CSE 676 - Project 01:
Generative Adversarial Networks (GANs)
DCGAN and SAGAN

Md Moniruzzaman Monir, # 50291708

October 23, 2019

Simple Neural Network works good in classification or detection but it can not generate new samples. GAN can generate something fake (e.g. image) that actually looks like real. It can generate a new 'cat' picture that we have not seen before. In this project, I have implemented two types of GAN : one is Deep Convolution GAN (DCGAN), and another is Self-Attention GAN (SAGAN). Some important aspects of my implementations are highlighted in Table 1.

Type	Loss Function	Evaluation metric	Conditional	Normalization	Self-Attention	Dataset
DC-GAN	BCE	FID	No	Batch	No	CIFAR-10
SA-GAN	Wasserstein	FID	Yes	Batch and Spectral	Yes	CIFAR-10

Table 1: An overview of loss function, eveluation metric, dataset, and other salient features of my DCGAN and SAGAN implementation. Fréchet Inception Distance (FID) is used as evaluation metric in both implementations. Both Discriminator and Generator are conditional in SAGAN.

1 Deep Convolutional Gan (DCGAN)

In a simple neural network (NN), we never focus on the weak points of the network. Like while a NN is learning digit recognition we never focus on for which digits the NN is actually making the most mistakes. It can happen that in the middle of the training the NN actually doing mistakes by recogniging 6 as 9. So, if we can focus on this weak point at that time then we can actually train the NN only for these two digits, and thus make it better classifier. But this is not happening in the training of a NN. This strategy is achieved in GAN through an adversary network which is known as `Discriminator`. This network keeps hammering on the weak points of the other neural network or any variant of NN. We can think these two networks as two players. Every time one player finds a strategy that is extremely good against the opponent player, then the opponent also learns a way to deal with that new strategy. Both players are continuously learning how to play a better and better opponent.

In DCGAN, I use two convolutional neural networks (CNN): one works as a 'Discriminator' (D) and other works as a 'Generator' (G). D and G compete with each other making each other stronger at the same time. I build the GAN by concatenating G and D using Keras framework. I am giving a random noise of 100 elements from a gaussian distribution (latent space) to G, and in the training it learns the mapping from the latent space to a sample from the original distribution of real data. So, basically G takes as input a random noise and transforms it into a sample from the real data distribution. D distinguises between output data point (fake) from G and training data samples (real) from cifar-10 dataset. G and D are working against each other like a zero-sum game (adversarial). G is constantly trying to fool the D into believing that, into making the decision that input generated by it is from the training distribution (real) while D is also learning how efficiently it can detect fake images. In this process weights of G learns a transformation which enables one to convert the random noise input vector into a sample from the model distribution.

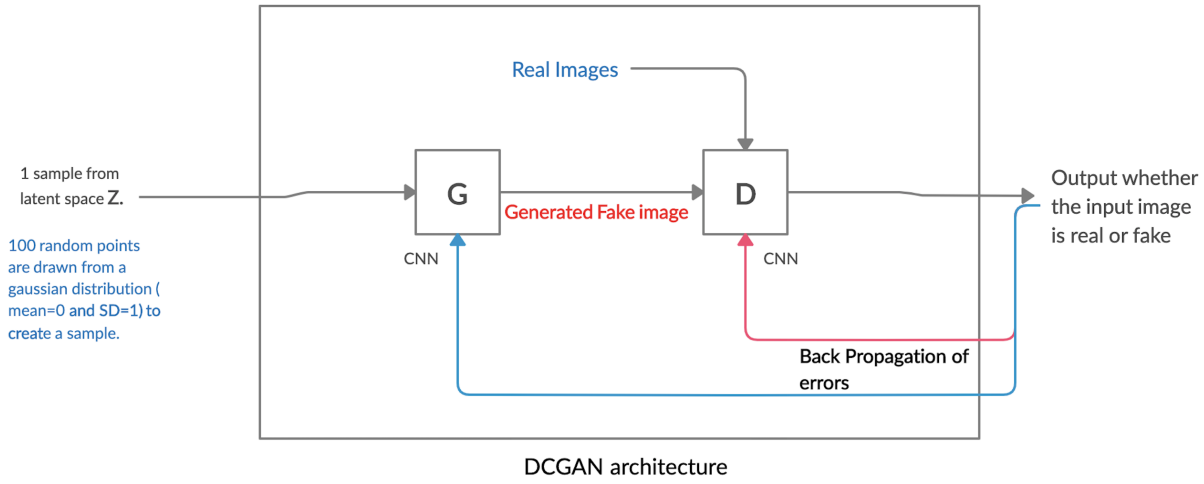


Figure 1: High architectural overview of the DCGAN.

