dcgan-face-generation-1

November 10, 2024

[1]:
```
!pip install kaggle
```

```
Requirement already satisfied: kaggle in /usr/local/lib/python3.10/dist-packages
(1.6.17)
Requirement already satisfied: six>=1.10 in /usr/local/lib/python3.10/dist-
packages (from kaggle) (1.16.0)
Requirement already satisfied: certifi>=2023.7.22 in
/usr/local/lib/python3.10/dist-packages (from kaggle) (2024.8.30)
Requirement already satisfied: python-dateutil in
/usr/local/lib/python3.10/dist-packages (from kaggle) (2.8.2)
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-
packages (from kaggle) (2.32.3)
Requirement already satisfied: tqdm in /usr/local/lib/python3.10/dist-packages
(from kaggle) (4.66.6)
Requirement already satisfied: python-slugify in /usr/local/lib/python3.10/dist-
packages (from kaggle) (8.0.4)
Requirement already satisfied: urllib3 in /usr/local/lib/python3.10/dist-
packages (from kaggle) (2.2.3)
Requirement already satisfied: bleach in /usr/local/lib/python3.10/dist-packages
(from kaggle) (6.2.0)
Requirement already satisfied: webencodings in /usr/local/lib/python3.10/dist-
packages (from bleach->kaggle) (0.5.1)
Requirement already satisfied: text-unidecode>=1.3 in
/usr/local/lib/python3.10/dist-packages (from python-slugify->kaggle) (1.3)
Requirement already satisfied: charset-normalizer<4,>=2 in
/usr/local/lib/python3.10/dist-packages (from requests->kaggle) (3.4.0)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-
packages (from requests->kaggle) (3.10)
```

[2]:
```
import os
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.transforms as transforms
import torchvision.datasets as dsets
import torchvision.utils as vutils
from torch.utils.data import DataLoader
import kagglehub
```

```
# Download CelebA dataset using kagglehub
path = kagglehub.dataset_download("jessicali9530/celeba-dataset")
dataset_path = path  # path to the downloaded dataset

print("Path to dataset files:", dataset_path)
```

```
Downloading from
https://www.kaggle.com/api/v1/datasets/download/jessicali9530/celeba-
dataset?dataset_version_number=2…

100%|      | 1.33G/1.33G [00:16<00:00, 87.1MB/s]

Extracting files…


Path to dataset files: /root/.cache/kagglehub/datasets/jessicali9530/celeba-
dataset/versions/2
```

[9]:
```
# Hyperparameters
image_size = 64
batch_size = 128
nz = 100         # Latent vector size
lr = 0.0002      # Learning rate
beta1 = 0.5      # Beta1 hyperparam for Adam optimizer
num_epochs = 50
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Dataset loading
transform = transforms.Compose([
    transforms.Resize(image_size),
    transforms.CenterCrop(image_size),
    transforms.ToTensor(),
    transforms.Normalize([0.5] * 3, [0.5] * 3)
])

dataset = dsets.ImageFolder(root=dataset_path, transform=transform)
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)
```

[10]:
```
import matplotlib.pyplot as plt
import numpy as np
import torchvision

# Function to show an image
def imshow(img):
    img = img / 2 + 0.5  # unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
```

```
    plt.show()

# Get some random training images
dataiter = iter(dataloader)
images, _ = next(dataiter)

# Show images
imshow(torchvision.utils.make_grid(images[:8]))  # Show a grid of 8 images
```

[11]:
```
# Weight initialization
def weights_init(m):
    if isinstance(m, (nn.Conv2d, nn.ConvTranspose2d, nn.BatchNorm2d)):
        nn.init.normal_(m.weight.data, 0.0, 0.02)
```

[12]:
```
# Generator model
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.main = nn.Sequential(
            nn.ConvTranspose2d(nz, 512, 4, 1, 0, bias=False),
            nn.BatchNorm2d(512),
            nn.ReLU(True),
            nn.ConvTranspose2d(512, 256, 4, 2, 1, bias=False),
            nn.BatchNorm2d(256),
            nn.ReLU(True),
            nn.ConvTranspose2d(256, 128, 4, 2, 1, bias=False),
            nn.BatchNorm2d(128),
            nn.ReLU(True),
            nn.ConvTranspose2d(128, 64, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(True),
            nn.ConvTranspose2d(64, 3, 4, 2, 1, bias=False),
            nn.Tanh()
        )

    def forward(self, input):
        return self.main(input)
```

[13]:
```
# Discriminator model
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.main = nn.Sequential(
            nn.Conv2d(3, 64, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(64, 128, 4, 2, 1, bias=False),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(128, 256, 4, 2, 1, bias=False),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(256, 512, 4, 2, 1, bias=False),
            nn.BatchNorm2d(512),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(512, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
        )

    def forward(self, input):
        return self.main(input)
```

[14]:
```
# Create models
netG = Generator().to(device)
netG.apply(weights_init)

netD = Discriminator().to(device)
netD.apply(weights_init)
```

