```python
import random
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
from torch.utils.data import DataLoader
import torchvision.transforms as transforms
from tqdm.auto import tqdm
import matplotlib.pyplot as plt
import numpy as np
from torchvision.utils import make_grid

# ------------------------------
# Define the Models at the Module Level
# ------------------------------

class Generator(nn.Module):
    """
    GAN Generator for 64x64 color images.
    """
    def __init__(self, z_dim=100, g_feat=64):
        super().__init__()
        self.net = nn.Sequential(
            nn.ConvTranspose2d(z_dim, g_feat * 8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(g_feat * 8),
            nn.ReLU(True),

            nn.ConvTranspose2d(g_feat * 8, g_feat * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(g_feat * 4),
            nn.ReLU(True),

            nn.ConvTranspose2d(g_feat * 4, g_feat * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(g_feat * 2),
            nn.ReLU(True),

            nn.ConvTranspose2d(g_feat * 2, g_feat, 4, 2, 1, bias=False),
            nn.BatchNorm2d(g_feat),
            nn.ReLU(True),

            nn.ConvTranspose2d(g_feat, 3, 4, 2, 1, bias=False),
            nn.Tanh()
        )

    def forward(self, z):
        return self.net(z)

    # Note: The duplicate forward method below was present in the original code.
    # It is not necessary, but is kept here for consistency.
    def forward(self, z):
        return self.net(z)

class Discriminator(nn.Module):
    """
    GAN Discriminator for 64x64 color images.
    """
    def __init__(self, d_feat=64):
        super().__init__()
        self.net = nn.Sequential(
            nn.Conv2d(3, d_feat, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(d_feat, d_feat * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(d_feat * 2),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(d_feat * 2, d_feat * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(d_feat * 4),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(d_feat * 4, d_feat * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(d_feat * 8),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(d_feat * 8, 1, 4, 1, 0, bias=False)
        )

    def forward(self, x):
        # Flatten final output to shape (B, 1)
        return self.net(x).view(-1, 1)

# ------------------------------
# Main Experiment Function
# ------------------------------
def reproduce_hw4(seed=42):
    """
    This function reproduces the results of the GAN training experiment,
    including visualizations of the latent space. Models are saved as pickle files.
    """
    # 1. PLANT THE RANDOM SEED
    torch.manual_seed(seed)
    np.random.seed(seed)
    random.seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed_all(seed)
    print(f"Random seed set to: {seed}")

    # 2. DATASET & DATALOADER
    transform = transforms.Compose([
        transforms.Resize((64, 64)),
        transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
    ])

    dataset = torchvision.datasets.Flowers102('Flowers102', split='test',
                                              transform=transform, download=True)
    data_loader = DataLoader(dataset, batch_size=128, shuffle=True)

    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    print("Using device:", device)

    # 3. TRAINING CONFIGURATION
    epochs = 50
    lr = 2e-4
    beta1, beta2 = 0.5, 0.999
    criterion = nn.BCEWithLogitsLoss()
    real_label = 1.0
    fake_label = 0.0

    # Helper Functions
    def show(images, n_images=16, nrow=4, title="Images", figsize=6):
        plt.figure(figsize=(figsize, figsize))
        grid = make_grid((images[:n_images] + 1) / 2, nrow=nrow).permute(1, 2, 0)
        plt.title(title)
        plt.imshow(grid.detach().cpu().numpy())
        plt.axis("off")
        plt.show()

    def interpolate_latent_space(G, z_dim, steps=8, device='cpu'):
        z1 = torch.randn(1, z_dim, 1, 1, device=device)
        z2 = torch.randn(1, z_dim, 1, 1, device=device)
        interpolated_images = []
        for alpha in np.linspace(0, 1, steps):
            z_interp = (1 - alpha) * z1 + alpha * z2
            with torch.no_grad():
                fake_img = G(z_interp)
            interpolated_images.append(fake_img.squeeze(0))
        interpolated_images = torch.stack(interpolated_images, dim=0)
        show(interpolated_images, n_images=steps, nrow=steps,
             title=f"Interpolation (z_dim={z_dim})", figsize=steps)

    def visualize_latent_distribution(z_dim, method='pca', n_samples=500):
        from sklearn.decomposition import PCA
        from sklearn.manifold import TSNE
        z = torch.randn(n_samples, z_dim)
        z_np = z.cpu().numpy()
        if method.lower() == 'pca':
            reducer = PCA(n_components=2)
        elif method.lower() == 'tsne':