1.1 Model Architecture

The architecture of my discriminator and generator is shown in image 2. The discriminator takes an image of shape 32x32x3 as input and outputs 0/1. I use 4 convolution layers and 1 fully connected layer to downsample the image into a single value. For downsampling I use strided convolution instead of max pooling as this gives better results. In every convolutional layer, I use **batch normalization** to avoid the imbalance of the gradients. In every iteration weights are updated via backpropagation so some weights can become very large comparing to the others, and thus can create the vanishing and exploding gradient problem. By using batch normalization this can be avoided. I use **Leaky ReLu** as activation functions in every convolutional layers as it gives better results than 'Relu' function. The last layer is a fully connected dense layer with **sigmoid** activation function. I also add dropout layer before this layer.

Model: "Generator"

Layer (type)	Output Shape	Param #
dense_2 (Dense)	(None, 2048)	206848
reshape_1 (Reshape)	(None, 2, 2, 512)	0
batch_normalization_4 (Batch	(None, 2, 2, 512)	2048
leaky_re_lu_5 (LeakyReLU)	(None, 2, 2, 512)	0
conv2d_transpose_1 (Conv2DTr	(None, 4, 4, 256)	3277056
batch_normalization_5 (Batch	(None, 4, 4, 256)	1024
leaky_re_lu_6 (LeakyReLU)	(None, 4, 4, 256)	0
conv2d_transpose_2 (Conv2DTr	(None, 8, 8, 128)	819328
batch_normalization_6 (Batch	(None, 8, 8, 128)	512
leaky_re_lu_7 (LeakyReLU)	(None, 8, 8, 128)	0
conv2d_transpose_3 (Conv2DTr	(None, 16, 16, 64)	204864
batch_normalization_7 (Batch	(None, 16, 16, 64)	256
leaky_re_lu_8 (LeakyReLU)	(None, 16, 16, 64)	0
conv2d_transpose_4 (Conv2DTr	(None, 32, 32, 3)	4803

Total params: 4,516,739
Trainable params: 4,514,819
Non-trainable params: 1,920

(a) Generator Model

Model: "Discriminator"

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 16, 16, 64)	4864
leaky_re_lu_1 (LeakyReLU)	(None, 16, 16, 64)	0
conv2d_2 (Conv2D)	(None, 8, 8, 128)	204928
batch_normalization_1 (Batch	(None, 8, 8, 128)	512
leaky_re_lu_2 (LeakyReLU)	(None, 8, 8, 128)	0
conv2d_3 (Conv2D)	(None, 4, 4, 256)	819456
batch_normalization_2 (Batch	(None, 4, 4, 256)	1024
leaky_re_lu_3 (LeakyReLU)	(None, 4, 4, 256)	0
conv2d_4 (Conv2D)	(None, 2, 2, 512)	3277312
batch_normalization_3 (Batch	(None, 2, 2, 512)	2048
leaky_re_lu_4 (LeakyReLU)	(None, 2, 2, 512)	0
flatten_1 (Flatten)	(None, 2048)	0
dropout_1 (Dropout)	(None, 2048)	0
dense_1 (Dense)	(None, 1)	2049

Total params: 4,312,193
Trainable params: 4,310,401
Non-trainable params: 1,792

(b) Discriminator Model

Figure 2: Architecture of Discriminator and Generator model in DCGAN

The generator takes a noise of shape 100x1 and outputs a 32x32x3 image. In generator, I use the same batch normalization in all hidden layers and Leaky Relu activation functions. Also I use 2D strided convolutional transpose in all hidden layers for upsampling. This actually does the opposite of strided convolution. In the final layer I use **tanh** activation function. So all the pixel values will be between -1 to 1 in the fake images generated by the generator.

As the discriminator can get real or fake images as input and fake images have pixel values between -1 to 1, so I preprocess the real images (50k training images and 10k testing images of cifar-10 dataset) to convert the pixel values from [0,255] range to [-1,1] range.

1.2 Training Description

I use **FID score** to evaluate how good my model is or how good images are generated by my model. I use the 'inceptionV3' model to calculate the FID score. Here, we give two sets of images to this 'inceptionV3' model and gets two sets of final activation feature maps from this model for the given two sets of images. We then calcualte the FID score from these two sets of activation feature maps. For each set we calculate the mean and sigma values and use those in the equation to calculate the FID score. If two sets of images are same, then the inceptionV3 model will generate same activation feature maps, and the FID score will be 0. It indicates that the lower the FID score the better the model is. I choose 1000 images from the test images of cifar10 dataset and 1000 images from my generated model to calcualte the FID. InceptionV3 model expects the input images shapes to be 299x299x3 (the lowest possible dimension is 75x75x3). As our images are 32x32x3, so I have to upscale them using a function from that model. I can choose the minimum value for upscaling as 75x75x3 but it gives very high FID score because inceptionV3 model works properly if the image shape is 299x299x3. But when I upscale to 299x299x3 then I faced a serious technical issue. I was training my model in google colab platform which allows me at most 25GB memory, and if I upscale my 10k testing images to 299x299x3 then all memory was exhausted, and the session was terminated. So, instead of upscaling the images I calculate the activation feature maps for every image and save it into a numpy array. This solves the memory issue as now I am storing only 2048 value of the activation maps for an image while previously when I was upscaling to 299x299x3 then I was storing 299*299*3 value for each image.

I trained my DCGAN model with different values of the hyper-parameters (e.g. negative slope value (alpha) of Leaky Relu, value of standard deviation for initializing the kernel in convolution, learning rate and beta value of the optimizers etc.). I use 'Adam' optimizer in both discriminator and generator. I find the best images and FID score in approximately 170k iterations with the following hyper-parameter settings: alpha = 0.2, generator learning rate =0.0001, discriminator learning rate = 0.0004, SD value for kernel initialize = 0.02, beta value of the optimizer = 0.5

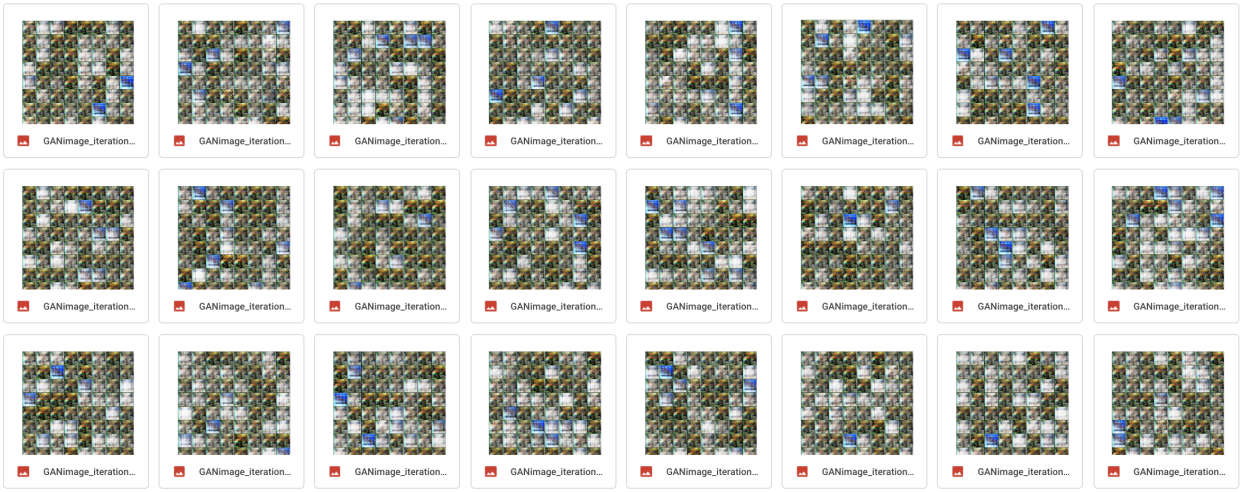


Figure 3: Mode collpase scenario in dcgan training

The lowest FID score was approximately 80. I use the two time scale update rule (TTUR) and get the best results. I also used the epoch (feed the whole training dataset batchwise in training) but this gives poor results. I think this is because there is not that much randomness in chossing 64 images (batch size was 64) as it is sequentially choosing 64 images one after another in every epoch. I also use label smoothing for getting best output model. For every real images instead of assigning label 1 I assigned a random value between 1 to 0.95. It actually helps the discriminator model not to overfit in the training. One interesting result I get in my experiments is the mode collapse [Image 10]. I trained the model with different value and model architecture, and for the below setting I stuck into mode collapse which actually represents the scenario where generator finds some fixed sets of images to fool the discriminator. That means there is no variety of the generated images from the generator and it is generating same set of images to fool the discriminator. In that training, I use label smoothing like Real images have label (1-0.05) and the hyper-parameter was : generator learning rate=0.0002, beta value of generator =0.5, discriminator learning rate=0.0002, beta value of discriminator =0.5 and leaky alpha=0.2. I added one extra layer in D and G (compared to my submitted model) with stride=1.

1.3 Results

Grid of generated images from my trained DCGAN :

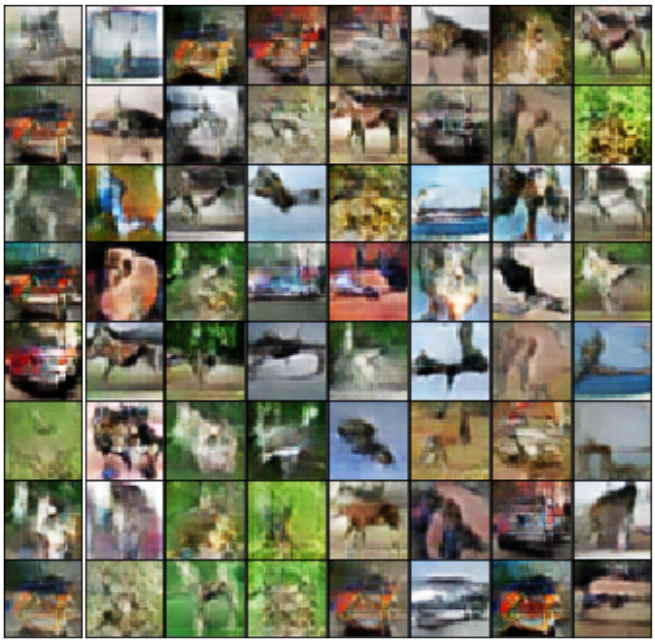


Figure 4: A grid of 64 generated images from the trained DCGAN

FID score is used as evaluation metric. FID over Iterations in training is given below :

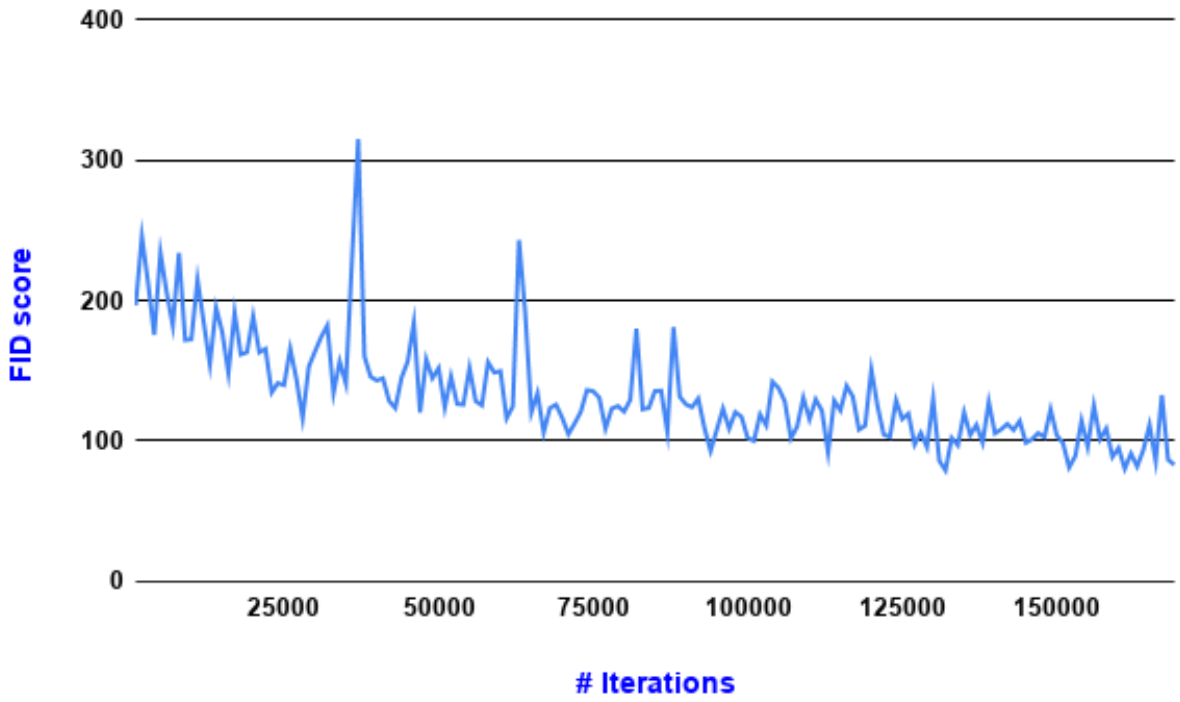


Figure 5: FID vs no of iterations. This graph shows that the FID score is reducing slowly indicating better model for large number of iterations.

2 Self-Attention GAN (SAGAN)

In DCGAN, there are some inherent problems due to the convolutional architecture like it can only learns the local structure of an image. In a convolution operation, it is not possible for an output on the top-left position to have any relation to the output at bottom-right. It can not model long term dependencies for image generation tasks. We can see that from the generated images of my DCGAN model. It detects the textures very well but can not learn the geometric shape from the images due to the limiting local view. If it can somehow get the global view while doing the local convolution then it can actually learn the geometric shape and model the long range dependencies. Since convolution operator has a local receptive field, long ranged dependencies can only be processed after passing through several convolutional layers or increasing the size of the kernel. But in that case we lose the computional efficiency of the CNN architecture. Researchers observed that these convolutional GANs have difficulty in modeling some image classes than others when trained on **multi-class datasets**. They observed that convolutional GANs could easily generate images with a simpler geometry like ocean, sky etc. but failed on images with some specific geometry like dogs, horses and many more. To solve this issue self-attention is introduced in the GAN model which is called self-attention GAN or SAGAN. It has a good balance between computational efficiency and large receptive field at the same time. On the left of the image 6, we get our feature maps from the previous convolutional layer. Lets suppose it is of the dimension (512 x 7x 7) where 512 is the number of channels (no of features) and 7 is the spatial dimension. We flatten the spatial dimension so 7 x 7 will be 49, and the overall dimension will be 512 x 49 (reduce ta a matrix). This is represented by the shape C x N where C denotes the no of channels/features and N denots no of attention (total no of points in the spatial structure). The reduced feature maps are passed through three 1x1 convolutions seperately. The filters in this three convolutions are f, g and h. What 1x1 convolution does is that it reduces the number of channels in the image. f and g have 64 (C/8, here 512/8 = 64) those filters, so the dimension of the filter becomes (64 x 512 x 1 x1). The h has 512 of those filters. After the image gets passed through we get three feature maps of dimensions (64 x 49),

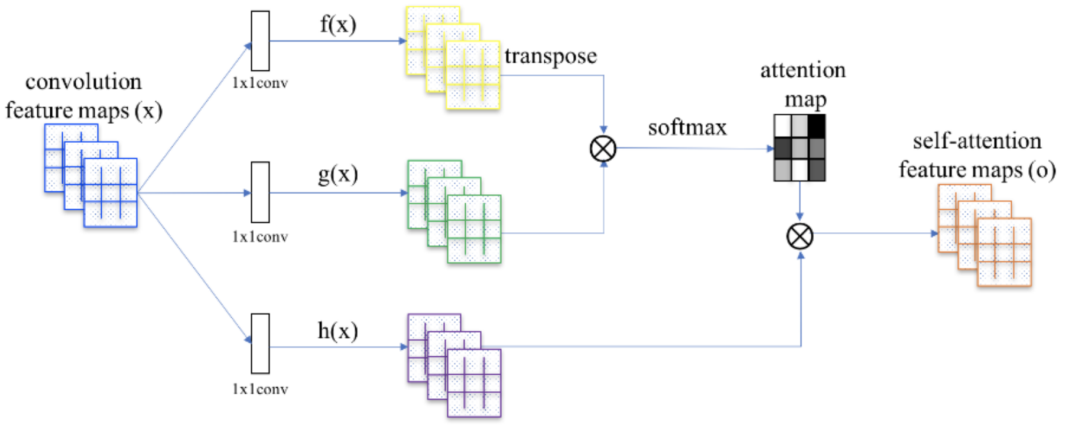


Figure 6: Architecture of a Self Attention module. The X denotes matrix multiplication. The softmax operation is performed on each row. This module is stacked after a selected convolution block present in a G and D. The attention layer helps the network capture the fine details from even distant parts of the image and remember, it does not replace convolution rather it is complementary to the convolution operation.

(64 x 49) and (512 x 49). Now we can perform the self-attention over it. We transpose the output from f layer (becomes 49 x 64) and matrix-multiply it by the output of g layer (64 x 49) and then passes though a softmax activation to get the attention layer. we get an output attention map of shape (49 x 49). Each row in this attention matrix actually infers for that pixel location in the image which other pixel locations will need to look at. Then we matrix multiply the output of h layer with the attention map and get the output of the shape (512 x 49). One last thing that is proposed in the research paper is that to multiply the final output by a learnable scale parameter and add back the input as a residual connection. Let's say the x was the image and o is the output, we multiply o by a parameter y. The final output O then becomes, $O = y * o + x$. Another problem in DCGAN is mode collapse shown in image 10. To solve this issue I use wasserstein loss function in the training.

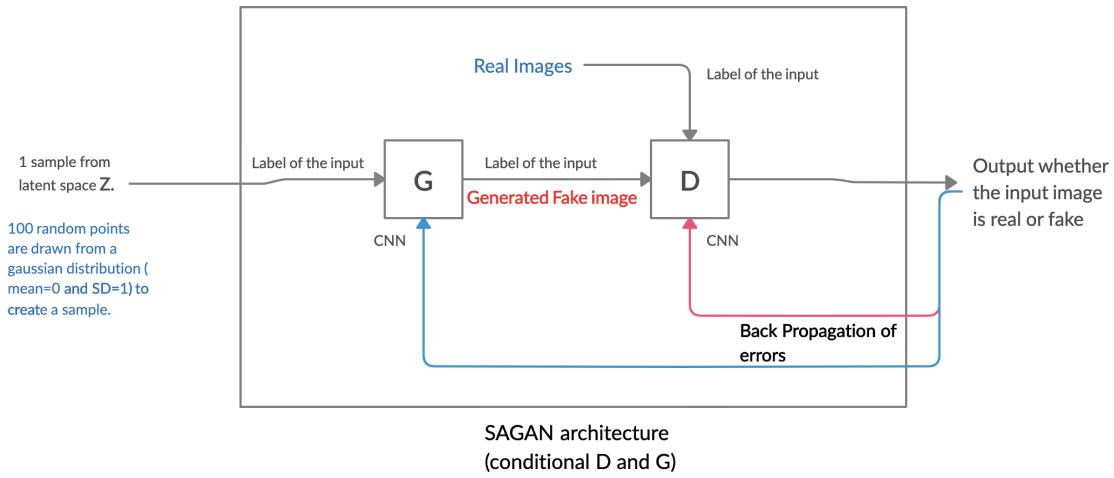


Figure 7: High architectural overview of the SAGAN.

2.1 Model Architecture

The architecture of my discriminator and generator is shown in image 12. The discriminator and generator both are conditional. Discriminator takes an image of shape 32x32x3 with a label as input and outputs 0/1. I use 4 convolution layers with spectral normalization and 1 fully connected layer to downsample the image into a single value. For downsampling I use strided convolution instead of max pooling as this gives better results. In every convolutional layer, I use batch normalization to avoid the imbalance of the gradients and **spectral normalization** to constrains the **Lipschitz constant** of the convolutional filters. Spectral norm is used as a way to stabilize the training of the discriminator network. I use the self attention layer after the first layer with 64 feature maps. I use **Leaky ReLu** as activation functions in every convolutional layers as it gives better results than 'Relu' function. The last layer is a fully connected dense layer with **linear** activation function.

Model: "generator"

Layer (type)	Output Shape	Param #	Connected to
input_3 (InputLayer)	(None, 100)	0	
input_4 (InputLayer)	(None, 10)	0	
concatenate_2 (Concatenate)	(None, 110)	0	input_3[0][0] input_4[0][0]
dense_3 (Dense)	(None, 2048)	227328	concatenate_2[0][0]
reshape_1 (Reshape)	(None, 2, 2, 512)	0	dense_3[0][0]
batch_normalization_4 (BatchNor	(None, 2, 2, 512)	2048	reshape_1[0][0]
leaky_re_lu_5 (LeakyReLU)	(None, 2, 2, 512)	0	batch_normalization_4[0][0]
up_sampling2d_1 (UpSampling2D)	(None, 4, 4, 512)	0	leaky_re_lu_5[0][0]
conv_s_n2d_5 (ConvSN2D)	(None, 4, 4, 256)	3277312	up_sampling2d_1[0][0]
batch_normalization_5 (BatchNor	(None, 4, 4, 256)	1024	conv_s_n2d_5[0][0]
leaky_re_lu_6 (LeakyReLU)	(None, 4, 4, 256)	0	batch_normalization_5[0][0]
up_sampling2d_2 (UpSampling2D)	(None, 8, 8, 256)	0	leaky_re_lu_6[0][0]
conv_s_n2d_6 (ConvSN2D)	(None, 8, 8, 128)	819456	up_sampling2d_2[0][0]
batch_normalization_6 (BatchNor	(None, 8, 8, 128)	512	conv_s_n2d_6[0][0]
leaky_re_lu_7 (LeakyReLU)	(None, 8, 8, 128)	0	batch_normalization_6[0][0]
up_sampling2d_3 (UpSampling2D)	(None, 16, 16, 128)	0	leaky_re_lu_7[0][0]
conv_s_n2d_7 (ConvSN2D)	(None, 16, 16, 64)	204928	up_sampling2d_3[0][0]
batch_normalization_7 (BatchNor	(None, 16, 16, 64)	256	conv_s_n2d_7[0][0]
leaky_re_lu_8 (LeakyReLU)	(None, 16, 16, 64)	0	batch_normalization_7[0][0]
self_attention_2 (SelfAttention	(None, 16, 16, 64)	5121	leaky_re_lu_8[0][0]
up_sampling2d_4 (UpSampling2D)	(None, 32, 32, 64)	0	self_attention_2[0][0]
conv_s_n2d_8 (ConvSN2D)	(None, 32, 32, 3)	4806	up_sampling2d_4[0][0]

