Assignment 4

Introduction:

In this assignment, we generate images using a generative model known as Generative Adversarial Networks(GANs). We use 2 types of GANs for this assignment which are:

1) Deep Convolutional GANs(DCGAN)
2) Self Attention GANs(SAGAN)

CIFAR10 dataset is used for this assignment. I am using 45,000 images out of 50,000 train images for training the model for both cases. Preprocessing used for both types of GANs is just normalizing the images to have values between range [-1,1] in all channels. No data augmentation was used as the training set is big enough considering the dimension of images. I am using Frechet Inception distance as a metric to evaluate performance of the GANs.

1) Deep Convolutional GAN(DCGAN):

DCGAN uses deep convolutional layers for generator and discriminator networks. Deep convolutional network were used in original GAN paper also, but DCGAN was able to produce much better images than original GAN on many datasets (like LSUN bedrooms etc.). DCGAN was able to perform much better than original GAN because of improvements in network architectures and better hyperparameter tuning. Conceptually both GAN and DCGAN were similar.

DCGAN used batch normalization layers after each layer of discriminator and generator except the last layer of generator and the first layer of discriminator. This helps the network to learn fast in the beginning and also helps gradient flow. They also used leaky-Relu instead of Relu activation function in the discriminator for better propagation of gradients. DCGAN also used transposed convolution in the generator for upsampling from noise to image output. DCGAN removed fully connected and max-pooling layers from the network. Max-pooling layers were removed so that the network can learn better down-sampling for the data . They also set the hyperparameters of the network carefully which helped to stabilize the training and give better results.

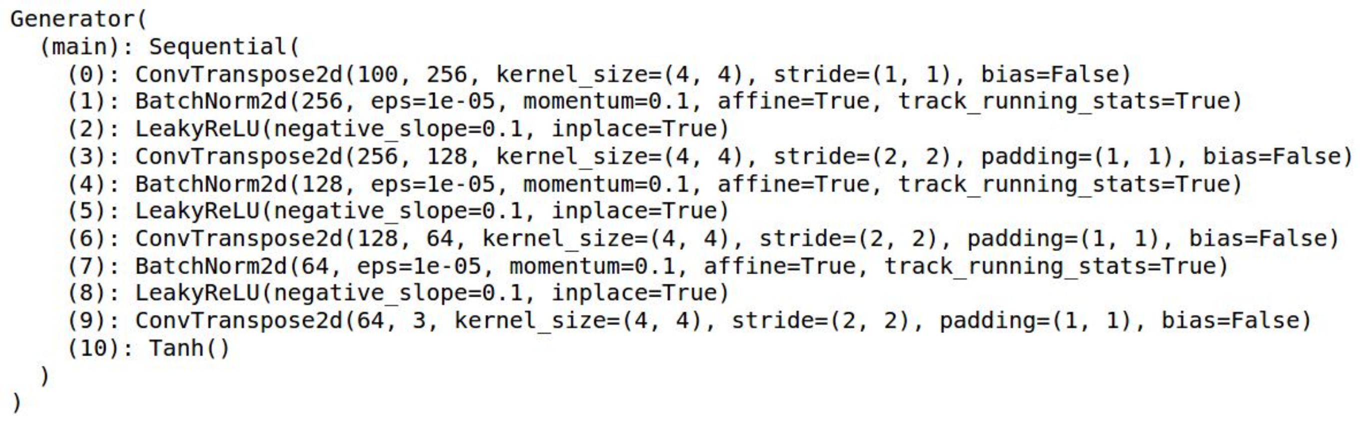
Hyperparameters Name	**Value or Method used**
Batch size	128

Optimizer	Adam
Beta1(for Adam optimizer)	0.5
Learning Rate	0.0002
Activation Function	leaky-Relu
Leaky-Relu negative slope(discriminator)	0.2
Leaky-Relu negative slope(Generator)	0.1
Weight Initialization	From Gaussian of 0 mean, std dev 0.02

Authors of the paper set beta1 to 0.5 to reduce the oscillations while optimization. Batch size of 128 also helps stabilize the training and decrease variance in gradient.

