Generative Adversarial Network (GAN)

Introduction

In recent years, the field of computer science and data generation has seen rapid advancements. Artificial Intelligence (AI) has emerged as a thriving domain, offering numerous meaningful applications and research opportunities. Within the AI community, machine learning plays an important role, influencing various aspects of our daily lives. However, all machine learning algorithms require an effective representation of the input data. When attempting to apply these algorithms to different tasks or fields, it often becomes challenging to extract relevant features manually.

Researchers introduced a new approach, representation learning, to address this issue by automating the extraction of useful features for tasks such as classification and detection. One prominent type is deep learning, which excels at extracting high-level, abstract features by combining simpler representations in multiple layers.

Categories of Machine Learning

Machine learning algorithms are classified into two categories based on whether the dataset contains labelled data:

1. **Supervised Learning**:
   - It requires a dataset with labelled examples.
   - Common tasks include classification, regression, and structured output problems.
2. **Unsupervised Learning**:
   - Involves datasets without labels.
   - The goal is to discover underlying structures in the data.
   - Common tasks include density estimation, clustering, synthesis, and denoising.

Collecting and annotating label data for supervised learning will be time-consuming and difficult. As a result, researchers have focused on unsupervised learning. In unsupervised learning, one of the most promising approaches is the generative model. These models learn the data distribution by fitting a large set of parameters to match the empirical distribution of the training data.

Traditional generative models often rely on techniques like Markov chains, maximum likelihood estimation, and approximate inference.

Generative Adversarial Networks (GANs)

In 2014, **Goodfellow et al.** introduced a generative model known as **Generative Adversarial Networks (GANs).** They are based on game theory and consist of two competing neural networks. one is a generator, and the other is a discriminator. The generator’s goal is to produce realistic data that can fool the discriminator, while the discriminator aims to correctly identify fake data. Techniques like backpropagation and dropout algorithms train both networks simultaneously. Unlike traditional models, GANs do not require approximate inference or Markov chains, making them more efficient and versatile.

GANs have since become a cornerstone of generative modelling, enabling advances in image synthesis, data augmentation, and other AI applications.

Architecture of GANs

As we have discussed, GANs will have two parts: one is a generator, and the other is a discriminator.

Generator:

- The generator's role is to create data that appears realistic.
- The generated data serves as negative (fake) examples for the discriminator, helping the generator improve over time.

Discriminator:

- The discriminator's role is to differentiate between real data and the data generated by the generator.
- The discriminator provides feedback by penalizing the generator when it produces unrealistic or implausible results.

Let's look at an example of how a generator and a discriminator interact to create realistic images.

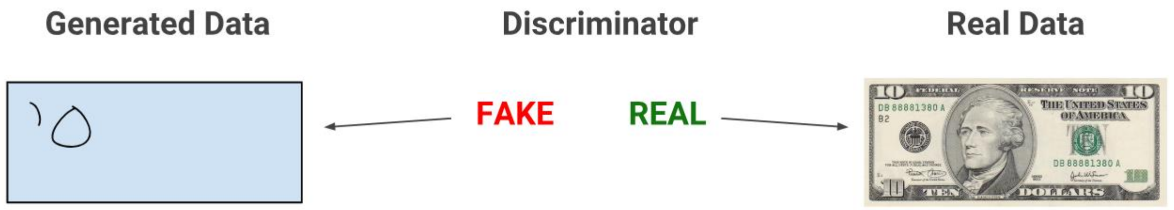


Figure 1: When training begins, the generator produces obviously fake data, and the discriminator quickly learns to tell that it's fake.

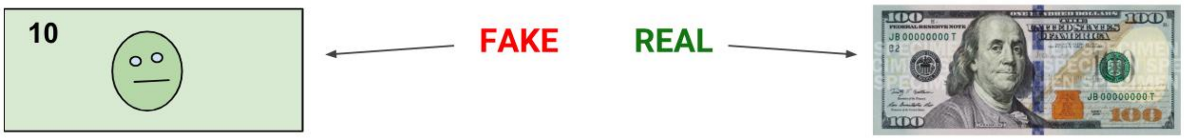


Figure 2: As training progresses, the generator gets closer to producing output that can fool the discriminator.

