Here's a complete implementation of a **Vanilla GAN in PyTorch** to generate MNIST digits, structured as a tutorial with explanations:

```
# File: 01_GAN_MNIST_Digit_Generation.py
# Topic: Vanilla GAN implementation for MNIST digit generation using
PyTorch

import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import numpy as np

# Set random seed for reproducibility
torch.manual_seed(42)
np.random.seed(42)

# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Hyperparameters
latent_dim = 100
batch_size = 64
epochs = 50
learning_rate = 0.0002

# MNIST dataset
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=(0.5,), std=(0.5,))  # Scale to [-1, 1]
])

train_dataset = torchvision.datasets.MNIST(
    root='./data', train=True, transform=transform, download=True)
train_loader = torch.utils.data.DataLoader(
    dataset=train_dataset, batch_size=batch_size, shuffle=True)

# Generator model
class Generator(nn.Module):
    def __init__(self, latent_dim):
        super(Generator, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(latent_dim, 256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, 512),
            nn.LeakyReLU(0.2),
            nn.Linear(512, 1024),
            nn.LeakyReLU(0.2),
            nn.Linear(1024, 28*28),
```

PROF

```
            nn.Tanh()  # Output in [-1, 1] to match input normalization
        )

    def forward(self, z):
        img = self.model(z)
        img = img.view(img.size(0), 1, 28, 28)
        return img

# Discriminator model
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(28*28, 1024),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.3),
            nn.Linear(1024, 512),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.3),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.3),
            nn.Linear(256, 1),
            nn.Sigmoid()  # Probability that input is real
        )

    def forward(self, img):
        flattened = img.view(img.size(0), -1)
        validity = self.model(flattened)
        return validity

# Initialize models
generator = Generator(latent_dim).to(device)
discriminator = Discriminator().to(device)

# Loss function and optimizers
adversarial_loss = nn.BCELoss()
optimizer_G = optim.Adam(generator.parameters(), lr=learning_rate)
optimizer_D = optim.Adam(discriminator.parameters(), lr=learning_rate)

# Training loop
for epoch in range(epochs):
    for i, (imgs, _) in enumerate(train_loader):

        # Configure input
        real_imgs = imgs.to(device)

        # ---------------------
        #  Train Discriminator
        # ---------------------

        optimizer_D.zero_grad()
```

PROF

```
        # Sample noise as generator input
        z = torch.randn(imgs.size(0), latent_dim).to(device)

        # Generate a batch of images
        fake_imgs = generator(z)

        # Real images label=1, fake images label=0
        real_labels = torch.ones(imgs.size(0), 1).to(device)
        fake_labels = torch.zeros(imgs.size(0), 1).to(device)

        # Compute discriminator loss
        real_loss = adversarial_loss(discriminator(real_imgs),
real_labels)
        fake_loss = adversarial_loss(discriminator(fake_imgs.detach()),
fake_labels)
        d_loss = (real_loss + fake_loss) / 2

        d_loss.backward()
        optimizer_D.step()

        # -----------------
        #  Train Generator
        # -----------------

        optimizer_G.zero_grad()

        # Generate images and compute generator loss
        gen_imgs = generator(z)
        g_loss = adversarial_loss(discriminator(gen_imgs), real_labels)

        g_loss.backward()
        optimizer_G.step()

        # Print progress
        if i % 200 == 0:
            print(f"[Epoch {epoch}/{epochs}] [Batch
{i}/{len(train_loader)}] "
                  f"[D loss: {d_loss.item():.4f}] [G loss:
{g_loss.item():.4f}]")

    # Save generated images at the end of each epoch
    with torch.no_grad():
        sample_z = torch.randn(16, latent_dim).to(device)
        generated = generator(sample_z).cpu()

        # Rescale images from [-1, 1] to [0, 1]
        generated = 0.5 * generated + 0.5

        # Create grid and save
        grid = torchvision.utils.make_grid(generated, nrow=4)
        plt.imshow(grid.permute(1, 2, 0))
        plt.axis('off')
        plt.savefig(f"gan_generated_epoch_{epoch}.png")
```

```
        plt.close()

# Save models
torch.save(generator.state_dict(), 'generator.pth')
torch.save(discriminator.state_dict(), 'discriminator.pth')
```

Key Components Explained:

1. **Generator Network**:
    - Takes random noise (latent vector) as input
    - Gradually upscales to MNIST image size (28x28)
    - Uses `Tanh()` activation to match input normalization [-1, 1]
2. **Discriminator Network**:
    - Classifies images as real (1) or fake (0)
    - Uses `LeakyReLU` and `Dropout` for better gradient flow
    - Outputs probability via `Sigmoid`
3. **Training Process**:
    - Alternates between:
        1. Discriminator training (maximize real/fake detection)
        2. Generator training (fool discriminator)
    - Uses Binary Cross-Entropy (BCE) loss
4. **MNIST Specifics**:
    - Images normalized to [-1, 1] range
    - Output images rescaled to [0, 1] for visualization

Expected Output:

PROF

- During training, you'll see the discriminator and generator losses fluctuating as they compete
- Saved PNG files will show generated digits improving over epochs
- Final saved models can generate new MNIST-like digits with:

```
generator.load_state_dict(torch.load('generator.pth'))
with torch.no_grad():
    z = torch.randn(1, latent_dim).to(device)
    generated_img = generator(z).cpu()
```

Common Issues & Fixes:

1. **Mode Collapse** (Generator produces limited variety):
    - Try increasing latent dimension size

- Add diversity terms to loss function

2. **Training Instability**:
   - Reduce learning rate
   - Use label smoothing (e.g., real_labels = 0.9 instead of 1.0)
3. **Blurry Outputs**:
   - Consider switching to DCGAN (convolutional architecture)
   - Add perceptual loss terms

Would you like me to modify any part of this implementation or add specific features like:

- TensorBoard logging?
- Early stopping?
- Different architecture variants?

