Creative Art Generation using Generative Adversarial Networks

Anand Kumar Maurya

930092

anand.k.maurya@student.fh-kiel.de

Master Project

MASTER OF SCIENCE

Information Engineering

Fachhochschule Kiel

Germany

July 1, 2019 to September 8, 2019

Table of Contents

Acknowledgement ……..……3

Abstract …….4

Introduction ……...……4

Generative Adversarial Networks Architecture …….5

Data set and Pre-proocessing …….6

Experiments ……6

Deep Convolution GAN Architecture …….6

Problems with DCGAN ……..7

Nearest Neighbor Interpolation ……..11

Wasserstein GAN ……...13

Self-Attention Generative Adversarial Networks ……..14

Attention Mechanism …….15

Understanding Attention Mechanism …….16

Controlling Lipschitz constant ……...17

Spectral Normalization ……..18

Exploring Singular Value Decomposition …….18

Layer Wise Spectral Normalization ……...18

Why Spectral Normalization …….19

Hinge Loss …… 21

SAGAN model, Discriminator ……..22

Self Attention Results ……23

Code Snippet ……..23

Conclusion …….24

References ……..25

Acknowledgement

I would like to thank my supervisor Prof. Dr. Hauke Schramm, for all the support and help provided to us to be able to complete my project. Furthermore, I would also like to thank you Mr. Gordon Böer and Mr. Alexander Oliver Mader for the advice and feedback during the project which helped in getting the right direction to carry on the project.

Abstract

Generative Adversarial Networks(GAN's) in the field of Art Generation has drawn a lot of attention because of it's artificially created art paintings which are a active source of interest for the painters worldwide. Variants of gan's such as Vanilla Gan, Deep Convolution Gan's, Wasserstein Gan have been used to study and improve the previous model. Self Attention Generative Adversarial Networks(SAGAN) which is used as the baseline model in BIG gan, which is the current state of art on ImageNet generation has produced the best results for the wiki art data set with a Frechet Inception Distance (FID) score of 62. This is the first time SAGAN has been used for the creative art generation.

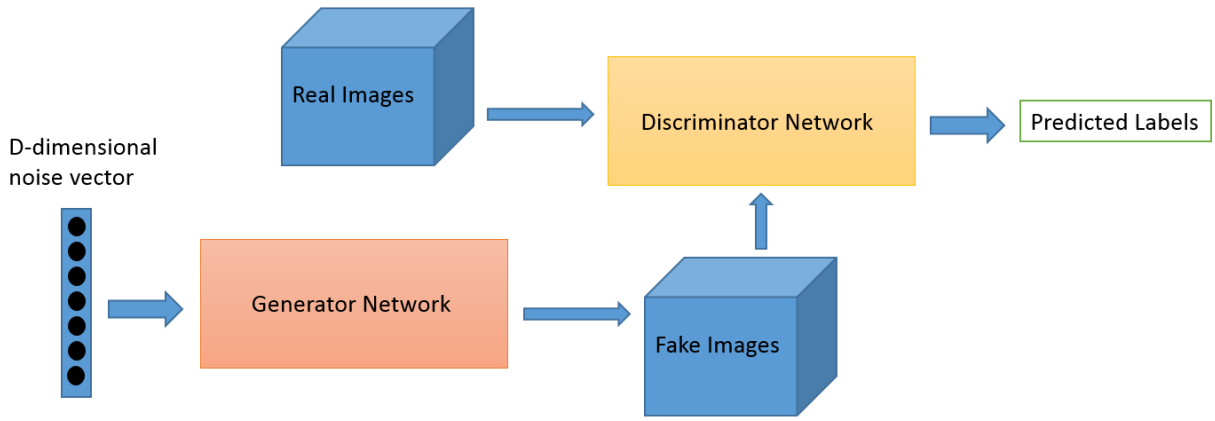
Introduction

Considering the applications of GAN's, there are ample of sufficient eye-catching projects. One of the interesting projects is Art generation. There hasn't been much work in the art generation but wherever it' been used, the promising results has gained a lot of attention. Vanilla GAN's were introduced by the author of GAN's Ian Goodfellow **[1]** which was used on MNIST digits dataset. The whole architecture consists of only Multi-layer perceptron. However, this architecture would be too simple to work on realistic images dataset as well as for paintings dataset for art generation. There was a another variant of GAN known as DCGAN **[2]** which was introduced a year later by Alec Radford and other fellow researchers. DCGAN**[2]** proved to perform very well on real images dataset as it involved Convolution operations which is actually performed on all images dataset. DCGAN is supposed to generate low-resolution images 64*64*3. The model on wikiart dataset faced problems of mode-collapse, checkerboard artifacts, image quality and it's contents. To solve the checkerboard artifacts problem we worked on some image resizing techniques switching from transposed convolution to Nearest neighbor interpolation. It successfully resolved this issue. We used Wasserstein GAN (WGAN's) to solve mode-collapse problem. This introduced much more restriction on weights of the networks than intended as it uses weight clipping technique **[3]**. While working on increasing image resolution, it improved the image quality to an extent but was not able to generate convincing results. The images weren't also very detailed enough. By detailed enough, I mean to say not able to produce high-level features. We switched to the current state of art GAN'S i.e. BIG GAN'S. While BIG gan's uses Self-Attention GAN'S **[4]** as a baseline model, so I first tried SAGAN on our art's data set. Techniques such as Attention maps and spectral normalization were able to solved the issues of image detailing and mode collapse respectively. We produced final

images of size 128*128*3 trained on Google colab with 112 hours of GPU training. The final images produced a Frechet Inception Distance (FID)**[13]** score of 62.

