A Style-Based Generator Architecture for Generative Adversarial Networks

Mohammed Khadeeruddin Vemana Vijay Kumar

Overview

- A quick introduction to GANs
- GAN Objective function
- GAN techniques
    - Progressive Growing
    - Style GAN
- Style GAN Model Architecture
- Playing with latent space of Style GAN (Project Execution)

Introduction

Generative Adversarial Networks(GANs) are a new concept in Machine Learning, introduced for the first time in **2014** by **Ian good fellow.**

StyleGAN was originally an open-source project by NVIDIA to create a generative model that could output high-resolution human faces. The basis of the model was established by a research paper published by Tero Karras, all researchers at NVIDIA.

This architecture is able to separate the high-level attributes from low-level attributes within an image. This doesn't affect the original image just apply the style to that image like hair-style, applying eyeglasses.

A quick introduction to GANs

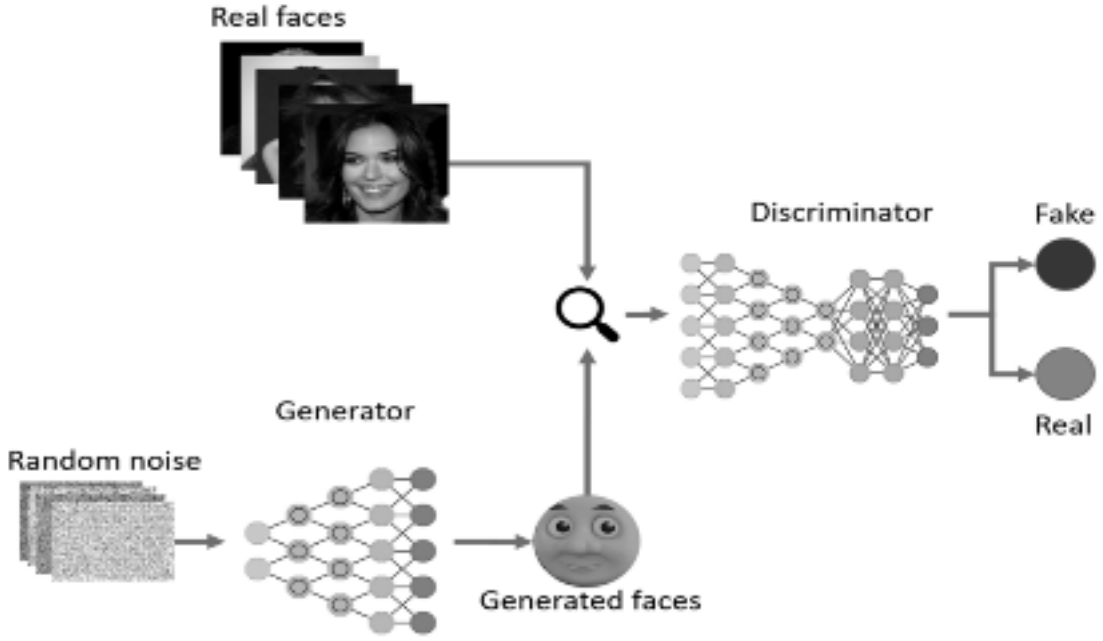
WHAT IS GAN?

A generative adversarial network (GAN) is a type of machine learning technique made up of two neural networks.

The two neural networks that make up a GAN are:

