TUBES MESIN LEARNING

HUGGING FACE

Hugging Face_UNIT 5-8

Oleh :

Tito Alfarabi Biwarno/1103213012

PRODI S1 TEKNIK KOMPUTER

FAKULTAS TEKNIK ELEKTRO

UNIVERSITAS TELKOM

BANDUNG

2024

UNIT 5

- Section 1

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

# Hyperparameters
batch_size = 128
latent_dim = 20
learning_rate = 1e-3
epochs = 10

# Data Loading (MNIST dataset)
transform = transforms.ToTensor()
train_dataset = datasets.MNIST(root='./data', train=True,
transform=transform, download=True)
train_loader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=True)

# Define VAE Model
class VAE(nn.Module):
    def __init__(self, input_dim=784, hidden_dim=400, latent_dim=20):
        super(VAE, self).__init__()
        # Encoder
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc_mu = nn.Linear(hidden_dim, latent_dim)  # Mean
        self.fc_logvar = nn.Linear(hidden_dim, latent_dim)  # Log-
variance

        # Decoder
        self.fc2 = nn.Linear(latent_dim, hidden_dim)
        self.fc3 = nn.Linear(hidden_dim, input_dim)

    def encode(self, x):
        h = torch.relu(self.fc1(x))
        mu = self.fc_mu(h)
        logvar = self.fc_logvar(h)
        return mu, logvar

    def reparameterize(self, mu, logvar):
        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std)
        return mu + eps * std
```

```
    def decode(self, z):
        h = torch.relu(self.fc2(z))
        return torch.sigmoid(self.fc3(h))

    def forward(self, x):
        mu, logvar = self.encode(x)
        z = self.reparameterize(mu, logvar)
        x_recon = self.decode(z)
        return x_recon, mu, logvar

# Define Loss Function
def vae_loss(recon_x, x, mu, logvar):
    # Reconstruction loss
    reconstruction_loss = nn.functional.binary_cross_entropy(recon_x,
x, reduction='sum')
    # KL Divergence
    kl_divergence = -0.5 * torch.sum(1 + logvar - mu.pow(2) -
logvar.exp())
    return reconstruction_loss + kl_divergence

# Initialize Model, Optimizer, and Device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = VAE().to(device)
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

# Training Loop
model.train()
for epoch in range(epochs):
    train_loss = 0
    for batch_idx, (data, _) in enumerate(train_loader):
        data = data.view(-1, 784).to(device)  # Flatten the images
        optimizer.zero_grad()
        recon_batch, mu, logvar = model(data)
        loss = vae_loss(recon_batch, data, mu, logvar)
        loss.backward()
        train_loss += loss.item()
        optimizer.step()

    print(f"Epoch {epoch + 1}, Loss: {train_loss /
len(train_loader.dataset):.4f}")

# Generate New Data
model.eval()
with torch.no_grad():
    z = torch.randn(64, latent_dim).to(device)  # Random latent vector
    generated_images = model.decode(z).view(-1, 1, 28, 28)
```

```
# Visualize Generated Images
import matplotlib.pyplot as plt
import torchvision

grid = torchvision.utils.make_grid(generated_images.cpu(), nrow=8)
plt.figure(figsize=(8, 8))
plt.imshow(grid.permute(1, 2, 0))
plt.title("Generated Images from VAE")
plt.axis("off")
plt.show()
```

Penjelasan code

1. Import Library dan Hyperparameters

- Import Library: Menggunakan PyTorch untuk membangun model, optimasi, dan transformasi data.
- Hyperparameters:
    - batch_size: Jumlah data dalam satu batch.
    - latent_dim: Dimensi ruang laten untuk encoding.
    - : Kecepatan pembelajaran model.
    - epochs: Jumlah iterasi pelatihan.

2. Data Loading (MNIST Dataset)

- MNIST Dataset: Dataset gambar angka (0-9) berukuran 28x28.
- Transformasi: Mengubah gambar menjadi tensor untuk diproses oleh model.
- DataLoader: Membagi dataset ke dalam batch untuk efisiensi pelatihan.

3. Definisi Model VAE

- Encoder:
    - fc1: Mengubah input menjadi representasi laten.
    - fc_mu dan fc_logvar: Menghasilkan mean (μ μ) dan log-variance (log f_0 (σ 2) log(σ 2)).
- Reparameterization Trick:
    - Membuat sampel dari distribusi laten menggunakan noise Gaussian agar sampling tetap diferensial.
- Decoder:
    - fc2: Mengubah data laten (z z) ke representasi tersembunyi.
    - fc3: Mengembalikan representasi tersembunyi ke bentuk asli (784 piksel).

4. Definisi Loss Function

- Reconstruction Loss:
    - Mengukur seberapa baik output (rekonstruksi) mendekati input asli.
    - Digunakan binary cross-entropy.
- KL Divergence:
    - Menghitung seberapa jauh distribusi laten (z z) dari prior distribusi Gaussian standar.
- Total Loss:
    - Kombinasi Reconstruction Loss dan KL Divergence.

5. Training Loop

- Model Initialization: Membuat objek model dan optimizer (Adam).
- Pelatihan:
    - Input data di-flatten menjadi vektor (784 dimensi).
    - Model menghitung rekonstruksi dan distribusi laten.
    - Loss dihitung dan dilakukan backpropagation.
    - Bobot diperbarui berdasarkan gradien.

6. Generasi Data Baru

- Mode Evaluasi: Model tidak melakukan pembaruan bobot.
- Random Sampling: Membuat data baru dengan sampling dari distribusi Gaussian standar (z z).
- Decoder: Mengonversi representasi laten menjadi gambar.

7. Visualisasi Gambar yang Dihasilkan

- Grid: Membuat susunan gambar dari hasil generasi.
- Visualisasi: Menggunakan matplotlib untuk menampilkan gambar hasil generasi.

Output

Model menghasilkan gambar angka baru (digit MNIST) yang menyerupai data asli tetapi belum pernah ada sebelumnya.

- **Section 2**

```
# Bagian 1: Import Library dan Hyperparameters
# Mengimpor pustaka PyTorch, torchvision, dan matplotlib.
# Menentukan parameter penting untuk pelatihan:
# - latent_dim: Dimensi noise vector (input untuk Generator).
# - batch_size: Jumlah data dalam satu batch.
```

```
# - learning_rate: Kecepatan pembelajaran untuk optimizer.
# - epochs: Jumlah iterasi pelatihan.

import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
import torchvision

# Hyperparameters
latent_dim = 100  # Dimensi noise vector
batch_size = 128
learning_rate = 2e-4
epochs = 50

# Bagian 2: Load Dataset MNIST
# Dataset MNIST digunakan sebagai input. Dataset ini berisi gambar
angka (0-9) berukuran 28x28 piksel.
# Data diubah menjadi tensor dan dinormalisasi ke rentang [-1, 1] agar
sesuai dengan output Tanh pada Generator.
# DataLoader membagi dataset menjadi batch agar lebih efisien selama
pelatihan.
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))  # Menormalkan data ke rentang
[-1, 1]
])
dataset = datasets.MNIST(root='./data', train=True,
transform=transform, download=True)
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)

# Bagian 3: Definisi Model Generator
# Generator adalah model yang menghasilkan gambar palsu dari noise acak
(latent vector).
# - Input berupa noise dengan dimensi `latent_dim`.
# - Model menggunakan lapisan fully connected dengan aktivasi ReLU.
# - Aktivasi akhir menggunakan Tanh untuk memastikan output berada
dalam rentang [-1, 1].
# - Output dirancang untuk memiliki ukuran gambar (28x28).
class Generator(nn.Module):
    def __init__(self, latent_dim):
        super(Generator, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(latent_dim, 256),
            nn.ReLU(),
            nn.Linear(256, 512),
```

```
            nn.ReLU(),
            nn.Linear(512, 1024),
            nn.ReLU(),
            nn.Linear(1024, 28 * 28),
            nn.Tanh()  # Output dengan rentang [-1, 1]
        )

    def forward(self, z):
        img = self.model(z)
        return img.view(-1, 1, 28, 28)  # Mengubah output menjadi
ukuran gambar

# Bagian 4: Definisi Model Discriminator
# Discriminator adalah model yang membedakan gambar asli dan palsu.
# - Input berupa gambar (28x28) yang di-flatten menjadi vektor datar.
# - Menggunakan beberapa lapisan fully connected dengan aktivasi
LeakyReLU.
# - Output berupa probabilitas (0 atau 1) menggunakan aktivasi Sigmoid.
# - Probabilitas ini menentukan apakah gambar asli (1) atau palsu (0).
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(28 * 28, 1024),
            nn.LeakyReLU(0.2),
            nn.Linear(1024, 512),
            nn.LeakyReLU(0.2),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, 1),
            nn.Sigmoid()  # Output berupa probabilitas (0 atau 1)
        )

    def forward(self, img):
        img_flat = img.view(img.size(0), -1)  # Mengubah gambar menjadi
vektor
        return self.model(img_flat)

# Bagian 5: Inisialisasi Model dan Optimizer
# - Inisialisasi model Generator dan Discriminator.
# - Optimizer menggunakan Adam dengan parameter learning_rate dan beta
umum untuk GANs.
# - Loss function menggunakan Binary Cross-Entropy Loss (BCELoss),
cocok untuk membedakan dua kelas: asli dan palsu.

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
generator = Generator(latent_dim).to(device)
discriminator = Discriminator().to(device)
```

```
optimizer_G = optim.Adam(generator.parameters(), lr=learning_rate,
betas=(0.5, 0.999))
optimizer_D = optim.Adam(discriminator.parameters(), lr=learning_rate,
betas=(0.5, 0.999))

# Loss Function
adversarial_loss = nn.BCELoss()  # Loss Binary Cross-Entropy

# Bagian 6: Training Loop
# Proses pelatihan GAN dilakukan dalam dua langkah:
# 1. Melatih Generator:
#    - Membuat noise acak dan menghasilkan gambar palsu.
#    - Generator dilatih untuk "menipu" Discriminator agar menganggap
gambar palsu sebagai gambar asli.
# 2. Melatih Discriminator:
#    - Melatih Discriminator untuk membedakan gambar asli dan palsu.
#    - Menghitung loss untuk kedua jenis input: gambar asli dan gambar
palsu.
# Proses ini dilakukan secara bergantian untuk menjaga keseimbangan
pelatihan.

for epoch in range(epochs):
    for i, (imgs, _) in enumerate(dataloader):
        # Gambar asli (real images)
        real_imgs = imgs.to(device)
        valid = torch.ones(imgs.size(0), 1, device=device)  # Label
untuk gambar asli (1)
        fake = torch.zeros(imgs.size(0), 1, device=device)  # Label
untuk gambar palsu (0)

        # -----------------
        # Latih Generator
        # -----------------
        optimizer_G.zero_grad()

        # Membuat noise dan menghasilkan gambar palsu
        z = torch.randn(imgs.size(0), latent_dim, device=device)
        generated_imgs = generator(z)

        # Hitung loss untuk Generator (berusaha "menipu" Discriminator)
        g_loss = adversarial_loss(discriminator(generated_imgs), valid)
        g_loss.backward()
        optimizer_G.step()

        # ---------------------
        # Latih Discriminator
        # ---------------------
```

```
        optimizer_D.zero_grad()

        # Hitung loss Discriminator (untuk gambar asli dan palsu)
        real_loss = adversarial_loss(discriminator(real_imgs), valid)
        fake_loss =
adversarial_loss(discriminator(generated_imgs.detach()), fake)
        d_loss = (real_loss + fake_loss) / 2
        d_loss.backward()
        optimizer_D.step()

    print(f"Epoch [{epoch + 1}/{epochs}]  D Loss:
{d_loss.item():.4f}  G Loss: {g_loss.item():.4f}")