[14]:
```
Discriminator(
  (main): Sequential(
    (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1),
bias=False)
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1),
bias=False)
    (3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (4): LeakyReLU(negative_slope=0.2, inplace=True)
    (5): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1),
bias=False)
    (6): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (7): LeakyReLU(negative_slope=0.2, inplace=True)
    (8): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1),
bias=False)
```

```
    (9): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (10): LeakyReLU(negative_slope=0.2, inplace=True)
    (11): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (12): Sigmoid()
  )
)
```

[15]:
```python
# Loss function and optimizers
criterion = nn.BCELoss()
fixed_noise = torch.randn(64, nz, 1, 1, device=device)
optimizerD = optim.Adam(netD.parameters(), lr=lr, betas=(beta1, 0.999))
optimizerG = optim.Adam(netG.parameters(), lr=lr, betas=(beta1, 0.999))
```

[20]:
```python
# Define training step function
def train_step(data):
    # Extract images from data (assuming data is a tuple or list: (images,␣
 ↪labels))
    real_images = data[0].to(device)
    label_real = torch.full((real_images.size(0),), 1., device=device)
    label_fake = torch.full((real_images.size(0),), 0., device=device)

    # Update Discriminator: maximize log(D(x)) + log(1 - D(G(z)))
    netD.zero_grad()
    output_real = netD(real_images).view(-1)
    lossD_real = criterion(output_real, label_real)
    lossD_real.backward()

    noise = torch.randn(real_images.size(0), nz, 1, 1, device=device)
    fake_images = netG(noise)
    output_fake = netD(fake_images.detach()).view(-1)
    lossD_fake = criterion(output_fake, label_fake)
    lossD_fake.backward()
    optimizerD.step()

    # Update Generator: maximize log(D(G(z)))
    netG.zero_grad()
    output = netD(fake_images).view(-1)
    lossG = criterion(output, label_real)
    lossG.backward()
    optimizerG.step()

    return lossD_real + lossD_fake, lossG, fake_images
```

[21]:
```python
# Recursive function to train the model for a given number of epochs
def train_epoch(epoch, num_epochs):
    if epoch >= num_epochs:
```

