Pictofacio

1. Motivation and significance of the study

This topic is using GAN as a model for generating random trained images as the beginning of the ideas which we want to generate “Bob Ross-style” art but due to lack of dataset training with complex textures so it ends up with Stanford dogs image dataset instead.

This topic is interesting because it reflects a frontier of modern AI, where neural networks are capable of imagining new content. GAN are among the most fascinating deep learning architectures, as they combine generator and discriminator, leading to remarkable visual realism.

2. Why deep learning is essential for this study

Traditional image generation techniques such as texture synthesis, rule-based pixel blending cannot produce highly complex, realistic images because they rely on handcrafted rules and lack the ability to learn abstract visual structures like shapes, textures, and lighting.

In contrast, Deep Learning, specifically GAN, excels by learning real image distributions through adversarial training, where the Generator creates realistic samples and the Discriminator detects fakes, driving mutual, continuous improvement.

3. Deep learning architecture

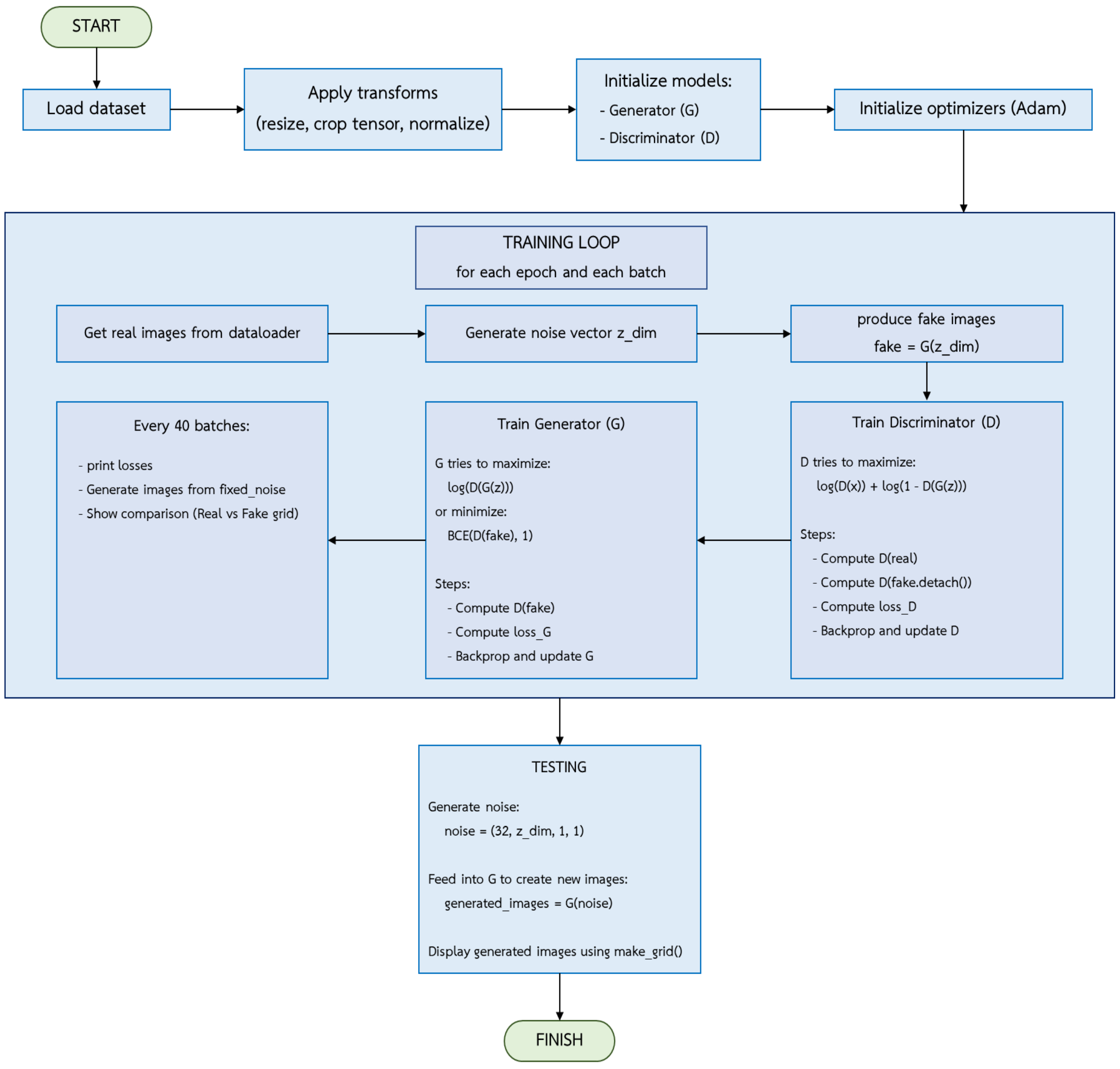


Figure 1. overall workflow procedure of the model.

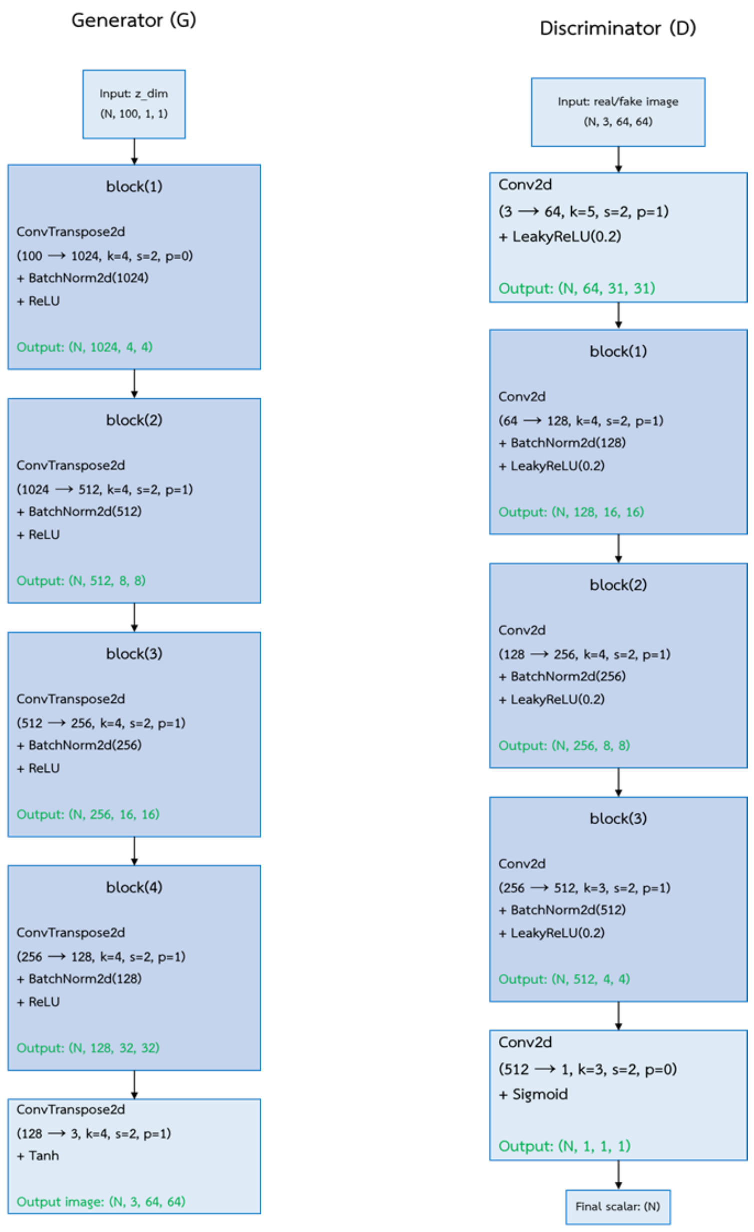
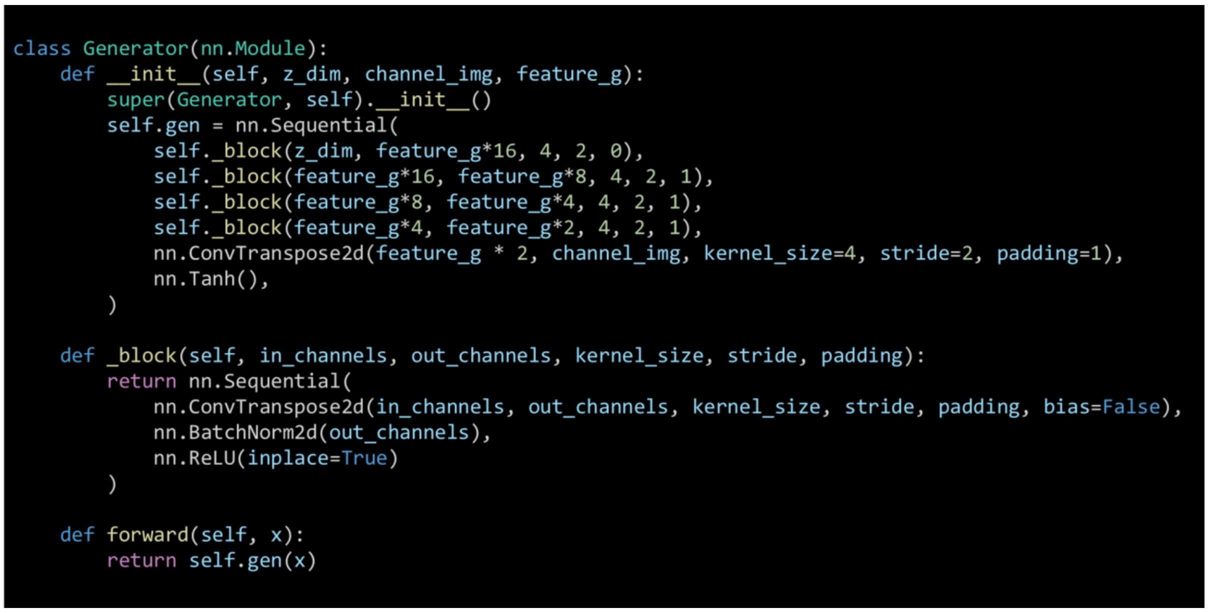