Generator artitechture:

Generator takes 100 dimensional noise and converts it into an image.



```
Generator(
  (main): Sequential(
    (0): ConvTranspose2d(100, 256, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): LeakyReLU(negative_slope=0.1, inplace=True)
    (3): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): LeakyReLU(negative_slope=0.1, inplace=True)
    (6): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (7): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): LeakyReLU(negative_slope=0.1, inplace=True)
    (9): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (10): Tanh()
  )
)
```

Discriminator Architecture:

Discriminator takes input and classifies it as fake or real.

```
Discriminator(
  (main): Sequential(
    (0): Conv2d(3, 32, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (3): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (4): LeakyReLU(negative_slope=0.2, inplace=True)
    (5): Conv2d(64, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (6): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (7): LeakyReLU(negative_slope=0.2, inplace=True)
    (8): Conv2d(64, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (9): Sigmoid()
  )
)
```

DCGAN Objective:

Original GAN objective is

$$\min_{\theta_g}\max_{\theta_d}\left[\mathbb{E}_{x\sim p_{data}}\log \underbrace{D_{\theta_d}(x)}_{\substack{\text{Discriminator output}\\ \text{for real data x}}} + \mathbb{E}_{z\sim p(z)}\log(1-\underbrace{D_{\theta_d}(G_{\theta_g}(z))}_{\substack{\text{Discriminator output for}\\ \text{generated fake data G(z)}}})\right]$$

But, in the beginning the generator will produce very bad images which will be easily classified as fake by discriminator. And if very low scores to fake images are given by discriminator then gradient propagating back to the generator will be almost zero. So to overcome the problem of vanishing gradient, this objective is modified for generator but remains the same for discriminator. Modified cost function used in GAN paper is:

$$\max_D V(D) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1-D(G(\boldsymbol{z})))]$$

$$\min_G V(G) = \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1-D(G(\boldsymbol{z})))] \;\textbf{or}\; \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})} - [\log D(G(\boldsymbol{z}))]$$

(original cost function) (alternative cost function)

Results :

For calculating FID score 2000 validation images and 1000 fake images was used. Ideally, more images should be used to calculate FID but it makes the training loop very slow. If more images are used to calculate FID score then FID score obtained would be lower(which is good). Last average pooling features of 2048 dimensions from Inceptionv3 network was used to calculate FID score. Output images for DCGAN can be seen in DCGAN_test.ipynb.