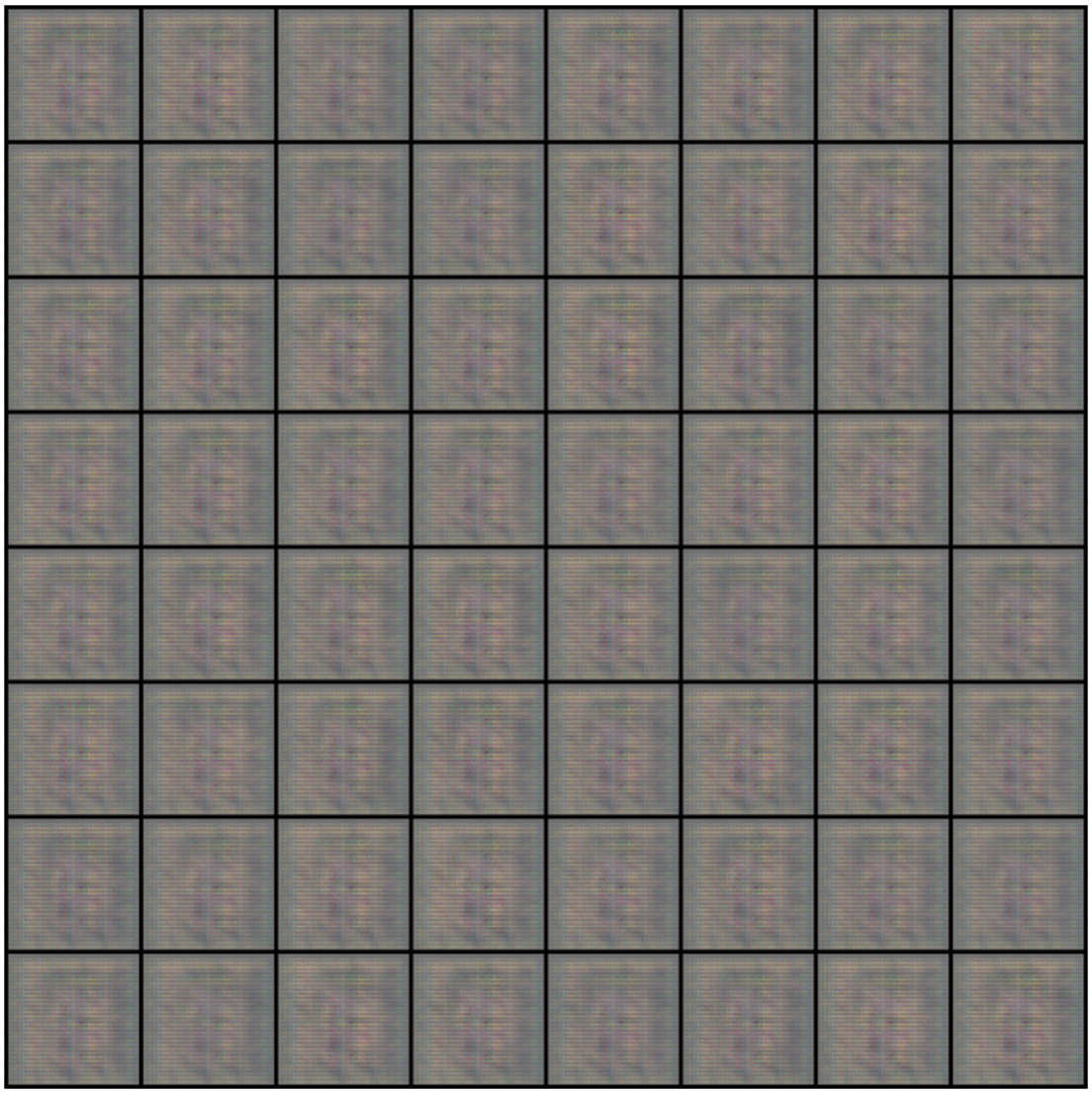
```
        print("Training completed.")
        return

    dataiter = iter(dataloader)
    lossD, lossG, fake_images = 0, 0, None
    try:
        data = next(dataiter)
        lossD, lossG, fake_images = train_step(data)
    except StopIteration:
        pass

    # Output progress for every epoch
    vutils.save_image(fake_images.detach(), f"fake_samples_epoch_{epoch}.png", normalize=True)
    print(f"Epoch [{epoch + 1}/{num_epochs}] Loss_D: {lossD:.4f} Loss_G: {lossG:.4f}")

    # Call next epoch recursively
    train_epoch(epoch + 1, num_epochs)

# Start training
train_epoch(0, num_epochs)
```

```
Epoch [1/50] Loss_D: 0.0158 Loss_G: 5.2741
Epoch [2/50] Loss_D: 0.0147 Loss_G: 5.1243
Epoch [3/50] Loss_D: 0.0186 Loss_G: 5.0230
Epoch [4/50] Loss_D: 0.0243 Loss_G: 5.2255
Epoch [5/50] Loss_D: 0.0203 Loss_G: 5.3063
Epoch [6/50] Loss_D: 0.0232 Loss_G: 5.2113
Epoch [7/50] Loss_D: 0.0239 Loss_G: 5.1544
Epoch [8/50] Loss_D: 0.0227 Loss_G: 5.1619
Epoch [9/50] Loss_D: 0.0193 Loss_G: 5.1851
Epoch [10/50] Loss_D: 0.0195 Loss_G: 5.1448
Epoch [11/50] Loss_D: 0.0172 Loss_G: 5.1133
Epoch [12/50] Loss_D: 0.0181 Loss_G: 5.1242
Epoch [13/50] Loss_D: 0.0176 Loss_G: 5.1390
Epoch [14/50] Loss_D: 0.0173 Loss_G: 5.1526
Epoch [15/50] Loss_D: 0.0166 Loss_G: 5.1637
Epoch [16/50] Loss_D: 0.0168 Loss_G: 5.1593
Epoch [17/50] Loss_D: 0.0158 Loss_G: 5.1579
Epoch [18/50] Loss_D: 0.0162 Loss_G: 5.1476
Epoch [19/50] Loss_D: 0.0161 Loss_G: 5.1434
Epoch [20/50] Loss_D: 0.0162 Loss_G: 5.1552
Epoch [21/50] Loss_D: 0.0170 Loss_G: 5.1504
Epoch [22/50] Loss_D: 0.0159 Loss_G: 5.1664
Epoch [23/50] Loss_D: 0.0175 Loss_G: 5.1564
Epoch [24/50] Loss_D: 0.0172 Loss_G: 5.1518
```