Generator (G)
Input: z_dim (N, 100, 1, 1)
block(1) ConvTranspose2d (100 ⟶ 1024, k=4, s=2, p=0) + BatchNorm2d(1024) + ReLU Output: (N, 1024, 4, 4)
block(2) ConvTranspose2d (1024 ⟶ 512, k=4, s=2, p=1) + BatchNorm2d(512) + ReLU Output: (N, 512, 8, 8)
block(3) ConvTranspose2d (512 ⟶ 256, k=4, s=2, p=1) + BatchNorm2d(256) + ReLU Output: (N, 256, 16, 16)
block(4) ConvTranspose2d (256 ⟶ 128, k=4, s=2, p=1) + BatchNorm2d(128) + ReLU Output: (N, 128, 32, 32)
ConvTranspose2d (128 ⟶ 3, k=4, s=2, p=1) + Tanh Output image: (N, 3, 64, 64)
Discriminator (D)
Input: real/fake image (N, 3, 64, 64)
Conv2d (3 ⟶ 64, k=5, s=2, p=1) + LeakyReLU(0.2) Output: (N, 64, 31, 31)
block(1) Conv2d (64 ⟶ 128, k=4, s=2, p=1) + BatchNorm2d(128) + LeakyReLU(0.2) Output: (N, 128, 16, 16)
block(2) Conv2d (128 ⟶ 256, k=4, s=2, p=1) + BatchNorm2d(256) + LeakyReLU(0.2) Output: (N, 256, 8, 8)
block(3) Conv2d (256 ⟶ 512, k=3, s=2, p=1) + BatchNorm2d(512) + LeakyReLU(0.2) Output: (N, 512, 4, 4)
Conv2d (512 ⟶ 1, k=3, s=2, p=0) + Sigmoid Output: (N, 1, 1, 1)
Final scalar: (N)

Figure 2. Deep convolutional generative adversarial networks model both generator and discriminator (DCGAN).

4. Code explanation

All code placed in the file pictofacio.ipynb is organized into sections based on topics. The first section "GAN model" defines both the generator and the discriminator. The second section "Train model" contains the training part which loads data from Stanford dogs image dataset.

