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

```
# 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):
```

PROF

```
        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),
            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
```

PROF

```
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,
```

PROF

```
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
```

PROF

```
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)
```

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

PROF

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**:

   ```
   # 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?