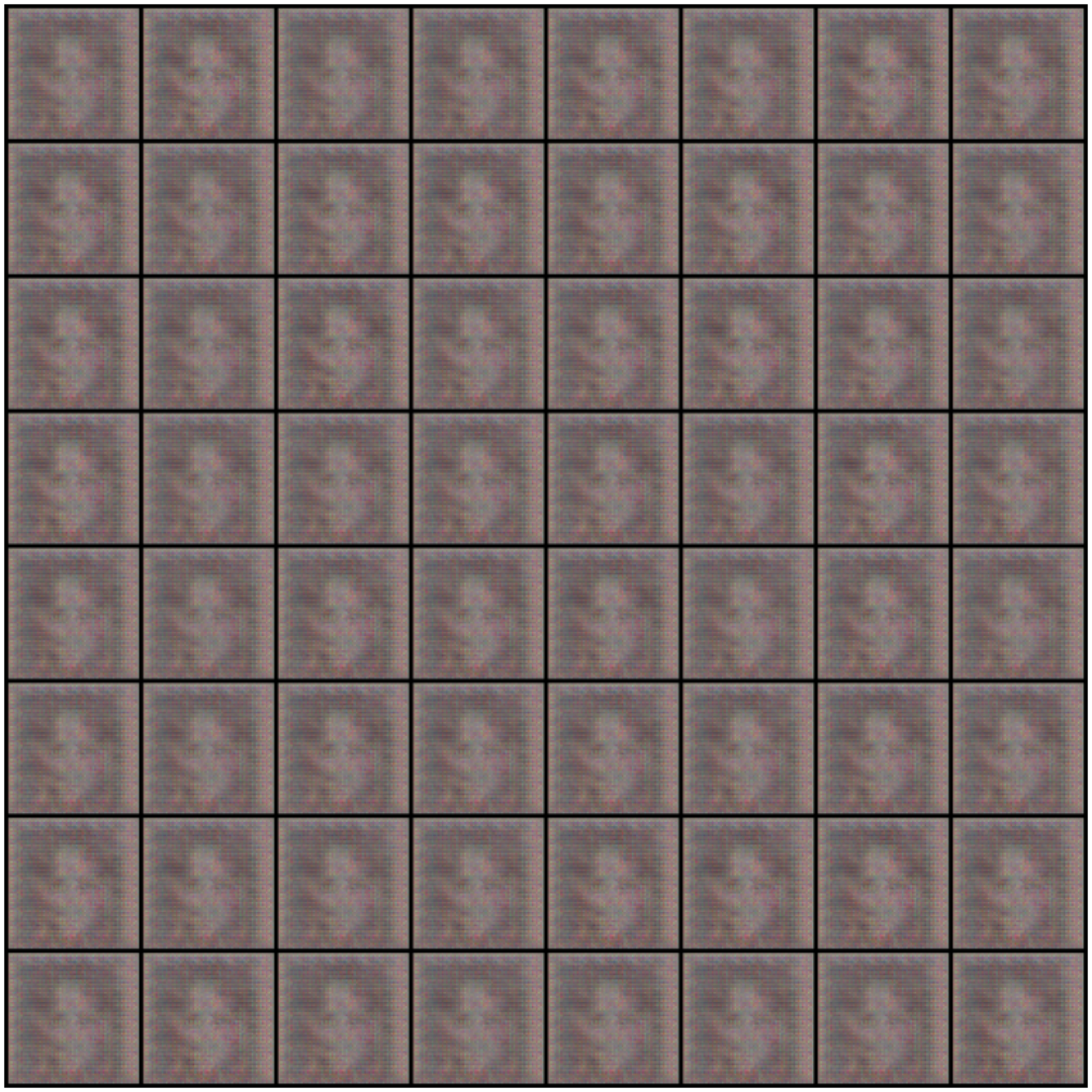
```
Epoch [25/50] Loss_D: 0.0169 Loss_G: 5.1633
Epoch [26/50] Loss_D: 0.0187 Loss_G: 5.1504
Epoch [27/50] Loss_D: 0.0230 Loss_G: 5.1364
Epoch [28/50] Loss_D: 0.0203 Loss_G: 5.2338
Epoch [29/50] Loss_D: 0.0173 Loss_G: 5.3329
Epoch [30/50] Loss_D: 0.0137 Loss_G: 5.3658
Epoch [31/50] Loss_D: 0.0165 Loss_G: 5.2800
Epoch [32/50] Loss_D: 0.0161 Loss_G: 5.2011
Epoch [33/50] Loss_D: 0.0153 Loss_G: 5.2210
Epoch [34/50] Loss_D: 0.0151 Loss_G: 5.2651
Epoch [35/50] Loss_D: 0.0150 Loss_G: 5.2745
Epoch [36/50] Loss_D: 0.0153 Loss_G: 5.2396
Epoch [37/50] Loss_D: 0.0142 Loss_G: 5.2545
Epoch [38/50] Loss_D: 0.0161 Loss_G: 5.1979
Epoch [39/50] Loss_D: 0.0170 Loss_G: 5.2005
Epoch [40/50] Loss_D: 0.0158 Loss_G: 5.2350
Epoch [41/50] Loss_D: 0.0151 Loss_G: 5.2597
Epoch [42/50] Loss_D: 0.0140 Loss_G: 5.3211
Epoch [43/50] Loss_D: 0.0143 Loss_G: 5.3175
Epoch [44/50] Loss_D: 0.0143 Loss_G: 5.3104
Epoch [45/50] Loss_D: 0.0136 Loss_G: 5.3201
Epoch [46/50] Loss_D: 0.0135 Loss_G: 5.3697
Epoch [47/50] Loss_D: 0.0133 Loss_G: 5.3754
Epoch [48/50] Loss_D: 0.0162 Loss_G: 5.2178
Epoch [49/50] Loss_D: 0.0139 Loss_G: 5.3399
Epoch [50/50] Loss_D: 0.0137 Loss_G: 5.4336
Training completed.
```