Total params: 4,542,791
Trainable params: 4,540,420
Non-trainable params: 2,371

(a) Generator Model

Model: "Discriminator"

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	(None, 32, 32, 3)	0	
conv_s_n2d_1 (ConvSN2D)	(None, 16, 16, 64)	4864	input_1[0][0]
leaky_re_lu_1 (LeakyReLU)	(None, 16, 16, 64)	0	conv_s_n2d_1[0][0]
self_attention_1 (SelfAttention	(None, 16, 16, 64)	5121	leaky_re_lu_1[0][0]
conv_s_n2d_2 (ConvSN2D)	(None, 8, 8, 128)	204928	self_attention_1[0][0]
batch_normalization_1 (BatchNor	(None, 8, 8, 128)	512	conv_s_n2d_2[0][0]
leaky_re_lu_2 (LeakyReLU)	(None, 8, 8, 128)	0	batch_normalization_1[0][0]
conv_s_n2d_3 (ConvSN2D)	(None, 4, 4, 256)	819456	leaky_re_lu_2[0][0]
batch_normalization_2 (BatchNor	(None, 4, 4, 256)	1024	conv_s_n2d_3[0][0]
leaky_re_lu_3 (LeakyReLU)	(None, 4, 4, 256)	0	batch_normalization_2[0][0]
conv_s_n2d_4 (ConvSN2D)	(None, 2, 2, 512)	3277312	leaky_re_lu_3[0][0]
batch_normalization_3 (BatchNor	(None, 2, 2, 512)	2048	conv_s_n2d_4[0][0]
leaky_re_lu_4 (LeakyReLU)	(None, 2, 2, 512)	0	batch_normalization_3[0][0]
flatten_1 (Flatten)	(None, 2048)	0	leaky_re_lu_4[0][0]
input_2 (InputLayer)	(None, 10)	0	
concatenate_1 (Concatenate)	(None, 2058)	0	flatten_1[0][0] input_2[0][0]
dense_1 (Dense)	(None, 512)	1054208	concatenate_1[0][0]
dense_2 (Dense)	(None, 1)	513	dense_1[0][0]

Total params: 5,369,986
Trainable params: 5,367,234
Non-trainable params: 2,752

(b) Discriminator Model

Figure 8: Architecture of Discriminator and Generator model in SAGAN

The generator takes a noise of shape 100x1 with a random label and outputs a 32x32x3 image. In generator, I use the attention layer before the last step with 64 feature maps. I also use the same batch normalization and spectral normalization in all hidden layers and Leaky Relu activation functions. Also I use 2D strided convolutional transpose in all hidden layers for upsampling. This actually does the oppostie of strided convolution. In the final layer I use **tanh** activation function. So all the pixel values will be between -1 to 1 in the fake images generated by the generator.

2.2 Training Description

I use wasserstein loss in the training and weight clipping mechanism to avoid gradient explosion. For every batch I trained the discriminator 5 times and the generator 1 time. In those 5 iteration for discriminator I clipped the weights using clip value as 0.01. Also as the models are conditional so I have to pass the labels with image and noise in the training. Other strategies are same as the DCGAN training. I train the model only for 30k iterations as it takes a lot of time in every iteration than dcgan training. I use TTUR in the gradient learning rate of the Adam optimizer. I use the following hyper-parameter settings: alpha = 0.2, generator learning rate =0.0001, discriminator learning rate = 0.0004, SD value for kernel initialize = 0.02, beta value of the optimizer = 0.5 in my training.

