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?