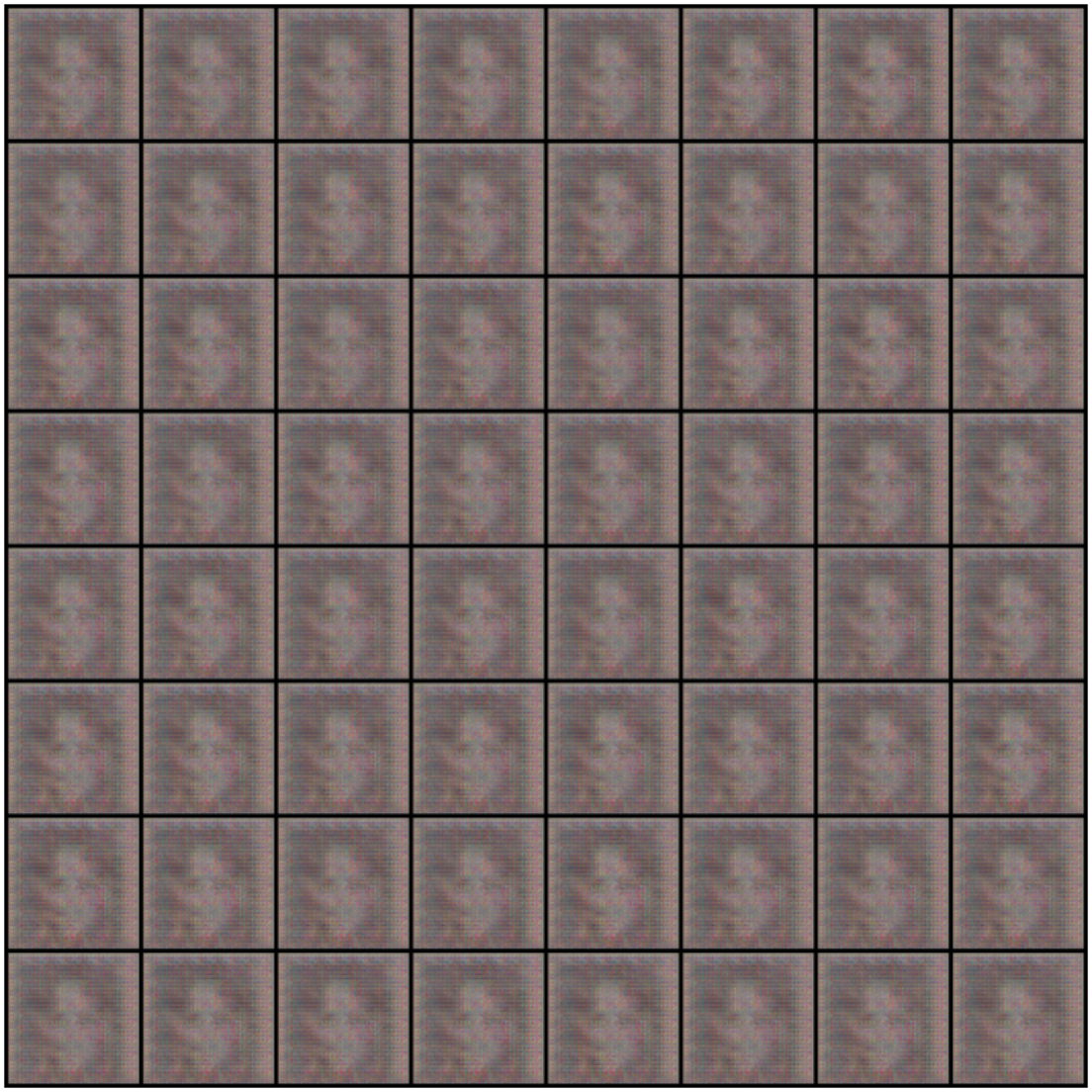
[33]:
```python
import matplotlib.pyplot as plt
import torchvision.utils as vutils
import numpy as np

def show_generated_images(fake_images, num_images=64):
    # Convert to grid of images
    # The 'range' argument is deprecated, use 'value_range' instead
    grid = vutils.make_grid(fake_images[:num_images], nrow=8, normalize=True,␣
 ↪value_range=(-1, 1))

    # Convert to numpy for displaying with matplotlib
    np_grid = grid.cpu().numpy().transpose((1, 2, 0))

    # Plot the images
    plt.figure(figsize=(10, 10))
    plt.imshow(np_grid)
    plt.axis('off')  # Hide axes for better visual appearance
    plt.show()
```

