CSE 676 : Deep Learning

Project 1 : DCGAN and SAGAN Report Fall 2019

Karthik Vikram Kuduva Gopal Kabirdoss
UB ID: 50290281

Brief Introduction and Problem Statement

GANs(Generative Adversarial Networks), they’ve been hype during the last few years. These algorithms are unsupervised algorithms, which aims to learn the underlying structure of the given data, without specifying a target value. Generative models learn the intrinsic distribution function of the input data, allowing them to generate both synthetic inputs and outputs/targets.

This problem description is about training two generative models to generate new images from samples.I will be implementing two types of Generative Adversarial Networks (GANs): One known as the Deep Convolution GAN (DCGAN) and Self-Attention GAN (SA-GAN) using CIFAR-10 dataset.

The basic DCGAN consists of two neural networks: a discriminator (D) and a generator (G) which will compete with each other, making each other stronger at the same time. One of the simple variants of GAN in Deep Convolutional GANs (DCGANs) in which G and D are both based on deep convolutional neural networks.

There are a few setbacks in the DCGANs. Since DCGANs use only convolutional layers which has a local receptive field, long range dependencies can be only processed after passing through several layers. In order to learn the long term dependencies, we might need to increase the number of convolutional layers, which might increase the representational capacity of the model thus making the training slower and mode collapses may occur.

In order to overcome the shortcomings of the DCGANs we will implement a SAGAN using self-attention layers and WGAN.

We will be using the very simple CIFAR10(Canadian Institute for Advanced Research) dataset for training the GAN models.

Dataset : CIFAR10

- The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes
- There are 6000 images per class.
- There are 50000 training images and 10000 test images.
- The classes of Images are airplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck

DCGAN(Deep Convolutional GAN)

Architecture

Like all other GANs, the DCGAN has 2 neural networks stacked one after the other, a Discriminator(D) and the Generator(G) network.

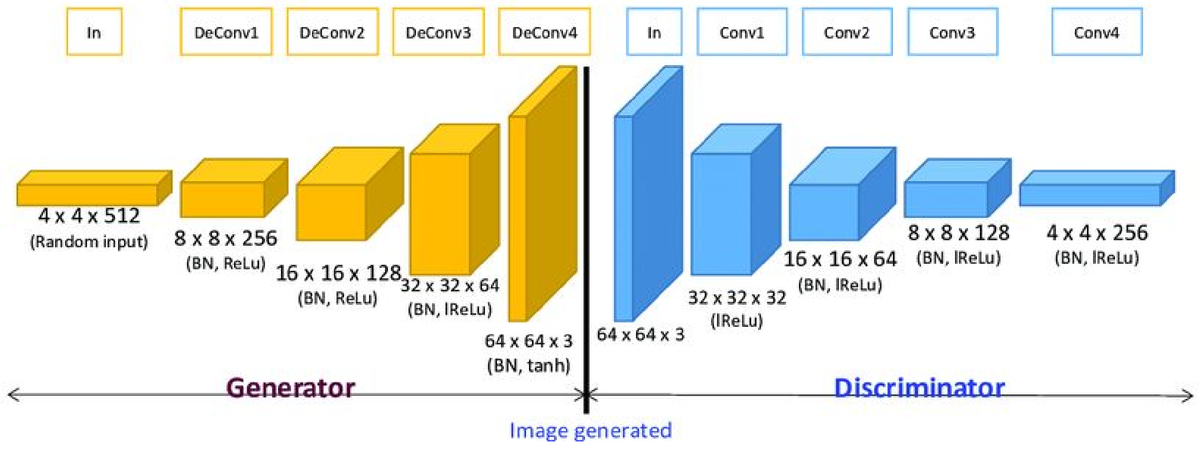

Fig: General structure of a DCGAN with generator and discriminator networks

Generator Model

- The input to this network is a random noise vector of the size of the latent vector dimension. The generator has a series of transpose convolution layers.

- There are 4 convolution2DTranspose layers(upsampling) which upscale or downscale the images extracting the feature maps. Batch normalization and ReLu is applied with the convolution2D layers.
- The last layer has tanh activation function which enables the output values in the image to be between -1 to 1 which aids in fast computation.
- The result is a 32x32x3 image in our case.

Discriminator Model

- The discriminator model has 4 convolutional 2D layers(downsampling).
- The first three layers are accompanied by batch normalization and leaky ReLu layers. While the last layer is a dense layer with sigmoid function.

The Generator and Discriminator nets were put together to form a GAN network.

Batch Normalization

Normalization of any data involves finding the mean and variance of the data and normalizing the data so that the data has 0 mean and unit variance. In batch normalization we will normalize each hidden unit activation. Consider we have d number of hidden units in a hidden layer of any Deep neural network. If the activation values of the layers is given as x=[x1,x2,………xd].
The formula for calculating hidden unit activation for Kth hidden unit is

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

Where,
E(x^k) is the expectation of the kth hidden units values/mean value
Var(x^k) is the variance of the kth hidden unit

Training of the DCGAN

- Initially the generator is not trained and is kept idle. While the discriminator is trained by passing the samples from training along with their ground truth labels.

- The discriminator model weights are updated and it learns from the training and provides a probability of the image given being a real image.
- Then the generator is given a random noise vector and is allowed to train based on the discriminators loss while keeping the discriminator idle.
- The generator is trained to cheat the discriminator. The discriminator is not updated during this operation but it provides the gradient information required to update the weights of the generator model. While the discriminator tries to outsmart the generator during the next iteration of the run. This happens like a cat and mouse chase.

  The loss function used for training is called the binary cross-entropy(BCE) which is given by the formula

$$V(D,G) = E_{p_{data}}[\log(D(x))] + E_{p_z}[\log(1 - D(G(z)))]$$

Evaluation Metrics for DCGAN: FID Score

- Most of the machine learning models use accuracy as the evaluation metrics. But for GANs we use the Fréchet Inception Distance or FID Score.
- FID Score in literal terms measures how close the distribution of the data in the generated sample is to the real images. We calculate the distance between the feature vectors of the real and fake images.
- For the calculations, we use the Inception V3 model which is used for image classification with a simple tweak.
- The Inception V3 model is fed with an image and the activation input of the layer before the last is collected and used for distance calculations.

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

- Lower FID score indicates poor performance of the generator. I.e. the generator has not learnt the data distribution good enough to replicate the training data accurately.

- A FID score around 300 -150 is considered as okay, while a score of 30 or less is considered state of the art for GANs which is good enough to cheat human eyes.

Graphs for DCGAN

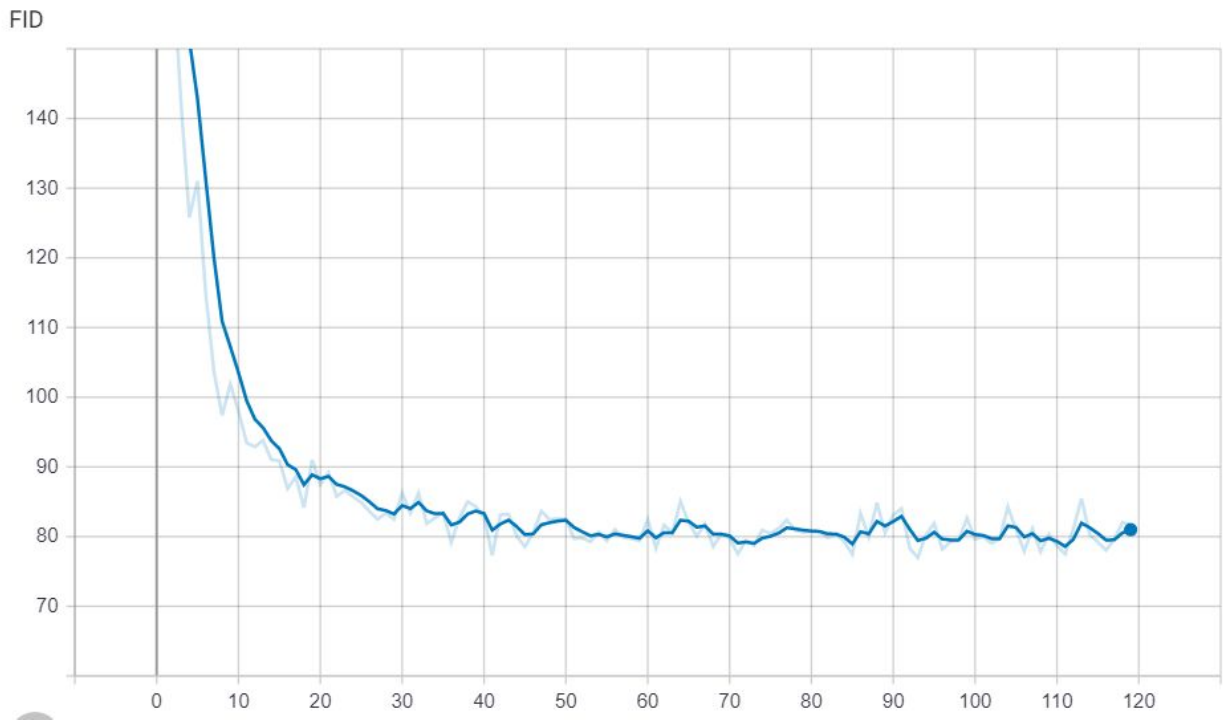

Fig: FID vs Epohs graphs for DCGANs

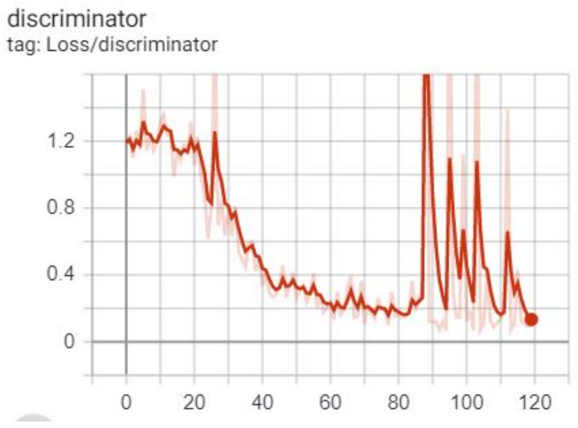

Fig: Discriminator Loss Vs Epochs

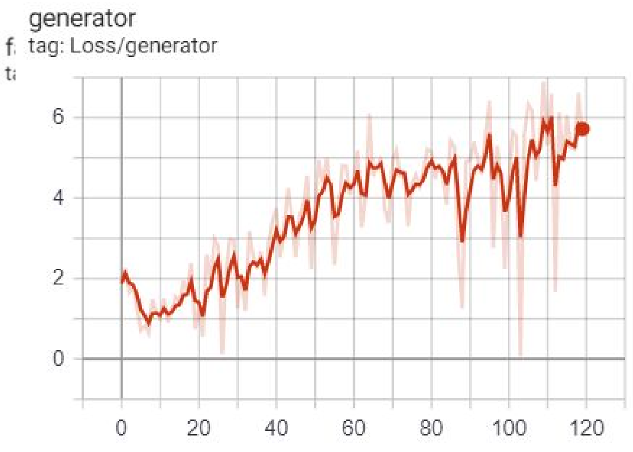

Fig: Generator Loss Vs Epochs

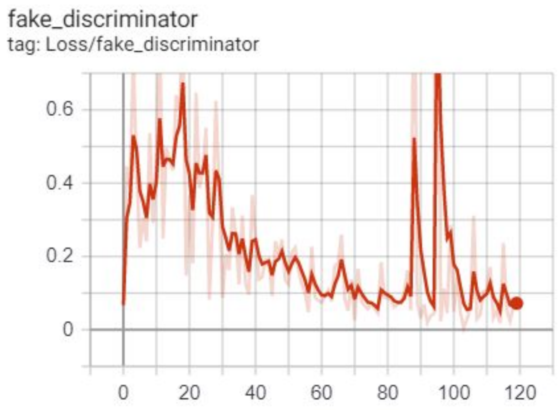

Fig: Fake Loss Vs Epochs

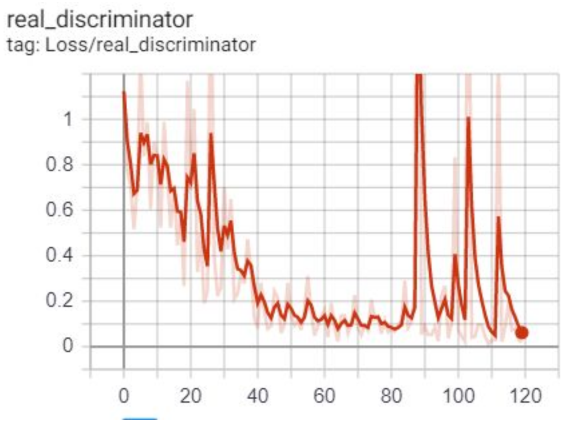

Fig: Real Loss Vs Epochs

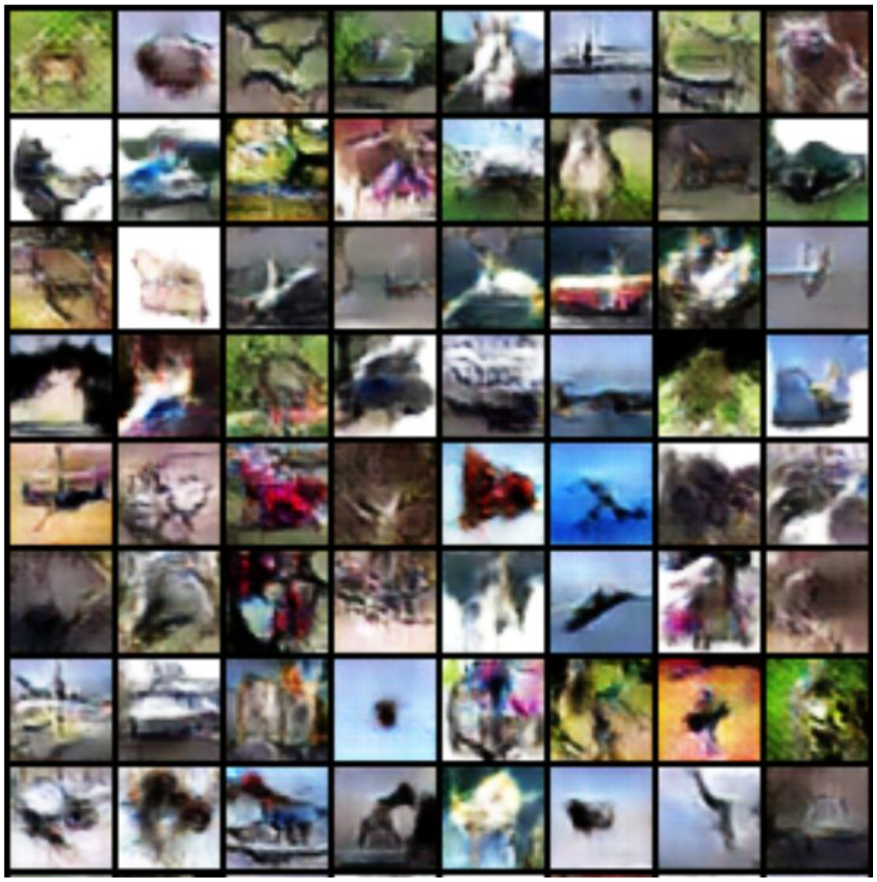

Fig: Fake Images Generated during epoch 5

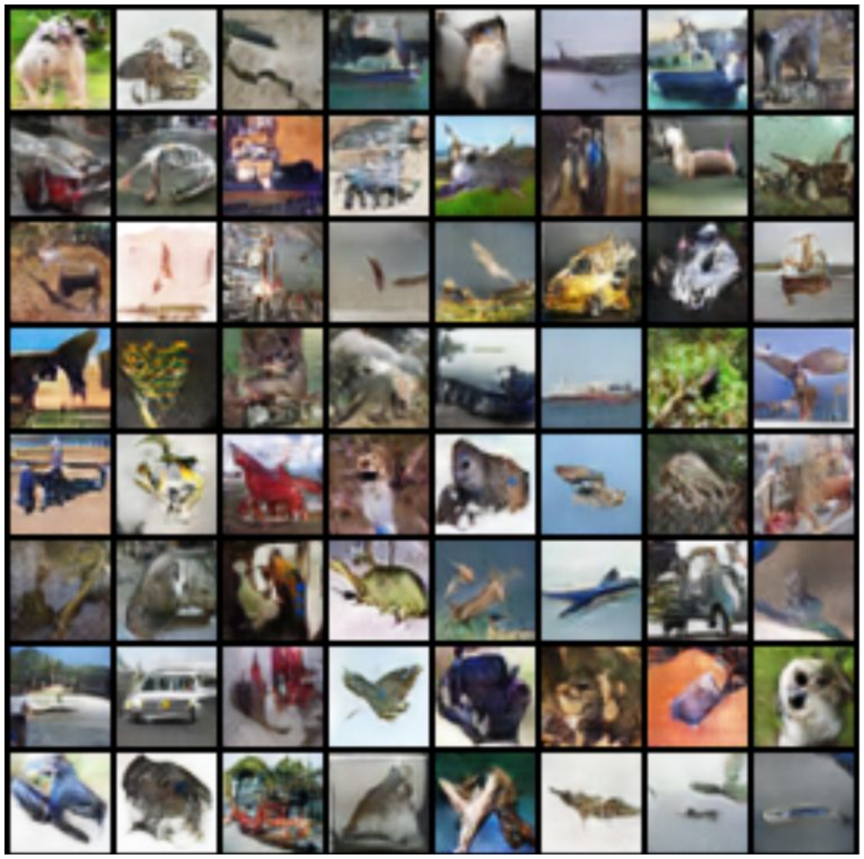

Fig: Fake Images generated after 120 epochs

FID Results

Epochs	FID Score
0	320
5	142
10	98
20	88
40	82
60	80
80	80
100	80
120	80

Table: Epochs Vs FID scores

SAGAN (Self Attention GAN)

What is attention and why is it beneficial for ML models?

- Traditional ML models usually use convolutional layers to extract features from images. But these layers concentrate on the local features only.
- To make the model look for more spatially scattered features we introduce the concept of attention which will enable models to look for related features that are spatially separated in a short span of time.
- Looking for scattered features using only convolutions is computationally expensive.

Self-Attention Layer

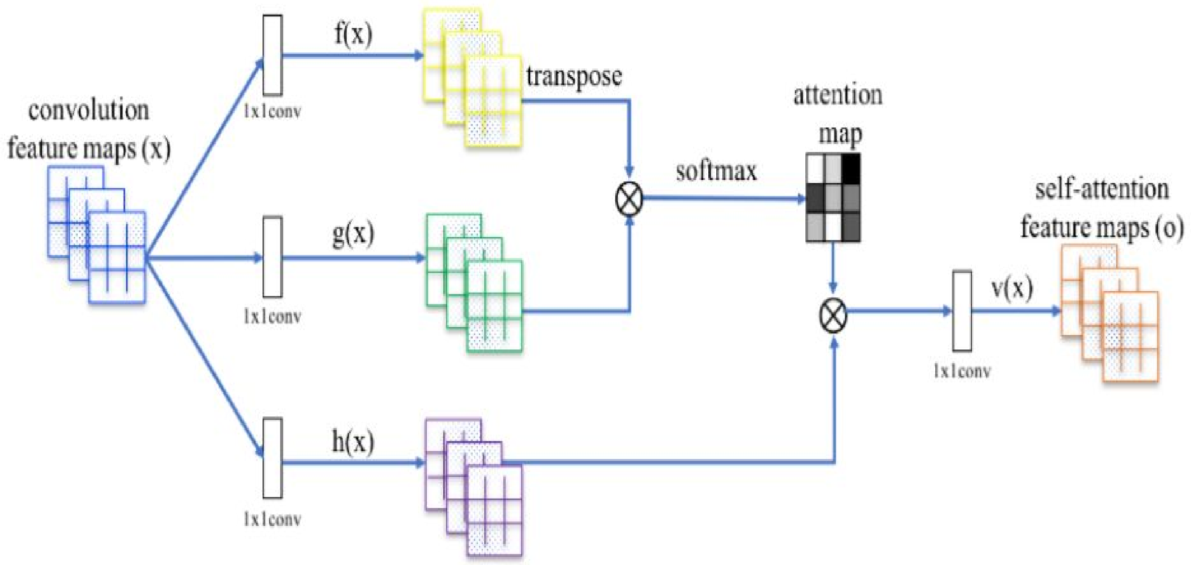

Fig: Architecture of the self attention layer

- Self attention is a layer that captures the long-range and multi-level dependencies in the image.
- Self attention layer allows the model to wider the gaze and look for widely separated features that are similar.

- Self attention layer produces an attention map in which each position is calculated as a weighted sum of features of all the positions.

Structure and Working

- Attention layer transforms the convolutional feature maps into three feature spaces [key f(x), value h(x), and query g(x)] by convolving through a 1x1 conv layer.
- Key f(x) is transposed and multiplied with query g(x) and then passed through a softmax function form an attention map.
- The attention map is then multiplied with the query h(x) and then convolved to form the self-attention feature maps of the same size.
- The scaling parameter gamma is initialized to 0 at first for the network to focus on the local areas in the image.
- During training gamma gets updated so that the network gradually learns to attend to the non-local areas of an image.

Spectral Normalization

- Spectral Normalization is a technique using which we normalize the spectral norm of the weights in each layer of the network so that it satisfies the K-Lipschitz continuity constraint. We use this to stabilize the training of the discriminator network.

$$W_{SN}(W) := W/\sigma(W)$$

- Spectral normalization regularizes the gradient of W, preventing over concentration of W in one direction.
- For every weight in our network,the vectors u and v are randomly initialized.For every iteration we only update need to perform an update on the current version of u and v which makes spectral normalization computationally efficient.

$$\sigma(W) = ||Wv|| = u^T Wv.$$

Weight clipping

- Weight clipping is used to satisfy the constraint of Lipschitz continuity.
- It avoids the problem of vanishing gradients by clipping the weights in all the layers within a particular range.

Wasserstein Loss

- The Wasserstein loss or the Earth-Mover distance is the cost of cheapest transport plan for moving data from distribution q to look like distribution p.
- It can also be defined as the amount of data that needs to be moved times the mean distance it has to be moved.
- 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.

The equation for calculating wasserstein loss is given by:

$$L(p_r, p_g) = W(p_r, p_g) = \max_{w \in W} \mathbb{E}_{x \sim p_r}[f_w(x)] - \mathbb{E}_{z \sim p_r(z)}[f_w(g_\theta(z))]$$

Conditional SAGAN

The conditional technique is introduced to ensure that the GAN is evenly trained on all classes of the images. CGAN will make the training faster, more stable and better. The CGAN can condition both the generator and discriminator with the class labels.

Implementation

- We can achieve even training by adding labels as one of the inputs to the GAN model.
- The generator is fed with a one hot vector of the image class, added with the latent noise vector.
- The one hot vector of classes generated is fed to the layer preceding the last layer in the discriminator.

- CGAN enables both generator and the discriminator to learn specific features of each label.

Architecture of SAGAN

Generator

- A functional model was constructed to accommodate the CGAN.
- The architecture of the generator is very similar to that of DCGAN except that we use spectral normalization instead of batch normalization at all layers. One hot vector is also fed to the generator along with the latent noise vector.
- The attention layer is added at the 16x16 level layer of the model.

Discriminator

- The structure of the discriminator is very similar to that of the generator with a few exceptions.
- The attention layer is added at the 16x16 level.
- The inverse labels +1 for fake images and -1 for real images so that we get small scores for fake images and large scores for real images.

Difference between SAGAN and DCGAN

- SAGAN Overcomes mode collapse problem by using Wasserstein loss function.
- Long-term dependencies are captured without losing computation and statistical efficiency without increasing the size of the convolutional filters.
- Introducing the conditional GAN increases in the SAGAN increases the class wise performance of both the generator and discriminator.
- SAGAN take longer time to train than DCGANs.
- The use of attention layers in the SAGAN must give images that make more sense that the ones produced by DCGAN. For Example: SAGAN produces better faces when compared to the DCGAN.

- SAGAN was able to train equally on all the classes of images due to conditional property.

Results of SAGAN

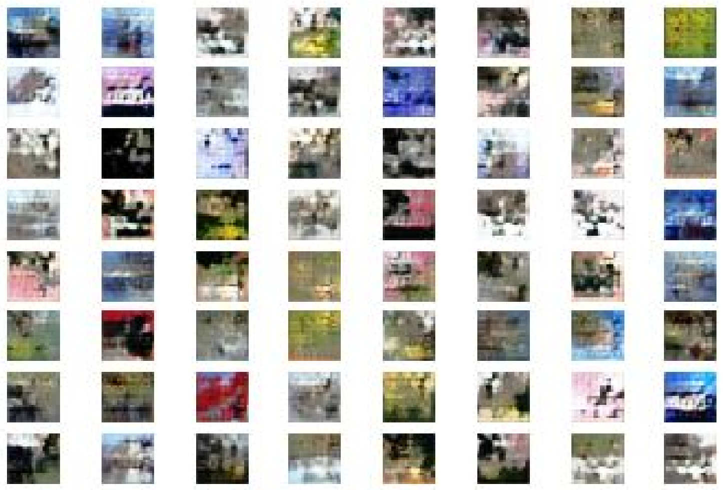

Fig: Fake image generated after 60 epochs

Fig: Fake Image generated after 110 epochs

FID vs Epochs Curve for the SAGAN

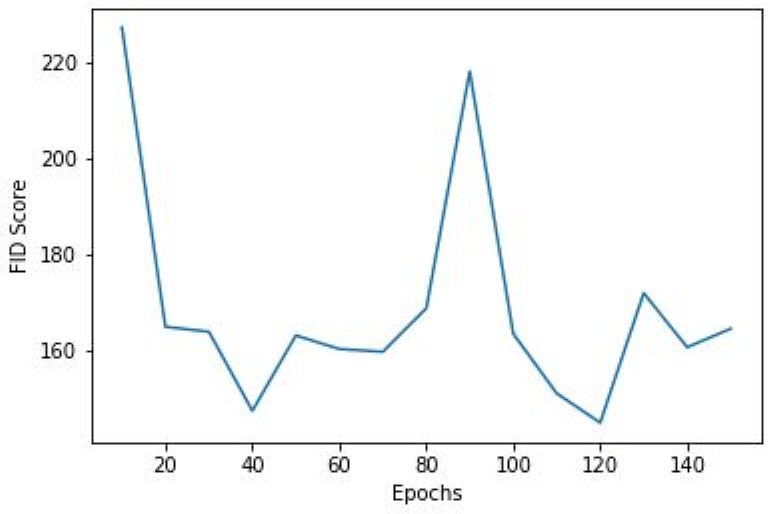

Fig: FID vs Epochs plot for SAGAN

References

1. https://github.com/pytorch/examples/tree/master/dcgan
2. https://christiancosgrove.com/blog/2018/01/04/spectral-normalization-explained.html
3. https://machinelearningmastery.com/how-to-implement-wasserstein-loss-for-generative-adversarial-networks/
4. https://medium.com/@sunnerli/the-story-about-wgan-784be5acd84c
5. https://developers.google.com/machine-learning/gan/loss
6. https://www.cs.toronto.edu/~kriz/cifar.html
7. https://lilianweng.github.io/lil-log/2017/08/20/from-GAN-to-WGAN.html