# Bagian 7: Visualisasi Hasil
# Setelah beberapa epoch, Generator akan mulai menghasilkan gambar yang
mirip dengan data asli.
# - Gambar palsu yang dihasilkan Generator divisualisasikan menggunakan
matplotlib.
# - Gambar ini disusun dalam grid untuk memudahkan analisis hasil
pelatihan.
# - Visualisasi dilakukan setiap 10 epoch untuk memantau perkembangan
Generator.
    if (epoch + 1) % 10 == 0:
        with torch.no_grad():
            z = torch.randn(64, latent_dim, device=device)
            generated_imgs = generator(z)
            grid = torchvision.utils.make_grid(generated_imgs, nrow=8,
normalize=True)
            plt.figure(figsize=(8, 8))
            plt.imshow(grid.permute(1, 2, 0).cpu().numpy())
            plt.title(f"Gambar Hasil Generasi Epoch {epoch + 1}")
            plt.axis("off")
            plt.show()
```

- **Section 3**

```
##Bagian 1
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
from torchvision.utils import make_grid

# Hyperparameters
latent_dim = 512  # Dimensi noise vector
image_size = 64  # Ukuran gambar output
learning_rate = 1e-4
epochs = 100
```

```
•
•  # Bagian 2
•  class MappingNetwork(nn.Module):
•      def __init__(self, latent_dim, style_dim=512):
•          super(MappingNetwork, self).__init__()
•          self.network = nn.Sequential(
•              nn.Linear(latent_dim, style_dim),
•              nn.ReLU(),
•              nn.Linear(style_dim, style_dim),
•              nn.ReLU()
•          )
•
•      def forward(self, z):
•          return self.network(z)  # Menghasilkan style vector
•
•  # Bagian 3
•  class ModulatedConv2D(nn.Module):
•      def __init__(self, in_channels, out_channels, kernel_size,
   style_dim):
•          super(ModulatedConv2D, self).__init__()
•          self.conv = nn.Conv2d(in_channels, out_channels,
   kernel_size, stride=1, padding=kernel_size // 2)
•          self.style_affine = nn.Linear(style_dim, in_channels)
•
•      def forward(self, x, style):
•          batch_size, _, height, width = x.size()
•          gamma = self.style_affine(style).view(batch_size, -1, 1,
   1)  # Sesuaikan dimensi batch
•          x = gamma * x  # Broadcast untuk setiap batch
•          return self.conv(x)
•
•  # Bagian 4
•  class Generator(nn.Module):
•      def __init__(self, latent_dim, style_dim, image_size):
•          super(Generator, self).__init__()
•          self.mapping = MappingNetwork(latent_dim, style_dim)
•          self.initial = nn.Parameter(torch.randn(1, 512, 4, 4))  #
   Tensor awal
•          self.conv1 = ModulatedConv2D(512, 256, 3, style_dim)
•          self.conv2 = ModulatedConv2D(256, 128, 3, style_dim)
•          self.conv3 = ModulatedConv2D(128, 64, 3, style_dim)
•          self.conv4 = ModulatedConv2D(64, 3, 3, style_dim)  # RGB
   output
•          self.upsample = nn.Upsample(scale_factor=2)
•
•      def forward(self, z):
•          style = self.mapping(z)
```

```
        x = self.initial.expand(z.size(0), -1, -1, -1)  # Menggunakan tensor awal
        x = self.upsample(torch.relu(self.conv1(x, style)))  # 8x8
        x = self.upsample(torch.relu(self.conv2(x, style)))  # 16x16
        x = self.upsample(torch.relu(self.conv3(x, style)))  # 32x32
        x = self.upsample(torch.relu(self.conv4(x, style)))  # 64x64
        return x

# Bagian 5
class Discriminator(nn.Module):
    def __init__(self, image_size):
        super(Discriminator, self).__init__()
        self.model = nn.Sequential(
            nn.Conv2d(3, 128, kernel_size=4, stride=2, padding=1),  # 64x64 -> 32x32
            nn.LeakyReLU(0.2),
            nn.Conv2d(128, 256, kernel_size=4, stride=2, padding=1),  # 32x32 -> 16x16
            nn.LeakyReLU(0.2),
            nn.Conv2d(256, 512, kernel_size=4, stride=2, padding=1),  # 16x16 -> 8x8
            nn.LeakyReLU(0.2),
            nn.Conv2d(512, 1024, kernel_size=4, stride=2, padding=1),  # 8x8 -> 4x4
            nn.LeakyReLU(0.2),
            nn.Flatten(),
            nn.Linear(1024 * (image_size // 16) * (image_size // 16), 1),  # 1024 * 4 * 4 -> 1
            nn.Sigmoid()
        )

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

# Initialize Models and Optimizers
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
generator = Generator(latent_dim, 512, image_size).to(device)
discriminator = Discriminator(image_size).to(device)

optimizer_G = optim.Adam(generator.parameters(), lr=learning_rate, betas=(0.5, 0.999))
optimizer_D = optim.Adam(discriminator.parameters(), lr=learning_rate, betas=(0.5, 0.999))
```

```
•
•  adversarial_loss = nn.BCELoss()
•
•  # Bagian 6
•  for epoch in range(epochs):
•      z = torch.randn(16, latent_dim).to(device)  # Noise input
•      real_imgs = torch.randn(16, 3, image_size,
   image_size).to(device)  # Dummy real images
•
•      # Train Generator
•      optimizer_G.zero_grad()
•      fake_imgs = generator(z)
•      g_loss = adversarial_loss(discriminator(fake_imgs),
   torch.ones(16, 1).to(device))  # Fool Discriminator
•      g_loss.backward()
•      optimizer_G.step()
•
•      # Train Discriminator
•      optimizer_D.zero_grad()
•      real_loss = adversarial_loss(discriminator(real_imgs),
   torch.ones(16, 1).to(device))
•      fake_loss =
   adversarial_loss(discriminator(fake_imgs.detach()),
   torch.zeros(16, 1).to(device))
•      d_loss = (real_loss + fake_loss) / 2
•      d_loss.backward()
•      optimizer_D.step()
•
•      # Logging and Visualization
•      if (epoch + 1) % 10 == 0:
•          print(f"Epoch [{epoch + 1}/{epochs}]  D Loss:
   {d_loss.item():.4f}  G Loss: {g_loss.item():.4f}")
•          with torch.no_grad():
•              generated_imgs = generator(z).cpu()
•              grid = make_grid(generated_imgs, nrow=4,
   normalize=True)
•              plt.figure(figsize=(8, 8))
•              plt.imshow(grid.permute(1, 2, 0))
•              plt.title(f"Gambar Hasil Generasi Epoch {epoch + 1}")
•              plt.axis("off")
•              plt.show()
•
```

Penjelasan code

Bagian 1:

- Mengimpor pustaka PyTorch untuk membuat model, optimizer, dan fungsi aktivasi.

- Hyperparameter:
    - latent_dim: Dimensi vektor noise yang menjadi input Generator.
    - image_size: Ukuran gambar output (dalam piksel).
    - learning_rate: Kecepatan pembelajaran untuk optimizer.
    - epochs: Jumlah iterasi pelatihan.

Bagian 2:

- Mapping Network mengonversi noise vector ((z)) menjadi style vector.
- Dua lapisan linear diikuti aktivasi ReLU menghasilkan representasi gaya yang digunakan untuk memodulasi Generator.

Bagian 3:

- Lapisan konvolusi dimodulasi menggunakan style vector.
- Style vector ((\gamma)) dihitung dengan affine transformation dan dikalikan dengan input gambar.
- Menggunakan broadcasting untuk memastikan kompatibilitas dimensi.

Bagian 4:

- Generator menghasilkan gambar dari noise vector ((z)).
- Memulai dengan tensor kecil ((4 \times 4)) dan memperbesar ukuran gambar menggunakan Upsample.
- Modulated convolution memodulasi gambar di setiap langkah menggunakan style vector.

Bagian 5:

- Discriminator membedakan gambar asli dan palsu.
- Menggunakan konvolusi bertahap untuk mengecilkan gambar dari (64 \to 4).
- Lapisan linear terakhir menghasilkan probabilitas (1: asli, 0: palsu).

Bagian 6:

- Train Generator:
    - Dilatih untuk menghasilkan gambar yang mampu "menipu" Discriminator.
- Train Discriminator:
    - Dilatih untuk membedakan gambar asli dan palsu.
- Visualisasi:
    - Gambar hasil Generator divisualisasikan setiap 10 epoch untuk memantau hasil pelatihan.

- **Section 4**

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import transforms, datasets
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt

# Hyperparameters
image_size = 128
batch_size = 16
learning_rate = 2e-4
epochs = 100

# Define Residual Block
class ResidualBlock(nn.Module):
    def __init__(self, channels):
        super(ResidualBlock, self).__init__()
        self.block = nn.Sequential(
            nn.Conv2d(channels, channels, kernel_size=3, stride=1, padding=1),
            nn.InstanceNorm2d(channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(channels, channels, kernel_size=3, stride=1, padding=1),
            nn.InstanceNorm2d(channels)
        )

    def forward(self, x):
        return x + self.block(x)  # Skip connection

# Generator
class Generator(nn.Module):
    def __init__(self, in_channels, out_channels, num_residuals=6):
        super(Generator, self).__init__()
        # Initial convolution
        self.initial = nn.Sequential(
            nn.Conv2d(in_channels, 64, kernel_size=7, stride=1, padding=3),
            nn.InstanceNorm2d(64),
            nn.ReLU(inplace=True)
        )

        # Downsampling
        self.downsample = nn.Sequential(
```

```
            nn.Conv2d(64, 128, kernel_size=3, stride=2,
padding=1),
            nn.InstanceNorm2d(128),
            nn.ReLU(inplace=True),
            nn.Conv2d(128, 256, kernel_size=3, stride=2,
padding=1),
            nn.InstanceNorm2d(256),
            nn.ReLU(inplace=True)
        )

        # Residual blocks
        self.residuals = nn.Sequential(*[ResidualBlock(256) for _
in range(num_residuals)])

        # Upsampling
        self.upsample = nn.Sequential(
            nn.ConvTranspose2d(256, 128, kernel_size=3, stride=2,
padding=1, output_padding=1),
            nn.InstanceNorm2d(128),
            nn.ReLU(inplace=True),
            nn.ConvTranspose2d(128, 64, kernel_size=3, stride=2,
padding=1, output_padding=1),
            nn.InstanceNorm2d(64),
            nn.ReLU(inplace=True)
        )

        # Output layer
        self.output = nn.Sequential(
            nn.Conv2d(64, out_channels, kernel_size=7, stride=1,
padding=3),
            nn.Tanh()
        )

    def forward(self, x):
        x = self.initial(x)
        x = self.downsample(x)
        x = self.residuals(x)
        x = self.upsample(x)
        return self.output(x)

# Discriminator
class Discriminator(nn.Module):
    def __init__(self, in_channels):
        super(Discriminator, self).__init__()
        self.model = nn.Sequential(
            nn.Conv2d(in_channels, 64, kernel_size=4, stride=2,
padding=1),
            nn.LeakyReLU(0.2, inplace=True),
```

```
            nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1),
            nn.InstanceNorm2d(128),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(128, 256, kernel_size=4, stride=2, padding=1),
            nn.InstanceNorm2d(256),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(256, 512, kernel_size=4, stride=1, padding=1),
            nn.InstanceNorm2d(512),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(512, 1, kernel_size=4, stride=1, padding=1)
        )

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

# Initialize models
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
G_XtoY = Generator(3, 3).to(device)
G_YtoX = Generator(3, 3).to(device)
D_X = Discriminator(3).to(device)
D_Y = Discriminator(3).to(device)

# Optimizers
optimizer_G = optim.Adam(list(G_XtoY.parameters()) + list(G_YtoX.parameters()), lr=learning_rate, betas=(0.5, 0.999))
optimizer_D_X = optim.Adam(D_X.parameters(), lr=learning_rate, betas=(0.5, 0.999))
optimizer_D_Y = optim.Adam(D_Y.parameters(), lr=learning_rate, betas=(0.5, 0.999))

# Losses
adversarial_loss = nn.MSELoss()
cycle_loss = nn.L1Loss()

# Dummy DataLoader (Replace with real data in practical use)
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))
])
dummy_dataset = datasets.FakeData(size=100, image_size=(3, image_size, image_size), transform=transform)
data_loader = DataLoader(dummy_dataset, batch_size=batch_size, shuffle=True)
```

```
•
•  # Training Loop
•  for epoch in range(epochs):
•      for i, (real_X, _) in enumerate(data_loader):
•          real_X = real_X.to(device)
•          real_Y = real_X.flip(-1).to(device)  # Dummy real_Y as
   flipped real_X for demonstration
•
•          # Train Generators
•          optimizer_G.zero_grad()
•
•          fake_Y = G_XtoY(real_X)
•          fake_X = G_YtoX(real_Y)
•          cycle_X = G_YtoX(fake_Y)
•          cycle_Y = G_XtoY(fake_X)
•
•          loss_G_XtoY = adversarial_loss(D_Y(fake_Y),
   torch.ones_like(D_Y(fake_Y)))
•          loss_G_YtoX = adversarial_loss(D_X(fake_X),
   torch.ones_like(D_X(fake_X)))
•          loss_cycle_X = cycle_loss(cycle_X, real_X)
•          loss_cycle_Y = cycle_loss(cycle_Y, real_Y)
•
•          loss_G = loss_G_XtoY + loss_G_YtoX + 10 * (loss_cycle_X +
   loss_cycle_Y)
•          loss_G.backward()
•          optimizer_G.step()
•
•          # Train Discriminators
•          optimizer_D_X.zero_grad()
•          optimizer_D_Y.zero_grad()
•
•          loss_D_X = (adversarial_loss(D_X(real_X),
   torch.ones_like(D_X(real_X))) +
•                      adversarial_loss(D_X(fake_X.detach()),
   torch.zeros_like(D_X(fake_X)))) * 0.5
•          loss_D_Y = (adversarial_loss(D_Y(real_Y),
   torch.ones_like(D_Y(real_Y))) +
•                      adversarial_loss(D_Y(fake_Y.detach()),
   torch.zeros_like(D_Y(fake_Y)))) * 0.5
•
•          loss_D_X.backward()
•          loss_D_Y.backward()
•          optimizer_D_X.step()
•          optimizer_D_Y.step()
•
```

```
        print(f"Epoch [{epoch + 1}/{epochs}]  Loss_G:
    {loss_G.item():.4f}  Loss_D_X: {loss_D_X.item():.4f}  Loss_D_Y:
    {loss_D_Y.item():.4f}")

