Generative Adversarial Network for Flower Image Generation

This report details the implementation of a Generative Adversarial Network (GAN) designed to create realistic flower images, focusing on the architecture and training methodology.

GAN Architecture Overview

A GAN consists of two competing neural networks: a generator that creates synthetic images from random noise, and a discriminator that evaluates whether images are real or generated. Through adversarial training, both networks continuously improve their capabilities.

Training Process

The networks engage in a competitive learning process where the generator aims to produce increasingly realistic images while the discriminator enhances its ability to distinguish between authentic and synthetic images.

Project Objective

Our implementation focuses on generating high-quality flower images across 102 distinct categories, demonstrating the GAN's ability to capture complex visual patterns and variations.

Dataset Overview

Dataset Composition

The training data comprises 8,189 images of flowers commonly found in the United Kingdom, spanning 102 different categories.

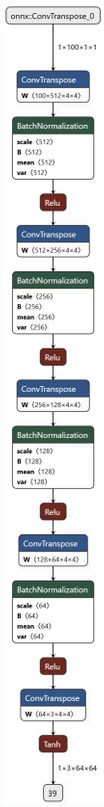
Image Diversity

Each category contains between 40-258 images, featuring varied scales, poses, and lighting conditions to ensure robust model training.

Learning Objectives

This diverse dataset enables the model to learn comprehensive flower characteristics and generate realistic synthetic images that reflect natural variations.

Generator Architecture



The generator is a sophisticated neural network that transforms random noise into realistic flower images through a series of precise mathematical operations. Starting with a 100-dimensional noise vector, it progressively builds up a 64x64x3 RGB image using transposed convolutional layers.

1

Input Layer

Random noise vector (dimension=100) is reshaped into a 3D tensor, preparing it for upsampling

2

Transformation Layers

Multiple transposed convolutional layers progressively upsample the tensor, with batch normalization and ReLU activations enhancing learning capacity

3

Final Layer

Produces 64x64x3 RGB image using tanh activation, normalizing pixels to [-1, 1] range

The generator's architecture is designed for unconditional image generation, meaning it learns the overall distribution of flower images without additional input conditions like categories or labels. Each layer serves a specific purpose:

Upsampling Process

Progressive spatial resolution increase through transposed convolutions while reducing depth dimensions

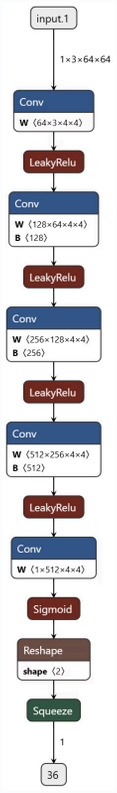
Stabilization Mechanism

Batch normalization after each transposed convolution (except final layer) prevents vanishing gradients

Non-linear Transformations

ReLU activations enable learning of complex relationships between noise input and output image

Discriminator Architecture



The discriminator is a critical component of the GAN architecture that evaluates images and determines whether they are real (from the dataset) or synthetic (created by the generator). It processes each image through a sophisticated neural network to output a probability score between 0 and 1, where 1 indicates a real image and 0 indicates a generated one.

Convolutional Layer Structure

Features multiple convolutional layers that progressively increase feature map depth while reducing spatial resolution by half. This hierarchical structure enables the network to capture both fine details and higher-level patterns in the images.

Normalization Strategy

Implements batch normalization after each convolutional layer, stabilizing feature distributions during the training process. This crucial addition makes the learning process more robust and helps prevent training instability.

Activation Functions

Uses LeakyReLU activations throughout the network, enabling effective learning of complex patterns while maintaining gradient flow for negative inputs. The final layer employs a sigmoid activation to produce a normalized probability output.

Output Processing

Concludes with a specialized convolutional layer that consolidates all extracted features into a single scalar value, representing the network's confidence in the image's authenticity.

Training Procedure

The GAN is trained for 150 epochs, with a batch size of 64, using the Adam optimizer with a learning rate of 0.0002 and betas of (0.5, 0.999). Binary Cross-Entropy (BCE) Loss is used to train both the generator and the discriminator.

Key Training Components

Loss Functions

The discriminator loss measures its ability to correctly classify real and fake images, while the generator loss measures its ability to "fool" the discriminator.

Adversarial Learning

The generator tries to create images that the discriminator would classify as real, while the discriminator learns to better distinguish real from fake images.