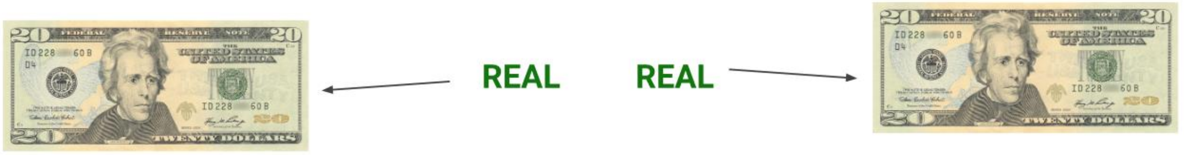


Figure 3: Finally, if generator training goes well, the discriminator gets worse at telling the difference between real and fake. It starts to classify fake data as real, and its accuracy decreases.

Here is a picture of the entire GAN system:

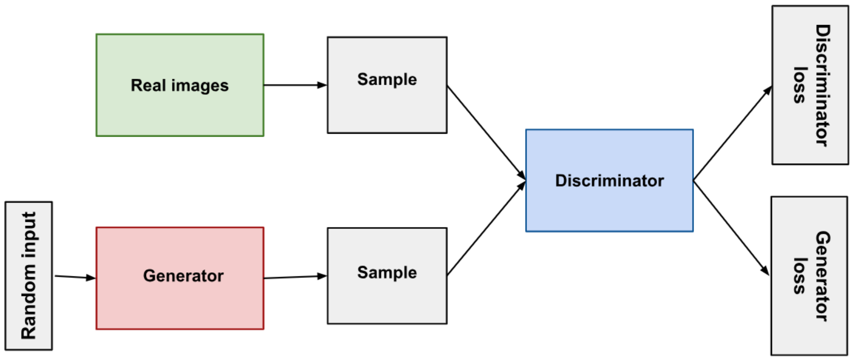


Figure 4: Architecture of GAN

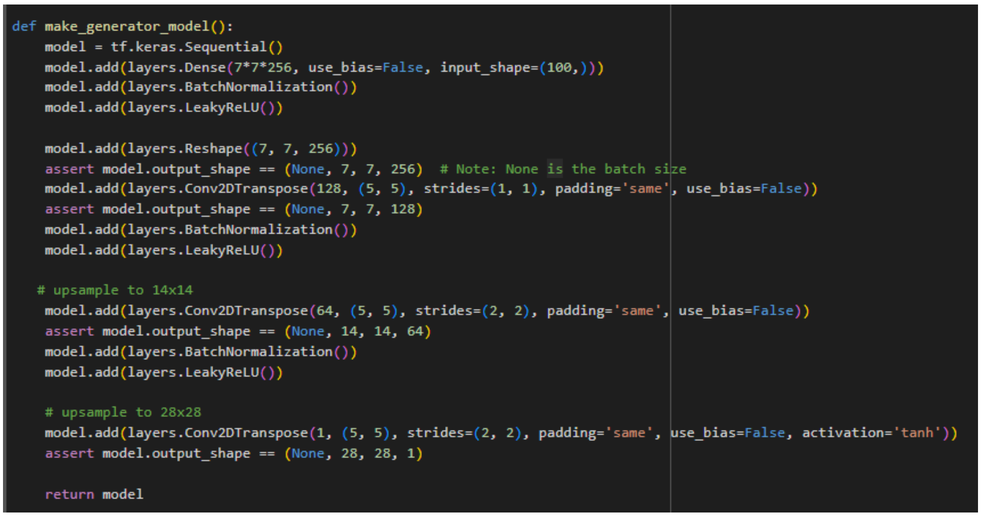
Both the generator and the discriminator are neural networks. The generator output is connected directly to the discriminator input. Through backpropagation, the discriminator's classification provides a signal that the generator uses to update its weights.

Generator

Generator is responsible for creating synthetic data that closely resembles real-world data. The generator learns by receiving feedback from the discriminator, which evaluates whether the generated data looks realistic. The goal of the generator is to improve to the point where the discriminator can no longer distinguish between real and generated data.

