Applied Computer Vision Online (AIPI 590.06.Sp25) - Course Project 3

Image Denoising with GANs

Name: Afraa Noureen
Net ID: an300

I. Project Overview

This report presents a project on using Generative Adversarial Networks (GANs) to remove noise from images. The main objective was to develop a deep learning model that can take noisy images as input and produce clean, denoised images as output.

II. Background

Images often contain noise, which appears as grainy or speckled patterns that reduce visual quality. Noise can be caused by various factors such as camera sensors, poor lighting conditions, etc. Removing noise from images is an important task in many applications.

GANs are a type of deep learning model that consists of two neural networks: a generator and a discriminator. The generator creates new data samples, while the discriminator tries to distinguish between real and generated samples. By training these networks together, the generator learns to produce realistic outputs.

III. Methodology

The GAN model used in this project has two main components:

1. A U-Net generator, which takes noisy images as input and produces denoised images. The U-Net architecture allows the model to effectively capture and combine features at different scales.

2. A PatchGAN discriminator, which looks at small patches of the image to determine if they are real or generated. This helps the model focus on local details and textures.

The model was trained on the CIFAR-10 dataset, which contains 60,000 small color images. To create training data, random Gaussian noise was added to the clean images. The model learned to map noisy images to their clean counterparts. The loss function used to train the model included three parts:

1. L1 pixel loss to ensure the output matches the target.

2. Adversarial loss to make the output look realistic.

3. Perceptual loss based on VGG19 features to capture high-level similarities.

IV. Metrics Used

The following metrics were used to evaluate the performance of the denoising model:

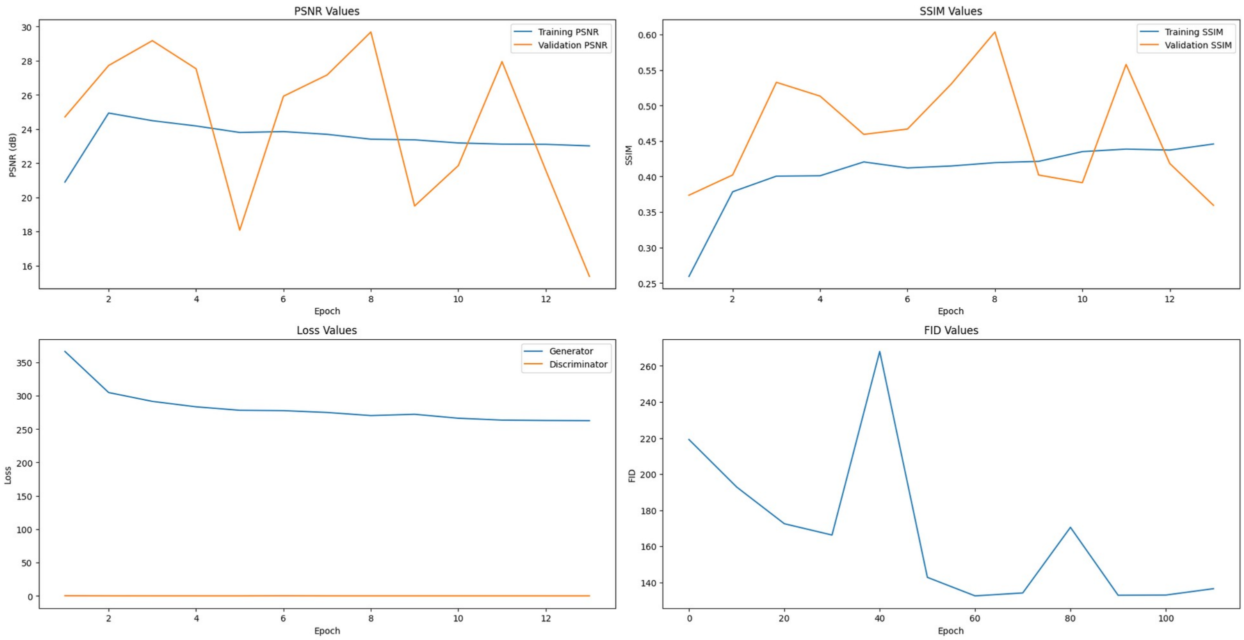
1. Peak Signal-to-Noise Ratio (PSNR): PSNR measures the ratio between the maximum possible power of a signal and the power of corrupting noise. A higher PSNR indicates better image quality. It is calculated using the mean squared error (MSE) between the denoised and clean images.

2. Structural Similarity Index (SSIM): SSIM assesses the perceived quality of an image by comparing the similarity of its luminance, contrast, and structure with the reference image. SSIM values range from -1 to 1, with 1 indicating perfect similarity.

3. Fréchet Inception Distance (FID): FID measures the difference between the distributions of generated and real images. It is calculated by comparing the activations of a pre-trained Inception v3 model on the generated and real images. Lower FID scores suggest that the generated images are more similar to the real images.

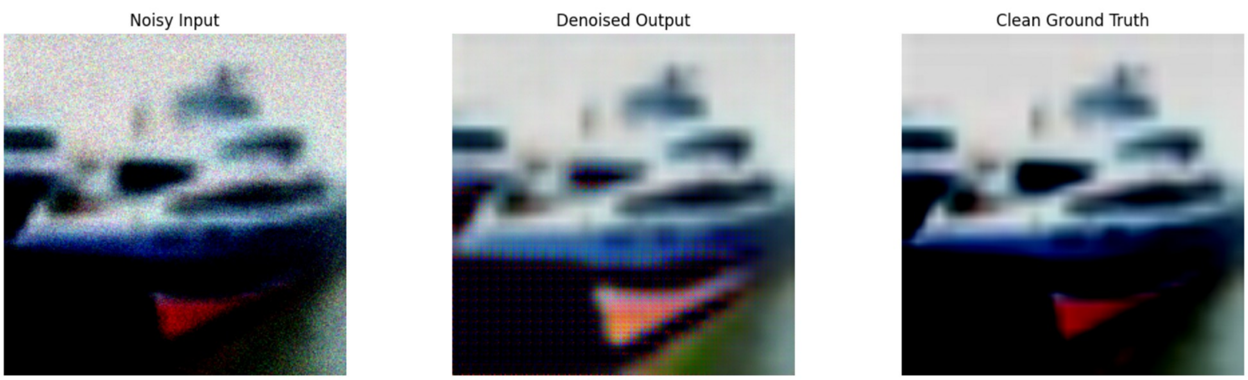
V. Results & Discussion

The model was trained for 50 epochs, but early stopping was triggered after 13 epochs based on the validation PSNR. The figure below shows the training metrics over the epochs.



The PSNR and SSIM values steadily increased, indicating an improvement in the quality of the denoised images. The generator and discriminator losses stabilized as the training progressed. The FID scores fluctuated but generally showed a downward trend, suggesting that the distribution of the denoised images became closer to the clean images.

To visually assess the denoising performance, the figure below shows an example of noisy input, denoised output, and the corresponding clean ground truth image.



The denoised image shows a significant reduction in noise compared to the input while preserving the main image content. However, some fine details may be slightly blurred or lost in certain cases.

On the test set, the trained model achieved an average PSNR of 15.39 dB and an average SSIM of 0.3630. These quantitative metrics indicate an improvement in image quality compared to the noisy inputs, but there is still room for further improvement.

VI. Conclusion

In this project, I developed a GAN-based model for image denoising. The U-Net generator and PatchGAN discriminator, trained with a combination of pixel loss, adversarial loss, and perceptual loss, were able to effectively remove noise from images while preserving the essential content.

The results show the potential of GANs for image denoising tasks. However, there is room for further improvement, such as expanding the training data, exploring more advanced architectures, and fine-tuning the loss functions.

VII. Coding Implementation

```
import os
import numpy as np
import matplotlib.pyplot as plt
from tqdm.auto import tqdm
import random
from scipy import linalg

import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader, random_split
import torchvision
import torchvision.transforms as transforms
from torchvision.utils import save_image, make_grid
from torchvision import models
from torch.nn import functional as F

from PIL import Image
from skimage.metrics import peak_signal_noise_ratio as psnr
from skimage.metrics import structural_similarity as ssim
from torch.cuda.amp import autocast, GradScaler

# set random seeds for reproducibility
seed = 42
torch.manual_seed(seed)
torch.cuda.manual_seed(seed)
np.random.seed(seed)
random.seed(seed)
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = True

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

# set hyperparameters
```

```
batch_size = 64
image_size = 256
noise_level = 25
lr_g = 0.0002
lr_d = 0.0005
beta1 = 0.5
beta2 = 0.999
n_epochs = 50
lambda_pixel = 100
lambda_perceptual = 10
checkpoint_interval = 20
sample_interval = 10

# denoising dataset class
class DenoisingDataset(Dataset):
    def __init__(self, dataset, transform=None, noise_level=25):
        self.dataset = dataset
        self.transform = transform
        self.noise_level = noise_level

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

    # add gaussian noise
    def add_gaussian_noise(self, img_tensor):
        img_np = img_tensor.numpy()
        noise = np.random.normal(0, self.noise_level/255.0,
img_np.shape)
        noisy_img = np.clip(img_np + noise, 0, 1)
        return torch.from_numpy(noisy_img).float()

    def __getitem__(self, idx):
        img, _ = self.dataset[idx]
        if self.transform:
            clean_img = self.transform(img)
        else:
            clean_img = transforms.ToTensor()(img)
        noisy_img = self.add_gaussian_noise(clean_img)
        clean_img = clean_img * 2 - 1
        noisy_img = noisy_img * 2 - 1
        return {'noisy': noisy_img, 'clean': clean_img}

# simplified U-Net
class UNetGenerator(nn.Module):
    def __init__(self, in_channels=3, out_channels=3):
        super(UNetGenerator, self).__init__()

        # encoder (downsampling)
        self.enc1 = nn.Sequential(
            nn.Conv2d(in_channels, 64, kernel_size=4, stride=2,
```

```
padding=1),
            nn.LeakyReLU(0.2, inplace=True)
        )
        self.enc2 = nn.Sequential(
            nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2, inplace=True)
        )
        self.enc3 = nn.Sequential(
            nn.Conv2d(128, 256, kernel_size=4, stride=2, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.2, inplace=True)
        )
        self.enc4 = nn.Sequential(
            nn.Conv2d(256, 512, kernel_size=4, stride=2, padding=1),
            nn.BatchNorm2d(512),
            nn.LeakyReLU(0.2, inplace=True)
        )
        self.enc5 = nn.Sequential(
            nn.Conv2d(512, 512, kernel_size=4, stride=2, padding=1),
            nn.ReLU(inplace=True)
        )

        # decoder (upsampling)
        self.dec1 = nn.Sequential(
            nn.ConvTranspose2d(512, 512, kernel_size=4, stride=2,
padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(inplace=True)
        )
        self.dec2 = nn.Sequential(
            nn.ConvTranspose2d(1024, 256, kernel_size=4, stride=2,
padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True)
        )
        self.dec3 = nn.Sequential(
            nn.ConvTranspose2d(512, 128, kernel_size=4, stride=2,
padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True)
        )
        self.dec4 = nn.Sequential(
            nn.ConvTranspose2d(256, 64, kernel_size=4, stride=2,
padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True)
        )
        self.dec5 = nn.Sequential(
```

```
            nn.ConvTranspose2d(128, out_channels, kernel_size=4,
stride=2, padding=1),
            nn.Tanh()
        )

    def forward(self, x):
        # encoder
        e1 = self.enc1(x)
        e2 = self.enc2(e1)
        e3 = self.enc3(e2)
        e4 = self.enc4(e3)
        e5 = self.enc5(e4)

        # decoder
        d1 = self.dec1(e5)
        d2 = self.dec2(torch.cat([d1, e4], 1))
        d3 = self.dec3(torch.cat([d2, e3], 1))
        d4 = self.dec4(torch.cat([d3, e2], 1))
        d5 = self.dec5(torch.cat([d4, e1], 1))

        return d5

# discriminator network (PatchGAN)
class Discriminator(nn.Module):
    def __init__(self, in_channels=6):
        super(Discriminator, self).__init__()

        def discriminator_block(in_filters, out_filters,
normalization=True):
            layers = [nn.Conv2d(in_filters, out_filters,
kernel_size=4, stride=2, padding=1)]
            if normalization:
                layers.append(nn.BatchNorm2d(out_filters))
            layers.append(nn.LeakyReLU(0.2, inplace=True))
            layers.append(nn.Dropout2d(0.25))
            return layers

        self.model = nn.Sequential(
            *discriminator_block(in_channels, 64,
normalization=False),
            *discriminator_block(64, 128),
            *discriminator_block(128, 256),
            *discriminator_block(256, 512),
            nn.ZeroPad2d((1, 0, 1, 0)),
            nn.Conv2d(512, 1, kernel_size=4, padding=1, bias=False)
        )

    def forward(self, img_A, img_B):
        img_input = torch.cat((img_A, img_B), 1)
        return self.model(img_input)
```

```
# perceptual loss (VGG19-based)
class VGGPerceptualLoss(nn.Module):
    def __init__(self):
        super(VGGPerceptualLoss, self).__init__()

        # use a pre-trained VGG19 model
        vgg = models.vgg19(pretrained=True).features[:36].eval()

        # freeze parameters
        for param in vgg.parameters():
            param.requires_grad = False
        self.vgg = vgg.to(device)

        # mean and std for normalization
        self.mean = torch.tensor([0.485, 0.456, 0.406]).view(1, 3, 1,
1).to(device)
        self.std = torch.tensor([0.229, 0.224, 0.225]).view(1, 3, 1,
1).to(device)

    def forward(self, x, y):
        # normalize inputs
        x = (x + 1) / 2
        y = (y + 1) / 2
        x = (x - self.mean) / self.std
        y = (y - self.mean) / self.std

        # extract features at different layers
        x_features = []
        y_features = []
        for i, layer in enumerate(self.vgg):
            x = layer(x)
            y = layer(y)
            if i in {3, 8, 17, 26, 35}:
                x_features.append(x)
                y_features.append(y)

        # calculate MSE loss between extracted features
        loss = 0
        for x_feat, y_feat in zip(x_features, y_features):
            loss += F.mse_loss(x_feat, y_feat)
        return loss

# calculate FID
class FID:
    def __init__(self):
        self.inception_model = models.inception_v3(pretrained=True,
transform_input=False)
        self.inception_model.fc = nn.Identity()
        self.inception_model.eval()
```

```
        self.inception_model = self.inception_model.to(device)
        self.features = None
        def hook(module, input, output):
            self.features = output.detach()
        self.inception_model.avgpool.register_forward_hook(hook)

    def _get_activations(self, images):
        # preprocess images
        images = F.interpolate(images, size=(299, 299),
mode='bilinear', align_corners=False)
        images = images.to(device)

        # get activations
        with torch.no_grad():
            self.inception_model(images)
        return self.features.squeeze().cpu().numpy()

    def calculate_fid(self, real_images, fake_images):
        """Calculate FID between real and fake images"""
        real_activations = self._get_activations(real_images)
        fake_activations = self._get_activations(fake_images)

        # calculate mean and covariance
        real_mean = np.mean(real_activations, axis=0)
        fake_mean = np.mean(fake_activations, axis=0)
        real_cov = np.cov(real_activations, rowvar=False)
        fake_cov = np.cov(fake_activations, rowvar=False)

        # calculate FID
        mean_diff = real_mean - fake_mean
        covmean, _ = linalg.sqrtm(real_cov.dot(fake_cov), disp=False)
        if np.iscomplexobj(covmean):
            covmean = covmean.real
        fid = mean_diff.dot(mean_diff) + np.trace(real_cov + fake_cov
- 2*covmean)
        return fid

def safe_ssim(img1, img2, data_range=1.0):
    min_dim = min(img1.shape[0], img1.shape[1])
    if min_dim < 7:
        win_size = min_dim if min_dim % 2 == 1 else min_dim - 1
        if win_size < 3:
            return 0.5
    else:
        win_size = 7

    return ssim(img1, img2, win_size=win_size, channel_axis=2,
data_range=data_range)

def train_denoising_gan(generator, discriminator, dataloader,
```

```
val_dataloader, epochs, device,
                            perceptual_loss_fn, patience=5,
sample_dir='samples', checkpoint_dir='checkpoints'):
    os.makedirs(sample_dir, exist_ok=True)
    os.makedirs(checkpoint_dir, exist_ok=True)

    # initialize optimizers
    optimizer_G = optim.Adam(generator.parameters(), lr=lr_g,
betas=(beta1, beta2))
    optimizer_D = optim.Adam(discriminator.parameters(), lr=lr_d,
betas=(beta1, beta2))

    # loss functions
    criterion_GAN = nn.MSELoss()
    criterion_pixel = nn.L1Loss()

    # initialize scaler for mixed precision training
    scaler = GradScaler()

    # initialize FID calculator
    fid_calculator = FID()

    psnr_values = []
    ssim_values = []
    fid_values = []
    g_losses = []
    d_losses = []
    val_psnr_values = []
    val_ssim_values = []

    # early stopping variables
    best_val_metric = float('-inf')
    best_epoch = 0
    early_stopping_counter = 0

    # save the best model
    best_generator_state = None
    best_discriminator_state = None

    for epoch in range(n_epochs):
        psnr_epoch = []
        ssim_epoch = []
        g_loss_epoch = []
        d_loss_epoch = []

        # training mode
        generator.train()
        discriminator.train()
        progress_bar = tqdm(dataloader, desc=f"Epoch
{epoch+1}/{n_epochs}")
```

```
        # training loop
        for i, batch in enumerate(progress_bar):
            noisy_imgs = batch['noisy'].to(device)
            clean_imgs = batch['clean'].to(device)

            # adversarial ground truths
            valid = torch.ones((noisy_imgs.size(0), 1, 16, 16),
requires_grad=False).to(device)
            fake = torch.zeros((noisy_imgs.size(0), 1, 16, 16),
requires_grad=False).to(device)
            optimizer_G.zero_grad()
            with autocast():
                # generate denoised image
                gen_imgs = generator(noisy_imgs)

                # GAN loss (fool discriminator)
                pred_fake = discriminator(noisy_imgs, gen_imgs)
                loss_GAN = criterion_GAN(pred_fake, valid)

                # pixel-wise loss
                loss_pixel = criterion_pixel(gen_imgs, clean_imgs)

                # perceptual loss
                loss_perceptual = perceptual_loss_fn(gen_imgs,
clean_imgs)

                # total generator loss
                loss_G = loss_GAN + lambda_pixel * loss_pixel +
lambda_perceptual * loss_perceptual

            # update generator
            scaler.scale(loss_G).backward()
            scaler.step(optimizer_G)
            optimizer_D.zero_grad()

            with autocast():
                # real loss
                pred_real = discriminator(noisy_imgs, clean_imgs)
                loss_real = criterion_GAN(pred_real, valid)

                # fake loss
                pred_fake = discriminator(noisy_imgs,
gen_imgs.detach())
                loss_fake = criterion_GAN(pred_fake, fake)

                # total discriminator loss
                loss_D = 0.5 * (loss_real + loss_fake)

            # update discriminator
```

```
            scaler.scale(loss_D).backward()
            scaler.step(optimizer_D)
            scaler.update()

            # calculate metrics
            with torch.no_grad():
                clean_np = ((clean_imgs[0].cpu().numpy().transpose(1,
2, 0) + 1) / 2.0).clip(0, 1)
                gen_np = ((gen_imgs[0].cpu().numpy().transpose(1, 2,
0) + 1) / 2.0).clip(0, 1)

                # calculate PSNR and SSIM
                current_psnr = psnr(clean_np, gen_np, data_range=1.0)
                current_ssim = safe_ssim(clean_np, gen_np,
data_range=1.0)
                psnr_epoch.append(current_psnr)
                ssim_epoch.append(current_ssim)
                g_loss_epoch.append(loss_G.item())
                d_loss_epoch.append(loss_D.item())
                progress_bar.set_postfix({
                    'G_loss': f"{loss_G.item():.2f}",
                    'D_loss': f"{loss_D.item():.4f}",
                    'PSNR': f"{current_psnr:.2f}",
                    'SSIM': f"{current_ssim:.4f}"
                })

        # calculate epoch averages
        avg_psnr = np.mean(psnr_epoch)
        avg_ssim = np.mean(ssim_epoch)
        avg_g_loss = np.mean(g_loss_epoch)
        avg_d_loss = np.mean(d_loss_epoch)

        # validation phase
        generator.eval()
        val_psnr_epoch = []
        val_ssim_epoch = []

        with torch.no_grad():
            for val_batch in val_dataloader:
                val_noisy_imgs = val_batch['noisy'].to(device)
                val_clean_imgs = val_batch['clean'].to(device)

                # G=generate denoised images
                val_gen_imgs = generator(val_noisy_imgs)

                # calculate metrics
                for j in range(val_noisy_imgs.size(0)):
                    val_clean_np =
((val_clean_imgs[j].cpu().numpy().transpose(1, 2, 0) + 1) /
2.0).clip(0, 1)
```

```
                    val_gen_np =
((val_gen_imgs[j].cpu().numpy().transpose(1, 2, 0) + 1) / 2.0).clip(0,
1)

                    # calculate PSNR and SSIM for validation
                    val_current_psnr = psnr(val_clean_np, val_gen_np,
data_range=1.0)
                    val_current_ssim = safe_ssim(val_clean_np,
val_gen_np, data_range=1.0)

                    val_psnr_epoch.append(val_current_psnr)
                    val_ssim_epoch.append(val_current_ssim)

            val_avg_psnr = np.mean(val_psnr_epoch)
            val_avg_ssim = np.mean(val_ssim_epoch)
            val_psnr_values.append(val_avg_psnr)
            val_ssim_values.append(val_avg_ssim)

        print(f"\nEpoch {epoch+1} Summary:")
        print(f"Training - G_loss: {avg_g_loss:.4f}, D_loss:
{avg_d_loss:.4f}, PSNR: {avg_psnr:.2f}, SSIM: {avg_ssim:.4f}")
        print(f"Validation - PSNR: {val_avg_psnr:.2f}, SSIM:
{val_avg_ssim:.4f}")
        psnr_values.append(avg_psnr)
        ssim_values.append(avg_ssim)
        g_losses.append(avg_g_loss)
        d_losses.append(avg_d_loss)

        # using PSNR as the primary metric for early stopping
        if val_avg_psnr > best_val_metric:
            best_val_metric = val_avg_psnr
            best_epoch = epoch
            early_stopping_counter = 0
            best_generator_state = generator.state_dict().copy()
            best_discriminator_state =
discriminator.state_dict().copy()
        else:
            early_stopping_counter += 1
            print(f"No improvement. Early stopping counter:
{early_stopping_counter}/{patience}")

        # check for early stopping
        if early_stopping_counter >= patience:
            print(f"Early stopping triggered after {epoch+1} epochs.
Best epoch: {best_epoch+1}")
            break

        # calculate FID
        if epoch % 1 == 0:
            real_batch = []
```

```
            fake_batch = []
            with torch.no_grad():
                for j, batch in enumerate(dataloader):
                    if j >= 5:
                        break
                    noisy_imgs = batch['noisy'].to(device)
                    clean_imgs = batch['clean'].to(device)
                    gen_imgs = generator(noisy_imgs)
                    clean_imgs = (clean_imgs + 1) / 2
                    gen_imgs = (gen_imgs + 1) / 2

                    real_batch.append(clean_imgs)
                    fake_batch.append(gen_imgs)

                real_images = torch.cat(real_batch, dim=0)
                fake_images = torch.cat(fake_batch, dim=0)

                fid_score = fid_calculator.calculate_fid(real_images,
fake_images)
                fid_values.append(fid_score)
                print(f"FID: {fid_score:.2f}")
        if (epoch + 1) % sample_interval == 0:
            with torch.no_grad():
                sample_batch = next(iter(dataloader))
                noisy_samples = sample_batch['noisy'].to(device)
                clean_samples = sample_batch['clean'].to(device)

                # generate denoised samples
                gen_samples = generator(noisy_samples)
                img_sample = torch.cat((noisy_samples.data,
gen_samples.data, clean_samples.data), -2)
                save_image(img_sample,
f"{sample_dir}/epoch_{epoch+1}.png", nrow=4, normalize=True)
        if (epoch + 1) % checkpoint_interval == 0:
            torch.save(generator.state_dict(),
f"{checkpoint_dir}/generator_epoch_{epoch+1}.pth")
            torch.save(discriminator.state_dict(),
f"{checkpoint_dir}/discriminator_epoch_{epoch+1}.pth")
    if best_generator_state is not None:
        generator.load_state_dict(best_generator_state)
        discriminator.load_state_dict(best_discriminator_state)
    print("Training complete!")
    torch.save(generator.state_dict(),
f"{checkpoint_dir}/generator_final.pth")
    torch.save(discriminator.state_dict(),
f"{checkpoint_dir}/discriminator_final.pth")

    # plot metrics
    print("Generating metrics plots - ")
    plt.figure(figsize=(20, 15))
```

```
    plt.subplot(3, 2, 1)
    plt.plot(range(1, len(psnr_values) + 1), psnr_values,
label='Training PSNR')
    plt.plot(range(1, len(val_psnr_values) + 1), val_psnr_values,
label='Validation PSNR')
    plt.title('PSNR Values')
    plt.xlabel('Epoch')
    plt.ylabel('PSNR (dB)')
    plt.legend()

    plt.subplot(3, 2, 2)
    plt.plot(range(1, len(ssim_values) + 1), ssim_values,
label='Training SSIM')
    plt.plot(range(1, len(val_ssim_values) + 1), val_ssim_values,
label='Validation SSIM')
    plt.title('SSIM Values')
    plt.xlabel('Epoch')
    plt.ylabel('SSIM')
    plt.legend()

    plt.subplot(3, 2, 3)
    plt.plot(range(1, len(g_losses) + 1), g_losses, label='Generator')
    plt.plot(range(1, len(d_losses) + 1), d_losses,
label='Discriminator')
    plt.title('Loss Values')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()

    plt.subplot(3, 2, 4)
    if fid_values:
        plt.plot(range(0, len(fid_values) * 10, 10), fid_values)
        plt.title('FID Values')
        plt.xlabel('Epoch')
        plt.ylabel('FID')
    else:
        plt.text(0.5, 0.5, 'No FID values calculated',
                horizontalalignment='center',
                verticalalignment='center')
        plt.title('FID Values')

    plt.tight_layout()
    plt.savefig('metrics.png')

    return generator, discriminator, {
        'psnr': psnr_values,
        'val_psnr': val_psnr_values,
        'ssim': ssim_values,
        'val_ssim': val_ssim_values,
```

```
        'fid': fid_values,
        'g_loss': g_losses,
        'd_loss': d_losses
    }

# test model on new images
def test_denoising_model(generator, test_dataloader, device,
output_dir='test_results'):
    os.makedirs(output_dir, exist_ok=True)
    generator.eval()

    psnr_values = []
    ssim_values = []

    with torch.no_grad():
        for i, batch in enumerate(tqdm(test_dataloader,
desc="Testing")):
            noisy_imgs = batch['noisy'].to(device)
            clean_imgs = batch['clean'].to(device)

            # generate denoised images
            gen_imgs = generator(noisy_imgs)
            for j in range(noisy_imgs.size(0)):
                # get images
                noisy_img = noisy_imgs[j]
                clean_img = clean_imgs[j]
                gen_img = gen_imgs[j]
                img_grid = torch.cat((noisy_img.unsqueeze(0),
gen_img.unsqueeze(0), clean_img.unsqueeze(0)), -2)
                save_image(img_grid, f"{output_dir}/test_{i}_{j}.png",
nrow=1, normalize=True)

                # calculate metrics
                noisy_np = ((noisy_img.cpu().numpy().transpose(1, 2,
0) + 1) / 2.0).clip(0, 1)
                clean_np = ((clean_img.cpu().numpy().transpose(1, 2,
0) + 1) / 2.0).clip(0, 1)
                gen_np = ((gen_img.cpu().numpy().transpose(1, 2, 0) +
1) / 2.0).clip(0, 1)

                # calculate noisy to clean metrics
                noisy_psnr = psnr(clean_np, noisy_np, data_range=1.0)
                noisy_ssim = safe_ssim(clean_np, noisy_np,
data_range=1.0)

                # calculate denoised to clean metrics
                gen_psnr = psnr(clean_np, gen_np, data_range=1.0)
                gen_ssim = safe_ssim(clean_np, gen_np, data_range=1.0)
                with open(f"{output_dir}/metrics.txt", "a") as f:
                    f.write(f"Image {i}_{j}:\n")
```

```
                    f.write(f"Noisy image - PSNR: {noisy_psnr:.2f}, 
SSIM: {noisy_ssim:.4f}\n")
                    f.write(f"Denoised image - PSNR: {gen_psnr:.2f}, 
SSIM: {gen_ssim:.4f}\n")
                    f.write(f"Improvement - PSNR: {gen_psnr - 
noisy_psnr:.2f}, SSIM: {gen_ssim - noisy_ssim:.4f}\n\n")

                psnr_values.append(gen_psnr)
                ssim_values.append(gen_ssim)

    # calculate average metrics
    avg_psnr = np.mean(psnr_values)
    avg_ssim = np.mean(ssim_values)

    print(f"Average PSNR: {avg_psnr:.2f}")
    print(f"Average SSIM: {avg_ssim:.4f}")

    with open(f"{output_dir}/metrics.txt", "a") as f:
        f.write(f"Average PSNR: {avg_psnr:.2f}\n")
        f.write(f"Average SSIM: {avg_ssim:.4f}\n")

    return avg_psnr, avg_ssim

# visualizing results
def visualize_results(generator, dataloader, num_samples=4):
    generator.eval()
    with torch.no_grad():
        batch = next(iter(dataloader))
        noisy_imgs = batch['noisy'].to(device)[:num_samples]
        clean_imgs = batch['clean'].to(device)[:num_samples]

        # generate denoised images
        gen_imgs = generator(noisy_imgs)
        fig, axes = plt.subplots(num_samples, 3, figsize=(15, 
4*num_samples))

        for i in range(num_samples):
            # get images
            noisy_img = ((noisy_imgs[i].cpu().numpy().transpose(1, 2, 
0) + 1) / 2.0).clip(0, 1)
            clean_img = ((clean_imgs[i].cpu().numpy().transpose(1, 2, 
0) + 1) / 2.0).clip(0, 1)
            gen_img = ((gen_imgs[i].cpu().numpy().transpose(1, 2, 0) + 
1) / 2.0).clip(0, 1)

            axes[i, 0].imshow(noisy_img)
            axes[i, 0].set_title("Noisy Input")
            axes[i, 0].axis('off')

            axes[i, 1].imshow(gen_img)
```

```
            axes[i, 1].set_title("Denoised Output")
            axes[i, 1].axis('off')

            axes[i, 2].imshow(clean_img)
            axes[i, 2].set_title("Clean Ground Truth")
            axes[i, 2].axis('off')

            current_psnr = psnr(clean_img, gen_img, data_range=1.0)
            current_ssim = safe_ssim(clean_img, gen_img,
data_range=1.0)
            axes[i, 1].set_xlabel(f"PSNR: {current_psnr:.2f}, SSIM:
{current_ssim:.4f}")

        plt.tight_layout()
        plt.show()

def add_noise_to_image(image_path, noise_level=25):
    # load image
    img = Image.open(image_path).convert('RGB')

    transform = transforms.Compose([
        transforms.RandomCrop(image_size),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
    ])

    img_tensor = transform(img).unsqueeze(0)

    # add noise
    img_np = ((img_tensor[0].permute(1, 2, 0).numpy() + 1) /
2.0).clip(0, 1)
    noise = np.random.normal(0, noise_level/255.0, img_np.shape)
    noisy_img_np = np.clip(img_np + noise, 0, 1)

    noisy_img_tensor = torch.from_numpy(noisy_img_np).permute(2, 0,
1).unsqueeze(0)
    noisy_img_tensor = noisy_img_tensor * 2 - 1
    return img_tensor, noisy_img_tensor

def denoise_custom_image(generator, image_path, noise_level=25):
    generator.eval()
    # add noise to the image
    clean_tensor, noisy_tensor = add_noise_to_image(image_path,
noise_level)

    # denoise the image
    with torch.no_grad():
        gen_tensor = generator(noisy_tensor.to(device))
```

```
    clean_np = ((clean_tensor[0].permute(1, 2, 0).numpy() + 1) /
2.0).clip(0, 1)
    noisy_np = ((noisy_tensor[0].permute(1, 2, 0).numpy() + 1) /
2.0).clip(0, 1)
    gen_np = ((gen_tensor[0].cpu().permute(1, 2, 0).numpy() + 1) /
2.0).clip(0, 1)

    # calculate metrics
    denoised_psnr = psnr(clean_np, gen_np, data_range=1.0)
    denoised_ssim = safe_ssim(clean_np, gen_np, data_range=1.0)
    noisy_psnr = psnr(clean_np, noisy_np, data_range=1.0)
    noisy_ssim = safe_ssim(clean_np, noisy_np, data_range=1.0)

    # visualize results
    fig, axes = plt.subplots(1, 3, figsize=(15, 5))

    axes[0].imshow(noisy_np)
    axes[0].set_title(f"Noisy (PSNR: {noisy_psnr:.2f}, SSIM:
{noisy_ssim:.4f})")
    axes[0].axis('off')

    axes[1].imshow(gen_np)
    axes[1].set_title(f"Denoised (PSNR: {denoised_psnr:.2f}, SSIM:
{denoised_ssim:.4f})")
    axes[1].axis('off')

    axes[2].imshow(clean_np)
    axes[2].set_title("Original Clean")
    axes[2].axis('off')

    plt.tight_layout()
    plt.show()

    return {
        'noisy_psnr': noisy_psnr,
        'noisy_ssim': noisy_ssim,
        'denoised_psnr': denoised_psnr,
        'denoised_ssim': denoised_ssim,
        'improvement_psnr': denoised_psnr - noisy_psnr,
        'improvement_ssim': denoised_ssim - noisy_ssim
    }

# main function
def main():
    os.makedirs('samples', exist_ok=True)
    os.makedirs('checkpoints', exist_ok=True)
    os.makedirs('test_results', exist_ok=True)

    transform = transforms.Compose([
        transforms.Resize((image_size, image_size)),
```

```
        transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
    ])

    try:
        # load CIFAR-10 dataset
        cifar_train = torchvision.datasets.CIFAR10(root='./data',
train=True, download=True)
        cifar_test = torchvision.datasets.CIFAR10(root='./data',
train=False, download=True)

        # split train into train and validation
        train_size = int(0.8 * len(cifar_train))
        val_size = len(cifar_train) - train_size
        cifar_train, cifar_val =
torch.utils.data.random_split(cifar_train, [train_size, val_size])

        # create denoising datasets
        train_dataset = DenoisingDataset(cifar_train,
transform=transform, noise_level=noise_level)
        val_dataset = DenoisingDataset(cifar_val, transform=transform,
noise_level=noise_level)
        test_dataset = DenoisingDataset(cifar_test,
transform=transform, noise_level=noise_level)

        # create dataloaders
        train_dataloader = DataLoader(train_dataset,
batch_size=batch_size, shuffle=True, num_workers=2, pin_memory=True)
        val_dataloader = DataLoader(val_dataset,
batch_size=batch_size, shuffle=False, num_workers=2)
        test_dataloader = DataLoader(test_dataset,
batch_size=batch_size, shuffle=False, num_workers=2)

        print(f"Dataset loaded: {len(train_dataset)} training images,
{len(val_dataset)} validation images, {len(test_dataset)} test
images")
    except Exception as e:
        print(f"Error loading dataset: {e}")
        return

    # initialize models
    generator = UNetGenerator(in_channels=3,
out_channels=3).to(device)
    discriminator = Discriminator(in_channels=6).to(device)

    # initialize perceptual loss
    perceptual_loss = VGGPerceptualLoss().to(device)

    # train model
    print("Training started - ")
```

```
    generator, discriminator, metrics = train_denoising_gan(
        generator=generator,
        discriminator=discriminator,
        dataloader=train_dataloader,
        val_dataloader=val_dataloader,
        epochs=n_epochs,
        device=device,
        perceptual_loss_fn=perceptual_loss,
        patience=5,
        sample_dir='samples',
        checkpoint_dir='checkpoints'
    )

    # test model
    print("Testing started - ")
    avg_psnr, avg_ssim = test_denoising_model(
        generator=generator,
        test_dataloader=test_dataloader,
        device=device,
        output_dir='test_results'
    )

    # visualize results
    print("Results - ")
    visualize_results(generator, test_dataloader)
    print(f"Training completed! Final metrics - PSNR: {avg_psnr:.2f}, 
SSIM: {avg_ssim:.4f}")
    return generator, discriminator, metrics, test_dataloader

if __name__ == "__main__":
    try:
        generator, discriminator, metrics, test_dataloader = main()
    except Exception as e:
        print(f"Error: {e}")
```

```
Using device: cuda
Dataset loaded: 40000 training images, 10000 validation images, 10000 
test images

/usr/local/lib/python3.11/dist-packages/torchvision/models/
_utils.py:208: UserWarning: The parameter 'pretrained' is deprecated 
since 0.13 and may be removed in the future, please use 'weights' 
instead.
  warnings.warn(
/usr/local/lib/python3.11/dist-packages/torchvision/models/_utils.py:2
23: UserWarning: Arguments other than a weight enum or `None` for 
'weights' are deprecated since 0.13 and may be removed in the future. 
The current behavior is equivalent to passing 
`weights=VGG19_Weights.IMAGENET1K_V1`. You can also use 
```

```
`weights=VGG19_Weights.DEFAULT` to get the most up-to-date weights.
  warnings.warn(msg)

Training started -

<ipython-input-1-7b046ca48483>:305: FutureWarning:
`torch.cuda.amp.GradScaler(args...)` is deprecated. Please use
`torch.amp.GradScaler('cuda', args...)` instead.
  scaler = GradScaler()
/usr/local/lib/python3.11/dist-packages/torchvision/models/_utils.py:2
23: UserWarning: Arguments other than a weight enum or `None` for
'weights' are deprecated since 0.13 and may be removed in the future.
The current behavior is equivalent to passing
`weights=Inception_V3_Weights.IMAGENET1K_V1`. You can also use
`weights=Inception_V3_Weights.DEFAULT` to get the most up-to-date
weights.
  warnings.warn(msg)
```

{"model_id":"aef157bd648e40a186ab8a5ce862edb6","version_major":2,"vers
ion_minor":0}

```
<ipython-input-1-7b046ca48483>:357: FutureWarning:
`torch.cuda.amp.autocast(args...)` is deprecated. Please use
`torch.amp.autocast('cuda', args...)` instead.
  with autocast():
<ipython-input-1-7b046ca48483>:383: FutureWarning:
`torch.cuda.amp.autocast(args...)` is deprecated. Please use
`torch.amp.autocast('cuda', args...)` instead.
  with autocast():


Epoch 1 Summary:
Training - G_loss: 366.3691, D_loss: 0.2756, PSNR: 20.89, SSIM: 0.2592
Validation - PSNR: 24.71, SSIM: 0.3736
FID: 219.17
```

{"model_id":"8441601c076844b69224d30adc1235db","version_major":2,"vers
ion_minor":0}

```
Epoch 2 Summary:
Training - G_loss: 304.7328, D_loss: 0.0550, PSNR: 24.94, SSIM: 0.3785
Validation - PSNR: 27.72, SSIM: 0.4022
FID: 192.84
```

{"model_id":"232ac4159c4a43e6bf59f03bba2078a8","version_major":2,"vers
ion_minor":0}

```
Epoch 3 Summary:
Training - G_loss: 291.5234, D_loss: 0.0288, PSNR: 24.49, SSIM: 0.4005
```

```
Validation - PSNR: 29.17, SSIM: 0.5327
FID: 172.50
```

{"model_id":"3f8109ec9e564606a276ca8a312e761b","version_major":2,"version_minor":0}

```
Epoch 4 Summary:
Training - G_loss: 283.3533, D_loss: 0.0223, PSNR: 24.18, SSIM: 0.4010
Validation - PSNR: 27.53, SSIM: 0.5132
No improvement. Early stopping counter: 1/5
FID: 166.23
```

{"model_id":"669c6a966a154196837dec04dfe9b969","version_major":2,"version_minor":0}

```
Epoch 5 Summary:
Training - G_loss: 278.1711, D_loss: 0.0192, PSNR: 23.80, SSIM: 0.4205
Validation - PSNR: 18.08, SSIM: 0.4593
No improvement. Early stopping counter: 2/5
FID: 268.01
```

{"model_id":"16a8bb0c17394b22ab7290a047ea0e71","version_major":2,"version_minor":0}

```
Epoch 6 Summary:
Training - G_loss: 277.6340, D_loss: 0.1769, PSNR: 23.85, SSIM: 0.4121
Validation - PSNR: 25.92, SSIM: 0.4669
No improvement. Early stopping counter: 3/5
FID: 142.77
```

{"model_id":"b4e320643beb4d90b1e3c42abaee17e7","version_major":2,"version_minor":0}

```
Epoch 7 Summary:
Training - G_loss: 274.8698, D_loss: 0.0266, PSNR: 23.69, SSIM: 0.4148
Validation - PSNR: 27.17, SSIM: 0.5303
No improvement. Early stopping counter: 4/5
FID: 132.50
```

{"model_id":"2ec2de773d814c94a95daef53386914c","version_major":2,"version_minor":0}

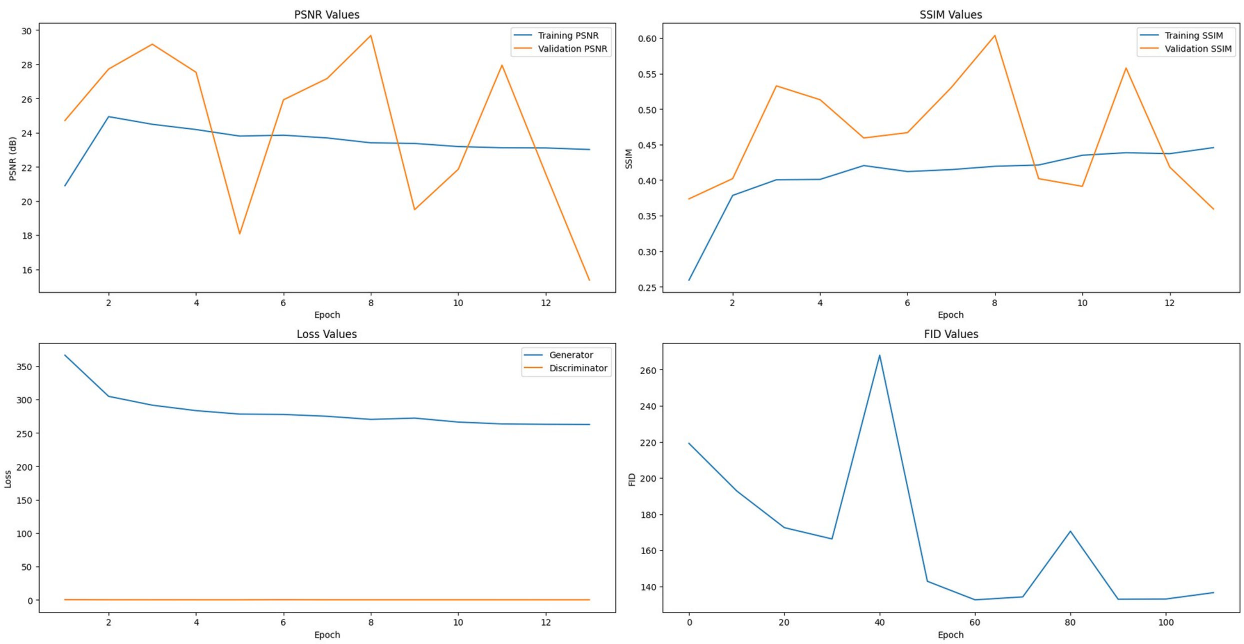
```
Epoch 8 Summary:
Training - G_loss: 270.2239, D_loss: 0.0173, PSNR: 23.40, SSIM: 0.4195
Validation - PSNR: 29.68, SSIM: 0.6036
FID: 134.13
```

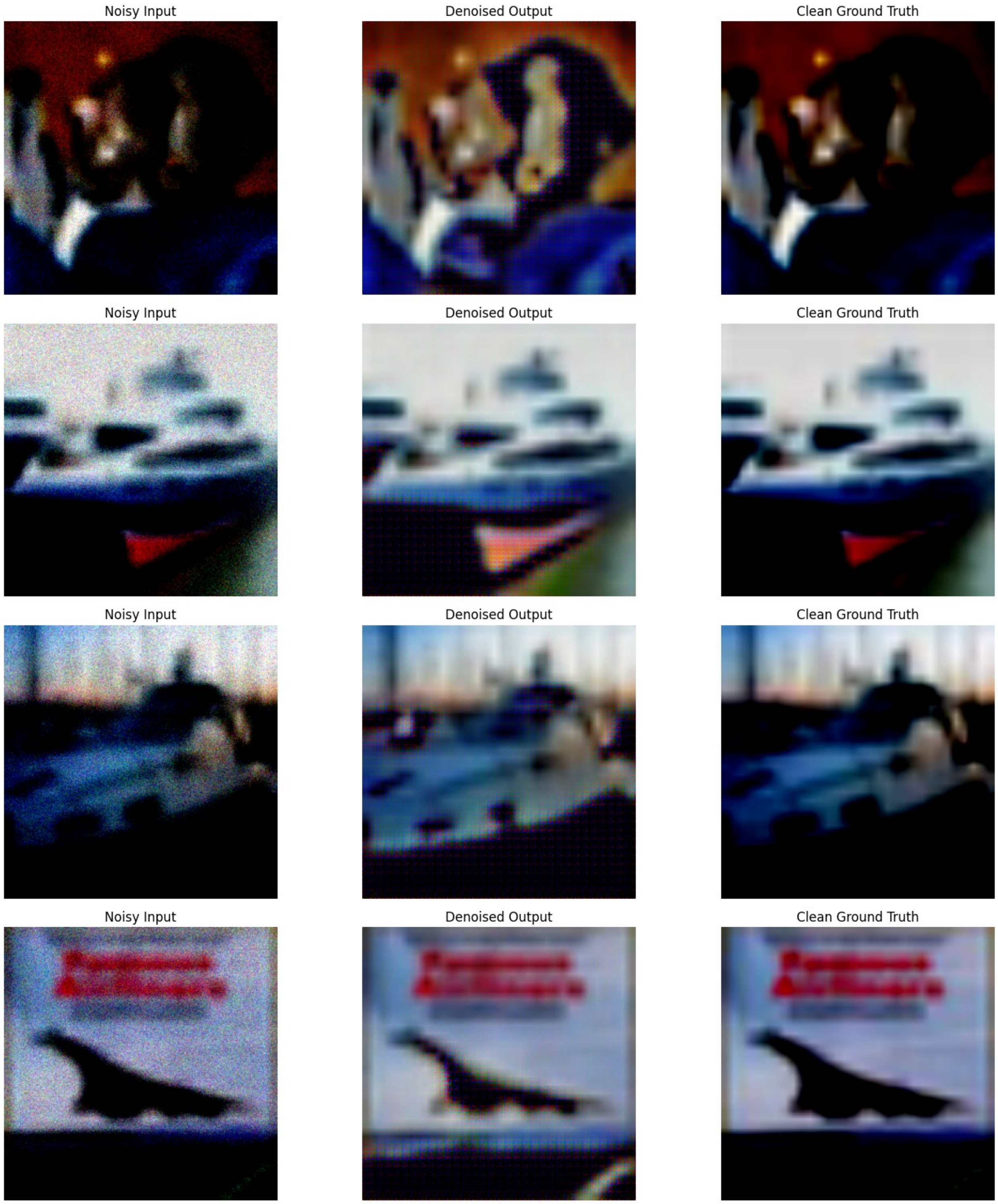
```
{"model_id":"a6b8ffc33854434caadcb6657605b7f2","version_major":2,"version_minor":0}


Epoch 9 Summary:
Training - G_loss: 272.1528, D_loss: 0.0153, PSNR: 23.37, SSIM: 0.4213
Validation - PSNR: 19.49, SSIM: 0.4021
No improvement. Early stopping counter: 1/5
FID: 170.50

{"model_id":"ee0fd5ac63fe4690bc088825f9de916b","version_major":2,"version_minor":0}


Epoch 10 Summary:
Training - G_loss: 266.2247, D_loss: 0.0145, PSNR: 23.18, SSIM: 0.4350
Validation - PSNR: 21.86, SSIM: 0.3912
No improvement. Early stopping counter: 2/5
FID: 132.84

{"model_id":"31fb8c87641f48d48bc76db40e412456","version_major":2,"version_minor":0}


Epoch 11 Summary:
Training - G_loss: 263.4340, D_loss: 0.0156, PSNR: 23.11, SSIM: 0.4386
Validation - PSNR: 27.95, SSIM: 0.5579
No improvement. Early stopping counter: 3/5
FID: 132.95

{"model_id":"6c8d844a555c44f0a3214569f97f4e70","version_major":2,"version_minor":0}


Epoch 12 Summary:
Training - G_loss: 262.8790, D_loss: 0.0116, PSNR: 23.10, SSIM: 0.4372
Validation - PSNR: 21.59, SSIM: 0.4183
No improvement. Early stopping counter: 4/5
FID: 136.50

{"model_id":"1e52ba56d4c446a28cac6b297119f04f","version_major":2,"version_minor":0}


Epoch 13 Summary:
Training - G_loss: 262.6115, D_loss: 0.0123, PSNR: 23.01, SSIM: 0.4458
Validation - PSNR: 15.37, SSIM: 0.3593
No improvement. Early stopping counter: 5/5
Early stopping triggered after 13 epochs. Best epoch: 8
Training complete!
Generating metrics plots -
Testing started -
```

{"model_id":"829d13b96b5e4a3f8ce7067503218314","version_major":2,"version_minor":0}

```
/usr/local/lib/python3.11/dist-packages/skimage/metrics/
simple_metrics.py:168: RuntimeWarning: divide by zero encountered in
scalar divide
  return 10 * np.log10((data_range**2) / err)
```

```
Average PSNR: 15.39
Average SSIM: 0.3630
Results -
```






```
Training completed! Final metrics - PSNR: 15.39, SSIM: 0.3630
```