Training Process: Iterative Improvement

1

Generator Creates Images

Generator produces synthetic images based on random noise input

2

Discriminator Evaluation

Discriminator assesses if images are real or fake

3

Feedback Loop

Both networks adjust their parameters based on results

4

Progressive Enhancement

Process repeats, leading to increasingly realistic images

Generator

The generator learns to create increasingly realistic images through a series of iterations. It uses the feedback from the discriminator to adjust its internal parameters, aiming to produce images that the discriminator will classify as real. This iterative process leads to the generator generating images that are increasingly indistinguishable from the real images in the dataset.

Discriminator

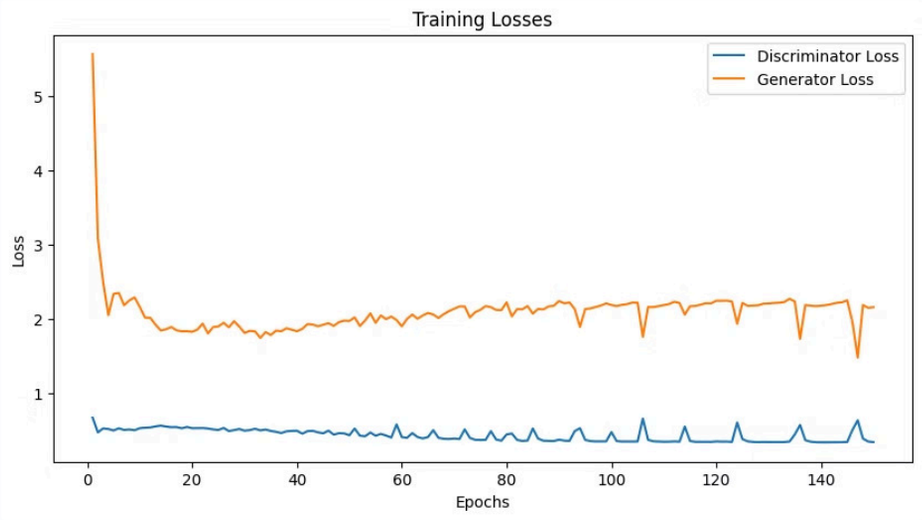
The discriminator also improves its ability to distinguish between real and fake images through iterative training. It adjusts its parameters based on the generated images from the generator, learning to identify patterns and features that distinguish between real and synthetic images. This process ensures that the discriminator remains a challenging opponent for the generator, driving the generator to produce even more realistic images.

Discussion of GAN Limitations

During training, mode collapse was observed in early epochs, with the generator producing repetitive patterns. This issue, common in GANs, was mitigated by employing label smoothing and carefully tuning learning rates. These techniques ensured diversity in the generated images, allowing the model to learn a broader distribution and produce more varied outputs as training progressed.

Training Convergence Plots

The plots display the discriminator and generator losses throughout the training process:



As training progresses, the generator is expected to improve its output quality, leading to a gradual reduction in its loss and a stabilization of the variability. This balance is crucial for achieving a state where both models effectively challenge and complement each other, resulting in high-quality generated images.

- **Discriminator Loss**:

The discriminator demonstrates consistent improvement in distinguishing real images from fake ones, as indicated by the steadily decreasing loss. This suggests the discriminator is learning effectively and becoming better at identifying authentic images compared to those generated by the generator.

- **Generator Loss**:

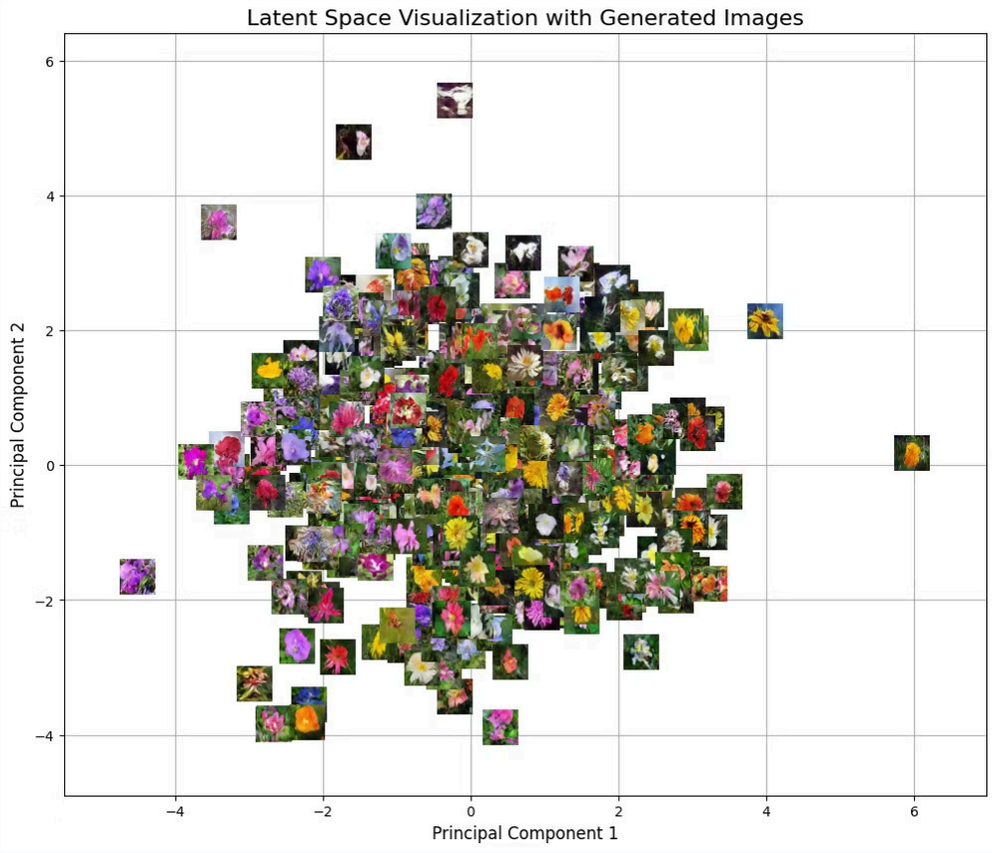
The generator's loss is noticeably higher and exhibits greater variance. This variability indicates the generator is struggling to create convincing images that can fool the discriminator. The fluctuating loss suggests periods of improvement and setbacks, which are common in adversarial training.

Latent Space Visualization and Analysis

The **latent space visualization** provides a powerful tool to interpret and analyze how the generator maps random latent vectors (input noise) into meaningful, structured outputs (images). This visualization technique is crucial for understanding both the diversity of generated images and the generator's ability to capture meaningful features in the latent space.

What is Latent Space?

The latent space represents a compressed dimensional representation where each point corresponds to a unique image generated by the GAN. Through careful manipulation of these latent vectors, we can systematically explore how subtle changes in the input affect the generated images, providing valuable insights into the generator's internal structure and learning process.

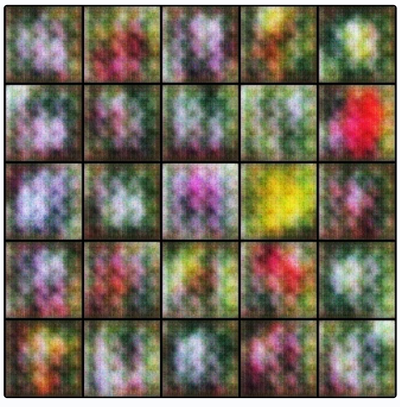


Observations from the Visualization:

- **Cluster Formation**: The generated images in the latent space naturally cluster together, demonstrating that similar features (e.g., flower shapes or colors) are consistently grouped in specific regions of the latent space.
- **Diversity**: The latent space exhibits a comprehensive range of generated outputs, clearly demonstrating the model's success in learning and representing the full underlying distribution of the training data.
- **Blending Features**: The smooth transitions observed between clusters indicate that the GAN has successfully learned to interpolate between different features, enabling the generation of natural-looking hybrid images when sampling points between established clusters.

Evaluation and Future Work

Our GAN architecture evolved through several iterations to achieve optimal performance. The training process exhibited characteristic GAN behavior, initially showing large loss fluctuations due to the inherent instability of adversarial training. As training progressed, we observed a significant improvement in the generator's capabilities, evidenced by decreasing and stabilizing discriminator loss values. The convergence toward equilibrium between generator and discriminator losses indicates successful adversarial training, though some generator loss fluctuations remain—a common phenomenon in GAN training. This progressive improvement suggests strong potential for high-quality image generation.