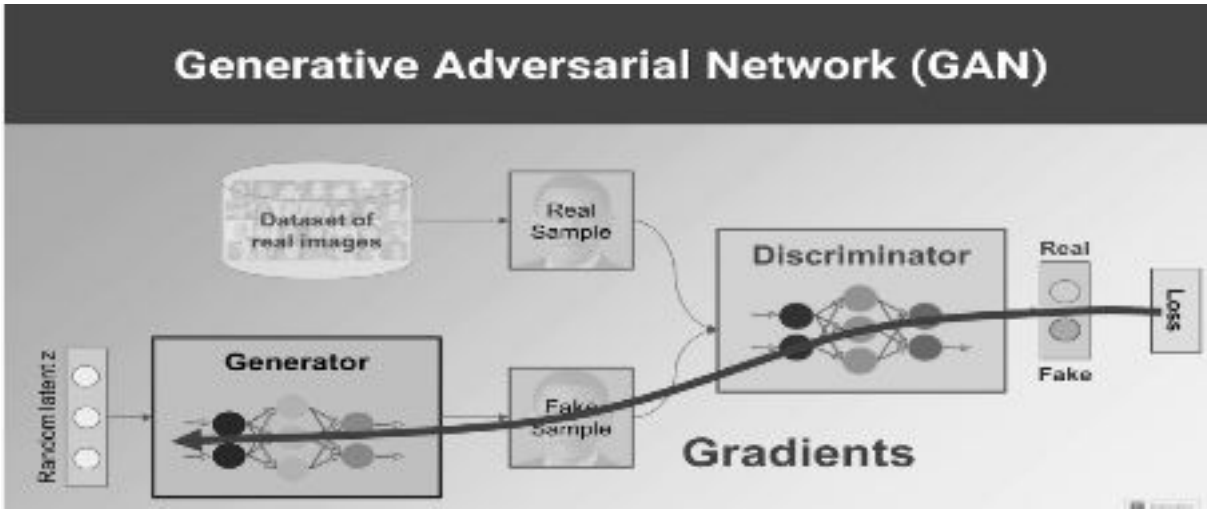
- A GENERATOR with a goal to generate new instances of an object that will be indistinguishable from the real ones, and
- A DISCRIMINATOR that takes samples from both the training data and the generator's output and predicts if they are "real" or "fake".

GANs can be used to create all types of content including images, video, audio and text. The generator input is a random vector (noise) and therefore its initial output is also noise. Over time, as it receives feedback from the discriminator, it learns to synthesize more "realistic" images. The discriminator also improves over time by comparing generated samples with real samples, making it harder for the generator to deceive it.



Basic GAN architecture

The discriminator is a classification model. Cross-entropy loss is better than MeanSquareError or Misclassification. So Discriminator produces loss and the nice thing is that the generator is also a fully differentiable network. So, if we stick these two networks back to back then we can backpropagate the learning signal. This way we can update generator and discriminator with the same loss function until they both are fully good at their job.



Discriminator loss is passed to the Generator as an objective function.

GAN Objective function

Basically generator and Discriminator plays Min-Max game. Because the generator makes images to fool the discriminator and discriminator is trying to be right all the time. Here we Minimize generator and Maximize the Discriminator.

$$\textbf{Min Max } V(D, G) = \mathbf{E}_{x \sim P_{data}(x)}[\log D(x)] + \mathbf{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))]$$

If we put this objective function into an Algorithm. This is what we get.
So every training Algorithm Is start with two steps:

We sample a batch of noise vectors and a batch of images from dataset. Then we use the objective function to update the parameter of Discriminator by doing gradient ascend with respect to its parameters.

We sample a new batch of noise vectors and generate images with them and apply gradient descend on the second part of the objective function in order to update the parameters of the generator.

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**

for k steps **do**

- Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
- Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
- Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

end for

- Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
- Update the generator by descending its stochastic gradient:

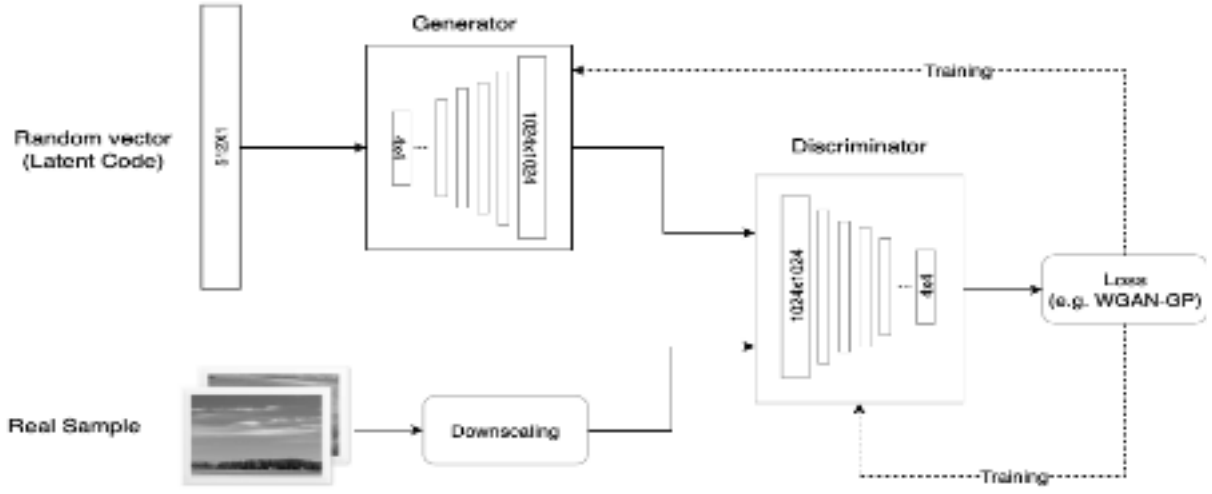
$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

