F74091132_GAI_Project4

1.Theoretical Justification

Provide a clear and coherent explanation of the proposed solution, highlighting how it combines the strengths of DDPM and DIP.
Justify the design choices and assumptions made in the proposed approach.
Discuss the potential benefits and limitations of the proposed solution compared to using DDPM or DIP alone.

Ans:

1.Explanation

(1)Using Example 1, accelerate the training process of Denoising Diffusion Probabilistic Models (DDPMs) by incorporating Deep Image Prior (DIP) as an initial prior.
This approach leverages the strengths of both DDPM and DIP to enhance efficiency and performance.

(2)Steps

Training a DIP model on the target image for a short duration. DIP utilizes the architecture of Convolutional Neural Networks (CNNs) to automatically learn image-specific priors. During the training period, DIP model captures high-level structures and patterns in the image without overfitting to the noise.

Generating an Initial Prior: The trained DIP model generates an initial prior that represents the high-level features and patterns of the

target image. This prior serves as a more informative starting point for the DDPM training, rather than starting from pure noise.

Incorporating the DIP-based Prior into DDPM Training: Modify the DDPM training algorithm to utilize the DIP-generated initial prior. Use the output of the DIP model instead of initializing the DDPM model with random noise. This provides a head start for the DDPM, potentially reducing the number of diffusion steps required to achieve convergence.

Experiments and Optimization: Conduct experiments with different DIP training durations and CNN architectures to find the optimal balance between capturing meaningful image priors and maintaining computational efficiency. Investigate the impact of the DIP-based initialization on the quality and diversity of samples generated by the DDPM.

Evaluation: Compare the convergence speed and sample quality of DDPM models with and without the DIP-based prior. Use quantitative metrics (e.g., number of diffusion steps to reach a certain sample quality) and qualitative assessments (visual comparison of generated samples).

2. Justify the design choices and assumptions

(1)DIP as an Initial Prior: Using DIP to provide an initial prior leverages its ability to quickly learn high-level image structures without extensive training.

Assume that the initial structure learned by DIP can significantly reduce the workload of the DDPM by providing a better starting point, hence speeding up the diffusion process.

(2)Modification of DDPM Training: Modifying the DDPM training to start from a DIP-based prior rather than pure noise.

Assume that a more informed initialization will require fewer diffusion steps to produce high-quality samples, thereby accelerating the training process.

(3)Experimentation with DIP Configurations: Testing different training durations and architectures for DIP to find the optimal setup. Assume that there exists an optimal point where the DIP model captures enough information without becoming computationally expensive or overfitting.

3. Potential benefits and limitations

(1)Benefits

Accelerated Convergence: Starting from a DIP-based prior can reduce the number of diffusion steps needed, leading to faster convergence and reduced training time for DDPMs.

Enhanced Sample Quality: The informative initialization can lead to higher-quality samples early in the training process, as the model begins with a better approximation of the target image structure.

Stable Training Process: The stable nature of DIP training can help mitigate some of the training instabilities commonly associated with DDPMs.

(2)Limitations

Computational Overhead: Introducing DIP into the training pipeline adds an additional computational step. Although this is intended to reduce overall training time, it may introduce overhead that needs careful balancing.

Optimal Configuration Challenge: Finding the right DIP training duration and architecture requires experimentation. The balance between

capturing sufficient image priors and maintaining efficiency might vary across different datasets and applications.

Limited Generalization: While DIP works well for image-specific priors, its effectiveness for other types of data (e.g., text, audio) is less certain and may require further adaptation and experimentation.

2. Experimental Verification

Implement the proposed solution and conduct experiments to validate its effectiveness.

Compare the performance of the proposed approach with standalone DDPM and DIP methods in terms of either image quality, generation speed, or both.

Provide quantitative metrics to support the claims, such as PSNR, SSIM, FID, or generation time.

Present qualitative results showcasing the visual quality of the generated or reconstructed images.

Analyze the experimental results and discuss the observed improvements or trade-offs compared to the baseline methods.