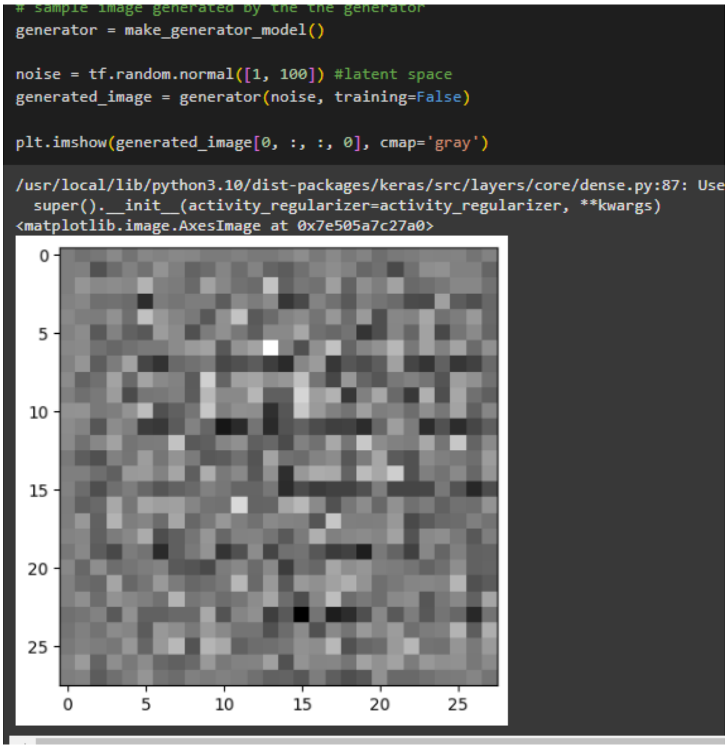
Let's explore how the Generator and Discriminator work and train together using real-world data, specifically the MNIST dataset (a collection of handwritten digits from 0 to 9). Our first step is to create a generator model to produce a random image from noise

Code snippet for Generator Model:



```
def make_generator_model():
    model = tf.keras.Sequential()
    model.add(layers.Dense(7*7*256, use_bias=False, input_shape=(100,)))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Reshape((7, 7, 256)))
    assert model.output_shape == (None, 7, 7, 256)  # Note: None is the batch size
    model.add(layers.Conv2DTranspose(128, (5, 5), strides=(1, 1), padding='same', use_bias=False))
    assert model.output_shape == (None, 7, 7, 128)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

   # upsample to 14x14
    model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same', use_bias=False))
    assert model.output_shape == (None, 14, 14, 64)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    # upsample to 28x28
    model.add(layers.Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same', use_bias=False, activation='tanh'))
    assert model.output_shape == (None, 28, 28, 1)

    return model
```

Code snippet for creating the image with random noise:



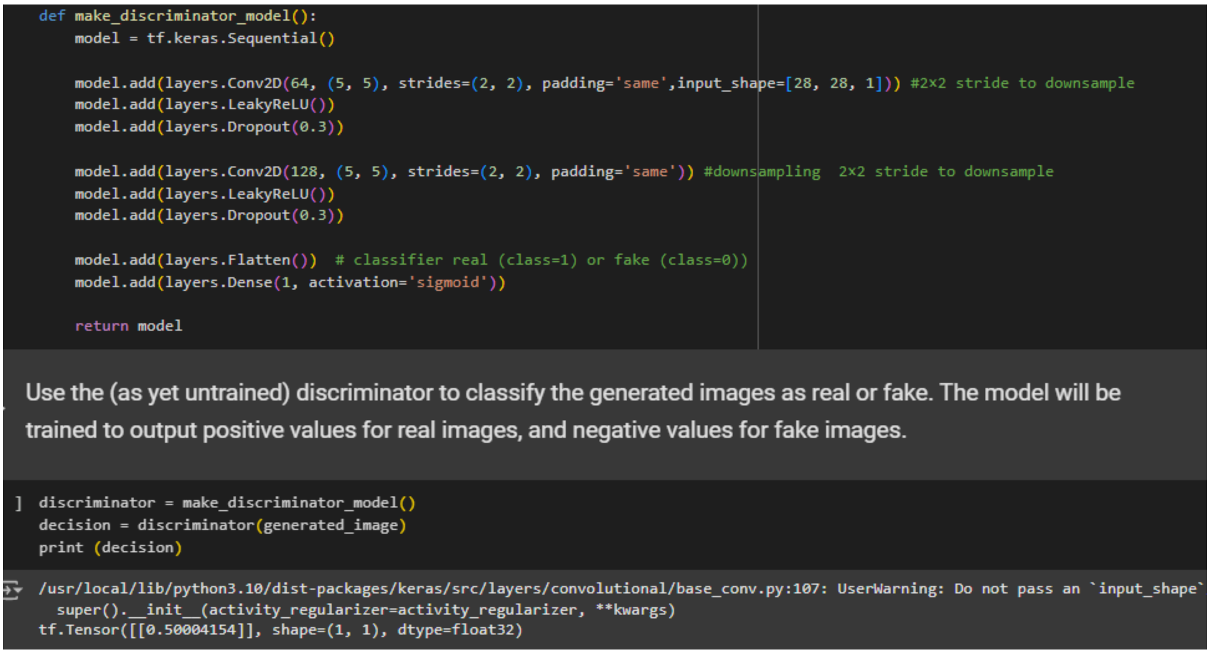
```
# sample image generated by the the generator
generator = make_generator_model()

noise = tf.random.normal([1, 100]) #latent space
generated_image = generator(noise, training=False)

plt.imshow(generated_image[0, :, :, 0], cmap='gray')
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:87: Use
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
<matplotlib.image.AxesImage at 0x7e505a7c27a0>
```

Discriminator

Discriminator acts as a classifier that evaluates whether the input data is real or generated (fake). It plays a critical role in improving the generator by providing feedback. Its goal is to accurately distinguish between real data (from the actual dataset) and fake data (produced by the generator).

Let’s go back to our code and create a discriminator model to classify of the generated image is real or fake



```
def make_discriminator_model():
    model = tf.keras.Sequential()

    model.add(layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same',input_shape=[28, 28, 1])) #2x2 stride to downsample
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

    model.add(layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same')) #downsampling  2x2 stride to downsample
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

    model.add(layers.Flatten())  # classifier real (class=1) or fake (class=0))
    model.add(layers.Dense(1, activation='sigmoid'))

    return model
```

Use the (as yet untrained) discriminator to classify the generated images as real or fake. The model will be trained to output positive values for real images, and negative values for fake images.

```
] discriminator = make_discriminator_model()
  decision = discriminator(generated_image)
  print (decision)
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/convolutional/base_conv.py:107: UserWarning: Do not pass an `input_shape`
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
tf.Tensor([[0.50004154]], shape=(1, 1), dtype=float32)
```

Min Max Loss Function

A loss function is a mathematical function used to measure the difference between predicted values and actual values in a model. It helps guide the learning process by providing feedback on how well the model is performing, allowing it to adjust its parameters to minimize errors.

In GANs, the loss function is essential for balancing the competition between the generator and discriminator, ensuring the generator produces data that closely mimics real data.

The minimax loss function in GANs reflects the adversarial nature of the training process, where the generator and discriminator are in competition. This loss function reflects the distance between the distribution of the data generated by the GAN and the distribution of the real data.

The generator tries to minimize the following function while the discriminator tries to maximize it:

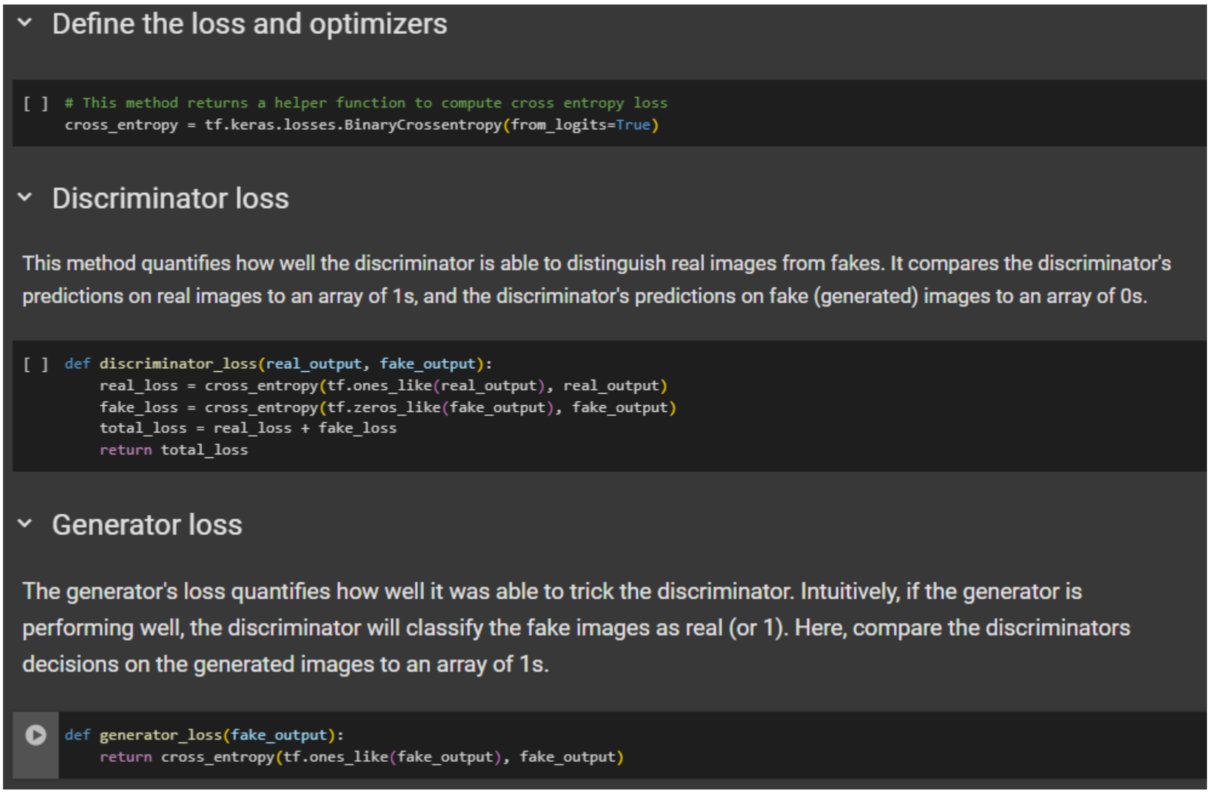
$$E_x[log(D(x))] + E_z[log(1 - D(G(z)))]$$

In this function:

- D(x) is the discriminator's estimate of the probability that real data instance x is real.
- E_x is the expected value over all real data instances.
- G(z) is the generator's output when given noise z.
- D(G(z)) is the discriminator's estimate of the probability that a fake instance is real.
- E_z is the expected value over all random inputs to the generator (in effect, the expected value over all generated fake instances G(z)).
- The formula derives from the cross entropy between the real and generated distributions.

The generator can't directly affect the log(D(x)) term in the function, so, for the generator, minimizing the loss is equivalent to minimizing log(1 - D(G(z))).

In practice, these objectives are implemented as follows using **binary cross-entropy loss**



Define the loss and optimizers

```
# This method returns a helper function to compute cross entropy loss
cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)
```

Discriminator loss

This method quantifies how well the discriminator is able to distinguish real images from fakes. It compares the discriminator's predictions on real images to an array of 1s, and the discriminator's predictions on fake (generated) images to an array of 0s.

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

Generator loss

The generator's loss quantifies how well it was able to trick the discriminator. Intuitively, if the generator is performing well, the discriminator will classify the fake images as real (or 1). Here, compare the discriminators decisions on the generated images to an array of 1s.

```
def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output), fake_output)
```

Training GAN

GAN training proceeds in alternating periods:

1. The discriminator trains for one or more epochs.
2. The generator trains for one or more epochs.
3. Repeat steps 1 and 2 to continue to train the generator and discriminator networks.

