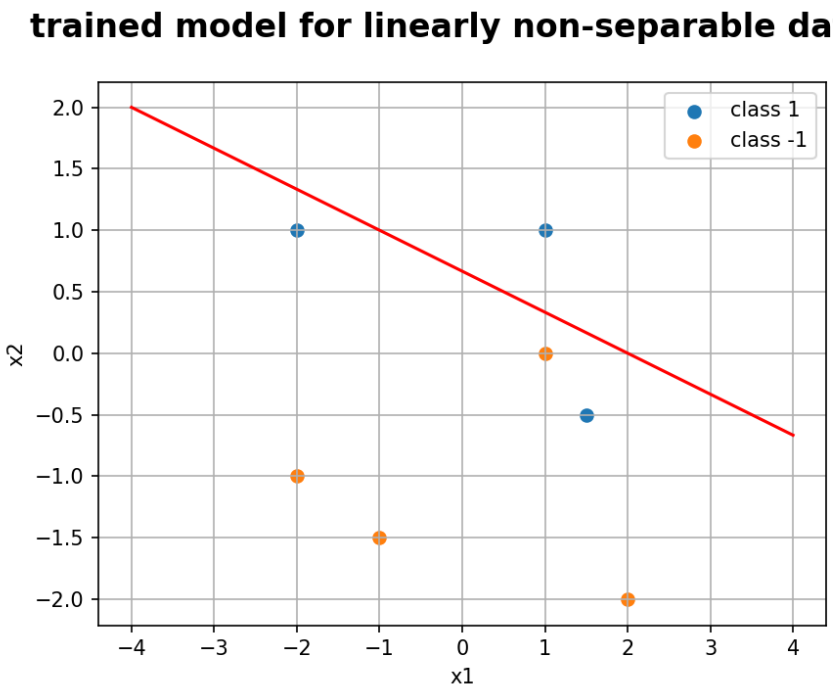
- The change in weights were plotted to show convergence of the perceptron algorithm for linearly separable data.



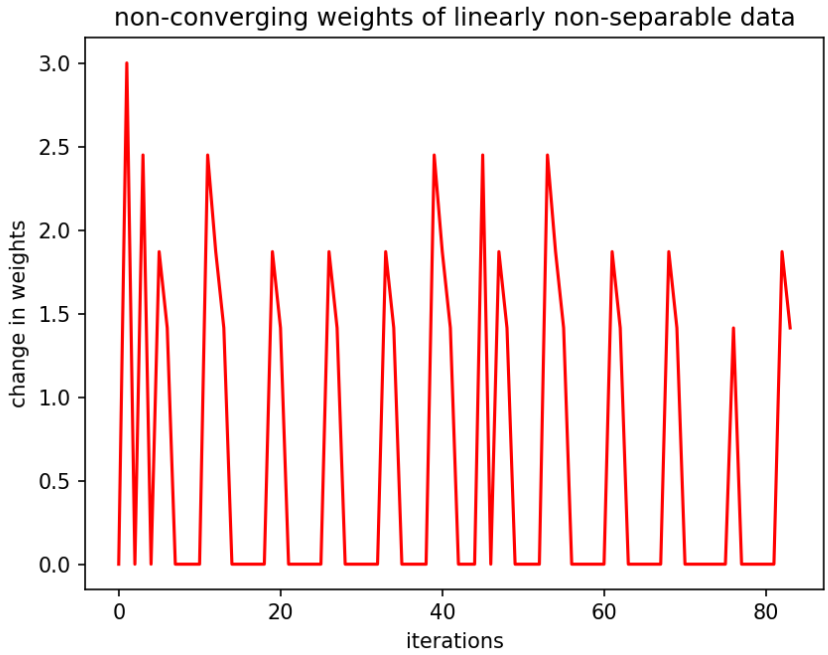
- Note: Each epoch has 6 iterations for the 6 corresponding input points. So the weights stop changing after 2 epochs.

Prob 1.3:

- To make the data not linearly separable, I added another data point (1,0) with label -1.
- The perceptron algorithm kept misclassifying one or more than 1 datapoint in each epoch showing that it can not work for a linearly non-separable data.
- I checked for 12 epochs, the final hyperplane is plotted below.