```

Penjelasan Kode

1. **Generator**:
    - Memiliki struktur dengan downsampling, residual blocks, dan upsampling.
    - Bertugas menerjemahkan gambar dari domain (X) ke domain (Y) (atau sebaliknya).
2. **Discriminator**:
    - Model PatchGAN untuk membedakan apakah input adalah gambar asli atau hasil terjemahan.
3. **Loss Functions**:
    - **Adversarial Loss**: Mendorong Generator menghasilkan gambar realistis.
    - **Cycle Consistency Loss**: Memastikan bahwa gambar dapat diterjemahkan kembali ke domain aslinya.
4. **DataLoader**:
    - Menggunakan dataset dummy untuk demonstrasi; dalam praktiknya, gunakan dataset nyata.
5. **Training Loop**:
    - Melatih Generator dan Discriminator secara bergantian.
    - Visualisasi hasil dapat ditambahkan untuk memantau kinerja.

- **Section 5**

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt

# Hyperparameters
image_size = 28  # Ukuran gambar (MNIST)
batch_size = 256
learning_rate = 1e-4
epochs = 5
timesteps = 100  # Jumlah langkah noise

```

```
# Dataset MNIST
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])
dataset = datasets.MNIST(root='./data', train=True,
transform=transform, download=True)
dataloader = DataLoader(dataset, batch_size=batch_size,
shuffle=True)

# Device setup
device = torch.device("cuda" if torch.cuda.is_available() else
"cpu")

# Noise Schedule
def linear_noise_schedule(timesteps):
    return torch.linspace(1e-4, 0.02, timesteps, device=device)

betas = linear_noise_schedule(timesteps)
alphas = 1 - betas
alphas_cumprod = torch.cumprod(alphas, axis=0).to(device)

# Diffusion Forward Process
def forward_diffusion(x0, t):
    noise = torch.randn_like(x0)
    alpha_t = alphas_cumprod[t].view(-1, 1, 1, 1).to(x0.device)
    noisy_x = torch.sqrt(alpha_t) * x0 + torch.sqrt(1 - alpha_t)
* noise
    return noisy_x, noise

# Simple Denoising Model (UNet-like)
class SimpleDenoisingModel(nn.Module):
    def __init__(self):
        super(SimpleDenoisingModel, self).__init__()
        self.model = nn.Sequential(
            nn.Conv2d(1, 64, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=3, stride=1,
padding=1),
            nn.ReLU(),
            nn.Conv2d(64, 1, kernel_size=3, stride=1, padding=1)
        )

    def forward(self, x, t):
        # Menggunakan time embedding sederhana
        t_embedding = t.view(-1, 1, 1, 1).expand_as(x)
        return self.model(x + t_embedding)

```

```
# Initialize Model and Optimizer
model = SimpleDenoisingModel().to(device)
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
criterion = nn.MSELoss()

# Training Loop
for epoch in range(epochs):
    for batch, (x, _) in enumerate(dataloader):
        x = x.to(device)  # MNIST grayscale (1 channel)
        t = torch.randint(0, timesteps, (x.size(0),),
device=device).long()

        noisy_x, noise = forward_diffusion(x, t)
        predicted_noise = model(noisy_x, t)

        loss = criterion(predicted_noise, noise)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    print(f"Epoch [{epoch + 1}/{epochs}] Loss:
{loss.item():.4f}")

# Visualizing Reverse Diffusion Process
def reverse_diffusion(x_t, model, timesteps):
    for t in range(timesteps - 1, -1, -1):
        with torch.no_grad():
            noise_pred = model(x_t, torch.tensor([t],
device=x_t.device).long())
            alpha_t = alphas_cumprod[t].view(-1, 1, 1,
1).to(x_t.device)
            beta_t = betas[t].view(-1, 1, 1, 1).to(x_t.device)
            x_t = (x_t - torch.sqrt(1 - alpha_t) * noise_pred) /
torch.sqrt(alpha_t)
            if t > 0:
                x_t += torch.sqrt(beta_t) * torch.randn_like(x_t)
    return x_t

# Generate Images
test_noise = torch.randn((16, 1, image_size,
image_size)).to(device)
generated_images = reverse_diffusion(test_noise, model,
timesteps).cpu()

# Visualize Generated Images
grid = torch.cat([img.squeeze(0) for img in generated_images],
dim=1)
plt.figure(figsize=(15, 15))
```

```
plt.imshow(grid, cmap='gray')
plt.axis("off")
plt.title("Generated Images via Diffusion")
plt.show()

```

Penjelasan kode

1. Dataset dan Transformasi

Diffusion Models menggunakan dataset gambar, seperti MNIST, untuk pelatihan. Data gambar dinormalisasi ke rentang ([-1, 1]) untuk memastikan kesesuaian dengan keluaran model generatif. Dataset dibagi menjadi batch menggunakan DataLoader untuk efisiensi selama pelatihan.

2. Noise Schedule

Proses difusi menggunakan noise schedule untuk menentukan tingkat noise yang ditambahkan pada setiap langkah (t). Pada implementasi ini, digunakan linear noise schedule, di mana nilai noise ((\beta_t)) meningkat secara linier seiring dengan langkah waktu. Produk kumulatif (\alpha_t) dihitung untuk menentukan proporsi data asli yang tetap terjaga pada setiap langkah.

[\alpha_t = \prod_{i=1}^{t} (1 - \beta_i)]

3. Forward Diffusion Process

Proses difusi maju menambahkan noise Gaussian ke data asli (x_0) untuk menghasilkan (x_t) pada langkah (t). Formula untuk menghasilkan (x_t) adalah sebagai berikut:

[x_t = \sqrt{\alpha_t} x_0 + \sqrt{1 - \alpha_t} \epsilon]

di mana:

- (\alpha_t): Proporsi data asli yang tetap.
- (\epsilon): Noise Gaussian yang ditambahkan.

Proses ini bertujuan untuk membawa data asli (x_0) menuju distribusi Gaussian murni secara bertahap.

4. Model Denoising

Model denoising adalah jaringan saraf convolutional sederhana yang dilatih untuk memprediksi noise Gaussian (\epsilon) yang ditambahkan selama proses difusi maju. Model menerima input berupa data noisy (x_t) dan embedding waktu (t). Embedding waktu ditambahkan ke data sebagai bagian dari input untuk memberikan informasi tentang langkah difusi saat ini.

5. Reverse Diffusion Process

Setelah model dilatih, proses difusi dibalik untuk menghasilkan kembali data asli dari noise Gaussian. Formula untuk membalikkan noise pada setiap langkah (t) adalah:

[x_{t-1} = \frac{x_t - \sqrt{1 - \alpha_t} \hat{\epsilon}_t}{\sqrt{\alpha_t}}]

di mana (\hat{\epsilon}_t) adalah prediksi noise yang dihasilkan oleh model denoising. Proses ini dilakukan secara iteratif dari (t = T) hingga (t = 0) untuk merekonstruksi data asli (x_0).

6. Hasil Generasi

Model ini menghasilkan gambar baru dengan memulai dari noise Gaussian acak (x_T) dan membalikkan proses difusi. Gambar hasil generasi menunjukkan bahwa model mampu belajar pola distribusi data asli selama pelatihan.

- **Section 6**

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torchvision.utils import save_image
from torch.utils.data import DataLoader

# Hyperparameters
latent_dim = 128
image_size = 64
batch_size = 16
epochs = 20
timesteps = 100
learning_rate = 1e-4

# Dataset MNIST
transform = transforms.Compose([
    transforms.Resize((image_size, image_size)),
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])
dataset = datasets.MNIST(root='./data', train=True,
transform=transform, download=True)
dataloader = DataLoader(dataset, batch_size=batch_size,
shuffle=True, num_workers=12)

# Device setup
device = torch.device("cuda" if torch.cuda.is_available() else
"cpu")

# Noise Schedule
def linear_noise_schedule(timesteps):
    return torch.linspace(1e-4, 0.02, timesteps, device=device)

betas = linear_noise_schedule(timesteps)
```

```
alphas = 1 - betas
alphas_cumprod = torch.cumprod(alphas, axis=0).to(device)

# Latent Diffusion Process
def forward_diffusion(latent, t):
    noise = torch.randn_like(latent)
    alpha_t = alphas_cumprod[t].view(-1, 1).to(latent.device)
    noisy_latent = torch.sqrt(alpha_t) * latent + torch.sqrt(1 - alpha_t) * noise
    return noisy_latent, noise

# Simple Latent Model (UNet-like)
class LatentDenoisingModel(nn.Module):
    def __init__(self, latent_dim, input_dim):
        super(LatentDenoisingModel, self).__init__()
        self.encoder = nn.Sequential(
            nn.Linear(input_dim, 512),
            nn.ReLU(),
            nn.Linear(512, latent_dim),
            nn.ReLU()
        )
        self.decoder = nn.Sequential(
            nn.Linear(latent_dim, 512),
            nn.ReLU(),
            nn.Linear(512, input_dim)
        )

    def forward(self, x, t):
        t_embedding = t.view(-1, 1).expand_as(x)
        x = x + t_embedding
        latent = self.encoder(x)
        return self.decoder(latent)

# Initialize Model and Optimizer
input_dim = image_size * image_size
model = LatentDenoisingModel(latent_dim, input_dim).to(device)
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
criterion = nn.MSELoss()

# Training Loop
for epoch in range(epochs):
    for batch, (x, _) in enumerate(dataloader):
        x = x.view(x.size(0), -1).to(device)  # Flatten MNIST images to vectors
        t = torch.randint(0, timesteps, (x.size(0),), device=device).long()

        noisy_latent, noise = forward_diffusion(x, t)
```

```
        predicted_noise = model(noisy_latent, t)

        loss = criterion(predicted_noise, noise)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    print(f"Epoch [{epoch + 1}/{epochs}] Loss:
{loss.item():.4f}")

# Generate Images
def reverse_diffusion(latent, model, timesteps):
    for t in range(timesteps - 1, -1, -1):
        with torch.no_grad():
            noise_pred = model(latent, torch.tensor([t],
device=latent.device).long())
            alpha_t = alphas_cumprod[t].view(-1,
1).to(latent.device)
            beta_t = betas[t].view(-1, 1).to(latent.device)
            latent = (latent - torch.sqrt(1 - alpha_t) *
noise_pred) / torch.sqrt(alpha_t)
            if t > 0:
                latent += torch.sqrt(beta_t) *
torch.randn_like(latent)
    return latent

# Test Stable Diffusion
test_noise = torch.randn((16, input_dim)).to(device)
generated_images = reverse_diffusion(test_noise, model,
timesteps)

# Reshape and Save
generated_images = generated_images.view(-1, 1, image_size,
image_size)
save_image(generated_images, "generated_stable_diffusion.png")
print("Generated images saved as
'generated_stable_diffusion.png'")

```

Penjelasan kode

- **Latent Space**:
    - Stable Diffusion bekerja dalam ruang laten (compressed representation) untuk mempercepat proses generasi.
- **Forward Diffusion**:

    - Menambahkan noise Gaussian ke vektor laten zzz, yang merupakan representasi gambar.