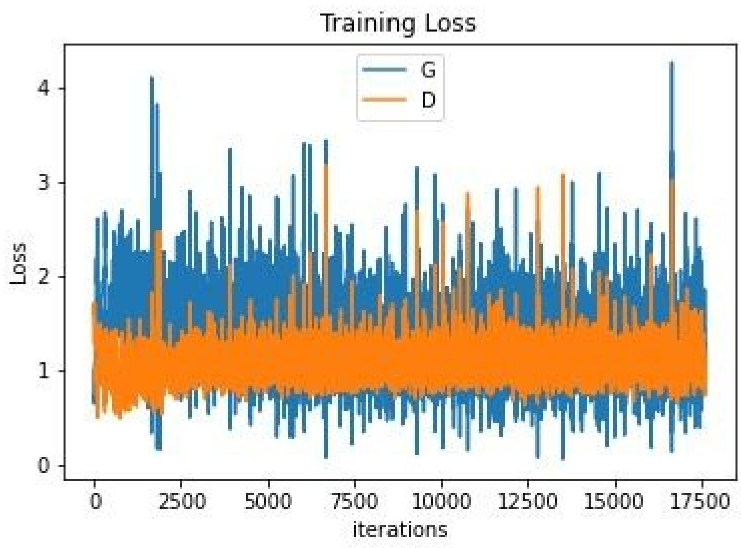


Fig) Loss of Generator and discriminator

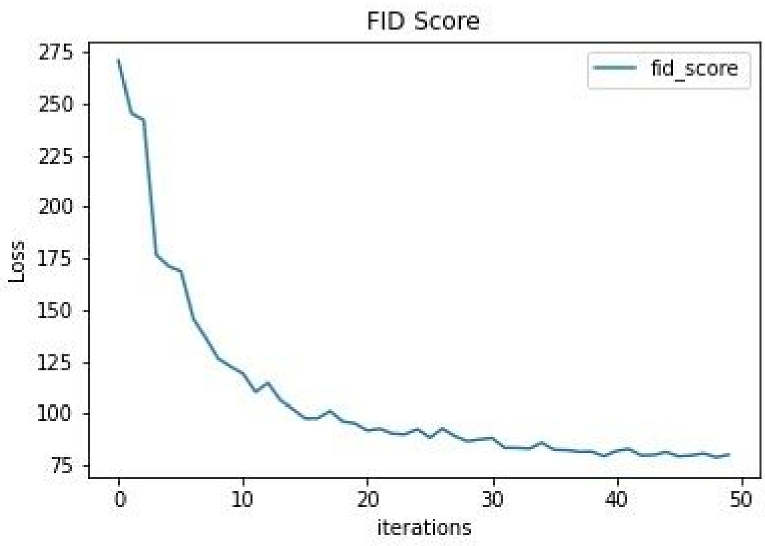


Fig) FID score vs iterations

Minimum FID score that was achieved was 78.9

Drawbacks:

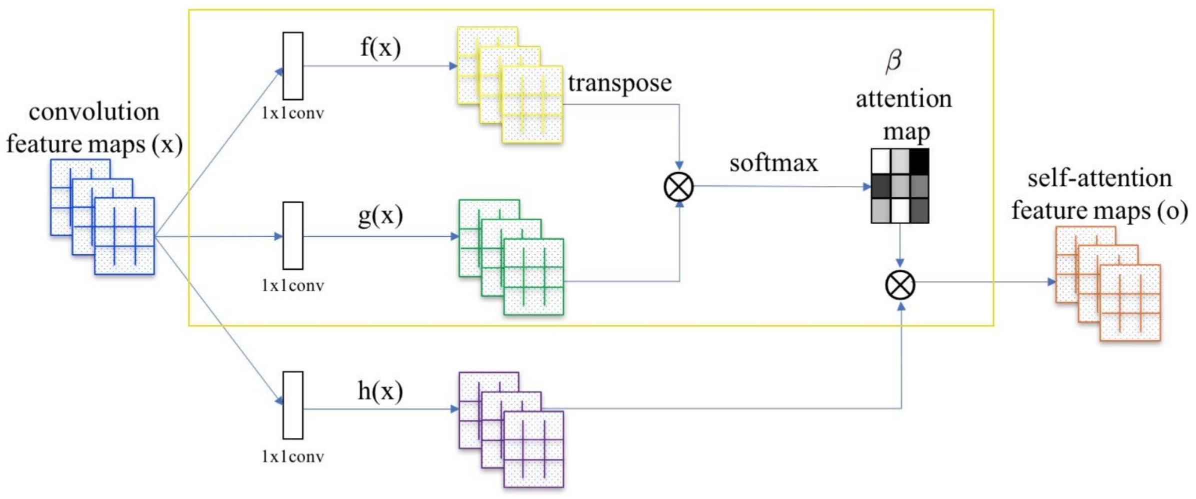
DCGANs are not able to generate good high resolution images. DCGAN also suffers to create images when we have many output classes as they try to output images of few classes mostly. Also DCGAN just uses only convolution operator which have local receptive field and the network is not able to look at different regions to generate output.

2) Self Attention GAN(SAGAN):

All the different GAN architectures were not able to produce realistic images on datasets before SAGAN because for predicting a pixel we required to look at different image regions that was not possible with just convolutional layers because convolution operator has local receptive field. So to overcome this Self-attention is used.

Self Attention:

Self-Attention so that we can provide context which is focused on the areas which are relevant to generate output at a particular location. Seft-attention layers are used in both generator and discriminator of the network. Following figure shows how self attention is applied.



$$g(x) = W_g x \qquad x \in \mathbb{R}^{C \times N},\ W_f \in \mathbb{R}^{\bar{C} \times C},\ \bar{C} = C/8$$

$$f(x) = W_f x \qquad W_g \in \mathbb{R}^{\bar{C} \times C}$$

$$s_{ij} = f(x_i)^T g(x_j)$$

$$\beta_{j,i} = \frac{\exp(s_{ij})}{\sum_{i=1}^{N} \exp(s_{ij})} \qquad \beta \in \mathbb{R}^{N \times N}$$

$\beta_{j,i}$ indicates the extent to which the model attends to the i^{th} location when synthesizing the j^{th} region.