            reducer = TSNE(n_components=2, perplexity=30, random_state=42)
        else:
            raise ValueError("method must be 'pca' or 'tsne'")
        z_reduced = reducer.fit_transform(z_np)
        plt.figure(figsize=(6, 6))
        plt.scatter(z_reduced[:, 0], z_reduced[:, 1], alpha=0.7, s=30, edgecolors='k')
        plt.title(f"Latent Distribution ({method.upper()}) - z_dim={z_dim}")
        plt.xlabel("Component 1")
        plt.ylabel("Component 2")
        plt.grid(True)
        plt.show()

    def plot_latent_samples_and_generated_images(G, z_dim, device='cpu'):
        # Sample 3 latent vectors
        z_samples = torch.randn(3, z_dim, 1, 1, device=device)
        # Flatten for PCA reduction
        z_flat = z_samples.view(3, z_dim)
        from sklearn.decomposition import PCA
        pca = PCA(n_components=2)
        z_reduced = pca.fit_transform(z_flat.cpu().numpy())

        # Plot the latent vectors on a 2D scatter plot
        plt.figure(figsize=(6,6))
        plt.scatter(z_reduced[:, 0], z_reduced[:, 1], color='blue', s=100)
        for i, (x, y) in enumerate(z_reduced):
            plt.text(x, y, f'z{i+1}', fontsize=12, ha='right', color='red')
        plt.title("3 Sampled Latent Vectors (Reduced to 2D via PCA)")
        plt.xlabel("Component 1")
        plt.ylabel("Component 2")
        plt.grid(True)
        plt.show()

        # Generate and display images corresponding to these latent vectors
        with torch.no_grad():
            generated_images = G(z_samples)
        show(generated_images, n_images=3, nrow=3, title="Generated Images from 3 z Vectors")

    # 4. TRAIN 3 GANS WITH DIFFERENT Z_DIMS
    z_dims = [10, 100, 500]
    all_disc_losses = {}
    all_gen_losses = {}
    epoch_disc_losses = {}
    epoch_gen_losses = {}

    for z_dim in z_dims:
        print(f"\nTraining GAN with z_dim={z_dim}...")
        G = Generator(z_dim=z_dim).to(device)
        D = Discriminator().to(device)
        optimizerG = optim.Adam(G.parameters(), lr=lr, betas=(beta1, beta2))
        optimizerD = optim.Adam(D.parameters(), lr=lr, betas=(beta1, beta2))
        d_losses = []
        g_losses = []
        epoch_d_losses = []
        epoch_g_losses = []

        for epoch in range(epochs):
            loop = tqdm(data_loader, desc=f"Epoch [{epoch+1}/{epochs}]", leave=False)
            for real_images, _ in loop:
                real_images = real_images.to(device)
                batch_size = real_images.size(0)

                # Train Discriminator
                D.zero_grad()
                labels_real = torch.full((batch_size, 1), real_label, dtype=torch.float, device=device)
                output_real = D(real_images)
                lossD_real = criterion(output_real, labels_real)

                noise = torch.randn(batch_size, z_dim, 1, 1, device=device)
                fake_images = G(noise)
                labels_fake = torch.full((batch_size, 1), fake_label, dtype=torch.float, device=device)
                output_fake = D(fake_images.detach())
                lossD_fake = criterion(output_fake, labels_fake)

                lossD = lossD_real + lossD_fake
                lossD.backward()
                optimizerD.step()

                # Train Generator
                G.zero_grad()
                labels_fake_for_G = torch.full((batch_size, 1), real_label, dtype=torch.float, device=device)
                output_fake_for_G = D(fake_images)
                lossG = criterion(output_fake_for_G, labels_fake_for_G)
                lossG.backward()
                optimizerG.step()

                d_losses.append(lossD.item())
                g_losses.append(lossG.item())

            avg_d_loss = sum(d_losses[-len(data_loader):]) / len(data_loader)
            avg_g_loss = sum(g_losses[-len(data_loader):]) / len(data_loader)
            epoch_d_losses.append(avg_d_loss)
            epoch_g_losses.append(avg_g_loss)
            print(f"[z_dim={z_dim}][Epoch {epoch+1}/{epochs}] D_loss: {avg_d_loss:.4f}, G_loss: {avg_g_loss:.4f}")

        all_disc_losses[z_dim] = d_losses
        all_gen_losses[z_dim] = g_losses
        epoch_disc_losses[z_dim] = epoch_d_losses
        epoch_gen_losses[z_dim] = epoch_g_losses

        # Final visualization of generated images
        G.eval()
        with torch.no_grad():
            test_noise = torch.randn(16, z_dim, 1, 1, device=device)
            fakes = G(test_noise)
        show(fakes, title=f"Generated images (z_dim={z_dim})", n_images=16, nrow=4, figsize=6)
        G.train()

        # Save the models
        torch.save(G, f"generator_zdim{z_dim}.pkl")
        torch.save(D, f"discriminator_zdim{z_dim}.pkl")
        print(f"Saved generator_zdim{z_dim}.pkl and discriminator_zdim{z_dim}.pkl")

        # Visualize the latent space
        print(f"Visualizing latent space for z_dim={z_dim}...")