- **Latent Denoising Model**:
    - Model sederhana untuk memprediksi noise di ruang laten. Model ini menerima input noisy laten ztz_tzt dan waktu ttt.

- **Reverse Diffusion**:
    - Membalikkan proses difusi untuk menghasilkan kembali vektor laten z0z_0z0. Vektor ini kemudian di-decode ke bentuk gambar.

- **Visualisasi**:
    - Gambar yang dihasilkan disimpan sebagai file PNG untuk memverifikasi hasil.

    - **SECTION 7**

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
from torchvision.utils import save_image

# Hyperparameters
image_size = 28  # Ukuran gambar MNIST
input_dim = image_size * image_size
latent_dim = 128
num_classes = 10  # Jumlah kelas (MNIST: 0-9)
batch_size = 64
epochs = 20
timesteps = 100
learning_rate = 1e-4

# Dataset MNIST
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])
dataset = datasets.MNIST(root='./data', train=True,
transform=transform, download=True)
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)

# Device setup
```

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Noise Schedule
def linear_noise_schedule(timesteps):
    return torch.linspace(1e-4, 0.02, timesteps, device=device)

betas = linear_noise_schedule(timesteps)
alphas = 1 - betas
alphas_cumprod = torch.cumprod(alphas, axis=0).to(device)

# Conditional Diffusion Model
class ConditionalDiffusionModel(nn.Module):
    def __init__(self, input_dim, latent_dim, num_classes):
        super(ConditionalDiffusionModel, self).__init__()
        self.label_embedding = nn.Embedding(num_classes, latent_dim)
        self.encoder = nn.Sequential(
            nn.Linear(input_dim, 512),
            nn.ReLU(),
            nn.Linear(512, latent_dim)
        )
        self.model = nn.Sequential(
            nn.Linear(latent_dim + 1, 256),  # Tambahkan dimensi waktu
            nn.ReLU(),
            nn.Linear(256, 256),
            nn.ReLU(),
            nn.Linear(256, latent_dim)
        )
        self.decoder = nn.Sequential(
            nn.Linear(latent_dim, 512),
            nn.ReLU(),
            nn.Linear(512, input_dim)  # Kembalikan ke dimensi asli
        )

    def forward(self, x, t, labels):
        label_embed = self.label_embedding(labels)  # Embedding untuk
label kelas
        encoded_x = self.encoder(x)  # Encode input gambar ke
latent_dim
        t_embed = t.view(-1, 1).expand(x.size(0), 1)  # Expand waktu
untuk batch size
        z = torch.cat([encoded_x + label_embed, t_embed], dim=1)  #
Gabungkan latent + waktu
        latent = self.model(z)
        return self.decoder(latent)  # Decode kembali ke dimensi input

# Forward Diffusion
def forward_diffusion(x, t):
    noise = torch.randn_like(x)
```

```
    alpha_t = alphas_cumprod[t].view(-1, 1).to(x.device)
    noisy_x = torch.sqrt(alpha_t) * x + torch.sqrt(1 - alpha_t) * noise
    return noisy_x, noise

# Initialize Model and Optimizer
model = ConditionalDiffusionModel(input_dim, latent_dim,
num_classes).to(device)
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
criterion = nn.MSELoss()

# Training Loop
for epoch in range(epochs):
    for batch, (x, labels) in enumerate(dataloader):
        x = x.view(x.size(0), -1).to(device)  # Flatten MNIST images
        labels = labels.to(device)
        t = torch.randint(0, timesteps, (x.size(0),),
device=device).long()

        noisy_x, noise = forward_diffusion(x, t)
        predicted_noise = model(noisy_x, t, labels)

        loss = criterion(predicted_noise, noise)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    print(f"Epoch [{epoch + 1}/{epochs}] Loss: {loss.item():.4f}")

# Reverse Diffusion with Control
def reverse_diffusion(z, model, timesteps, labels):
    for t in range(timesteps - 1, -1, -1):
        with torch.no_grad():
            t_tensor = torch.tensor([t],
device=z.device).repeat(z.size(0)).long()  # Perbaiki dimensi waktu
            noise_pred = model(z, t_tensor, labels)
            alpha_t = alphas_cumprod[t].view(-1, 1).to(z.device)
            beta_t = betas[t].view(-1, 1).to(z.device)
            z = (z - torch.sqrt(1 - alpha_t) * noise_pred) /
torch.sqrt(alpha_t)
            if t > 0:
                z += torch.sqrt(beta_t) * torch.randn_like(z)
    return z

# Generate Controlled Images
test_noise = torch.randn((10, input_dim)).to(device)  # 10 samples for
10 classes
labels = torch.arange(0, 10).to(device)  # Class labels 0-9
```

```
generated_images = reverse_diffusion(test_noise, model, timesteps,
labels)

# Save Generated Images
generated_images = generated_images.view(-1, 1, image_size, image_size)
save_image(generated_images, "controlled_diffusion_fixed_v2.png")
print("Generated images saved as 'controlled_diffusion_fixed_v2.png'")
```

Penjelasan Controlled Diffusion Models

Controlled Diffusion Models merupakan pengembangan dari diffusion models yang memungkinkan generasi gambar diarahkan berdasarkan input tertentu, seperti label kelas. Pada implementasi ini, dataset MNIST digunakan untuk melatih model, dengan gambar (28 \times 28) yang dinormalisasi ke rentang ([-1, 1]).

1. Dataset dan Transformasi

Dataset MNIST diambil menggunakan torchvision.datasets dan diterapkan transformasi berikut:

1. ToTensor: Mengubah gambar menjadi tensor.
2. Normalize: Menormalisasi piksel gambar ke rentang ([-1, 1]).

Batch data diatur menggunakan DataLoader untuk meningkatkan efisiensi pelatihan.

2. Noise Schedule

Noise ditambahkan ke gambar secara bertahap menggunakan Linear Noise Schedule: [\beta_t \text{ (noise level)} \in [0.0001, 0.02]]

- (\alpha_t): Proporsi data asli yang dipertahankan, dihitung sebagai (1 - \beta_t).
- (\alpha_{cumprod}): Produk kumulatif (\alpha_t), digunakan dalam proses difusi maju.

3. Conditional Diffusion Model

Model dirancang untuk mengarahkan proses generasi berdasarkan label kelas. Model ini terdiri dari:

1. Label Embedding: Representasi vektor embedding untuk label kelas.
2. Encoder: Mengubah gambar asli ((x)) menjadi representasi ruang laten ((latent_dim)).
3. Latent Model: Memprediksi noise dalam ruang laten berdasarkan (x_t), waktu (t), dan label kelas.
4. Decoder: Mengembalikan prediksi dari ruang laten ke dimensi asli gambar ((28 \times 28 = 784)).

4. Forward Diffusion

Pada proses difusi maju, noise Gaussian (\epsilon) ditambahkan ke gambar asli (x_0) secara bertahap: [x_t = \sqrt{\alpha_t} x_0 + \sqrt{1 - \alpha_t} \epsilon] Hasilnya adalah (x_t), yaitu data noisy pada langkah (t), serta noise yang ditambahkan.

5. Reverse Diffusion

Proses difusi dibalik untuk menghilangkan noise secara bertahap dari data noisy (x_t), hingga menghasilkan gambar (x_0): [x_{t-1} = \frac{x_t - \sqrt{1 - \alpha_t} \hat{\epsilon}_t}{\sqrt{\alpha_t}}] di mana:

- (x_t): Data noisy pada langkah (t).
- (\hat{\epsilon}_t): Noise prediksi oleh model.

Proses ini menghasilkan data asli (x_0) atau gambar baru yang diarahkan oleh label kelas.

6. Training Loop

Pada setiap epoch, langkah-langkah berikut dilakukan:

1. Forward Diffusion: Noise ditambahkan ke gambar asli.
2. Model Denoising: Model memprediksi noise dari data noisy (x_t).
3. Loss Calculation: Menggunakan Mean Squared Error (MSE) untuk membandingkan noise prediksi dengan noise sebenarnya.
4. Optimization: Model diperbarui berdasarkan gradien dari loss.

7. Hasil Generasi

Gambar baru dihasilkan dari noise Gaussian acak ((z_T)), diarahkan oleh label kelas tertentu. Gambar ini disimpan dalam file "controlled_diffusion_fixed.png", dengan setiap gambar sesuai dengan label kelas (0-9).

Kesimpulan

Model Controlled Diffusion memungkinkan kontrol eksplisit pada proses generasi gambar. Dengan menambahkan embedding label dan waktu, model dapat diarahkan untuk menghasilkan gambar yang sesuai dengan distribusi data tertentu, seperti angka pada MNIST.

- **SECTION 8**

```
from diffusers import StableDiffusionPipeline
import torch
import matplotlib.pyplot as plt

# Menggunakan pipeline dari diffusers untuk menghasilkan gambar
pipe = StableDiffusionPipeline.from_pretrained("CompVis/stable-
diffusion-v1-4-original", torch_dtype=torch.float32)

# Tentukan perangkat (GPU jika tersedia, jika tidak menggunakan CPU)
pipe.to("cuda" if torch.cuda.is_available() else "cpu")

# Prompt yang digunakan untuk menghasilkan gambar
```

```
prompt = "a futuristic city skyline"

# Menghasilkan gambar dari prompt
generated_image = pipe(prompt).images[0]

# Menampilkan gambar yang dihasilkan
plt.imshow(generated_image)
plt.title('Generated Image')
plt.axis('off')
plt.show()
```

Penjelasan:

1. Install Dependencies: Pertama, Anda perlu menginstal pustaka diffusers, torch, dan transformers untuk mengakses dan menggunakan model Stable Diffusion.
2. Pipeline Stable Diffusion: Menggunakan StableDiffusionPipeline dari diffusers untuk memuat model Stable Diffusion dan menghasilkan gambar berdasarkan prompt teks.
3. Perangkat GPU/CPU: Menentukan perangkat yang akan digunakan (GPU jika tersedia, jika tidak menggunakan CPU).
4. Prompt: Memberikan deskripsi teks sebagai prompt untuk menghasilkan gambar.
5. Menampilkan Gambar: Gambar yang dihasilkan ditampilkan menggunakan matplotlib.