We keep the generator constant during the discriminator training phase. As discriminator training tries to figure out how to distinguish real data from fake, it must learn how to recognize the generator's flaws. That's a different problem for a thoroughly trained generator than it is for an untrained generator that produces random output.

Similarly, we keep the discriminator constant during the generator training phase. Otherwise, the generator would be trying to hit a moving target and might never converge.

Generator Training Process Flow

1. **Generate Data**:

   The generator takes a random noise vector as input and produces a synthetic data instance.

2. **Discriminator Evaluation**:

   The generated data is passed to the discriminator, which returns a probability of whether the data is real or fake.

3. **Compute Generator Loss**:

   The generator's loss is calculated based on the discriminator's output.

   The goal is to reduce this loss by making the discriminator believe the generated data is real.

4. **Backpropagation**:

   The generator uses **backpropagation** and an **optimizer** (e.g., Adam) to update its weights and improve its ability to produce realistic data.

5. **Iterate**:

   This process is repeated for many iterations, gradually improving the generator's outputs.

Discriminator Training Process

1. **Input**: The discriminator takes both real and fake data as inputs.

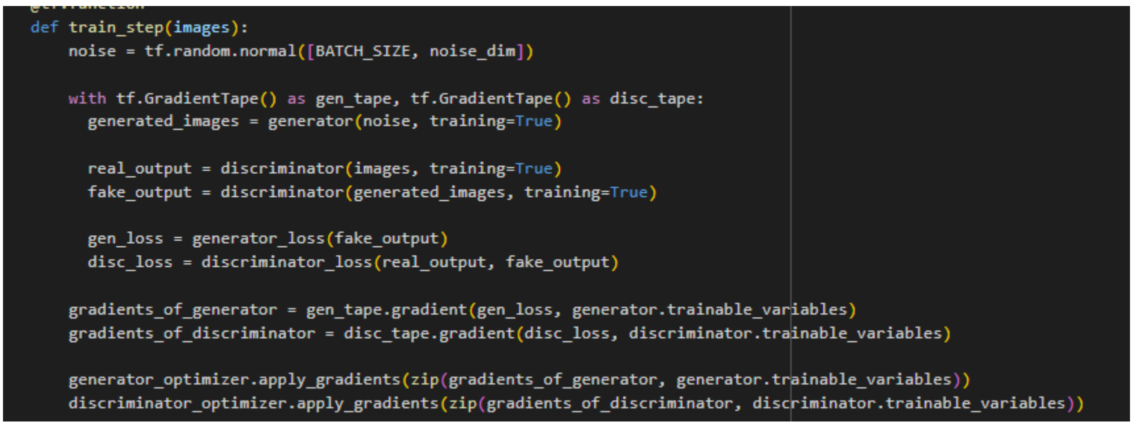
2. **Output**: It outputs a probability indicating whether the input is real or fake (values close to 1 for real, close to 0 for fake).

3. **Loss Calculation**: The discriminator computes a loss based on how well it classifies the real and fake data correctly.

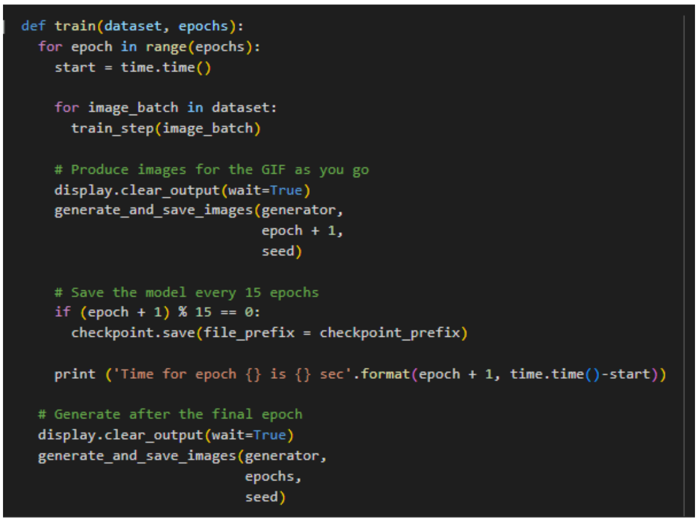
4. **Backpropagation**: The discriminator updates its weights through backpropagation to improve its classification accuracy.

By training on both real and fake data, the discriminator learns to become a better judge, which in turn forces the generator to create increasingly realistic data. This adversarial process drives both networks to improve over time.

Let’s try to create a training loop that begins with generator receiving a random seed as input. That seed is used to produce an image, then discriminator is used to classify images. The loss is calculated for each of these models. Later, gradients are used to update the generator and discriminator.

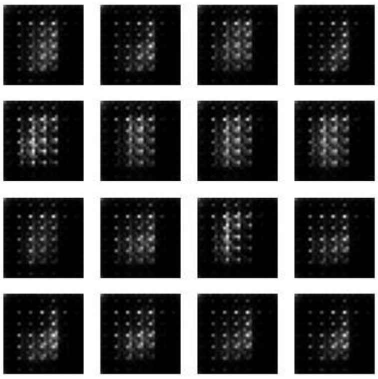


```
@tf.function
def train_step(images):
  noise = tf.random.normal([BATCH_SIZE, noise_dim])

  with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
   generated_images = generator(noise, training=True)

   real_output = discriminator(images, training=True)
   fake_output = discriminator(generated_images, training=True)

   gen_loss = generator_loss(fake_output)
   disc_loss = discriminator_loss(real_output, fake_output)

  gradients_of_generator = gen_tape.gradient(gen_loss, generator.trainable_variables)
  gradients_of_discriminator = disc_tape.gradient(disc_loss, discriminator.trainable_variables)

  generator_optimizer.apply_gradients(zip(gradients_of_generator, generator.trainable_variables))
  discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator, discriminator.trainable_variables))
```



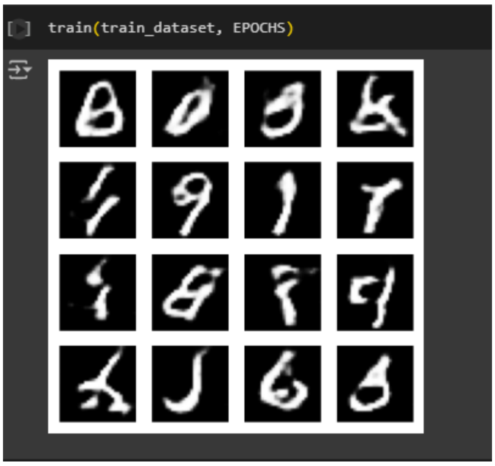
```
def train(dataset, epochs):
  for epoch in range(epochs):
    start = time.time()

    for image_batch in dataset:
      train_step(image_batch)

    # Produce images for the GIF as you go
    display.clear_output(wait=True)
    generate_and_save_images(generator,
                             epoch + 1,
                             seed)

    # 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)
```

At the beginning of the training, the generated images look like random noise.



As training progresses, the generated digits will look increasingly real.

After about 50 epochs, they resemble MNIST digits.



```
train(train_dataset, EPOCHS)
```

Applications of GANs

Researchers continue to find improved GAN techniques and new uses for GANs. Below are some of the applictaions:

Image-to-Image Translation

Image-to-Image translation GANs take an image as input and map it to a generated output image with different properties. For example, we can take a mask image with blob of color in the shape of a car, and the GAN can fill in the shape with photorealistic car details.

Cycle GAN

Cycle GANs learn to transform images from one set into images that could plausibly belong to another set.

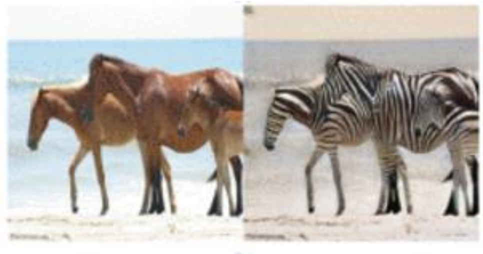


Figure 5 : A CycleGAN produced the righthand image below when given the lefthand image as input. It took an image of a horse and turned it into an image of a zebra.

Super-resolution

Super-resolution GANs increase the resolution of images, adding detail where necessary to fill in blurry areas.

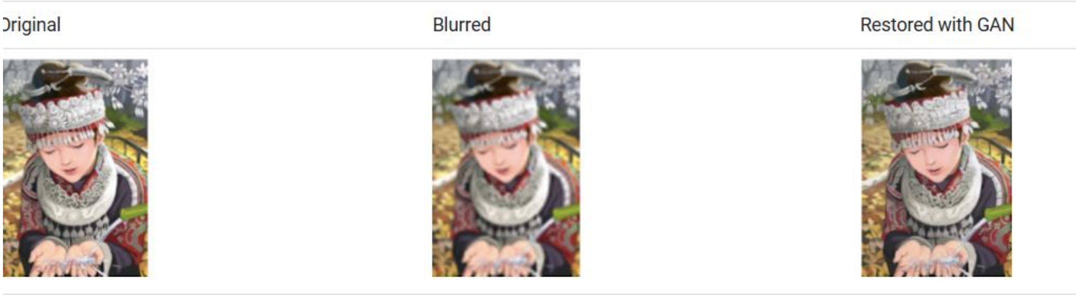


Figure 6 : The blurry middle image below is a down sampled version of the original image on the left. Given the blurry image, a GAN produced the sharper image on the right.

Face Inpainting

GANs have been used for the *semantic image inpainting* task. In the inpainting task, chunks of an image are blacked out, and the system tries to fill in the missing chunks.

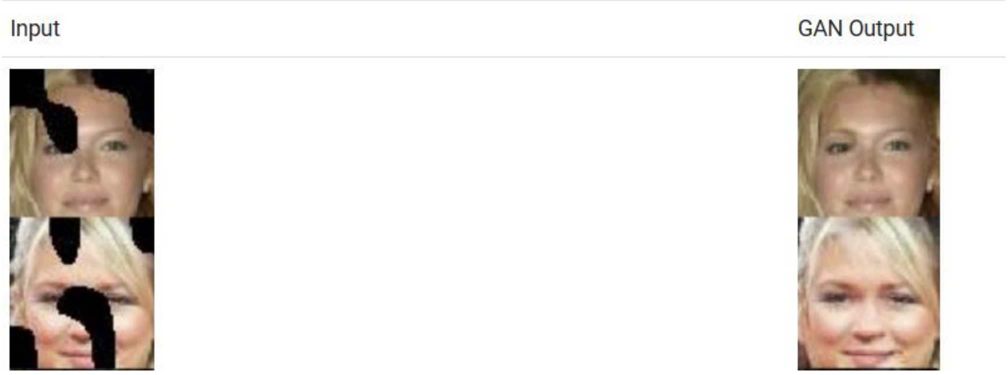


Figure 7 : Gan generated images for blacked out images.

Limitations

Generator Fails to Create Meaningful Images

- In the early stages of training, the generator may struggle to create coherent or recognizable images.
- This can happen if the generator's network is not sufficiently trained or if the feedback from the discriminator is inconsistent.

Convergence Issues

- As the generator improves with training, the discriminator performance gets worse because the discriminator can't easily tell the difference between real and fake. If the generator succeeds perfectly, then the discriminator has a 50% accuracy. In effect, the discriminator makes random predictions.
- This progression poses a problem for convergence of the GAN as a whole: the discriminator feedback gets less meaningful over time. If the GAN continues training past the point when the discriminator is giving completely random feedback, then the generator starts to train on junk feedback, and its own quality may collapse.

Summary

GANs have revolutionized generative modelling, enabling significant advances in fields like image synthesis, data augmentation, and style transfer. Despite their limitations, ongoing research continues to enhance GAN techniques and explore new applications.

References

Research papers used in the tutorial:

Generative Adversarial Nets by Ian J. Goodfellow and team

Recent Progress on Generative Adversarial Networks (GANs) by Zhaoqing Pan and team

GitHub link for code and images generated:

https://github.com/pratapponnam/Machinelearning

Author: Pratap Bhargav Ponnam

ID:23027654