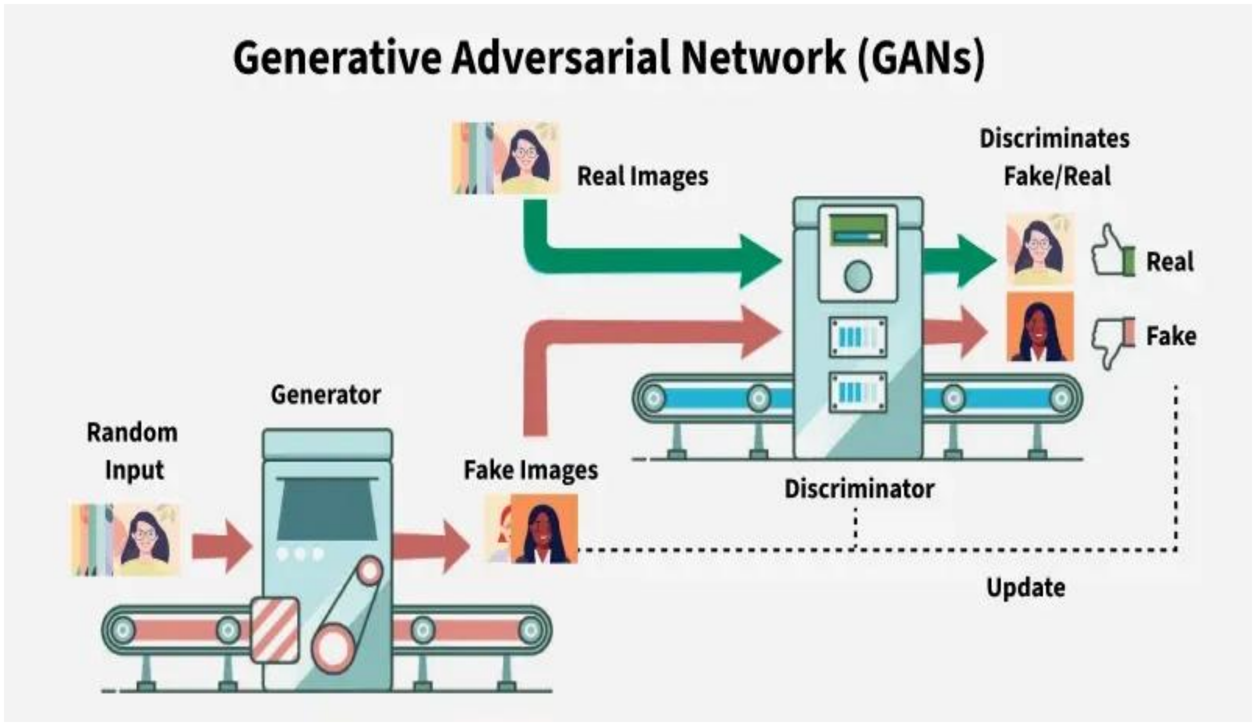

Generative Adversarial Networks (GANs) in Computer Vision

Learning Objectives

By the end of this lesson, you will:

1. Understand the architecture of GANs.
2. Learn how the Generator and Discriminator networks interact.
3. Explore the challenges in training GANs and best practices.
4. Implement a simple GAN for image generation in Python.

---

1. GAN Architecture

Generative Adversarial Networks (GANs) were introduced by Ian Goodfellow in 2014. GANs consist of two neural networks that compete with each other in a game-theoretic framework:

1.1 Generator (G)

- The Generator takes a random noise vector zz as input.
- It learns to generate realistic-looking images.
- The goal is to **fool the Discriminator** into classifying the generated images as real.

Mathematically, the Generator G(z)G(z) maps a random noise zz (sampled from a distribution, such as a normal distribution) to the data space (images):

G(z;θG)→XfakeG(z; \theta_G) \rightarrow X_{\text{fake}}

where θG\theta_G are the parameters of the Generator.

1.2 Discriminator (D)

- The Discriminator takes an image as input and predicts whether it's real or fake.
- It is a binary classifier trained using cross-entropy loss.
- It learns to distinguish between real images from the dataset and fake images from the Generator.

Mathematically, the Discriminator D(X)D(X) outputs a probability that the input XX is real:

D(X;θD)→[0,1]D(X; \theta_D) \rightarrow [0,1]

where θD\theta_D are the parameters of the Discriminator.

---

2. Training GANs

2.1 Objective Function (Minimax Game)

The Generator and Discriminator are trained simultaneously using the following loss function:

minGmaxDV(D,G)=EX∼pdata[logD(X)]+EZ∼pz[log(1−D(G(Z)))]\min_G \max_D V(D, G) = \mathbb{E}_{X \sim p_{\text{data}}}[\log D(X)] + \mathbb{E}_{Z \sim p_z}[\log (1 - D(G(Z)))]

- The **Discriminator** tries to maximize logD(X)+log(1−D(G(Z)))\log D(X) + \log (1 - D(G(Z))) (it gets better at distinguishing real and fake images).
- The **Generator** tries to minimize log(1−D(G(Z)))\log (1 - D(G(Z))) (it tries to fool the Discriminator).