UNIT 6

- SECTION 1

```
import requests
from PIL import Image
import torch
import torchvision
from torchvision.transforms import functional as F
import matplotlib.pyplot as plt

# URL gambar untuk diunduh
image_url =
"https://cdn.pixabay.com/photo/2017/10/06/22/27/communication-
2824850_1280.jpg"  # Contoh URL gambar
output_image_path = "downloaded_image.jpg"

# Fungsi untuk mendownload gambar dari URL
def download_image(url, save_path):
    """
    Mendownload gambar dari URL yang diberikan.
    Parameter:
```

```
        url: URL gambar.
        save_path: Path untuk menyimpan gambar yang diunduh.
    """
    response = requests.get(url, stream=True)
    if response.status_code == 200:
        with open(save_path, 'wb') as image_file:
            for chunk in response.iter_content(chunk_size=1024):
                image_file.write(chunk)
        print(f"Gambar berhasil diunduh ke {save_path}")
    else:
        print(f"Gagal mengunduh gambar. Status kode:
{response.status_code}")

# Mendownload gambar
download_image(image_url, output_image_path)

# Memuat model deteksi objek
model =
torchvision.models.detection.fasterrcnn_resnet50_fpn(pretrained=T
rue)
model.eval()

# Fungsi untuk memuat gambar dan melakukan transformasi
def load_image(image_path):
    image = Image.open(image_path).convert("RGB")  # Membuka
gambar dan mengonversinya ke RGB
    image_tensor = F.to_tensor(image)  # Mengubah gambar menjadi
tensor
    return image, image_tensor.unsqueeze(0)  # Mengembalikan
gambar asli dan tensor yang diubah

# Memuat gambar yang diunduh
original_image, image_tensor = load_image(output_image_path)

# Melakukan prediksi pada gambar
with torch.no_grad():
    predictions = model(image_tensor)[0]

# Menampilkan hasil deteksi
def display_detections(image, predictions, threshold=0.5):
    plt.imshow(image)
    ax = plt.gca()

    for idx, score in enumerate(predictions["scores"]):
        if score > threshold:
            bbox = predictions["boxes"][idx].tolist()
            label = predictions["labels"][idx].item()
            ax.add_patch(plt.Rectangle(
```

```
                (bbox[0], bbox[1]), bbox[2] - bbox[0], bbox[3] -
bbox[1],
                edgecolor="red", fill=False, linewidth=2
            ))
            ax.text(bbox[0], bbox[1], f"Label: {label}, Score:
{score:.2f}",
                    bbox=dict(facecolor="yellow", alpha=0.5),
fontsize=10, color="black")

    plt.axis("off")
    plt.show()

# Menampilkan hasil deteksi objek
display_detections(original_image, predictions)
```

1. **Mendownload Gambar dari URL**

   - **Tujuan**: Bagian ini bertanggung jawab untuk mendownload gambar dari URL yang diberikan dan menyimpannya ke file lokal (downloaded_image.jpg).
   - **Fungsi**: download_image menerima dua parameter: url (URL gambar yang ingin didownload) dan save_path (path tempat menyimpan gambar). Menggunakan requests.get() untuk mengambil gambar, kemudian menyimpannya ke disk jika status kode HTTP menunjukkan berhasil (200).

2. **Memuat dan Menyiapkan Model untuk Deteksi Objek**

   - **Tujuan**: Bagian ini memuat model deteksi objek **Faster R-CNN** yang telah dilatih sebelumnya (pretrained). Model ini digunakan untuk mendeteksi objek dalam gambar.
   - **Penjelasan**: Model yang digunakan adalah fasterrcnn_resnet50_fpn, yang merupakan model deteksi objek berbasis ResNet50 dengan Feature Pyramid Network (FPN). Dengan menggunakan pretrained=True, model ini sudah dilatih pada dataset COCO dan siap digunakan untuk prediksi.

3. **Memuat dan Mengonversi Gambar**

   - **Tujuan**: Fungsi ini membuka gambar dari disk, mengonversinya ke format RGB (untuk memastikan gambar memiliki tiga saluran warna), dan mengubahnya menjadi tensor yang dapat diterima oleh model.
   - **Proses**:
     - Image.open(image_path) membuka gambar dari path yang diberikan.
     - .convert("RGB") memastikan gambar memiliki tiga saluran warna.
     - F.to_tensor(image) mengubah gambar menjadi tensor, yang diperlukan untuk pemrosesan dengan PyTorch. Fungsi unsqueeze(0) menambahkan dimensi batch, yang diperlukan oleh model (model menerima input dalam bentuk batch).

4. **Melakukan Prediksi Menggunakan Model**

- **Tujuan**: Melakukan prediksi menggunakan model Faster R-CNN pada gambar yang telah diubah menjadi tensor.
- **Penjelasan**:
    - torch.no_grad() digunakan untuk menonaktifkan penghitungan gradien, karena kita hanya melakukan inferensi dan tidak perlu menghitung gradien.
    - model(image_tensor) memberikan output prediksi untuk gambar yang dimasukkan. Outputnya adalah dictionary yang berisi berbagai informasi, seperti boxes (koordinat bounding boxes), labels (label objek yang terdeteksi), dan scores (kepercayaan model terhadap deteksi).

5. **Menampilkan Hasil Deteksi Objek**

- **Tujuan**: Fungsi ini bertanggung jawab untuk menampilkan hasil deteksi objek dalam gambar.
- **Proses**:
    - Gambar ditampilkan dengan plt.imshow().
    - Iterasi dilakukan pada setiap deteksi dalam prediksi, dan jika skor deteksi melebihi ambang batas (default 0.5), bounding box digambar di atas objek yang terdeteksi.
    - Setiap bounding box diberi label yang berisi Label dan Score, serta ditampilkan di atas gambar.
    - Gambar hasil deteksi ditampilkan menggunakan plt.show().

6. **Menampilkan Hasil Deteksi**

- **Tujuan**: Memanggil fungsi display_detections untuk menampilkan gambar dengan bounding box dan label pada objek yang terdeteksi.
- **Penjelasan**: Fungsi ini menerima gambar asli dan hasil prediksi, kemudian menampilkan hasil deteksi objek.

- **SECTION 2**

```
import torch
import torchvision
from torchvision import transforms
from PIL import Image
import matplotlib.pyplot as plt

# Memuat model DeepLabV3 pre-trained
model =
torchvision.models.segmentation.deeplabv3_resnet101(pretrained=True)
```

```
model.eval()

# Fungsi untuk memuat dan mengubah gambar menjadi tensor
def load_image(image_path):
    image = Image.open(image_path).convert("RGB")
    transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229,
0.224, 0.225])  # Normalisasi sesuai ImageNet
    ])
    image_tensor = transform(image).unsqueeze(0)  # Menambahkan dimensi
batch
    return image, image_tensor

# Membaca gambar input
image_path = "downloaded_image.jpg"  # Ganti dengan path gambar Anda
original_image, image_tensor = load_image(image_path)

# Melakukan segmentasi gambar dengan model
with torch.no_grad():
    output = model(image_tensor)['out'][0]  # Mendapatkan output dari
model
    output_predictions = output.argmax(0)  # Mengambil kelas dengan
probabilitas tertinggi

# Menampilkan hasil segmentasi
def display_segmentation(image, segmentation_map):
    plt.figure(figsize=(10, 10))
    plt.imshow(image)
    plt.imshow(segmentation_map, alpha=0.5)  # Overlay hasil segmentasi
dengan transparansi
    plt.axis('off')
    plt.show()

# Menampilkan gambar dengan overlay segmentasi
display_segmentation(original_image, output_predictions)
```

Penjelasan Kode:

1. Memuat Model DeepLabV3:
    - Menggunakan model deeplabv3_resnet101 dari torchvision yang telah dilatih sebelumnya pada dataset ImageNet.
    - model.eval() digunakan untuk mengatur model ke mode evaluasi sehingga dapat digunakan untuk inferensi.
2. Memuat dan Mengonversi Gambar:

- Gambar dimuat dengan PIL.Image.open() dan diubah menjadi format RGB untuk memastikan gambar memiliki tiga saluran warna.
- Gambar diubah menjadi tensor dengan transforms.ToTensor() dan dinormalisasi menggunakan transforms.Normalize() agar sesuai dengan data yang digunakan saat pelatihan model (ImageNet).

3. Melakukan Segmentasi:
   - output = model(image_tensor)['out'][0]: Model menghasilkan output berupa peta segmentasi untuk setiap piksel dalam gambar.
   - output.argmax(0) digunakan untuk mengambil kelas dengan probabilitas tertinggi di setiap piksel, yang memberikan peta segmentasi.
4. Menampilkan Hasil Segmentasi:
   - Fungsi display_segmentation() menampilkan gambar asli dengan overlay peta segmentasi yang ditampilkan dengan transparansi.
   - plt.imshow(segmentation_map, alpha=0.5) digunakan untuk menampilkan hasil segmentasi dengan transparansi, sehingga memungkinkan kita melihat gambar asli di bawah segmentasi.

UNIT 7

- **SECTION 1**

```
import imageio
import numpy as np
import cv2
import matplotlib.pyplot as plt

# URL video yang akan diambil (URL video yang valid)
video_url = "https://www.sample-
videos.com/video321/mp4/720/big_buck_bunny_720p_1mb.mp4"  # Ganti
dengan URL video yang valid

# Membaca video dari URL menggunakan imageio
reader = imageio.get_reader(video_url)

# Mendapatkan informasi tentang video (misalnya, frame rate dan ukuran
frame)
fps = reader.get_meta_data()['fps']  # Frame rate video
frame_width = reader.get_meta_data()['size'][0]  # Lebar frame
frame_height = reader.get_meta_data()['size'][1]  # Tinggi frame

print(f"FPS: {fps}, Frame width: {frame_width}, Frame height:
{frame_height}")

# Membaca dan memproses video frame by frame
for frame in reader:
```

```
    # Frame yang dibaca oleh imageio adalah array numpy (RGB)
    frame_rgb = np.array(frame)

    # Menambahkan teks pada frame
    font = cv2.FONT_HERSHEY_SIMPLEX
    frame_rgb = cv2.putText(frame_rgb, "Video Processing", (50, 50),
font, 1, (255, 0, 0), 2, cv2.LINE_AA)

    # Menampilkan frame menggunakan matplotlib
    plt.imshow(frame_rgb)
    plt.axis('off')
    plt.show()

    # Tunggu sebentar sesuai frame rate
    cv2.waitKey(int(1000 / fps))

# Setelah selesai, tutup jendela
cv2.destroyAllWindows()
```

Penjelasan code

1. Import Pustaka yang Diperlukan

- imageio: Pustaka untuk membaca video dari URL dan memprosesnya frame-by-frame. imageio mendukung berbagai format file video dan sumber video dari URL atau file lokal.
- numpy: Digunakan untuk memanipulasi frame video dalam bentuk array numpy.
- cv2: Pustaka OpenCV digunakan untuk menambahkan teks pada frame video.
- matplotlib.pyplot: Digunakan untuk menampilkan setiap frame dari video di dalam notebook menggunakan matplotlib.

2. Menentukan URL Video

- Di sini, kita menentukan URL video yang akan diambil untuk diproses. Video yang digunakan adalah **Big Buck Bunny 720p 1MB**, sebuah video contoh dengan kualitas rendah yang dapat diakses secara bebas.

3. Membaca Video dari URL Menggunakan imageio

- **imageio.get_reader(video_url)**: Fungsi ini membuka video dari URL yang diberikan dan mengembalikan objek pembaca video (reader). Objek ini memungkinkan kita untuk membaca setiap frame dari video secara terpisah.

4. Mendapatkan Metadata Video (FPS, Dimensi Frame)

- **reader.get_meta_data()**: Mengambil metadata dari video, yang mencakup informasi seperti **frame rate (FPS)**, **lebar frame** (width), dan **tinggi frame** (height).
- **fps**: Mengambil frame rate video yang menentukan seberapa cepat video diputar.
- **frame_width dan frame_height**: Mengambil dimensi (lebar dan tinggi) dari setiap frame video.