GAN techniques

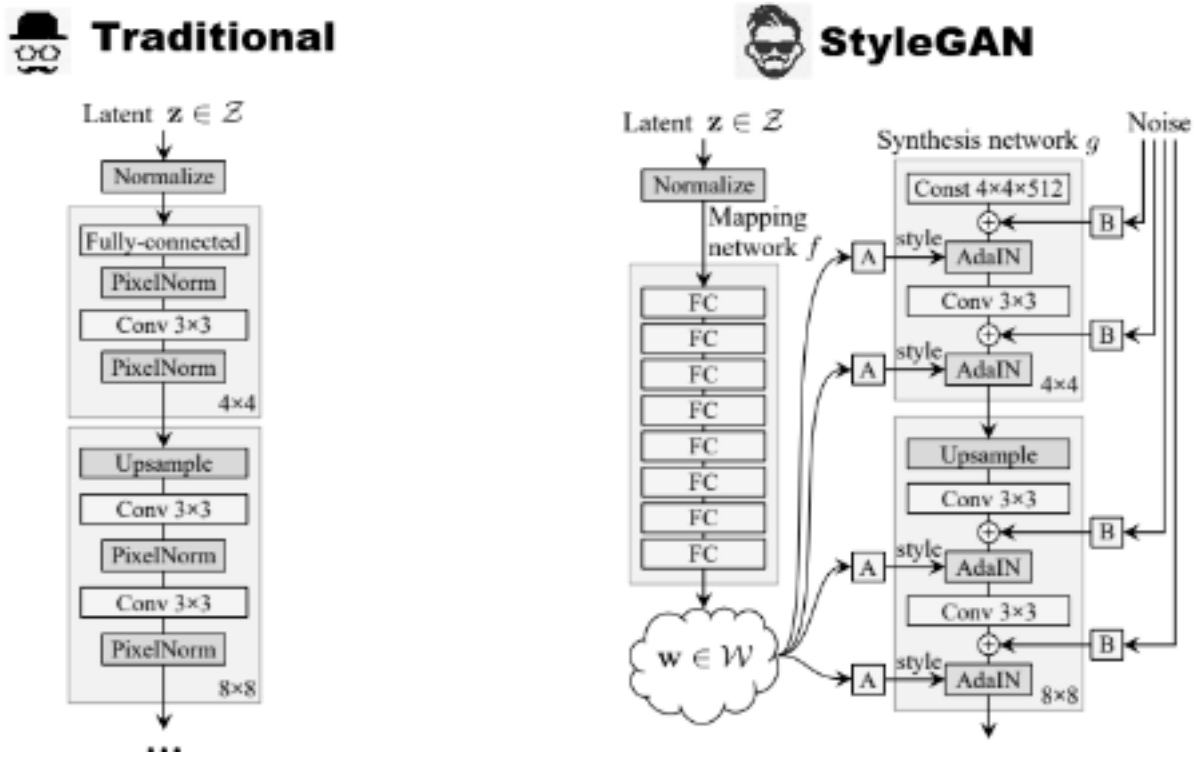
Progressive Growing: It is the layers in the Generative model which is published by NVIDIA in a paper called Progressive Growing of GANs. We basically start with a generative model which generate very low images with low resolution and same time discriminator also start with the low-resolution images. It is simple and stabilized quickly. Once t stabilized then we add another layer to both and keep on training.