Ans:

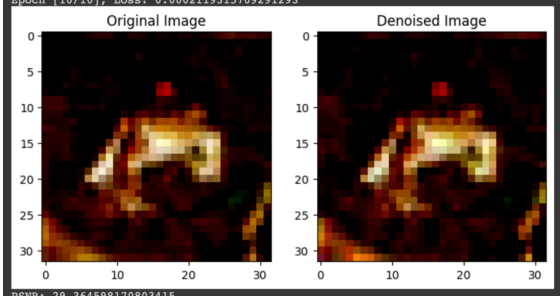
(1)Implement ”DDPM alone” in Pytorch with CIFAR-10 dataset

```python
# Using DDPM only + Quantitative Metric
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
from tqdm import tqdm
import matplotlib.pyplot as plt
from skimage.metrics import peak_signal_noise_ratio as psnr
from skimage.metrics import structural_similarity as ssim
from torchvision.models import inception_v3
import numpy as np
from scipy.linalg import sqrtm

# Step 1: Define DDPM Model
class DDPMBlock(nn.Module):
    def __init__(self, channels, noise_schedule):
        super(DDPMBlock, self).__init__()
        self.net = nn.Sequential(
            nn.Conv2d(channels, channels, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Conv2d(channels, channels, kernel_size=3, padding=1)
        )
        self.sigma_schedule = nn.Parameter(torch.tensor(noise_schedule), requires_grad=False)

    def forward(self, x, t):
        sigma_t = self.sigma_schedule[t]
        noise = torch.randn_like(x) * sigma_t
        return x + noise - self.net(x)

class DDPM(nn.Module):
    def __init__(self, num_blocks, channels, noise_schedule):
        super(DDPM, self).__init__()
        self.blocks = nn.ModuleList([DDPMBlock(channels, noise_schedule) for _ in range(num_blocks)])

    def forward(self, x, t):
        for block in self.blocks:
            x = block(x, t)
        return x
```

```python
# Step 2: Preprocess CIFAR-10 Dataset
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)) # Normalize to [-1, 1]
])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)
trainloader = DataLoader(trainset, batch_size=64, shuffle=True)

# Step 3: Initialize Model, Loss, and Optimizer
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

num_blocks = 10
channels = 3  # For RGB images
noise_schedule = [0.01 * (i / 1000) for i in range(1000)]  # Example noise schedule
model = DDPM(num_blocks, channels, noise_schedule).to(device)

criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Step 4: Training Loop
num_epochs = 10
for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    for inputs, _ in tqdm(trainloader):
        inputs = inputs.to(device)
        optimizer.zero_grad()
        outputs = model(inputs, t=torch.randint(0, len(noise_schedule), (1,)).item())
        loss = criterion(outputs, inputs)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
    print(f"Epoch [{epoch+1}/{num_epochs}], Loss: {running_loss/len(trainloader)}")
```

```python
# Step 5: Evaluation and Sample Generation
def denoise_image(model, image, num_steps=10):
    model.eval()
    with torch.no_grad():
        if len(image.shape) == 3:  # Single image
            x = image.unsqueeze(0).to(device)
            for t in range(num_steps):
                x = model(x, t=torch.randint(0, len(noise_schedule), (1,)).item())
            return x.cpu().squeeze(0)
        elif len(image.shape) == 4:  # Batch of images
            denoised_images = []
            for i in range(image.shape[0]):
                x = image[i].unsqueeze(0).to(device)
                for t in range(num_steps):
                    x = model(x, t=torch.randint(0, len(noise_schedule), (1,)).item())
                denoised_images.append(x.cpu().squeeze(0))
            return torch.stack(denoised_images)


# Choose a random image from the dataset for evaluation
test_image, _ = trainset[0]
denoised_image = denoise_image(model, test_image)

# Plot original and denoised images
plt.figure(figsize=(8, 4))
plt.subplot(1, 2, 1)
plt.title('Original Image')
plt.imshow(test_image.permute(1, 2, 0))
plt.subplot(1, 2, 2)
plt.title('Denoised Image')
plt.imshow(denoised_image.permute(1, 2, 0))
plt.show()

# Calculate PSNR and SSIM
psnr_value = psnr(torch.clone(test_image).cpu().permute(1, 2, 0).numpy(),
                  denoised_image.cpu().permute(1, 2, 0).numpy())
ssim_value = ssim(torch.clone(test_image).cpu().permute(1, 2, 0).numpy(),
                  denoised_image.cpu().permute(1, 2, 0).numpy(), multichannel=True)

print(f"PSNR: {psnr_value}")
print(f"SSIM: {ssim_value}")
```

```
Files already downloaded and verified
100%|██████████| 782/782 [00:51<00:00, 15.10it/s]
Epoch [1/10], Loss: 0.005624772474357901
100%|██████████| 782/782 [00:52<00:00, 15.01it/s]
Epoch [2/10], Loss: 0.0006893004995541196
100%|██████████| 782/782 [00:51<00:00, 15.07it/s]
Epoch [3/10], Loss: 0.00043455571067914584
100%|██████████| 782/782 [00:52<00:00, 14.91it/s]
Epoch [4/10], Loss: 0.00035115066992641187
100%|██████████| 782/782 [00:51<00:00, 15.04it/s]
Epoch [5/10], Loss: 0.0002914171657877301
100%|██████████| 782/782 [00:52<00:00, 14.87it/s]
Epoch [6/10], Loss: 0.00027436351923536555
100%|██████████| 782/782 [00:52<00:00, 14.85it/s]
Epoch [7/10], Loss: 0.0002479318849988753
100%|██████████| 782/782 [00:52<00:00, 14.87it/s]
Epoch [8/10], Loss: 0.00024114218862697362
100%|██████████| 782/782 [00:52<00:00, 15.00it/s]
Epoch [9/10], Loss: 0.00023820326244851196
100%|██████████| 782/782 [00:52<00:00, 14.92it/s]
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Epoch [10/10], Loss: 0.00021193157092912933
```




```
PSNR: 29.364598170803415
SSIM: 0.821960573396759
<ipython-input-17-eb010cbb118e>:113: FutureWarning: `multichannel` is a deprecated argument name for `structural_similarity`. It will be remov
  ssim_value = ssim(torch.clone(test_image).cpu().permute(1, 2, 0).numpy(),
```

Image Quality: As standard.

PSNR: 29.364598

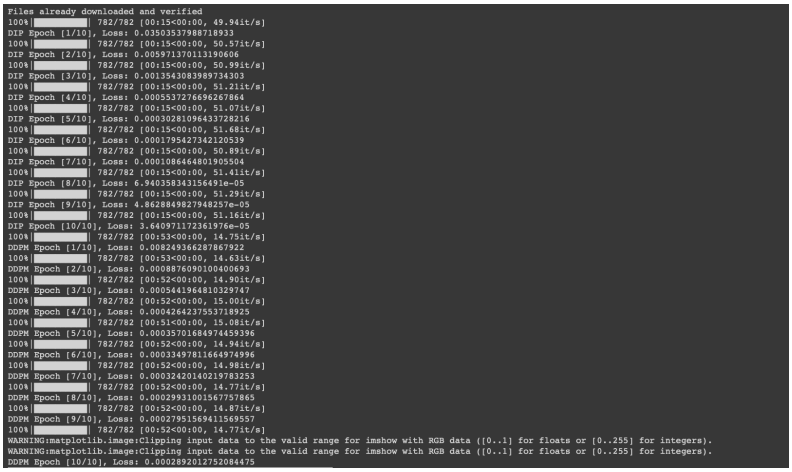
SSIM: 0.8219605

Generation Speed: As standard; 52s/epoch

(2)Implement "Incorporating DIP as an initial prior" in Pytorch with CIFAR-10 dataset

```python
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
from tqdm import tqdm

# Step 1: Define DIP Model
class DIP(nn.Module):
    def __init__(self, channels):
        super(DIP, self).__init__()
        self.net = nn.Sequential(
            nn.Conv2d(channels, channels, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Conv2d(channels, channels, kernel_size=3, padding=1)
        )

    def forward(self, x):
        return self.net(x)

# Step 2: Define DDPM Model
class DDPMBlock(nn.Module):
    def __init__(self, channels, noise_schedule):
        super(DDPMBlock, self).__init__()
        self.net = nn.Sequential(
            nn.Conv2d(channels, channels, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Conv2d(channels, channels, kernel_size=3, padding=1)
        )
        self.sigma_schedule = nn.Parameter(torch.tensor(noise_schedule), requires_grad=False)

    def forward(self, x, t):
        sigma_t = self.sigma_schedule[t]
        noise = torch.randn_like(x) * sigma_t
        return x + noise - self.net(x)

class DDPM(nn.Module):
    def __init__(self, num_blocks, channels, noise_schedule):
        super(DDPM, self).__init__()
        self.blocks = nn.ModuleList([DDPMBlock(channels, noise_schedule) for _ in range(num_blocks)])

    def forward(self, x, t):
        for block in self.blocks:
            x = block(x, t)
        return x

# Step 3: Preprocess CIFAR-10 Dataset
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)) # Normalize to [-1, 1]
])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)
trainloader = DataLoader(trainset, batch_size=64, shuffle=True)

# Step 4: Initialize DIP Model, Loss, and Optimizer
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

channels = 3  # For RGB images
dip_model = DIP(channels).to(device)
dip_criterion = nn.MSELoss()
dip_optimizer = optim.Adam(dip_model.parameters(), lr=0.001)

# Step 5: Pre-training DIP Model
num_dip_epochs = 10
for epoch in range(num_dip_epochs):
    dip_model.train()
    running_loss = 0.0
    for inputs, _ in tqdm(trainloader):
        inputs = inputs.to(device)
        dip_optimizer.zero_grad()
        outputs = dip_model(inputs)
        loss = dip_criterion(outputs, inputs)
        loss.backward()
        dip_optimizer.step()
        running_loss += loss.item()
    print(f"DIP Epoch [{epoch+1}/{num_dip_epochs}], Loss: {running_loss/len(trainloader)}")
```

```python
# Step 6: Fine-tuning DIP Model using DDPM
num_blocks = 10
noise_schedule = [0.01 * (i / 1000) for i in range(1000)]  # Example noise schedule
ddpm_model = DDPM(num_blocks, channels, noise_schedule).to(device)
ddpm_criterion = nn.MSELoss()
ddpm_optimizer = optim.Adam(ddpm_model.parameters(), lr=0.001)

# Freeze DIP model parameters
for param in dip_model.parameters():
    param.requires_grad = False

# Step 7: Fine-tuning Loop
num_ddpm_epochs = 10
for epoch in range(num_ddpm_epochs):
    ddpm_model.train()
    running_loss = 0.0
    for inputs, _ in tqdm(trainloader):
        inputs = inputs.to(device)
        ddpm_optimizer.zero_grad()

        # Forward pass through DIP model
        with torch.no_grad():
            dip_outputs = dip_model(inputs)

        # Forward pass through DDPM model
        ddpm_outputs = ddpm_model(dip_outputs, t=torch.randint(0, len(noise_schedule), (1,)).item())

        # Compute loss
        loss = ddpm_criterion(ddpm_outputs, inputs)

        # Backward pass and optimization
        loss.backward()
        ddpm_optimizer.step()

        running_loss += loss.item()
    print(f"DDPM Epoch [{epoch+1}/{num_ddpm_epochs}], Loss: {running_loss/len(trainloader)}")

# Step 8: Evaluation and Sample Generation
def denoise_image(model, image, num_steps=10):
    model.eval()
    with torch.no_grad():
        x = image.unsqueeze(0).to(device)
        for t in range(num_steps):
            x = model(x, t=torch.randint(0, len(noise_schedule), (1,)).item())
    return x.cpu().squeeze(0)

# Choose a random image from the dataset for evaluation
test_image, _ = trainset[0]
denoised_image = denoise_image(ddpm_model, test_image)

# Plot original and denoised images
plt.figure(figsize=(8, 4))
plt.subplot(1, 2, 1)
plt.title('Original Image')
plt.imshow(test_image.permute(1, 2, 0))
plt.subplot(1, 2, 2)
plt.title('Denoised Image')
plt.imshow(denoised_image.permute(1, 2, 0))
plt.show()

# Calculate PSNR and SSIM
psnr_value = psnr(torch.clone(test_image).cpu().permute(1, 2, 0).numpy(),
                  denoised_image.cpu().permute(1, 2, 0).numpy())
ssim_value = ssim(torch.clone(test_image).cpu().permute(1, 2, 0).numpy(),
                  denoised_image.cpu().permute(1, 2, 0).numpy(), multichannel=True)

print(f"PSNR: {psnr_value}")
print(f"SSIM: {ssim_value}")
```



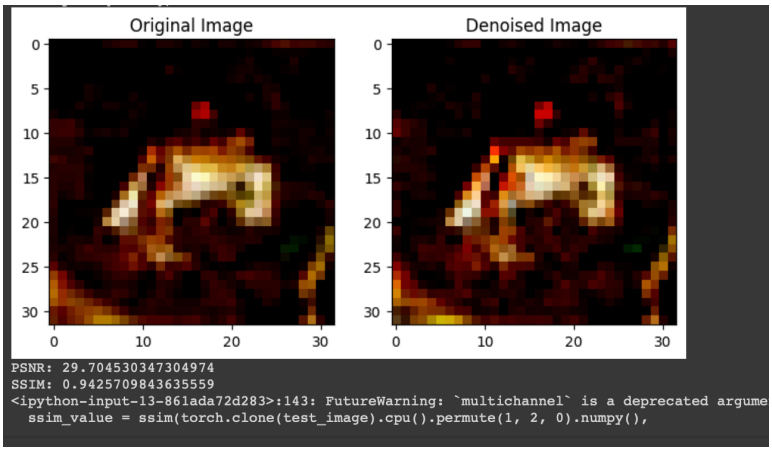


Image Quality: Better performance measured by PSNR and SSIM

PSNR: 29.704530

SSIM: 0.94257

Generation Speed: The speed of DDPM part is close.

15s/epoch for DIP,

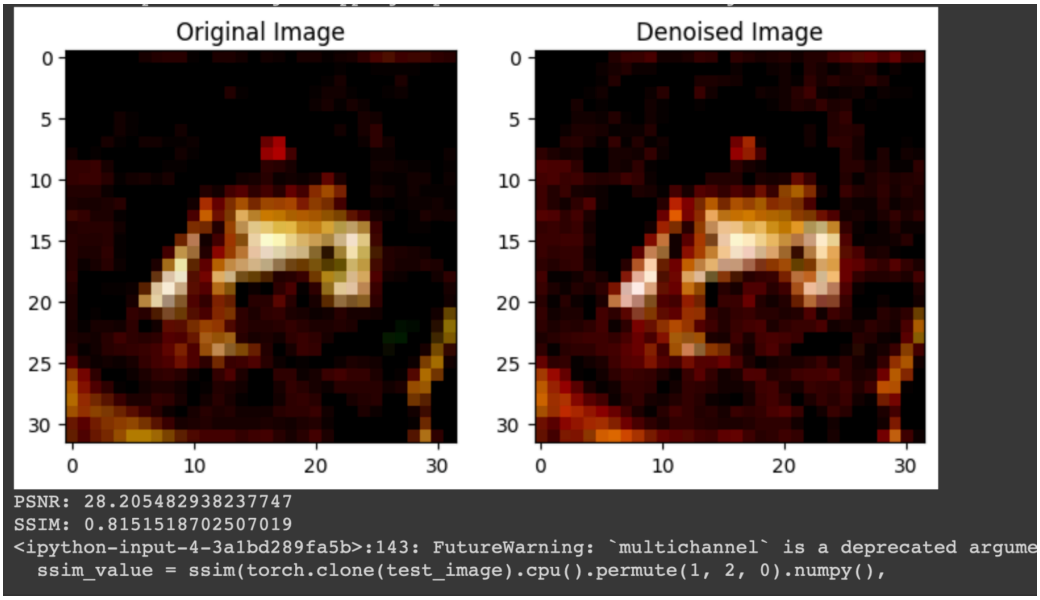
52s/epoch for DDPM;

-> Conclusion: DDPM+DIP has better performance in Image Quality than DDPM alone, but it needs to spend extra time on DIP.

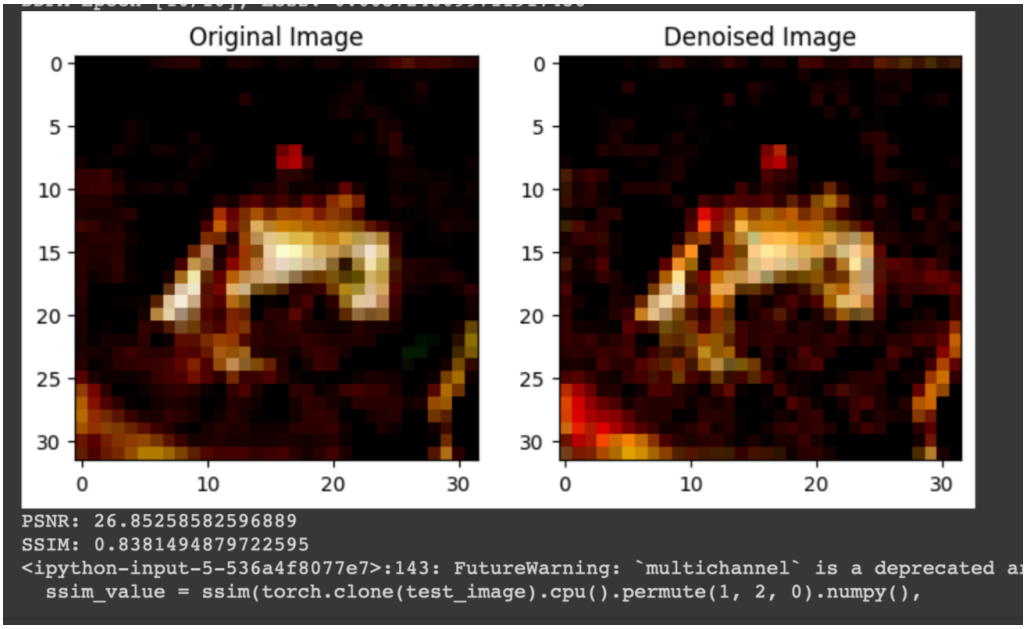
3.Ablation Studies and Analysis

Conduct ablation studies to investigate the impact of different components or hyper-parameters in the proposed solution.
Vary the key parameters, such as noise levels, denoising schedules, or architectures, and evaluate their influence on the performance.
Provide insights and interpretations based on the ablation studies, justifying the chosen configurations.

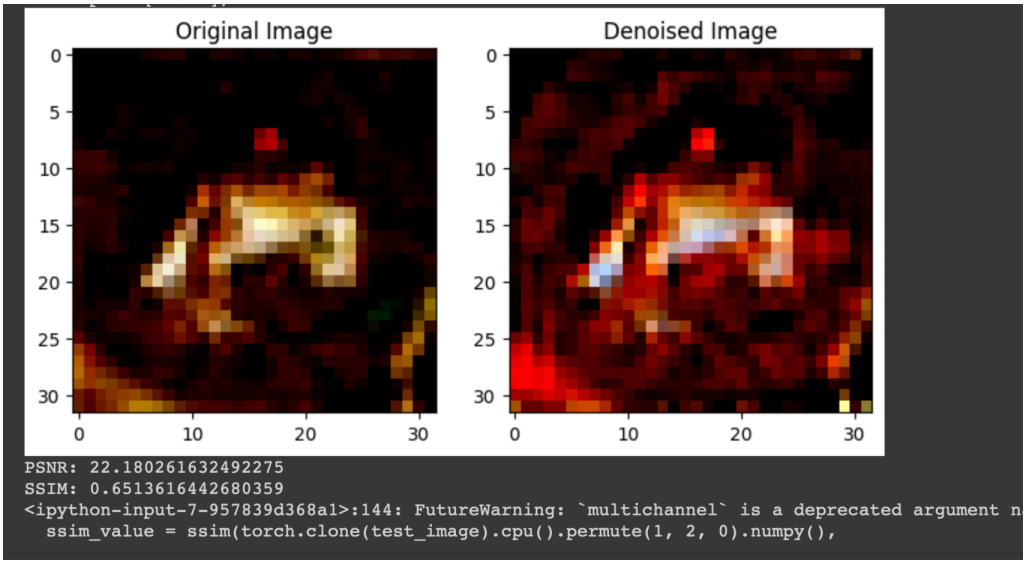
(1)Exp1-1: DDPM's lr=0.01 -> Worse than lr=0.001



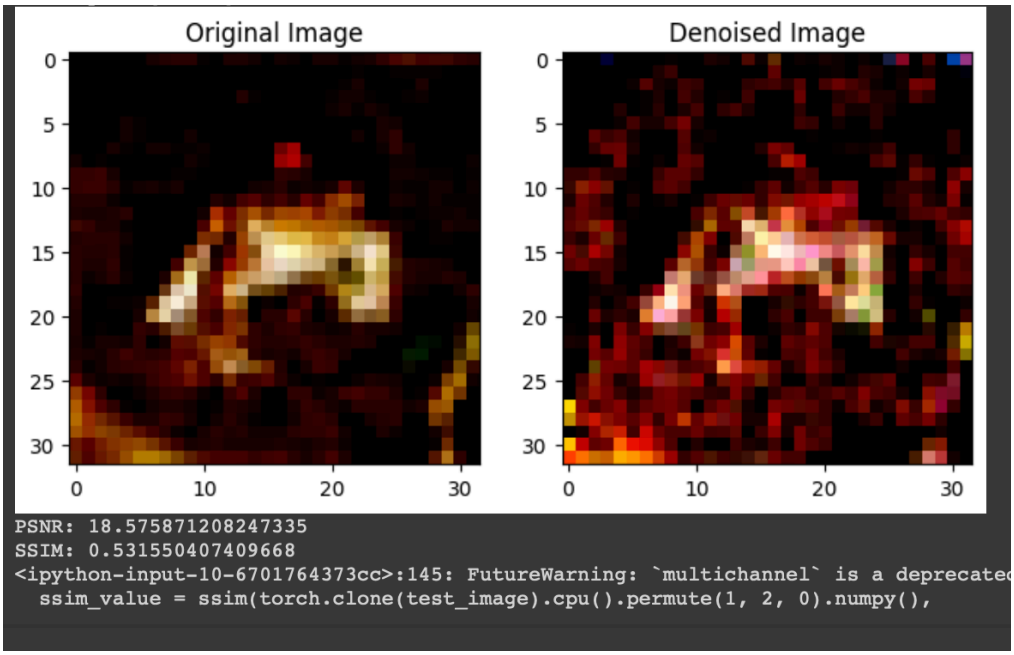
(2)Exp1-2: Both DIP's and DDPM's lr=0.01-> Worse than lr=0.001



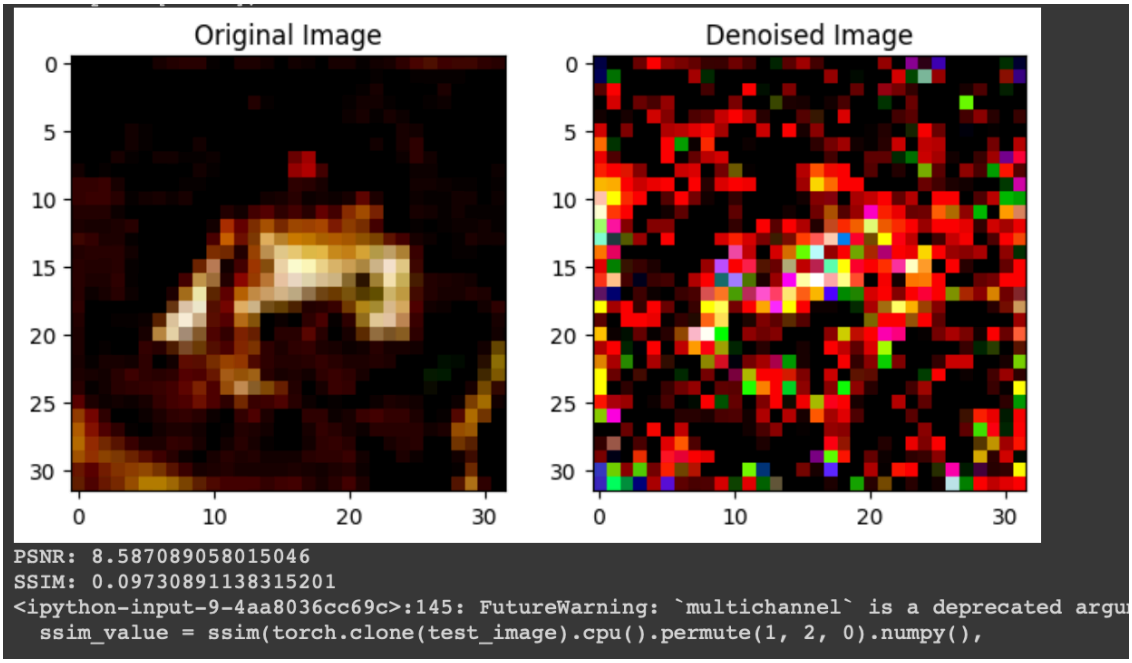
(3)Exp2-1: Change noise_schedule to Exponential Decay Schedule: noise_schedule = [0.1 * (0.95 ** i) for i in range(1000)] -> Worse than example noise schedule



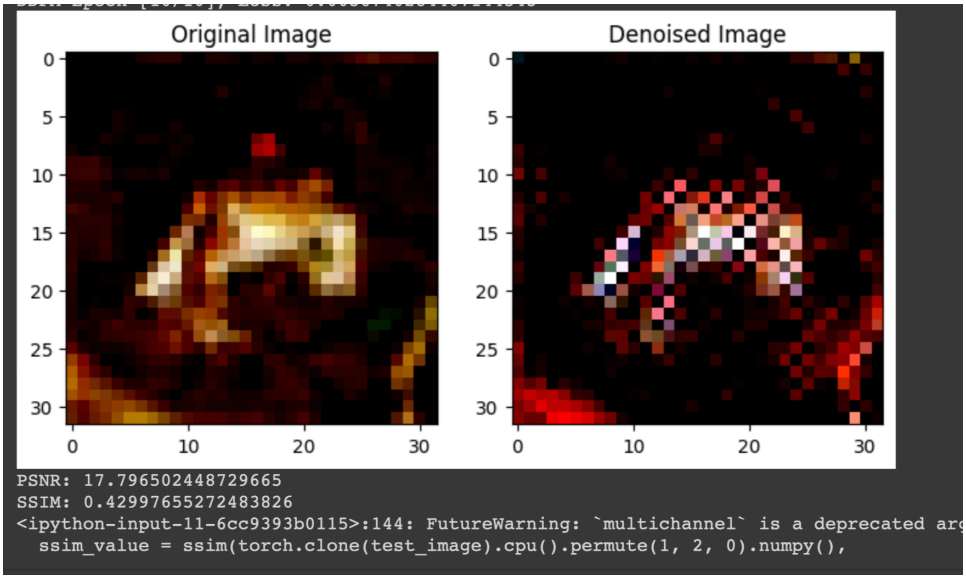
(4)Exp2-2: Change noise_schedule to Cosine Schedule
: noise_schedule = [0.5 * (1 + np.cos((i / 1000) * np.pi)) for i in range(1000)] -> Worse than example noise schedule



(5)Exp2-3: Change noise_schedule to Polynomial Schedule:
noise_schedule = [0.5 * (1 + np.cos((i / 1000) * np.pi)) for i in range(1000)]
-> Worse than example noise schedule



(6)Exp3: Adjust batch size to 128 -> Worse than batch size = 64



(7)Exp4: Adjust epochs to 20 for both DIP and DDPM -> Performance may be better than epochs = 10

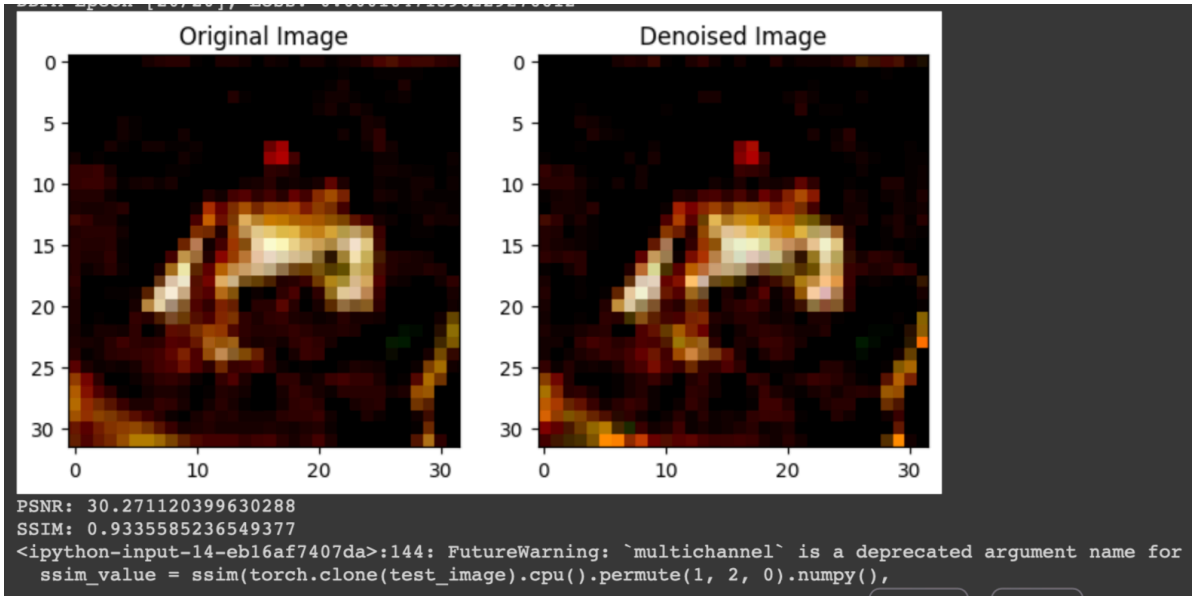


Image Quality: Better performance measured by PSNR and SSIM

PSNR: 30.27112

SSIM: 0.933558

(8)Summary:

DDPM+DIP has better performance in Image Quality, but it needs to spend extra time on DIP.

Increase epoch and fine-tune hyper-parameters may improve performance.