At Epoch 3, the initial phase of training produced highly abstract and unstructured outputs, as expected in early GAN development. The generated images exhibited extreme blurriness and lacked definitive features, reflecting the generator's preliminary attempts to map random latent vectors to meaningful visual patterns. During this stage, the discriminator was still developing its ability to effectively distinguish between real and generated samples.

By Epoch 20, we observed substantial improvements in image quality. The generator demonstrated markedly better structure and pattern formation compared to Epoch 3, indicating significant progress in the learning process. While the outputs remained somewhat blurry, the emergence of more distinct features suggested positive trajectory in the model's development, though further training would be necessary to achieve optimal clarity and detail.

After training the model for **110 epochs**, I observed that while there were notable improvements in image quality, the results still lacked sufficient clarity and fine details. By epoch 108, we could see progress in image generation, but the outputs weren't meeting our quality standards. This motivated me to extend the training process to **150 epochs** to achieve better image clarity and overall quality.

FINAL

With the final version of my GAN, training took approximately **2 to 3 minutes per epoch**, allowing me to complete **150 epochs**. Although additional epochs could further enhance image quality, the final results indicate significant improvements in detail, sharpness, and overall realism compared to earlier attempts.

Key Research Conclusions

1

Discriminator Success

The discriminator's performance improved consistently, showing its increasing ability to distinguish real and fake images effectively. Its stable performance helped guide the generator to produce more realistic outputs over time.

2

Generator Performance

The generator exhibited high variance in its loss, likely due to adversarial training dynamics and potential hyperparameter optimizations. However, its ability to learn and replicate underlying features of the dataset improved significantly across epochs.

3

Latent Space Insights

The interpolation in the latent space revealed smooth transitions between different generated samples, demonstrating the generator's ability to capture a continuous and meaningful representation of the data distribution. This highlights the generator's success in modeling the latent space effectively.

4

Overall Results

The generated images demonstrate that the model successfully captures underlying features of the dataset, producing recognizable outputs. Further training, the use of more advanced architectures, and higher resolution models could enhance image accuracy and realism even further.

In [ ]:

```
# %%
import torch
import torch.nn as nn

import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
import torchvision.datasets as dsets
from torch.utils.data import DataLoader
from torchvision import transforms
from torchvision.utils import save_image

import pickle, json
import os

import matplotlib.pyplot as plt


from gan import device, Generator,latent_dim,Discriminator,load_pickle_as_state_dict

device = device



from torchvision.utils import save_image
import os
import matplotlib.pyplot as plt

def reproduce_hw4(generator_weights_path, discriminator_weights_path, num_images=24, save_path="/home/student/"):

    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

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

    # Load model weights from `.pkl` files
    generator.load_state_dict(load_pickle_as_state_dict(generator_weights_path))
    discriminator.load_state_dict(load_pickle_as_state_dict(discriminator_weights_path))

    generator.eval()
    discriminator.eval()

    # Generate new images

    with torch.no_grad():
        z = torch.randn(num_images, latent_dim, 1, 1).to(device)
        gen_imgs = generator(z)

    # Save the generated images
    os.makedirs(save_path, exist_ok=True)
    save_image((gen_imgs + 1) / 2, os.path.join(save_path, "HW4_generated_images_final.png"), nrow=8, normalize=True)


    # Display the generated images

    plt.figure(figsize=(18, 18))
    for j, img in enumerate(gen_imgs):
        img = img.cpu().detach()
        img = (img + 1) / 2
        img = img.permute(1, 2, 0)
        plt.subplot(6, 4, j + 1)
        plt.imshow(img.numpy().squeeze())
        plt.axis('off')
    plt.show()




if __name__ == "__main__":
    reproduce_hw4("/home/student/HW4_generator_weights_final_mirel.pkl",
              "/home/student/HW4_discriminator_weights_final_mirel.pkl")
```

