

Gen-AI practical answers



this is practical question....give me a 7points algorithm and simple code for this.....1)

A) Implement a simple GAN using TensorFlow or PyTorch to learn a low-dimensional synthetic distribution (e.g., Gaussian blobs or moons from `sklearn.datasets`). Use fully connected layers and visualize how the generated data evolves compared to the real distribution.

Here's a **7-point algorithm** and **simple PyTorch code** to implement a **basic GAN** on a low-dimensional synthetic dataset like ``make_moons`` from ``sklearn.datasets``.

✓ 7-Point Algorithm to Build a Simple GAN:

1. **Prepare the Dataset:** Generate synthetic 2D data using ``make_moons`` or ``make_blobs``.
2. **Define Generator Network:** Takes random noise as input and outputs synthetic data points (2D).
3. **Define Discriminator Network:** Takes a data point and classifies it as real or fake.
4. **Define Loss Functions:** Use Binary Cross-Entropy Loss for both generator and discriminator.
5. **Setup Optimizers:** Use Adam optimizers for both networks.
6. **Training Loop:**
 - Train discriminator on real and fake data.
 - Train generator to fool the discriminator.
7. **Visualize Results:** Plot generated vs real data every few epochs.

🔧 Simple GAN Code (PyTorch + ``make_moons``)

```
python

import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
from sklearn.datasets import make_moons
import numpy as np

# Data
X, _ = make_moons(n_samples=1000, noise=0.1)
X = torch.tensor(X, dtype=torch.float32)

# Generator
class Generator(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(2, 16),
            nn.ReLU(),
            nn.Linear(16, 2)
        )
```

```

    )

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

# Discriminator
class Discriminator(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(2, 16),
            nn.ReLU(),
            nn.Linear(16, 1),
            nn.Sigmoid()
        )

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

# Initialize
G = Generator()
D = Discriminator()
criterion = nn.BCELoss()
g_opt = optim.Adam(G.parameters(), lr=0.01)
d_opt = optim.Adam(D.parameters(), lr=0.01)

# Training
for epoch in range(5001):
    # === Train Discriminator ===
    real_data = X[torch.randint(0, X.size(0), (128,))]
    fake_data = G(torch.randn(128, 2)).detach()

    real_labels = torch.ones(128, 1)
    fake_labels = torch.zeros(128, 1)

    d_loss_real = criterion(D(real_data), real_labels)
    d_loss_fake = criterion(D(fake_data), fake_labels)
    d_loss = d_loss_real + d_loss_fake

    d_opt.zero_grad()
    d_loss.backward()
    d_opt.step()

    # === Train Generator ===
    noise = torch.randn(128, 2)
    fake = G(noise)
    g_loss = criterion(D(fake), real_labels) # want D(fake) to be 1

    g_opt.zero_grad()
    g_loss.backward()
    g_opt.step()

    # === Visualization ===
    if epoch % 1000 == 0:
        with torch.no_grad():
            fake_samples = G(torch.randn(1000, 2)).numpy()
            plt.figure(figsize=(4, 4))
            plt.scatter(X[:, 0], X[:, 1], c='blue', alpha=0.5, label='Real')
            plt.scatter(fake_samples[:, 0], fake_samples[:, 1], c='red', alpha=0.5, label='Fake')
            plt.title(f'Epoch {epoch}')
            plt.legend()
            plt.show()

```

This code uses fully connected layers and trains the GAN on the 2D moon-shaped dataset. You'll see the fake data distribution evolve toward the real one over epochs.

Would you like a TensorFlow version or a diagram of the GAN architecture as well?



1, B) Using a trained GAN on synthetic data, interpolate between two random noise vectors and generate samples along the path. Visualize the transition to observe how smoothly the generator maps the latent space to outputs.

Great follow-up! For **1B**, you're asking to **interpolate between two latent vectors** and **observe how the generated samples morph** between them using a trained GAN.

✓ 7-Point Algorithm for Latent Space Interpolation in GAN:

1. **Train the GAN** on low-dimensional synthetic data (already done in 1A).
2. **Select two random latent vectors** `z1` and `z2` (e.g., 2D noise vectors).
3. **Interpolate between them** using linear interpolation (or spherical interpolation for better continuity).
4. **Generate samples** from the interpolated vectors using the generator.
5. **Store the outputs** to observe gradual transformation.
6. **Plot the generated samples** sequentially to visualize the transition.
7. **(Optional)** Animate or create a GIF to enhance the interpolation visualization.