2.2 Training Steps

1. **Train the Discriminator**:
   - Feed real images from the dataset and train DD to classify them as real.
   - Feed generated images from GG and train DD to classify them as fake.
2. **Train the Generator**:
   - Generate fake images using random noise.
   - Compute the loss based on how well the Discriminator is fooled.
   - Update GG to produce more realistic images.

2.3 Challenges in Training

Training GANs is difficult due to:

- **Mode Collapse**: The Generator produces limited variations of images.
- **Vanishing Gradients**: The Discriminator may become too strong, making it hard for the Generator to learn.
- **Instability**: Training is highly sensitive to hyperparameters.

2.4 Best Practices

- Use **batch normalization** in both networks.
- Use **Leaky ReLU** activation in the Discriminator.
- Apply **label smoothing** to make training more stable.
- Use **Adam optimizer** with proper hyperparameters.

---

3. Implementation: Building a Simple GAN in Python

We will use TensorFlow and Keras to implement a GAN for generating handwritten digits using the **MNIST dataset**.

Step 1: Import Libraries

```
import tensorflow as tf

from tensorflow import keras

from tensorflow.keras import layers

import numpy as np

import matplotlib.pyplot as plt
```

Step 2: Build the Generator

```
def build_generator():

    model = keras.Sequential([

        layers.Dense(128, activation="relu", input_shape=(100,)),

        layers.BatchNormalization(),

        layers.Dense(256, activation="relu"),

        layers.BatchNormalization(),
        layers.Dense(512, activation="relu"),
        layers.BatchNormalization(),
        layers.Dense(28 * 28, activation="tanh"),
        layers.Reshape((28, 28))
    ])
    return model
```

- Input: 100-dimensional noise vector.
- Output: 28×28 grayscale image.
- Uses **Batch Normalization** for stable training.

Step 3: Build the Discriminator

```
def build_discriminator():
    model = keras.Sequential([
        layers.Flatten(input_shape=(28, 28)),
        layers.Dense(512, activation="leaky_relu"),
        layers.Dense(256, activation="leaky_relu"),
        layers.Dense(1, activation="sigmoid")  # Binary classification
    ])
    return model
```

- Input: 28×28 grayscale image.
- Output: Probability (0 = fake, 1 = real).
- Uses **Leaky ReLU** for better gradient flow.

Step 4: Compile the GAN

```
def compile_gan(generator, discriminator):
    discriminator.compile(loss="binary_crossentropy", optimizer=keras.optimizers.Adam(0.0002, 0.5))

    gan = keras.Sequential([generator, discriminator])
    discriminator.trainable = False  # Freeze Discriminator while training Generator
    gan.compile(loss="binary_crossentropy", optimizer=keras.optimizers.Adam(0.0002, 0.5))

    return gan
```

Step 5: Train the GAN

```
def train_gan(generator, discriminator, gan, epochs=10000, batch_size=128):
    (X_train, _), _ = keras.datasets.mnist.load_data()
    X_train = (X_train / 127.5) - 1  # Normalize images to [-1, 1]

    for epoch in range(epochs):
        # Train Discriminator
        real_images = X_train[np.random.randint(0, X_train.shape[0], batch_size)]
        noise = np.random.normal(0, 1, (batch_size, 100))
        fake_images = generator.predict(noise)

        X = np.concatenate([real_images, fake_images])
        y = np.concatenate([np.ones((batch_size, 1)), np.zeros((batch_size, 1))])

        d_loss = discriminator.train_on_batch(X, y)

        # Train Generator
        noise = np.random.normal(0, 1, (batch_size, 100))
        y_fake = np.ones((batch_size, 1))  # Trick Discriminator
        g_loss = gan.train_on_batch(noise, y_fake)

        # Print progress
        if epoch % 1000 == 0:
            print(f"Epoch {epoch} | D Loss: {d_loss:.4f} | G Loss: {g_loss:.4f}")
```

```
train_gan(build_generator(), build_discriminator(), compile_gan(build_generator(),
build_discriminator()))
```

Step 6: Generate Images

```
def generate_images(generator, n=10):
    noise = np.random.normal(0, 1, (n, 100))
    images = generator.predict(noise)
    images = (images + 1) / 2  # Rescale to [0,1]

    plt.figure(figsize=(10, 2))
    for i in range(n):
        plt.subplot(1, n, i + 1)
        plt.imshow(images[i], cmap="gray")
        plt.axis("off")
    plt.show()

generator = build_generator()
discriminator = build_discriminator()
gan = compile_gan(generator, discriminator)

train_gan(generator, discriminator, gan, epochs=10000, batch_size=128)
generate_images(generator)
```

Conclusion

- **GANs** consist of a Generator and a Discriminator in a competitive setting.
- **Training is challenging**, but best practices like Batch Normalization and Leaky ReLU help.
- **We implemented a simple GAN** to generate handwritten digits using TensorFlow/Keras.

Video resource: https://youtu.be/Fe1MzID2BNg?si=rLX6ytS0hzEi43Ps