In [ ]:

```python
import torch
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms
from PIL import Image
import os
import torch.nn as nn
import torch.optim as optim
from torchvision.utils import save_image
import matplotlib.pyplot as plt
import numpy as np
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt

# Device setup
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

# Custom Dataset for flat directory
class CustomDataset(Dataset):
    def __init__(self, folder_path, transform=None):
        self.folder_path = folder_path
        self.image_filenames = [f for f in os.listdir(folder_path) if f.lower().endswith(('.jpg', '.jpeg', '.png'))]
        if len(self.image_filenames) == 0:
            raise ValueError(f"No images found in directory: {folder_path}")
        print(f"Found {len(self.image_filenames)} images in {folder_path}")
        self.transform = transform

    def __len__(self):  # תיקון len
        return len(self.image_filenames)

    def __getitem__(self, idx):  # תיקון getitem
        img_path = os.path.join(self.folder_path, self.image_filenames[idx])
        image = Image.open(img_path).convert("RGB")
        if self.transform:
            image = self.transform(image)
        return image, 0  # No labels for GANs

# Paths and hyperparameters
data_dir = r"/home/student/102flowers/jpg"  # Update with your path
image_size = 64
batch_size = 64
latent_dim = 100
epochs = 150

# Transformations
transform = transforms.Compose([
    transforms.Resize((image_size, image_size)),
    transforms.ToTensor(),
    transforms.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5])
])

# Dataset and DataLoader
dataset = CustomDataset(folder_path=data_dir, transform=transform)
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)

# Weight Initialization
def weights_init_normal(m):
    if isinstance(m, (nn.ConvTranspose2d, nn.Conv2d)):
        nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif isinstance(m, nn.BatchNorm2d):
        nn.init.normal_(m.weight.data, 1.0, 0.02)
        nn.init.constant_(m.bias.data, 0)

# Generator Model
class Generator(nn.Module):
    def __init__(self, latent_dim):  # תיקון init
        super(Generator, self).__init__()
        self.model = nn.Sequential(
            nn.ConvTranspose2d(latent_dim, 512, kernel_size=4, stride=1, padding=0, bias=False),
            nn.BatchNorm2d(512),
            nn.ReLU(True),
            nn.ConvTranspose2d(512, 256, kernel_size=4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(256),
            nn.ReLU(True),
            nn.ConvTranspose2d(256, 128, kernel_size=4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(128),
            nn.ReLU(True),
            nn.ConvTranspose2d(128, 64, kernel_size=4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(True),
            nn.ConvTranspose2d(64, 3, kernel_size=4, stride=2, padding=1, bias=False),
            nn.Tanh()
        )

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

```python
# Discriminator Model
class Discriminator(nn.Module):
    def __init__(self):  # תיקון init
        super(Discriminator, self).__init__()
        self.model = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=4, stride=2, padding=1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(128, 256, kernel_size=4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(256, 512, kernel_size=4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(512),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(512, 1, kernel_size=4, stride=1, padding=0, bias=False),
            nn.Sigmoid()
        )

    def forward(self, img):
        return self.model(img).view(-1, 1).squeeze(1)

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

# Loss and optimizers
adversarial_loss = nn.BCELoss()
optimizer_G = optim.Adam(generator.parameters(), lr=0.0002, betas=(0.5, 0.999))
optimizer_D = optim.Adam(discriminator.parameters(), lr=0.0002, betas=(0.5, 0.999))

# Lists to store losses
discriminator_losses = []
generator_losses = []
avg_d_losses = []
avg_g_losses = []

# Training Loop
for epoch in range(epochs):
    epoch_d_losses = []
    epoch_g_losses = []

    for i, (imgs, _) in enumerate(dataloader):
        imgs = imgs.to(device)
        batch_size = imgs.size(0)

        # Real and fake labels
        valid = torch.ones(batch_size, device=device) * 0.9  # Label smoothing
        fake = torch.zeros(batch_size, device=device) + 0.1

        # Train Discriminator
        optimizer_D.zero_grad()
        real_loss = adversarial_loss(discriminator(imgs), valid)

        z = torch.randn(batch_size, latent_dim, 1, 1, device=device)
        gen_imgs = generator(z)
        fake_loss = adversarial_loss(discriminator(gen_imgs.detach()), fake)

        d_loss = (real_loss + fake_loss) / 2
        d_loss.backward()
        optimizer_D.step()

        # Train Generator
        optimizer_G.zero_grad()
        g_loss = adversarial_loss(discriminator(gen_imgs), valid)
        g_loss.backward()
        optimizer_G.step()

        # Record losses for iterations
        discriminator_losses.append(d_loss.item())
        generator_losses.append(g_loss.item())

        # Save generated samples
        if i % 50 == 0:
            print(f"[Epoch {epoch}/{epochs}] [Batch {i}/{len(dataloader)}] "
                  f"[D loss: {d_loss.item():.4f}] [G loss: {g_loss.item():.4f}]")
        num_images_to_save = 25
        z = torch.randn(num_images_to_save, latent_dim, 1, 1, device=device)
        gen_imgs = generator(z)
        normalized_gen_imgs = (gen_imgs + 1) / 2
        save_image(normalized_gen_imgs, f"HW4_MIREL-generated_images_epoch_{epoch}.png", nrow=5)

