Module 6

What is GAN?

GAN is an algorithm that uses two neural networks- Generator G and Discriminator D. The two networks compete against one another (hence the term 'adversarial').

The generator creates synthetic data, while the Discriminator tries to distinguish between the generated data and real data. This leads to creating highly realistic data that can often pass for real.

Over the past few years, GAN has made significant progress. A remarkable example is an AI-generated portrait that sold for $432,000—an artwork produced by a GAN. Impressive, right?

To fully understand GANs, it's crucial to grasp how the Generator and Discriminator work together.

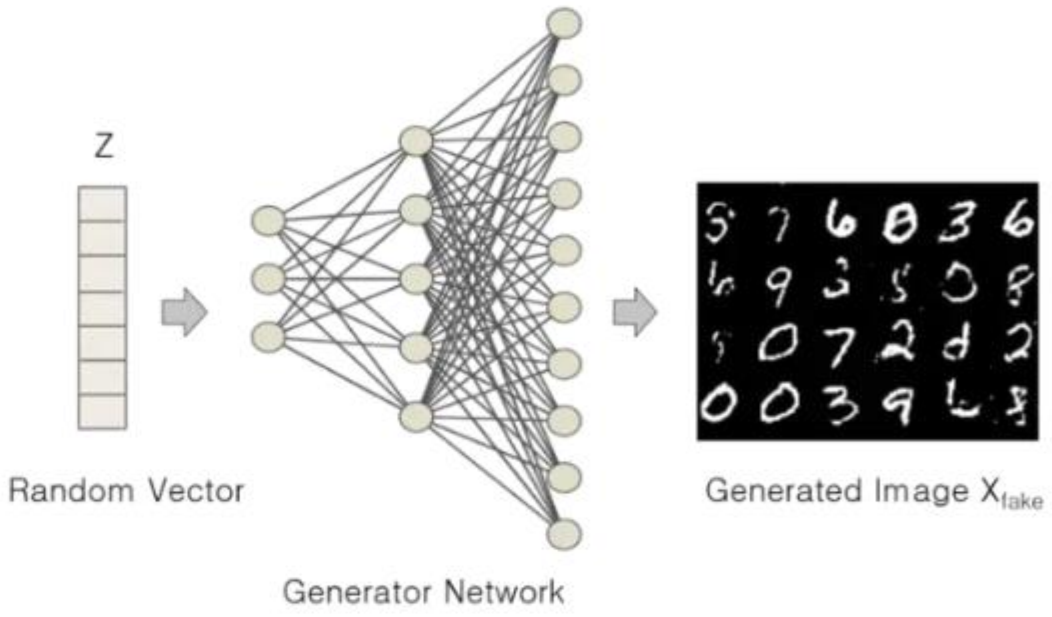
Generator

The Generator (G) in GAN is responsible for producing synthetic samples from random inputs, often just a set of noise or values. It acts as the core of the GAN architecture, gradually learning to create highly realistic images as the training progresses.

For example, if you train the Generator on images of cats, it will compute and generate an image that resembles a cat—though the image is entirely synthetic. Each output will be different, as the Generator takes a new random input (called the noise vector) each time, ensuring the creation of diverse images with every run.

Since the Generator is in constant competition with the Discriminator, it strives to create fake images so realistic that the Discriminator mistakenly classifies them as real.