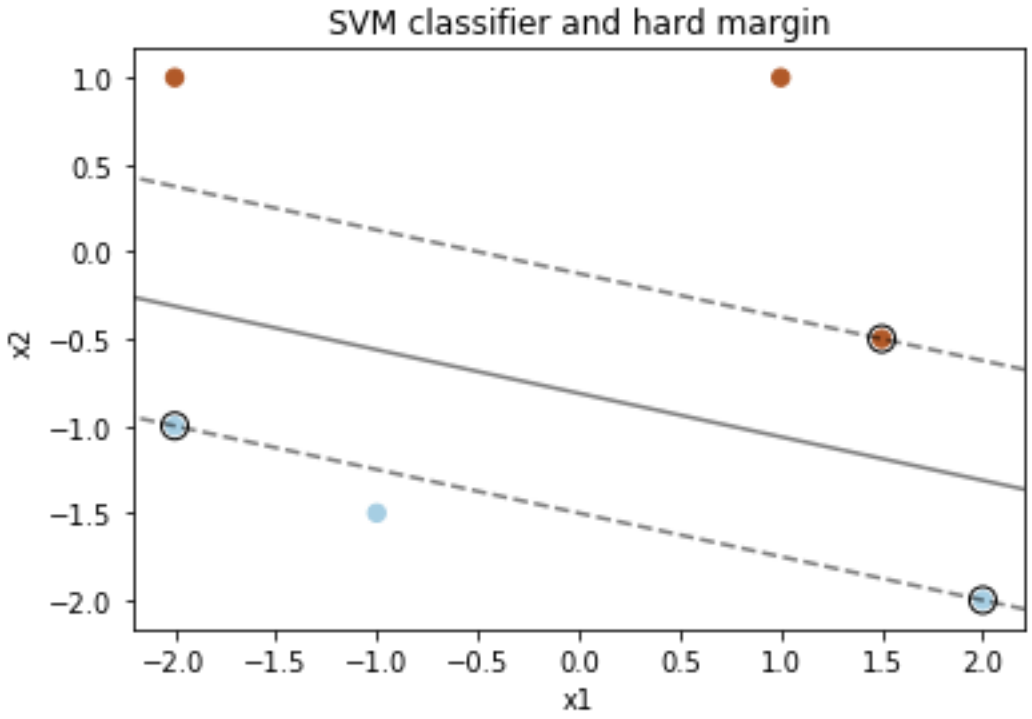


- To show non-convergence, the change in weights was plotted w.r.t the iterations.



Problem 2: SVM

The same data was classified using a linear SVM classifier. The code can be found in the codes folder and the final plot is visualized below.



Problem 3.1: Image Classification on CIFAR-10

3.1: LeNet on CIFAR10 dataset

- The dataset has images of 10 classes, and the network needs to learn to classify them.
- The LeNet architecture that I used has the following setup.
    - Input 3 channel RGB images -> 2D convolution layer with 5x5 kernel -> Relu -> output 6 channel -> maxPool2d -> 2D convolution layer with 5x5 kernel -> Relu -> output 16 channel
    - Input 16x5x5 nodes -> fully connected Network to output 120 for each node -> ReLu
    - Input 120 nodes -> fully connected Network to output 84 for each node -> ReLu
    - Input 84 nodes -> fully connected Network to output 10 for each node -> 10 output each giving the probability of each of the 10 classes.
- The loss function was cross-entropy loss
- Hyperparameters: Learning rate = 0.001, momentum = 0.9
- I first trained the model for 2 epochs but it gave bad results which was just a little higher than 54%.
- To increase the accuracy, I increased the epochs and retrained the model by re-initializing the weights so that I could compare its performance in the next problem.
- After 20 epochs, the LeNet model had a
    - loss: 0.694 on the train data.
    - accuracy: 60% on the test data.