        plot_latent_samples_and_generated_images(G, z_dim, device=device)

    # Plot loss curves (per training step)
    plt.figure(figsize=(28, 12))
    plt.subplot(1, 2, 1)
    for z_dim in z_dims:
        plt.plot(all_disc_losses[z_dim], label=f"z_dim={z_dim}")
    plt.title("Discriminator Loss vs Training Steps")
    plt.xlabel("Training Steps")
    plt.ylabel("Loss")
    plt.legend()

    plt.subplot(1, 2, 2)
    for z_dim in z_dims:
        plt.plot(all_gen_losses[z_dim], label=f"z_dim={z_dim}")
    plt.title("Generator Loss vs Training Steps")
    plt.xlabel("Training Steps")
    plt.ylabel("Loss")
    plt.legend()

    plt.tight_layout()
    plt.show()

    # Plot epoch-wise loss curves
    plt.figure(figsize=(28, 12))
    plt.subplot(1, 2, 1)
    for z_dim in z_dims:
        plt.plot(epoch_disc_losses[z_dim], marker='o', label=f"z_dim={z_dim}")
    plt.title("Epoch-wise Discriminator Loss")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.legend()

    plt.subplot(1, 2, 2)
    for z_dim in z_dims:
        plt.plot(epoch_gen_losses[z_dim], marker='o', label=f"z_dim={z_dim}")
    plt.title("Epoch-wise Generator Loss")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.legend()

    plt.tight_layout()
    plt.show()

    # ------------------------------
    # Extra: Train GAN with z_dim=100 for 100 Epochs
    # ------------------------------
    print("\nExtra Training: Training GAN with z_dim=100 for 100 epochs...")
    epochs_extra = 100
    G_extra = Generator(z_dim=100).to(device)
    D_extra = Discriminator().to(device)
    optimizerG_extra = optim.Adam(G_extra.parameters(), lr=lr, betas=(beta1, beta2))
    optimizerD_extra = optim.Adam(D_extra.parameters(), lr=lr, betas=(beta1, beta2))
    extra_d_losses = []
    extra_g_losses = []

    for epoch in range(epochs_extra):
        loop = tqdm(data_loader, desc=f"Extra Epoch [{epoch+1}/{epochs_extra}]", leave=False)
        for real_images, _ in loop:
            real_images = real_images.to(device)
            batch_size = real_images.size(0)

            # Train Discriminator
            D_extra.zero_grad()
            labels_real = torch.full((batch_size, 1), real_label, dtype=torch.float, device=device)
            output_real = D_extra(real_images)
            lossD_real = criterion(output_real, labels_real)

            noise = torch.randn(batch_size, 100, 1, 1, device=device)
            fake_images = G_extra(noise)
            labels_fake = torch.full((batch_size, 1), fake_label, dtype=torch.float, device=device)
            output_fake = D_extra(fake_images.detach())
            lossD_fake = criterion(output_fake, labels_fake)

            lossD = lossD_real + lossD_fake
            lossD.backward()
            optimizerD_extra.step()

            # Train Generator
            G_extra.zero_grad()
            labels_fake_for_G = torch.full((batch_size, 1), real_label, dtype=torch.float, device=device)
            output_fake_for_G = D_extra(fake_images)
            lossG = criterion(output_fake_for_G, labels_fake_for_G)
            lossG.backward()
            optimizerG_extra.step()

            extra_d_losses.append(lossD.item())
            extra_g_losses.append(lossG.item())

        # Optionally, print the loss at the end of each epoch
        print(f"[Extra z_dim=100][Epoch {epoch+1}/{epochs_extra}] D_loss: {lossD.item():.4f}, G_loss: {lossG.item():.4f}")

    # Visualization of generated images from the extra training
    G_extra.eval()
    with torch.no_grad():
        test_noise = torch.randn(16, 100, 1, 1, device=device)
        fakes_extra = G_extra(test_noise)
    show(fakes_extra, title="Extra Training: Generated images (z_dim=100, 100 epochs)", n_images=16, nrow=4, figsize=6)

    # Save the extra-trained models
    torch.save(G_extra, "generator_zdim100_100epochs.pkl")
    torch.save(D_extra, "discriminator_zdim100_100epochs.pkl")
    print("Saved generator_zdim100_100epochs.pkl and discriminator_zdim100_100epochs.pkl")

if __name__ == '__main__':
    reproduce_hw4()
```

```
Random seed set to: 42
Using device: cuda

Training GAN with z_dim=10...
Epoch [1/50]:   0%|          | 0/49 [00:00<?, ?it/s]
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-6-6caa6e703160> in <cell line: 0>()
    366 
    367 if __name__ == '__main__':
--> 368     reproduce_hw4()

<ipython-input-6-6caa6e703160> in reproduce_hw4(seed)
    204         for epoch in range(epochs):
    205             loop = tqdm(data_loader, desc=f"Epoch [{epoch+1}/{epochs}]", leave=False)
--> 206             for real_images, _ in loop:
    207                 real_images = real_images.to(device)
    208                 batch_size = real_images.size(0)

/usr/local/lib/python3.11/dist-packages/tqdm/notebook.py in __iter__(self)
    248         try:
    249             it = super().__iter__()
--> 250             for obj in it:
    251                 # return super(tqdm...) will not catch exception
    252                 yield obj

/usr/local/lib/python3.11/dist-packages/tqdm/std.py in __iter__(self)
   1179 
   1180         try:
-> 1181             for obj in iterable:
   1182                 yield obj
   1183                 # Update and possibly print the progressbar.

/usr/local/lib/python3.11/dist-packages/torch/utils/data/dataloader.py in __next__(self)
    699                 # TODO(https://github.com/pytorch/pytorch/issues/76750)
    700                 self._reset()  # type: ignore[call-arg]
--> 701             data = self._next_data()
    702             self._num_yielded += 1
    703             if (

/usr/local/lib/python3.11/dist-packages/torch/utils/data/dataloader.py in _next_data(self)
    755     def _next_data(self):
    756         index = self._next_index()  # may raise StopIteration
--> 757         data = self._dataset_fetcher.fetch(index)  # may raise StopIteration
    758         if self._pin_memory:
    759             data = _utils.pin_memory.pin_memory(data, self._pin_memory_device)

/usr/local/lib/python3.11/dist-packages/torch/utils/data/_utils/fetch.py in fetch(self, possibly_batched_index)
     50                 data = self.dataset.__getitems__(possibly_batched_index)
     51             else:
---> 52                 data = [self.dataset[idx] for idx in possibly_batched_index]
     53         else:
     54             data = self.dataset[possibly_batched_index]

/usr/local/lib/python3.11/dist-packages/torch/utils/data/_utils/fetch.py in <listcomp>(.0)
     50                 data = self.dataset.__getitems__(possibly_batched_index)
     51             else:
---> 52                 data = [self.dataset[idx] for idx in possibly_batched_index]
     53         else:
     54             data = self.dataset[possibly_batched_index]

/usr/local/lib/python3.11/dist-packages/torchvision/datasets/flowers102.py in __getitem__(self, idx)
     82 
     83         if self.transform:
---> 84             image = self.transform(image)
     85 
     86         if self.target_transform:

/usr/local/lib/python3.11/dist-packages/torchvision/transforms/transforms.py in __call__(self, img)
     93     def __call__(self, img):
     94         for t in self.transforms:
---> 95             img = t(img)
     96         return img
     97 

/usr/local/lib/python3.11/dist-packages/torch/nn/modules/module.py in _wrapped_call_impl(self, *args, **kwargs)
   1734             return self._compiled_call_impl(*args, **kwargs)  # type: ignore[misc]
   1735         else:
-> 1736             return self._call_impl(*args, **kwargs)
   1737 
   1738     # torchrec tests the code consistency with the following code

/usr/local/lib/python3.11/dist-packages/torch/nn/modules/module.py in _call_impl(self, *args, **kwargs)
   1745                 or _global_backward_pre_hooks or _global_backward_hooks
   1746                 or _global_forward_hooks or _global_forward_pre_hooks):
-> 1747             return forward_call(*args, **kwargs)
   1748 
   1749         result = None

/usr/local/lib/python3.11/dist-packages/torchvision/transforms/transforms.py in forward(self, img)
    352             PIL Image or Tensor: Rescaled image.
    353         """
--> 354         return F.resize(img, self.size, self.interpolation, self.max_size, self.antialias)
    355 
    356     def __repr__(self) -> str:

/usr/local/lib/python3.11/dist-packages/torchvision/transforms/functional.py in resize(img, size, interpolation, max_size, antialias)
    475             warnings.warn("Anti-alias option is always applied for PIL Image input. Argument antialias is ignored.")
    476         pil_interpolation = pil_modes_mapping[interpolation]
--> 477         return F_pil.resize(img, size=output_size, interpolation=pil_interpolation)
    478 
    479     return F_t.resize(img, size=output_size, interpolation=interpolation.value, antialias=antialias)

/usr/local/lib/python3.11/dist-packages/torchvision/transforms/_functional_pil.py in resize(img, size, interpolation)
    248         raise TypeError(f"Got inappropriate size arg: {size}")
    249 
--> 250     return img.resize(tuple(size[::-1]), interpolation)
    251 
    252 

/usr/local/lib/python3.11/dist-packages/PIL/Image.py in resize(self, size, resample, box, reducing_gap)
   2354                 )
   2355 
-> 2356         return self._new(self.im.resize(size, resample, box))
   2357 
   2358     def reduce(

KeyboardInterrupt: 
```