GAN Architecture

Gan's consists of two networks, Discriminator and the Generator. Both are adversarial to each other, hence the name Generative Adversarial Networks. Let's denote Discriminator and Generator by D & G respectively. Theoretically, both consist of either convolution or fully connected layers in case of vanilla gan. The task of the discriminator is to discriminate between real and fake images while on the other hand Generator's task is to produce images as close as to the data set. During the training process, they are locked in a minmax game each trying to oppose each other. They are locked in a fierce competition. G learns from D each time when it discriminate's real and fake images till to the point the model has reached an equilibrium known as Nash equilibrium where the Discriminator can't actually discriminate between real and fake images. G learns through D. Notice, the discriminator always uses the generator as a black box —i.e., never examines its internal parameters —whereas the generator needs the discriminator's parameters to compute its gradient direction. Also, the Generator never uses the real images directly for its computation but relies indirectly as it uses the parameters from the discriminator. Below is the diagrammatic representation for the GAN architecture.



source:- skymind.ai

Fig. 1 Generative Adversarial Networks Architecture

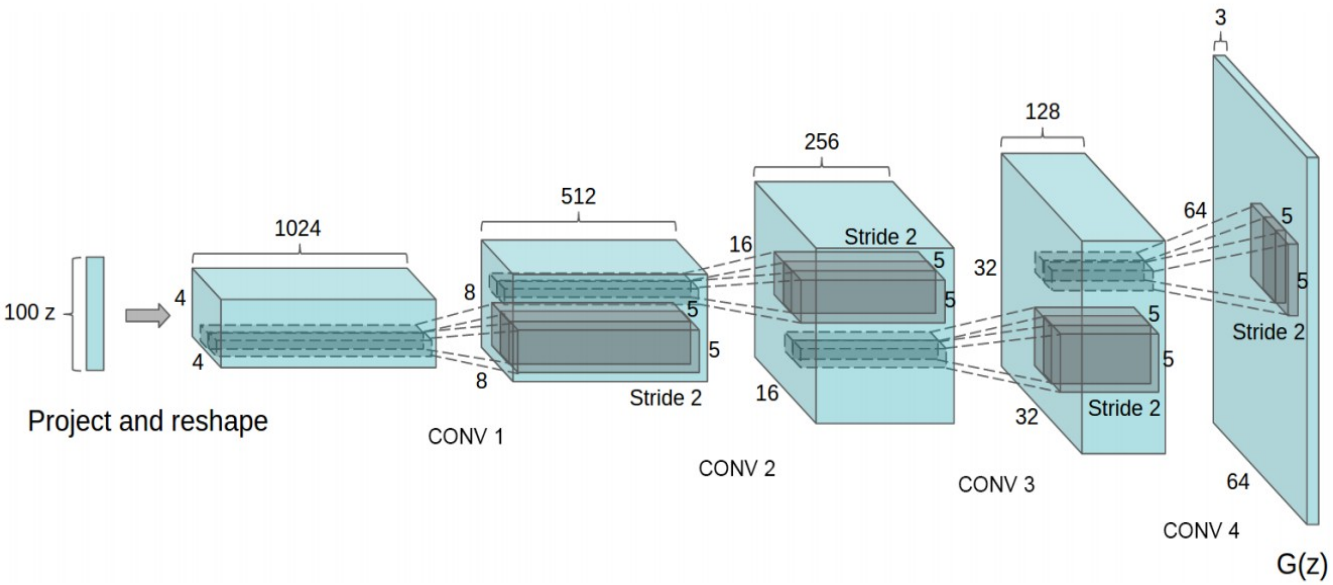
As a loss function, we prefer Binary cross-entropy as the loss function but it depends on user's choice as to which loss function one wants to apply. In the later models, there has been changes in the loss function being used which lead to better results. For optimization, Adam optimizer**[17]** performs the best for our model.

Data set and Pre-Processing

We have performed our experiments on the dataset provided by wikiart. We have performed our experiments on the dataset provided by wikiart. Though, not downloaded directly from wikiart, rather from kaggle. It consists of over 80,000 painting of different genres. We have narrowed down and just performed our experiments on just one genre, on landscape. In total, there were 9349 relevant images for our experiments. Some of the included paintings were grayscale. They were sorted and removed from the training data set using the .csv file provided by kaggle [5]. Every image was of different resolution and were resized to either 64*64*3 or 128*128*3 for training. All experiments were performed on Google colab Tesla K80 GPU. There is also code snippet of data preprocessing at the end.

Experiments

Deep convolution GAN's was a huge success for reproducing the realistic images. We began our experiments using the DCGAN's. The code has been taken over from github repository named 'carpedm20' from Taehoon Kim. Below is the basic architecture of DCGAN's Generator.



source:- literature from Unsupervised Representation Learning with DCGAN by Alec Radford

Fig. 2 Deep Convolution GAN's Generator architecture

In the beginning, we have a latent vector z , which is passed to the Generator and produces a final image of size 64*64*3. It consists of 4 convolution layers, in each layer doubling the input size.

Deep Convolution GAN Architecture

There are some general operation used in DCGAN**[2]** architecture.