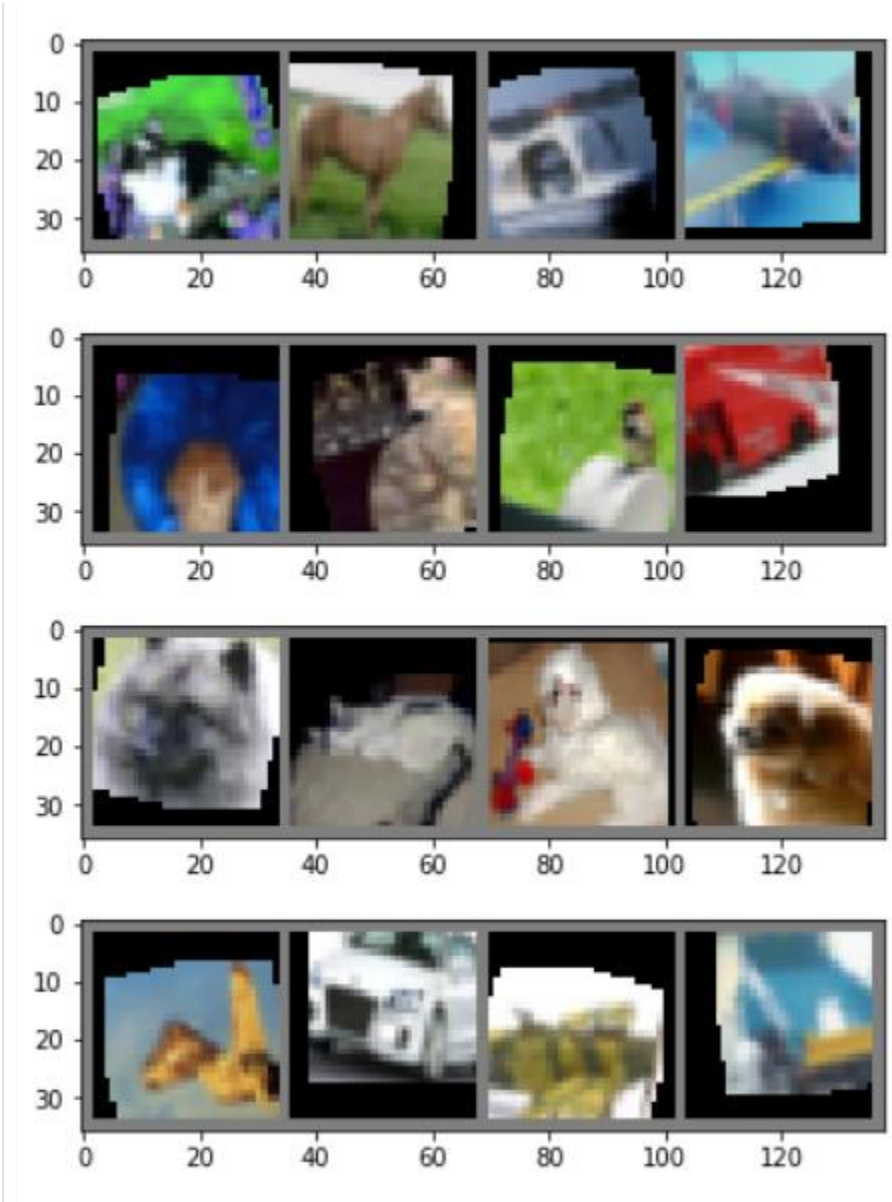
3.2: Data Augmentation

- To increase the model's accuracy on the same architecture, a technique of modulating the data in different ways like random rotation, random translation, random flipping and adding random noise was used.

Prob 3.2 Data Augmentation

```
[3]  train_transform = transforms.Compose(
         [
             transforms.ColorJitter(brightness=0.2, saturation=.05), # adding color jitter/noise

     #         transforms.RandomVerticalFlip(),
             transforms.RandomHorizontalFlip(),              #random flipping
             transforms.RandomRotation(20, resample=PIL.Image.BILINEAR),               #randomly rotate
             transforms.RandomAffine(0, (0.2,0.2)),        # randomly translate
             transforms.ToTensor(),
     #         transforms.Lambda(lambda x : x + torch.randn_like(x)),
             transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
```



- The one major insight I gained in this activity was that: Although the loss on the training data after 20 epochs was more than that of the model trained on the data without data augmentation, its accuracy on the test data was far better. This happens since the model doesn't get overfit to the same train data

over each epoch as it sees the same data in a different perspective every epoch and learns something new.

- After 20 epochs:
    - loss: 0.80 on the train data.
    - accuracy: 68% on the test data.