PyTorch Code for Latent Space Interpolation (Continuing from 1A):

```
python

import numpy as np
import matplotlib.pyplot as plt

def interpolate(z1, z2, steps=10):
    return [(1 - t) * z1 + t * z2 for t in np.linspace(0, 1, steps)]

# Sample two latent vectors
z1 = torch.randn(1, 2)
z2 = torch.randn(1, 2)

# Interpolate
interpolated_z = interpolate(z1, z2, steps=10)

# Generate samples
with torch.no_grad():
    interpolated_samples = [G(z).numpy()[0] for z in interpolated_z]

# Visualize
interpolated_samples = np.array(interpolated_samples)
plt.figure(figsize=(6, 6))
plt.plot(interpolated_samples[:, 0], interpolated_samples[:, 1], '-o', label="Interpolated Path")
plt.scatter(X[:, 0], X[:, 1], alpha=0.2, label='Real Data')
plt.title("Latent Space Interpolation")
plt.legend()
plt.grid(True)
plt.show()
```

What You'll See:

- A smooth path of generated points transitioning from `z1` to `z2`.
- The interpolated points lie in or near the real data distribution if the GAN learned properly.

Would you like me to generate a GIF from the interpolation frames, or add spherical interpolation (SLERP) for smoother morphing?



2)
A) Build and train both a basic GAN and a DCGAN on a small grayscale image set (20–50 images). Compare their outputs, training stability, and image quality. Compare the generated outputs from both models and analyze differences in training behavior, convergence speed, and image quality.

Great — you're now moving to **comparing a basic GAN and a DCGAN** on a **small grayscale image dataset (20–50 images)**. This is an excellent exercise in understanding how architectural choices affect GAN performance.

✓ 7-Point Algorithm to Compare Basic GAN and DCGAN:

1. **Prepare Dataset:** Load/resize 20–50 grayscale images to a fixed size (e.g., 64x64), normalize pixel values to `[-1, 1]`.
2. **Define Basic GAN:** Use fully connected layers for both Generator and Discriminator.
3. **Define DCGAN:** Use transposed convolutions in Generator and strided convolutions in Discriminator.
4. **Train Both Models Separately:**
 - Same loss (e.g., Binary Cross-Entropy), optimizers, and latent dimensions.
5. **Track Metrics:**
 - Discriminator & Generator loss curves.
 - Visual samples during training (e.g., every 100 epochs).
6. **Compare Outputs:**
 - Visual clarity, diversity, overfitting signs.
7. **Analyze Differences:**
 - Training stability (loss oscillation, mode collapse),
 - Convergence speed,
 - Output quality.

Simple Implementation Plan (PyTorch-based)

Dataset (Step 1):

python

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

transform = transforms.Compose([
    transforms.Grayscale(),
    transforms.Resize((64, 64)),
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,)) # [-1, 1]
])

dataset = datasets.ImageFolder("your_data_folder", transform=transform)
dataloader = DataLoader(dataset, batch_size=8, shuffle=True)
```

Basic GAN Architectures (Step 2):

python

```
# Generator (Basic GAN - FC)
class BasicGenerator(nn.Module):
    def __init__(self, z_dim=100):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(z_dim, 256),
            nn.ReLU(True),
            nn.Linear(256, 64*64),
            nn.Tanh()
        )

    def forward(self, z):
        return self.net(z).view(-1, 1, 64, 64)

# Discriminator (Basic GAN - FC)
class BasicDiscriminator(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Flatten(),
            nn.Linear(64*64, 256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, 1),
            nn.Sigmoid()
        )

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

DCGAN Architectures (Step 3):

python

```
# Generator (DCGAN)
class DCGenerator(nn.Module):
    def __init__(self, z_dim=100):
        super().__init__()
        self.net = nn.Sequential(
            nn.ConvTranspose2d(z_dim, 128, 4, 1, 0), # (4x4)
            nn.BatchNorm2d(128),
            nn.ReLU(True),
            nn.ConvTranspose2d(128, 64, 4, 2, 1), # (8x8)
            nn.BatchNorm2d(64),
            nn.ReLU(True),
            nn.ConvTranspose2d(64, 1, 4, 2, 1), # (16x16)
            nn.Tanh()
        )
```

```
def forward(self, z):
    return self.net(z.view(-1, z_dim, 1, 1))

# Discriminator (DCGAN)
class DCDiscriminator(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Conv2d(1, 64, 4, 2, 1),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(64, 128, 4, 2, 1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Flatten(),
            nn.Linear(128*16*16, 1),
            nn.Sigmoid()
        )

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

Training Loop (Step 4+5):

You can reuse the same loss, optimizer (Adam), and steps for both versions.