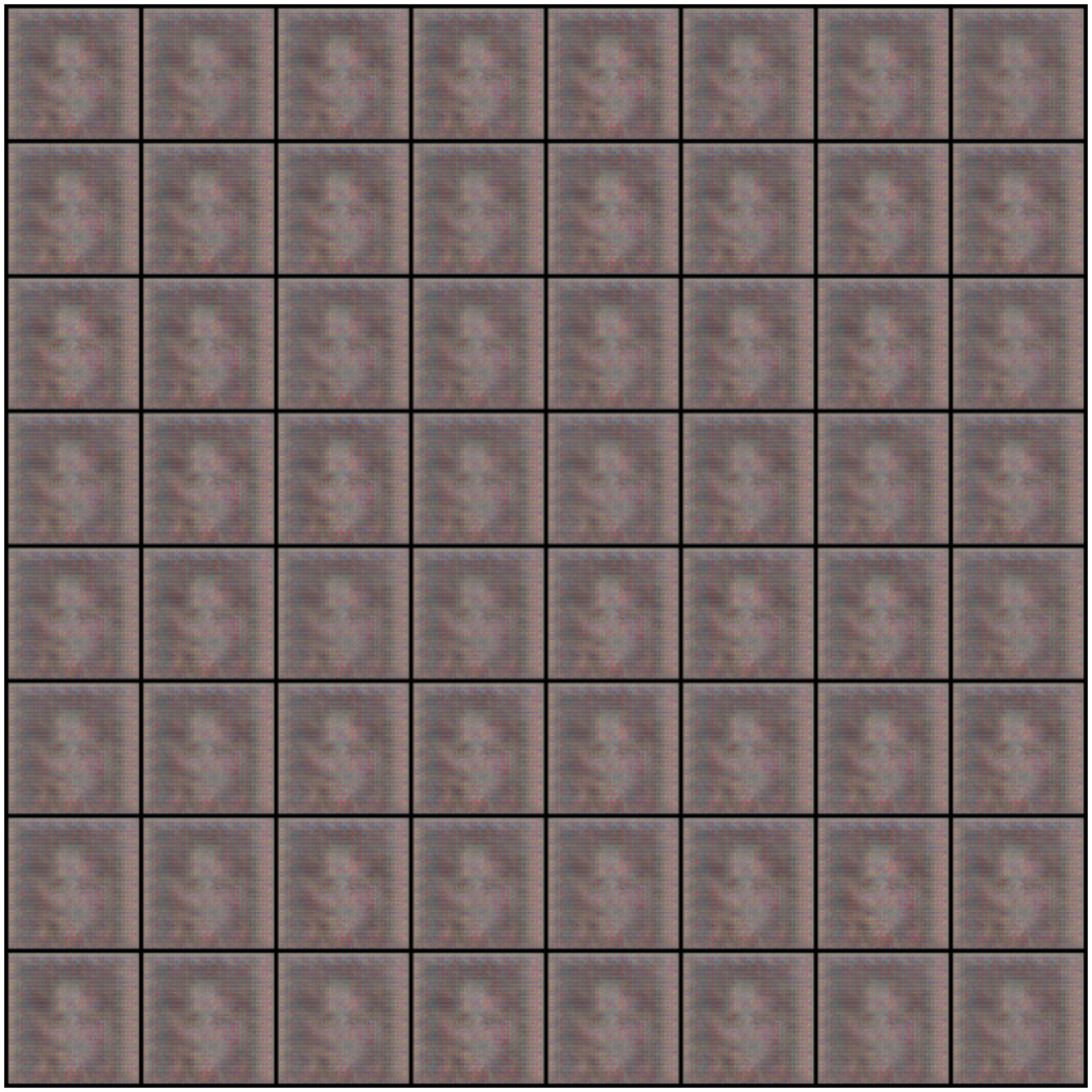
```
# Assuming fake_images is a batch of generated images from the generator
show_generated_images(fake_images)
```

