Exercise Sheet 8: Generative Adversarial Networks

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.nn.utils import spectral_norm
import torch.optim as optim

from sklearn.manifold import TSNE
import torchvision
import torchvision.utils as vutils
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

import os
import numpy as np
import random
import time
import matplotlib.pyplot as plt
plt.style.use('ggplot')

# folder path
data_path = './data'
model_path = './model'
result_path = './results'

# random seed np/torch
seed = 42
random.seed(seed)
np.random.seed(seed)
random.seed(seed)
torch.manual_seed(seed)

# hyperparameters
img_size = 32
latent_dim = 100
batch_size = 128
num_epochs = 5
lr = 0.0001
momentum = 0.5
betas = (0.5, 0.999)


# set device
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

```
# Create folder if not exist
if not os.path.exists(data_path):
    os.makedirs(data_path)
if not os.path.exists(model_path):
    os.makedirs(model_path)
if not os.path.exists(result_path):
    os.makedirs(result_path)
```

Task 1.1: Derive optial discriminator D*

Use equation (1) as a starting point to derive the optimal discriminator D∗ in terms of data probability pdata (x) and generator probability pG (x). Assume generator G is fixed.

The objective function V(D,G) is:

$$V(D,G) = E_{x p_{data}(x)}[log(D(x))] + E_{z p(z)}[log(1 - D(G(z)))]$$

Rewriting the objective function in terms of p_{data}(x) and p_z(x):

$ V(D,G) = \int{p_{data}(x) log(D(x)) dx + \int{p_g(x) log(1 - D(g(z))) dx}} \ = \int{p_{data}(x) log(D(x)) dx + p_g(x) log(1-D(x)) dx} $

Task 1.2: Find optimal point minimizing V

Use the obtained D∗ to find the optimal point minimizing V. What value does D∗ have at this point and what would this value imply?

Taking the derivative of V with respect to D(x) and setting it to zero:

$$\frac{dV}{dD(x)} = \frac{p_{data}(x)}{D(x)} - \frac{p_g(x)}{(1 - D(x))} = 0$$

Solving for D*(x):

$$D_G*(x) = \frac{p_{data}(x)}{(p_{data}(x) + p_g(x))}$$

Task 2: Training a GAN

- Use Adam optimizer with a learning rate of 0.0001 and β1 = 0.5.
- Fashion-MNIST has a standard resolution of 28 × 28 so make sure to resize it to 32 × 32
- You may have to adapt the DCGAN architecture slightly to generate 32×32 images instead of 64 × 64
- Using spectral normalization on the weights of the Discriminator can help with mode collapse and make training more stable
- You should already see decent results after a couple of epochs.

```
# Download Fashion MNIST dataset

# Define a transform to normalize the data
```

```
transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Resize((img_size,
img_size)),
                                transforms.Normalize((0.5,), (0.5,))])

# Download and load the training data
trainset = datasets.FashionMNIST(data_path, download=True, train=True,
transform=transform)
trainloader = DataLoader(trainset, batch_size=batch_size,
shuffle=True)

# Download and load the test data
testset = datasets.FashionMNIST(data_path, download=True, train=False,
transform=transform)
testloader = DataLoader(testset, batch_size=batch_size, shuffle=True)
```

```
Downloading http://fashion-mnist.s3-website.eu-central-
1.amazonaws.com/train-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-
1.amazonaws.com/train-images-idx3-ubyte.gz to
./data/FashionMNIST/raw/train-images-idx3-ubyte.gz

100%|██████████| 26421880/26421880 [00:02<00:00, 9566264.62it/s]

Extracting ./data/FashionMNIST/raw/train-images-idx3-ubyte.gz to
./data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-
1.amazonaws.com/train-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-
1.amazonaws.com/train-labels-idx1-ubyte.gz to
./data/FashionMNIST/raw/train-labels-idx1-ubyte.gz

100%|██████████| 29515/29515 [00:00<00:00, 146520.16it/s]

Extracting ./data/FashionMNIST/raw/train-labels-idx1-ubyte.gz to
./data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-
1.amazonaws.com/t10k-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-
1.amazonaws.com/t10k-images-idx3-ubyte.gz to
./data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz

100%|██████████| 4422102/4422102 [00:01<00:00, 2809806.17it/s]

Extracting ./data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz to
./data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-
1.amazonaws.com/t10k-labels-idx1-ubyte.gz
```

```
Downloading http://fashion-mnist.s3-website.eu-central-
1.amazonaws.com/t10k-labels-idx1-ubyte.gz to
./data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz

100%|██████████| 5148/5148 [00:00<00:00, 20350873.70it/s]

Extracting ./data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz to
./data/FashionMNIST/raw
```

```
# Get a batch of real images
real_batch = next(iter(trainloader))
images, _ = real_batch

# Normalize the images to [0, 1] range for plotting
images = (images + 1) / 2

# Create a grid of images
grid = torchvision.utils.make_grid(images[:6], padding=2,
normalize=False)

# Transpose the grid to match the expected shape for imshow
grid = np.transpose(grid.cpu().numpy(), (1, 2, 0))

# Plot the grid of images
plt.figure(figsize=(6, 4))
plt.axis("off")
plt.title("Training Images")
plt.imshow(grid)
plt.savefig(os.path.join(result_path, 'training_images.png'))
plt.show()
```


```
# Spectral Normalization - weight normalization
# https://arxiv.org/abs/1802.05957

class SpectralNormConv2d(nn.Conv2d):
    def __init__(self, *args, **kwargs):
        super(SpectralNormConv2d, self).__init__(*args, **kwargs)
        self.weight = spectral_norm(self.weight)

class SpectralNormLinear(nn.Linear):
```

```
    def __init__(self, *args, **kwargs):
        super(SpectralNormLinear, self).__init__(*args, **kwargs)
        self.weight = spectral_norm(self.weight)

class Generator(nn.Module):
    """ Generator generates fake images from random noise."""
    def __init__(self):
        super(Generator, self).__init__()
        # Input is 100, going into a convolution.
        self.conv1 = nn.ConvTranspose2d(100, 512, (4, 4), (1, 1), (0,
0), bias=False)
        self.bn1 = nn.BatchNorm2d(512)
        # state size. 512 x 4 x 4
        self.conv2 = nn.ConvTranspose2d(512, 256, (4, 4), (2, 2), (1,
1), bias=False)
        self.bn2 = nn.BatchNorm2d(256)
         # state size. 256 x 8 x 8
        self.conv3 = nn.ConvTranspose2d(256, 128, (4, 4), (2, 2), (1,
1), bias=False)
        self.bn3 = nn.BatchNorm2d(128)
        # state size. 128 x 16 x 16
        self.conv4 = nn.ConvTranspose2d(128, 64, (4, 4), (2, 2), (1,
1), bias=False)
        self.bn4 = nn.BatchNorm2d(64)
        # state size. 64 x 32 x 32
        self.conv5 = nn.ConvTranspose2d(64, 1, (4, 4), (1, 1), (0, 0),
bias=False)
        # state size. 1 x 32 x 32

    def forward(self, z):
        # Reshape the input
        z = z.view(-1, 100, 1, 1)
        x = F.relu(self.bn1(self.conv1(z)), inplace=True)  # 100 ->
512 x 4 x 4
        x = F.relu(self.bn2(self.conv2(x)), inplace=True)  # 512 x 4 x
4 -> 256 x 8 x 8
        x = F.relu(self.bn3(self.conv3(x)), inplace=True)  # 256 x 8 x
8 -> 128 x 16 x 16
        x = F.relu(self.bn4(self.conv4(x)), inplace=True)  # 128 x 16
x 16 -> 64 x 32 x 32
        # Tanh activation function
        x = torch.tanh(self.conv5(x))  # 64 x 32 x 32 -> 1 x 32 x 32
        return x

class Discriminator(nn.Module):
    """ Discriminator learns to distinguish between real and fake
images."""
    def __init__(self):
        super(Discriminator, self).__init__()
        # Input is 1 x 32 x 32
```

```
        self.conv1 = nn.utils.spectral_norm(nn.Conv2d(1, 64, (4, 4), 
(2, 2), (1, 1), bias=False))
        self.bn1 = nn.BatchNorm2d(64)
        self.dropout1 = nn.Dropout2d(p=0.2)
        self.conv2 = nn.utils.spectral_norm(nn.Conv2d(64, 128, (4, 4), 
(2, 2), (1, 1), bias=False))
        self.bn2 = nn.BatchNorm2d(128)
        self.dropout2 = nn.Dropout2d(p=0.2)
        self.conv3 = nn.utils.spectral_norm(nn.Conv2d(128, 256, (4, 
4), (2, 2), (1, 1), bias=False))
        self.bn3 = nn.BatchNorm2d(256)
        self.conv4 = nn.utils.spectral_norm(nn.Conv2d(256, 1, (4, 4), 
(1, 1), (0, 0), bias=False))

    def forward(self, x):
        x = F.leaky_relu(self.bn1(self.conv1(x)), 0.2, inplace=True)
        x = self.dropout1(x)
        x = F.leaky_relu(self.bn2(self.conv2(x)), 0.2, inplace=True)
        x = self.dropout2(x)
        x = F.leaky_relu(self.bn3(self.conv3(x)), 0.2, inplace=True)
        x = self.conv4(x)
        # Flatten the output
        x = x.view(-1, 1)
        # Sigmoid activation function
        x = F.sigmoid(x)
        return x

class DCGAN(nn.Module):
    """ DCGAN combines a generator and discriminator."""
    def __init__(self, latent_dim=100, img_size=32, lr=1e-4, 
betas=(0.5, 0.999), device=None):
        super(DCGAN, self).__init__()

        if device is None:
            device = torch.device("cuda" if torch.cuda.is_available() 
else "cpu")

        self._device = device
        self._lr = lr
        self._betas = betas
        self._latent_dim = latent_dim
        self._img_size = img_size
        self._fixed_noise = torch.randn(self._img_size, 
self._latent_dim, 1, 1, device=self._device)
        # Establish convention for real and fake labels during 
training
        self._real_label = 1
        self._fake_label = 0

        # Initialize generator and discriminator
```

```
        self.generator = Generator().to(self._device)
        self.discriminator = Discriminator().to(self._device)

        # Initialize weights
        self.generator.apply(self.weights_init)
        self.discriminator.apply(self.weights_init)

        # Initialize optimizers and criterion
        self.criteria = nn.BCELoss()
        self.binary_accuracy = nn.BCEWithLogitsLoss()
        self.optimizer_G = optim.Adam(self.generator.parameters(),
lr=self._lr, betas=self._betas)
        self.optimizer_D = optim.Adam(self.discriminator.parameters(),
lr=self._lr, betas=self._betas)

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

    def sample_random_z(self, n):
        sample = torch.randn(n, self._latent_dim, 1, 1,
device=self._device)
        return sample

    def sample_fixed_z(self):
        return self._fixed_noise

    def sample_G(self, n=1, type='fixed'):
        # Sample random noise z
        if type == 'fixed':
            z = self.sample_fixed_z()
        else:
            z = self.sample_random_z(n)
        # Generate fake images
        fake_img = self.generator(z)
        return fake_img

    def adversarial_loss(self, y_hat, y):
        # binary cross-entropy loss
        return self.criteria(y_hat, y)

    def generator_loss(self, fake):
        # Generator loss
        return self.adversarial_loss(self.discriminator(fake),
torch.ones(fake.size(0), 1, device=self._device))

    def discriminator_loss(self, real, fake):
        # Discriminator loss
        real_pred = self.discriminator(real)
        real_loss = self.adversarial_loss(real_pred,
torch.ones(real.size(0), 1, device=self._device))
```

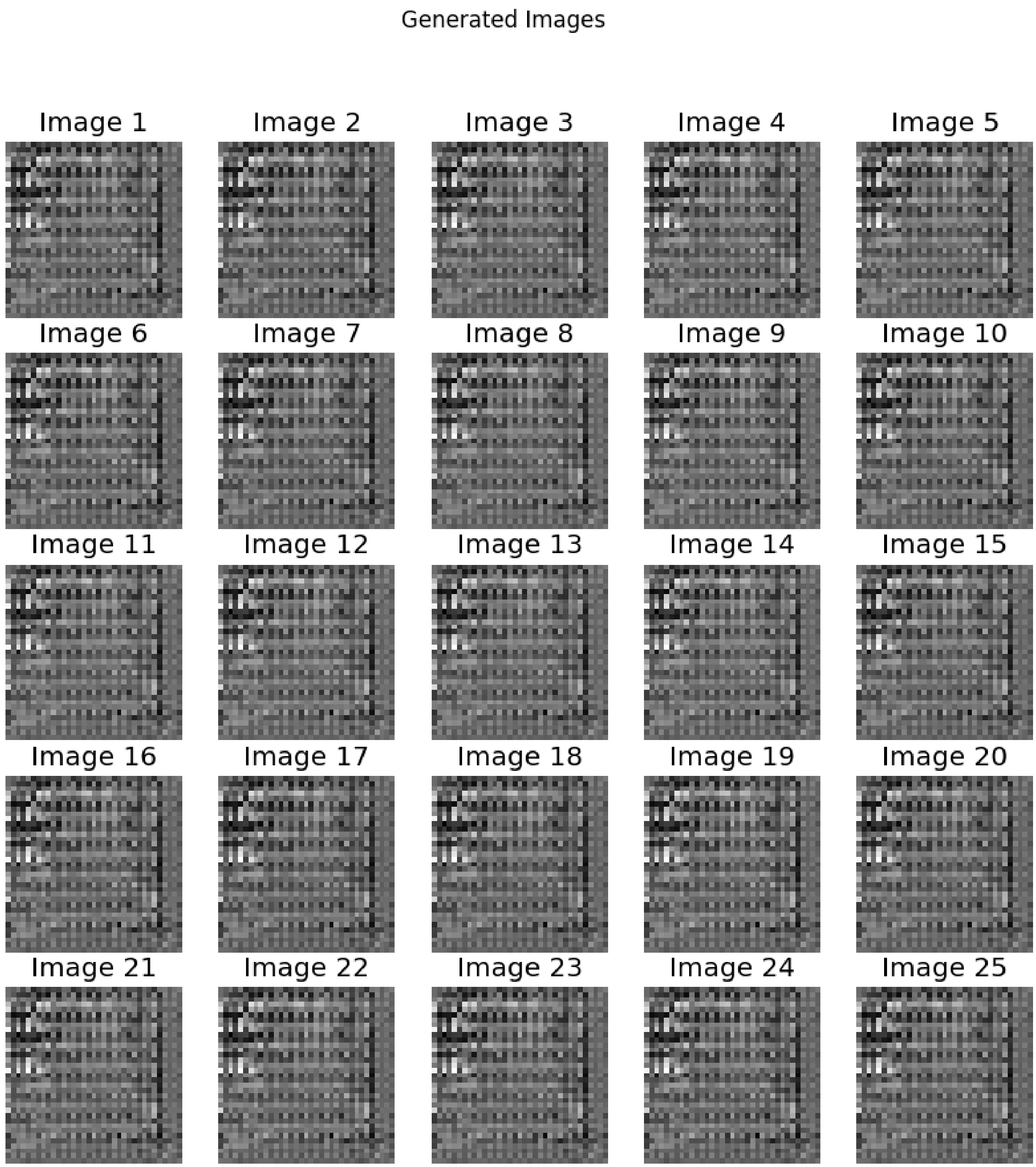
```
        fake_pred = self.discriminator(fake.detach())
        fake_loss = self.adversarial_loss(fake_pred,
torch.zeros(fake.size(0), 1, device=self._device))
        return real_loss + fake_loss

    def train_step(self, real_images):
        #
============================================================== #
        # ================== Train the discriminator
================== #
        #
============================================================== #
        # (1) Update D network: maximize log(D(x)) + log(1 - D(G(z)))

        # Train the discriminator with real images
        self.discriminator.zero_grad()
        label = torch.full((real_images.size(0), ), self._real_label,
dtype=torch.float32, device=self._device)

        # Forward pass
        real_pred = self.discriminator(real_images).view(-1)
        real_loss = self.adversarial_loss(real_pred, label)
        real_acc = self.binary_accuracy(real_pred, label)
        real_loss.backward()

        # Train the discriminator with fake images
        z = self.sample_random_z(real_images.size(0))
        fake_images = self.generator(z)
        label.fill_(self._fake_label)
        # Forward pass
        fake_pred = self.discriminator(fake_images.detach()).view(-1)
        fake_loss = self.adversarial_loss(fake_pred, label)
        fake_acc = self.binary_accuracy(fake_pred, label)
        fake_loss.backward()

        # Total Discriminator loss
        d_loss = real_loss + fake_loss

        # Update Discriminator weights
        self.optimizer_D.step()

        # Compute overall avg. accuracy
        acc = (real_acc + fake_acc) / 2

        #
============================================================== #
        # ===================== Train the generator
=================== #
        #
============================================================== #
```

```
        # (2) Update G network: maximize log(D(G(z)))

        # Train the generator with random noise z
        self.generator.zero_grad()
        label.fill_(self._real_label) # fake labels are real for
generator cost
        # Forward pass
        output = self.discriminator(fake_images).view(-1)

        # Generator loss
        g_loss = self.adversarial_loss(output, label)
        g_loss.backward()

        # Update Generator weights
        self.optimizer_G.step()

        return d_loss, g_loss, real_acc, fake_acc, acc

    # custom weights initialization called on ``netG`` and ``netD``
    def weights_init(self, m):
        classname = m.__class__.__name__
        if classname.find('Conv') != -1:
            nn.init.normal_(m.weight.data, 0.0, 0.02)
        elif classname.find('BatchNorm') != -1:
            nn.init.normal_(m.weight.data, 1.0, 0.02)
            nn.init.constant_(m.bias.data, 0)

    def save(self, path):
        torch.save(self.state_dict(), path)

    def load(self, path):
        self.load_state_dict(torch.load(path))

    def plot(self, n=25, save=False, path='results/result.png'):
        with torch.no_grad():
            z = self.sample_random_z(n)
            fake = self.generator(z).cpu()
            fake = fake.numpy()
            fig, axes = plt.subplots(5, 5, figsize=(10, 10))
            fig.suptitle('Generated Images')
            for i, ax in enumerate(axes.flat):
                ax.imshow(fake[i, 0], cmap='gray')  # Reshape to (32,
32)

                ax.axis('off')
                ax.title.set_text(f'Image {i+1}')
            if save:
                plt.savefig(path)
            plt.show()
```

```
# Initialize DCGAN
dcgan = DCGAN(latent_dim=latent_dim, img_size=img_size, lr=lr,
betas=betas, device=device)

d_fake_img_acc = []
d_real_img_acc = []
d_training_losses = []
g_training_losses = []

fake_img_list = []

# Train DCGAN
for epoch in range(num_epochs):
    for i, data in enumerate(trainloader):
        real_images = data[0].to(device)

        # Train the DCGAN
        d_loss, g_loss, real_acc, fake_acc, acc =
dcgan.train_step(real_images)

        if (i+1) % 100 == 0:
            print('Epoch [{}/{}], Step [{}/{}], d_loss: {:.4f},
g_loss: {:.4f}'
                  .format(epoch, num_epochs, i+1, len(trainloader),
d_loss.item(), g_loss.item()))

        # Save Losses for plotting later
        d_training_losses.append(d_loss.item())
        g_training_losses.append(g_loss.item())

        # Save Discriminator Accuracy for plotting later
        d_fake_img_acc.append(fake_acc.item())
        d_real_img_acc.append(real_acc.item())

        # Collect G's output on a fixed_noise sample
        if (i+1) % (len(trainloader) // 200) == 0 or ((epoch ==
num_epochs-1) and (i == len(trainloader)-1)):
            with torch.no_grad():
                fake = dcgan.sample_G(type='fixed').detach().cpu()
            fake_img_list.append(fake)

# Plot some images generated by the DCGAN
dcgan.plot(n=25, save=True, path=result_path +
'/dcgan_train_results.png')

# Save the model under model_path
dcgan.save(model_path + '/dcgan.pth')
```

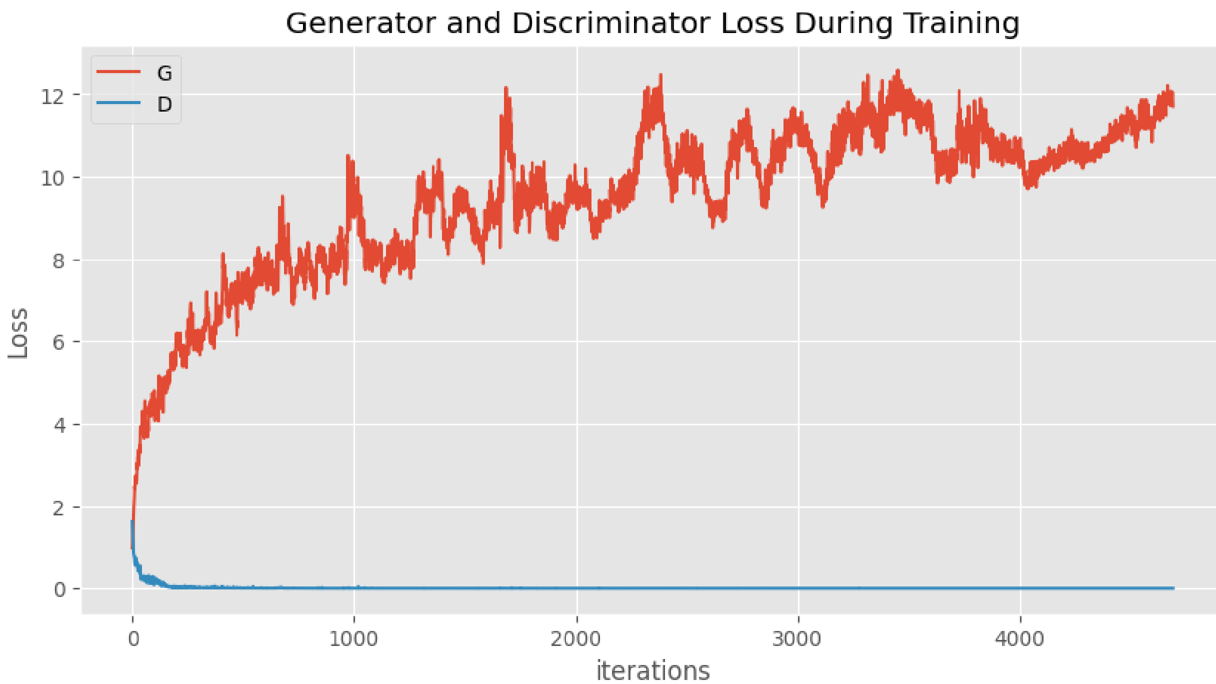
```
Epoch [0/10], Step [100/469], d_loss: 0.1283, g_loss: 4.4091
Epoch [0/10], Step [200/469], d_loss: 0.0306, g_loss: 6.0115
```

```
Epoch [0/10], Step [300/469], d_loss: 0.0164, g_loss: 6.1200
Epoch [0/10], Step [400/469], d_loss: 0.0167, g_loss: 6.8290
Epoch [1/10], Step [100/469], d_loss: 0.0028, g_loss: 8.0765
Epoch [1/10], Step [200/469], d_loss: 0.0139, g_loss: 8.0366
Epoch [1/10], Step [300/469], d_loss: 0.0022, g_loss: 7.6263
Epoch [1/10], Step [400/469], d_loss: 0.0015, g_loss: 8.1086
Epoch [2/10], Step [100/469], d_loss: 0.0015, g_loss: 8.4755
Epoch [2/10], Step [200/469], d_loss: 0.0021, g_loss: 7.5670
Epoch [2/10], Step [300/469], d_loss: 0.0009, g_loss: 8.1408
Epoch [2/10], Step [400/469], d_loss: 0.0010, g_loss: 9.2266
Epoch [3/10], Step [100/469], d_loss: 0.0004, g_loss: 9.4784
Epoch [3/10], Step [200/469], d_loss: 0.0014, g_loss: 8.6925
Epoch [3/10], Step [300/469], d_loss: 0.0020, g_loss: 10.9239
Epoch [3/10], Step [400/469], d_loss: 0.0010, g_loss: 9.2538
Epoch [4/10], Step [100/469], d_loss: 0.0003, g_loss: 9.4819
Epoch [4/10], Step [200/469], d_loss: 0.0004, g_loss: 8.6064
Epoch [4/10], Step [300/469], d_loss: 0.0002, g_loss: 9.6201
Epoch [4/10], Step [400/469], d_loss: 0.0010, g_loss: 10.4929
Epoch [5/10], Step [100/469], d_loss: 0.0005, g_loss: 10.1548
Epoch [5/10], Step [200/469], d_loss: 0.0050, g_loss: 10.0846
Epoch [5/10], Step [300/469], d_loss: 0.0004, g_loss: 9.2235
Epoch [5/10], Step [400/469], d_loss: 0.0002, g_loss: 11.2378
Epoch [6/10], Step [100/469], d_loss: 0.0001, g_loss: 10.9267
Epoch [6/10], Step [200/469], d_loss: 0.0001, g_loss: 10.7535
Epoch [6/10], Step [300/469], d_loss: 0.0004, g_loss: 9.5545
Epoch [6/10], Step [400/469], d_loss: 0.0002, g_loss: 11.1540
Epoch [7/10], Step [100/469], d_loss: 0.0001, g_loss: 11.1895
Epoch [7/10], Step [200/469], d_loss: 0.0000, g_loss: 12.0543
Epoch [7/10], Step [300/469], d_loss: 0.0000, g_loss: 11.6614
Epoch [7/10], Step [400/469], d_loss: 0.0001, g_loss: 9.8580
Epoch [8/10], Step [100/469], d_loss: 0.0001, g_loss: 10.9126
Epoch [8/10], Step [200/469], d_loss: 0.0003, g_loss: 10.8696
Epoch [8/10], Step [300/469], d_loss: 0.0001, g_loss: 9.9025
Epoch [8/10], Step [400/469], d_loss: 0.0000, g_loss: 10.5910
Epoch [9/10], Step [100/469], d_loss: 0.0001, g_loss: 10.5632
Epoch [9/10], Step [200/469], d_loss: 0.0000, g_loss: 10.9473
Epoch [9/10], Step [300/469], d_loss: 0.0001, g_loss: 11.3214
Epoch [9/10], Step [400/469], d_loss: 0.0001, g_loss: 11.5810
```

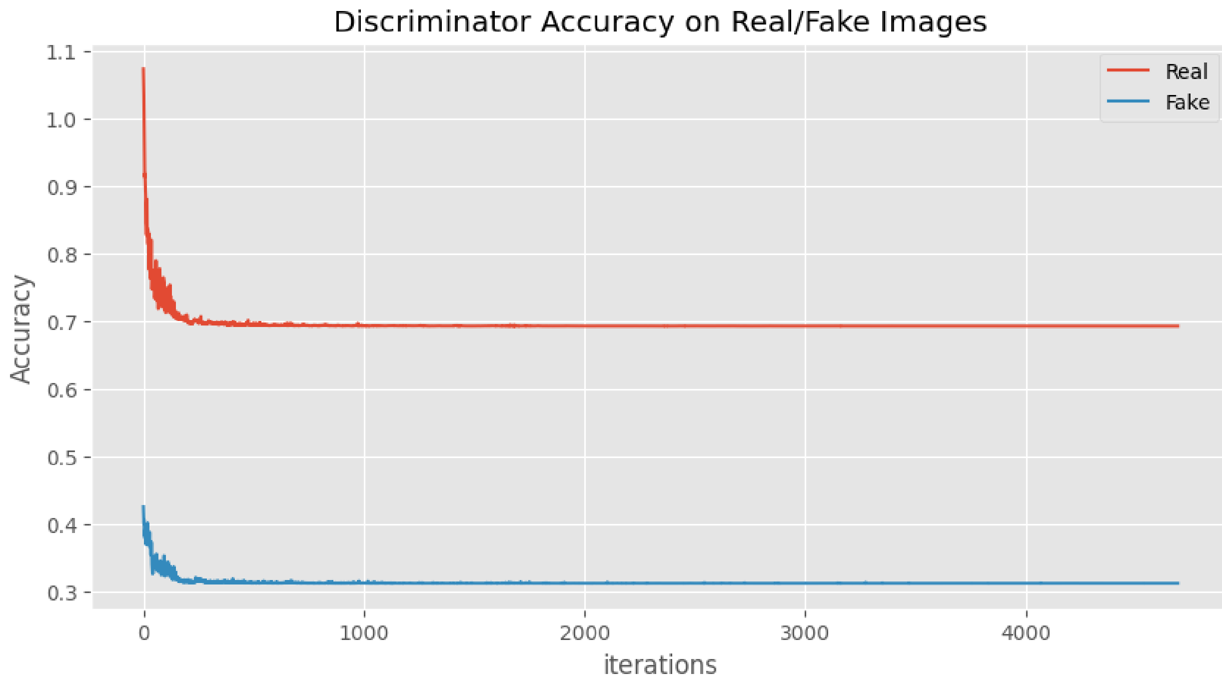



```
# Plot evolution of the training losses for the generator and
discriminator
plt.figure(figsize=(10, 5))
plt.title("Generator and Discriminator Loss During Training")
plt.plot(g_training_losses, label="G")
plt.plot(d_training_losses, label="D")
plt.xlabel("iterations")
plt.ylabel("Loss")
```

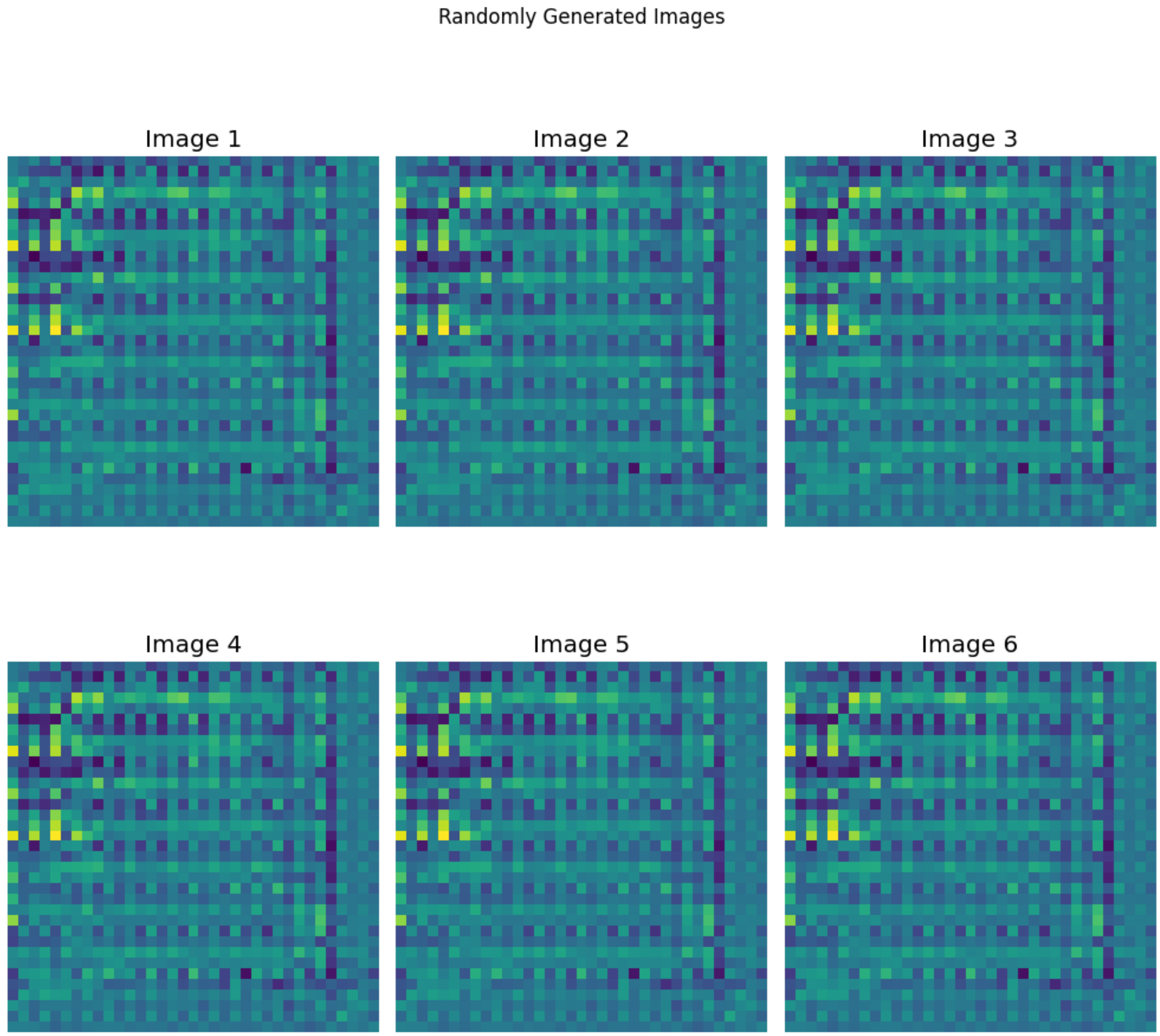
```
plt.legend()
plt.savefig(result_path + '/loss.png')
plt.show()
```



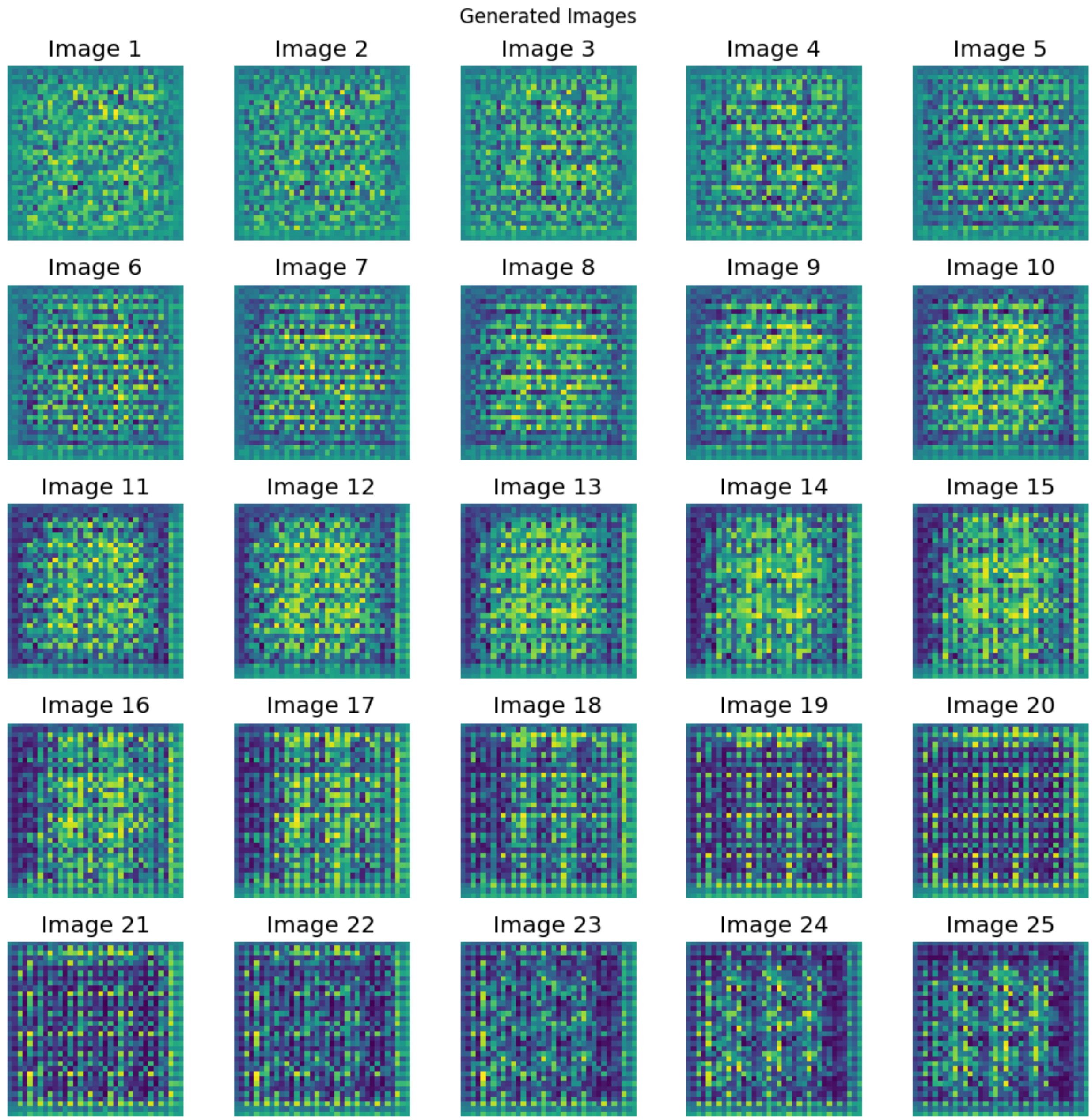

```
# Plot classification accuracy of the discriminator on real/fake
samples
plt.figure(figsize=(10, 5))
plt.title("Discriminator Accuracy on Real/Fake Images")
plt.plot(d_fake_img_acc, label="Real")
plt.plot(d_real_img_acc, label="Fake")
plt.xlabel("iterations")
plt.ylabel("Accuracy")
plt.legend()

plt.savefig(result_path + '/accuracy.png')
plt.show()
```




```
# Plot 6 randomly synthesized images from the generator in a grid
fig, axs = plt.subplots(2, 3, figsize=(10, 10))
fig.suptitle("Randomly Generated Images")
for i, ax in enumerate(axs.flat):
    ax.imshow(np.transpose(fake_img_list[-1][i], (1, 2, 0)))
    ax.axis('off')
    ax.title.set_text(f"Image {i+1}")
plt.tight_layout()
plt.savefig(result_path + '/generated_images.png')
plt.show()
```



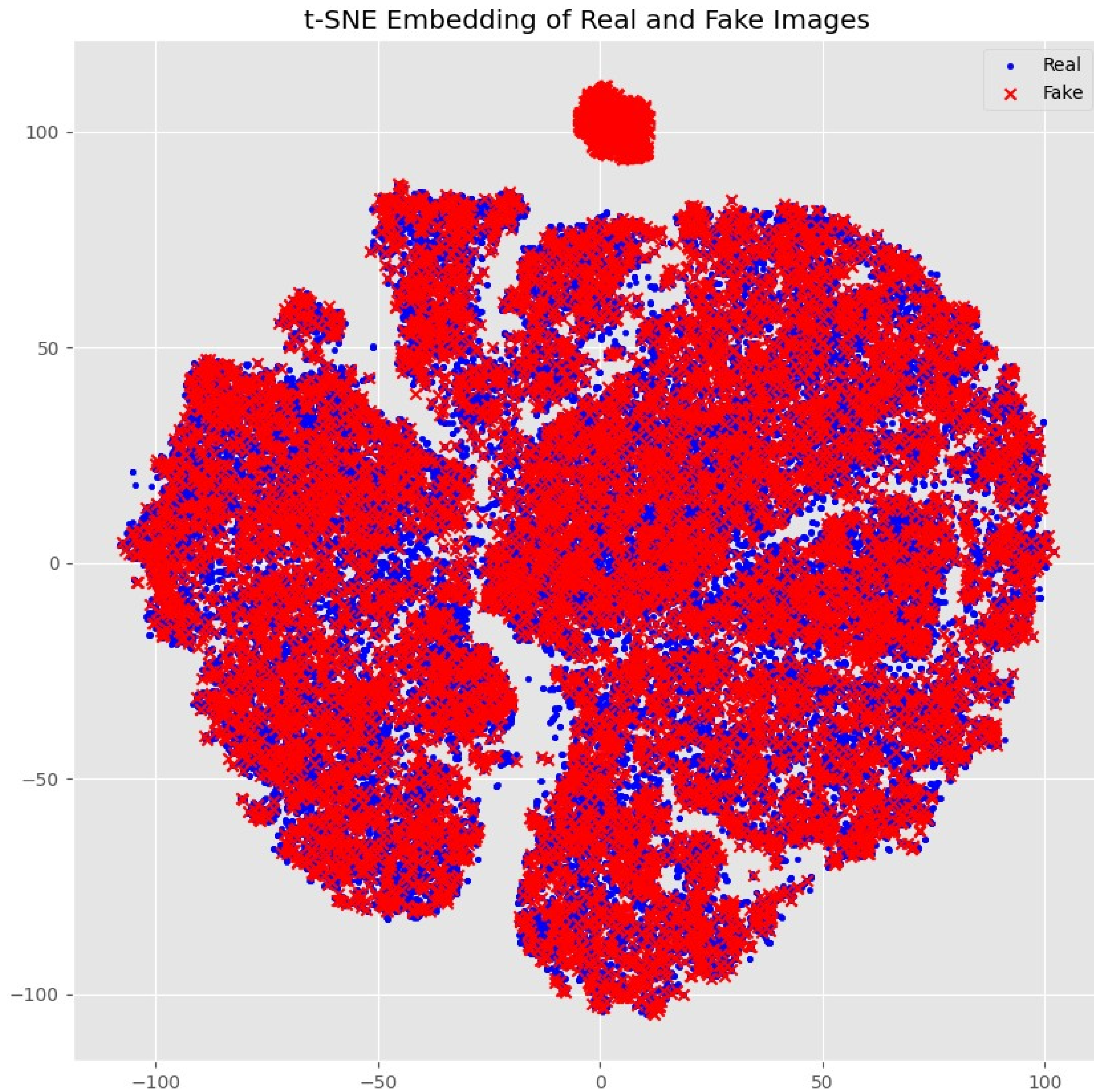

```python
# Plot collected images for evolution of the generator during training
fig, axs = plt.subplots(5, 5, figsize=(10, 10))
fig.suptitle("Generated Images")
for i, ax in enumerate(axs.flat):
    ax.imshow(np.transpose(fake_img_list[i][0], (1, 2, 0)))
    ax.axis('off')
    ax.title.set_text(f"Image {i+1}")
plt.tight_layout()
plt.savefig(result_path + '/generated_images_evolution.png')
plt.show()
```




```
from tqdm import tqdm

# Load the pretrained model and remove the last layer
resnet18 = torchvision.models.resnet18(pretrained=True)
# Update the first layer to accept grayscale images
resnet18.conv1 = nn.Conv2d(1, 64, kernel_size=7, stride=2, padding=3,
bias=False)
resnet18 = nn.Sequential(*list(resnet18.children())[:-1]).to(device)
resnet18.eval()

# Extract features from the last convolutional layer of the model
```

```
def get_features(model, dataloader):
    features = []
    labels = []
    with torch.no_grad():
        for images, labels_ in tqdm(dataloader):
            images = images.to(device)
            output = model(images)
            output = output.view(output.size(0), -1)  # Flatten the
output
            features.append(output.detach().cpu())
            labels.append(labels_)
    return torch.cat(features), torch.cat(labels)

# Extract features from the train and test splits
train_features, train_labels = get_features(resnet18, trainloader)
test_features, test_labels = get_features(resnet18, testloader)

# Generate images using the generator and extract features
n = 1000
batch_size = 16
fake_features = []

for i in range(0, n, batch_size):
    z = dcgan.sample_random_z(batch_size)
    fake_images = dcgan.generator(z)
    with torch.no_grad():
        batch_features = resnet18(fake_images).view(batch_size, -
1).detach().cpu()
        fake_features.append(batch_features)

fake_features = torch.cat(fake_features)

# Concatenate the features
features = torch.cat([train_features, test_features, fake_features])
labels = torch.cat([train_labels, test_labels, torch.ones(n) * 10])

100%|██████████| 469/469 [00:27<00:00, 17.37it/s]
100%|██████████| 79/79 [00:04<00:00, 18.92it/s]
/usr/local/lib/python3.10/dist-packages/torch/nn/modules/conv.py:952:
UserWarning: Plan failed with a cudnnException:
CUDNN_BACKEND_EXECUTION_PLAN_DESCRIPTOR: cudnnFinalize Descriptor
Failed cudnn_status: CUDNN_STATUS_NOT_SUPPORTED (Triggered internally
at ../aten/src/ATen/native/cudnn/Conv_v8.cpp:919.)
  return F.conv_transpose2d(

# Perform t-SNE embedding
tsne = TSNE(n_components=2, random_state=42)
X_embedded = tsne.fit_transform(features)
```

```
# Plot the t-SNE embedding
plt.figure(figsize=(10, 10))
plt.title("t-SNE Embedding of Real and Fake Images")

# Plot real images as blue dots
plt.scatter(X_embedded[:train_features.shape[0], 0],
X_embedded[:train_features.shape[0], 1], c='blue', marker='.',
label='Real')
# Plot fake images as red x's
plt.scatter(X_embedded[train_features.shape[0]:, 0],
X_embedded[train_features.shape[0]:, 1], c='red', marker='x',
label='Fake')
plt.legend()
plt.savefig(result_path + '/tsne.png')
plt.show()
```




```python
# Plot the classification accuracy on generated data using the
previous ResNet18 classifier
acc = []
n = 1000

# Concatenate the tensors in fake_img_list
fake_imgs = torch.cat(fake_img_list)

# Evaluate the classifier on the generated images
for i in range(0, n, batch_size):
    with torch.no_grad():
        output = resnet18(fake_imgs[i:i+batch_size].to(device))
```

```python
        output = output.view(output.size(0), -1)
        pred = torch.argmax(output, 1)
        acc.append((pred == 10).sum().item() / batch_size)

# Plot the classification accuracy
plt.figure(figsize=(10, 5))
plt.title("Classification Accuracy on Generated Images")
plt.plot(acc)
plt.xlabel("iterations")
plt.ylabel("Accuracy")
plt.savefig(result_path + '/classification_accuracy.png')
plt.show()
```