Link : https://github.com/Hamtak0/pictofacio/blob/main/pictofacio.ipynb

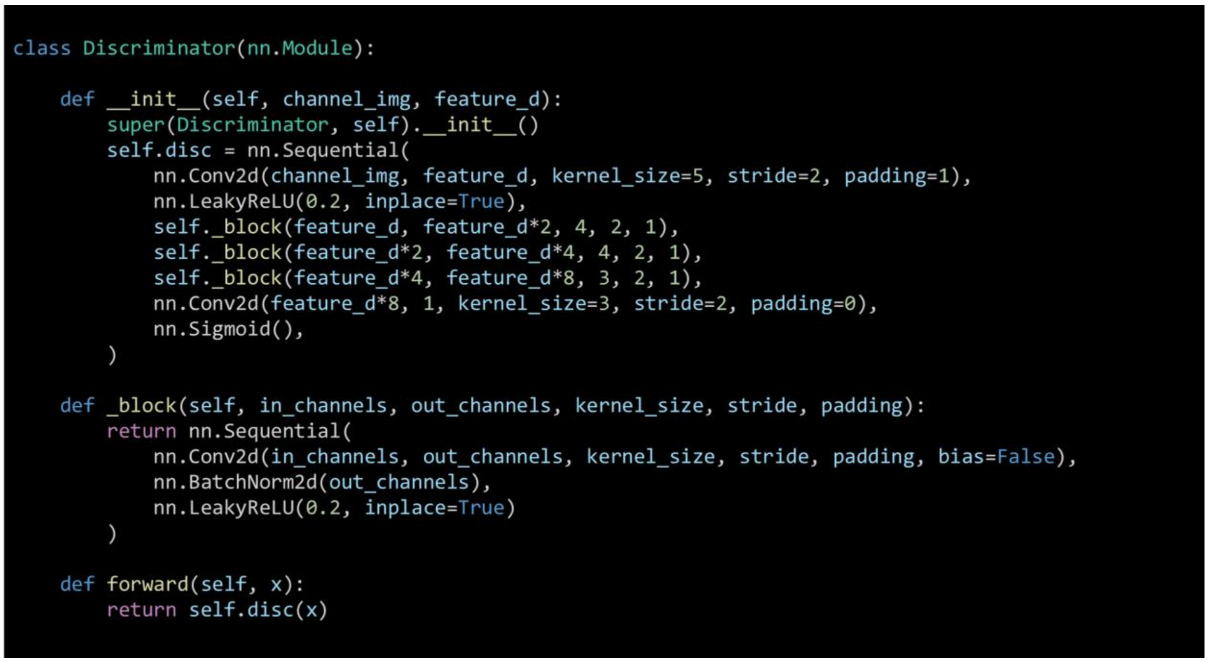


```python
class Generator(nn.Module):
    def __init__(self, z_dim, channel_img, feature_g):
        super(Generator, self).__init__()
        self.gen = nn.Sequential(
            self._block(z_dim, feature_g*16, 4, 2, 0),
            self._block(feature_g*16, feature_g*8, 4, 2, 1),
            self._block(feature_g*8, feature_g*4, 4, 2, 1),
            self._block(feature_g*4, feature_g*2, 4, 2, 1),
            nn.ConvTranspose2d(feature_g * 2, channel_img, kernel_size=4, stride=2, padding=1),
            nn.Tanh(),
        )

    def _block(self, in_channels, out_channels, kernel_size, stride, padding):
        return nn.Sequential(
            nn.ConvTranspose2d(in_channels, out_channels, kernel_size, stride, padding, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True)
        )

    def forward(self, x):
        return self.gen(x)
```

Figure 3. Generator code implementation.