Progressive Growing of GANs.

Style GAN

A new paper by NVIDIA, A **Style-Based Generator Architecture** for GANs (Style GAN). Style GAN generates the artificial image gradually, starting from a very low resolution and continuing to a high resolution (1024×1024). By modifying the input of each level separately, it controls the visual features that are expressed in that level, from coarse features (pose, face shape) to fine details (hair colour), without affecting other levels.



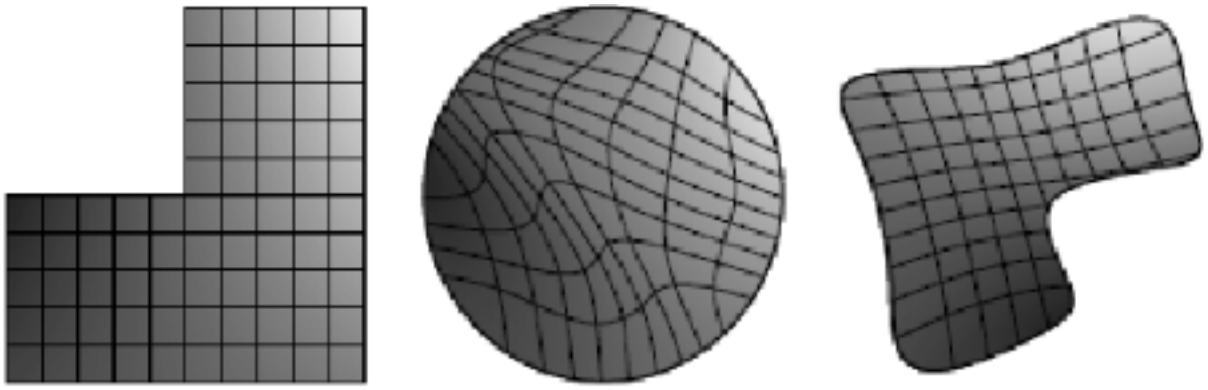
Style GAN Model Architecture

First, it has a mapping network, It takes noise vector ‘z’ and transfers into a different vector called ‘w’. The ‘w’ vector does not have to be Gaussian anymore. It can be like whatever the generator want to be then actual generator architecture does not start from noise vector. It optimized during the training. It kinda fixed seed at the Beginning. The output ’w’ is plugged into multiple layers using layer called AdaIN (Adaptive Instance Normalization) and in training, we add noise to this parameter.

Why we are using mapping layer?

Imagine we have a dataset and we look at two properties gender and facial hair. So we have male and female and we have a beard and no beard. In most of the

women, you find very little women have a beard, that's why it has a gap there. In other words, data distribution has a gap and if we sample from Gaussian distribution it is uniform, it does not have gaps. It allows you to sample for uniform distribution and work such a way that you have a gap. It is done by 'w' vector then the generator takes an input and generates images.



(a) Distribution of features in training set

(b) Mapping from $\mathcal{Z}$ to features

(c) Mapping from $\mathcal{W}$ to features

Conclusion

The techniques presented in Style GAN, especially the Mapping Network and the Adaptive Normalization (AdaIN), will likely be the basis for many future innovations in GANs.

System requirements

- Both Linux and Windows are supported, but we strongly recommend Linux for performance and compatibility reasons.
- 64-bit Python 3.6 installation. We recommend Anaconda3 with NumPy 1.14.3 or newer.
- Tensor Flow 1.10.0 or newer with GPU support.
- One or more high-end NVIDIA GPUs with at least 11GB of DRAM. We recommend NVIDIA DGX-1 with 8 Tesla V100 GPUs.
- NVIDIA driver 391.35 or newer, CUDA toolkit 9.0 or newer, cuDNN 7.3.1 or newer.

- Make sure to run this code on a GPU instance. GPU is assumed.
- First, map your G-Drive, this is where your GANs will be written to.

Examining the Latent Vector

Setup to use TF 1.0 (as required by NVidia).

