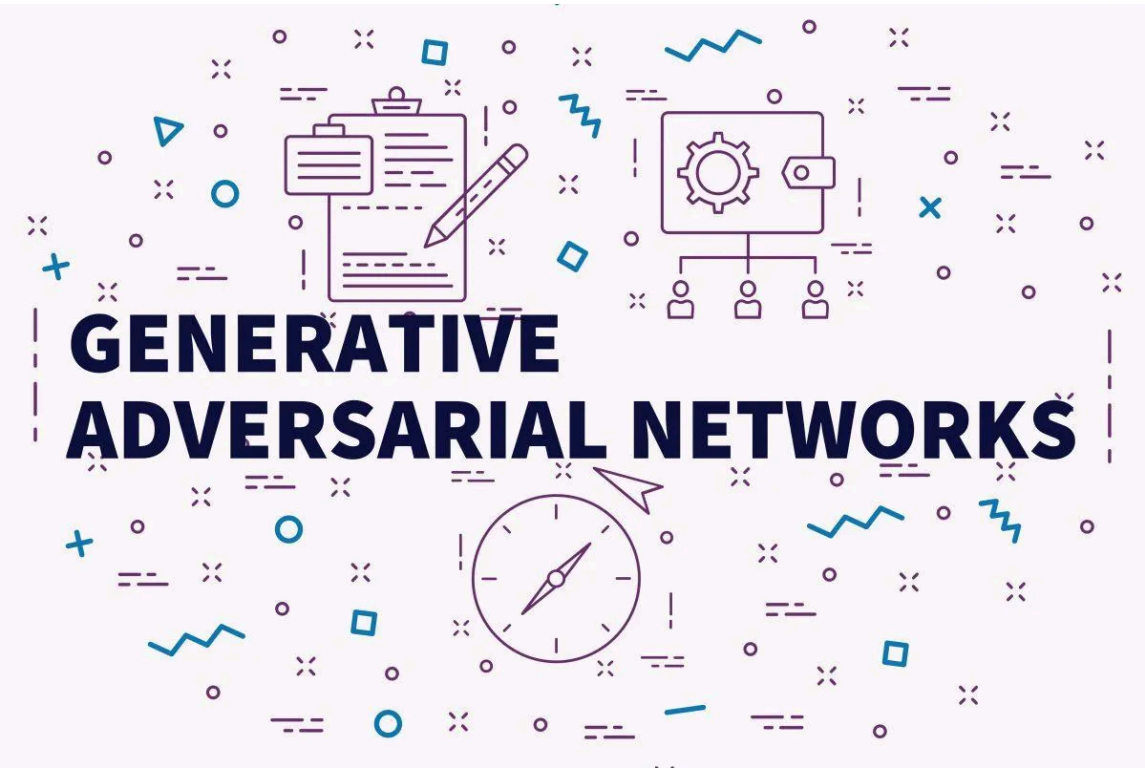


GENERATIVE ADVERSARIAL NETWORKS With PyTorch:

Introduction to GANs

What are GANs?

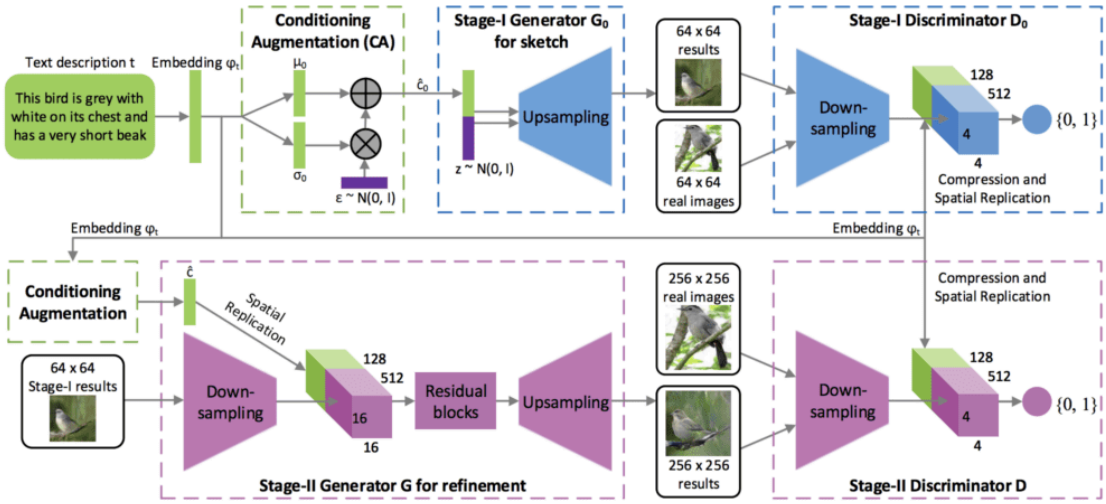
Generative Adversarial Networks (GANs) are a type of machine learning framework where two neural networks, the Generator and the Discriminator, are trained simultaneously through adversarial processes. The Generator creates fake data samples, while the Discriminator evaluates them against real data samples to distinguish the fake from the real.

Applications of GANs

GANs have a wide range of applications including, but not limited to:

- **Image Generation**: Creating realistic images, enhancing image resolution, and generating art.
- **Image-to-Image Translation**: Tasks like image colorization, super-resolution, and style transfer.
- **Video Generation**: Creating realistic video sequences from images or text.
- **Data Augmentation**: Generating synthetic data to augment training datasets.
- **Healthcare**: Enhancing medical imaging and aiding in drug discovery.
- **Entertainment**: Generating realistic textures and environments in video games, and creating virtual reality content.

THE GAN ARCHITECTURE



The architecture of a Generative Adversarial Network (GAN) consists of two primary components: the Generator and the Discriminator. These two neural networks are trained simultaneously through an adversarial process, where the Generator aims to produce realistic data, and the Discriminator aims to distinguish between real and fake data.

Generator

The Generator is a neural network that takes random noise as input and generates data that mimics the real data. The goal of the Generator is to create outputs that are indistinguishable from real data samples to the Discriminator. The Generator's architecture typically includes:

- **Input Layer**: Takes a random noise vector (e.g., sampled from a normal distribution) as input.
- **Hidden Layers**: Series of fully connected (or sometimes convolutional) layers with activation functions such as ReLU. These layers learn to transform the noise into a more structured output.
- **Output Layer**: Produces data of the same shape as the real data, often using a Tanh activation function to ensure the output values are in a specific range.

Example of Generator network class in Pytorch:

```
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.main = nn.Sequential(
            nn.Linear(100, 256),
            nn.ReLU(True),
            nn.Linear(256, 512),
            nn.ReLU(True),
            nn.Linear(512, 1024),
            nn.ReLU(True),
            nn.Linear(1024, 28*28),
            nn.Tanh()
        )
```

```
    def forward(self, x):
        return self.main(x).view(-1, 1, 28, 28)
```

Discriminator

The Discriminator is a neural network that takes both real data and fake data (generated by the Generator) as input and tries to classify them as real or fake. The goal of the Discriminator is to correctly identify real versus fake data samples. The Discriminator's architecture typically includes:

- **Input Layer**: Takes a data sample (e.g., an image) as input.
- **Hidden Layers**: Series of fully connected (or sometimes convolutional) layers with activation functions such as Leaky ReLU. These layers learn to extract features that help in distinguishing real data from fake data.
- **Output Layer**: Produces a single value (often using a Sigmoid activation function) representing the probability that the input data is real.

Example of the Discriminator network in Pytorch:

```
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.main = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(256, 1),
            nn.Sigmoid()
        )

    def forward(self, x):
        return self.main(x.view(-1, 28*28))
```

Adversarial Training

The training process of GANs is adversarial in nature, where the Generator and Discriminator compete against each other:

- **Generator's Objective**: Generate data that is realistic enough to fool the Discriminator. It aims to minimize the Discriminator's ability to correctly classify fake data as fake.
- **Discriminator's Objective**: Correctly classify real data as real and fake data as fake. It aims to maximize its classification accuracy.

This adversarial process can be summarized by the following loss functions:

- **Generator Loss**: Measures how well the Generator is able to fool the Discriminator. It is typically the negative log probability of the Discriminator predicting the fake data as real.
- **Discriminator Loss**: Measures how well the Discriminator is able to distinguish between real and fake data. It is the sum of the log probabilities of correctly classifying real data and fake data.

The following code snippets define the loss functions and optimizers used for training the GAN:

```
criterion = nn.BCELoss()

optimizer_g = optim.Adam(generator.parameters(), lr=0.0002)
optimizer_d = optim.Adam(discriminator.parameters(), lr=0.0002)
```

During training, both networks are updated in a loop until the Generator produces realistic data that the Discriminator cannot easily distinguish from real data. This adversarial process leads to the Generator learning to produce increasingly realistic data samples over time.

This architecture and training process form the foundation of Generative Adversarial Networks, enabling them to generate high-quality synthetic data across various applications.

PROJECT WORKSPACE

In [ ]:
```
#Installing some dependencies
!pip install torch torchvision matplotlib
```

In [7]:
```
# Doing some basic imports
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.transforms as transforms
import torchvision.datasets as datasets # torchvision for computervision ,
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt # For data visualization
```

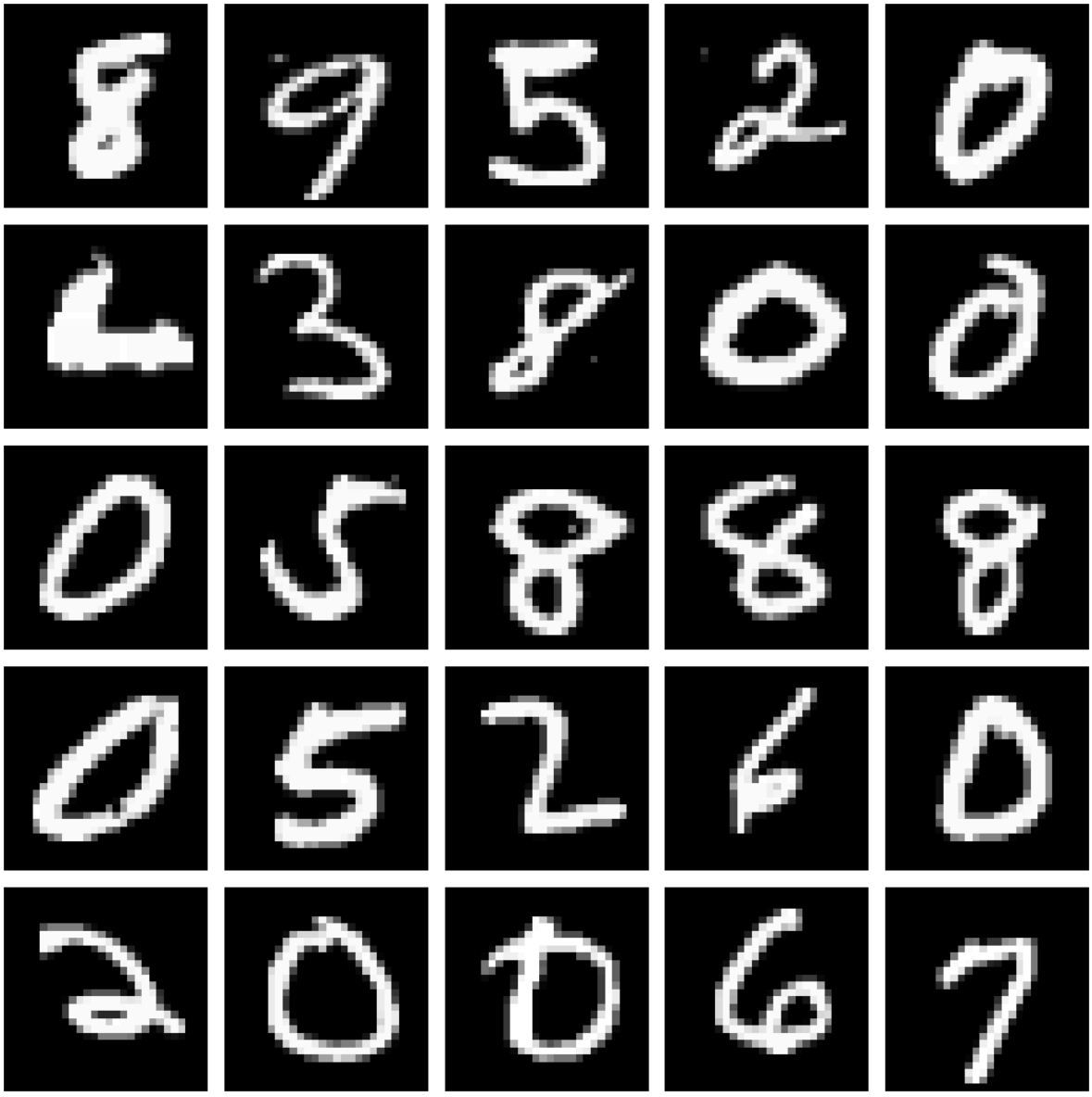
DATA PREPARATION

For this project, I'll use the MNIST dataset, a collection of 70,000 images of handwritten digits.

In [8]:
```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

dataset = datasets.MNIST(root='data', train=True, transform=transform, download=Tru
dataloader = DataLoader(dataset, batch_size=64, shuffle=True)
```

In [11]:
```
import numpy as np
# Plotting  some images from the dataset
def show_images(dataset, num_images=25):
    data_loader = DataLoader(dataset, batch_size=num_images, shuffle=True)
    images, labels = next(iter(data_loader))
    images = images.numpy()

    fig, axes = plt.subplots(5, 5, figsize=(10, 10))
    axes = axes.flatten()
    for img, ax in zip(images, axes):
        ax.imshow(np.squeeze(img), cmap='gray')
        ax.axis('off')
    plt.tight_layout()
```

```
    plt.show()

show_images(dataset)
```



PYTORCH CODE IMPLEMENTATION

MODEL BUILDING

In [ ]:
```
# Building a generator and discriminator using nn.Module package


class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.main = nn.Sequential(
            nn.Linear(100, 256),
            nn.ReLU(True),
            nn.Linear(256, 512),
            nn.ReLU(True),
            nn.Linear(512, 1024),
            nn.ReLU(True),
            nn.Linear(1024, 28*28),
```

```
            nn.Tanh()
        )

    def forward(self, x):
        return self.main(x).view(-1, 1, 28, 28)

class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.main = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(256, 1),
            nn.Sigmoid()
        )

    def forward(self, x):
        return self.main(x.view(-1, 28*28))
```

MODEL TRAINING

In [ ]:
```
generator = Generator()
discriminator = Discriminator()

# Using Adam Optimizer and BinaryCrossEntropy Loss
criterion = nn.BCELoss()
optimizer_g = optim.Adam(generator.parameters(), lr=0.0002)
optimizer_d = optim.Adam(discriminator.parameters(), lr=0.0002)
```

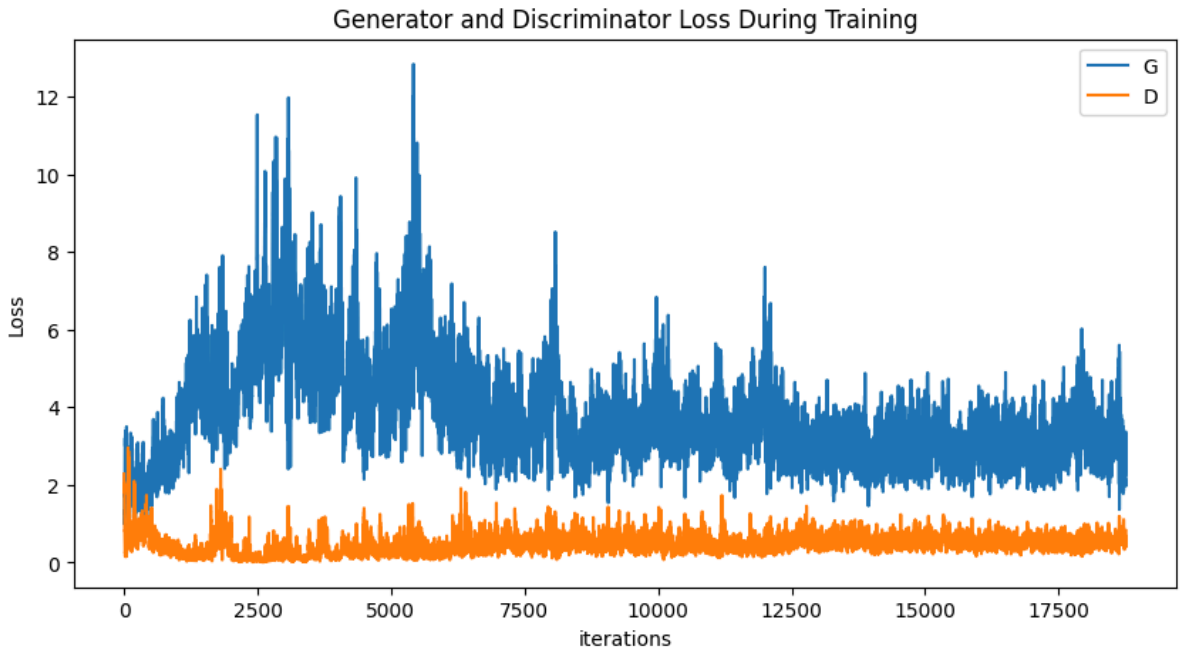
In [14]:
```
# Initialize lists to store the loss values
# This will enable us plot the losses later on
d_losses = []
g_losses = []

num_epochs = 20
for epoch in range(num_epochs):
    for i, (real_images, _) in enumerate(dataloader):
        batch_size = real_images.size(0)

        # Train Discriminator
        real_labels = torch.ones(batch_size, 1)
        fake_labels = torch.zeros(batch_size, 1)

        outputs = discriminator(real_images)
        d_loss_real = criterion(outputs, real_labels)
        real_score = outputs

        z = torch.randn(batch_size, 100)
        fake_images = generator(z)
        outputs = discriminator(fake_images.detach())
        d_loss_fake = criterion(outputs, fake_labels)
        fake_score = outputs

        d_loss = d_loss_real + d_loss_fake
        optimizer_d.zero_grad()
        d_loss.backward()
        optimizer_d.step()

        # Train Generator
```

```
        outputs = discriminator(fake_images)
        g_loss = criterion(outputs, real_labels)

        optimizer_g.zero_grad()
        g_loss.backward()
        optimizer_g.step()

        # Record the losses
        d_losses.append(d_loss.item())
        g_losses.append(g_loss.item())

    print(f'Epoch [{epoch}/{num_epochs}], d_loss: {d_loss.item():.4f}, g_loss: {g_l
          f'D(x): {real_score.mean().item():.2f}, D(G(z)): {fake_score.mean().item(
```

```
Epoch [0/20], d_loss: 0.2938, g_loss: 3.2088, D(x): 0.87, D(G(z)): 0.11
Epoch [1/20], d_loss: 0.6490, g_loss: 4.5835, D(x): 0.86, D(G(z)): 0.19
Epoch [2/20], d_loss: 0.5929, g_loss: 9.6355, D(x): 0.88, D(G(z)): 0.01
Epoch [3/20], d_loss: 0.3569, g_loss: 5.7651, D(x): 0.87, D(G(z)): 0.08
Epoch [4/20], d_loss: 0.2780, g_loss: 5.4575, D(x): 0.93, D(G(z)): 0.09
Epoch [5/20], d_loss: 0.2873, g_loss: 5.6587, D(x): 0.91, D(G(z)): 0.12
Epoch [6/20], d_loss: 0.4554, g_loss: 3.9616, D(x): 0.86, D(G(z)): 0.09
Epoch [7/20], d_loss: 0.7824, g_loss: 2.8026, D(x): 0.75, D(G(z)): 0.11
Epoch [8/20], d_loss: 1.2747, g_loss: 1.8294, D(x): 0.70, D(G(z)): 0.24
Epoch [9/20], d_loss: 0.7637, g_loss: 3.2661, D(x): 0.72, D(G(z)): 0.10
Epoch [10/20], d_loss: 0.2501, g_loss: 3.2513, D(x): 0.92, D(G(z)): 0.12
Epoch [11/20], d_loss: 0.3131, g_loss: 2.6590, D(x): 0.91, D(G(z)): 0.17
Epoch [12/20], d_loss: 0.2366, g_loss: 2.9754, D(x): 0.96, D(G(z)): 0.16
Epoch [13/20], d_loss: 0.4572, g_loss: 3.3137, D(x): 0.85, D(G(z)): 0.12
Epoch [14/20], d_loss: 0.4166, g_loss: 3.7680, D(x): 0.84, D(G(z)): 0.08
Epoch [15/20], d_loss: 0.4858, g_loss: 2.6184, D(x): 0.84, D(G(z)): 0.15
Epoch [16/20], d_loss: 0.2739, g_loss: 3.0489, D(x): 0.90, D(G(z)): 0.11
Epoch [17/20], d_loss: 0.7272, g_loss: 2.8237, D(x): 0.81, D(G(z)): 0.22
Epoch [18/20], d_loss: 0.2626, g_loss: 3.0699, D(x): 0.96, D(G(z)): 0.17
Epoch [19/20], d_loss: 0.8250, g_loss: 2.1163, D(x): 0.91, D(G(z)): 0.43
```

In [15]:

```
# Plotting  the loss curves
def plot_losses(d_losses, g_losses):
    plt.figure(figsize=(10, 5))
    plt.title("Generator and Discriminator Loss During Training")
    plt.plot(g_losses, label="G")
    plt.plot(d_losses, label="D")
    plt.xlabel("iterations")
    plt.ylabel("Loss")
    plt.legend()
    plt.show()

plot_losses(d_losses, g_losses)
```



MODEL EVALUATION

In [18]:
```
# Here I get images at random from my testing set
def generate_and_save_images(model, epoch, test_input):
    predictions = model(test_input).detach().cpu().numpy()
    fig = plt.figure(figsize=(4, 4))

    for i in range(predictions.shape[0]):
        plt.subplot(4, 4, i+1)
        plt.imshow(predictions[i, 0], cmap='gray')
        plt.axis('off')

    plt.savefig('image_at_epoch_{:04d}.png'.format(epoch))
    plt.show()

test_input = torch.randn(16, 100)
generate_and_save_images(generator, num_epochs, test_input)
```

SAVING AND LOADING THE MODEL

In [19]:
```
torch.save(generator.state_dict(), 'generator.pth')
torch.save(discriminator.state_dict(), 'discriminator.pth')

# Loading the models
generator.load_state_dict(torch.load('generator.pth'))
discriminator.load_state_dict(torch.load('discriminator.pth'))
```

Out[19]:
```
<All keys matched successfully>
```

In [ ]:
```

```