2.3 Results

Grid of generated images from my trained SAGAN :

Figure 9: A grid of 64 generated images from the trained SAGAN

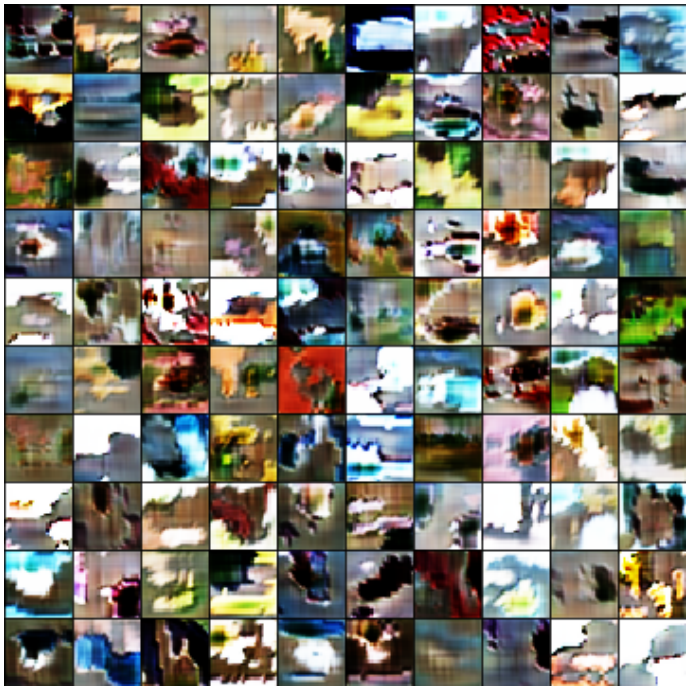


Figure 10: Another grid of 100 generated images from the trained SAGAN

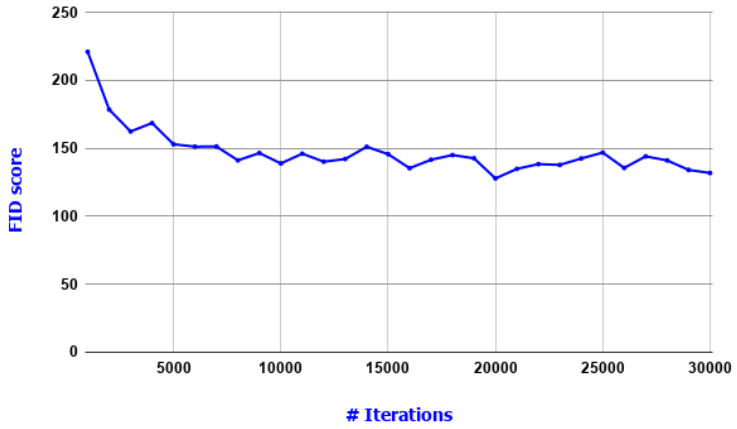
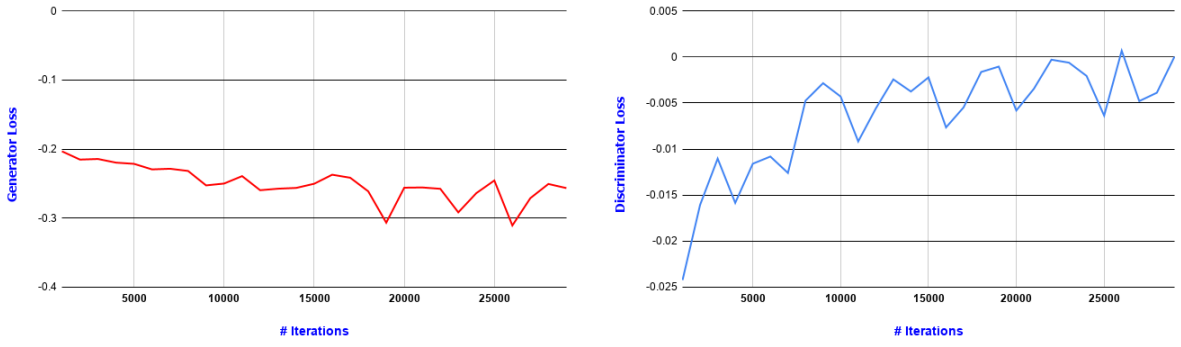


Figure 11: FID vs no of iterations. This graph shows that the FID score is reducing slowly indicating better model for large number of iterations.



(a) Generator Loss **(b)** Discriminator Loss

Figure 12: Validation Loss curve for generator and discriminator in SAGAN