    # Save losses
    avg_d_losses.append(np.mean(epoch_d_losses))
    avg_g_losses.append(np.mean(epoch_g_losses))
```

```python
from sklearn.decomposition import PCA
from matplotlib.offsetbox import OffsetImage, AnnotationBbox
import matplotlib.pyplot as plt
import torch

def visualize_latent_space_with_images(generator, latent_dim, device, num_samples=100):
    """
    Visualizes the latent space with the generated images embedded in the graph.
    """
    generator.eval()

    # Step 1: Generate latent vectors and images
    with torch.no_grad():
        z = torch.randn(num_samples, latent_dim, 1, 1).to(device)  # Latent vectors
        gen_imgs = generator(z).cpu()  # Generate images
        gen_imgs = (gen_imgs + 1) / 2  # Normalize images to [0, 1] for display

    # Step 2: Flatten latent vectors and apply PCA
    z_flat = z.view(num_samples, -1).cpu().numpy()  # Flatten latent vectors
    pca = PCA(n_components=2)  # Reduce to 2D
    reduced_latent_space = pca.fit_transform(z_flat)

    # Step 3: Plot latent space with images
    fig, ax = plt.subplots(figsize=(12, 10))
    for i, (x, y) in enumerate(reduced_latent_space):
        img = gen_imgs[i].permute(1, 2, 0).numpy()  # Convert image to (H, W, C) for display
        img = (img - img.min()) / (img.max() - img.min())  # Normalize to [0, 1]
        imagebox = OffsetImage(img, zoom=0.4)  # Create image thumbnail
        ab = AnnotationBbox(imagebox, (x, y), frameon=False)  # Place image on graph
        ax.add_artist(ab)

    ax.set_title("Latent Space Visualization with Generated Images", fontsize=16)
    ax.set_xlabel("Principal Component 1", fontsize=12)
    ax.set_ylabel("Principal Component 2", fontsize=12)
    ax.set_xlim(reduced_latent_space[:, 0].min() - 1, reduced_latent_space[:, 0].max() + 1)
    ax.set_ylim(reduced_latent_space[:, 1].min() - 1, reduced_latent_space[:, 1].max() + 1)
    plt.grid()
    plt.show()


visualize_latent_space_with_images(generator, latent_dim, device, num_samples=100)
```

```python
import torch
import pickle

# שמירת ה-state_dict עם pickle
with open('/home/student/HW4_generator_weights_final_mirel.pkl', 'wb') as f:
    pickle.dump(generator.state_dict(), f)

with open('/home/student/HW4_discriminator_weights_final_mirel.pkl', 'wb') as f:
    pickle.dump(discriminator.state_dict(), f)

# In[262]:

def load_pickle_as_state_dict(pickle_file_path):
    with open(pickle_file_path, 'rb') as f:
        return pickle.load(f)

# טעינת המשקלים למודל
generator.load_state_dict(load_pickle_as_state_dict('/home/student/HW4_generator_weights_final_mirel.pkl'))
discriminator.load_state_dict(load_pickle_as_state_dict('/home/student/HW4_discriminator_weights_final_mirel.pkl'))
```

```python
def load_pickle_as_state_dict(pickle_file_path):
    with open(pickle_file_path, 'rb') as f:
        return pickle.load(f)


epochs = range(1, len(discriminator_losses) // len(dataloader) + 1)
avg_d_losses = [np.mean(discriminator_losses[i * len(dataloader):(i + 1) * len(dataloader)]) for i in range(len(epochs))]
avg_g_losses = [np.mean(generator_losses[i * len(dataloader):(i + 1) * len(dataloader)]) for i in range(len(epochs))]

plt.figure(figsize=(10, 5))
plt.plot(epochs, avg_d_losses, label='Discriminator Loss')
plt.plot(epochs, avg_g_losses, label='Generator Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training Losses')
plt.legend()
plt.show()


# In[260]:
```

```
plt.figure(figsize=(10, 5))
plt.plot(discriminator_losses, label='Discriminator Loss', alpha=0.7)
plt.plot(generator_losses, label='Generator Loss', alpha=0.7)
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.title('Training Losses')
plt.legend()
plt.show()
```