3.3: ResNet on CIFAR10 dataset

- The core take-away from the paper I got was how the authors solved the problem of vanishing or exploding gradients by forwarding the residuals to the layers that are a few steps away.
- The architecture I used was taken from the following 2 references:

[1] Kuangliu's github repo: https://github.com/kuangliu/pytorch-cifar/blob/master/models/resnet.py

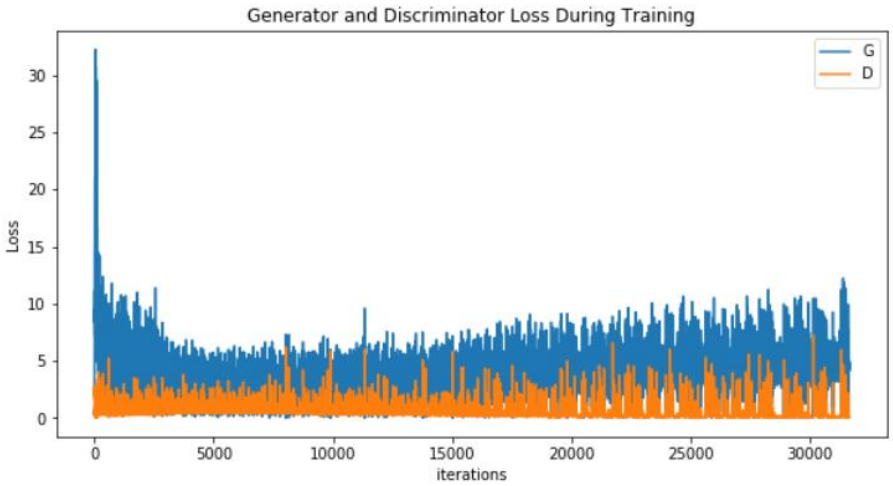
[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun
Deep Residual Learning for Image Recognition. arXiv:1512.03385

- I first tried 15 epochs with ResNet18 architecture on data without any data augmentation.
- I found that the network was quickly converging to a 0.001 loss in just 10 epochs and couldn't further decrease the loss. The model's accuracy scored 80% on the test data which was much better than the LeNet architecture.
- In order to solve the aforementioned issue, I introduced dataset augmentation and ran 5 epochs and found that the model's loss on the train set went up again to a 0.5 and ended at 0.4.
- However, this model did even better on the test data and gave an 84% accuracy on the test set.

Problem 4: GANs

4.1: DC GAN on Celebrity faces:

- I first tried with 5 epochs on the tutorial to which I got bad results.
- Next, I tried changing a few hyperparameters like the learning rate and feature depths, but realized the parameters were well set with respect to the losses as mentioned in the paper.
- The following are the results of 20 epochs on the entire dataset.



- The loss function used for this DC GAN was Binary Cross Entropy loss.

4.2 New Dataset of Colored Squares

- Following is a self-explanatory code of how I created the squares dataset.

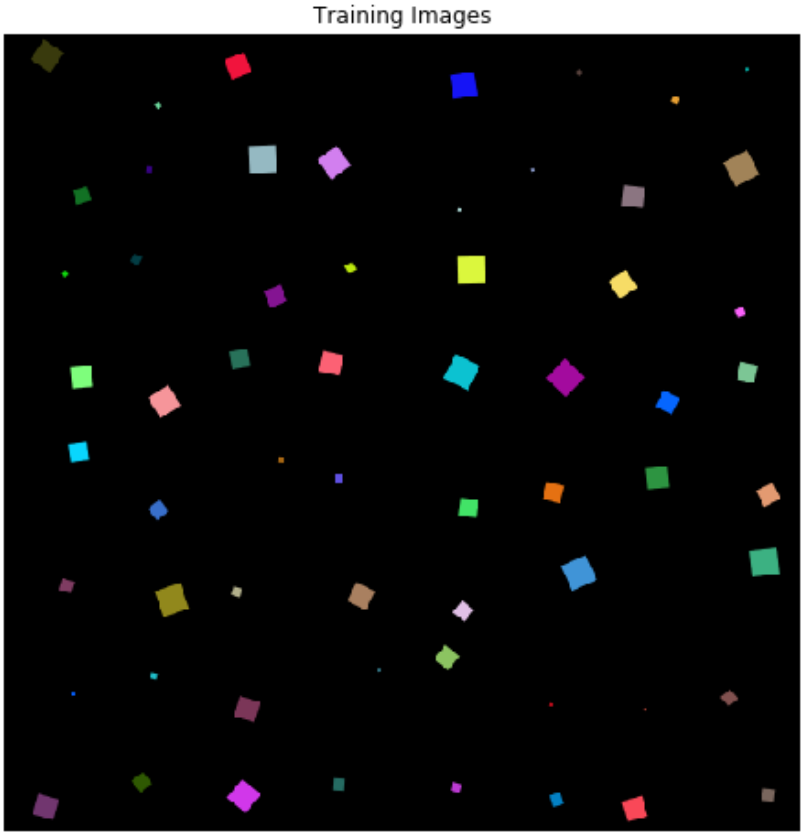
In [6]:

```
for i in range(5000):
    x = random.randrange(0,10)
    img = np.zeros((64,64,3), np.uint8)
    color = (random.randrange(1,255), random.randrange(1,255), random.randrange(1,255))

    img[32-x:32+x, 32-x:32+x, :] = tuple(reversed(color))
    rotationMat = cv2.getRotationMatrix2D((32,32), random.randrange(0,90), 1)
    img = cv2.warpAffine(img, rotationMat,(img.shape[0],img.shape[1]))
    trans_mat = np.float32([[1,0,random.randrange(-20,20)], [0,1,random.randrange(-20,20)]])
    img = cv2.warpAffine(img, trans_mat, (img.shape[0],img.shape[1]))
    # cv2.war
    file = drive + 'square'+ str(i) + '.png'
    plt.imsave(file, img)

```

- A sample batch of these squares visualized together is shown in the figure below



4.3 DC – GAN to generate squares

- I used 1000 generated images to train the DC GAN but it was too easy for the discriminator to differentiate between actual squares and fake squares and the model kept getting stuck near the D(x) > 0.9 which meant Discriminator could easily classify reals as reals with more than 90% confidence.
- I tried increasing the epochs to 200 and observed that the GAN first learned to create different colors, but when still the discriminator's loss did not increase, the Generator started learning the shapes.

Figure 2: Squares generated by the generator in the first half of training

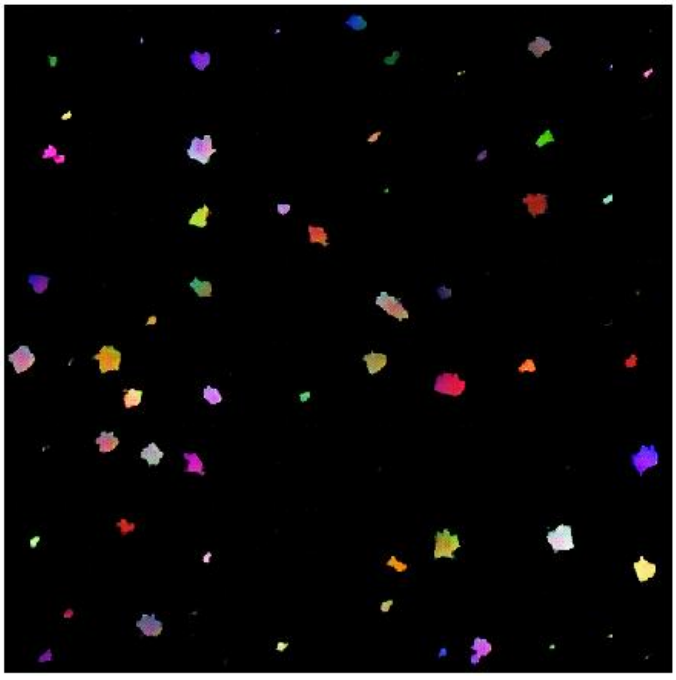
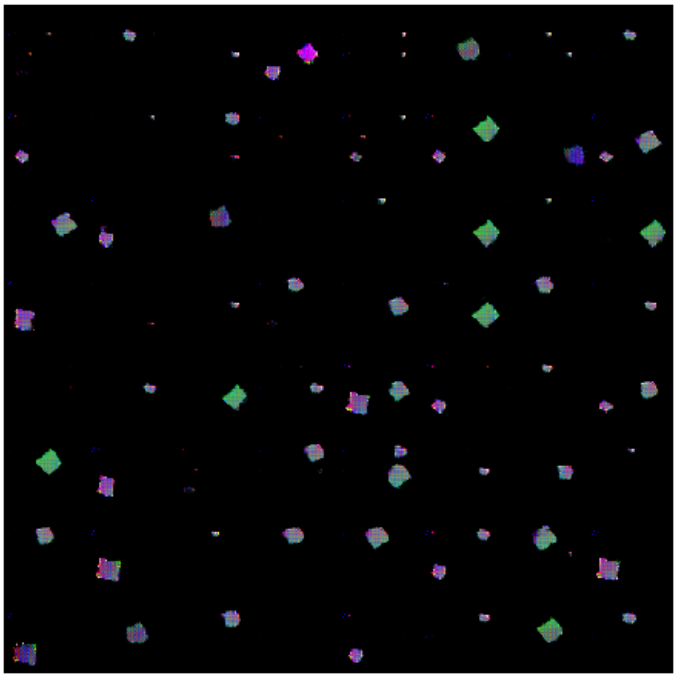
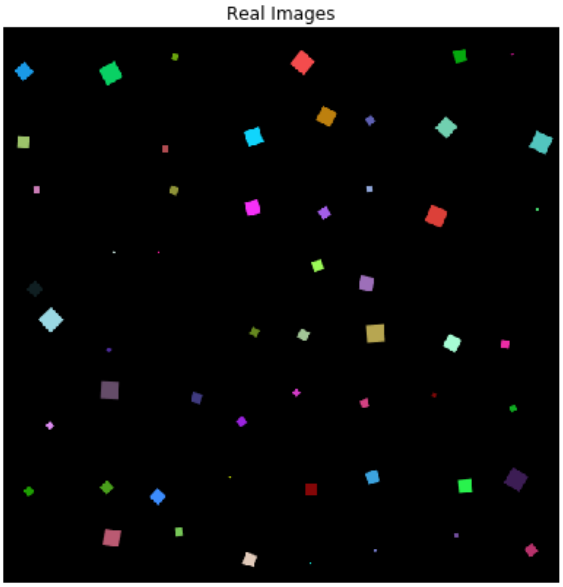
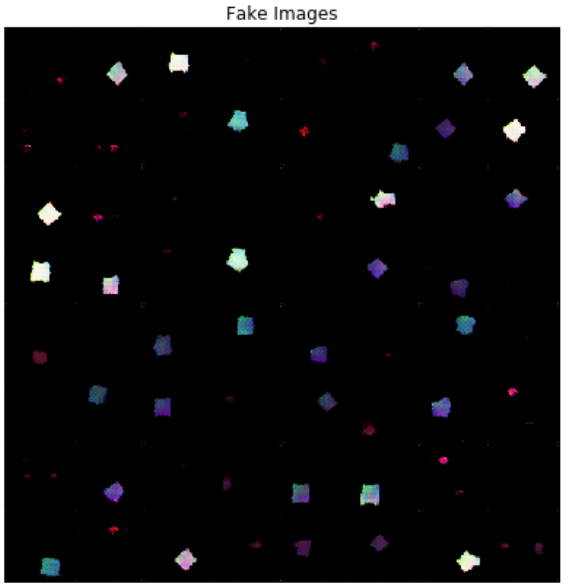


Figure 1: Squares generated by the generator in the second half of training

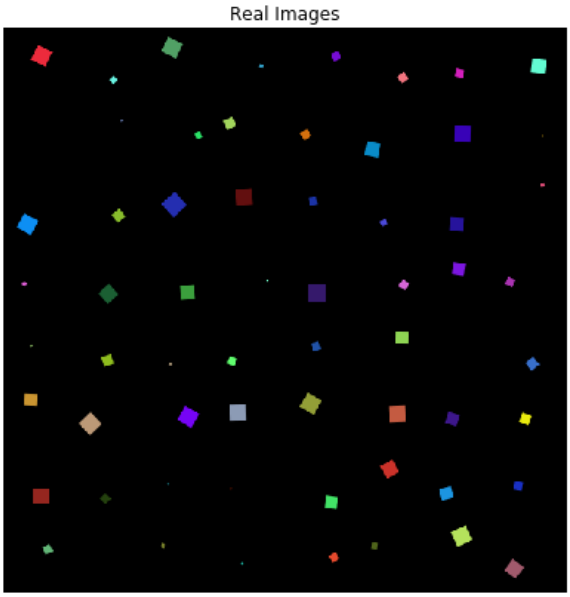


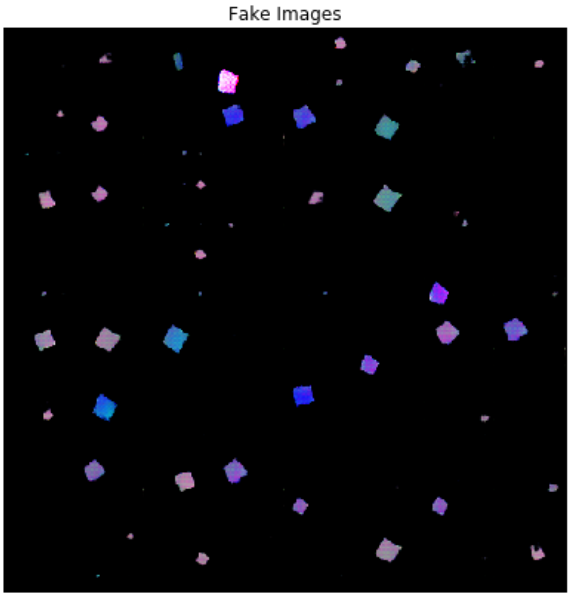
- Hence, I tried increasing the dataset size to 5000 and then to 10,000 images of square and the trained the model again for another 200 epochs.
- The model seemed to do a decent job. Following are the results.
- For GAN trained on 5000 images:





- For GAN trained on 10000 images:





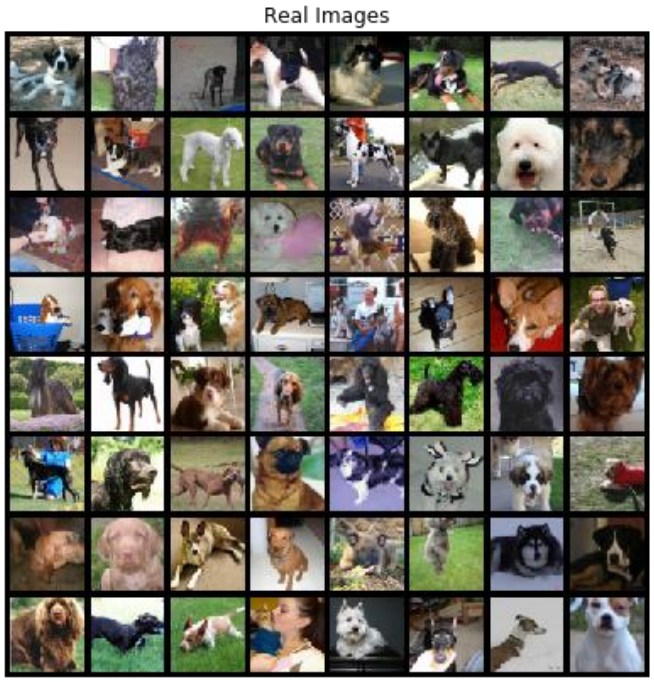
4.4 Collect a Dataset of Favorite Animal:

- For this task I collected Dogs images as I love them!
- There were many iterations to this process. Based on the results and the observations from the trained GAN, I decided to shift from a full dog picture in the image to just the dog's faces.
- The two main resources that I used were:

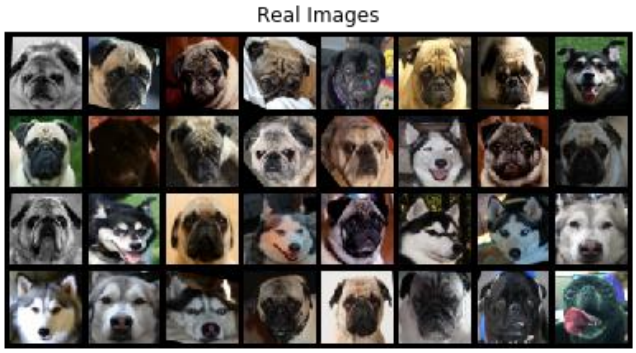
1. Flick-Dogs Dataset
2. ImageNet Dogs

4.5 Training DC-GAN to generate Dog images:

- I tried many techniques learnt by far like data augmentation, increasing the dataset decreasing the image complexity and got the following results.
- When I used the dataset with all the dog in the image, I could get the following result.



- Since, this clearly seemed very weird, I decided to use only faces of 2 breeds of dogs to make the Generator's task a little easier.
- I took only pugs and huskies and trained the DC-GAN. Increased the learning rate to 0.005 from 0.0001 to get better results.



- GAN trained on only 300 huskies: