Lab 9

Generative Adversarial Network

```python
import torch
from torch import nn
from tqdm import tqdm
from PIL import Image
import torchvision
from torchvision import datasets, transforms
from torchvision.datasets import ImageFolder
from torch.utils.data import DataLoader
import torch.optim as optim
from torchvision.utils import make_grid
from torch.utils.data import DataLoader
from torch.utils.tensorboard import SummaryWriter
import matplotlib.pyplot as plt
import numpy as np
import os
import random
```

ที่จำเป็นต้องใช้ใน lab นี้

```python
import tarfile

file_name = 'data.tar.gz'

# Open the tar.gz file
with tarfile.open(file_name, 'r:gz') as tar:
    # Extract all contents into the current directory
    tar.extractall()

print("Extraction complete!")
```

[2] โค้ดนี้ทำหน้าที่แยกไฟล์: data.tar.gz ที่ถูกบีบอัดอยู่ในรูปแบบ tar.gz ออกมาในที่ที่เรากำหนดไว้ในปัจจุบัน

```python
### START CODE HERE ###
class Generator(nn.Module):
    def __init__(self, z_dim=256, im_ch=3, hidden_dim=1024):
        super(Generator, self).__init__()

        self.z_dim = z_dim
        self.gen = nn.Sequential(
            self.conv_block(z_dim, hidden_dim, kernel_size=4, stride=1),
            self.conv_block(hidden_dim, hidden_dim // 2, kernel_size=4, stride=2, padding=1),
            self.conv_block(hidden_dim // 2, hidden_dim // 4, kernel_size=4, stride=2, padding=1),
            self.conv_block(hidden_dim // 4, hidden_dim // 8, kernel_size=4, stride=2, padding=1),
            self.conv_block(hidden_dim // 8, hidden_dim // 16, kernel_size=4, stride=2, padding=1),
            self.conv_block(hidden_dim // 16, im_ch, kernel_size=4, stride=2, padding=1, final_layer=True),
        )

    def conv_block(self, in_ch, out_ch, kernel_size=4, stride=1, padding=0, final_layer=False):
        if not final_layer:
            return nn.Sequential(
                nn.ConvTranspose2d(in_channels=in_ch, out_channels=out_ch, kernel_size=kernel_size, stride=stride, padding=padding),
                nn.BatchNorm2d(out_ch),
                nn.ReLU()
            )
        else:
            return nn.Sequential(
                nn.ConvTranspose2d(in_channels=in_ch, out_channels=out_ch, kernel_size=kernel_size, stride=stride, padding=padding),
                nn.Tanh()  # For generating images, often use Tanh activation
            )

    def unsqueeze_noise(self, noise):
        return noise.view(len(noise), self.z_dim, 1, 1)

    def forward(self, noise):
        x = self.unsqueeze_noise(noise)
        return self.gen(x)
### END CODE HERE ###
```

Generator ซึ่งเป็นส่วนหนึ่งของโครงข่ายประสาทเทียมแบบ Generative Adversarial Network (GAN) ซึ่ง class นี้มีหน้าที่สร้างภาพใหม่จาก noise vector

- The Generator class defines a neural network that upsamples a latent noise vector (of size z_dim) through several transposed convolution layers to generate an image with im_ch channels. The network architecture progressively refines the noise into an image using ReLU activations and Batch Normalization in intermediary layers and a Tanh activation in the final output layer for image scaling.

Initialization (__init__):

- **Parameters**:
  - z_dim=256: The dimension of the input noise vector (latent space).
  - im_ch=3: The number of output image channels (e.g., 3 for RGB).
  - hidden_dim=1024: The dimension for the first hidden layer, which decreases by half at each subsequent layer.
- **Architecture**:
  - Uses a sequence of transposed convolutional layers (conv_block) to progressively upsample the noise vector.
  - The conv_block method is called multiple times with decreasing channel sizes, indicating a step-by-step refinement of the image from the noise.

Method: conv_block:

- สร้างโครงข่ายประสาทเทียมโดยใช้ nn.Sequential ซึ่งประกอบด้วยหลาย layer
- layer แต่ละ layer เป็น conv_block ซึ่งเป็นฟังก์ชันที่สร้างขึ้นเพื่อให้โครงข่ายสามารถเพิ่มขนาดของภาพได้ (upsampling)

- **Purpose**: Defines a block of layers consisting of:
  - **Non-final layers**:
    - **nn.ConvTranspose2d:** Performs upsampling with a transposed convolution (similar to "deconvolution").
    - **nn.BatchNorm2d:** Normalizes the output of the transposed convolution for stable training.
    - **nn.ReLU():** Applies a ReLU activation to introduce non-linearity.
  - **Final layer**:

- If it's the final layer, the block applies **nn.ConvTranspose2d** followed by **nn.Tanh()** activation, which ensures the output pixel values are in the range of **[-1, 1],** common for image generation tasks.

Method: unsqueeze_noise:

- **Purpose**: Reshapes (unsqueezes) the 1D noise vector into a 4D tensor of shape (batch_size, z_dim, 1, 1) to prepare it for convolutional operations. The extra dimensions are needed for the transposed convolutions.

Method: forward:

- **Purpose**: Defines the forward pass of the network.
  - รับ noise vector เป็นอินพุต
  - เปลี่ยนรูปร่างของ noise vector ให้เหมาะสม
  - ส่ง noise vector ผ่านโครงข่ายประสาทเทียมเพื่อสร้างภาพ
    - The noise input is unsqueezed and passed through the generator's sequential layers (self.gen) to produce an image-like output.

```
### START CODE HERE ###
class Discriminator(nn.Module):
    def __init__(self, im_ch=3, hidden_dim=32):
        super(Discriminator, self).__init__()

        self.disc = nn.Sequential(
            self.conv_block(im_ch, hidden_dim, kernel_size=4, stride=2, padding=1),
            self.conv_block(hidden_dim, hidden_dim * 2, kernel_size=4, stride=2, padding=1),
            self.conv_block(hidden_dim * 2, hidden_dim * 4, kernel_size=4, stride=2, padding=1),
            self.conv_block(hidden_dim * 4, hidden_dim * 8, kernel_size=4, stride=2, padding=1),
            self.conv_block(hidden_dim * 8, 1, final_layer=True),
        )

    def conv_block(self, in_ch, out_ch, kernel_size=4, stride=1, padding=0, final_layer=False):
        if not final_layer:
            return nn.Sequential(
                nn.Conv2d(in_channels=in_ch, out_channels=out_ch, kernel_size=kernel_size, stride=stride, padding=padding),
                nn.BatchNorm2d(out_ch),
                nn.LeakyReLU(negative_slope=0.2)
            )
        else:
            return nn.Sequential(
                nn.Conv2d(in_channels=in_ch, out_channels=out_ch, kernel_size=kernel_size, stride=stride, padding=padding),
                nn.Sigmoid()
            )

    def forward(self, image):
        disc_pred = self.disc(image)
        return disc_pred.view(len(disc_pred), -1)
### END CODE HERE ###
```

Discriminator ซึ่งเป็นอีกส่วนหนึ่งของโครงข่ายประสาทเทียมแบบ Generative Adversarial Network (GAN) ซึ่งมีหน้าที่ตัดสินว่าภาพที่ได้รับเป็นภาพจริงหรือภาพปลอม

- The Discriminator class defines a neural network that processes an image through a series of convolutional layers. The image is downsampled, and feature maps are extracted through convolutions with LeakyReLU activations and Batch Normalization. The final output is a single probability score produced by a Sigmoid activation, which classifies the image as either real or fake. The network progressively increases the number of filters while reducing the spatial dimensions to condense the image information for the final classification.

Initialization (__init__):

- **Parameters**:
  - im_ch=3: Number of channels in the input image (e.g., 3 for RGB).
  - hidden_dim=32: The base dimension for the hidden layers, which increases by a factor of two in subsequent layers.
- **Architecture**:
  - The nn.Sequential block consists of multiple convolutional layers (conv_block), where each one progressively reduces the spatial dimensions of the input and increases the number of feature maps.
  - The final layer maps the features to a single value (the prediction of real vs. fake) using a Sigmoid activation function, which outputs a probability score.

Method: conv_block:

- **Purpose**: Defines a block of layers consisting of:
  - **Non-final layers**:
    - **nn.Conv2d:** Standard convolutional operation to extract features from the input. (Layer: Convolutional สำหรับ downsampling)
    - **nn.BatchNorm2d:** Normalizes the output of the convolution for stable training.
    - **nn.LeakyReLU():** Applies a LeakyReLU activation function with a small negative slope (0.2) to introduce non-linearity and prevent dying ReLUs.
  - **Final layer**:
    - If it's the final layer, the block applies **nn.Conv2d** followed by **nn.Sigmoid()**. The Sigmoid activation compresses the output to a probability value **between 0 and 1,**

indicating how likely the image is real. (ซึ่งจะใช้สำหรับการจำแนกว่าเป็นภาพจริงหรือภาพปลอม)

Method: forward:

- **Purpose**: Defines the forward pass of the network.
  - รับภาพเป็นอินพุต
  - ส่งภาพผ่านโครงข่ายประสาทเทียม
  - เปลี่ยนรูปร่างของผลลัพธ์ให้เป็น 1D เพื่อให้สามารถใช้ในการคำนวณ loss function
    - The input image is passed through the sequential convolutional blocks (self.disc).
    - After processing, the output is flattened into a 2D tensor using .view(len(disc_pred), -1), where each entry corresponds to the discriminator's prediction for each image in the batch.

```
### START CODE HERE ###
def get_noise(n_sample, z_dim, device='cuda'):
    # Check if CUDA is available, otherwise use CPU
    device = 'cuda' if torch.cuda.is_available() else 'cpu'
    # Generate random noise tensor from a normal distribution
    noise = torch.randn(n_sample, z_dim, device=device)
    return noise
### END CODE HERE ###
```

get_noise ซึ่งมีหน้าที่สร้าง noise vector สำหรับใช้ในการสร้างภาพใหม่ในโครงข่ายประสาทเทียมแบบ Generative Adversarial Network (GAN)

- **สร้างเวกเตอร์สุ่ม**
  - ใช้ **torch.randn(n_sample, z_dim, device=device)** เพื่อสร้างเทนเซอร์ของเวกเตอร์สุ่มจากการแจกแจงแบบปกติ (normal distribution):
    - **n_sample:** คือจำนวนแถวในเทนเซอร์
    - **z_dim:** คือจำนวนคอลัมน์ในแต่ละแถว
    - **device:** ระบุว่าเทนเซอร์นี้จะอยู่ที่ไหน (CPU หรือ GPU)
- The get_noise function generates a batch of random noise vectors, which will be fed into the generator as input for generating images. The noise is sampled from a normal distribution, and the function ensures that it is created on the appropriate device (GPU if available, otherwise CPU). The noise vectors introduce variation, allowing the GAN to generate diverse outputs.

```python
n_sample = 25
z_dim = 100
noise = get_noise(n_sample, z_dim)
assert noise.shape == (n_sample, z_dim), f"Expected shape {(n_sample, z_dim)}, but got {noise.shape}"

noise_cpu = get_noise(n_sample, z_dim, device='cpu')
assert noise_cpu.device.type == 'cpu', f"Expected tensor to be on 'cpu', but got {noise_cpu.device.type}"
assert noise.dtype == torch.float32, f"Expected dtype to be torch.float32, but got {noise.dtype}"
```

เพื่อตรวจสอบให้แน่ใจว่า noise vector ถูกสร้างขึ้นอย่างถูกต้อง

```python
# Create a simple DataLoader from the image paths
class ImageDataset(torch.utils.data.Dataset):
    def __init__(self, image_paths, transform=None):
        self.image_paths = image_paths
        self.transform = transform

    def __len__(self):
        return len(self.image_paths)

    def __getitem__(self, idx):
        img_path = self.image_paths[idx]
        image = Image.open(img_path).convert('RGB')

        if self.transform:
            image = self.transform(image)

        return image
```

[7] สร้าง class: ImageDataset ซึ่งเป็น class ย่อยของ torch.utils.data.Dataset ซึ่งมีหน้าที่โหลดรูปภาพจาก image paths

- The ImageDataset class provides a way to load images from file paths and apply transformations, enabling easy integration with PyTorch's DataLoader. It uses the file paths to open and convert each image to RGB format, and optional transformations are applied for preprocessing. The class also implements methods to retrieve the total number of images (__len__) and access a specific image by index (__getitem__).

```python
# Define transformations
transform = transforms.Compose([
    transforms.Resize((128, 128)),  # Resize images to 128x128
    transforms.ToTensor(),          # Convert images to tensors
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))  # Normalize to [-1, 1]
])

# Load dataset
relative_dir = "data/img_align_celeba"
data_dir = os.path.abspath(relative_dir)
# print(data_dir)
image_paths = [os.path.join(data_dir, img) for img in os.listdir(data_dir) if img.endswith(".jpg")]

#---------------------
# sample_image_paths = random.sample(image_paths, 10)
# # สร้าง dataset และ dataloader
# dataset = ImageDataset(sample_image_paths, transform=transform)
# batch_size = 2
# dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)
#---------------------

# Instantiate the dataset
dataset = ImageDataset(image_paths=image_paths, transform=transform)

# Create DataLoader
batch_size = 64
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)

# Define model dimensions
z_dim = 256
im_ch = 3
hidden_dim = 1024

# Instantiate models
generator = Generator(z_dim=z_dim, im_ch=im_ch, hidden_dim=hidden_dim)
discriminator = Discriminator(im_ch=im_ch, hidden_dim=32)

# Check device availability
device = 'cuda' if torch.cuda.is_available() else 'cpu'
generator.to(device)
discriminator.to(device)
### END CODE HERE ###
```

- กำหนดการแปลงรูปภาพ (Transformations)
    - สร้าง function: transform โดยใช้ transforms.Compose เพื่อแปลงรูปภาพก่อนนำไปใช้กับโมเดล
    - ฟังก์ชัน transform ประกอบด้วยการแปลงหลายขั้นตอน:
        - **transforms.Resize((128, 128)):** ปรับขนาดรูปภาพให้เป็น 128x128 พิกเซล
        - **transforms.ToTensor():** แปลงรูปภาพเป็น Tensor (ข้อมูลตัวเลข)
        - **transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)):** Normalize ค่าสีของแต่ละช่อง (แดง, เขียว, น้ำเงิน) ให้มีค่าอยู่ในช่วง [-1, 1] ซึ่งเหมาะสำหรับการนำไปใช้กับโมเดลประสาทเทียม
- โหลดชุดข้อมูล (Dataset)
    - กำหนด path ของชุดข้อมูล (data_dir) โดยอ้างอิงจากโฟลเดอร์ data/img_align_celeba

- สร้าง list เก็บรูปภาพใน image_paths โดยวนลูปผ่านไฟล์ทั้งหมดในโฟลเดอร์ data_dir และเก็บเฉพาะไฟล์ที่ลงท้ายด้วย ".jpg"

- สร้าง instance ของ ImageDataset
  - สร้าง instance ของคลาส ImageDataset โดยส่งรายการเส้นทางของรูปภาพ (image_paths) และฟังก์ชันการแปลง (transform) ไป
- สร้าง DataLoader
  - สร้าง DataLoader โดยใช้ชุดข้อมูล (dataset) ที่สร้างขึ้น
  - กำหนดขนาดของ batch (batch_size) เป็น 64 ภาพ
  - กำหนดให้สุ่มรูปภาพในแต่ละ batch (shuffle=True)
- กำหนดขนาดของโมเดล
  - Code นี้กำหนดค่าพารามิเตอร์สำหรับโมเดลดังนี้:
    - **z_dim:** ขนาดของ noise vector (256)
    - **im_ch:** จำนวนช่องสีของรูปภาพ (3 ช่องสำหรับ RGB)
    - **hidden_dim:** ขนาดของ layer ซ่อนในโครงข่ายประสาทเทียม (1024)
- สร้างอินสแตนซ์ของโมเดล
  - สร้าง instamce ของคลาส Generator และ Discriminator โดยกำหนดค่าพารามิเตอร์ที่กำหนดไว้ก่อนหน้านี้
- ตรวจสอบการใช้งาน GPU
  - ตรวจสอบว่าสามารถใช้การ์ดประมวลผลกราฟิกส์ (GPU) ได้หรือไม่ และกำหนดอุปกรณ์ที่ใช้สำหรับการคำนวณ (device)
- ย้ายโมเดลไปยังอุปกรณ์ที่เลือก
  - ใช้ generator.to(device) และ discriminator.to(device) เพื่อย้ายโมเดลไปยังอุปกรณ์ที่เลือก

```python
def display_images(images, n_cols=4, n_rows=4):
    """
    Display a grid of images.

    Parameters:
    - images: NumPy array of shape (N, C, H, W) where N is the number of images,
      C is the number of channels, H is height, and W is width.
    - n_cols: Number of columns in the grid.
    - n_rows: Number of rows in the grid.
    """
    # Ensure images are in the shape (N, H, W, C)
    if images.ndim == 4 and images.shape[1] in [1, 3]:  # (N, C, H, W)
        images = np.transpose(images, (0, 2, 3, 1))  # Change to (N, H, W, C)


    plt.figure(figsize=(n_cols * 2, n_rows * 2))  # Adjust size based on grid dimensions
    for i in range(min(n_cols * n_rows, len(images))):
        ax = plt.subplot(n_rows, n_cols, i + 1)
        ax.imshow(images[i])
        ax.axis('off')  # Turn off axis
    plt.tight_layout()
    plt.show()
```

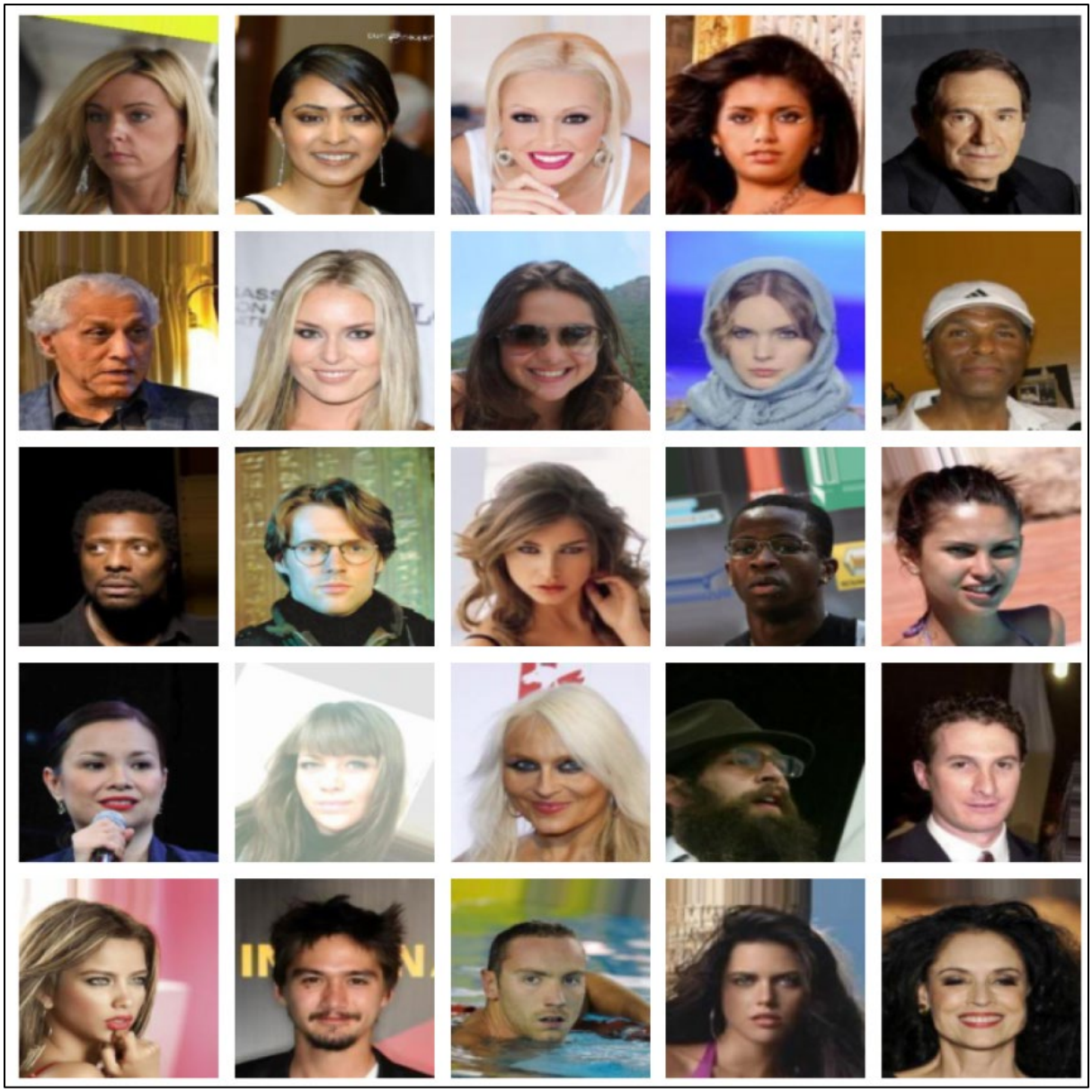
```
# Display first batch of dataset
data_iter = iter(dataloader)
images = next(data_iter)

# Convert tensor to numpy for plotting
images = images * 0.5 + 0.5  # Denormalize images to [0, 1]
images = images.numpy()

display_images(images, n_cols=5, n_rows=5)
### END CODE HERE ###
```

[10] This code fetches the first batch of images from the DataLoader, denormalizes the images (scaling them back to the [0, 1] range), and converts them into a format that can be displayed using matplotlib. It then uses the display_images function to plot a grid of 5x5 images from the dataset.

ผลลัพธ์:



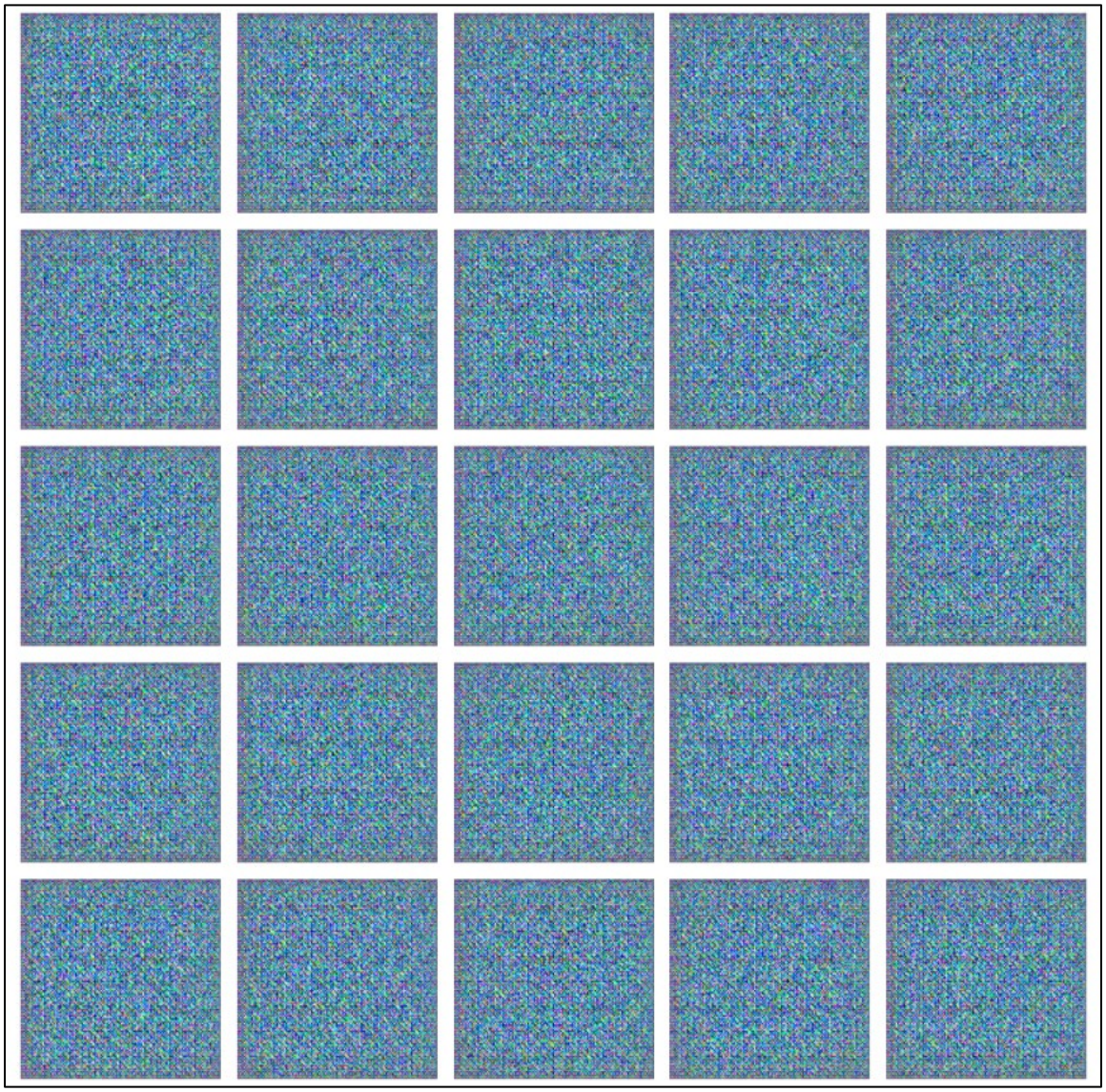
```python
### START CODE HERE ###
# Generate noise and create images
n_samples = 25
noise = get_noise(n_samples, z_dim, device)

# Generate images using the generator
with torch.no_grad():  # Disable gradient calculation for inference
    generated_images = generator(noise)

# Convert tensor to numpy for plotting
generated_images = generated_images * 0.5 + 0.5  # Denormalize images to [0, 1]
generated_images = generated_images.cpu().numpy()  # Move to CPU and convert to numpy

display_images(generated_images, n_cols=5, n_rows=5)
### END CODE HERE ###
```

ผลลัพธ์:



```python
def plot_losses_and_images(generator_losses, discriminator_losses, test_gen_img):
    plt.clf()  # Clear the current figure

    n_images = test_gen_img.size(0)  # Number of generated images
    grid_size = int(n_images ** 0.5)  # Calculate grid size for plotting

    # Create a figure for losses
    fig, ax = plt.subplots(1, 2, figsize=(10, 4))

    # Plot generator losses
    if generator_losses:
        ax[0].plot(generator_losses, label='Generator Loss', color='blue')
        ax[0].set_title('Generator Loss Over Time')
        ax[0].set_xlabel('Epochs')
        ax[0].set_ylabel('Loss')
        ax[0].legend()
    else:
        print("No generator loss data to plot.")

    # Plot discriminator losses
    if discriminator_losses:
        ax[1].plot(discriminator_losses, label='Discriminator Loss', color='red')
        ax[1].set_title('Discriminator Loss Over Time')
        ax[1].set_xlabel('Epochs')
        ax[1].set_ylabel('Loss')
        ax[1].legend()
    else:
        print("No discriminator loss data to plot.")

    plt.tight_layout()  # Adjust layout

    # Create a figure for generated images
    plt.figure(figsize=(12, 12))
    for i in range(n_images):
        ax = plt.subplot(grid_size, grid_size, i + 1)
        ax.imshow(test_gen_img[i].permute(1, 2, 0).clamp(0, 1))  # Change from (C, H, W) to (H, W, C) and clamp values
        ax.axis('off')  # Turn off axis

    plt.suptitle('Generated Images')
    plt.show()
```

```python
def train(generator, discriminator, gen_opt, disc_opt, criterion, dataloader, test_noise, z_dim, epochs=10, writer=None, checkpoint_path=None, device='c
    generator.train()
    discriminator.train()

    generator_losses = []
    discriminator_losses = []

    for epoch in range(epochs):
        mean_generator_loss = 0
        mean_discriminator_loss = 0
        cur_step = 0

        # Use tqdm for a progress bar
        train_bar = tqdm(dataloader, total=len(dataloader))

        for real in train_bar:
            real = real.to(device)  # Move to the correct device
            cur_batch_size = real.size(0)  # Get current batch size

            # Train discriminator
            disc_opt.zero_grad()

            # Generate fake images
            fake_noise = get_noise(cur_batch_size, z_dim, device=device)
            fake = generator(fake_noise)

            # Discriminator predictions
            disc_fake_pred = discriminator(fake.detach())
            disc_real_pred = discriminator(real)

            # Loss calculations
            disc_fake_loss = criterion(disc_fake_pred, torch.zeros_like(disc_fake_pred, device=device))
            disc_real_loss = criterion(disc_real_pred, torch.ones_like(disc_real_pred, device=device))

            # Total discriminator loss
            disc_loss = (disc_fake_loss + disc_real_loss) / 2
            mean_discriminator_loss += disc_loss.item()
            disc_loss.backward()
            disc_opt.step()

            # Train generator
            gen_opt.zero_grad()
```

```
        # Generate fake images again
        fake_noise_2 = get_noise(cur_batch_size, z_dim, device=device)
        fake_2 = generator(fake_noise_2)
        disc_fake_pred = discriminator(fake_2)

        # Generator loss
        gen_loss = criterion(disc_fake_pred, torch.ones_like(disc_fake_pred, device=device))
        gen_loss.backward()
        gen_opt.step()

        mean_generator_loss += gen_loss.item()
        cur_step += 1

        # Update progress bar
        train_bar.set_postfix(DiscLoss=disc_loss.item(), GenLoss=gen_loss.item())

    # Average losses for this epoch
    mean_generator_loss /= len(dataloader)
    mean_discriminator_loss /= len(dataloader)

    generator_losses.append(mean_generator_loss)
    discriminator_losses.append(mean_discriminator_loss)

    print(f'Epoch [{epoch+1}/{epochs}], Generator Loss: {mean_generator_loss:.4f}, Discriminator Loss: {mean_discriminator_loss:.4f}')

    # Generate test images for visualization
    with torch.no_grad():  # Disable gradient calculation for inference
        test_gen_img = generator(test_noise)
        plot_losses_and_images(generator_losses, discriminator_losses, test_gen_img)

    # Optionally save checkpoints
    if checkpoint_path:
        checkpoint = {
            'epoch': epoch,
            'generator_state_dict': generator.state_dict(),
            'discriminator_state_dict': discriminator.state_dict(),
            'gen_opt_state_dict': gen_opt.state_dict(),
            'disc_opt_state_dict': disc_opt.state_dict(),
        }
        torch.save(checkpoint, checkpoint_path)
```

ที่ใช้สำหรับฝึกโมเดล Generator และ Discriminator ใน Generative Adversarial Networks (GANs)

- The train function handles the entire training loop for a GAN, including updating both the generator and discriminator, tracking losses, displaying generated images, and optionally saving model checkpoints. It uses noise vectors to generate images and the discriminator's classification to guide the training of both models. Additionally, it provides visual feedback of the model's performance at the end of each epoch, which helps in monitoring the progress of the generator.

Parameters:

- **generator**: The generator model that produces images from noise.
- **discriminator**: The discriminator model that classifies images as real or fake.
- **gen_opt**: The optimizer for the generator model.
- **disc_opt**: The optimizer for the discriminator model.
- **criterion**: The loss function used for both the generator and discriminator.
- **dataloader**: The DataLoader that provides batches of real images.

- **test_noise**: Fixed noise used to generate images after each epoch for visualizing the generator's progress.
- **z_dim**: The dimensionality of the noise vector for the generator.
- **epochs**: Number of epochs to train the models (default is 10).
- **writer**: Optionally used for TensorBoard logging (not used in this version).
- **checkpoint_path**: Path to save the model checkpoints (if provided).
- **device**: Device to train the models on (either 'cpu' or 'cuda').

Main Steps:

```
def train(generator, discriminator, gen_opt, disc_opt, criterion, dataloader, test_noise, z_dim, epochs=10, writer=None, checkpoint_path=None, device='cpu'):
    generator.train()
    discriminator.train()

    generator_losses = []
    discriminator_losses = []

    for epoch in range(epochs):
        mean_generator_loss = 0
        mean_discriminator_loss = 0
        cur_step = 0

        # Use tqdm for a progress bar
        train_bar = tqdm(dataloader, desc=f'🚀 Training Epoch [{epoch + 1}/{epochs}]', unit='batch')

        for real in train_bar:
            real = real.to(device)  # Move to the correct device
            cur_batch_size = real.size(0)  # Get current batch size
```

1. **Model Setup** (เตรียมการฝึก):
    - The generator and discriminator are set to training mode using generator.train() and discriminator.train().
2. **Loss Tracking**:
    - Lists generator_losses and discriminator_losses are initialized to store the losses over epochs.
3. **Epoch Loop**:
    - วนลูป for ตามจำนวน epoch ที่กำหนด (ค่าเริ่มต้นคือ 10 epoch)
        - ภายในลูปจะเก็บค่า loss เฉลี่ยของทั้ง Generator และ Discriminator ไว้ในตัวแปร mean_generator_loss และ mean_discriminator_loss
        - The main training loop runs for the specified number of epochs.
    - **Progress Bar**:
        - ใช้ tqdm สร้าง progress bar เพื่อแสดงความคืบหน้าการฝึกในแต่ละ epoch
    - **Batch Loop**:
        - วนลูป for สำหรับข้อมูลจริง (real) ในแต่ละ batch ของ dataloader

- ย้ายข้อมูลจริงไปยังอุปกรณ์ที่เลือก (CPU หรือ GPU) ด้วย real.to(device)
- บันทึกขนาดของ batch ปัจจุบันไว้ใน cur_batch_size

- For each batch in the dataloader, the following steps are performed:

```
# Train discriminator
disc_opt.zero_grad()

# Generate fake images
fake_noise = get_noise(cur_batch_size, z_dim, device=device)
fake = generator(fake_noise)

# Discriminator predictions
disc_fake_pred = discriminator(fake.detach())
disc_real_pred = discriminator(real)

# Loss calculations
disc_fake_loss = criterion(disc_fake_pred, torch.zeros_like(disc_fake_pred, device=device))
disc_real_loss = criterion(disc_real_pred, torch.ones_like(disc_real_pred, device=device))

# Total discriminator loss
disc_loss = (disc_fake_loss + disc_real_loss) / 2
mean_discriminator_loss += disc_loss.item()
disc_loss.backward()
disc_opt.step()
```

- **Train Discriminator**:
  - **Generate fake images**: Noise is generated using get_noise, and fake images are produced by the generator. (สร้าง noise vector สำหรับสร้างภาพปลอมด้วย get_noise, สร้างภาพปลอมจาก noise vector ด้วย fake = generator(fake_noise))
  - **Discriminator predictions**: The discriminator is evaluated on both the real images from the dataset and the fake images generated by the generator.
  - **Discriminator loss**: The loss for the discriminator is computed as the average of two losses คำนวณ loss สำหรับ Discriminator ทั้งภาพจริงและภาพปลอม
    - โดยการคำนวณ loss รวมของ Discriminator (disc_loss) จะใช้เป็นค่าเฉลี่ยของ loss จากภาพจริงและภาพปลอม:
      - **disc_fake_loss:** The loss for classifying fake images as "fake" (comparing its output to a tensor of zeros). **(ส่งค่าใกล้เคียง 0)**
      - **disc_real_loss:** The loss for classifying real images as "real" (comparing its output to a tensor of ones). **(ส่งค่าใกล้เคียง 1)**
  - **Update discriminator**: The discriminator's loss is backpropagated, and its optimizer updates the model weights.
    - บันทึกค่า loss ของ Discriminator (disc_loss.item()) และคำนวณ gradient ด้วย disc_loss.backward()
    - ปรับปรุงน้ำหนักของ Discriminator ด้วย disc_opt.step()

```
# Train generator
gen_opt.zero_grad()

# Generate fake images again
fake_noise_2 = get_noise(cur_batch_size, z_dim, device=device)
fake_2 = generator(fake_noise_2)
disc_fake_pred = discriminator(fake_2)

# Generator loss
gen_loss = criterion(disc_fake_pred, torch.ones_like(disc_fake_pred, device=device))
gen_loss.backward()
gen_opt.step()

mean_generator_loss += gen_loss.item()
cur_step += 1

# Update progress bar
train_bar.set_postfix(DiscLoss=disc_loss.item(), GenLoss=gen_loss.item())
```

- **Train Generator**:
  - **Generate fake images again**: New noise is generated and passed to the generator to produce another batch of fake images.
    - สร้าง noise vector ใหม่ (fake_noise_2) สำหรับฝึก Generator อย่างแยกอิสระ
    - สร้างภาพปลอมจาก noise vector ใหม่ (fake_2 = generator(fake_noise_2))
  - **Generator loss**: The generator's loss is computed based on the discriminator's classification of these fake images. The generator's goal is to fool the discriminator, so the loss compares the discriminator's outputs to a tensor of ones (i.e., the generator wants the fake images to be classified as "real").
    - คำนวณผลลัพธ์ของ Discriminator สำหรับภาพปลอมชุดที่สอง (disc_fake_pred)
    - คำนวณ loss สำหรับ Generator โดยคาดหวังให้ Discriminator แยกแยะภาพปลอมชุดที่สองไม่ได้ **(ส่งค่าใกล้เคียง 1 (คิดว่าเป็นรูปจริง))**
  - **Update generator**: The generator's loss is backpropagated, and its optimizer updates the generator's weights.
    - บันทึกค่า loss ของ Generator (gen_loss.item()) และคำนวณ gradient ด้วย gen_loss.backward()
    - ปรับปรุงน้ำหนักของ Generator ด้วย gen_opt.step()
- **Progress Bar Update**:
  - The progress bar is updated with the current discriminator and generator losses after processing each batch.

- อัพเดท progress bar ของ tqdm ด้วยค่า loss ของ Discriminator (DiscLoss) และ Generator (GenLoss)

```
# Average losses for this epoch
mean_generator_loss /= len(dataloader)
mean_discriminator_loss /= len(dataloader)

generator_losses.append(mean_generator_loss)
discriminator_losses.append(mean_discriminator_loss)

print(f'Epoch [{epoch+1}/{epochs}], Generator Loss: {mean_generator_loss:.4f}, Discriminator Loss: {mean_discriminator_loss:.4f}')

# Generate test images for visualization
with torch.no_grad():  # Disable gradient calculation for inference
    test_gen_img = generator(test_noise)
    plot_losses_and_images(generator_losses, discriminator_losses, test_gen_img)

# Optionally save checkpoints
if checkpoint_path:
    checkpoint = {
        'epoch': epoch,
        'generator_state_dict': generator.state_dict(),
        'discriminator_state_dict': discriminator.state_dict(),
        'gen_opt_state_dict': gen_opt.state_dict(),
        'disc_opt_state_dict': disc_opt.state_dict(),
    }
    torch.save(checkpoint, checkpoint_path)
```

4. **End of Epoch**:
    - After each epoch, the average generator and discriminator losses are calculated and stored in the respective lists.
        - คำนวณค่า loss เฉลี่ยของ Discriminator (mean_discriminator_loss) และ Generator (mean_generator_loss) โดยหารด้วยจำนวน batch ใน dataloader
5. **Visualize Generated Images**:
    - After each epoch, the generator produces images from the fixed test_noise, and these images are displayed using the plot_losses_and_images function to visually inspect the training progress. (สร้างภาพจาก Generator โดยใช้ test_noise และแสดงผลลัพธ์)
6. **Checkpoint Saving**:
    - If a checkpoint_path is provided, the function saves the current state of the generator, discriminator, and optimizers as a checkpoint. This can be used to resume training later.

```
### START CODE HERE ###
# Define optimizers
lr = 0.0002
beta1 = 0.5

gen_opt = optim.Adam(generator.parameters(), lr=lr, betas=(beta1, 0.999))
disc_opt = optim.Adam(discriminator.parameters(), lr=lr, betas=(beta1, 0.999))

# Define loss function
criterion = torch.nn.BCELoss()

# Create a fixed noise vector for generating sample images
test_noise = get_noise(25, z_dim, device)

# Initialize TensorBoard writer
writer = SummaryWriter(log_dir='logs/gan_experiment')

# Specify checkpoint path
checkpoint_path = 'checkpoints/gan_checkpoint.pth'
os.makedirs(os.path.dirname(checkpoint_path), exist_ok=True)  # Create directories if they don't exist

num_epochs=10
# Call the training function
train(generator, discriminator, gen_opt, disc_opt, criterion, dataloader, test_noise, z_dim, epochs=num_epochs, writer=writer, checkpoint_path=checkpoint_path, device=device)

# Close the TensorBoard writer
writer.close()
### END CODE HERE ###
```

[14] This code defines the key components for training the GAN, including optimizers, the loss function, and logging mechanisms using TensorBoard. It creates a fixed noise vector for consistent visualization of generated images and specifies a path for saving model checkpoints. The train function is then called to run the training loop for a specified number of epochs. After the training process, the TensorBoard writer is closed to finalize the logs.

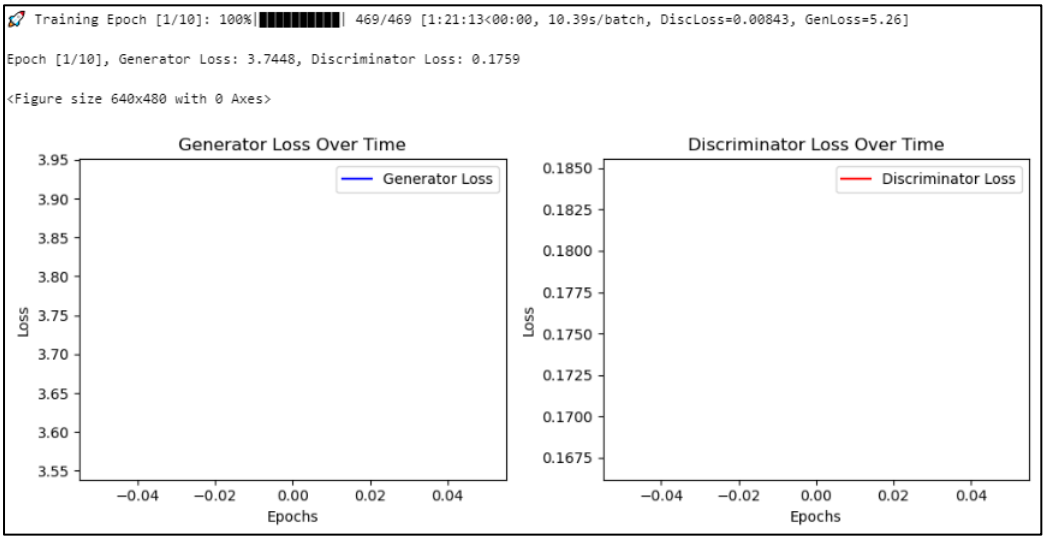
1. กำหนด Optimizer
   - **gen_opt** และ **disc_opt** เป็นตัวเพิ่มประสิทธิภาพ (optimizer) สำหรับปรับปรุงน้ำหนักของโมเดล Generator และ Discriminator ตามลำดับ
   - ใช้ **optim.Adam** ซึ่งเป็นอัลกอริทึมการเพิ่มประสิทธิภาพที่นิยมใช้ใน GAN
   - กำหนดอัตราการเรียนรู้ (learning rate) เป็น 0.0002 และค่า beta1 และ beta2 สำหรับอัลกอริทึม Adam
2. กำหนด Loss Function
   - criterion เป็นฟังก์ชันการคำนวณ loss (ค่าความผิดพลาด)
   - ใช้ **torch.nn.BCELoss** ซึ่งเหมาะสำหรับปัญหาการจำแนกแบบ binary (เช่น แยกภาพจริงจากภาพปลอม)

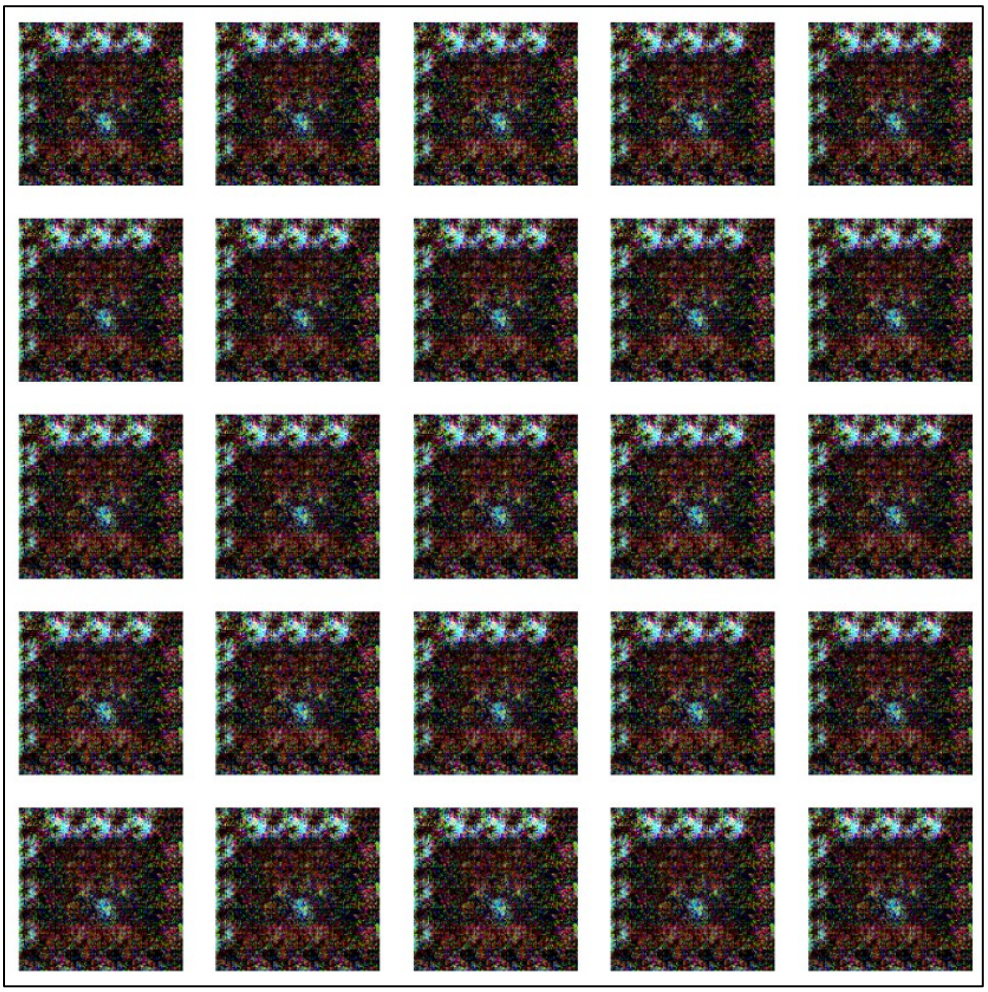
3. สร้าง Noise Vector สำหรับสร้างภาพตัวอย่าง
   - **test_noise** เป็น noise vector ที่ถูกสร้างขึ้นโดยใช้ฟังก์ชัน get_noise
   - Noise vector นี้จะถูกใช้เพื่อสร้างภาพตัวอย่างในระหว่างการฝึกเพื่อตรวจสอบประสิทธิภาพของ Generator
4. สร้าง TensorBoard Writer
   - writer เป็นตัวเขียนสำหรับ TensorBoard ซึ่งใช้ในการบันทึกและแสดงข้อมูลการฝึกในรูปแบบที่เป็นภาพและกราฟ

5. กำหนดเส้นทางสำหรับบันทึก Checkpoint
    - checkpoint_path เป็น path ที่ใช้ในการบันทึกสถานะของโมเดลและตัวเพิ่มประสิทธิภาพในระหว่างการฝึก
6. เรียกฟังก์ชัน train ที่เขียนไปข้างต้น โดยกำหนดรอบ = 10 รอบ
    - เรียกฟังก์ชัน train เพื่อเริ่มการฝึกโมเดล GAN โดยส่งพารามิเตอร์ต่าง ๆ เช่น โมเดล, ตัวเพิ่มประสิทธิภาพ, ฟังก์ชัน loss, ชุดข้อมูล, noise vector, จำนวน epoch, ตัวเขียนสำหรับ TensorBoard, เส้นทางสำหรับบันทึก checkpoint และอุปกรณ์ที่ใช้ (CPU หรือ GPU)
7. ปิด TensorBoard Writer
    - หลังจากการฝึกเสร็จสิ้น ให้นำ writer.close() เพื่อปิดการบันทึกข้อมูลใน TensorBoard

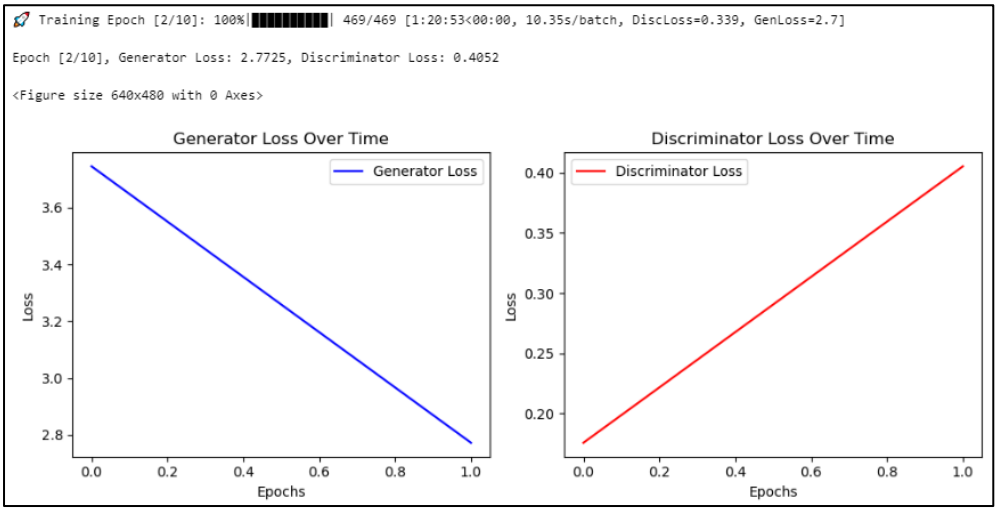
ผลลัพธ์:

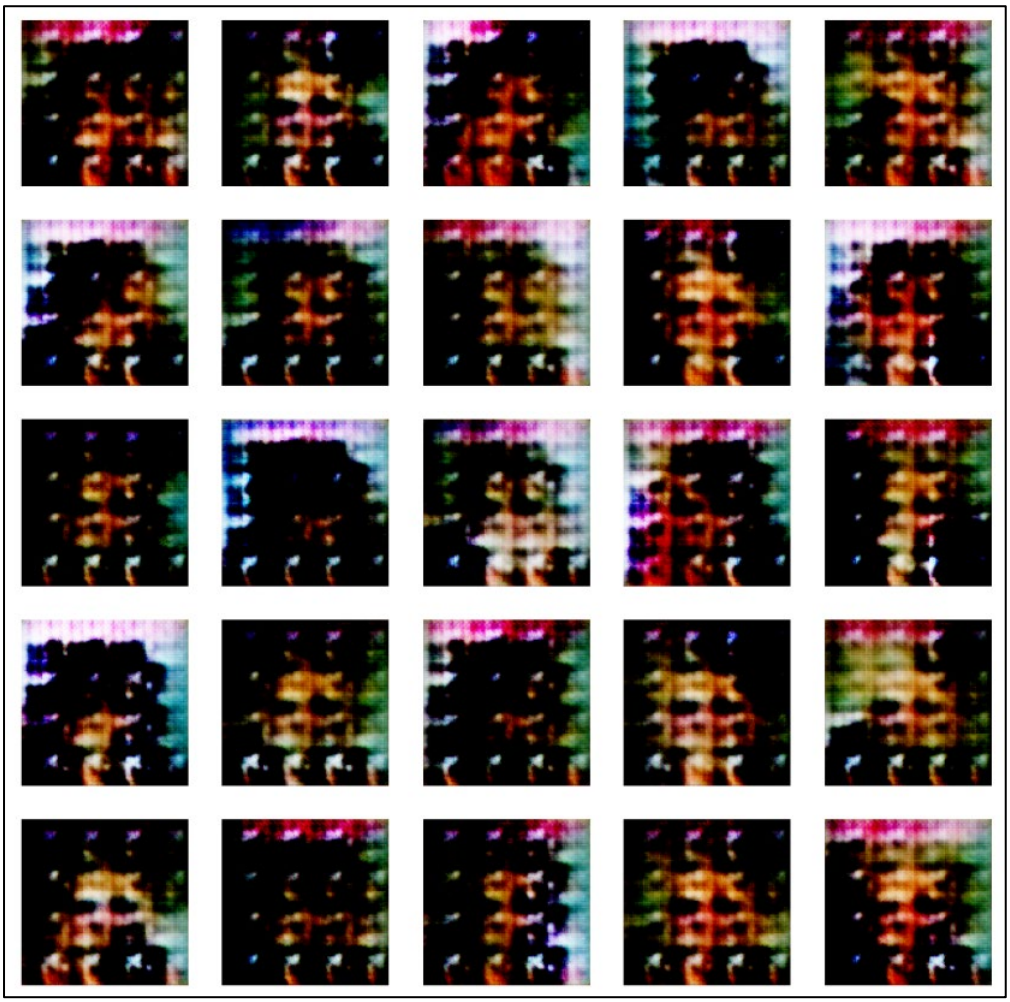
- **รอบที่ 1**



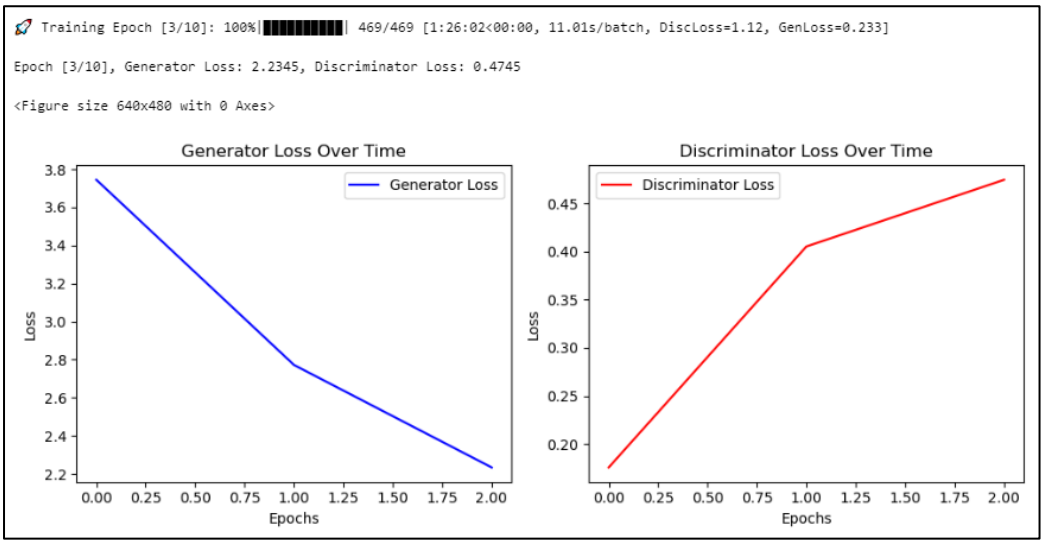


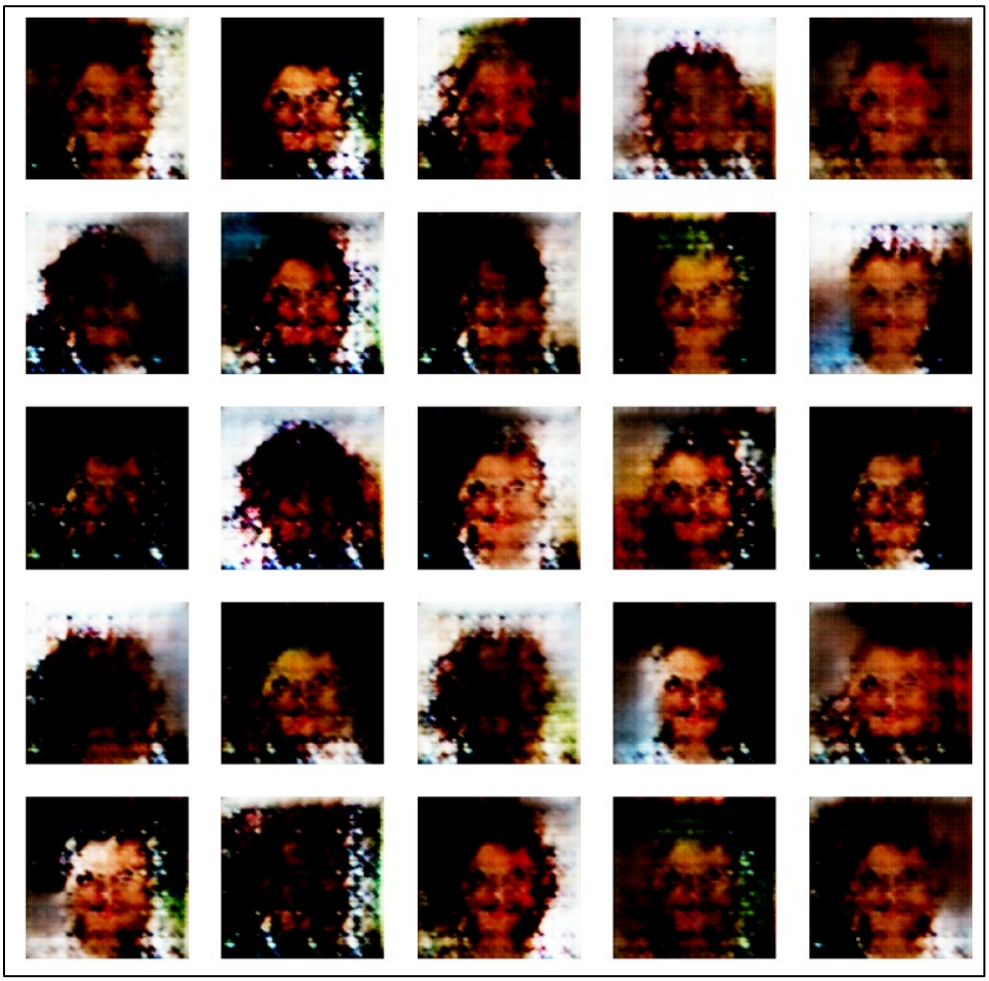
- **รอบที่ 2**


Training Epoch [2/10]: 100%| 469/469 [1:20:53<00:00, 10.35s/batch, DiscLoss=0.339, GenLoss=2.7]
Epoch [2/10], Generator Loss: 2.7725, Discriminator Loss: 0.4052
<Figure size 640x480 with 0 Axes>
Generator Loss Over Time
Generator Loss
Loss
Epochs
Discriminator Loss Over Time
Discriminator Loss
Loss
Epochs

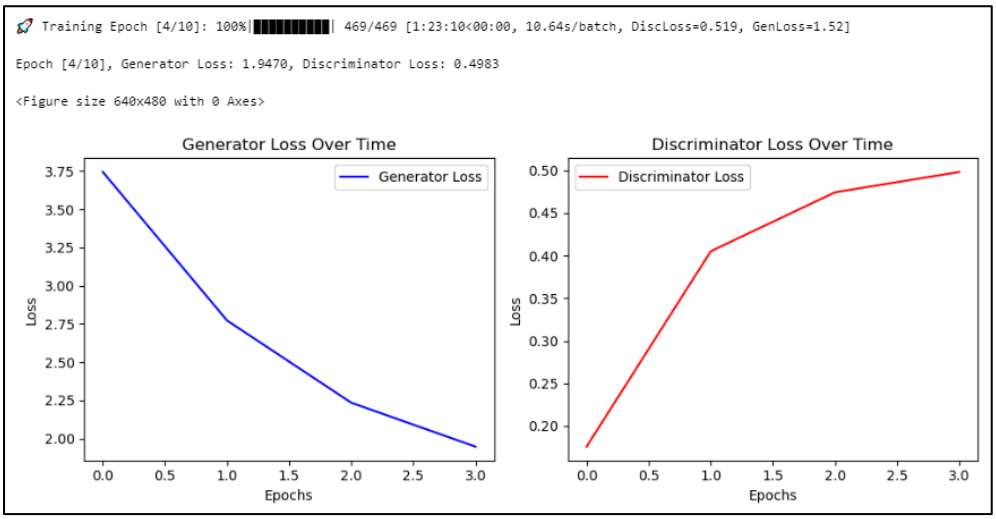


- **รอบที่ 3**

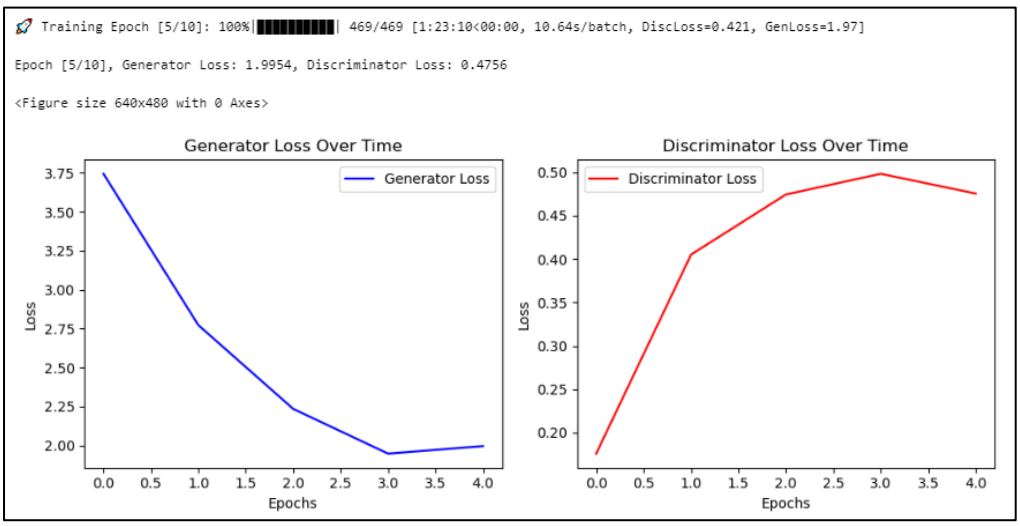

Training Epoch [3/10]: 100%| 469/469 [1:26:02<00:00, 11.01s/batch, DiscLoss=1.12, GenLoss=0.233]
Epoch [3/10], Generator Loss: 2.2345, Discriminator Loss: 0.4745
<Figure size 640x480 with 0 Axes>
Generator Loss Over Time
Generator Loss
Loss
Epochs
Discriminator Loss Over Time
Discriminator Loss
Loss
Epochs

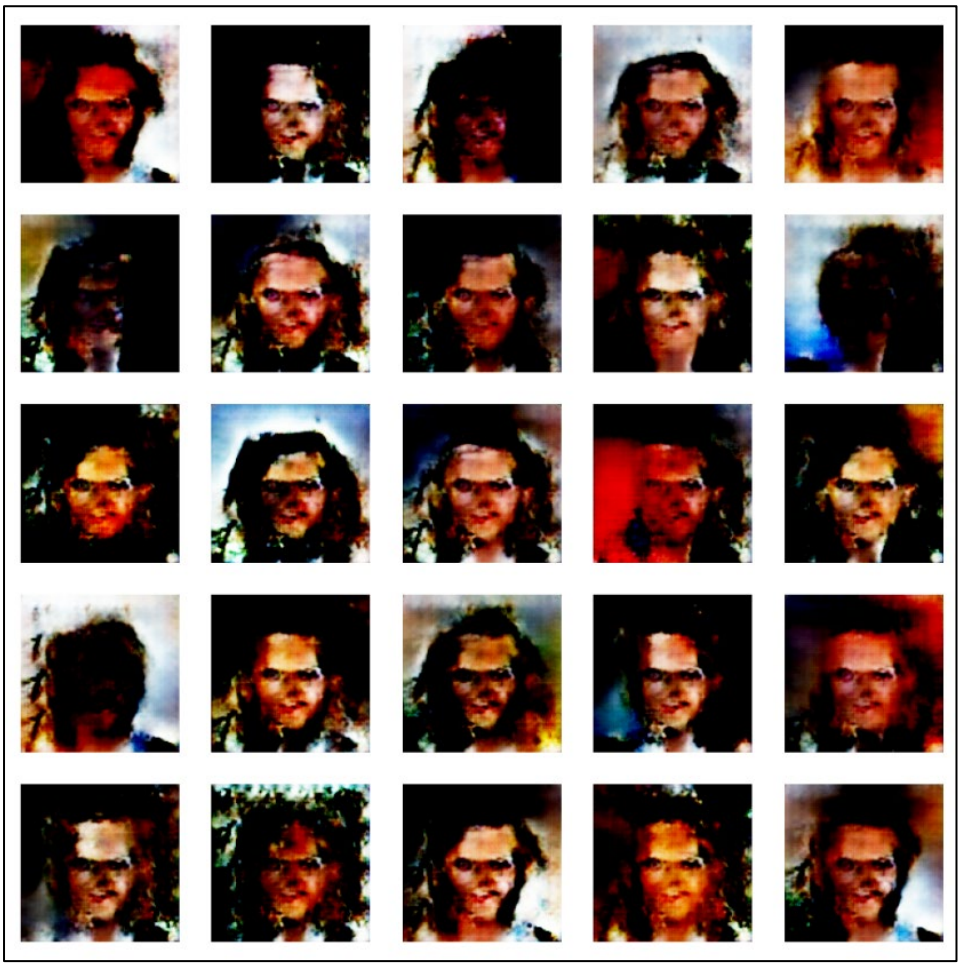


- **รอบที่ 4**

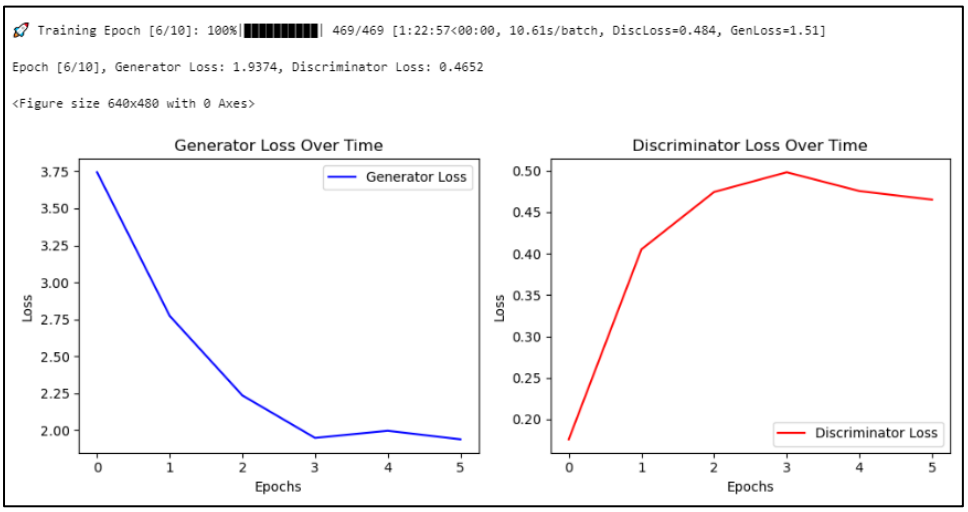

Training Epoch [4/10]: 100%| 469/469 [1:23:10<00:00, 10.64s/batch, DiscLoss=0.519, GenLoss=1.52]
Epoch [4/10], Generator Loss: 1.9470, Discriminator Loss: 0.4983
<Figure size 640x480 with 0 Axes>
Generator Loss Over Time
Generator Loss
Loss
Epochs
Discriminator Loss Over Time
Discriminator Loss
Loss
Epochs

- **รอบที่ 5**

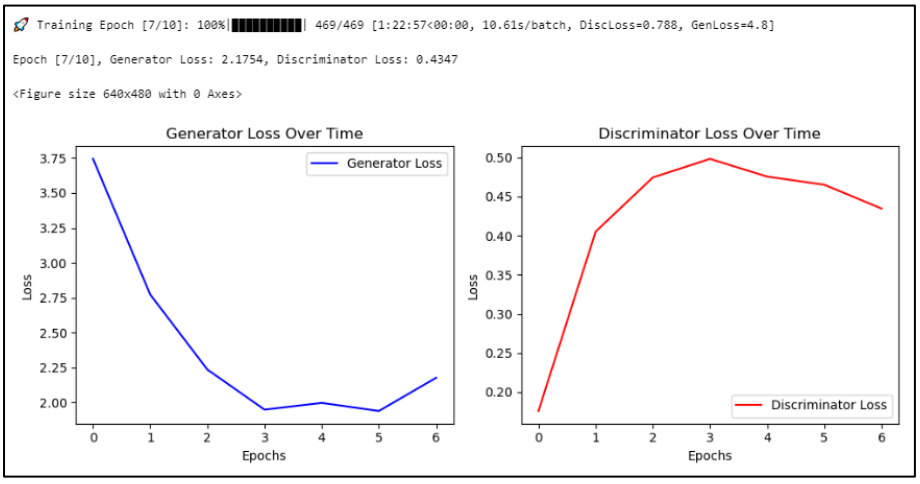

Training Epoch [5/10]: 100%| 469/469 [1:23:10<00:00, 10.64s/batch, DiscLoss=0.421, GenLoss=1.97]
Epoch [5/10], Generator Loss: 1.9954, Discriminator Loss: 0.4756
<Figure size 640x480 with 0 Axes>
Generator Loss Over Time
Generator Loss
Loss
Epochs
Discriminator Loss Over Time
Discriminator Loss
Loss
Epochs

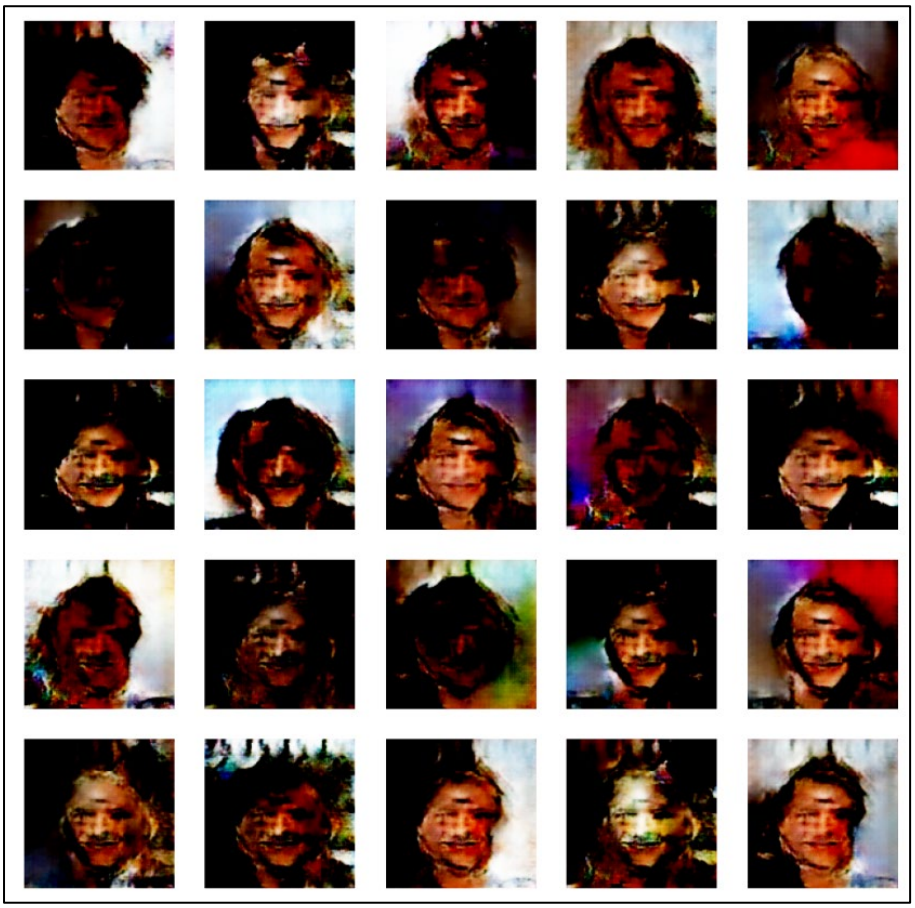


- **รอบที่ 6**

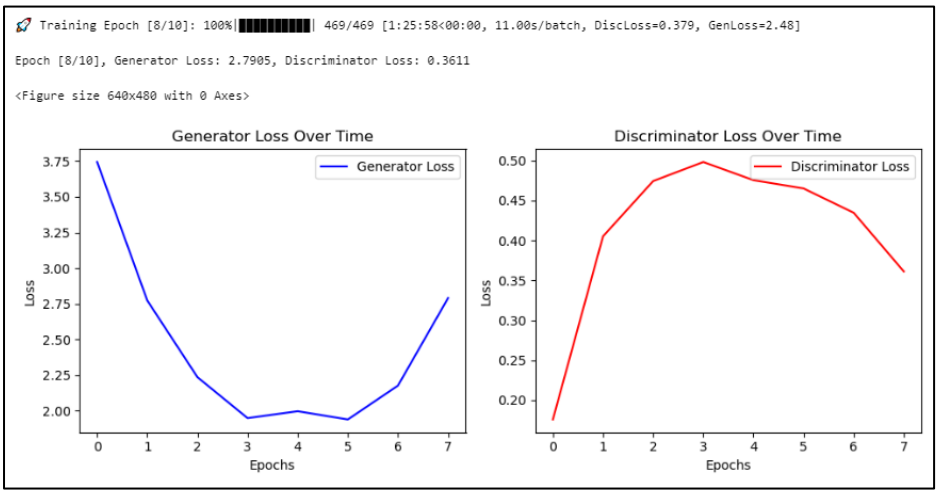

Training Epoch [6/10]: 100%| 469/469 [1:22:57<00:00, 10.61s/batch, DiscLoss=0.484, GenLoss=1.51]
Epoch [6/10], Generator Loss: 1.9374, Discriminator Loss: 0.4652
<Figure size 640x480 with 0 Axes>
Generator Loss Over Time
Generator Loss
Loss
Epochs
Discriminator Loss Over Time
Discriminator Loss
Loss
Epochs

- **รอบที่ 7**

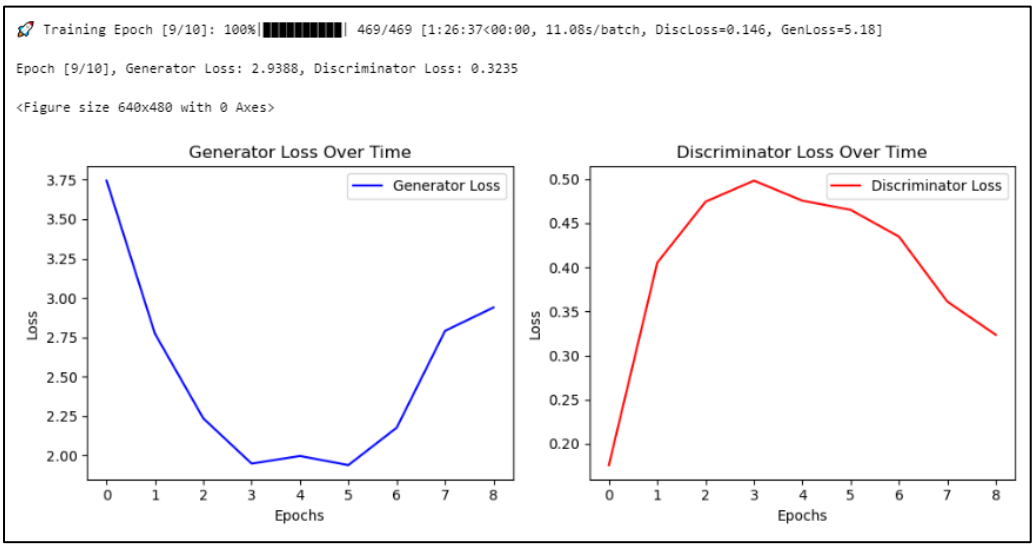

Training Epoch [7/10]: 100%| 469/469 [1:22:57<00:00, 10.61s/batch, DiscLoss=0.788, GenLoss=4.8]
Epoch [7/10], Generator Loss: 2.1754, Discriminator Loss: 0.4347
<Figure size 640x480 with 0 Axes>
Generator Loss Over Time
Generator Loss
Loss
Epochs
Discriminator Loss Over Time
Discriminator Loss
Loss
Epochs

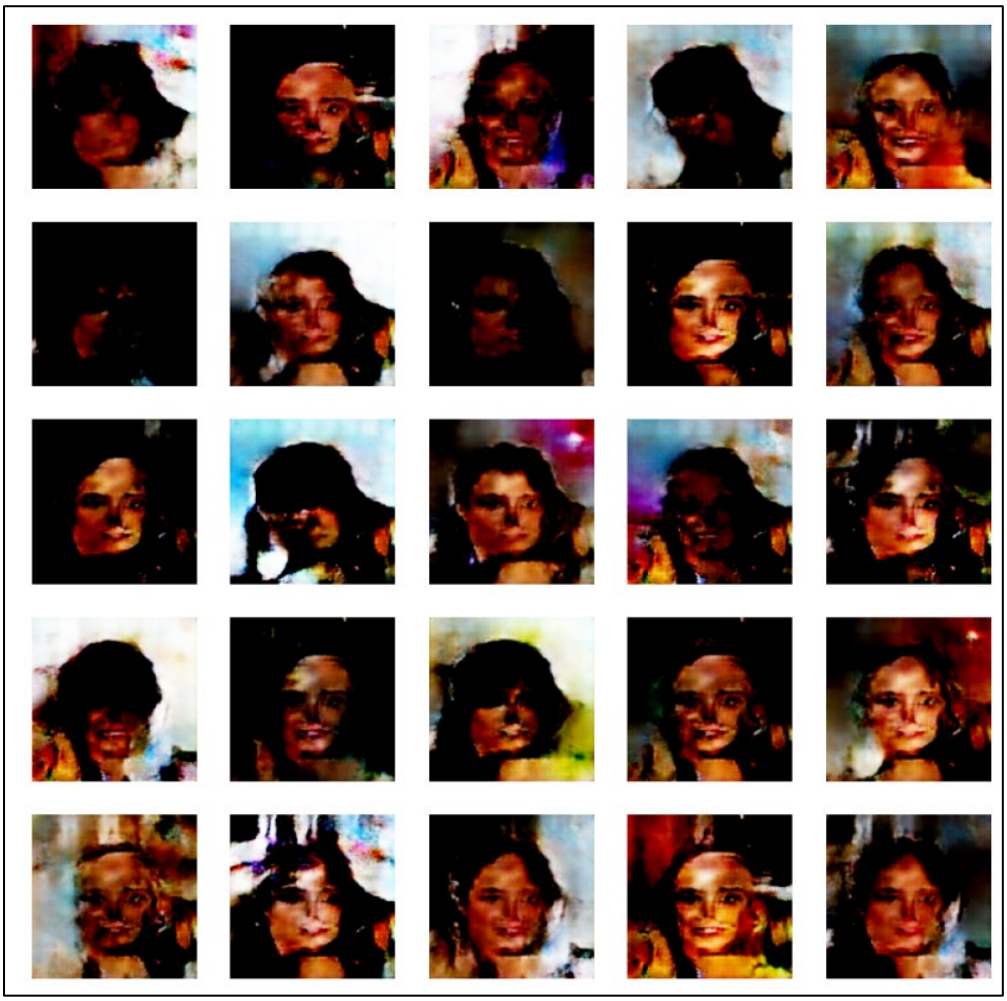


- **รอบที่ 8**

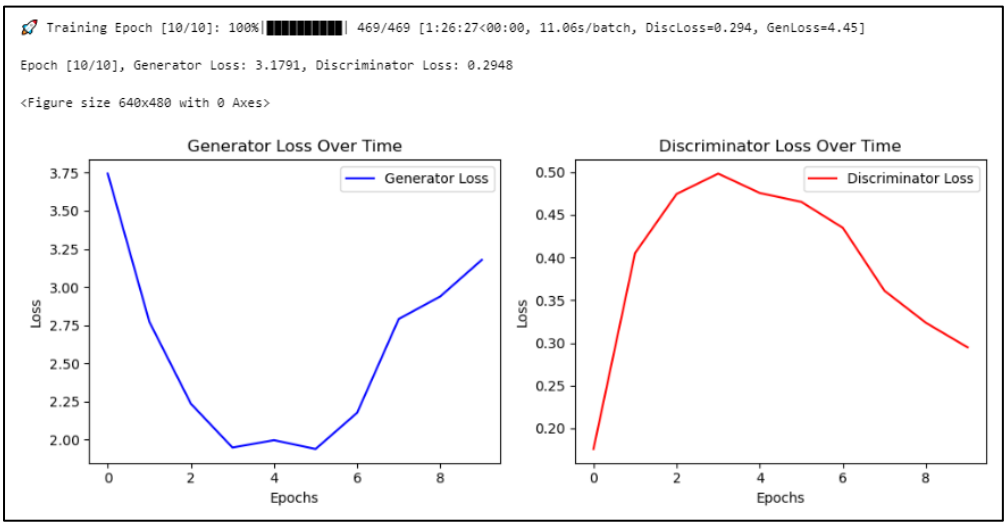

Training Epoch [8/10]: 100%| 469/469 [1:25:58<00:00, 11.00s/batch, DiscLoss=0.379, GenLoss=2.48]
Epoch [8/10], Generator Loss: 2.7905, Discriminator Loss: 0.3611
<Figure size 640x480 with 0 Axes>
Generator Loss Over Time
Generator Loss
Loss
Epochs
Discriminator Loss Over Time
Discriminator Loss
Loss
Epochs

- **รอบที่ 9**


Training Epoch [9/10]: 100%| 469/469 [1:26:37<00:00, 11.08s/batch, DiscLoss=0.146, GenLoss=5.18]
Epoch [9/10], Generator Loss: 2.9388, Discriminator Loss: 0.3235
<Figure size 640x480 with 0 Axes>
Generator Loss Over Time
Generator Loss
Loss
Epochs
Discriminator Loss Over Time
Discriminator Loss
Loss
Epochs



- **รอบที่ 10**


Training Epoch [10/10]: 100%| | 469/469 [1:26:27<00:00, 11.06s/batch, DiscLoss=0.294, GenLoss=4.45]
Epoch [10/10], Generator Loss: 3.1791, Discriminator Loss: 0.2948
<Figure size 640x480 with 0 Axes>
Generator Loss Over Time
Generator Loss
Loss
Epochs
Discriminator Loss Over Time
Discriminator Loss
Loss
Epochs

สรุปผล:

- **Generator Loss:** เป็นค่า Loss ของส่วนที่สร้างข้อมูลใหม่ (Generator) ค่านี้ควรจะลดลงเรื่อยๆ หมายความว่า Generator กำลังเรียนรู้ที่จะสร้างข้อมูลที่เหมือนจริงมากขึ้น
- **Discriminator Loss:** เป็นค่า Loss ของส่วนที่แยกแยะข้อมูลจริงและปลอม (Discriminator) ค่านี้ก็ควรจะลดลงในช่วงแรกๆ เพราะ Discriminator กำลังเรียนรู้ที่จะแยกแยะข้อมูลได้ดีขึ้น แต่หลังจากนั้นค่า Loss อาจจะผันผวนขึ้นลงได้บ้าง เพราะ Generator ก็พยายามสร้างข้อมูลที่หลอก Discriminator ได้มากขึ้น

วิเคราะห์ผลการฝึกสอน:

- **ช่วงแรก (Epoch 1-5):** ในช่วงเริ่มต้น ค่าการสูญเสียของ Generator ลดลงอย่างชัดเจน จาก 3.7448 มาที่ 1.9954 ขณะที่ Discriminator Loss มีการเพิ่มขึ้นจาก 0.1759 มาที่ 0.4756 นี่แสดงว่า Generator เริ่มสร้างข้อมูลที่มีคุณภาพดีขึ้น ขณะที่ Discriminator ยังสามารถจำแนกข้อมูลจริงและปลอมได้ดีอยู่

- **ช่วงกลาง (Epoch 6-7):** ค่าการสูญเสียของ Generator เริ่มมีความไม่แน่นอน โดยมีการเพิ่มขึ้นจาก 1.9374 เป็น 2.1754 ขณะที่ Discriminator Loss มีแนวโน้มลดลงเล็กน้อย ซึ่งบ่งชี้ว่า Discriminator เริ่มทำงานได้ดีขึ้นในการจำแนกข้อมูล
- **ช่วงท้าย (Epoch 8-10):** Generator Loss เริ่มเพิ่มขึ้นอย่างต่อเนื่อง โดยจาก 2.7905 เป็น 3.1791 ขณะที่ Discriminator Loss ลดลงจาก 0.3611 เป็น 0.2948 ซึ่งหมายความว่า Discriminator อาจเริ่มทำงานได้ดีเกินไป จนทำให้ Generator มีความยากลำบากในการปรับปรุงคุณภาพของข้อมูลที่สร้างขึ้น

```python
# Generate and display images
def generate_images(generator, num_images=25, z_dim=z_dim, device='cpu'):
    generator.eval()  # Set to evaluation mode
    with torch.no_grad():
        noise = get_noise(num_images, z_dim, device)
        generated_imgs = generator(noise).clamp(-1, 1)  # Clamp to [-1, 1]
        return (generated_imgs + 1) / 2  # Scale to [0, 1]

### START CODE HERE ###
# Generate and display images after training
generated_images = generate_images(generator, num_images=25, z_dim=z_dim, device=device)

display_images(generated_images, n_cols=5, n_rows=5)
### END CODE HERE ###
```

ผลลัพธ์:

- ซึ่งอาจจะปรับปรุงการ train โดยการปรับ Hyperparameter: ลองปรับค่า learning rate ให้ต่ำลง หรือใช้ learning rate scheduler เพื่อปรับค่า learning rate แบบอัตโนมัติ