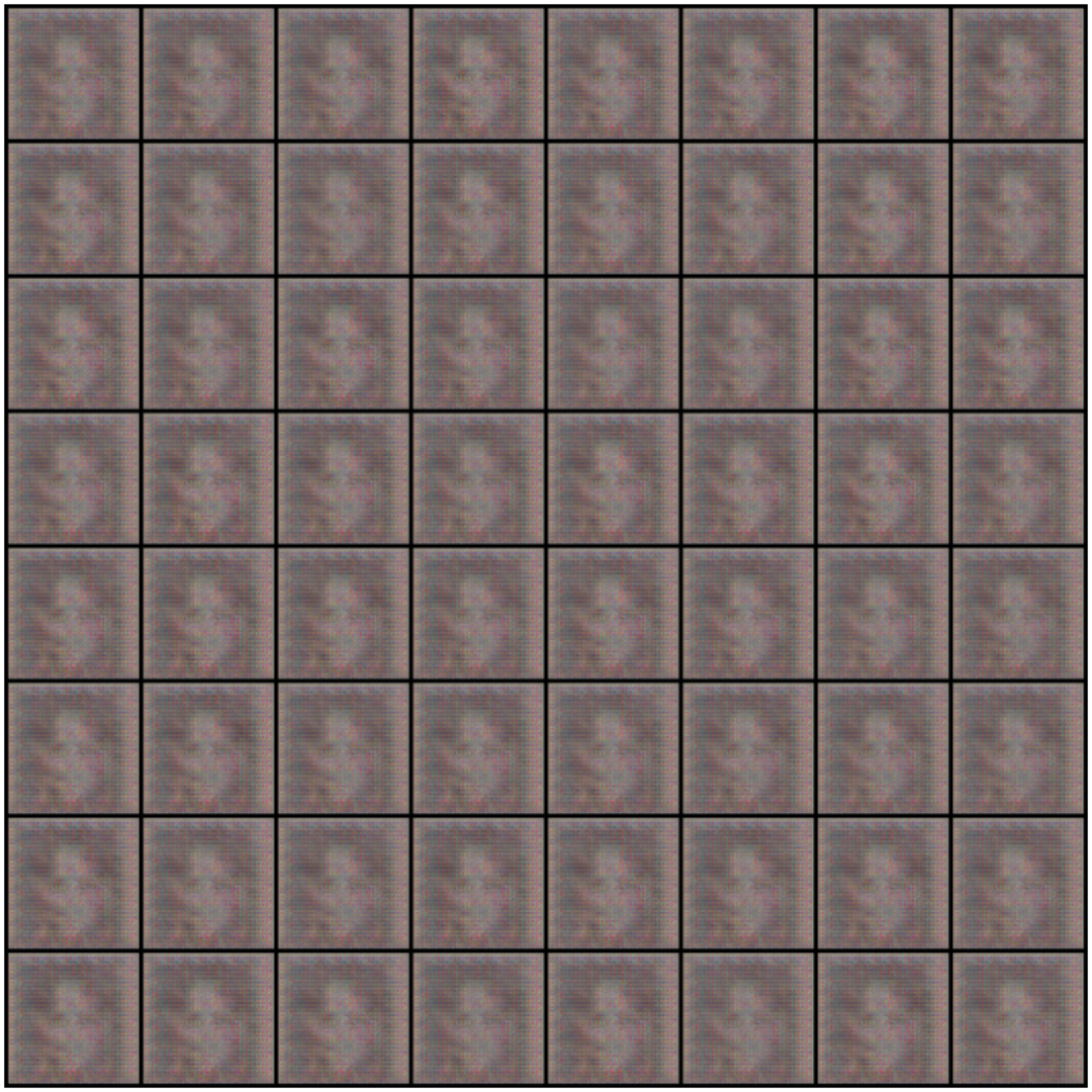


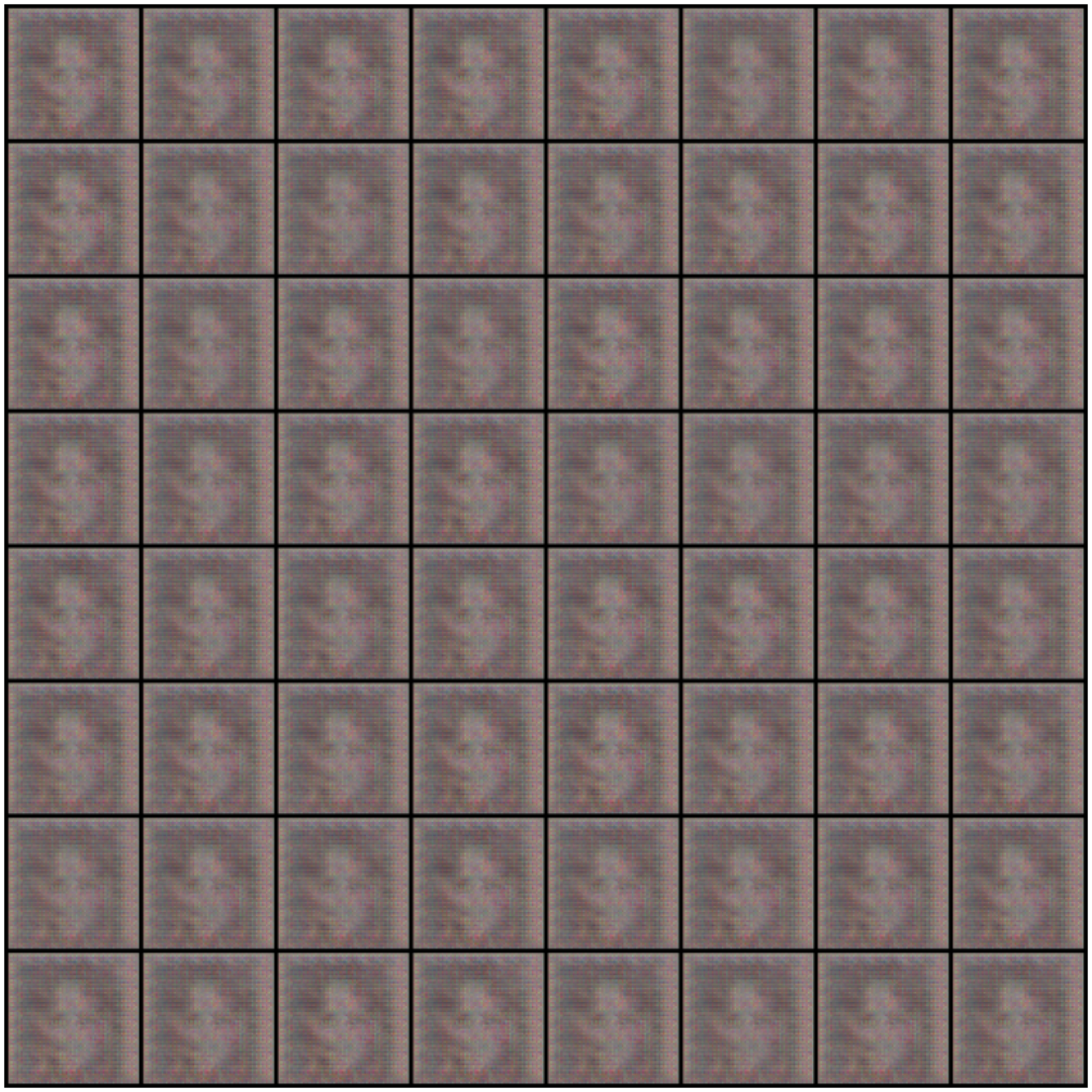
[34]:
```
def display_multiple_batches(generator, num_batches=5, batch_size=64):
    for batch_num in range(num_batches):
        noise = torch.randn(batch_size, nz, 1, 1, device=device)
        generated_batch = generator(noise)
        show_generated_images(generated_batch)

# Display 5 batches of generated images
display_multiple_batches(netG, num_batches=5)
```