Input and output are of the same dimensions in the self-attention layer. First the querry and key are calculated by 1*1 convolution with C_bar(another hyper-parameter) channels. These are then reshaped to N*C_bar where N=Heights*Width of image. Key and Value are then multiplied to get s in the above equation. Then we take softmax along rows to get the attention map beta which is of the shape N*N.

$$\boldsymbol{h}(\boldsymbol{x_i}) = \boldsymbol{W_h}\boldsymbol{x_i} \qquad \boldsymbol{W_h} \in \mathbb{R}^{C\times C}$$

$$\boldsymbol{o_j} = \sum_{i=1}^{N} \beta_{j,i}\boldsymbol{h}(\boldsymbol{x_i}) \qquad \boldsymbol{o} \in \mathbb{R}^{C\times N}$$

The final output of this convolutional layer is:

$$\boldsymbol{y_i} = \gamma\boldsymbol{o_i} + \boldsymbol{x_i}$$

Then we calculate value(h in figure) which is calculated using 1*1 convolution with the same channels as input. We then reshape this to dimension N*C and then it is multiplied with the attention map to get o. This is multiplied with learnable scalar gamma and added to the input x to get final output y. This learnable parameter gamma is set to 0 in the beginning of training so that the model will learn to predict good output without using attention and after some iterations the model will be able to refine output using self-attention.

Generator Architecture:

Generator architecture is similar to the DCGAN architecture, but the self-attention is applied to the last 2 layers of generator.

```
Generator(
  (main): Sequential(
    (0): SpectralNorm(
      (module): ConvTranspose2d(100, 256, kernel_size=(4, 4), stride=(1, 1), bias=False)
    )
    (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): LeakyReLU(negative_slope=0.1, inplace=True)
    (3): SpectralNorm(
      (module): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    )
    (4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): LeakyReLU(negative_slope=0.1, inplace=True)
  )
  (main1): Sequential(
    (0): SpectralNorm(
      (module): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    )
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): LeakyReLU(negative_slope=0.1, inplace=True)
  )
  (attn): Self_Attn(
    (query_conv): Conv2d(128, 64, kernel_size=(1, 1), stride=(1, 1))
    (key_conv): Conv2d(128, 64, kernel_size=(1, 1), stride=(1, 1))
    (value_conv): Conv2d(128, 128, kernel_size=(1, 1), stride=(1, 1))
    (softmax): Softmax(dim=-1)
  )
  (attn2): Self_Attn(
    (query_conv): Conv2d(64, 32, kernel_size=(1, 1), stride=(1, 1))
    (key_conv): Conv2d(64, 32, kernel_size=(1, 1), stride=(1, 1))
    (value_conv): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1))
    (softmax): Softmax(dim=-1)
  )
  (main2): Sequential(
    (0): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): Tanh()
  )
)
```

Discriminator Architecture:

Discriminator architecture is also similar to the DCGAN discriminator, just self-attention is apple to the last 2 layers.

```
Discriminator(
  (main): Sequential(
    (0): SpectralNorm(
      (module): Conv2d(3, 32, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    )
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): SpectralNorm(
      (module): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    )
    (3): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (4): LeakyReLU(negative_slope=0.2, inplace=True)
  )
  (main1): Sequential(
    (0): SpectralNorm(
      (module): Conv2d(64, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    )
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): LeakyReLU(negative_slope=0.2, inplace=True)
  )
  (attn1): Self_Attn(
    (query_conv): Conv2d(64, 32, kernel_size=(1, 1), stride=(1, 1))
    (key_conv): Conv2d(64, 32, kernel_size=(1, 1), stride=(1, 1))
    (value_conv): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1))
    (softmax): Softmax(dim=-1)
  )
  (attn2): Self_Attn(
    (query_conv): Conv2d(64, 32, kernel_size=(1, 1), stride=(1, 1))
    (key_conv): Conv2d(64, 32, kernel_size=(1, 1), stride=(1, 1))
    (value_conv): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1))
    (softmax): Softmax(dim=-1)
  )
  (main2): Sequential(
    (0): Conv2d(64, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
  )
)
```

