1. **Roles of Generator and Discriminator in a GAN**

A **Generative Adversarial Network (GAN)** consists of two competing neural networks:

Generator

* Purpose: **Create synthetic (fake) data** that mimics the real data distribution.
* Input: **Random noise** (latent vector).
* Goal: **Fool the discriminator** into thinking generated images are real.

Discriminator

* Purpose: **Classify inputs** as **real** (from dataset) or **fake** (from generator).
* Input: **Image (either real or generated)**.
* Goal: **Correctly distinguish real vs. fake images**.

Adversarial Training

* The **generator** improves by producing more realistic data.
* The **discriminator** improves by becoming better at spotting fakes.
* Over time, **both models improve together** in a balanced "zero-sum game".

When trained successfully, the generator can create **extremely realistic** images that are **indistinguishable from real ones**.

1. **Create Artwork by Manipulating Latent Space (Facial Features)**
2. **Define functions**

def interpolate\_hypersphere(v1, v2, n\_steps):

v1\_norm = v1 / tf.norm(v1)

v2\_norm = v2 / tf.norm(v2)

dot = tf.reduce\_sum(v1\_norm \* v2\_norm)

omega = tf.acos(dot)

sin\_omega = tf.sin(omega)

steps = []

for t in np.linspace(0, 1, n\_steps):

factor1 = tf.sin((1 - t) \* omega) / sin\_omega

factor2 = tf.sin(t \* omega) / sin\_omega

vector = factor1 \* v1\_norm + factor2 \* v2\_norm

steps.append(vector)

return tf.stack(steps)

# Function to animate images into a GIF

def animate(images, filename="latent\_interpolation.gif"):

images = np.clip((images + 1) / 2.0, 0, 1) # Rescale [-1,1] to [0,1]

images = (images \* 255).astype(np.uint8)

with imageio.get\_writer(filename, mode='I') as writer:

for img in images:

writer.append\_data(img)

return filename

# Display single image helper

def display\_image(img):

img = (img + 1) / 2.0 # Rescale [-1,1] to [0,1]

plt.imshow(img)

plt.axis('off')

plt.show()

1. **Interpolate between two random latent vectors**

def interpolate\_between\_vectors():

v1 = tf.random.normal([latent\_dim])

v2 = tf.random.normal([latent\_dim])

interpolated\_vectors = interpolate\_hypersphere(v1, v2, 50)

generated\_images = progan(interpolated\_vectors)['default']

return generated\_images

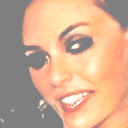
# Generate and visualize

interpolated\_images = interpolate\_between\_vectors()

# Animate

animate(interpolated\_images)

display.Image(filename="latent\_interpolation.gif")

****

1. **Find closest vector to target**

def upload\_target\_image():

uploaded = files.upload()

image = imageio.imread(io.BytesIO(uploaded[list(uploaded.keys())[0]]))

image = transform.resize(image, (128, 128))

return image

image\_from\_model = True

if image\_from\_model:

target\_vector = tf.random.normal([1, latent\_dim])

target\_image = progan(target\_vector)['default'][0]

else:

target\_image = upload\_target\_image()

display\_image(target\_image)

1. **Optimization**

if not image\_from\_model:

target\_image = (target\_image \* 2.0) - 1.0 # Scale from [0, 1] → [-1, 1]

# Optimization function to find closest latent vector

def find\_closest\_latent\_vector(initial\_vector, target\_image, steps=300, steps\_per\_image=5):

images = []

losses = []

vector = tf.Variable(initial\_vector)

optimizer = tf.keras.optimizers.Adam(learning\_rate=0.01)

loss\_fn = tf.keras.losses.MeanAbsoluteError(reduction="sum")

for step in range(steps):

with tf.GradientTape() as tape:

generated\_image = progan(vector.read\_value())['default'][0]

loss = loss\_fn(generated\_image, target\_image[:,:,:3])

# Regularize vector magnitude

regularizer = tf.abs(tf.norm(vector) - np.sqrt(latent\_dim))

total\_loss = loss + 0.1 \* regularizer

grads = tape.gradient(total\_loss, [vector])

optimizer.apply\_gradients(zip(grads, [vector]))

if step % steps\_per\_image == 0:

images.append(generated\_image.numpy())

losses.append(total\_loss.numpy())

if step % 1000 == 0:

print(f"Step {step}, Loss: {total\_loss.numpy():.4f}")

return images, losses

# Run optimization

initial\_vector = tf.random.normal([1, latent\_dim])

optimized\_images, loss\_curve = find\_closest\_latent\_vector(initial\_vector, target\_image)

# Save GIF animation of optimization steps

animate(np.stack(optimized\_images), filename="optimization.gif")

display.Image(filename="optimization.gif")

****

Step 0, Loss: 4322.8569

Step 100, Loss: 1619.2676

Step 200, Loss: 359.6121

1. **Roles of Generator and Discriminator in a GAN**

We use StyleGAN2 with pretrained weights to generate realistic, high-quality artworks.

1. **Modules Installation and Download the weights:**

!git clone https://github.com/NVlabs/stylegan2-ada-pytorch.git

%cd stylegan2-ada-pytorch

# FFHQ 1024x1024 config-f weights from NVIDIA's original StyleGAN2 release

!wget https://nvlabs-fi-cdn.nvidia.com/stylegan2-ada-pytorch/pretrained/ffhq.pkl

1. **Generate Random Artwork**

import torch

import legacy

import numpy as np

import PIL.Image

import dnnlib

# Load network

network\_path = 'ffhq.pkl'

device = torch.device('cpu')

with open(network\_path, 'rb') as f:

G = legacy.load\_network\_pkl(f)['G\_ema'].to(device) # Load generator

# Generate image

z = torch.randn([1, G.z\_dim], device=device) # Latent vector

label = torch.zeros([1, G.c\_dim], device=device) # No conditional label

img = G(z, label, truncation\_psi=0.7, noise\_mode='const')

# Convert and save image

img = (img.clamp(-1, 1) + 1) / 2 # Normalize to [0,1]

img = (img \* 255).to(torch.uint8)

img = img[0].permute(1, 2, 0).cpu().numpy()

PIL.Image.fromarray(img, 'RGB').save('stylegan2\_art.png')

PIL.Image.open('stylegan2\_art.png')

**A person smiling for the camera

Description automatically generated**