5. Membaca dan Memproses Video Frame by Frame

- **for frame in reader**: Ini adalah loop yang membaca video frame per frame. Setiap frame yang dibaca oleh imageio adalah dalam bentuk **array numpy** dengan format **RGB**.
- **np.array(frame)**: Mengonversi frame video yang dibaca menjadi array numpy agar bisa dimodifikasi.

6. Menambahkan Teks pada Setiap Frame

- **cv2.putText**: Fungsi ini digunakan untuk menambahkan teks ke frame video. Teks "Video Processing" akan ditambahkan pada setiap frame.
- **Posisi (50, 50)**: Menentukan posisi teks di dalam frame.
- **font**: Jenis font yang digunakan, di sini kita menggunakan **FONT_HERSHEY_SIMPLEX**.
- **(255, 0, 0)**: Warna teks dalam format RGB (merah di sini).
- **2**: Ketebalan garis teks.
- **cv2.LINE_AA**: Menentukan tipe garis (anti-aliasing).

7. Menampilkan Frame Menggunakan Matplotlib

- **plt.imshow(frame_rgb)**: Menampilkan frame yang telah diproses (dengan teks) menggunakan matplotlib.
- **plt.axis('off')**: Menonaktifkan sumbu agar hanya gambar yang ditampilkan, tanpa label sumbu.
- **plt.show()**: Memanggil fungsi ini untuk menampilkan frame di dalam notebook.

8. Menunggu Sebentar Sebelum Menampilkan Frame Berikutnya

- **cv2.waitKey(int(1000 / fps))**: Fungsi ini digunakan untuk menunggu selama interval waktu yang sesuai dengan frame rate video, sehingga frame ditampilkan dengan kecepatan yang benar.

UNIT 8

- **SECTION 1**

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
import requests
from PIL import Image
from io import BytesIO

# URL gambar stereo (kiri dan kanan)
left_image_url =
"https://raw.githubusercontent.com/opencv/opencv/master/samples/data/al
oeL.jpg"
```

```
right_image_url =
"https://raw.githubusercontent.com/opencv/opencv/master/samples/data/al
oeR.jpg"

# Fungsi untuk mengunduh gambar dari URL
def download_image(url):
    """
    Mengunduh gambar dari URL dan mengonversinya menjadi array
grayscale.
    """
    response = requests.get(url)
    if response.status_code == 200:
        img = Image.open(BytesIO(response.content))
        return cv2.cvtColor(np.array(img), cv2.COLOR_RGB2GRAY)  #
Mengonversi ke grayscale
    else:
        print(f"Error: Tidak dapat mengunduh gambar dari {url}")
        return None

# Mengunduh gambar stereo dari URL
left_image = download_image(left_image_url)
right_image = download_image(right_image_url)

# Memeriksa apakah gambar berhasil diunduh
if left_image is None or right_image is None:
    print("Gambar tidak tersedia. Harap periksa URL.")
else:
    # Menampilkan gambar stereo
    plt.figure(figsize=(12, 6))
    plt.subplot(1, 2, 1)
    plt.title("Left Image")
    plt.imshow(left_image, cmap="gray")
    plt.subplot(1, 2, 2)
    plt.title("Right Image")
    plt.imshow(right_image, cmap="gray")
    plt.show()

    # Membuat StereoBM (Block Matching) untuk Depth Estimation
    stereo = cv2.StereoBM_create(numDisparities=16, blockSize=15)
    disparity = stereo.compute(left_image, right_image)

    # Menormalkan peta kedalaman untuk visualisasi
    disparity_normalized = cv2.normalize(disparity, None, alpha=0,
beta=255, norm_type=cv2.NORM_MINMAX, dtype=cv2.CV_8U)

    # Menampilkan peta kedalaman
    plt.figure(figsize=(10, 6))
    plt.title("Disparity Map (Depth Map)")
```

```
    plt.imshow(disparity_normalized, cmap="plasma")
    plt.colorbar()
    plt.show()
```

Penjelasan Kode

1. Import Pustaka

- cv2: Pustaka OpenCV untuk pengolahan gambar.
- numpy: Untuk manipulasi array gambar.
- matplotlib.pyplot: Untuk visualisasi gambar dan peta kedalaman.
- requests: Untuk mengunduh gambar langsung dari URL.
- PIL.Image: Untuk membuka gambar dari byte yang diunduh.

2. Fungsi Mengunduh Gambar

- requests.get(url): Mengunduh gambar dari URL.
- Image.open(BytesIO(response.content)): Membuka gambar dari byte yang diunduh.
- cv2.cvtColor(): Mengonversi gambar ke grayscale, karena algoritma StereoBM bekerja dengan gambar grayscale.

3. Mengunduh Gambar Stereo

- left_image_url dan right_image_url: URL gambar stereo kiri dan kanan.
- Hasil: Gambar kiri dan kanan disimpan dalam format grayscale sebagai array numpy.

4. Menampilkan Gambar Stereo

- Menampilkan kedua gambar stereo (kiri dan kanan) secara berdampingan.

5. Stereo Depth Estimation

- cv2.StereoBM_create(): Membuat algoritma StereoBM untuk menghitung peta kedalaman.
- numDisparities=16: Rentang disparitas (harus kelipatan 16).
- blockSize=15: Ukuran blok untuk mencocokkan piksel antara gambar stereo.
- stereo.compute(left_image, right_image): Menghitung peta disparitas (depth map) dari gambar stereo.

6. Normalisasi dan Menampilkan Depth Map

- cv2.normalize(): Menormalkan nilai disparitas untuk meningkatkan visualisasi.
- alpha=0 dan beta=255: Menentukan rentang normalisasi (0-255 untuk visualisasi).
- norm_type=cv2.NORM_MINMAX: Normalisasi berdasarkan nilai minimum dan maksimum dalam array.
- plt.imshow(): Menampilkan peta kedalaman menggunakan skema warna plasma.
- plt.colorbar(): Menambahkan bar warna untuk menunjukkan skala kedalaman.

- **SECTION 2**
  Sejarah Singkat 3D Vision
  3D Vision adalah bidang ilmu yang berfokus pada cara komputer memahami dunia dalam bentuk tiga dimensi, mirip seperti bagaimana manusia melihat dunia menggunakan kedua matanya. Perjalanan perkembangan 3D Vision dimulai dari teori

dasar di abad ke-15 hingga menjadi teknologi yang sangat canggih seperti yang kita lihat sekarang. Berikut adalah kisah singkat tentang bagaimana 3D Vision berkembang:

1. Awal Mula: Seni dan Geometri Proyeksi (Abad ke-15)

- Perspektif dalam Seni:
    - Di masa Renaisans, seniman seperti Leonardo da Vinci mulai memahami bagaimana menciptakan ilusi kedalaman pada lukisan datar menggunakan perspektif. Inilah awal mula pemahaman tentang dunia 3D.
- Kamera Lubang Jarum (Pinhole Camera):
    - Pada dasarnya, ini adalah konsep sederhana bagaimana cahaya dari dunia nyata diproyeksikan ke permukaan datar, yang kemudian menjadi fondasi teori kamera modern.

2. Fotogrametri: Memahami Dunia Lewat Foto (Abad ke-19)

- Fotogrametri adalah teknik yang menggunakan foto untuk mengukur jarak dan membangun model dunia nyata.
- Teknologi ini banyak digunakan dalam kartografi (pemetaan) dan survei geografis, di mana gambar dari udara digunakan untuk menciptakan peta 3D permukaan bumi.

3. Era Komputer: Langkah Awal dalam Algoritma (1960-an - 1970-an)

- Algoritma untuk Gambar Stereo:
    - Ketika komputer mulai berkembang, peneliti mencoba menggunakan dua gambar (dari sudut pandang berbeda) untuk memahami kedalaman dan struktur objek.
- David Marr dan Teori Visual:
    - David Marr, seorang ilmuwan komputer terkenal, mempelajari bagaimana manusia memahami informasi visual. Ia mencoba mereplikasi proses ini pada komputer, yang menjadi dasar teori penglihatan komputer modern.

4. Kamera Stereo dan Gerakan: Era Praktis (1980-an)

- Kamera Stereo:
    - Kamera stereo adalah dua kamera yang ditempatkan berdekatan, seperti mata manusia. Dengan membandingkan perbedaan antara dua gambar, komputer dapat menghitung kedalaman objek.
- Structure from Motion (SfM):
    - Algoritma ini memungkinkan komputer membangun model 3D dari serangkaian gambar atau video. Misalnya, jika Anda mengambil foto dari berbagai sudut, algoritma ini dapat menyatukannya menjadi model 3D.

5. Peta 3D dan SLAM: Membangun Dunia Virtual (1990-an - 2000-an)

- Peta 3D:
    - Peneliti mulai membuat algoritma untuk menghasilkan peta 3D menggunakan banyak gambar sekaligus. Buku seperti *Multiple View Geometry in Computer Vision* menjadi panduan utama di bidang ini.

- SLAM (Simultaneous Localization and Mapping):
    - Robot dan drone menggunakan teknologi ini untuk memetakan lingkungan mereka sambil menentukan posisi mereka sendiri. Ini seperti memberi robot kemampuan "melihat dan mengingat" lingkungan.

6. Sensor 3D: Mata Baru untuk Komputer (2010-an)
- Sensor Kedalaman:
    - Sensor seperti Microsoft Kinect membawa teknologi 3D Vision ke banyak aplikasi, mulai dari game hingga penelitian medis.
- LiDAR dan Time-of-Flight:
    - Teknologi ini digunakan untuk mendeteksi jarak dengan memantulkan sinar laser ke objek. Mobil otonom, seperti Tesla, menggunakan sensor ini untuk memahami jalan dan lingkungan di sekitarnya.

7. Era AI: Memadukan 3D Vision dengan Pembelajaran Mesin (2015 - Sekarang)
- Jaringan Saraf untuk 3D:
    - Dengan deep learning, komputer kini dapat memperkirakan kedalaman dari satu gambar saja, tanpa memerlukan gambar stereo.
- Point Clouds dan Data Voxel:
    - Data 3D sering direpresentasikan sebagai kumpulan titik (point cloud) atau blok kecil (voxel). Algoritma modern seperti PointNet digunakan untuk memproses data ini.

8. Aplikasi Modern
- Mobil Otonom:
    - Mobil tanpa pengemudi menggunakan 3D Vision untuk mendeteksi rintangan, pejalan kaki, dan jalan raya.
- Robotika:
    - Robot modern menggunakan peta 3D untuk bernavigasi di dunia nyata.
- Gaming dan AR/VR:
    - Teknologi ini digunakan untuk menciptakan dunia virtual yang sangat realistis dalam game dan perangkat seperti Oculus Rift.
- Medis:
    - Dalam pencitraan medis, 3D Vision membantu dokter melihat organ tubuh dengan lebih jelas melalui CT scan atau MRI.
- Pelestarian Budaya:
    - Monumen dan bangunan bersejarah dapat direkonstruksi dalam bentuk digital 3D untuk melestarikan warisan budaya.

Kesimpulan
3D Vision telah berkembang dari teori sederhana tentang perspektif menjadi teknologi canggih yang digunakan dalam kehidupan sehari-hari. Dari kamera stereo hingga mobil otonom, dari game VR hingga rekonstruksi budaya, 3D Vision telah menjadi bagian tak terpisahkan dari dunia modern. Dengan perkembangan perangkat keras dan pembelajaran mesin, masa depan 3D Vision akan semakin menjanjikan, membawa lebih banyak inovasi di berbagai bidang.

- **SECTION 3**

```
import numpy as np
import matplotlib.pyplot as plt

# Definisi parameter kamera
focal_length = 50  # Panjang fokus (dalam satuan mm atau piksel)
image_width = 640  # Lebar gambar (piksel)
image_height = 480  # Tinggi gambar (piksel)
principal_point = (image_width / 2, image_height / 2)  # Titik pusat
gambar

# Matriks intrinsik kamera (3x3)
K = np.array([
    [focal_length, 0, principal_point[0]],  # Baris 1
    [0, focal_length, principal_point[1]],  # Baris 2
    [0, 0, 1]                               # Baris 3
])

print("Matriks Intrinsik Kamera (K):")
print(K)