- Convolution stride – for downscaling the image size used in Discriminator layer.
- Transposed convolution – For up scaling image size used in Generator.
- Fully connected layers - there aren't any fully connected layers except the discriminator's last layer and the generator's first layer.
- Batch normalization – used for normalize the input layer by adjusting and scaling the activations
- ReLu function – used only in Generator as an activation function
- LeakyReLu - used only in discriminator as an activation function.

In the beginning, we started with a small data set of size 1216, with input image of size 128*128*3, batch size of 64, z latent vector of size 100, producing output image of size 64*64*3.

Fig. 3 DCGAN results after 250 epochs

We can notice, in the landscape there are many bright strong colors . Can we say that neural networks don't like bright colors ? There are also a lot of jagged edges where there is a mixing of edges colors like a staircase of mixed colors.

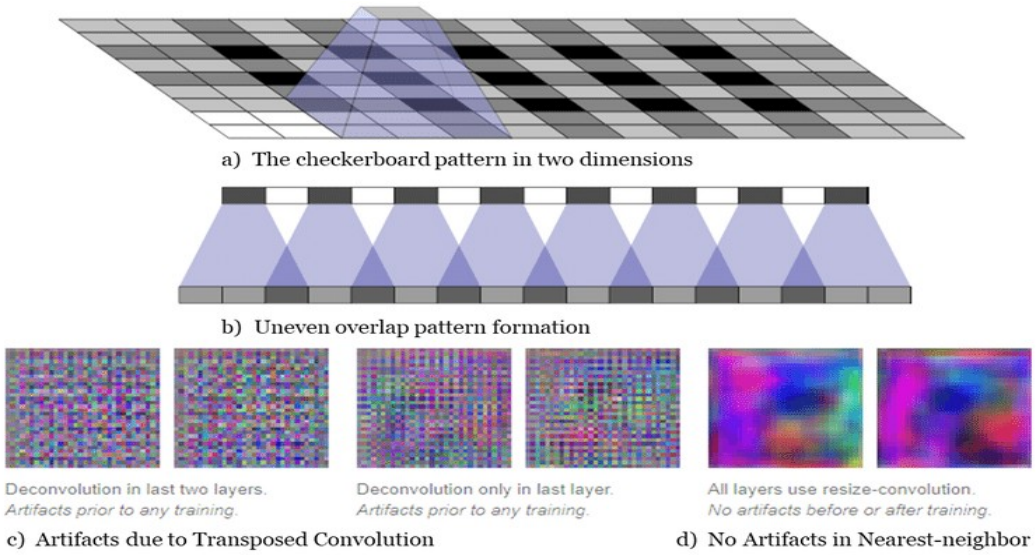
Problems with DCGAN

Some of the problems noticed in the above image are:-

- checkerboard artifacts
- mode collapse
- less details of the image

The reason for the checkerboard artifacts**[6]** is due to image up scaling technique transposed convolution. In the transposed convolution, image up scaling is done in the following way explained below.

During the upscaling, when performing convolution with the receptive window, there are some pixels, which get overlapped called as uneven overlap, producing a checkerboard **[6]** like pattern due to which there is significant amount of blurriness in the image. They seem to get more prominent and extreme in two dimensions. Neural networks in itself is a powerful technique which can learn to adjust weights and cancel out the artifacts but in practice, neural networks struggle to avoid it completely.



source - wikipedia

Fig. 4 Transposed convolution Checkerboard Artifacts

There were two solutions that were proposed to solve the checkerboard artifacts

- kernel/ stride is an integer
- use nearest neighbor interpolation

After applying 1st technique, kernel size of 4, instead 5, output image size - 128*128*3

Fig. DCGAN results, kernel / stride is an integer

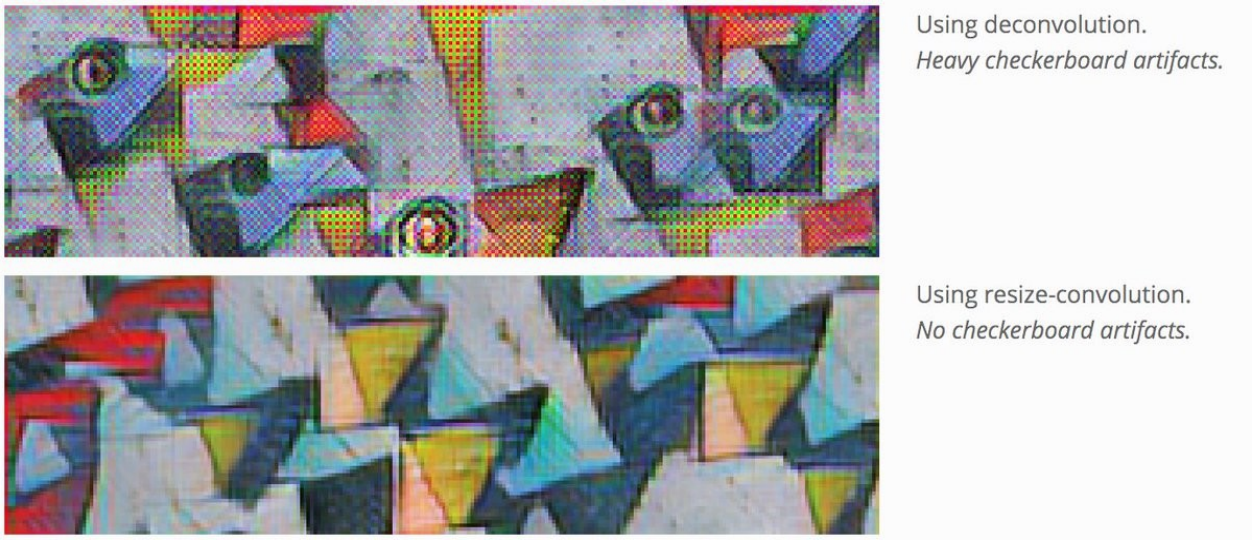
As we can see, the jagged edges got removed in the above image and there is smoothness in the above image.

Using 2nd technique, Nearest neighbor interpolation**[7]**, output image size 64*64*3, Epochs – 200

Below are the results for nearest neighbor interpolation

Fig. 5 Results Using Nearest Neighbor Interpolation

The image isn’t much clear due to low resolution output image, thought the motive behind using using Nearest Neighbor interpolation method was successful as there are more clear boundaries and less rough edges. In total, we can say that using nearest neighbor, we can solve the rough edges and uneven pixel overlap problem. Below is one result showing the difference between Transposed convolution and Nearest neighbor interpolation**[7]**.



source- **https://distill.pub/2016/deconv-checkerboard/**

Fig. 6 Results Comparison of Transposed convolution and Nearest Neighbor Interpolation

Nearest Neighbor Interpolation

A short note on how nearest neighbor **[7]** works

(0, 0) (2, 0)

1	2
3	4

2 x 2 matrix

P1	P2		

co-ordinates of P1(0.25, 0.25) , for P2 (0.75, 0.25) and so on.

We have a matrix of size (2, 2). The top left corner can be represented as origin (0, 0). The elements of the matrix can be represented by the center middle co-ordinates. Eg. ‘1’ has co-ordinates (0.5,0.5), ‘2’:(1.5,0.5), ’3’:(0.5,1.5), ’4’:(1.5,1.5).

To upscale the image by a factor of 2, following steps can be performed:-

- Projecting our 4*4 matrix to our 2*2 matrix so that we can easily find out the co-ordinates in our new matrix P1, P2 and so on.
- Now as we have the co-ordinates of our new matrix, we need to fill it with the appropriate nearest neighbor value from the original 2*2 matrix. e.g. ‘P1′(0.25,0.25) is nearest to 1

(0.5,0.5) so we assign ‘P1’ value of 1. Similarly, for other pixels, we can find their nearest pixel.

Finally, we get the final matrix as below

1	1	2	2
1	1	2	2
3	3	4	4
3	3	4	4

Since, Nearest neighbor solved the problem of checkerboard artifacts, we decided to increase the image resolution to factor of 2, i.e. to 128*128*3.

The results are shown in the below figure.

Fig. 7 Nearest neighbor approach, final image resolution 128*128*3 epochs- 250

We can see that everything is messed up. There is the problem of mode collapse and the image also aren’t detailed image.

One possible reason, that we are able to give regarding this is that it is because we double the image size from 64 to 128, which could have lead to disruption in the images.

The first technique used to upscale the image where kernel/stride is an integer produced a Frechet Inception Distance(FID) of 143. The FID score, the more low it is, the better it gets. Current state of art on ImageNet generation has the FID score of ‘7.4’ .

Wasserstein GAN

Since, in all the above techniques, we weren’t sure as to till what number of epochs we should be training our network, so that it would lead to convergence. We decided to switch over to Wasserstein Gan(WGAN). WGAN**[10]** is the gan in which the loss represent the quality of images generated over the period of epochs. In WGAN, there were some minor changes that were done in the standard DCGAN. They are listed as follows.

- No log in the loss. The output of D is no longer a probability, hence we do not apply sigmoid at the output of D
- Discriminator is replaced by the critic
- Clip the weight of D
- Train D more than G(5:1)
- Use RMSProp instead of ADAM
- Lower learning rate, the paper uses α=0.00005

In a standard DCGAN**[2]**, Discriminator gives a probability as to image being real or fake. It Is being replaced by the critic that scores the realness or fakeness of a given image.

The main idea behind wasserstein gan is that its main motive is seek minimization of the distance between distribution observed in the training data and the generated samples. This distance is known as earth mover distance, referred to as Wasserstein distance.

Moreover, we later found out that due to weight clipping, it imposed much more restriction on the network than intended . It allowed the network to help differentiate the real and fake images with just a very few features, which as a result failed to provide convincing results as mentioned in the Spectral Normalization literature.

Fig. 8 Wgan (loss oscillates and after 250 epochs remains constant at 1.57)

Well, to the motive, we introduced wgan was to see if we can actually have some metric to see and visualize the convergence of our model. The loss value over the epochs did actually represented the quality of images being generated, but it didn't improved any further.

We later saw that, most of the model we used heavily relied on convolution which is able to learn simpler representations but long range dependencies might be hard to learn for these model. Since, the detailing of the images with the fine quality was still not in the picture, a model with techniques to avoid mode collapse as well as content with quality was required. Self-Attention Generative Adversarial Networks comes with all the above mentioned characteristics therefore we switched to Self-Attention Generative Adversarial Networks(SAGAN)

Self-Attention Generative Adversarial Networks (SAGAN)

SAGAN**[4]** comes with three important changes in the model listed below:-

- Attention mechanism.
- Controls lipschitz constraint on D/G using Spectral normalisation.
- Hinge Loss

At first, we will be discussing about Attention Mechanism which is considered as the most important step to learn the long range dependencies in the network.

Long range dependencies in the image represent the relationship between one part of the image located anywhere to another part of the image. Below figure demonstrates the long range dependencies in an image.

Figure 1. The proposed SAGAN generates images by leveraging complementary features in distant portions of the image rather than local regions of fixed shape to generate consistent objects/scenarios. In each row, the first image shows five representative query locations with color coded dots. The other five images are attention maps for those query locations, with corresponding color coded arrows summarizing the most-attended regions.

source:- Literature of Self-Attention Generative Adversarial Networks

Fig. 9 Long range dependencies in an Image

To learn the long range dependencies, authors of Sagan**[4]** came up with the Attention Mechanism

Attention Mechanism

Attention mechanism**[18]** is just applied after the convolution layers which at the end generates Attention maps that are added with the convolution features. Representation of the Attention mechanism

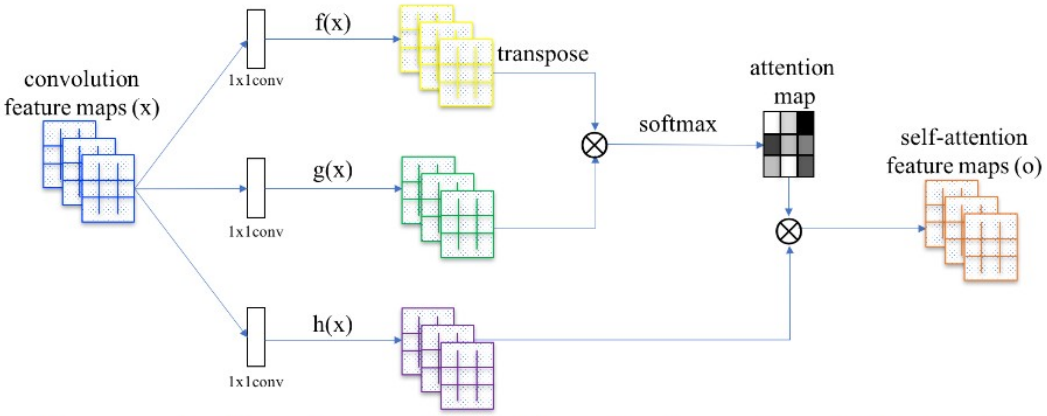


Figure 2: The proposed self-attention mechanism. The ⊗ denotes matrix multiplication. The softmax operation is performed on each row.

source: sagan paper

Fig. 10 Attention mechanism

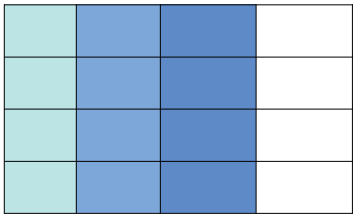
Suppose we have a set of convolution feature maps(x) generated at the end of a convolution layer. After that, we pass our feature maps through three 1x1 convolution separately to generate three set of feature maps known as f(x), g(x), h(x). what this 1x1 convolution does is that reduces the number of feature maps by a factor of 8. One point to notice is that feature maps are reduced in f and g only. h(x) has all complete set of 512 feature maps.

In order to perform self-attention on the complete image, we flatten out last two dimensions to make a matrix a row vector. eg. 512x7x7 will represent as 512x49. Now, we can perform self-attention over it. We do this by performing a transpose of the g, then multiplying f with g. Afterwards, we apply a row wise softmax on the matrix generated. These feature maps generated are known as Attention Maps. Attention maps are then multiplied with the h(x) feature maps to generate Self-Attention feature maps**[18]**. The Attention maps are then added to the feature maps x with a learnable parameter 'γ' and is initialized as 0. The final equation becomes $O=\gamma*o+x$, $\gamma \rightarrow$ *learnable parameter.*

Understanding Attention Mechanism

1 Matrix multiplication

Given two matrix, when we multiply it, we process it by multiplying first row with all the columns of the other matrix. For each respective row and column multiplication, we get a number for our new matrix.



Considering the matrix representation as our feature map, we take the first row and corresponding multiply with all the other columns. As a results, we are able to get a relationship between first row and all columns in the form of a number. This is being continued with all rows and the columns.

Later on, we perform a row wise softmax function on the generated feature maps. Softmax function converts a set of numbers to a set of probability number whose sum of probability is 1. In our row wise softmax, we are able to get how strong is the first row with the respective 1^{st}, 2^{nd}, 3^{rd} columns, then second row 1^{st}, 2^{nd}, 3^{rd} columns and so on relative to each other by means of softmax function.

To this, we multiply these probability scores to the original feature maps x, which in result gives us the long range dependencies. This is called self attention mechanism and the feature maps generated are the Self Attention feature maps.

We perform self attention mechanism on our discriminator as well as generator after middle to high set of feature maps because the network tends to give the same results with the normal convolution when attention mechanism is applied on less feature maps because in the beginning there aren't fine details available while in latter feature maps there are more details available, to which attention mechanism is beneficial.

Since the authors of SAGAN didn't notice any performance decrease when reducing the number of channels on feature maps x, by a factor of 8, we reduce the number of channels by a factor of 8 in every attention step.

The γ is initialized to 0 as in the beginning network learns information from the spatial local neighborhood and later on gradually learns from the non-local neighborhood.

The final equation can be written as follows

Attention maps (a) = $\text{softmax}_{\text{row wise}}(f(x)^T * g(x))$

Self Attention maps (o) = h(x) * a,

where f(x), g(x), h(x) are the feature maps generated after applying the 1*1 convolution.

Controlling Lipschitz constant

Lipschitz constant**[16]** is the non-negative real number L, which is the smallest upper bound of the slope, (i.e. limiter of the slope).

An example for the same, function

$$f = x^2y,\ \ x \in [-5,2],\ y \in [3,\infty]$$

$$\mathbf{L = mod(}\mathit{df/dy}\mathbf{) = sup|x^2| = 25}$$

Controlling lipschitz constant**[16]** in our network is done by performing spectral normalization on our model.

Spectral Normalization

Given a matrix A, it is defined as the largest singular value of the matrix A , i.e., the square root of the largest eigenvalue of A^TA.

$$\sigma(W) = \text{Singular value decomposition} = \text{sqrt}\ (\lambda_{max}\ (A^TA)$$

We normalize our weight matrix using the following equation

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

Exploring Singular Value Decomposition

For a given matrix W, to find the singular value**[12]**, we need to re-organize our matrix in the form of three matrix.

$$\mathbf{W = u\ \Sigma\ v^T}$$

u, v is an orthogonal matrix, Σ diagonal matrix with singular values

W can be written as $\mathbf{[u_1\ u_2]\ \ \Sigma\ \begin{bmatrix} v_1^T \\ v_2^T \end{bmatrix}}$

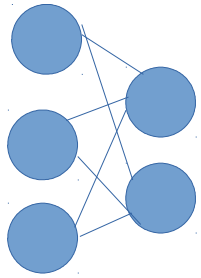
u_1, u_2 are the left singular vectors, v_1 , v_2 are the right singular vectors

but we are interested in only singular values.

The obvious question why are we breaking matrix in those three different matrix ? Decomposition to a singular value and the singular vectors actually represent the most important information in the matrix which is in the first left singular vector, first singular value, first right singular vector. The most important variance in the matrix can be learned when performing a singular value decomposition**[12]**. So, we want to control our lipschitz constant in our model using spectral normalization through singular value decomposition**[12]**. Controlling the lipschitz constant actually restricts out weight matrix to the weight elements having the most strongest variation.

Layer wise spectral normalization

Consider we have a full connected layer h as below, with weight matrix as W



W =>3x2, assume layer h, g(h) =Wh ...eq(1)

$\|g\|_{lip} = \sup_h \sigma(\nabla g(h)) = \sup_h \sigma(W)$, ….eq(2)

we know that smallest upper bound $\|a_l\| = 1$, activation function being reLu or Leaky reLu

by the lipschitz property, $\|g1 \circ g2\|_{Lip} \leq \|g1\|_{Lip} \cdot \|g2\|_{Lip}$ **[3]**

$$\|f\|_{Lip} <= \|(w^{l+1}h_l)\|_{Lip}. \|a_L\|_{Lip} . \|(w^l h_{l-1})\|_{Lip}. \|a_{L-1}\|_{Lip} \ldots\ldots\ldots$$

f is the discriminator function and we know $\|a_l\| = 1$

$$\prod_{l=1 \text{ to } L+1} \sigma(w^l) \quad \text{from eq 1 \& 2}$$

$$W_{sn}(W) = W/\sigma(W)$$

Why Spectral Normalization ?

There are some other normalization techniques, one of them is weight normalization. Weight normalization imposes much more constraint on the matrix than intended. Weight clipping also imposes such constraints.

It also reduces the rank of the matrix to 1 which means that the model will be using just one feature to discriminate the model probability distribution to the target distribution and the model will be concentrate on just one feature and will exploit to distinguish between real and fake images.

Since, singular value decomposition is computationally heavy. Doing this a lot of times in the model will just make the model computationally heavy. Therefore, we switch to an iteration method known as power iteration method which is described below taken directly from the SAGAN paper.

Algorithm 1 SGD with spectral normalization

- Initialize $\tilde{\boldsymbol{u}}_l \in \mathcal{R}^{d_l}$ for $l = 1, \ldots, L$ with a random vector (sampled from isotropic distribution).
- For each update and each layer l:
  1. Apply power iteration method to a unnormalized weight W^l:

$$\tilde{\boldsymbol{v}}_l \leftarrow (W^l)^{\mathrm{T}} \tilde{\boldsymbol{u}}_l / \|(W^l)^{\mathrm{T}} \tilde{\boldsymbol{u}}_l\|_2 \tag{20}$$

$$\tilde{\boldsymbol{u}}_l \leftarrow W^l \tilde{\boldsymbol{v}}_l / \|W^l \tilde{\boldsymbol{v}}_l\|_2 \tag{21}$$

  2. Calculate $\bar{W}_{\mathrm{SN}}$ with the spectral norm:

$$\bar{W}^l_{\mathrm{SN}}(W^l) = W^l / \sigma(W^l), \text{ where } \sigma(W^l) = \tilde{\boldsymbol{u}}_l^{\mathrm{T}} W^l \tilde{\boldsymbol{v}}_l \tag{22}$$

  3. Update W^l with SGD on mini-batch dataset $\mathcal{D}_M$ with a learning rate α:

$$W^l \leftarrow W^l - \alpha \nabla_{W^l} \ell(\bar{W}^l_{\mathrm{SN}}(W^l), \mathcal{D}_M) \tag{23}$$

source:- spectral normal for generative adversarial networks

Fig. 11 Spectral Normalization Algorithm

Spectral normalization only on Discriminator

Authors of SAGAN argues about applying spectral normalization on generator as well. They said that as the generator is an important factor in training as well, spectral normalization should be applied on it as well. Applying spectral normalization**[4]** just on discriminator with 1:1 balanced updates for the discriminator resulted in a very unstable training behavior. It showed mode collapse very early.

Mitigation for the unstable training behavior

This effect is mitigated by applying 5:1 update rule on discriminator and generator respectively. This is due to more updates of the discriminator per generator update. Thus, the gradients from the discriminator are supplied more frequently. There are more frequent gradient updates from the discriminator.

Spectral Normalization On Discriminator and Generator

To improve the model stability, we apply a 5:1 update rule. Later on, SAGAN authors found that applying spectral normalization on D/G can actually avoid to update the 5:1 update restriction and also stabilized the training to a great extent. After applying spectral normalization on D/G, D and G were updated 1:1 update rule. Using this, the quality of the samples didn't increased monotonically.

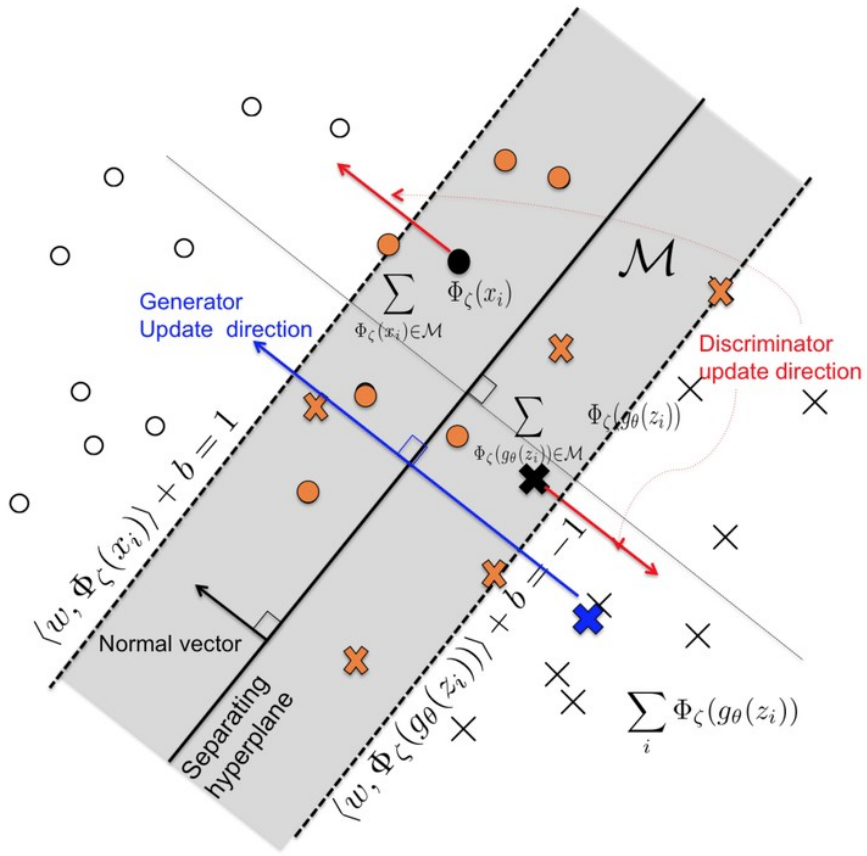
To this they introduced a new update rule known as two-timescale update rule (TTUR)**[13]**. In this we use an imbalanced learning rates where the learning rate of the discriminator is a few point more

that the generator. This helped in maintaining the sample quality over the epochs. So, in all we used spectral normalization on D/G with TTUR **[13]**. By default, the learning rate for the discriminator is 0.0004 and the learning rate for the generator is 0.0001 **[15].**

Spectral normalization	**On D only**	**With 5:1 update**	**On D/G**	**With TTUR**
Problems	Unstable training and mode collapse	Need to search for suitable number ratio of D:G.	Stabilized model, balanced updates but not monotonic increase in quality of samples	Qualitative and quantitative results with a stabilized model

Hinge loss

Loss function is one of the crucial step in the training process to do the back propogation. We have used a new loss function known in the GAN's knows as Hinge loss**[8]**. This is also used by SAGAN authors. There's also the possibility to try other loss function as well. Hinge loss is basically used in Support vector machines. Explained below is to why we are using the hinge loss.



Source:- Literature of Geometric GAN

Fig. 12 Hinge loss in SAGAN

The above figure represents the Hinge loss **[8]** and gradient updates in the discriminator and generator. Well, it looks a bit complex to understand but it isn't as we don't have to focus on all the equation and things. **A small attention is needed towards the separating hyper plane and the blue and red arrows and their corresponding direction.**

We know that task of the discriminator is to distinguish between real and fake images. So, the discriminator task in the above diagram is to update the points away from the hyper plane. In this way only, the discriminator can perform its task to distinguish between real and fake images. We see the red arrow is making updates away from the hyper plane while the blue arrow is making updates of the gradients towards the hyper plane which signifies that the generator is trying to trick the discriminator into identifying fake images as real.

SAGAN model, Discriminator

Self Attention Results :-

Fig. 13 Sagan results after 8 training(10k iterations / epoch) epochs

Conclusion

Creative Art Generation is one of the potential applications that is getting attention in art as well as Computer vision research field too. If explored well, it can replace the human art painting in many domains. Various Generative Adversarial Networks(GAN) models such as Deep Convolution Gan's, Wasserstein Gan's, Self Attention have been used to achieve the results close to state of the art. Self

Attention was the most successful in during the above experiments. DCGAN, WGAN to an extent were good to an extent but suffered problems of image content, mode collapse, too much restrictive network and image quality. Self-Attention Gan successfully resolved all the issues and produced image with good variant and quality with an Frechet Inception Distance score**[13]** of 62. Self Attention with its attention mechanism and spectral normalization produced excellent results. This is the first time SAGAN model is being used for Art Generation. Despite these successful results in Art generation, more research could be done to produce art of different genre and styles and have more refined quality.

Code Snippet (data pre-processing)

```
from __future__ import division
import numpy as np
import imageio
import math
import matplotlib.pyplot as plt
from skimage.transform import resize
from skimage.io import imread
import tensorflow as tf
from skimage import transform
from PIL import Image

import imageio
import shutil
import scipy.misc
import os
import pandas as pd


print(tf.__version__)

def sort_images(csv_file):
    c = 0
    df = pd.read_csv(os.getcwd() + '/' + csv_file)
     image_column = df.path                                  # contain the name of the user specific genre image 'eg -
abc.jpg'
    os.mkdir(os.getcwd() + '/data/train_9_landscape')
    for i in image_column:
        try:
              shutil.move(os.getcwd() + '/data/train_9/' + str(i), os.getcwd() + '/data/train_9_landscape') #move
img(all genres) from a folder to img (usr choice genre) to another folder
            c = c+1
        except:
            print('pic not there')
    return c

def prepare_data(folder_name, resize_ht, resize_wt):
    image_path = os.getcwd() + '/data/' + folder_name
    file_images = os.listdir(image_path)
```

```
    os.mkdir(os.getcwd() + '/data/discarded_images')
    os.mkdir(os.getcwd() + '/data/resized_images')
    for i in file_images:
        image = imageio.imread(image_path + '/' + i)
            if len(image.shape) != 3 or image.shape[-1] != 3:         # check for color or gray scale ; remove if grayscale
            shutil.move(image_path + '/' + i, os.getcwd()+'/data/discarded_images')
            print('%i Image Moved', i)
        else:
            resize_image = resize(image, [resize_ht, resize_wt])
            data = 255 * resize_image  # Now scale by 255
            img = data.astype(np.uint8)
            imageio.imsave(os.getcwd()+'/data/resized_images/'+i, img, format='jpeg')



# FIRST RUN THIS FUNCTION
# count = sort_images('landscape_9.csv') # csv contains column(named path) with all image names
# print(count)

# SECOND RUN THIS FUNCTION
# prepare_data('train_9_landscape', 128, 128) # name of folder containing images to be resized
```

References

[1] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley,Sherjil Ozair, Aaron Courville, Yoshua Bengio, Generative Adversarial Networks, 2014.

[2] Alec Radform, Luke Metz, Soumith Chintala, Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks, 2015.

[3] Takeru Miyato, Toshiki Kataoka, Masaonri Koyama, Yuichi Yoshida, Spectral Normalization for Generative Adversarial Networks, pages 4-5, 2018.

[4] Hang Zhang, Ian Goodfellow, Dimitris Metaxas, Augustus Odena, Self-Attention Generative Generative Adversarial Networks, 2018.

[5] Wikiart.org, Painter by Numbers, 2015, https://www.kaggle.com/c/painter-by-numbers.

[6] Augustus Odena, Vincent Dumoulin, Chris Olah, Deconvolution and Checkerboard Artifcats, 2016.

[7] Kang, Atul, Image Processing – Nearest Neighbour Interpolation, 2018.

[8] Lim Jae Hyun, Ye Jong Chul, Geometric GAN, 2017.

[9] Wang Xiaolong, Girshick Ross, Gupta Abhinav, He Kaiming, Non-local neural networks, 2018.

[10] Arjovsky Martin, Chintala Soumith, Bottou Leon, Wasserstein GAN, 2017.

[11] Gulrajani Ishaan, Ahmed Faruk, Arjovsky Martin, Dumoulin Vincent, Courville Aaron, Improved Training of Wasserstein GAN, 2017.

[12] Singular Value Decomposition, MIT OpenCourseWare, 2016 https://www.youtube.com/watch?v=mBcLRGuAFUk.

[13] Heusel, M., Ramsauer, H., Unterthiner, T., Nessler, B., andHochreiter, S. GANs trained by a two time-scale updaterule converge to a local nash equilibrium. InNIPS, pp.6629–6640, 2017.

[14] Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. In ICLR, 2015

[15] Cosgrove Christian, Spectral Normalization Explained, 2018
[16] Lipschitz Constant , https://www.eee.hku.hk/~msang/Numerical_Lipschitz.pdf, 2013
[17] Walia Anish, Types of Optimization Algorithms used in Neural Networks and Ways to Optimize Gradient Descent, 2017
[18] Jha Divyansh, Not just another GAN paper — SAGAN, 2018