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

```
# 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__()

        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),
        )
```

PROF

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

PROF

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?