Module 6

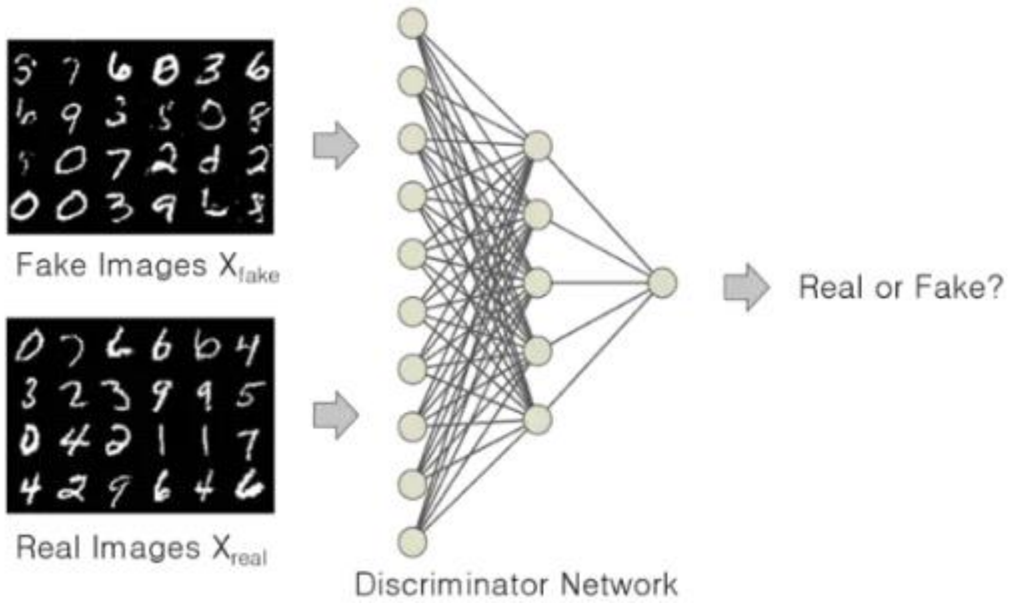


Discriminator

The Discriminator, on the other hand, processes the images created by the Generator and classifies them as either real or fake. It is presented with both the real samples from the training data and the fake image produced by the generator.

Gradually, the Discriminator learns to distinguish between the two samples and offers crucial feedback to the Generator about the quality of the generated samples.

Module 6



A Closer Look at How GAN Works

The constant confrontation between the Generator and Discriminator results in an iterative learning cycle.

As the training progresses, the Generator becomes better at producing realistic samples, while the Discriminator becomes more adept at distinguishing real data from fake. Over time, this adversarial process leads to the Generator creating samples that look increasingly authentic.

Ideally, the training process must reach an equilibrium, where the Generator produces data and the Discriminator can no longer distinguish between real and synthetic data. At this point, the Discriminator's accuracy drops to around 50%, meaning it is essentially guessing whether the input is real or fake.

However, reaching this equilibrium can be challenging. Several factors—like the network's architecture, the choice of hyperparameters, and the complexity of the dataset—can affect training stability. If not balanced properly, the Generator and Discriminator may enter into a loop of oscillating solutions or encounter mode collapse, where the Generator produces limited, repetitive outputs instead of diverse samples.

Module 6

Additionally, if the Discriminator becomes too powerful early in the training process, it can easily reject the Generator's outputs. This hampers the Generator's ability to learn, as it receives little feedback on how to improve its performance in generating realistic data.

Generating Images with GAN

Now that we understand how GANs work, let's move into the details of how to implement GANs to generate images.

STEP 1: Import the necessary libraries

The relevant libraries must first be loaded.

```
import os

import time

import numpy as np

import tensorflow as tf

from tensorflow.keras import layers

import argparse

from IPython import display

import matplotlib.pyplot as plt

# %matplotlib inline

from tensorflow import keras
```

STEP 2: Load the data and conduct data preprocessing

Module 6

```
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.fashion_mnist.load_data()

x_train = x_train.reshape(x_train.shape[0], 28, 28, 1).astype('float32')

x_train = (x_train - 127.5) / 127.5 # Normalize the images to [-1, 1]

# Batch and shuffle the data

train_dataset = tf.data.Dataset.from_tensor_slices(x_train).\

shuffle(60000).batch(args.batch_size)
```

In this example, we load the Fashion MNIST dataset using the 'tf_keras' datasets module. We don't need labels to solve this problem, hence we only make use of the training images, x_train. Next, we reshape the images, and because the data is in unit8 format by default, we cast them to float32.

Our preprocessing also involves normalizing the data from [0, 255] to [-1, 1]. Then we build the TensorFlow input pipeline. Summarily, we feed the 'tf.data.Dataset.from_tensor_slices' with the training data, shuffle, and slice it into tensors which allows us to access tensors of defined batch size during training.

STEP 3: Create the Generator Network

Here, we have fed the generator with a 100-D noise vector which was sampled from a normal distribution. The next thing we do is to define the input layer with the shape (100,). 'he_uniform' is the default weight initializer for the linear layers in Tensor Flow.

Then, we use 'tf.reshape' to reshape the 784-D tensor to batch sizes 28, 28, and 1, the first parameter being the input tensor and the second being the new shape of the tensor.

We finally pass the generator function's input and output layer to create the model.

Module 6

```
def generator(image_dim):
  inputs = keras.Input(shape=(100,), name='input_layer')
  x = layers.Dense(128, kernel_initializer=tf.keras.initializers.he_uniform, name='dense_1')(inputs)
  #print(x.dtype)
  x = layers.LeakyReLU(0.2, name='leaky_relu_1')(x)
  x = layers.Dense(256, kernel_initializer=tf.keras.initializers.he_uniform, name='dense_2')(x)
  x = layers.BatchNormalization(momentum=0.1,  epsilon=0.8, name='bn_1')(x)
  x = layers.LeakyReLU(0.2, name='leaky_relu_2')(x)
  x = layers.Dense(512, kernel_initializer=tf.keras.initializers.he_uniform, name='dense_3')(x)
  x = layers.BatchNormalization(momentum=0.1,  epsilon=0.8, name='bn_2')(x)
  x = layers.LeakyReLU(0.2, name='leaky_relu_3')(x)
  x = layers.Dense(1024, kernel_initializer=tf.keras.initializers.he_uniform,  name='dense_4')(x)
  x = layers.BatchNormalization(momentum=0.1,  epsilon=0.8, name='bn_3')(x)
  x = layers.LeakyReLU(0.2, name='leaky_relu_4')(x)
  x = layers.Dense(image_dim, kernel_initializer=tf.keras.initializers.he_uniform, activation='tanh',
name='dense_5')(x)
  outputs = tf.reshape(x, [-1, 28, 28, 1], name='Reshape_Layer')
  model = tf.keras.Model(inputs, outputs, name="Generator")
  return model
```

STEP 4: Create the discriminator network

Module 6

```
def discriminator():

  inputs = keras.Input(shape=(28,28,1), name='input_layer')

  input = tf.reshape(inputs, [-1, 784], name='reshape_layer')

  x = layers.Dense(512, kernel_initializer=tf.keras.initializers.he_uniform, name='dense_1')(input)

  x = layers.LeakyReLU(0.2, name='leaky_relu_1')(x)

  x = layers.Dense(256, kernel_initializer=tf.keras.initializers.he_uniform, name='dense_2')(x)

  x = layers.LeakyReLU(0.2, name='leaky_relu_2')(x)

  outputs = layers.Dense(1, kernel_initializer=tf.keras.initializers.he_uniform, activation='sigmoid',
name='dense_3') (x)

  model = tf.keras.Model(inputs, outputs, name="Discriminator")

  return model
```

The discriminator is a binary classifier and only has fully connected layers. Because of this, it's only expecting a tensor of shape (Batch Size, 28, 28, 1). However, the discriminator function only has dense layers, which means we have to reshape the tensor to a vector of shape which is Batch Size, 784. You'd see the sigmoid activation function on the last layer, and what this does is produce the output value between 0 (fake) and 1 (real).

STEP 5: Define the loss function

```
binary_cross_entropy = tf.keras.losses.BinaryCrossentropy()
```

This line of code is the binary-cross-entropy loss.

```
def generator_loss(fake_output):

  gen_loss = binary_cross_entropy(tf.ones_like(fake_output), fake_output)

  #print(gen_loss)

  return gen_loss
```

Module 6

This is the generator's individual loss

```
def discriminator_loss(real_output, fake_output):
  real_loss = binary_cross_entropy(tf.ones_like(real_output), real_output)
  fake_loss = binary_cross_entropy(tf.zeros_like(fake_output), fake_output)
  total_loss = real_loss + fake_loss
  #print(total_loss)
  return total_loss
```

And this is the discriminator's loss.

STEP 6: Optimize both the generator and discriminator

```
generator_optimizer = tf.keras.optimizers.Adam(learning_rate = args.lr, beta_1 = args.b1, beta_2 =
args.b2 )
discriminator_optimizer = tf.keras.optimizers.Adam(learning_rate = args.lr, beta_1 = args.b1, beta_2 =
args.b2 )
```

To optimize both the generator and discriminator, we use 'Adam Optimizer,' and this takes two arguments which are the learning rate and beta coefficients.

During backpropagation, these arguments compute the running averages of gradients.

Module 6

```
def train(dataset, epochs):
 for epoch in range(epochs):
  start = time.time()
  i = 0
  D_loss_list, G_loss_list = [], []
  for image_batch in dataset:
   i += 1
   train_step(image_batch)

  display.clear_output(wait=True)
  generate_and_save_images(generator,
                           epoch + 1,
                           seed)
```

Module 6

```
    # Save the model every 15 epochs
    if (epoch + 1) % 15 == 0:
      checkpoint.save(file_prefix = checkpoint_prefix)

    print ('Time for epoch {} is {} sec'.format(epoch + 1, time.time()-start))

  # Generate after the final epoch
  display.clear_output(wait=True)
  generate_and_save_images(generator,
                           epochs,
                           seed)
```

The core of the whole GAN training is the 'train_step' function. This is because we can combine all the training functions as defined above.

What the '@tf.function' does is compile the train_step function into a TensorFlow graph that we can call. Additionally, it reduces training time.

STEP 7: Final Training

Finally, the time has arrived for us to sit and view the magic, but let's pause for a minute. The function above requires two parameters (training data and number of epochs). When you give it those parameters, you can then proceed to run the program and watch as GAN does its magic.