```
python

loss_fn = nn.BCELoss()
g_opt = optim.Adam(generator.parameters(), lr=0.0002, betas=(0.5, 0.999))
d_opt = optim.Adam(discriminator.parameters(), lr=0.0002, betas=(0.5, 0.999))
```

Save generated samples and loss curves for each model every few epochs.

What to Compare After Training (Step 6-7):

Feature	Basic GAN	DCGAN
Convergence Speed	Slower, less stable	Faster, more stable
Image Quality	Blurry, less structured	Sharper, more realistic
Overfitting	More prone due to small dataset	Better generalization with Conv
Loss Curves	Fluctuate more	Smoother with batch norm
Output Diversity	Lower	Higher

Would you like me to provide a full training loop template that handles both models side-by-side and logs images automatically?



2)B) Train the same GAN model using three different learning rates (e.g., 0.01, 0.001, and 0.0001) to evaluate how the learning rate influences training dynamics. Record the generator and discriminator losses, and observe the visual quality of the generated samples for each configuration.

Great — **2B** focuses on evaluating the **impact of learning rate on GAN training dynamics** by training the **same model** (e.g., a basic GAN or DCGAN) with **three different learning rates** and comparing:

- Loss behavior
- Training stability
- Visual quality of generated samples

✓ 7-Point Algorithm to Analyze Learning Rate Effects in GAN Training:

1. **Choose a GAN architecture** (same for all experiments — Basic GAN or DCGAN).
2. **Prepare the dataset** (same preprocessed grayscale image set).
3. **Define three learning rates** (e.g., `0.01`, `0.001`, `0.0001`).
4. **Train the GAN** three times — once per learning rate, keeping all other hyperparameters identical.
5. **Record generator & discriminator losses** over epochs.
6. **Save generated images** at key intervals (e.g., every 200 epochs).
7. **Compare results:**
 - Analyze loss curves (oscillation, convergence),
 - Compare image quality (sharpness, realism, mode collapse),
 - Assess convergence speed & training stability.

Example PyTorch Training Loop with Variable Learning Rates

Training Function (Reusable)

```
python

def train_gan(lr, generator, discriminator, dataloader, epochs=2000, z_dim=100):
    print(f"\n--- Training with learning rate: {lr} ---")

    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    generator, discriminator = generator.to(device), discriminator.to(device)

    criterion = nn.BCELoss()
    g_opt = optim.Adam(generator.parameters(), lr=lr, betas=(0.5, 0.999))
    d_opt = optim.Adam(discriminator.parameters(), lr=lr, betas=(0.5, 0.999))

    fixed_noise = torch.randn(16, z_dim).to(device)

    g_losses, d_losses = [], []

    for epoch in range(epochs):
        for real_images, _ in dataloader:
            real_images = real_images.to(device)
            batch_size = real_images.size(0)
            z = torch.randn(batch_size, z_dim).to(device)

            fake_images = generator(z)
            real_labels = torch.ones(batch_size, 1).to(device)
            fake_labels = torch.zeros(batch_size, 1).to(device)

            # Train Discriminator
            d_real = discriminator(real_images)
            d_fake = discriminator(fake_images.detach())
            d_loss = criterion(d_real, real_labels) + criterion(d_fake, fake_labels)
```

```

d_opt.zero_grad()
d_loss.backward()
d_opt.step()

# Train Generator
g_fake = discriminator(fake_images)
g_loss = criterion(g_fake, real_labels)
g_opt.zero_grad()
g_loss.backward()
g_opt.step()

g_losses.append(g_loss.item())
d_losses.append(d_loss.item())

# Save images
if epoch % 500 == 0:
    with torch.no_grad():
        samples = generator(fixed_noise).cpu()
        grid = torchvision.utils.make_grid(samples, nrow=4, normalize=True)
        plt.imshow(grid.permute(1, 2, 0).numpy())
        plt.title(f"LR={lr}, Epoch={epoch}")
        plt.axis('off')
        plt.show()

return g_losses, d_losses

```

Run for 3 Learning Rates:

```

python

learning_rates = [0.01, 0.001, 0.0001]
for lr in learning_rates:
    G = BasicGenerator()
    D = BasicDiscriminator()
    g_loss, d_loss = train_gan(lr, G, D, dataloader)

# You can store losses and plot them later

```

What to Look For in Results

Learning Rate	Expected Behavior
0.01	Likely unstable — GAN may diverge or show severe oscillation in loss
0.001	Generally balanced — stable training and decent image quality
0.0001	Very slow — losses might stay high longer, convergence is slow but stable

Plot Losses

```

python

plt.plot(g_loss, label='Generator Loss')
plt.plot(d_loss, label='Discriminator Loss')
plt.title(f"Loss Curves (LR={lr})")
plt.legend()
plt.show()

```

Would you like a script that runs all three settings automatically and saves side-by-side image comparisons and loss plots?



- 4)
A) Train a word-level RNN model on a simple custom text (e.g., a few manually written sentences). Implement training with and without teacher forcing, and compare the loss curves and sample outputs from both approaches.

Excellent — this task (4A) focuses on **training a word-level RNN** on a small custom dataset, and comparing performance **with vs. without teacher forcing**.

✓ 7-Point Algorithm for Word-Level RNN with Teacher Forcing Comparison:

1. **Prepare custom dataset:** Tokenize a few sentences at the word level, build a vocabulary, and convert sentences into word indices.
2. **Build an RNN model:** Input is a word index, and the model predicts the next word in the sequence.
3. **Define training process** with two modes:
 - **With teacher forcing:** feed the true previous word.
 - **Without teacher forcing:** feed the model's previous output.
4. **Train both versions** on the same dataset.
5. **Track training loss curves** over epochs for both setups.
6. **Generate text samples** after training from both models using a seed word.
7. **Compare outputs:** Observe quality, fluency, and error accumulation in generated sentences.

👉 Sample Sentences (Custom Dataset)

```
python

sentences = [
    "the cat sat on the mat",
    "the dog barked at the cat",
    "the cat chased the mouse",
    "the dog chased the ball",
    "the mouse ran away"
]
```

🔧 PyTorch Implementation Overview

🔧 Preprocessing

```
python

from collections import defaultdict
import torch
import torch.nn as nn
```

```
import torch.optim as optim

# Tokenize and build vocabulary
tokens = " ".join(sentences).split()
vocab = sorted(set(tokens))
word2idx = {w: i for i, w in enumerate(vocab)}
idx2word = {i: w for w, i in word2idx.items()}
vocab_size = len(vocab)

# Encode sequences
data = [[word2idx[word] for word in sentence.split()] for sentence in sentences]
```

RNN Model

python

```
class WordRNN(nn.Module):
    def __init__(self, vocab_size, hidden_size=64):
        super().__init__()
        self.embed = nn.Embedding(vocab_size, 32)
        self.rnn = nn.RNN(32, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, vocab_size)

    def forward(self, x, hidden=None):
        x = self.embed(x)
        out, hidden = self.rnn(x, hidden)
        out = self.fc(out)
        return out, hidden
```

Training Function

python

```
def train_model(model, data, teacher_forcing=True, epochs=100):
    model.train()
    optimizer = optim.Adam(model.parameters(), lr=0.01)
    loss_fn = nn.CrossEntropyLoss()
    losses = []

    for epoch in range(epochs):
        total_loss = 0
        for seq in data:
            model.zero_grad()
            input_seq = torch.tensor(seq[:-1]).unsqueeze(0)
            target_seq = torch.tensor(seq[1:])

            hidden = None
            outputs = []

            if teacher_forcing:
                out, _ = model(input_seq)
                loss = loss_fn(out.squeeze(0), target_seq)
            else:
                input_token = input_seq[0, 0].unsqueeze(0).unsqueeze(0)
                output_seq = []
                for t in range(len(seq)-1):
                    out, hidden = model(input_token, hidden)
                    input_token = out.argmax(-1)
                    output_seq.append(out.squeeze(0))
                output_seq = torch.cat(output_seq, dim=0)
                loss = loss_fn(output_seq, target_seq)

            loss.backward()
            optimizer.step()
            total_loss += loss.item()
```

```
losses.append(total_loss / len(data))
return losses
```

Sample Text Generation

python

```
def generate(model, start_word, max_len=5):
    model.eval()
    idx = word2idx[start_word]
    input = torch.tensor([[idx]])
    hidden = None
    output = [start_word]

    for _ in range(max_len):
        out, hidden = model(input, hidden)
        next_idx = out.argmax(-1).item()
        word = idx2word[next_idx]
        output.append(word)
        input = torch.tensor([[next_idx]])

    return " ".join(output)
```

Compare Models

python

```
rnn1 = WordRNN(vocab_size)
rnn2 = WordRNN(vocab_size)

loss_tf = train_model(rnn1, data, teacher_forcing=True)
loss_no_tf = train_model(rnn2, data, teacher_forcing=False)

import matplotlib.pyplot as plt
plt.plot(loss_tf, label="With Teacher Forcing")
plt.plot(loss_no_tf, label="Without Teacher Forcing")
plt.legend()
plt.title("Loss Comparison")
plt.show()

print("TF Sample:", generate(rnn1, "the"))
print("No TF Sample:", generate(rnn2, "the"))
```

What You'll Observe

Feature	With Teacher Forcing	Without Teacher Forcing
Loss curve	Faster convergence	Slower or noisier training
Text quality	More coherent sequences	May drift or repeat quickly
Stability	More stable gradients	Harder to train due to exposure bias

Would you like this example extended with **GRU** or **LSTM** instead of plain RNN?



4)B) Using a trained character-level or word-level RNN model, implement beam search decoding (beam width = 2 or 3) and compare it with greedy decoding. Generate and evaluate sample outputs from the same seed input using both strategies.

Excellent — in **4B**, you're building on a trained RNN (word- or character-level) to implement and **compare greedy decoding vs. beam search decoding**.

✓ 7-Point Algorithm for Beam Search vs. Greedy Decoding:

1. **Use a trained RNN model** from 4A (word- or character-level).
2. **Define greedy decoding**: always pick the token with the highest probability at each step.
3. **Define beam search decoding** with a beam width (e.g., 2 or 3):
 - Keep top-k most likely partial sequences at each step.
4. **Generate sequences** from the same seed token using both methods.
5. **Track total probabilities** or log-likelihoods during decoding.
6. **Compare outputs**: coherence, diversity, and quality.
7. **Analyze performance**: greedy is fast but short-sighted; beam is slower but more globally aware.

Greedy vs Beam Search Decoding (PyTorch, Word-Level Example)

```
python

import torch.nn.functional as F

# Greedy Decoding
def greedy_decode(model, start_word, max_len=5):
    model.eval()
    input = torch.tensor([[word2idx[start_word]]])
    hidden = None
    output = [start_word]

    for _ in range(max_len):
        out, hidden = model(input, hidden)
        next_idx = out.argmax(-1).item()
        word = idx2word[next_idx]
        output.append(word)
        input = torch.tensor([[next_idx]])

    return output
```

Beam Search Decoding

```
python

def beam_search_decode(model, start_word, beam_width=3, max_len=5):
    model.eval()
    start_idx = word2idx[start_word]
```

```
sequences = [[ [start_idx], 0.0, None ]] # [sequence, score, hidden]

for _ in range(max_len):
    all_candidates = []
    for seq, score, hidden in sequences:
        input = torch.tensor([[seq[-1]]])
        out, hidden_new = model(input, hidden)
        probs = F.log_softmax(out[0, -1], dim=0)
        topk = torch.topk(probs, beam_width)

        for i in range(beam_width):
            word_idx = topk.indices[i].item()
            new_seq = seq + [word_idx]
            new_score = score + topk.values[i].item() # log-prob
            all_candidates.append([new_seq, new_score, hidden_new])

    # Keep top k sequences
    ordered = sorted(all_candidates, key=lambda tup: tup[1], reverse=True)
    sequences = ordered[:beam_width]

# Choose the best sequence
best_seq = sequences[0][0]
return [idx2word[i] for i in best_seq]
```

Example Output Comparison

```
python

seed = "the"
greedy = greedy_decode(rnn1, seed)
beam = beam_search_decode(rnn1, seed, beam_width=3)

print("Greedy:", " ".join(greedy))
print("Beam Search:", " ".join(beam))
```

Expected Behavior

Feature	Greedy Decoding	Beam Search Decoding
Speed	Fast	Slower (keeps multiple sequences)
Strategy	Locally optimal	Globally more aware
Quality	May repeat words	Often more fluent and logical
Diversity	Low	Slightly better with beam width >1

Would you like this beam search adapted to character-level RNNs or upgraded with length normalization or top-k filtering?



5)
A) Implement neural style transfer based on the method by Gatys et al. Use small built-in or custom content and style images (e.g., a geometric pattern and a simple photograph). Generate and display the stylized output image.

Great — task **5A** involves implementing **Neural Style Transfer (NST)** based on the classic **Gatys et al. (2015)** approach, which uses a **pre-trained CNN** (typically VGG19) to blend content and style from two

images.

✓ 7-Point Algorithm for Neural Style Transfer (Gatys et al.):

1. **Load content and style images** (resize + normalize).
2. **Use a pre-trained CNN** (e.g., `VGG19`) to extract feature maps.
3. **Define content loss**: MSE between features of content and generated image.
4. **Define style loss**: MSE between Gram matrices of style and generated image.
5. **Create a target image** initialized from content (or random).
6. **Optimize target image** using gradient descent to minimize total loss:

$$\text{Loss}_{\text{total}} = \alpha \cdot \text{Loss}_{\text{content}} + \beta \cdot \text{Loss}_{\text{style}}$$

7. **Display stylized result.**

Simple PyTorch Implementation

1. Load and preprocess images

```
python

from PIL import Image
import torchvision.transforms as transforms
import torch

def load_image(path, max_size=400):
    image = Image.open(path).convert('RGB')
    transform = transforms.Compose([
        transforms.Resize(max_size),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406],
                             [0.229, 0.224, 0.225])
    ])
    image = transform(image).unsqueeze(0)
    return image.to(torch.float)
```

2. Load VGG19 and extract layers

```
python

import torchvision.models as models
import torch.nn as nn

class VGGFeatures(nn.Module):
    def __init__(self, layers):
        super().__init__()
        self.vgg = models.vgg19(pretrained=True).features.eval()
        self.layers = layers

    def forward(self, x):
        features = {}
        for name, layer in self.vgg._modules.items():
            x = layer(x)
            if name in self.layers:
```

```

        features[name] = x
    return features

```

3. Style and content loss functions

python

```

def gram_matrix(tensor):
    b, c, h, w = tensor.size()
    features = tensor.view(c, h * w)
    return torch.mm(features, features.t()) / (c * h * w)

def compute_loss(gen_features, content_features, style_features, content_weight, style_weight):
    content_loss = torch.mean((gen_features['21'] - content_features['21']) ** 2)

    style_loss = 0
    for layer in style_features:
        G = gram_matrix(gen_features[layer])
        A = gram_matrix(style_features[layer])
        style_loss += torch.mean((G - A) ** 2)

    return content_weight * content_loss + style_weight * style_loss

```

4. Style Transfer Loop

python

```

def style_transfer(content_img, style_img, steps=300, style_weight=1e6, content_weight=1):
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    model = VGGFeatures({'0', '5', '10', '19', '21'}).to(device).eval()

    content_features = model(content_img)
    style_features = model(style_img)

    generated = content_img.clone().requires_grad_(True)
    optimizer = torch.optim.Adam([generated], lr=0.02)

    for step in range(steps):
        optimizer.zero_grad()
        gen_features = model(generated)
        loss = compute_loss(gen_features, content_features, style_features, content_weight, style_weight)
        loss.backward()
        optimizer.step()

        if step % 50 == 0:
            print(f"Step {step}, Loss: {loss.item():.4f}")

    return generated

```

5. Load images and run NST

python

```

content = load_image('path/to/content.jpg')
style = load_image('path/to/style.jpg')

output = style_transfer(content, style)

```

6. Display Stylized Output

```
python

import matplotlib.pyplot as plt

def im_convert(tensor):
    image = tensor.clone().detach().squeeze().cpu()
    image = image * torch.tensor([0.229, 0.224, 0.225])[:, None, None] + \
        torch.tensor([0.485, 0.456, 0.406])[:, None, None]
    return image.clamp(0, 1).permute(1, 2, 0)

plt.imshow(im_convert(output))
plt.title("Stylized Image")
plt.axis('off')
plt.show()
```

Would you like me to generate or recommend public-domain sample images (content & style) for testing this right away?



5)B) Use a custom content image (such as a photo from your local system) and a simple style image (e.g., sketch or texture). Apply neural style transfer and visualize how the style features are applied to the content image.

Perfect — in **5B**, you're extending the neural style transfer task to use a **custom content image (e.g., a photo)** and a **simple style image** like a sketch or texture. The goal is to visualize how the model **transfers visual patterns from the style** onto your content image while **preserving the main structure**.

✓ Here's a Clear Plan (aligned with Gatys et al.):

1. Choose two images:

- Content: Your photo (e.g., portrait, landscape).
- Style: Sketch, scribble, pattern, or texture (e.g., brush strokes).

2. Load and preprocess images: Resize, normalize for VGG19.

3. Extract feature maps using a pre-trained VGG19 model.

4. Compute content and style losses:

- Content = feature difference at one deep layer (e.g., `conv4_2`)
- Style = difference of **Gram matrices** at multiple layers.

5. Initialize the generated image from the content image.

6. Run optimization: Minimize combined content + style loss.

7. Post-process and visualize the final stylized image.

Visualization Tip

After transfer, you should expect:

- **Overall structure** of the photo (faces, buildings, objects) preserved.
- **Textures and strokes** from the style applied over the photo.
- The style might "wrap" around the contours, especially in low-resolution images.

Code Adjustment

From step 5A, you only need to:

- Replace image paths with your own images.
- Optionally adjust ``style_weight`` and ``steps``.

```
python

# Replace these with your own image paths
content = load_image("your_photo.jpg", max_size=400)
style = load_image("your_sketch.png", max_size=400)

# Perform style transfer
output = style_transfer(content, style, steps=300, style_weight=1e6, content_weight=1)

# Display output
plt.imshow(im_convert(output))
plt.title("Stylized Output")
plt.axis('off')
plt.show()
```

Optional Tweaks

Parameter	Effect
<code>`style_weight`</code>	↑ more dramatic style effect, ↓ softer stylization
<code>`steps`</code>	More steps → smoother blending (but slower)
<code>`max_size`</code>	Higher resolution, but also increases memory/compute time

Would you like help finding or generating a free style image (e.g., pencil sketch or paint texture) for this experiment?



6)
A) Implement a simple fast style transfer model that applies a pre-learned style to content images in real time. Test it with two or three different content images and compare its speed and output quality with traditional iterative style transfer.

Excellent — task **6A** is about implementing a **Fast Neural Style Transfer (FST)** model, as introduced by **Johnson et al. (2016)**, which applies style in **real-time** by training a **feedforward network** for one style.

This is significantly faster than Gatys' iterative optimization method and is ideal for deployment or video-style transfer.

✓ 7-Point Algorithm for Fast Style Transfer (Johnson et al.)

1. **Preprocess images:** Normalize content and style images.
2. **Use a feedforward style transfer network:**
 - Architecture: Conv → Residual blocks → Upsampling Conv.
3. **Use pre-trained VGG16/VGG19** to compute losses:
 - **Content loss:** from a middle layer (e.g., `relu2_2`).
 - **Style loss:** from Gram matrices across multiple layers.
4. **Train the transformer network** on a dataset (e.g., MS COCO).
5. **Freeze the style model after training.**
6. **Apply model on 2-3 test content images** to stylize in real time.
7. **Compare speed and quality** with traditional Gatys-style optimization.

PyTorch Code Sketch

1. Style Transfer Network (Transformer)