# Titik 3D di dunia nyata (x, y, z)
points_3D = np.array([
    [100, 200, 1000],  # Titik pertama
    [-50, 150, 1200],  # Titik kedua
    [300, -100, 1500], # Titik ketiga
    [0, 0, 800]        # Titik keempat
])

print("\nTitik 3D di Dunia Nyata:")
print(points_3D)

# Memproyeksikan titik 3D ke gambar 2D menggunakan model kamera
points_2D = []
for point in points_3D:
    # Menggunakan persamaan: u = (f * x / z) + cx, v = (f * y / z) + cy
    x, y, z = point
    u = (focal_length * x / z) + principal_point[0]
    v = (focal_length * y / z) + principal_point[1]
    points_2D.append([u, v])

points_2D = np.array(points_2D)

print("\nTitik 2D di Gambar (Setelah Proyeksi):")
print(points_2D)

# Visualisasi titik 3D dan hasil proyeksi 2D
```

```
plt.figure(figsize=(12, 6))

# Plot titik 3D
ax = plt.subplot(1, 2, 1, projection='3d')
ax.scatter(points_3D[:, 0], points_3D[:, 1], points_3D[:, 2], c='r',
label='Titik 3D')
ax.set_xlabel("X (mm)")
ax.set_ylabel("Y (mm)")
ax.set_zlabel("Z (mm)")
ax.set_title("Titik 3D di Dunia Nyata")
ax.legend()

# Plot titik 2D
plt.subplot(1, 2, 2)
plt.scatter(points_2D[:, 0], points_2D[:, 1], c='b', label='Titik 2D')
plt.gca().invert_yaxis()  # Membalikkan sumbu Y (format gambar)
plt.xlabel("u (piksel)")
plt.ylabel("v (piksel)")
plt.title("Titik 2D di Gambar (Proyeksi)")
plt.legend()

plt.tight_layout()
plt.show()
```

Penjelasan Kode

1. Definisi Parameter Kamera:
    - Focal length: Jarak antara pusat kamera dan bidang gambar (diukur dalam satuan seperti mm atau piksel).
    - Image width dan height: Resolusi gambar dalam piksel.
    - Principal point: Titik pusat kamera dalam gambar (biasanya di tengah gambar).
2. Matriks Intrinsik Kamera (K):
    - Matriks intrinsik adalah matriks 3x3 yang menghubungkan koordinat dunia 3D ke koordinat gambar 2D.
    - Elemen-elemen matriks intrinsik:
        - [0, 0]: Panjang fokus dalam sumbu X.
        - [1, 1]: Panjang fokus dalam sumbu Y.
        - [0, 2]: Posisi pusat gambar di sumbu X.
        - [1, 2]: Posisi pusat gambar di sumbu Y.
3. Titik 3D:

- Titik-titik 3D di dunia nyata diberikan dalam koordinat (x, y, z).

4. Proyeksi ke Titik 2D:
    - Titik-titik 3D diproyeksikan ke gambar 2D menggunakan persamaan pinhole camera model: [u = \frac{f \cdot x}{z} + c_x] [v = \frac{f \cdot y}{z} + c_y]
    - Di sini, (z) adalah jarak dari kamera ke titik, (c_x) dan (c_y) adalah koordinat pusat gambar.
5. Visualisasi:
    - Titik 3D divisualisasikan dalam plot 3D menggunakan matplotlib.
    - Titik 2D divisualisasikan dalam bidang gambar (2D) dengan membalikkan sumbu Y agar menyerupai format gambar.

- **SECTION 4**

```
import numpy as np
import matplotlib.pyplot as plt

# Definisi titik 3D
points_3D = np.array([
    [1, 2, 3],  # Titik 1
    [4, 5, 6],  # Titik 2
    [7, 8, 9],  # Titik 3
    [2, 4, 6]   # Titik 4
])

print("Titik 3D Asli:")
print(points_3D)

# Fungsi untuk memvisualisasikan titik 3D
def plot_points_3D(points, title="Titik 3D"):
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    ax.scatter(points[:, 0], points[:, 1], points[:, 2], c='b',
marker='o', label='Titik')
    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')
    ax.set_title(title)
    ax.legend()
    plt.show()

# Visualisasi titik asli
plot_points_3D(points_3D, title="Titik 3D Asli")
```

```
# --- 1. Translasi ---
# Matriks translasi
translation_vector = np.array([3, -2, 5])  # Translasi di X, Y, Z
translated_points = points_3D + translation_vector

print("\nTitik 3D Setelah Translasi:")
print(translated_points)

# Visualisasi hasil translasi
plot_points_3D(translated_points, title="Titik 3D Setelah Translasi")

# --- 2. Rotasi ---
# Matriks rotasi di sekitar sumbu Z
theta = np.radians(45)  # Rotasi 45 derajat
rotation_matrix = np.array([
    [np.cos(theta), -np.sin(theta), 0],
    [np.sin(theta),  np.cos(theta), 0],
    [0,              0,             1]
])

# Melakukan rotasi
rotated_points = points_3D @ rotation_matrix.T

print("\nTitik 3D Setelah Rotasi di sekitar sumbu Z:")
print(rotated_points)

# Visualisasi hasil rotasi
plot_points_3D(rotated_points, title="Titik 3D Setelah Rotasi di
sekitar sumbu Z")

# --- 3. Transformasi Homogen ---
# Membuat matriks transformasi homogen
homogeneous_matrix = np.eye(4)  # Matriks identitas 4x4
homogeneous_matrix[:3, :3] = rotation_matrix  # Masukkan matriks rotasi
homogeneous_matrix[:3, 3] = translation_vector  # Masukkan vektor
translasi

# Menambahkan dimensi homogen ke titik 3D
points_3D_homogeneous = np.hstack((points_3D,
np.ones((points_3D.shape[0], 1))))

# Melakukan transformasi homogen
transformed_points_homogeneous = points_3D_homogeneous @
homogeneous_matrix.T
transformed_points = transformed_points_homogeneous[:, :3]  # Konversi
kembali ke koordinat 3D

print("\nTitik 3D Setelah Transformasi Homogen (Rotasi + Translasi):")
```

```
print(transformed_points)

# Visualisasi hasil transformasi homogen
plot_points_3D(transformed_points, title="Titik 3D Setelah Transformasi
Homogen")
```

Penjelasan Kode

- Titik 3D Asli:
  - Kumpulan titik didefinisikan dalam koordinat ((x, y, z)).
- Translasi:
  - Translasi menggeser titik 3D menggunakan vektor translasi ([dx, dy, dz]).
  - Vektor ditambahkan langsung ke setiap titik 3D.
- Rotasi:
  - Matriks rotasi untuk rotasi 45 derajat di sekitar sumbu Z digunakan.
  - Setiap titik 3D dikalikan dengan matriks rotasi untuk mendapatkan koordinat baru.
- Transformasi Homogen:
  - Matriks homogen 4x4 digunakan untuk menggabungkan translasi dan rotasi dalam satu langkah.
  - Titik 3D diubah menjadi bentuk homogen dengan menambahkan dimensi tambahan ((w = 1)).
  - Hasil transformasi dikonversi kembali ke koordinat 3D.
- Visualisasi:
  - Fungsi plot_points_3D digunakan untuk memvisualisasikan titik-titik dalam ruang 3D sebelum dan setelah setiap operasi (translasi, rotasi, transformasi homogen).

- **SECTION 5**

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d.art3d import Poly3DCollection

# --- 1. Point Cloud Representation ---
def visualize_point_cloud():
    np.random.seed(42)
    point_cloud = np.random.rand(100, 3) * 10  # 100 titik acak dalam
ruang 3D

    # Visualisasi Point Cloud
```

```
    fig = plt.figure(figsize=(10, 8))
    ax = fig.add_subplot(111, projection='3d')
    ax.scatter(point_cloud[:, 0], point_cloud[:, 1], point_cloud[:, 2],
c='b', marker='o', label='Point Cloud')
    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')
    ax.set_title("Point Cloud Representation")
    ax.legend()
    plt.show()

# --- 2. Voxel Grid Representation ---
def visualize_voxel_grid():
    voxel_grid = np.zeros((10, 10, 10))  # Membuat grid kosong 10x10x10
    voxel_grid[3, 3, 3] = 1  # Mengaktifkan satu voxel di (3, 3, 3)
    voxel_grid[6, 6, 6] = 1  # Mengaktifkan voxel lain di (6, 6, 6)

    # Visualisasi Voxel Grid
    fig = plt.figure(figsize=(10, 8))
    ax = fig.add_subplot(111, projection='3d')
    ax.voxels(voxel_grid, facecolors='blue', edgecolor='k', alpha=0.6)
    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')
    ax.set_title("Voxel Grid Representation")
    plt.show()

# --- 3. Mesh Representation ---
def visualize_mesh():
    vertices = np.array([
        [0, 0, 0], [1, 0, 0], [1, 1, 0], [0, 1, 0],  # Alas kubus
        [0, 0, 1], [1, 0, 1], [1, 1, 1], [0, 1, 1]   # Atap kubus
    ])

    edges = [
        [vertices[0], vertices[1], vertices[2], vertices[3]],  # Alas
        [vertices[4], vertices[5], vertices[6], vertices[7]],  # Atap
        [vertices[0], vertices[1], vertices[5], vertices[4]],  # Sisi
depan
        [vertices[2], vertices[3], vertices[7], vertices[6]],  # Sisi
belakang
        [vertices[1], vertices[2], vertices[6], vertices[5]],  # Sisi
kanan
        [vertices[0], vertices[3], vertices[7], vertices[4]]   # Sisi
kiri
    ]

    # Visualisasi Mesh
```

```
    fig = plt.figure(figsize=(10, 8))
    ax = fig.add_subplot(111, projection='3d')

    poly3d = [[edge for edge in edges]]
    ax.add_collection3d(Poly3DCollection(edges, facecolors='cyan',
linewidths=1, edgecolor='r', alpha=0.6))

    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')
    ax.set_title("Mesh Representation")
    plt.show()

# --- Menjalankan Visualisasi ---
print("Visualisasi Representasi Data 3D")
visualize_point_cloud()
visualize_voxel_grid()
visualize_mesh()
```

Penjelasan Kode

- Point Cloud Representation:
    - Menghasilkan 100 titik acak di ruang 3D menggunakan np.random.rand.
    - Titik-titik divisualisasikan menggunakan scatter plot 3D.
- Voxel Grid Representation:
    - Membuat grid 3D kosong (10x10x10) dengan nilai 0.
    - Mengaktifkan beberapa voxel (dengan nilai 1) untuk menunjukkan objek.
    - ax.voxels digunakan untuk memvisualisasikan voxel grid.
- Mesh Representation:
    - Membuat kubus menggunakan simpul (vertices) dan sisi (edges).
    - Poly3DCollection digunakan untuk membentuk mesh 3D dari simpul dan sisi kubus.
- Modularisasi:
    - Masing-masing representasi memiliki fungsi terpisah: visualize_point_cloud, visualize_voxel_grid, dan visualize_mesh.
    - Fungsi dijalankan satu per satu dalam urutan untuk memvisualisasikan ketiga representasi.