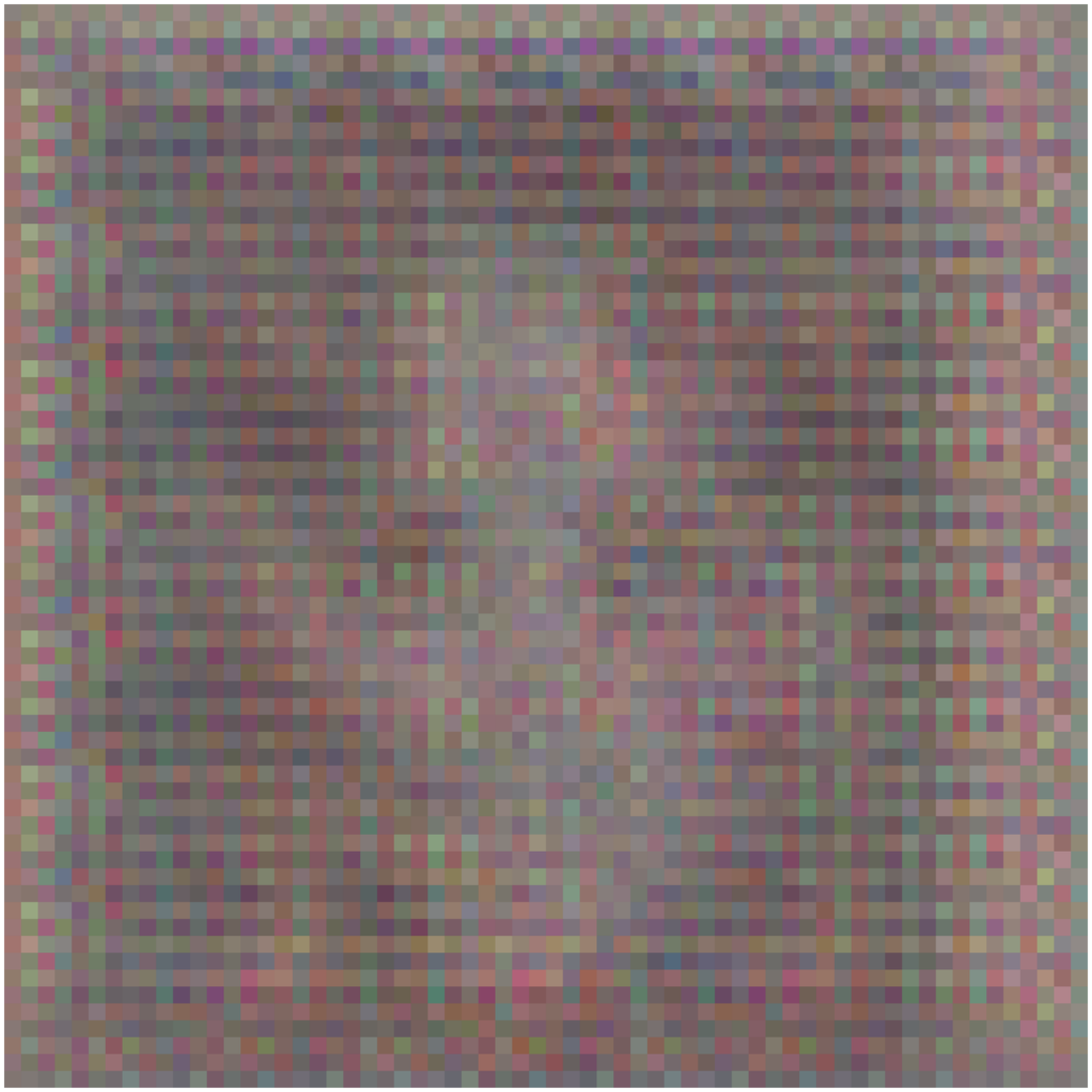








[36]:
```
# Generate random noise as input
noise = torch.randn(1, nz, 1, 1, device=device)  # 'nz' is the size of the␣
 ↪noise vector

# Generate a fake human face
fake_image = netG(noise)

# Display the generated image
show_generated_images(fake_image)
```



[ ]: