E0-270 Machine Learning
Final Project Report

Generative Adversarial Networks

Mohit Kumar
M.Tech in Artificial Intelligence
SR No. : 04-01-03-10-51-21-1-19825
mohitk2@iisc.ac.in

Problem Statement

This project is about learning a generative model based on differentiable generator networks. In this project, two types of Generative Adversarial Networks (GANs): One known as the Deep Convolutional GAN (DCGAN) and the other known as Self-Attention GAN (SA-GAN) have been implemented using the CIFAR10 dataset. As a part of this project,following tasks were performed :

1. Implemention of a basic DCGAN that consists of the two networks Generator(G) and Discriminator(D) based on deep convolutional networks. This is an unconditional/unsupervised GAN as they do not take the label of the images as inputs to the discriminator and generator networks.
2. Implementation of SAGAN
   (a) The generator and discriminator networks of the basic DCGAN were made conditional so that the generator and discriminator networks also take the labels of the images as inputs.
   (b) Spectral Normalization(SN) was applied to the generator and discriminator networks (existing implementation of SN was used).
   (c) A Self Attention module was implemented for use in the discriminator and generator networks.
   (d) Wasserstein Loss function was implemented with gradient penalty and used for training the discriminator network.
3. Frechet Inception Distance (FID) was used as the evaluation metric (existing implementation of FID was used). FID score was computed every 500 iterations and plotted.
4. The Generator and Discriminator loss curves obtained during training was plotted.
5. 50 images were generated from each of the models and saved in the respective DCGAN and SAGAN directories.

1 Introduction

Generative modelling is of two types:

Density Estimation Methods Models that generate samples and infer probability distributions.

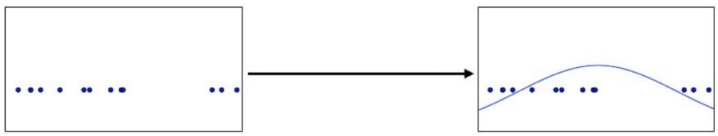


Figure 1: density estimation(11)

Sample Generation Methods Models that generate samples but do not infer probability distributions.

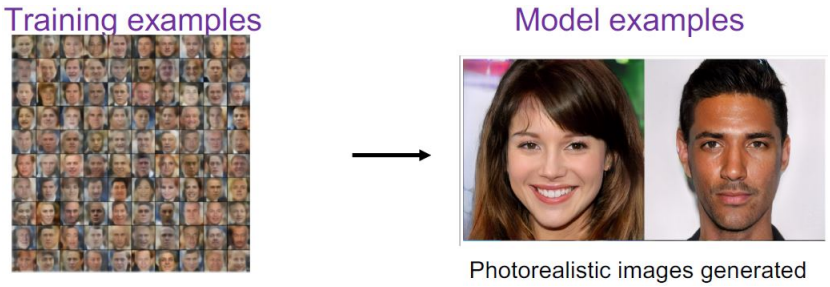


Figure 2: sample generation(11)

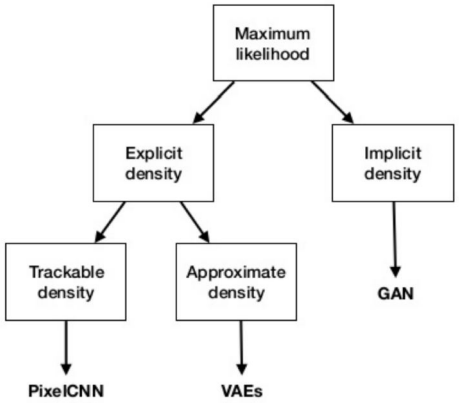
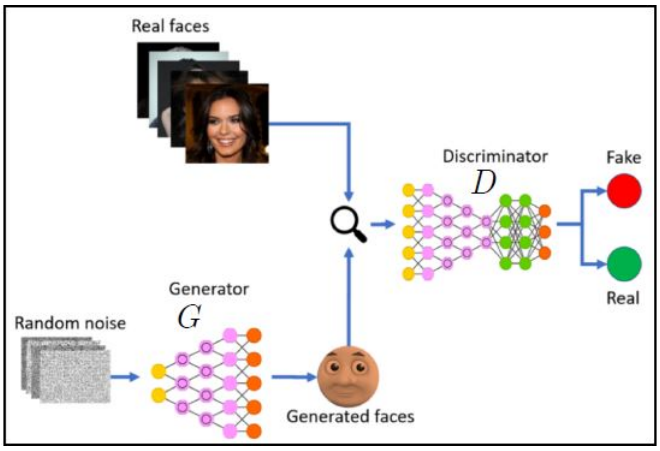


Figure 3: Taxonomy of Generative Models(11)

GANs belong to the second class of models. They are based on Game Theory and correspond to a two player minimax game. They consist of two separate networks referred to as the generator and the discriminator networks. The generator captures the data distribution and is capable of generating fake images from random noise. The discriminator is optimized to differentiate between real images and the fake images generated from the generator. The generator is provided with the gradient from the discriminator and is optimized to create fake images the discriminator considers as real.

Value Function of Minimax played by G, D
$\min_G \max_D \quad E_{x\sim p_X}[\log D(x)] + E_{z\sim pz}[\log(1 - D(G(z))]$
p_X : data distribution, represented by samples
$p_{G(z)}$: model distribution, z modeled as Gaussian.



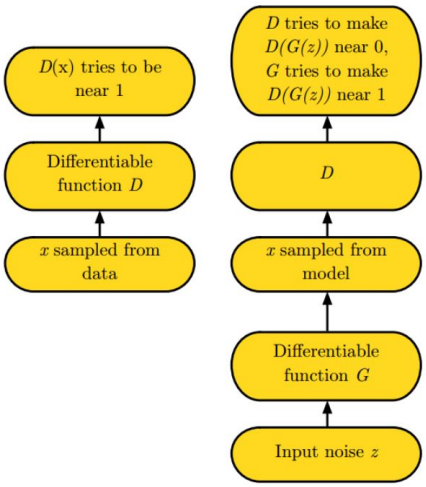


Figure 4: GAN framework(11)

Conditional GANs learn to sample from a distribution p(x|y) rather than simply sampling from the marginal distribution p(x). Conditional GAN objective function is given by

$$\min_{G}\max_{D} V(D,G) = E_{x\sim pdata(x)} \log D(x \mid y)\Big] \; + E_{z\sim p(z)} \log(1 - D(G(z \mid y)))\Big]$$

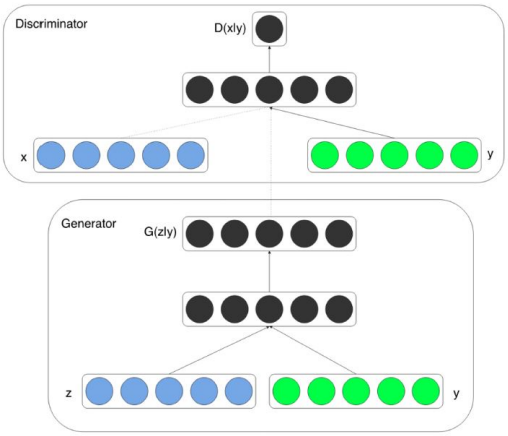


Figure 5: cGAN architecture(11)

2 Literature Review

In a DCGAN, the discriminator and generator networks are deep CNNs. The generator samples a vector from a random distribution and manipulates it with transposed convolutions until it is in the shape of an input image. Discriminator network is just the opposite of the Generator network with Deconv(Transpose) convolutions replaced with Conv layers.The final output from the discriminator is of dimension 1 representing the probability of the input being real. It is used for unsupervised learning since it does not take into account any labels for the data. It uses batchnorm in both the generator and the discriminator. It uses ReLU activation in generator for all layers except for the output, which uses Tanh and uses LeakyReLU activation in the discriminator for all layers. All the hyperparameters used are as mentioned in the DCGAN paper.(9)

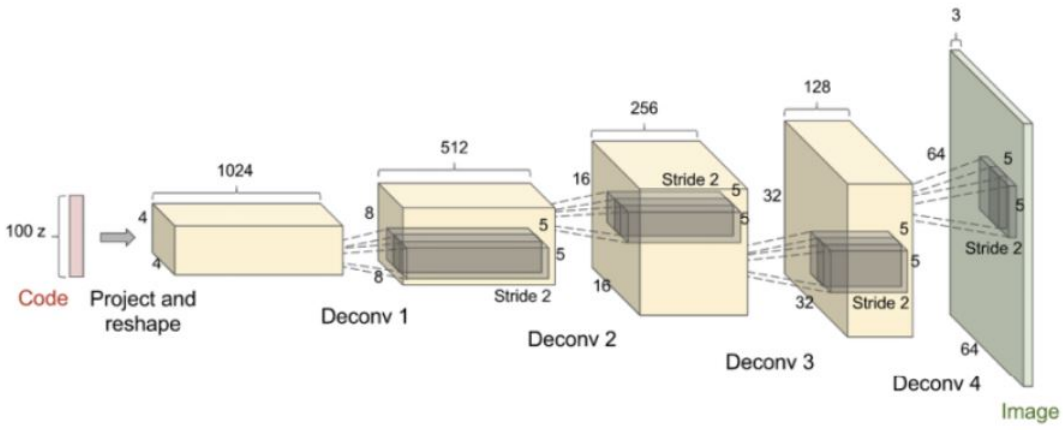


Figure 6: DCGAN generator(9)

DCGANs exhibit limitations while modelling long range dependencies for image generation tasks. For example, dogs are often drawn with realistic fur texture but without clearly defined separate feet. The problem with DCGANs exists because the model relies heavily on convolution to model the dependencies across different image regions. Since convolution operator has a local receptive field, long ranged dependencies can only be processed after passing through several convolutional layers. This could prevent learning about long-term dependencies for a variety of reasons:

- A small model may not be able to represent them.

- Increasing the size of the convolution kernels can increase the representational capability of the network but doing so also loses the computational and statistical efficiency obtained by using local convolutional structure.
- **Mode collapse**Generator collapses providing limited sample variety.

To mitigate first two issues self-attention(SA) module is introduced in DCGANs. Self Attention GANs exhibits a better balance between the ability to model long-range dependencies and the computational and statistical efficiency.

To alleviate the third issue, GANs are trained with Wasserstein Loss. The Wasserstein distance is the minimum cost of transporting mass to convert the data distribution q to the data distribution p. Even when two distributions are located in lower dimensional manifolds without overlaps, Wasserstein distance can still provide a meaningful and smooth representation of the distance in between them. The Equation for Wasserstein distance is given as follows:
$W\left(P_r, P_g\right) = \inf_{\gamma \in (1, P_s)} \mathbb{E}_{(x,y)\sim r}[\|x - y\|]$ is intractable
Using Kantorovich-Rubinstein duality, simplify to
$W\left(\mathbb{P}_r, \mathbb{P}_\theta\right) = \sup_{f\|_<} \mathbb{E}_{x\sim\mathbb{P}_r}[f(x)] - \mathbb{E}_{x\sim\mathbb{P}_\theta}[f(x)]$
where sup is the least upper bound f is a 1-Lipschitz function following the constraint: $|f\left(x_1\right) - f\left(x_2\right)| \leq |x_1 - x_2|$
i.e, a function with a bounded first derivative So to calculate the Wasserstein distance, we just need to find a 1-Lipschitz function. A Lipschitz function is such that

$$|f(x) - f(y)| \leq \alpha |x - y|$$

i.e., slope of secant line joining $(x, f(x))$ and $(y, f(y))$ is always bounded above by α

The Wasserstein GAN optimizes a different loss function.(1) The only two changes are to remove the sigmoid activation from the final layer to directly be used for optimization and to introduce weight clipping to satisfy the Lipschitz constraint.

Improved Training for Wasserstein GANs(3) found that weight clipping to enforce the Lipschitz constraint still led to undesired behavior.Instead, they penalized the norm of the gradient of the critic (discriminator) with the respect to its input to enforce the constraint which lead to significantly improved results.

Batch normalization causes the entire batch of outputs to be dependent on the entire batch of inputs which makes the gradient penalty invalid. Layer normalization is used as a replacement.(I have used instance normalization). I have used the Wasserstein loss with gradient penalty for implementing the SAGAN.

Algorithm 1 WGAN with gradient penalty. We use default values of $\lambda = 10$, $n_{\text{critic}} = 5$, $\alpha = 0.0001$, $\beta_1 = 0$, $\beta_2 = 0.9$.

Require: The gradient penalty coefficient λ, the number of critic iterations per generator iteration n_{critic}, the batch size m, Adam hyperparameters α, β_1, β_2.
Require: initial critic parameters w_0, initial generator parameters θ_0.
while θ has not converged **do**
  for $t = 1, ..., n_{\text{critic}}$ **do**
    for $i = 1, ..., m$ **do**
      Sample real data $\boldsymbol{x} \sim \mathbb{P}_r$, latent variable $\boldsymbol{z} \sim p(\boldsymbol{z})$, a random number $\epsilon \sim U[0, 1]$.
      $\tilde{\boldsymbol{x}} \leftarrow G_\theta(\boldsymbol{z})$
      $\hat{\boldsymbol{x}} \leftarrow \epsilon \boldsymbol{x} + (1 - \epsilon)\tilde{\boldsymbol{x}}$
      $L^{(i)} \leftarrow D_w(\tilde{\boldsymbol{x}}) - D_w(\boldsymbol{x}) + \lambda(\|\nabla_{\hat{\boldsymbol{x}}} D_w(\hat{\boldsymbol{x}})\|_2 - 1)^2$
    end for
    $w \leftarrow \text{Adam}(\nabla_w \frac{1}{m} \sum_{i=1}^m L^{(i)}, w, \alpha, \beta_1, \beta_2)$
  end for
  Sample a batch of latent variables $\{\boldsymbol{z}^{(i)}\}_{i=1}^m \sim p(\boldsymbol{z})$.
  $\theta \leftarrow \text{Adam}(\nabla_\theta \frac{1}{m} \sum_{i=1}^m -D_w(G_\theta(\boldsymbol{z})), \theta, \alpha, \beta_1, \beta_2)$
end while

(3)

2.1 Attention in Machine Learning

It is the ability to dynamically highlight and use the salient parts of the information at hand, in a similar manner as it does in the human brain, this makes attention an attractive concept in machine learning.

An attention-based system is thought to consist of three components(2):

- A process that "reads" raw data (such as source words in a source sentence), and converts them into distributed representations, with one feature vector associated with each word position.
- A list of feature vectors storing the output of the reader. This can be understood as a "memory" containing a sequence of facts, which can be retrieved later, not necessarily in the same order, without having to visit all of them.
- A process that "exploits" the content of the memory to sequentially perform a task, at each time step having the ability put attention on the content of one memory element (or a few, with a different weight).

In SAGAN, details can be generated using cues from all feature locations. Moreover, the discriminator can check that highly detailed features in distant portions of the image are consistent with each other. Furthermore, recent work has shown that generator conditioning affects GAN performance. Leveraging this insight, we apply spectral normalization to the GAN generator and discriminator networks and find that this improves training dynamics. The self-attention module calculates response at a position as a weighted sum of the features at all positions, where the weights or attention vectors are calculated with only a small computational cost. DCGANs are unable to capture long range dependencies in images. They work well for images that do not contain a lot of structural and geometric information. They fail to represent global relationships. These non-local dependencies consistently appear in certain classes of images. For example, GANs can draw animal images with realistic fur, but often fail to draw separate feet.

In SAGAN, the self-attention module works in conjunction with the convolution network and uses the key-value-query model. This module takes the feature map, created by the convolutional layer, and transforms it into three feature spaces. These feature spaces, called key f(x), value h(x), and query g(x), are created by passing the original feature map through three different 1 x 1 convolution maps. Key f(x) and query g(x) matrices are then multiplied. Next, the softmax operation is applied on each row of the multiplication result. The attention map generated from softmax identifies which areas of the image the network should attend to.

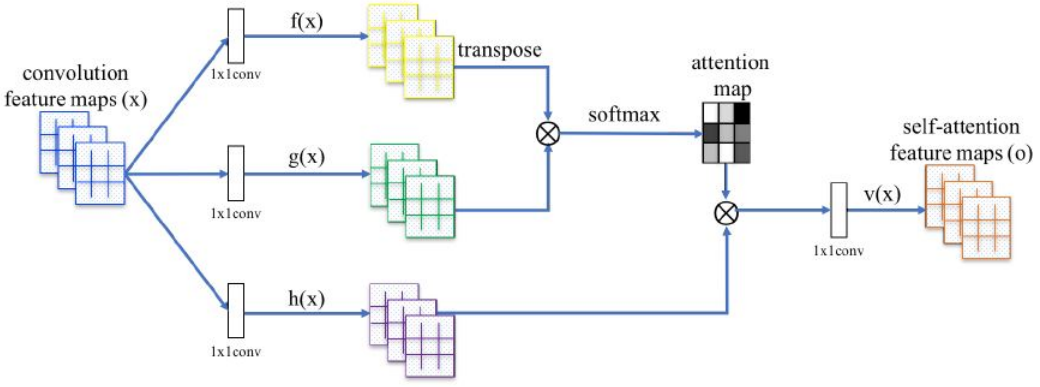


Figure 7: Self Attention module for SAGAN. The x denotes matrix multiplication. The softmax operation is performed on each row.(12)

The image features from the previous hidden layer $x \in \mathbb{R}^{C \times N}$ are first transformed into two feature spaces f, g to calculate the attention, where
$f(x) = W_f x, \qquad g(x) = W_g x$
$\beta_{j,i} = \frac{\exp(s_{ij})}{\sum_{i=1}^{N} \exp(s_{ij})}$,
where $s_{ij} = f(x_i)^T g(x_j)$ and $\beta_{j,i}$ indicates the extent to which the model attends to the i^{th} location when synthesizing the j^{th} region. Here, C is the number of channels and N is the number of feature locations of features from the previous hidden layer. The output of the attention layer is $o = (o_1, o_2, \ldots, o_j, \ldots, o_N) \in \mathbb{R}^{C \times N}$,
where, $\quad o_j = v\left(\sum_{i=1}^{N} \beta_{j,i} h(x_i)\right), \quad h(x_i) = W_h x_i, \quad v(x_i) = W_v x_i$
In the above formulation, $W_g \in \mathbb{R}^{\bar{C} \times C}, W_f \in \mathbb{R}^{\bar{C} \times C}$, $W_h \in \mathbb{R}^{\bar{C} \times C}$, and $W_v \in \mathbb{R}^{C \times \bar{C}}$ are the learned weight matrices, which are implemented as 1×1 convolutions.
$\bar{C} = {}^{C}/_{8}$

In addition, we further multiply the output of the attention layer by a scale parameter and add back the input feature map. Therefore, the final output is given by,

$$y_i = \gamma o_i + x_i$$

where γ is a learnable scalar and it is initialized as 0. Introducing the learnable γ allows the network to first rely on the cues in the local neighborhood since this is easier and then gradually learn to assign more weight to the non-local evidence. Self attention module is applied to both the generator and discriminator networks. We use Two Timescale Update Rule(TTUR) specifically to compensate for the problem of slow learning in a regularized discriminator, making it possible to use fewer discriminator steps per generator step.. The hyperparmeters used for the SAGAN model are as

mentioned in the SAGAN paper.(12)

2.2 Fretchet Inception Distance(FID)

FID metric was used for evaluating the performance of the two models.(4) The calculation can be divided into three parts:

1. We use the Inception network to extract 2048-dimensional activations from the pool3 layer for real and generated samples respectively.
2. Then we model the data distribution for these features using a multivariate Gaussian distribution with mean μ and covariance Σ.
3. Finally Wasserstein-2 distance is calculated for the mean and covariance of real images(x) and generated images (g).

$$FID(x, g) = \|\mu_x - \mu_g\|_2^2 + Tr\left(\Sigma_x + \Sigma_g - 2\left(\Sigma_x \Sigma_g\right)^{\frac{1}{2}}\right)$$

Lower FID means better quality and diversity. The implementation for FID was taken from(10)

2.3 Spectral Normalization

(7) The matrix norm or spectral norm is defined as:

$$\|A\| = \underbrace{\max}_{norm} \frac{\|Ax\|}{\|x\|}$$

Conceptually, it measures how far the matrix A can stretch a vector x.The spectral norm of a matrix A is the maximum singular value of matrix A.

Let's construct a deep network with the following formulation.

$$f(x, \theta) = W^{L+1} a_L\left(W^L\left(a_{L-1}\left(W^{L-1}\left(\ldots a_1\left(W^1 x\right)\ldots\right)\right)\right)\right)$$

For each layer L, it outputs $y = Ax$ followed by an activation function a, say ReLU. The Lipschitz constant (or Lipschitz norm) of function g can be computed with its derivative as:

$$\|g\|_{Lip} = \sup_h \sigma(\nabla g(h))$$

The Lipschitz constant for $g = Ax$ is the spectral norm of W.

$$\|g\|_{Lip} = \sup_h \sigma(\nabla g(h)) = \sup_h \sigma(W) = \sigma(W)\nabla g(h) = \nabla W h = W$$

The Lipschitz constant for RELU is 1 as its maximum slope equals one. Therefore the Lipschitz constant for the whole deep network f is

$$\|f\|_{Lip} \leq \left\|\left(h_L \mapsto W^{L+1}h_L\right)\right\|_{Lip} \cdot \|a_L\|_{Lip} \cdot \left\|\left(h_{L-1} \mapsto W^L h_{L-1}\right)\right\|_{Lip}$$

$$\cdots \|a_1\|_{Lip} \cdot \left\|\left(h_0 \mapsto W^1 h_0\right)\right\|_{Lip} = \prod_{l=1}^{L+1} \left\|\left(h_{l-1} \mapsto W^l h_{l-1}\right)\right\|_{Lip} = \prod_{l=1}^{L+1} \sigma\left(W^l\right)$$

Spectral normalization normalizes the weight for each layer with the spectral norm $\sigma(W)$ such that the Lipschitz constant for each layer as well as the whole network equals one.

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

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

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

With Spectral Normalization, we can renormalize the weight whenever it is updated. This creates a network that mitigates gradient explosion problems.

3 Solution description

3.1 DCGAN

The generator of the DCGAN was implemented as shown in the figure. It takes as input a 100 dimensional noise vector. It then performs a series of transpose convolution followed by batch normalization and passes the feature maps through ReLU activation function and repeatedly upscales the feature maps to finally generate a 64 X 64 X 3 image. It finally passes the image through the tanh activation function so that the pixel values are in the [-1,1] range.

```
Generator(
  (g): Sequential(
    (0): Sequential(
      (0): ConvTranspose2d(100, 1024, kernel_size=(4, 4), stride=(1, 1), bias=False)
      (1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU()
    )
    (1): Sequential(
      (0): ConvTranspose2d(1024, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU()
    )
    (2): Sequential(
      (0): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
      (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU()
    )
    (3): Sequential(
      (0): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
      (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU()
    )
    (4): ConvTranspose2d(128, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (5): Tanh()
  )
)
```

Figure 8: DCGAN generator

```
Discriminator(
  (d): Sequential(
    (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (1): LeakyReLU(negative_slope=0.2)
    (2): Sequential(
      (0): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
      (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): LeakyReLU(negative_slope=0.2)
    )
    (3): Sequential(
      (0): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
      (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): LeakyReLU(negative_slope=0.2)
    )
    (4): Sequential(
      (0): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): LeakyReLU(negative_slope=0.2)
    )
    (5): Conv2d(512, 1, kernel_size=(4, 4), stride=(2, 2))
    (6): Sigmoid()
  )
)
```

Figure 9: DCGAN discriminator

The discriminator of the DCGAN was implemented as shown in the figure. It takes as input a 64 X 64 X 3 with pixel values in the range[-1,1]. It then performs a series of convolution followed by batch

normalization and passes the feature maps through Leaky ReLU activation function and repeatedly downscales the feature maps to finally generate a 1 X 1 X 1 feature map. It finally passes the image through the sigmoid activation function that outputs a value between [0,1] indicating the probability of the input image being real.

3.2 SAGAN

We first make the generator and discriminator implemented for the DCGAN conditional so that both of these networks also take the label of the image as input. The generator of the SAGAN was implemented as shown in the figure. It takes as input a 100 dimensional noise vector. It then performs a series of transpose convolution followed by batch normalization and passes the feature maps through ReLU activation function and repeatedly upscales the feature maps to finally generate a 64 X 64 X 3 image. The Self Attention mechanism was implemented at 32 X 32 feature map size. It finally passes the image through the tanh activation function so that the pixel values are in the [-1,1] range.

```
Generator(
  (gen): Sequential(
    (0): Sequential(
      (0): ConvTranspose2d(200, 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): ReLU()
    )
    (1): Sequential(
      (0): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
      (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU()
    )
    (2): Sequential(
      (0): 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): ReLU()
    )
    (3): Self_Attention_Layer(
      (snconv1x1_theta): Conv2d(64, 8, kernel_size=(1, 1), stride=(1, 1))
      (snconv1x1_phi): Conv2d(64, 8, kernel_size=(1, 1), stride=(1, 1))
      (snconv1x1_g): Conv2d(64, 32, kernel_size=(1, 1), stride=(1, 1))
      (snconv1x1_attn): Conv2d(32, 64, kernel_size=(1, 1), stride=(1, 1))
      (maxpool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
      (softmax): Softmax(dim=-1)
    )
    (4): Sequential(
      (0): ConvTranspose2d(64, 32, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
      (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU()
    )
    (5): ConvTranspose2d(32, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (6): Tanh()
  )
  (embed): Embedding(10, 100)
)
```

Figure 10: SAGAN generator

The discriminator of the SAGAN was implemented as shown in the figure. It takes as input a 64 X 64 X 3 with pixel values in the range[-1,1]. It then performs a series of convolution followed by instance normalization and passes the feature maps through Leaky ReLU activation function and repeatedly downscales the feature maps to finally generate a 1 X 1 X 1 feature map.The Self Attention

mechanism was implemented at 32 X 32 feature map size. Spectral normalization was used in all the layers of the generator and discriminator.

```
Discriminator(
  (disc): Sequential(
    (0): Conv2d(4, 16, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (1): LeakyReLU(negative_slope=0.2)
    (2): Sequential(
      (0): Conv2d(16, 32, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
      (1): InstanceNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=False)
      (2): LeakyReLU(negative_slope=0.2)
    )
    (3): Self_Attention_Layer(
      (snconv1x1_theta): Conv2d(32, 4, kernel_size=(1, 1), stride=(1, 1))
      (snconv1x1_phi): Conv2d(32, 4, kernel_size=(1, 1), stride=(1, 1))
      (snconv1x1_g): Conv2d(32, 16, kernel_size=(1, 1), stride=(1, 1))
      (snconv1x1_attn): Conv2d(16, 32, kernel_size=(1, 1), stride=(1, 1))
      (maxpool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
      (softmax): Softmax(dim=-1)
    )
    (4): Sequential(
      (0): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
      (1): InstanceNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=False)
      (2): LeakyReLU(negative_slope=0.2)
    )
    (5): Sequential(
      (0): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
      (1): InstanceNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=False)
      (2): LeakyReLU(negative_slope=0.2)
    )
    (6): Conv2d(128, 1, kernel_size=(4, 4), stride=(2, 2))
  )
  (embed): Embedding(10, 4096)
)
```

Figure 11: SAGAN discriminator

4 Empirical Methodology

4.1 DCGAN

The DCGAN model was designed to generate 64 x 64 images. No pre-processing was applied to the training images besides scaling to the range of the tanh activation function [-1, 1]. DCGAN was trained with mini-batch stochastic gradient descent (SGD) with a mini-batch size of 128. All weights were initialized from a zero-centered Normal distribution with standard deviation 0.02. In the LeakyReLU, the slope of the leak was set to 0.2. Adam optimizer was used with beta1 = 0.5 and beta2 = 0.999. The generator and discriminator learning rates were set to be 0.0001 and 0.0004 respectively. The dimension of the noise vector was set to 100. Binary Cross Entropy Loss function was used for training the discriminator and generator networks. The training procedure is as described in figure.

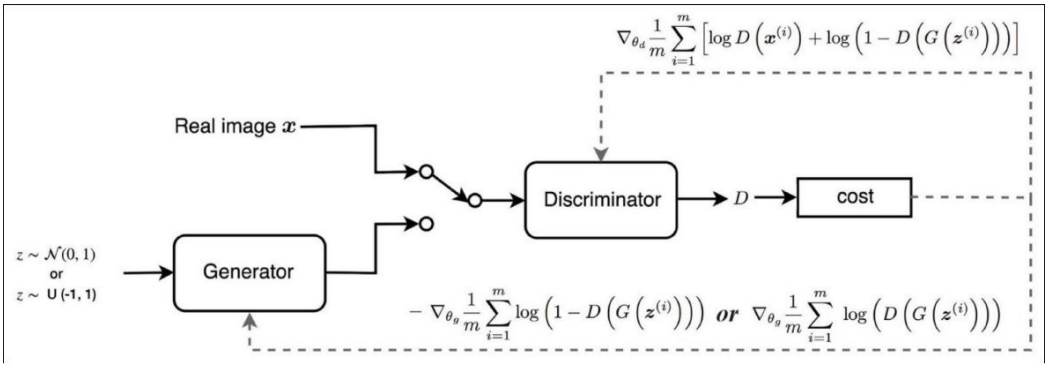


Figure 12: GAN architecture(11)

4.2 SAGAN

The SAGAN model was designed to generate 64 x 64 images. Spectral Normalization is used for the layers in both the generator and discriminator. I have used Instance Normalization in the layers of the discriminator and Batch Normalization for the layers of the generator. I have used the Adam optimizer with beta1 = 0 and beta2 = 0:9 for training. The Self Attention mechanism was implemented at 32 X 32 feature map size. All weights were initialized from a zero-centered Normal distribution with standard deviation 0.02. In the LeakyReLU, the slope of the leak was set to 0.2. SAGAN was trained with mini-batch stochastic gradient descent (SGD) with a mini-batch size of 64. I have used separate learning rates (TTUR)(4) for the generator and the discriminator. The learning rate for the discriminator was set to 0.0004 and the learning rate for the generator was set to 0.0001. The SAGAN paper suggested doing so to compensate for the problem of slow learning in a regularized discriminator, making it possible to use fewer discriminator steps per generator step. Wasserstein loss with gradient penalty was used for training the generator and discriminator networks. The lambda parameter was set to 10 for gradient penalty. No. of critic iterations per generator iteration was set to 5. The generator embedding size and dimension of the noise vector was set to 100. I passed random labels to the generator along with the noise vector. The generator concatenates these two inputs and generates images. While training the discriminator, we pass the generated fake images with their labels to the discriminator. We also train the discriminator on real images with their labels. The training procedure of Wasserstein GAN is described in the figure.

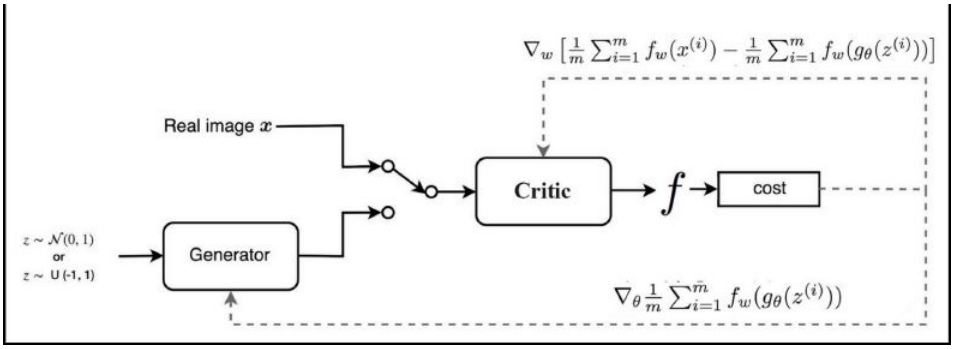


Figure 13: WGAN architecture(11)

5 Results and Discussion

5.1 DCGAN

I trained the DCGAN model for 87500 iterations. The plots of the generator loss, discriminator loss and FID scores achieved are presented below. The best FID score achieved was 238.4646. The corresponding images generated by the generator is also shown here. After every 100 iterations the

generator and discriminator losses was printed. Every 500 iterations, the FID score was computed. The FID scores, generator and discriminator losses were saved to a csv file for plotting later. After every 500 iterations, a 64 X 64 grid of fake images generated by the generator was saved to the DCGAN output directory. Every 5 epochs the checkpoint was saved for the generator and discriminator models.

In the DCGAN we face the problem of Mode Collapse in which the generator tries to fool the discriminator by generating images that are very similar. We see that the FID scores decrease slowly on average as we train for more number of iterations.

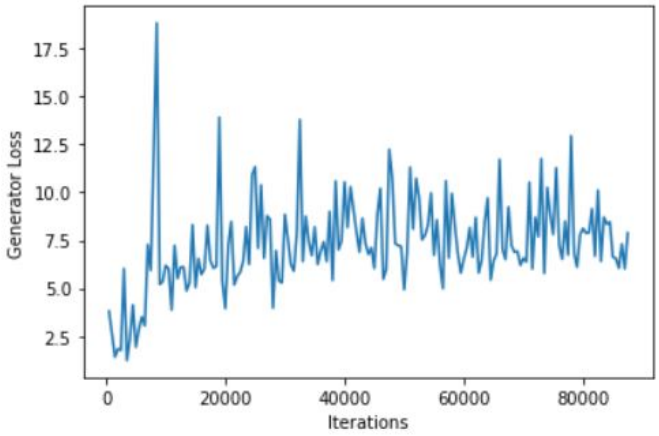


Figure 14: DCGAN Generator Loss

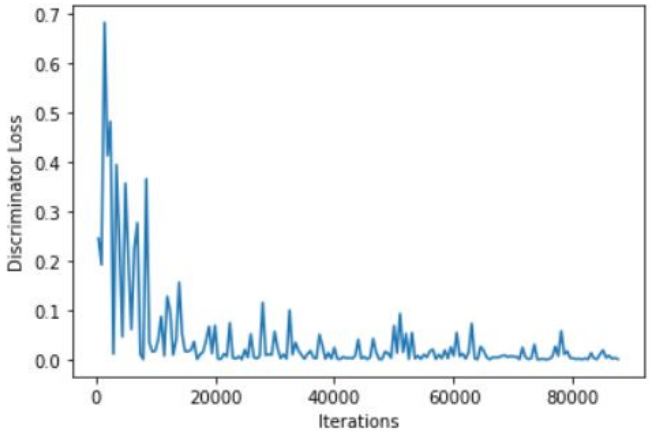


Figure 15: DCGAN Discriminator Loss

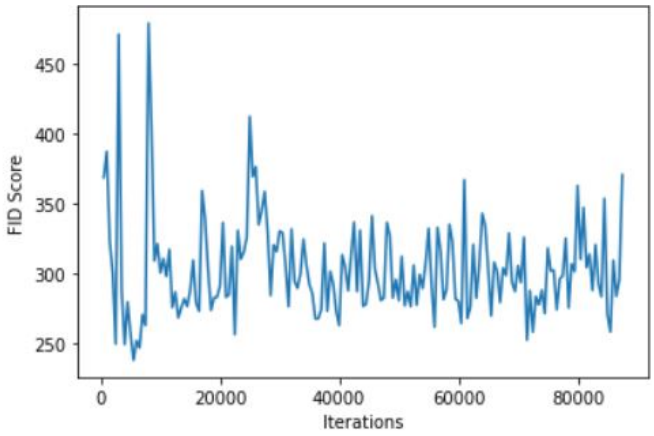


Figure 16: DCGAN FID Score

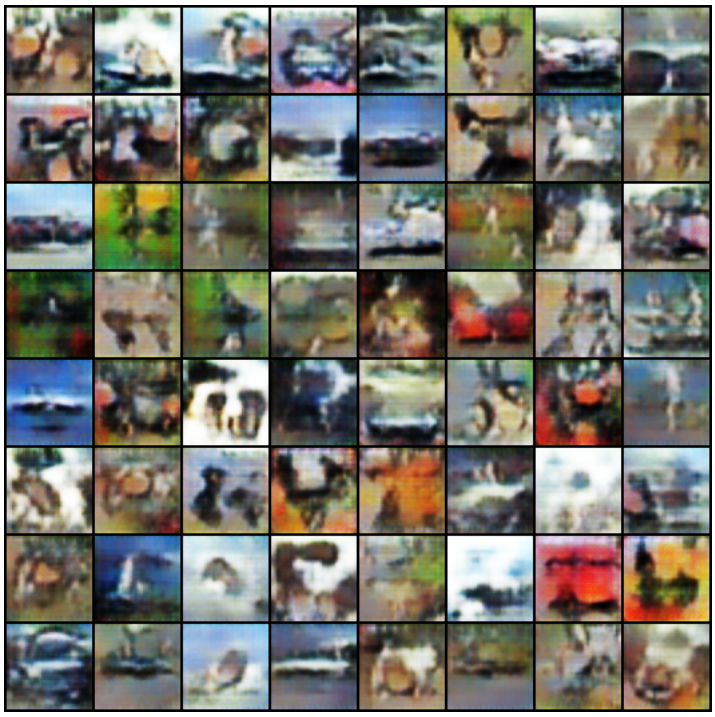


Figure 17: fake images generated by the DCGAN generator

5.2 SAGAN

I trained the SAGAN model for 114000 iterations. The plots of the generator loss, discriminator loss and FID scores achieved are presented below. The best FID score achieved was 229.8714. The corresponding images generated by the generator is also shown here. After every 100 iterations the generator and discriminator losses was printed. Every 500 iterations, the FID score was computed. The FID scores, generator and discriminator losses were saved to a csv file for plotting later. After every 500 iterations, a 64 X 64 grid of fake images generated by the generator was saved to the SAGAN output directory. Every 5 epochs the checkpoint was saved for the generator and discriminator models.

In the SAGAN, for some reason after training for a considerable number of iterations, I found that the images generated by the generator was not improving over time and the FID score was not decreasing.

The generator and discriminator losses were becoming negative and the model did not seem to learn over time. Also the training of SAGAN was found to be very slow.