- **SECTION 6**

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Definisi posisi kamera (views yang diketahui)
camera_positions = np.array([
    [0, 0, 0],  # Kamera 1
    [10, 0, 0], # Kamera 2
    [0, 10, 0], # Kamera 3
])

# Titik 3D objek
object_points = np.array([
    [2, 2, 5],
    [4, 4, 5],
    [6, 2, 5],
    [3, 7, 5]
])

# Visualisasi pandangan awal dari semua kamera
def visualize_cameras_and_points():
    fig = plt.figure(figsize=(10, 8))
    ax = fig.add_subplot(111, projection='3d')

    # Plot posisi kamera
    ax.scatter(camera_positions[:, 0], camera_positions[:, 1],
camera_positions[:, 2], c='r', label="Camera Positions")

    # Plot objek 3D
    ax.scatter(object_points[:, 0], object_points[:, 1],
object_points[:, 2], c='b', label="Object Points")

    # Menambahkan garis pandangan dari kamera ke objek
    for camera in camera_positions:
        for point in object_points:
            ax.plot([camera[0], point[0]], [camera[1], point[1]],
[camera[2], point[2]], 'g--', alpha=0.5)

    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')
    ax.set_title("Initial Camera Views and Object Points")
    ax.legend()
    plt.show()

visualize_cameras_and_points()
```

```
# --- Novel View Synthesis: Interpolasi Kamera Baru ---
def generate_novel_views(camera_positions, object_points, steps=10):
    # Membuat interpolasi kamera baru antara Kamera 1 dan Kamera 2
    novel_views = []
    for t in np.linspace(0, 1, steps):
        # Interpolasi linier antara Kamera 1 dan Kamera 2
        novel_camera_position = (1 - t) * camera_positions[0] + t *
camera_positions[1]
        novel_views.append(novel_camera_position)

    # Visualisasi pandangan baru
    fig = plt.figure(figsize=(10, 8))
    ax = fig.add_subplot(111, projection='3d')

    # Plot posisi kamera asli dan kamera baru
    ax.scatter(camera_positions[:, 0], camera_positions[:, 1],
camera_positions[:, 2], c='r', label="Original Cameras")
    ax.scatter(object_points[:, 0], object_points[:, 1],
object_points[:, 2], c='b', label="Object Points")
    ax.scatter(
        [view[0] for view in novel_views],
        [view[1] for view in novel_views],
        [view[2] for view in novel_views],
        c='purple',
        label="Novel Views"
    )

    # Menambahkan garis pandangan untuk kamera baru
    for novel_camera in novel_views:
        for point in object_points:
            ax.plot([novel_camera[0], point[0]], [novel_camera[1],
point[1]], [novel_camera[2], point[2]], 'orange', alpha=0.5)

    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')
    ax.set_title("Novel Views Generated by Interpolation")
    ax.legend()
    plt.show()

generate_novel_views(camera_positions, object_points, steps=10)
```

Penjelasan Kode

1. Posisi Kamera

- **camera_positions**: Mengatur posisi kamera yang diketahui (dalam koordinat 3D). Kamera ini adalah posisi awal yang kita miliki untuk menghasilkan pandangan baru.
    - Kamera 1 di [0,0,0][0, 0, 0][0,0,0]

- Kamera 2 di [10,0,0][10, 0, 0][10,0,0]
- Kamera 3 di [0,10,0][0, 10, 0][0,10,0]

2. Titik Objek

- **object_points**: Mengatur posisi titik-titik objek dalam ruang 3D.

3. Visualisasi Kamera dan Objek

- Fungsi **visualize_cameras_and_points** menampilkan posisi kamera dan objek 3D di ruang 3D, bersama dengan garis pandangan dari kamera ke objek.

4. Generasi Novel View

- Fungsi **generate_novel_views** menghasilkan pandangan baru dengan cara interpolasi linier antara kamera yang diketahui.
  - **Interpolasi Linier**:
    - ttt adalah parameter interpolasi dari 0 hingga 1.
    - Semakin banyak langkah interpolasi (stepsstepssteps), semakin halus pandangan baru yang dihasilkan.
  - Posisi kamera baru ditambahkan ke daftar dan divisualisasikan.

5. Visualisasi Pandangan Baru

- Kamera baru dan garis pandangan dari kamera baru ke titik-titik objek divisualisasikan dalam plot 3D.

- **SECTION 7**

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

# URL gambar stereo
left_image_url =
"https://raw.githubusercontent.com/opencv/opencv/master/samples/data/al
oeL.jpg"
right_image_url =
"https://raw.githubusercontent.com/opencv/opencv/master/samples/data/al
oeR.jpg"

# Fungsi untuk mengunduh gambar dari URL
def download_image(url):
    response = requests.get(url, stream=True)
    if response.status_code == 200:
        img_data = np.asarray(bytearray(response.raw.read()),
dtype=np.uint8)
```

```
        return cv2.imdecode(img_data, cv2.IMREAD_GRAYSCALE)  # Gambar
dalam grayscale
    else:
        print(f"Error: Tidak dapat mengunduh gambar dari {url}")
        return None

# Mengunduh gambar stereo (kiri dan kanan)
left_image = download_image(left_image_url)
right_image = download_image(right_image_url)

# Periksa apakah gambar berhasil diunduh
if left_image is None or right_image is None:
    print("Gagal mengunduh gambar stereo. Periksa URL.")
else:
    # Menampilkan gambar stereo
    plt.figure(figsize=(12, 6))
    plt.subplot(1, 2, 1)
    plt.title("Left Image")
    plt.imshow(left_image, cmap='gray')
    plt.axis('off')
    plt.subplot(1, 2, 2)
    plt.title("Right Image")
    plt.imshow(right_image, cmap='gray')
    plt.axis('off')
    plt.show()

    # Membuat StereoBM untuk menghasilkan peta kedalaman
    stereo = cv2.StereoBM_create(numDisparities=64, blockSize=15)
    disparity = stereo.compute(left_image, right_image)

    # Normalisasi peta disparitas untuk visualisasi
    disparity_normalized = cv2.normalize(disparity, None, alpha=0,
beta=255, norm_type=cv2.NORM_MINMAX, dtype=cv2.CV_8U)

    # Menampilkan peta kedalaman
    plt.figure(figsize=(10, 6))
    plt.title("Disparity Map (Depth Map)")
    plt.imshow(disparity_normalized, cmap='plasma')
    plt.colorbar(label="Depth Value")
    plt.axis('off')
    plt.show()
```

Penjelasan Kode

1. Pengunduhan dan Persiapan Gambar Stereo

- **download_image(url)**:
    - Mengunduh gambar dari URL dan mengonversinya menjadi format grayscale menggunakan OpenCV.

- **left_image dan right_image**:
    - Gambar kiri dan kanan dari pasangan stereo yang digunakan untuk menghitung kedalaman.
- **Menampilkan Gambar Stereo**:
    - Gambar kiri dan kanan divisualisasikan berdampingan untuk memahami sudut pandang kamera stereo.

2. StereoBM untuk Peta Kedalaman

- **cv2.StereoBM_create(numDisparities, blockSize)**:
    - Membuat objek StereoBM (Block Matching), salah satu metode populer untuk menghitung disparitas.
    - **numDisparities**: Rentang perbedaan piksel antara gambar kiri dan kanan. Harus kelipatan 16.
    - **blockSize**: Ukuran blok pencocokan piksel. Ukuran blok yang lebih besar meningkatkan akurasi tetapi mengurangi detail.
- **stereo.compute(left_image, right_image)**:
    - Menghitung disparitas antara gambar kiri dan kanan, menghasilkan **peta disparitas (disparity map)**.

3. Normalisasi Peta Disparitas

- **cv2.normalize**:
    - Digunakan untuk mengubah nilai disparitas ke rentang 0-255 untuk visualisasi.
    - **alpha=0, beta=255**: Rentang normalisasi.
    - **norm_type=cv2.NORM_MINMAX**: Normalisasi berbasis nilai minimum dan maksimum.

4. Menampilkan Peta Kedalaman

- **Peta Kedalaman (Depth Map)**:
    - Nilai dalam peta kedalaman berkaitan dengan jarak relatif dari kamera ke objek.
    - **Warna terang**: Objek lebih dekat ke kamera.
    - **Warna gelap**: Objek lebih jauh dari kamera.

- **SECTION 8**

```
import torch
import torch.nn as nn
```

```
import matplotlib.pyplot as plt
import numpy as np

# --- 1. Definisi Neural Network untuk NeRF ---
class SimpleNeRF(nn.Module):
    def __init__(self):
        super(SimpleNeRF, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(5, 256),  # Input: [x, y, z, theta, phi]
            nn.ReLU(),
            nn.Linear(256, 256),
            nn.ReLU(),
            nn.Linear(256, 3),  # Output: RGB
            nn.Sigmoid()  # Membatasi output warna antara [0, 1]
        )

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

# --- 2. Membuat data pelatihan ---
def generate_training_data():
    # Koordinat 3D [x, y, z] di dalam ruang [-1, 1]
    coords = np.random.uniform(-1, 1, (1000, 3))
    # Sudut pandang [theta, phi]
    angles = np.random.uniform(-np.pi, np.pi, (1000, 2))
    # Warna dummy (RGB)
    colors = np.clip(coords + 1, 0, 2) / 2  # Membuat warna dummy
berbasis koordinat

    # Gabungkan koordinat dan sudut pandang menjadi input NeRF
    inputs = np.hstack((coords, angles))
    return torch.tensor(inputs, dtype=torch.float32),
torch.tensor(colors, dtype=torch.float32)

inputs, colors = generate_training_data()

# --- 3. Melatih Model NeRF ---
def train_nerf(model, inputs, colors, epochs=1000, lr=1e-3):
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)
    criterion = nn.MSELoss()
    losses = []

    for epoch in range(epochs):
        optimizer.zero_grad()
        predictions = model(inputs)
        loss = criterion(predictions, colors)
        loss.backward()
        optimizer.step()
```

```
        losses.append(loss.item())

        if (epoch + 1) % 100 == 0:
            print(f"Epoch [{epoch+1}/{epochs}], Loss:
{loss.item():.4f}")

    return losses

# Inisialisasi model
model = SimpleNeRF()

# Latih model
losses = train_nerf(model, inputs, colors)

# Visualisasi loss training
plt.plot(losses)
plt.title("Training Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.show()

# --- 4. Menggunakan Model untuk Novel View Synthesis ---
def generate_novel_view(model, num_points=100):
    coords = np.random.uniform(-1, 1, (num_points, 3))  # Koordinat 3D
baru
    angles = np.random.uniform(-np.pi, np.pi, (num_points, 2))  # Sudut
pandang baru
    inputs = torch.tensor(np.hstack((coords, angles)),
dtype=torch.float32)
    predictions = model(inputs).detach().numpy()

    # Visualisasi hasil prediksi
    fig = plt.figure(figsize=(8, 6))
    ax = fig.add_subplot(111, projection='3d')
    ax.scatter(coords[:, 0], coords[:, 1], coords[:, 2], c=predictions,
marker='o', label='Novel Views')
    ax.set_title("Novel View Synthesis (NeRF)")
    ax.set_xlabel("X")
    ax.set_ylabel("Y")
    ax.set_zlabel("Z")
    ax.legend()
    plt.show()

# Hasilkan pandangan baru
generate_novel_view(model)
```

Penjelasan Kode

1. Definisi Neural Network untuk NeRF

- Input NeRF:
    - Koordinat 3D ([x, y, z]).
    - Sudut pandang (direksi pandang) ([theta, phi]).
- Output NeRF:
    - Warna RGB ([r, g, b]).
- Jaringan saraf sederhana terdiri dari lapisan fully connected (dense layers) dengan fungsi aktivasi ReLU dan Sigmoid untuk menghasilkan warna dalam rentang ([0, 1]).

2. Membuat Data Pelatihan

- Data pelatihan terdiri dari:
    - Koordinat 3D: Diambil secara acak dalam ruang ([-1, 1]).
    - Sudut Pandang: Dipilih secara acak dalam rentang ([-\pi, \pi]).
    - Warna Dummy: Dibuat berdasarkan koordinat untuk simulasi warna RGB.

3. Melatih Model

- Loss Function:
    - Menggunakan Mean Squared Error (MSE) untuk membandingkan warna prediksi dan warna target.
- Optimizer:
    - Menggunakan Adam Optimizer dengan pembelajaran adaptif.

4. Novel View Synthesis

- Setelah model dilatih, kita menghasilkan pandangan baru dengan memasukkan koordinat dan sudut pandang baru.
- Model memprediksi warna untuk setiap titik, menghasilkan warna yang konsisten dengan distribusi data pelatihan.