Here's a complete implementation of a **DCGAN (Deep Convolutional GAN)** trained on CIFAR-10 with image visualization:

PROF

```
# File: 02_DCGAN_CIFAR10_Visualization.py
# Topic: DCGAN implementation for CIFAR-10 generation with PyTorch

import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import numpy as np
from torch.utils.tensorboard import SummaryWriter
import os

# Set random seed for reproducibility
torch.manual_seed(42)
np.random.seed(42)

# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Hyperparameters
latent_dim = 100
batch_size = 128
epochs = 100
learning_rate = 0.0002
img_channels = 3
img_size = 32

# Create output directory
os.makedirs('dcgan_results', exist_ok=True)
```

```
os.makedirs('dcgan_results/images', exist_ok=True)
writer = SummaryWriter('dcgan_results/runs')

# CIFAR-10 dataset
transform = transforms.Compose([
    transforms.Resize(img_size),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))  # Scale to
[-1, 1]
])

train_dataset = torchvision.datasets.CIFAR10(
    root='./data', train=True, transform=transform, download=True)
train_loader = torch.utils.data.DataLoader(
    dataset=train_dataset, batch_size=batch_size, shuffle=True)

# DCGAN Generator
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()

        self.init_size = img_size // 4
        self.l1 = nn.Sequential(
            nn.Linear(latent_dim, 128 * self.init_size**2)
        )

        self.model = nn.Sequential(
            nn.BatchNorm2d(128),
            nn.Upsample(scale_factor=2),
            nn.Conv2d(128, 128, 3, stride=1, padding=1),
            nn.BatchNorm2d(128, 0.8),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Upsample(scale_factor=2),
            nn.Conv2d(128, 64, 3, stride=1, padding=1),
            nn.BatchNorm2d(64, 0.8),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(64, img_channels, 3, stride=1, padding=1),
            nn.Tanh()
        )

    def forward(self, z):
        out = self.l1(z)
        out = out.view(out.shape[0], 128, self.init_size,
self.init_size)
        img = self.model(out)
        return img

# DCGAN Discriminator
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
```

PROF

```
        def discriminator_block(in_filters, out_filters, bn=True):
            block = [
                nn.Conv2d(in_filters, out_filters, 3, 2, 1),
                nn.LeakyReLU(0.2, inplace=True),
                nn.Dropout2d(0.25)
            ]
            if bn:
                block.append(nn.BatchNorm2d(out_filters, 0.8))
            return block

        self.model = nn.Sequential(
            *discriminator_block(img_channels, 16, bn=False),
            *discriminator_block(16, 32),
            *discriminator_block(32, 64),
            *discriminator_block(64, 128),
        )

        ds_size = img_size // 2**4
        self.adv_layer = nn.Sequential(
            nn.Linear(128 * ds_size**2, 1),
            nn.Sigmoid()
        )

    def forward(self, img):
        out = self.model(img)
        out = out.view(out.shape[0], -1)
        validity = self.adv_layer(out)
        return validity

# Initialize models
generator = Generator().to(device)
discriminator = Discriminator().to(device)

# Loss function and optimizers
adversarial_loss = nn.BCELoss()
optimizer_G = optim.Adam(generator.parameters(), lr=learning_rate,
betas=(0.5, 0.999))
optimizer_D = optim.Adam(discriminator.parameters(), lr=learning_rate,
betas=(0.5, 0.999))

# Fixed noise for sample generation
fixed_noise = torch.randn(64, latent_dim, device=device)

def save_image_grid(epoch, samples):
    """Save generated images as grid"""
    # Rescale images from [-1, 1] to [0, 1]
    samples = (samples + 1) / 2

    grid = torchvision.utils.make_grid(samples, nrow=8, padding=2,
normalize=False)
    plt.figure(figsize=(15, 15))
    plt.imshow(grid.permute(1, 2, 0).cpu().numpy())
```

```
    plt.axis('off')
    plt.savefig(f'dcgan_results/images/epoch_{epoch:03d}.png')
    plt.close()

# Training loop
for epoch in range(epochs):
    for i, (imgs, _) in enumerate(train_loader):

        # Configure input
        real_imgs = imgs.to(device)
        batch_size = real_imgs.size(0)

        # Adversarial ground truths
        valid = torch.ones(batch_size, 1, device=device)
        fake = torch.zeros(batch_size, 1, device=device)

        # ---------------------
        #  Train Discriminator
        # ---------------------
        optimizer_D.zero_grad()

        # Sample noise as generator input
        z = torch.randn(batch_size, latent_dim, device=device)

        # Generate a batch of images
        gen_imgs = generator(z)

        # Real images loss
        real_loss = adversarial_loss(discriminator(real_imgs), valid)
        # Fake images loss
        fake_loss = adversarial_loss(discriminator(gen_imgs.detach()),
fake)
        # Total discriminator loss
        d_loss = (real_loss + fake_loss) / 2

        d_loss.backward()
        optimizer_D.step()

        # -----------------
        #  Train Generator
        # -----------------
        optimizer_G.zero_grad()

        # Generate images and compute generator loss
        gen_imgs = generator(z)
        g_loss = adversarial_loss(discriminator(gen_imgs), valid)

        g_loss.backward()
        optimizer_G.step()

        # Logging
        batches_done = epoch * len(train_loader) + i
        if batches_done % 100 == 0:
```

```
            print(
                f"[Epoch {epoch}/{epochs}] [Batch
{i}/{len(train_loader)}] "
                f"[D loss: {d_loss.item():.4f}] [G loss:
{g_loss.item():.4f}]"
            )
            writer.add_scalar('Discriminator Loss', d_loss.item(),
batches_done)
            writer.add_scalar('Generator Loss', g_loss.item(),
batches_done)

    # Generate and save sample images
    with torch.no_grad():
        samples = generator(fixed_noise)
        save_image_grid(epoch, samples)

        # Save sample to TensorBoard
        img_grid = torchvision.utils.make_grid(
            (samples + 1) / 2,  # Scale to [0,1]
            nrow=8
        )
        writer.add_image('Generated Images', img_grid, epoch)

# Save final models
torch.save(generator.state_dict(), 'dcgan_results/generator.pth')
torch.save(discriminator.state_dict(),
'dcgan_results/discriminator.pth')
writer.close()
```

Key Features of This Implementation:

1. **DCGAN Architecture**:
   - Generator uses transposed convolutions (`Upsample` + `Conv2d`)
   - Discriminator uses strided convolutions for downsampling
   - Batch normalization in both networks
   - LeakyReLU activations
2. **CIFAR-10 Specifics**:
   - Input size: 32x32x3 (RGB)
   - Normalization to [-1, 1] range
   - Output images rescaled to [0, 1] for visualization
3. **Training Enhancements**:
   - TensorBoard logging for loss tracking
   - Fixed noise samples for consistent visualization
   - Adam optimizer with momentum (β1=0.5)
   - Periodic image grid generation

PROF

4. **Visualization**:
    - Saves image grids every epoch
    - Real-time progress monitoring via TensorBoard

Expected Output:

- Generated images will start as random noise and gradually resemble CIFAR-10 objects
- Sample output directory structure:

```
dcgan_results/
├── images/
│   ├── epoch_000.png
│   ├── epoch_001.png
│   └── ...
├── runs/ (TensorBoard logs)
├── generator.pth
└── discriminator.pth
```

Monitoring Training:

Launch TensorBoard to monitor progress:

```
tensorboard --logdir dcgan_results/runs
```

Common Improvements:

1. **For Better Quality**:
    - Add spectral normalization
    - Implement progressive growing
    - Use Wasserstein loss with gradient penalty
2. **For Stability**:
    - Add label smoothing
    - Use different learning rates for G/D
    - Implement exponential moving average of weights
3. **For Faster Training**:
    - Mixed precision training (`torch.cuda.amp`)
    - Larger batch sizes (if VRAM permits)

Would you like me to add any of these enhancements or modify the architecture for specific use cases?

PROF

Here's a complete implementation of a **Variational Autoencoder (VAE)** for Fashion-MNIST reconstruction in PyTorch, with visualization:

```
# File: 03_VAE_FashionMNIST_Reconstruction.py
# Topic: VAE implementation for Fashion-MNIST reconstruction using
PyTorch

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision
from torchvision import transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
import numpy as np
from torch.utils.tensorboard import SummaryWriter
import os

# Set random seed for reproducibility
torch.manual_seed(42)
np.random.seed(42)

# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Hyperparameters
latent_dim = 20
batch_size = 128
epochs = 30
learning_rate = 1e-3
image_size = 28 * 28  # Fashion-MNIST dimensions

# Create output directory
os.makedirs('vae_results', exist_ok=True)
os.makedirs('vae_results/reconstructions', exist_ok=True)
writer = SummaryWriter('vae_results/runs')

# Fashion-MNIST dataset
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))  # Scale to [-1, 1]
])

train_dataset = torchvision.datasets.FashionMNIST(
    root='./data', train=True, transform=transform, download=True)
train_loader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=True)

# VAE Model
class VAE(nn.Module):
```

PROF

```
    def __init__(self):
        super(VAE, self).__init__()

        # Encoder
        self.fc1 = nn.Linear(image_size, 400)
        self.fc21 = nn.Linear(400, latent_dim)  # μ (mean)
        self.fc22 = nn.Linear(400, latent_dim)  # logvar

        # Decoder
        self.fc3 = nn.Linear(latent_dim, 400)
        self.fc4 = nn.Linear(400, image_size)

    def encode(self, x):
        h1 = F.relu(self.fc1(x))
        return self.fc21(h1), self.fc22(h1)

    def reparameterize(self, mu, logvar):
        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std)
        return mu + eps * std

    def decode(self, z):
        h3 = F.relu(self.fc3(z))
        return torch.tanh(self.fc4(h3))  # Tanh to match input range
[-1,1]

    def forward(self, x):
        mu, logvar = self.encode(x.view(-1, image_size))
        z = self.reparameterize(mu, logvar)
        return self.decode(z), mu, logvar

# Loss function
def loss_function(recon_x, x, mu, logvar):
    BCE = F.mse_loss(recon_x, x.view(-1, image_size), reduction='sum')
    KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
    return BCE + KLD

# Initialize model and optimizer
model = VAE().to(device)
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

# Fixed input for visualization
fixed_input = next(iter(train_loader))[0][:32].to(device)

def save_reconstructions(epoch, original, reconstructed):
    """Save original and reconstructed images for comparison"""
    original = original.cpu().numpy()
    reconstructed = reconstructed.cpu().numpy()

    plt.figure(figsize=(10, 4))
    for i in range(8):  # Display first 8 samples
        # Original images
        ax = plt.subplot(2, 8, i + 1)
```

```
            plt.imshow(original[i].reshape(28, 28), cmap='gray')
            plt.axis('off')

            # Reconstructed images
            ax = plt.subplot(2, 8, i + 8 + 1)
            plt.imshow(reconstructed[i].reshape(28, 28), cmap='gray')
            plt.axis('off')

        plt.savefig(f'vae_results/reconstructions/epoch_{epoch:03d}.png')
        plt.close()

    # Training loop
    for epoch in range(epochs):
        model.train()
        train_loss = 0
        for batch_idx, (data, _) in enumerate(train_loader):
            data = data.to(device)

            optimizer.zero_grad()
            recon_batch, mu, logvar = model(data)
            loss = loss_function(recon_batch, data, mu, logvar)
            loss.backward()
            train_loss += loss.item()
            optimizer.step()

            if batch_idx % 100 == 0:
                print(f'Train Epoch: {epoch} [{batch_idx *
    len(data)}/{len(train_loader.dataset)} '
                      f'({100. * batch_idx /
    len(train_loader):.0f}%)]\tLoss: {loss.item() / len(data):.6f}')

        # Validation and visualization
        model.eval()
        with torch.no_grad():
            # Calculate validation loss
            val_loss = 0
            for data, _ in train_loader:  # Using same data for simplicity
                data = data.to(device)
                recon, mu, logvar = model(data)
                val_loss += loss_function(recon, data, mu, logvar).item()

            # Generate reconstructions
            recon_fixed, _, _ = model(fixed_input)

            # Save reconstructions
            save_reconstructions(epoch, fixed_input, recon_fixed)

            # TensorBoard logging
            writer.add_scalar('Training Loss', train_loss /
    len(train_loader.dataset), epoch)
            writer.add_scalar('Validation Loss', val_loss /
    len(train_loader.dataset), epoch)
```

```
        # Save sample to TensorBoard
        comparison = torch.cat([
            fixed_input[:8],
            recon_fixed.view(-1, 1, 28, 28)[:8]
        ])
        writer.add_images(
            'Original vs Reconstructed',
            (comparison + 1) / 2,  # Scale to [0,1]
            epoch
        )

    print(f'====> Epoch: {epoch} Average loss: {train_loss /
len(train_loader.dataset):.4f}')

# Save model
torch.save(model.state_dict(), 'vae_results/vae_fashionmnist.pth')
writer.close()

# Generate new samples
with torch.no_grad():
    sample = torch.randn(64, latent_dim).to(device)
    generated = model.decode(sample).cpu()

    plt.figure(figsize=(10, 10))
    for i in range(64):
        plt.subplot(8, 8, i + 1)
        plt.imshow(generated[i].view(28, 28), cmap='gray')
        plt.axis('off')
    plt.savefig('vae_results/generated_samples.png')
    plt.close()
```

Key Components Explained:

1. **VAE Architecture**:
   - **Encoder**: Maps input to latent space (μ and logvar)
   - **Reparameterization Trick**: Enables backpropagation through random sampling
   - **Decoder**: Reconstructs input from latent space
2. **Loss Function**:
   - **Reconstruction Loss (MSE)**: Measures pixel-wise difference
   - **KL Divergence**: Regularizes latent space to be close to standard normal
3. **Fashion-MNIST Specifics**:
   - Input images normalized to [-1, 1] range
   - Tanh activation in decoder matches input range
   - 28x28 grayscale images
4. **Visualization Features**:

PROF

- Saves original vs reconstructed comparisons each epoch
- Generates new samples from random latent vectors
- TensorBoard integration for loss tracking

Expected Output:

```
vae_results/
├── reconstructions/
│   ├── epoch_000.png
│   ├── epoch_001.png
│   └── ...
├── runs/ (TensorBoard logs)
├── generated_samples.png
└── vae_fashionmnist.pth
```

Monitoring Training:

```
tensorboard --logdir vae_results/runs
```

Common Improvements:

1. **For Better Quality**:

```
# Replace fc layers with convolutional layers
class ConvVAE(nn.Module):
    def __init__(self):
        super().__init__()
        # Encoder
        self.encoder = nn.Sequential(
            nn.Conv2d(1, 32, 3, stride=2, padding=1),
            nn.ReLU(),
            nn.Conv2d(32, 64, 3, stride=2, padding=1),
            nn.ReLU()
        )
        # Latent space
        self.fc_mu = nn.Linear(64*7*7, latent_dim)
        self.fc_logvar = nn.Linear(64*7*7, latent_dim)
        # Decoder
        self.decoder_input = nn.Linear(latent_dim, 64*7*7)
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(64, 32, 3, stride=2, padding=1,
output_padding=1),
            nn.ReLU(),
            nn.ConvTranspose2d(32, 1, 3, stride=2, padding=1,
output_padding=1),
```

```
            nn.Tanh()
        )
```

2. **For Different Objectives**:
    - Use BCE loss instead of MSE for sharper images
    - Add perceptual loss with a pre-trained network
3. **For Advanced Use**:
    - Implement β-VAE for disentangled representations
    - Add classifier guidance for conditional generation

Would you like me to modify any aspect of this implementation or add specific features like:

- Latent space interpolation visualization?
- Anomaly detection capabilities?
- Quantitative evaluation metrics (FID, etc.)?

Here's a complete implementation of a **Conditional GAN (cGAN)** in PyTorch for generating specific MNIST digits, with class conditioning and visualization:

PROF

```
# File: 04_ConditionalGAN_MNIST.py
# Topic: Conditional GAN for class-specific MNIST generation using
PyTorch

import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import numpy as np
from torch.utils.tensorboard import SummaryWriter
import os

# Set random seed for reproducibility
torch.manual_seed(42)
np.random.seed(42)

# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Hyperparameters
latent_dim = 100
num_classes = 10
img_size = 28
batch_size = 64
epochs = 50
```

```
learning_rate = 0.0002

# Create output directory
os.makedirs('cgan_results', exist_ok=True)
os.makedirs('cgan_results/images', exist_ok=True)
writer = SummaryWriter('cgan_results/runs')

# MNIST dataset with labels
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))  # Scale to [-1, 1]
])

train_dataset = torchvision.datasets.MNIST(
    root='./data', train=True, transform=transform, download=True)
train_loader = torch.utils.data.DataLoader(
    dataset=train_dataset, batch_size=batch_size, shuffle=True)

# Conditional Generator
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()

        self.label_emb = nn.Embedding(num_classes, num_classes)

        self.model = nn.Sequential(
            nn.Linear(latent_dim + num_classes, 256),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(256, 512),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(512, 1024),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(1024, img_size * img_size),
            nn.Tanh()
        )

    def forward(self, z, labels):
        # Concatenate noise and label embedding
        label_input = self.label_emb(labels)
        gen_input = torch.cat((label_input, z), -1)
        img = self.model(gen_input)
        img = img.view(img.size(0), 1, img_size, img_size)
        return img

# Conditional Discriminator
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()

        self.label_emb = nn.Embedding(num_classes, num_classes)

        self.model = nn.Sequential(
            nn.Linear(img_size * img_size + num_classes, 1024),
```

PROF

```
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.3),
            nn.Linear(1024, 512),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.3),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.3),
            nn.Linear(256, 1),
            nn.Sigmoid()
        )

    def forward(self, img, labels):
        # Concatenate image and label embedding
        img_flat = img.view(img.size(0), -1)
        label_input = self.label_emb(labels)
        d_in = torch.cat((img_flat, label_input), -1)
        validity = self.model(d_in)
        return validity

# Initialize models
generator = Generator().to(device)
discriminator = Discriminator().to(device)

# Loss function and optimizers
adversarial_loss = nn.BCELoss()
optimizer_G = optim.Adam(generator.parameters(), lr=learning_rate)
optimizer_D = optim.Adam(discriminator.parameters(), lr=learning_rate)

# Fixed noise and labels for sample generation
fixed_noise = torch.randn(10, latent_dim, device=device)  # 10 samples
fixed_labels = torch.arange(0, 10, dtype=torch.long, device=device)  #
Digits 0-9

def save_image_grid(epoch, samples):
    """Save generated images as grid with labels"""
    samples = (samples + 1) / 2  # Rescale to [0,1]

    fig, axes = plt.subplots(1, 10, figsize=(15, 2))
    for i, ax in enumerate(axes):
        ax.imshow(samples[i].squeeze().cpu().numpy(), cmap='gray')
        ax.set_title(f"Label: {i}")
        ax.axis('off')
    plt.savefig(f'cgan_results/images/epoch_{epoch:03d}.png')
    plt.close()

# Training loop
for epoch in range(epochs):
    for i, (imgs, labels) in enumerate(train_loader):

        # Configure input
        real_imgs = imgs.to(device)
        real_labels = labels.to(device)
```

```
        batch_size = real_imgs.size(0)

        # Adversarial ground truths
        valid = torch.ones(batch_size, 1, device=device)
        fake = torch.zeros(batch_size, 1, device=device)

        # ---------------------
        #  Train Discriminator
        # ---------------------
        optimizer_D.zero_grad()

        # Sample noise and labels
        z = torch.randn(batch_size, latent_dim, device=device)
        gen_labels = torch.randint(0, num_classes, (batch_size,),
device=device)

        # Generate fake images
        fake_imgs = generator(z, gen_labels)

        # Real images loss
        real_loss = adversarial_loss(discriminator(real_imgs,
real_labels), valid)
        # Fake images loss
        fake_loss = adversarial_loss(discriminator(fake_imgs.detach(),
gen_labels), fake)
        # Total discriminator loss
        d_loss = (real_loss + fake_loss) / 2

        d_loss.backward()
        optimizer_D.step()

        # -----------------
        #  Train Generator
        # -----------------
        optimizer_G.zero_grad()

        # Generate images and compute generator loss
        gen_imgs = generator(z, gen_labels)
        g_loss = adversarial_loss(discriminator(gen_imgs, gen_labels),
valid)

        g_loss.backward()
        optimizer_G.step()

        # Logging
        batches_done = epoch * len(train_loader) + i
        if batches_done % 200 == 0:
            print(f"[Epoch {epoch}/{epochs}] [Batch
{i}/{len(train_loader)}] "
                  f"[D loss: {d_loss.item():.4f}] [G loss:
{g_loss.item():.4f}]")
            writer.add_scalar('Discriminator Loss', d_loss.item(),
batches_done)
```

```
            writer.add_scalar('Generator Loss', g_loss.item(),
batches_done)

    # Generate and save sample images
    with torch.no_grad():
        samples = generator(fixed_noise, fixed_labels)
        save_image_grid(epoch, samples)

        # Save to TensorBoard
        img_grid = torchvision.utils.make_grid(
            (samples + 1) / 2,  # Scale to [0,1]
            nrow=10
        )
        writer.add_image('Generated Images', img_grid, epoch)

# Save models
torch.save(generator.state_dict(), 'cgan_results/generator.pth')
torch.save(discriminator.state_dict(), 'cgan_results/discriminator.pth')
writer.close()

# Interactive generation demo
def generate_digit(digit, num_samples=1):
    """Generate specific digit samples"""
    generator.load_state_dict(torch.load('cgan_results/generator.pth'))
    generator.eval()

    z = torch.randn(num_samples, latent_dim, device=device)
    labels = torch.full((num_samples,), digit, dtype=torch.long,
device=device)

    with torch.no_grad():
        samples = generator(z, labels).cpu()

    plt.figure(figsize=(5, 5))
    for i in range(num_samples):
        plt.subplot(1, num_samples, i+1)
        plt.imshow(samples[i].squeeze(), cmap='gray')
        plt.title(f"Generated {digit}")
        plt.axis('off')
    plt.show()

# Example: Generate 5 samples of digit 7
generate_digit(7, 5)
```

PROF

Key Features:

1. **Conditioning Mechanism**:
    - Uses `nn.Embedding` for label conditioning
    - Concatenates label embeddings with:
        - Noise vector (Generator input)

- Flattened image (Discriminator input)

2. **Class-Specific Generation**:
   - Fixed noise + sequential labels (0-9) for visualization
   - `generate_digit()` function for targeted generation
3. **Training Process**:
   - Alternates between real/fake discrimination
   - Generator learns to produce class-specific digits
   - Labels provided to both generator and discriminator
4. **Visualization**:
   - Saves image grid showing all 10 digits each epoch
   - TensorBoard integration for loss tracking
   - Interactive generation function

Expected Output:

```
cgan_results/
├── images/
│   ├── epoch_000.png (shows digits 0-9)
│   ├── epoch_001.png
│   └── ...
├── runs/ (TensorBoard logs)
├── generator.pth
└── discriminator.pth
```

How to Use:

1. **Generate Specific Digits**:

```
generate_digit(3)  # Creates digit 3
generate_digit(9, 3)  # Creates 3 samples of digit 9
```

2. **Monitor Training**:

```
tensorboard --logdir cgan_results/runs
```

Common Improvements:

1. **For Better Quality**:

PROF

```
# Replace linear layers with convolutional architecture
class ConvGenerator(nn.Module):
    def __init__(self):
        super().__init__()
        self.label_emb = nn.Embedding(num_classes, num_classes)
        self.init_size = img_size // 4

        self.model = nn.Sequential(
            nn.Linear(latent_dim + num_classes,
128*self.init_size**2),
            nn.BatchNorm1d(128*self.init_size**2),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Unflatten(1, (128, self.init_size, self.init_size)),
            nn.ConvTranspose2d(128, 64, 4, stride=2, padding=1),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.2, inplace=True),
            nn.ConvTranspose2d(64, 1, 4, stride=2, padding=1),
            nn.Tanh()
        )
```

2. **For More Control**:
    - Add auxiliary classifier loss
    - Implement infoGAN for unsupervised feature discovery
3. **For Stability**:
    - Use spectral normalization
    - Implement WGAN-GP loss

Would you like me to add any specific features like:

- Latent space interpolation between classes?
- Quantitative evaluation metrics (e.g., Inception Score)?
- Multi-label conditioning?

PROF

```
# File: 05_StyleGAN2_Custom_Dataset_FineTuning.py
# Topic: Fine-tune StyleGAN2 on custom dataset using PyTorch

import os
import torch
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import transforms
from PIL import Image
import numpy as np
import dnnlib
import legacy
from tqdm import tqdm
```

```python
# Configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
batch_size = 4
num_epochs = 50
learning_rate = 0.002
save_interval = 5
dataset_path = "./custom_dataset"  # Directory with your images
output_dir = "./stylegan2_finetune_results"
os.makedirs(output_dir, exist_ok=True)

# 1. Load Pre-trained StyleGAN2
print("Loading pre-trained StyleGAN2...")
network_pkl = "https://nvlabs-fi-cdn.nvidia.com/stylegan2-ada-
pytorch/pretrained/ffhq.pkl"
with dnnlib.util.open_url(network_pkl) as f:
    G = legacy.load_network_pkl(f)['G_ema'].to(device)

# 2. Prepare Custom Dataset
class CustomDataset(torch.utils.data.Dataset):
    def __init__(self, root, transform=None):
        self.root = root
        self.transform = transform
        self.image_files = [f for f in os.listdir(root) if
f.endswith(('.jpg', '.png', '.jpeg'))]

    def __len__(self):
        return len(self.image_files)

    def __getitem__(self, idx):
        img_path = os.path.join(self.root, self.image_files[idx])
        image = Image.open(img_path).convert('RGB')

        if self.transform:
            image = self.transform(image)

        return image

transform = transforms.Compose([
    transforms.Resize((256, 256)),  # StyleGAN2 default size
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

dataset = CustomDataset(dataset_path, transform=transform)
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)

# 3. Setup for Fine-Tuning
# Freeze early layers (optional for better stability)
for name, param in G.named_parameters():
    if 'b4' not in name and 'b8' not in name and 'b16' not in name and
'b32' not in name and 'b64' not in name:
        param.requires_grad = False
```

```
optimizer = optim.Adam(
    filter(lambda p: p.requires_grad, G.parameters()),
    lr=learning_rate,
    betas=(0.0, 0.99)
)

# 4. Training Loop
print("Starting fine-tuning...")
for epoch in range(num_epochs):
    progress_bar = tqdm(dataloader, desc=f"Epoch
{epoch+1}/{num_epochs}")

    for i, real_images in enumerate(progress_bar):
        real_images = real_images.to(device)

        # Generate random latent vectors
        z = torch.randn(batch_size, G.z_dim).to(device)

        # Generate fake images
        fake_images = G(z, None, truncation_psi=0.7, noise_mode='const')

        # Compute loss (simple L1 loss for reconstruction)
        loss = torch.nn.functional.l1_loss(fake_images, real_images)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        progress_bar.set_postfix({"Loss": loss.item()})

    # Save checkpoint periodically
    if (epoch + 1) % save_interval == 0:
        torch.save({
            'epoch': epoch,
            'model_state_dict': G.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
            'loss': loss,
        }, os.path.join(output_dir,
f'stylegan2_ft_epoch_{epoch+1}.pth'))

        # Generate sample images
        with torch.no_grad():
            test_z = torch.randn(4, G.z_dim).to(device)
            sample_images = G(test_z, None, truncation_psi=0.7)
            sample_images = (sample_images.permute(0, 2, 3, 1) * 127.5 +
128).clamp(0, 255).to(torch.uint8)
            sample_images = sample_images.cpu().numpy()

            # Save sample images
            for j, img in enumerate(sample_images):
                Image.fromarray(img).save(os.path.join(output_dir,
```

PROF

```
    f'epoch_{epoch+1}_sample_{j}.png'))

print("Fine-tuning complete!")

# 5. Generate New Samples
def generate_samples(num_samples=8, truncation_psi=0.7):
    """Generate samples from fine-tuned model"""
    G.eval()
    with torch.no_grad():
        z = torch.randn(num_samples, G.z_dim).to(device)
        samples = G(z, None, truncation_psi=truncation_psi)
        samples = (samples.permute(0, 2, 3, 1) * 127.5 + 128).clamp(0,
255).to(torch.uint8)
        return samples.cpu().numpy()

# Generate and save final samples
final_samples = generate_samples(16)
for i, sample in enumerate(final_samples):
    Image.fromarray(sample).save(os.path.join(output_dir,
f'final_sample_{i}.png'))
```

Key Components Explained:

1. **Pre-trained Model Loading**:
    - Uses NVIDIA's official StyleGAN2-ADA implementation
    - Loads FFHQ (1024x1024) as base model
2. **Custom Dataset Preparation**:
    - Processes images to 256x256 resolution
    - Normalizes to [-1, 1] range (StyleGAN2 standard)
    - Handles common image formats (JPG, PNG, JPEG)

PROF

3. **Fine-Tuning Strategy**:
    - Freezes early layers (more stable training)
    - Only trains higher-resolution layers (b4-b64)
    - Uses L1 loss for reconstruction (can be changed to perceptual loss)
4. **Training Process**:
    - Saves checkpoints periodically
    - Generates sample images during training
    - Uses Adam optimizer with β2=0.99

Setup Instructions:

1. **Prerequisites**:

```
```

```
pip install torch torchvision numpy pillow tqdm requests
git clone https://github.com/NVlabs/stylegan2-ada-pytorch.git
cd stylegan2-ada-pytorch
```

2. **Dataset Preparation**:
    - Place your images in `./custom_dataset/`
    - Recommended: 1000+ images for good results
    - Images should be cropped and aligned (like FFHQ)
3. **Recommended Folder Structure**:

```
project/
├── stylegan2-ada-pytorch/  # Official repo
├── custom_dataset/
│   ├── image1.jpg
│   ├── image2.png
│   └── ...
└── 05_StyleGAN2_Custom_Dataset_FineTuning.py
```

Advanced Options:

1. **For Better Quality**:

```
# Replace L1 loss with perceptual loss
from lpips import LPIPS
percept_loss = LPIPS(net='vgg').to(device)

# In training loop:
loss = percept_loss(fake_images, real_images).mean()
```

PROF

2. **For Larger Datasets**:

```
# Use larger batch sizes if VRAM permits
batch_size = 8  # or higher

# Enable mixed precision training
scaler = torch.cuda.amp.GradScaler()
with torch.cuda.amp.autocast():
    fake_images = G(z, None)
    loss = percept_loss(fake_images, real_images)
scaler.scale(loss).backward()
scaler.step(optimizer)
scaler.update()
```

3. **For Different Resolutions**:

```
# Load config-e for 256x256
network_pkl = "https://nvlabs-fi-cdn.nvidia.com/stylegan2-ada-
pytorch/pretrained/cifar10.pkl"

# Or config-f for 1024x1024
network_pkl = "https://nvlabs-fi-cdn.nvidia.com/stylegan2-ada-
pytorch/pretrained/ffhq.pkl"
```

Expected Output:

```
stylegan2_finetune_results/
├── stylegan2_ft_epoch_5.pth
├── stylegan2_ft_epoch_10.pth
├── ...
├── epoch_5_sample_0.png
├── epoch_10_sample_0.png
├── ...
└── final_sample_0.png
```

Troubleshooting:

1. **Out of Memory Errors**:
    - Reduce batch size
    - Use gradient accumulation:

```
accumulation_steps = 4
loss = loss / accumulation_steps
loss.backward()
if (i + 1) % accumulation_steps == 0:
    optimizer.step()
    optimizer.zero_grad()
```

2. **Mode Collapse**:
    - Increase diversity penalty
    - Add noise to latent vectors:

```
z = z + 0.01 * torch.randn_like(z)
```

3. **Poor Quality**:

PROF

- Try different learning rates (0.001-0.01)
- Unfreeze more layers
- Increase dataset size/variety

Here's a detailed **comparison between a Wasserstein GAN (WGAN)** and a **standard GAN**, particularly from the perspective of **performance in PyTorch**, including **training stability, loss behavior, and generated sample quality**.

🔁 1. **Objective Function**

Feature	Standard GAN	Wasserstein GAN (WGAN)
Loss Function	Binary Cross-Entropy (BCE)	Wasserstein Loss (Earth Mover's Distance)
Discriminator Output	Probability (0–1) via sigmoid	Real-valued (no sigmoid)
Goal	Minimize JS Divergence	Minimize Wasserstein (EM) distance

---

🧠 2. **Training Stability**

Aspect	Standard GAN	WGAN
Mode Collapse	Frequent, especially with high learning rates	Rare, due to smoother gradients
Gradient Vanishing	Common when discriminator gets too good	Solved by removing sigmoid + using continuous output space
Stability	Sensitive to hyperparameters and architecture	More stable and robust across a range of hyperparameters

---

PROF

📊 3. **Loss Behavior**

Loss Curve Interpretation	Standard GAN	WGAN
Generator Loss	Often unstable, hard to interpret	Linearly correlated with sample quality
Discriminator Loss	May saturate (0 or 1 probabilities)	Meaningful during the entire training
Monitoring Training	Difficult; loss does not correlate well	Easier to debug and monitor

---

🧪 4. **PyTorch Implementation Differences**

Generator – Both are similar.

Discriminator Differences:

- **Standard GAN** uses `nn.Sigmoid` and `BCELoss`.
- **WGAN** removes `Sigmoid`, avoids `BCELoss`, and uses **mean output** as the critic score.

```
# WGAN Discriminator example
class Critic(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(784, 512),
            nn.LeakyReLU(0.2),
            nn.Linear(512, 1)  # No Sigmoid here!
        )

    def forward(self, x):
        return self.model(x)
```

Optimizer:

- **Standard GAN**: Adam with β1=0.5, β2=0.999.
- **WGAN**: RMSprop or Adam with smaller β1 (e.g., β1=0) recommended.

Weight Clipping:

```
# Required in original WGAN
for p in critic.parameters():
    p.data.clamp_(-0.01, 0.01)
```

PROF

(Optional) WGAN-GP: Gradient Penalty instead of clipping — improves further stability.

🎯 5. Sample Quality & Convergence

Factor	Standard GAN	WGAN
Sample Diversity	Lower, more prone to mode collapse	Higher, less collapse
Visual Quality	Acceptable, but can fluctuate	More consistent and better with fewer epochs
Training Time	Faster per epoch	Slightly slower due to multiple critic steps
Epochs to Converge	Varies, often needs tuning	Usually converges faster with consistent loss

📌 Summary Table

Feature	Standard GAN	Wasserstein GAN (WGAN)
Stability	× Unstable	✅ Stable
Loss Meaning	× Not meaningful	✅ Correlated with performance
Gradient Issues	× Exploding/vanishing	✅ Smoother gradients
Implementation Ease	✅ Simple	⚠ Requires clipping / GP
Mode Collapse	× Common	✅ Rare
Sample Quality	⚠ Moderate	✅ High (especially with GP)

✅ When to Use WGAN over GAN

Scenario	Use WGAN if:
You face mode collapse in vanilla GANs	
Your generator loss doesn't correlate with sample quality	
You're training on complex, high-dimensional data	
You want more interpretable training behavior	

🧪 Bonus: Code Comparison Repository

You can explore real PyTorch examples:

- **WGAN**: https://github.com/eriklindernoren/PyTorch-GAN/blob/master/implementations/wgan
- **Standard GAN**: https://github.com/eriklindernoren/PyTorch-GAN/blob/master/implementations/gan

---