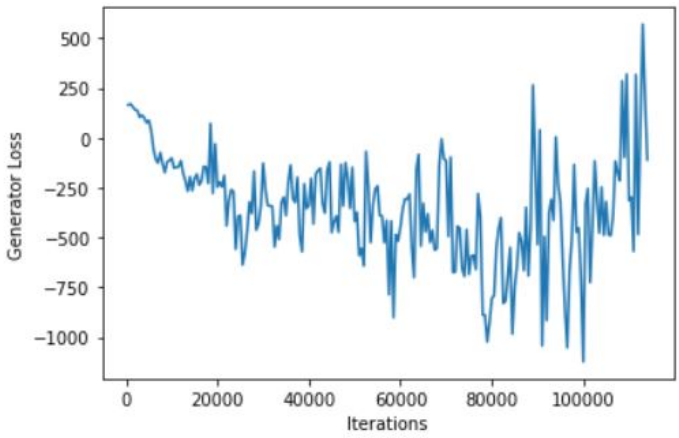


Figure 18: SAGAN Generator Loss

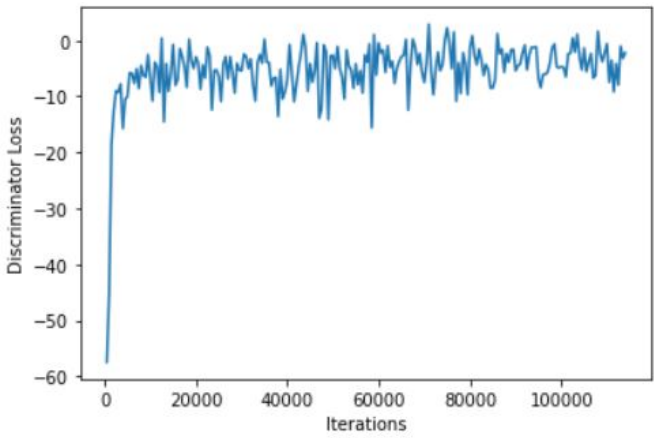


Figure 19: SAGAN Discriminator Loss

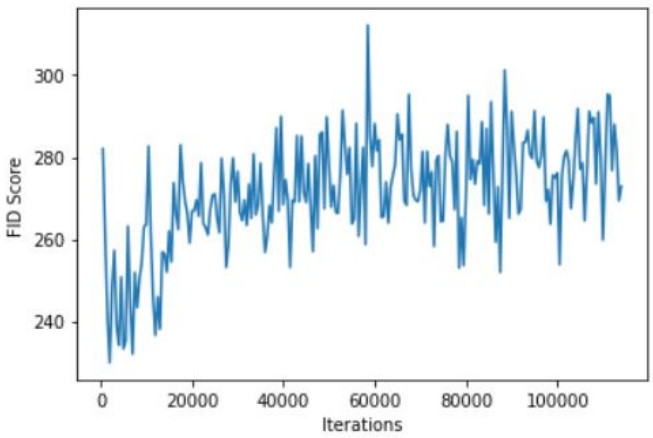


Figure 20: SAGAN FID Score

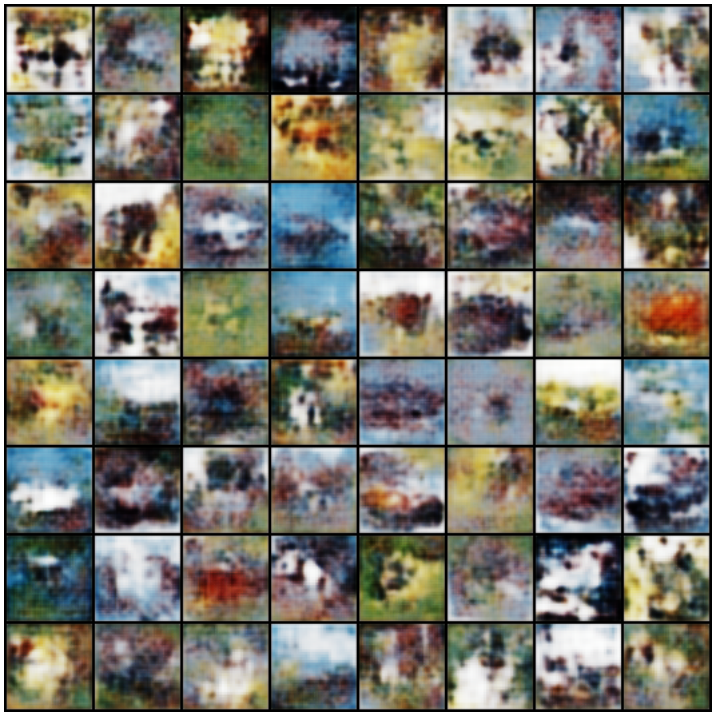


Figure 21: fake images generated by the SAGAN generator

References

[1] Arjovsky, M., Chintala, S., Bottou, L.: Wasserstein generative adversarial networks. In: Precup, D., Teh, Y.W. (eds.) Proceedings of the 34th International Conference on Machine Learning. Proceedings of Machine Learning Research, vol. 70, pp. 214–223. PMLR (06–11 Aug 2017), https://proceedings.mlr.press/v70/arjovsky17a.html

[2] Goodfellow, I.J., Bengio, Y., Courville, A.: Deep Learning. MIT Press, Cambridge, MA, USA (2016), http://www.deeplearningbook.org

[3] Gulrajani, I., Ahmed, F., Arjovsky, M., Dumoulin, V., Courville, A.C.: Improved training of wasserstein gans. In: Guyon, I., Luxburg, U.V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., Garnett, R. (eds.) Advances in Neural Information Processing Systems. vol. 30. Curran Associates, Inc. (2017), https://proceedings.neurips.cc/paper/2017/file/892c3b1c6dccd52936e27cbd0ff683d6-Paper.pdf

[4] Heusel, M., Ramsauer, H., Unterthiner, T., Nessler, B., Hochreiter, S.: Gans trained by a two time-scale update rule converge to a local nash equilibrium. Advances in neural information processing systems 30 (2017), https://proceedings.neurips.cc/paper/2017/hash/8a1d694707eb0fefe65871369074926d-Abstract.html

[5] Hui, J.: GAN - Spectral Normalization. https://jonathan-hui.medium.com/gan-spectral-normalization-893b6a4e8f53 (2020), Medium Blog

[6] Irpan, A.: Wasserstein GAN. https://www.alexirpan.com/2017/02/22/wasserstein-gan.html, (February 2017), Sorta Insightful

[7] Miyato, T., Kataoka, T., Koyama, M., Yoshida, Y.: Spectral normalization for generative adversarial networks. arXiv preprint arXiv:1802.05957 (2018), https://doi.org/10.48550/arXiv.1802.05957

[8] Persson, A.: Generative Adversarial Networks(GANs) playlist. https://www.youtube.com/playlist?list=PLhhyoLH6IjfwIp8bZnzX8QR30TRcHO8Va (2020), Youtube

[9] Radford, A., Metz, L., Chintala, S.: Unsupervised representation learning with deep convolutional generative adversarial networks (2015), https://arxiv.org/abs/1511.06434

[10] Seitzer, M.: pytorch-fid: FID Score for PyTorch. https://github.com/mseitzer/pytorch-fid (August 2020), version 0.2.1

[11] Srihari, S.N.: Lecture notes on GANs. https://cedar.buffalo.edu/~srihari/CSE676/index.html, (Spring 2020), Deep Learning Course Lectures: 22.1-22.5

[12] Zhang, H., Goodfellow, I., Metaxas, D., Odena, A.: Self-attention generative adversarial networks. In: Chaudhuri, K., Salakhutdinov, R. (eds.) Proceedings of the 36th International Conference on Machine Learning. Proceedings of Machine Learning Research, vol. 97, pp. 7354–7363. PMLR (09–15 Jun 2019), https://proceedings.mlr.press/v97/zhang19d.html