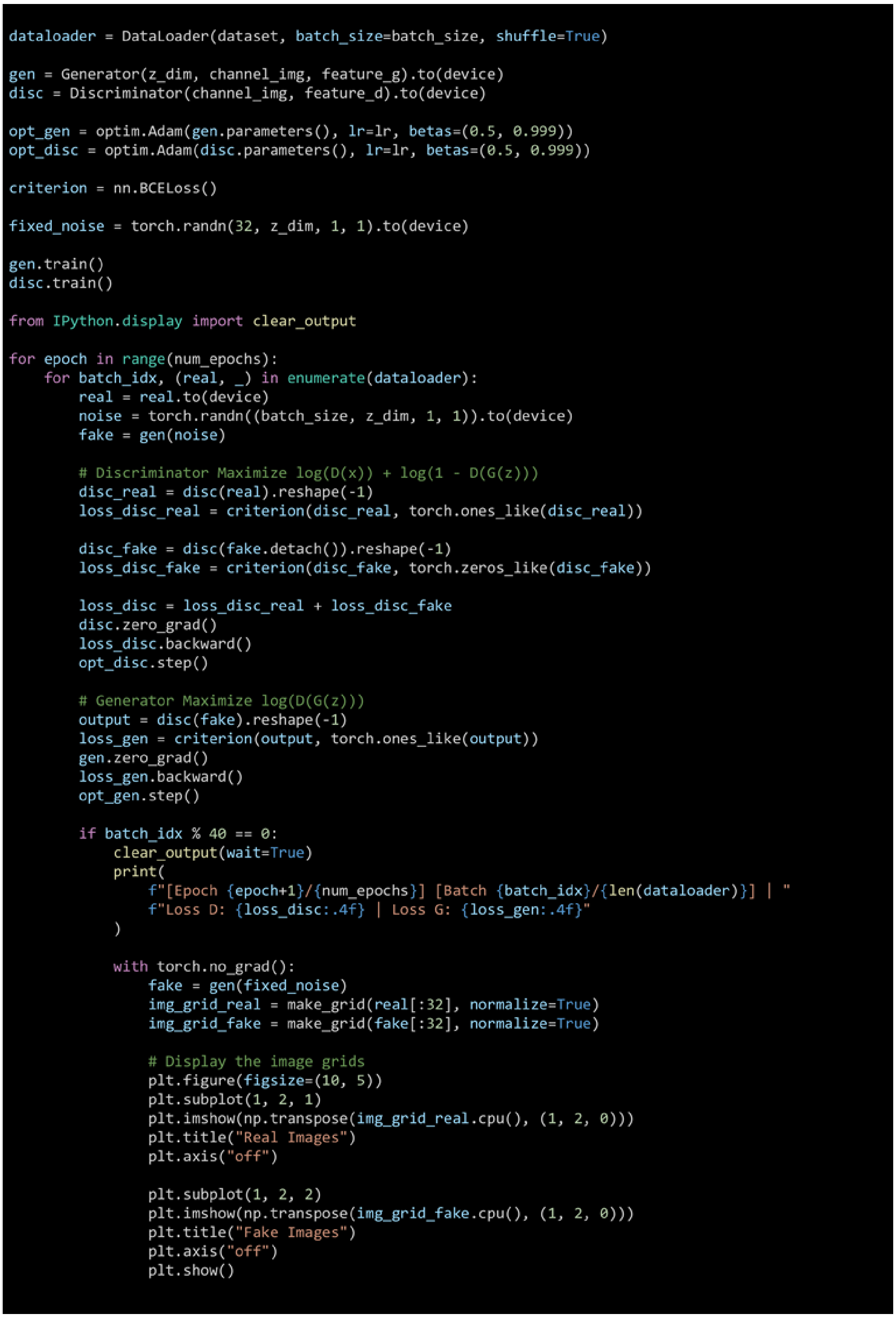
```python
class Discriminator(nn.Module):

    def __init__(self, channel_img, feature_d):
        super(Discriminator, self).__init__()
        self.disc = nn.Sequential(
            nn.Conv2d(channel_img, feature_d, kernel_size=5, stride=2, padding=1),
            nn.LeakyReLU(0.2, inplace=True),
            self._block(feature_d, feature_d*2, 4, 2, 1),
            self._block(feature_d*2, feature_d*4, 4, 2, 1),
            self._block(feature_d*4, feature_d*8, 3, 2, 1),
            nn.Conv2d(feature_d*8, 1, kernel_size=3, stride=2, padding=0),
            nn.Sigmoid(),
        )

    def _block(self, in_channels, out_channels, kernel_size, stride, padding):
        return nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size, stride, padding, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.LeakyReLU(0.2, inplace=True)
        )

    def forward(self, x):
        return self.disc(x)
```