Run this for Google CoLab (use Tensor Flow 1.x)

```
%tensorflow_version 1.x
from google.colab import files

# next, clone StyleGAN2 from GitHub.

!git clone https://github.com/NVlabs/stylegan2.git

# Verify that StyleGAN has been cloned.
!ls /content/stylegan2/
```

Add the StyleGAN folder to Python so that you can import it. The code below is based on code from NVidia. This actually generates your images.

```
import sys
sys.path.insert(0, "/content/stylegan2")

import dnnlib

# Copyright (c) 2019, NVIDIA Corporation. All rights reserved.
# This work is made available under the Nvidia Source Code License-NC.
# To view a copy of this license, visit
# https://nvlabs.github.io/stylegan2/license.html


import argparse
import numpy as np
import PIL.Image
```

```
import dnnlib
import dnnlib.tflib as tflib
import re
import sys
import os
import pretrained_networks

#----------------------------------------------------------------------

def expand_seed(seeds, vector_size):
  result = []

  for seed in seeds:
    rnd = np.random.RandomState(seed)
    result.append( rnd.randn(1, vector_size) )
  return result

def generate_images(Gs, seeds, truncation_psi):
    noise_vars = [var for name, var in Gs.components.synthesis.vars.items() if na
me.startswith('noise')]

    Gs_kwargs = dnnlib.EasyDict()
    Gs_kwargs.output_transform = dict(func=tflib.convert_images_to_uint8, nch
w_to_nhwc=True)
    Gs_kwargs.randomize_noise = False
    if truncation_psi is not None:
        Gs_kwargs.truncation_psi = truncation_psi

    for seed_idx, seed in enumerate(seeds):
        print('Generating image for seed %d/%d ...' % (seed_idx, len(seeds)))
        rnd = np.random.RandomState(seed=1)
        tflib.set_vars({var: rnd.randn(*var.shape.as_list()) for var in noise_vars}) #
 [height, width]
        images = Gs.run(seed, None, **Gs_kwargs) # [minibatch, height, width, c
hannel]
```

```
        path = f"/content/tmp/image-{seed_idx}.png"
        PIL.Image.fromarray(images[0], 'RGB').save(path)

# Specify the two seeds that we wish to morph between.

sc = dnnlib.SubmitConfig()
sc.num_gpus = 1
sc.submit_target = dnnlib.SubmitTarget.LOCAL
sc.local.do_not_copy_source_files = True
sc.run_dir_root = "/content/drive/My Drive/projects/stylegan2"
sc.run_desc = 'generate-images'
network_pkl = 'gdrive:networks/stylegan2-ffhq-config-f.pkl'

print('Loading networks from "%s"...' % network_pkl)
_G, _D, Gs = pretrained_networks.load_networks(network_pkl)
vector_size = Gs.input_shape[1:][0]

vec = expand_seed( [100,860], vector_size) # Morph between these two seeds

print(vec[0].shape)

# We can now do a morph over 300 steps.
STEPS = 300
diff = vec[1] - vec[0]
step = diff / STEPS
current = vec[0].copy()

vec2 = []
for i in range(STEPS):
  vec2.append(current)
  current = current + step

temp_path = "/content/tmp"

# Create a temporary directory to hold video frames
```

```
try:
  os.mkdir(temp_path)
except OSError:
  print("Temp dir already exists.")

generate_images (Gs, vec2, truncation_psi=0.5)

# We are now ready to build the video.

! ffmpeg -r 30 -i /content/tmp/image-%d.png -vcodec mpeg4 -y /content/gan_m
orph.mp4


# Download the video.
files.download('/content/gan_morph.mp4')
```

Output in the form of video:

References

Goodfellow, I. J., Warde-Farley, D., Lamblin, P., Dumoulin, V., Mirza, M., Pascanu, R., Bergstra, J., Bastien, F., and Bengio, Y. (2013c). Pylearn2: a machine learning research library. ArXiv preprint arXiv: 1308.4214

L. A. Gatys, A. S. Ecker, and M. Bethge. Image style transfer using convolutional neural networks. In *Proc. CVPR*, 2016.