Wasserstein Loss:

Original SAGAN paper used the hinge loss because it correlates well with the FID score, but for this assignment wasserstein loss function is used. Wasserstein loss function is given by

$$W(\mathbb{P}_r, \mathbb{P}_g) = \inf_{\gamma \in \Pi(\mathbb{P}_r, \mathbb{P}_g)} \mathbb{E}_{(x,y)\sim\gamma} \big[\, \|x - y\| \,\big]$$

Where gamma belongs to all the joint distributions over x,y such that marginals are P_r and P_g respectively. The Wasserstein distance is the minimum cost of transporting mass in converting the data distribution P_r to the data distribution P_g. This can also be represented in its dual form as

$$W(\mathbb{P}_r, \mathbb{P}_\theta) = \sup_{\|f\|_L \le 1} \mathbb{E}_{x\sim\mathbb{P}_r}[f(x)] - \mathbb{E}_{x\sim\mathbb{P}_\theta}[f(x)]$$

Where f is 1-Lipschitz function. Probability for generated data x=G(z) is just the probability of z. Using deep neural network we can represent f and optimize this using gradient ascent. This function then is very similar to discriminator but it does not output probability but a score which tells how real the image is. Due to this discriminator here is sometimes called critic because of its relation to critic in reinforcement learning. To impose the 1-Lipschitz constraint we can clip the weights of each layer.

Wasserstein loss function is used because it is a better measure of distance if the data only takes values in some lower-dimensional manifold of the complete space. We know z(noise) is 100-D in our network and is used to generate images of size 32*32 which is a much larger dimension. And from natural scene statistics we know that there is evidence that natural images lie on the lower dimensional manifold of all images possible. So this is a good loss function for GAN training.

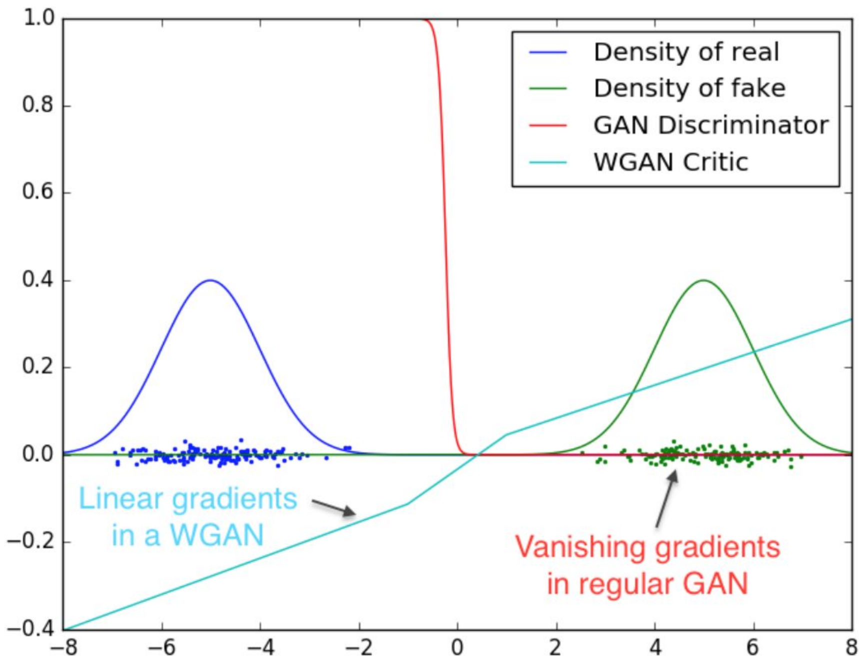


Fig) Figure showing gradients for GAN and WGAN

In this picture we can see that the gradient of the WGAN critic is smoother and we do not have a problem of vanishing/exploding gradient as in GAN discriminator. So we are able to learn even if the generator is performing poorly unlike using original GANs.

Spectral normalization:

Spectral normalization is another way to enforce 1-Lipschitz constraint than using weight clipping or gradient penalty. Spectral normalization controls the lipschitz constant of the discriminator and helps in avoiding vanishing/exploding gradient problem. In this approach we use spectral norm, which is just the largest singular value for the matrix. We need to calculate the Lipschitz constant of discriminator function

$$f(\boldsymbol{x}, \theta) = W^{L+1} \underset{\text{activation function}}{a_L}(W^L(a_{L-1}(\underset{\text{weights for each layer}}{W^{L-1}}(\dots a_1(W^1 \boldsymbol{x}) \dots))))$$

We can calculate Lipschitz constant of a function g using:

$$\|g\|_{\text{Lip}} = \sup_{\boldsymbol{h}} \sigma(\nabla g(\boldsymbol{h}))$$

We know that the Lipschitz constant for Relu is 1 because its maximum slope is 1.

So for a linear transformation W Lipschitz constant is just the spectral norm of matrix W.

$$\|g\|_{\text{Lip}} = \sup_{\boldsymbol{h}} \sigma(\nabla g(\boldsymbol{h})) = \sup_{\boldsymbol{h}} \sigma(W) = \sigma(W)$$

$$\nabla g(\boldsymbol{h}) = \nabla W \boldsymbol{h} = W$$

So the Lipschitz constant of the whole network is just a product of the spectral norm of the weight matrix of each layer.

$$\|f\|_{\text{Lip}} \leq \|(\boldsymbol{h}_L \mapsto W^{L+1} \boldsymbol{h}_L)\|_{\text{Lip}} \cdot \|a_L\|_{\text{Lip}} \cdot \|(\boldsymbol{h}_{L-1} \mapsto W^L \boldsymbol{h}_{L-1})\|_{\text{Lip}}$$
$$\cdots \|a_1\|_{\text{Lip}} \cdot \|(\boldsymbol{h}_0 \mapsto W^1 \boldsymbol{h}_0)\|_{\text{Lip}} = \prod_{l=1}^{L+1} \|(\boldsymbol{h}_{l-1} \mapsto W^l \boldsymbol{h}_{l-1})\|_{\text{Lip}} = \prod_{l=1}^{L+1} \sigma(W^l).$$

So if we divide each layer by its spectral norm then the spectral norm of the whole network will be 1.

$$\bar{W}_{\text{SN}}(W) := W / \sigma(W)$$

$$\sigma\left(\bar{W}_{\text{SN}}(W)\right) = 1$$

$$\|f\|_{\text{Lip}} = 1$$

Power Iteration:
This is an algorithm for calculating the largest singular value for a matrix. This is an iterative algorithm but the weights of the layers do not change much after one iteration so we just do one iteration of this algorithm to calculate the largest singular value. Due to this algorithm spectral normalization is computationally efficient.

$\tilde{\boldsymbol{u}}$ is first randomly initalized

$$\tilde{\boldsymbol{v}} \leftarrow W^{\mathrm{T}}\tilde{\boldsymbol{u}}/\|W^{\mathrm{T}}\tilde{\boldsymbol{u}}\|_2, \ \tilde{\boldsymbol{u}} \leftarrow W\tilde{\boldsymbol{v}}/\|W\tilde{\boldsymbol{v}}\|_2. \qquad \text{(iteratively)}$$

Spectral norm

$$\sigma(W) \approx \tilde{\boldsymbol{u}}^{\mathrm{T}}W\tilde{\boldsymbol{v}}.$$

Hyperparameters used for training:

Hyperparameter	Value or method used
Batch size	128
Optimizer	RMSprop
Learning Rate(Discriminator)	0.0001
Learning Rate(Generator)	0.0005
Activation Function	leaky-Relu
Leaky-Relu negative slope(discriminator)	0.2
Leaky-Relu negative slope(Generator)	0.1
Weight Clipping	[-0.011, 0.011]
Channels for attention layer(G)	First layer-64 Second layer-32
Channels for attention layer(D)	First layer-64 Second layer-64

Results:

Same method for calculating the FID score is used as mentioned in the result section of DCGAN. Output images of SAGAN can be seen in SAGAN_train.ipynb.

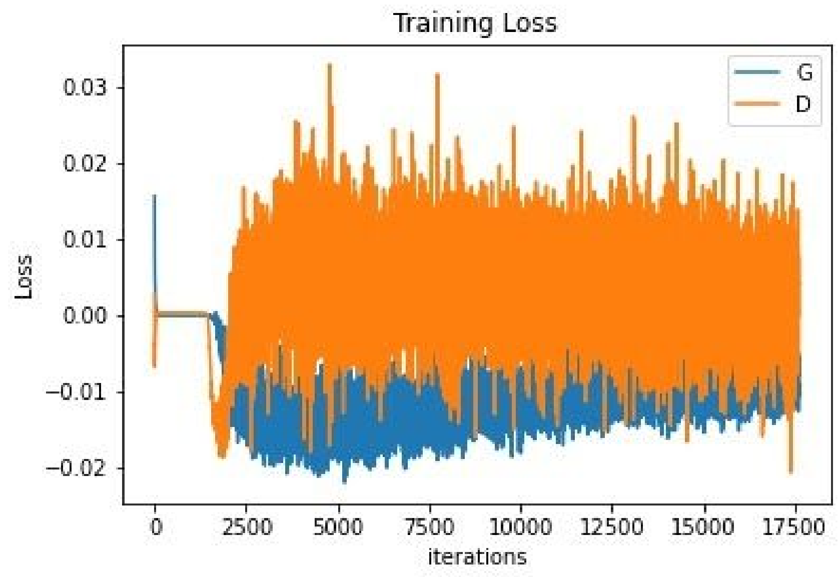


Fig) Loss of Generator and discriminator

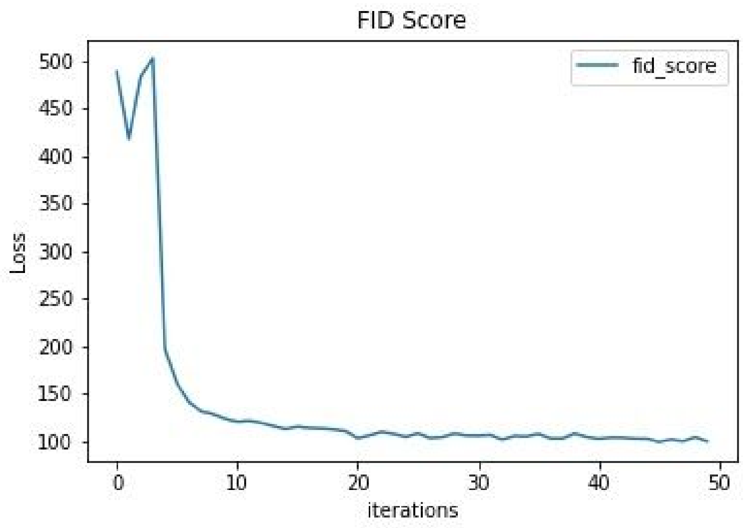


Fig) FID score vs iterations

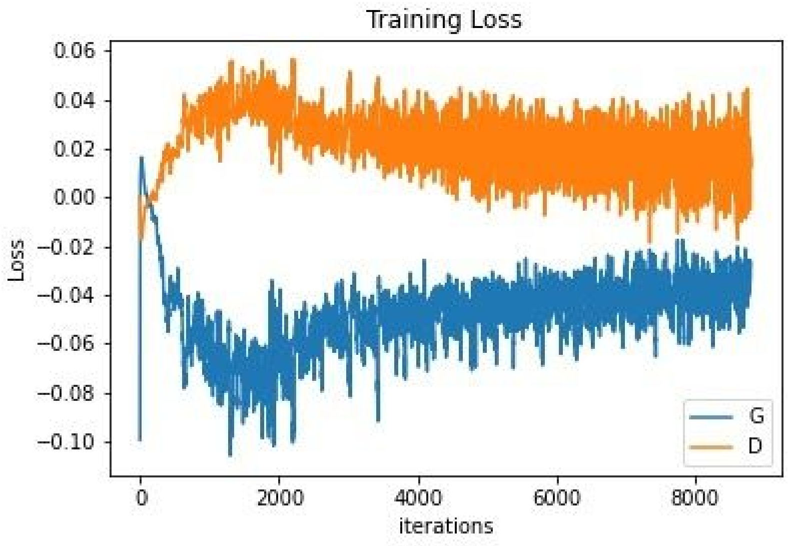
Minimum FID score that was achieved was 102.1

This is surprising as the DCGAN performs better then SAGAN. FID score with SAGAN can probably be improved by increasing the size of model and training the model for longer. Other hyper-parameters can also be tuned optimally to get better results. DCGAN may be performing better because the output size of the image is small and the DCGAN receptive field is enough to produce good output. DCGAN are also easier so they can achieve good results without much effort.

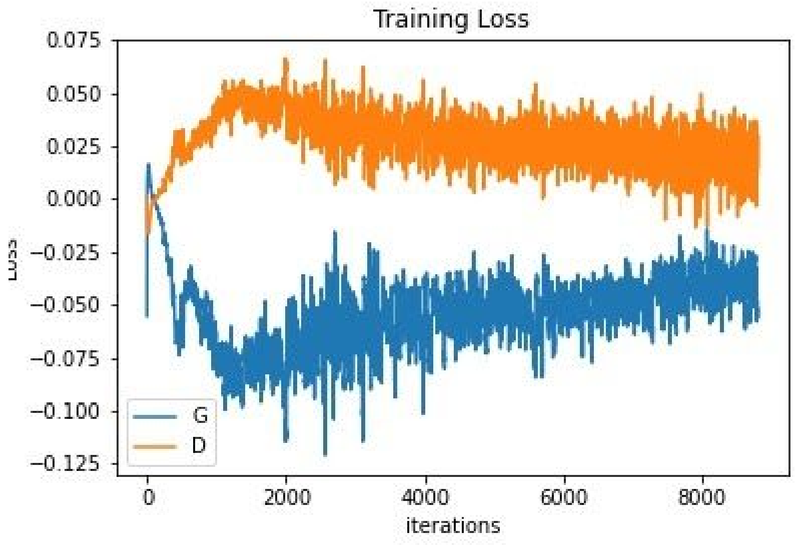
Experiments:

Experiments described below are run for fewer epochs and fewer number of images were used to calculate FID scores, so these FID scores should not be compared with FID scores used above. Plots of FID score for these are present in the experiments folder. There are not many variations of models to try for this small images, but some variations are tried and the results are provided below.

1) Using only 1 attention layer with 16 channels in Generator and Discriminator. Performance improved using 2 attention layer network which was described above.

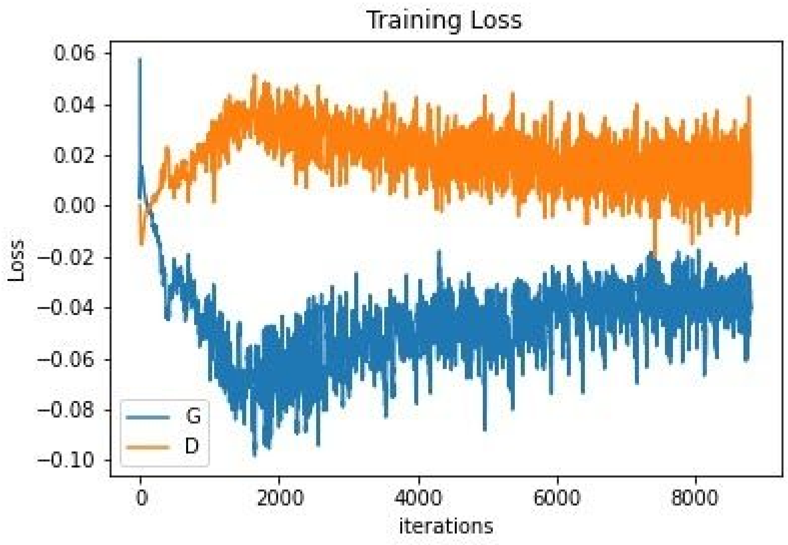


2) Using only 1 attention layer with 32 channels in Generator and Discriminator. Performance improved but 2 attention layers is better.



3) Using 2 attention layers with 16 channels each for both discriminator and generator. This network performed a little better than the previous network in

the previous point. So we can say that adding a second attention layer improved performance.



References:

1) For spectral normalization:
https://github.com/heykeetae/Self-Attention-GAN/blob/master/spectral.py

2) For FID score:
https://github.com/mseitzer/pytorch-fid