Figure 4. Discriminator code implementation.



```python
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)

gen = Generator(z_dim, channel_img, feature_g).to(device)
disc = Discriminator(channel_img, feature_d).to(device)

opt_gen = optim.Adam(gen.parameters(), lr=lr, betas=(0.5, 0.999))
opt_disc = optim.Adam(disc.parameters(), lr=lr, betas=(0.5, 0.999))

criterion = nn.BCELoss()

fixed_noise = torch.randn(32, z_dim, 1, 1).to(device)

gen.train()
disc.train()

from IPython.display import clear_output

for epoch in range(num_epochs):
    for batch_idx, (real, _) in enumerate(dataloader):
        real = real.to(device)
        noise = torch.randn((batch_size, z_dim, 1, 1)).to(device)
        fake = gen(noise)

        # Discriminator Maximize log(D(x)) + log(1 - D(G(z)))
        disc_real = disc(real).reshape(-1)
        loss_disc_real = criterion(disc_real, torch.ones_like(disc_real))

        disc_fake = disc(fake.detach()).reshape(-1)
        loss_disc_fake = criterion(disc_fake, torch.zeros_like(disc_fake))

        loss_disc = loss_disc_real + loss_disc_fake
        disc.zero_grad()
        loss_disc.backward()
        opt_disc.step()

        # Generator Maximize log(D(G(z)))
        output = disc(fake).reshape(-1)
        loss_gen = criterion(output, torch.ones_like(output))
        gen.zero_grad()
        loss_gen.backward()
        opt_gen.step()

        if batch_idx % 40 == 0:
            clear_output(wait=True)
            print(
                f"[Epoch {epoch+1}/{num_epochs}] [Batch {batch_idx}/{len(dataloader)}] | "
                f"Loss D: {loss_disc:.4f} | Loss G: {loss_gen:.4f}"
            )

            with torch.no_grad():
                fake = gen(fixed_noise)
                img_grid_real = make_grid(real[:32], normalize=True)
                img_grid_fake = make_grid(fake[:32], normalize=True)

                # Display the image grids
                plt.figure(figsize=(10, 5))
                plt.subplot(1, 2, 1)
                plt.imshow(np.transpose(img_grid_real.cpu(), (1, 2, 0)))
                plt.title("Real Images")
                plt.axis("off")

                plt.subplot(1, 2, 2)
                plt.imshow(np.transpose(img_grid_fake.cpu(), (1, 2, 0)))
                plt.title("Fake Images")
                plt.axis("off")
                plt.show()
```

Figure 5. Model training implementation.

5. Training strategy and objective functions

Train using BCEloss which is the standard loss function for the original GAN. The training use minimax as an objective switching between maximum of the discriminator log(D(x)) + log(1 - D(G(z))) and maximum of the generator log(D(G(z)) and the dataset is from stanford dogs dataset which developed by Stanford University (based on ImageNet data) contains the images of 120 different dog breeds from around the world. The image type is a real-world photograph of dogs in various poses, lighting conditions, and backgrounds.

Link : http://vision.stanford.edu/aditya86/ImageNetDogs/

6. Evaluation metrics and assessment methods

No direct loss or accuracy can be used to evaluate a GAN model because it operates in an unsupervised learning environment, where the generator and discriminator continuously correct each other’s error. As a result, traditional metrics such as accuracy, precision, or recall are not directly applicable.

Instead, modern evaluation methods such as the Fréchet inception distance (FID) and Precision-Recall for Distribution (PRD) are used. FID measures how close the generated images are to real ones, while PRD evaluates both the quality and diversity of generated samples.

7. References

[1] Radford, A., Metz, L., & Chintala, S. (2016). *Unsupervised representation learning with deep convolutional generative adversarial networks*. arXiv preprint arXiv:1511.06434. https://arxiv.org/abs/1511.06434

[2] Google. (3 November 2025). *Building PyTorch GAN Components*. [Generative AI chat]. Gemini 2.5 Pro. https://gemini.google.com/share/087d06d2eae9

[3] Roy, R. (2025). *Generative Adversarial Network (GAN)*. [Online] Available: https://www.geeksforgeeks.org/deep-learning/generative-adversarial-network-gan/

8. Team

1. Kongphop Tirattanaprakom 6610505250 (65%)
    - Models based on Unsupervised deep convolutional generative adversarial networks paper.
    - Finding and training model with Stanford dogs dataset.
    - Tuned hyperparameters to improve image quality and training stability.
2. Kasidit Boonnum 6610501980 (35%)
    - Wrote and edited sections of the reports.
    - Create visual diagrams for presentation.