```
python

import torch.nn as nn

class StyleTransformerNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            ConvLayer(3, 32, 9, 1),
            nn.ReLU(),
            ConvLayer(32, 64, 3, 2),
            nn.ReLU(),
            ConvLayer(64, 128, 3, 2),
            nn.ReLU(),
            ResidualBlock(128),
            ResidualBlock(128),
            ResidualBlock(128),
            UpsampleConvLayer(128, 64, 3, 1, 2),
            nn.ReLU(),
            UpsampleConvLayer(64, 32, 3, 1, 2),
            nn.ReLU(),
            ConvLayer(32, 3, 9, 1)
        )

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

# Helper Conv and Residual Blocks
class ConvLayer(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size, stride):
        super().__init__()
        padding = kernel_size // 2
        self.conv = nn.Conv2d(in_channels, out_channels, kernel_size, stride, padding)
    def forward(self, x): return self.conv(x)

class ResidualBlock(nn.Module):
    def __init__(self, channels):
        super().__init__()
        self.block = nn.Sequential(
            ConvLayer(channels, channels, 3, 1),
```

```

        nn.ReLU(),
        ConvLayer(channels, channels, 3, 1)
    )
    def forward(self, x): return x + self.block(x)

class UpsampleConvLayer(nn.Module):
    def __init__(self, in_c, out_c, kernel_size, stride, scale_factor):
        super().__init__()
        self.upsample = nn.Upsample(scale_factor=scale_factor)
        self.conv = ConvLayer(in_c, out_c, kernel_size, stride)
    def forward(self, x): return self.conv(self.upsample(x))

```

2. Loss Network (VGG19) and Training

You train the model using:

- **Content loss** from `relu2_2`
- **Style loss** from Gram matrices of `relu1_1`, `relu2_1`, `relu3_1`, `relu4_1`

Pretrained VGG layers are **frozen** during training. You optimize the **transformer model** only.

3. After Training: Apply in Real-Time

```

python

# Assume model is trained and loaded
model = StyleTransformerNet()
model.load_state_dict(torch.load("starry_night_style.pth"))
model.eval()

# Load content image
img = load_image("your_photo.jpg", size=512).unsqueeze(0)
stylized = model(img).squeeze()

# Convert to display format
output_img = im_convert(stylized)
plt.imshow(output_img)
plt.title("Fast Stylized Output")
plt.axis("off")
plt.show()

```

Speed Comparison Summary

Feature	Gatys et al. Method	Fast Style Transfer (FST)
Runtime	Minutes per image	Real-time (~10–50 ms on GPU)
Quality	Higher fidelity, customizable	Slightly less accurate, faster
Flexibility	Any style, any time	One model = one style
Use Case	Art restoration, research	Mobile, video, AR filters

Test 2–3 Images

To complete the task:

- Run your trained FST model on **2-3 photos**.
- Measure the **inference time** using `time.time()`.
- Compare stylized outputs side-by-side with Gatys' method.

Would you like a link to a pre-trained style model (e.g., "Udnie" or "The Wave") for fast testing?



7.B) Run the style transfer algorithm on the same pair of content and style images while adjusting the style-to-content ratio. Use different values (e.g., low, medium, high) and compare how the balance affects the stylization and content preservation.

7B involves experimenting with the style-to-content ratio during style transfer, which controls how much influence the style image has on the content image. By adjusting this ratio (low, medium, high), we can examine how it affects the final stylization and content preservation.

✓ Steps to Implement and Evaluate the Adjustments:

1. **Choose a content and style image:** Use the same pair of images for comparison.
2. **Adjust the style-to-content ratio:**
 - Low ratio: More content preservation, less style influence.
 - Medium ratio: Balanced influence of both content and style.
 - High ratio: Strong style influence, with less content detail preserved.
3. **Apply the style transfer model:** Run the style transfer with different ratios and generate outputs.
4. **Compare results:** Visually analyze the balance between style transfer and content preservation for each ratio.

Key Idea:

The **style-to-content ratio** is controlled by adjusting the weight given to the style and content losses in the total loss function:

$$\text{Loss}_{\text{total}} = \alpha \cdot \text{Loss}_{\text{content}} + \beta \cdot \text{Loss}_{\text{style}}$$

Where:

- α controls the weight of the **content loss** (content preservation).
- β controls the weight of the **style loss** (style influence).

By adjusting these weights, we change the balance between how much content is preserved and how much the style is transferred.

Code Example:

Here's how you would modify the `style_transfer` function to handle different **style-to-content ratios**:

```
python

import matplotlib.pyplot as plt

def style_transfer_with_ratio(content_img, style_img, style_weight, content_weight, steps=300):
    # Using the previously defined neural style transfer model
    model = VGGFeatures({'0', '5', '10', '19', '21'}).to(device).eval()

    content_features = model(content_img)
    style_features = model(style_img)

    generated = content_img.clone().requires_grad_(True)
    optimizer = torch.optim.Adam([generated], lr=0.02)

    for step in range(steps):
        optimizer.zero_grad()
        gen_features = model(generated)

        # Compute total loss based on content and style weights
        content_loss = torch.mean((gen_features['21'] - content_features['21']) ** 2)
        style_loss = 0
        for layer in style_features:
            G = gram_matrix(gen_features[layer])
            A = gram_matrix(style_features[layer])
            style_loss += torch.mean((G - A) ** 2)

        total_loss = content_weight * content_loss + style_weight * style_loss
        total_loss.backward()
        optimizer.step()

        if step % 50 == 0:
            print(f"Step {step}, Loss: {total_loss.item():.4f}")

    return generated

# Load your content and style images
content = load_image("your_content_image.jpg", max_size=400)
style = load_image("your_style_image.jpg", max_size=400)

# Try different style-to-content ratios
ratios = [(1e5, 1), (1e6, 1), (1e7, 1)] # Low, Medium, High style weights

fig, axs = plt.subplots(1, 3, figsize=(15, 5))

for i, (style_w, content_w) in enumerate(ratios):
    stylized_img = style_transfer_with_ratio(content, style, style_weight=style_w, content_weight=content_w)
    output_img = im_convert(stylized_img.squeeze())
    axs[i].imshow(output_img)
    axs[i].set_title(f"Style Weight: {style_w}")
    axs[i].axis('off')

plt.show()
```

Explanation of the Code:

- **style_weight** and **content_weight** control the balance between style and content.
- **Three different ratios** are tested: low (`1e5`), medium (`1e6`), and high (`1e7`).
- Each ratio results in a different level of style influence.
- **Resulting images** are plotted side by side for easy comparison.

Expected Behavior:

1. **Low Style Weight** (e.g., `1e5`):

- **Content preservation:** High; the generated image will closely resemble the original content image.
- **Style influence:** Low; the style (e.g., brush strokes, patterns) will be faint.

2. Medium Style Weight (e.g., $1e6$):

- **Content and style balance:** Balanced. The generated image will retain content structure while applying the style in noticeable patterns.

3. High Style Weight (e.g., $1e7$):

- **Content preservation:** Low; the generated image will be dominated by the style.
- **Style influence:** High; the content image might lose recognizable structure as the style overwhelms it.

What to Expect:

- With **low style weight**, the generated image should look mostly like the content image with subtle style effects.
- With **medium style weight**, you should observe a **balanced mix** where both content and style are clearly visible.
- With **high style weight**, the **style features** (like colors, textures) should overpower the content, with **minimal preservation** of the original content's structure.

Analysis:

1. **Low style-to-content ratio:** This will preserve the **content features** more clearly, while the **style is only subtly applied**.
2. **Medium ratio:** Offers a balance where both **style and content are preserved**, resulting in visually appealing outputs.
3. **High ratio:** The content structure is **distorted**, with the **style dominating** the image.

Comparison:

By comparing the **output quality**, you'll clearly see how adjusting the balance impacts the **visual appeal** of the generated image. It's a trade-off between keeping the **realism** of the content and infusing **artistic style**.

Would you like to try the code on specific images or adjust the parameters further?



7)

A) Implement a simple GAN to generate synthetic data for augmentation. Use a small dataset of images (e.g., 2-3 categories, or random shapes), train the GAN, and generate additional samples. Visualize the generated images and analyze their quality.

7A: Implementing a Simple GAN for Data Augmentation

In **7A**, you are tasked with implementing a **Generative Adversarial Network (GAN)** that generates synthetic data for augmentation. We'll use a small dataset (e.g., images of 2-3 categories or random shapes), train the GAN on this dataset, and then generate additional samples. The main goal is to **visualize** the generated images and analyze their **quality**.

Steps to Implement the GAN:

1. Create a small dataset:

- Either use an existing simple dataset (e.g., MNIST, or create synthetic data like random shapes).

2. Define the GAN architecture:

- **Generator (G):** Takes a random noise vector and generates synthetic images.
- **Discriminator (D):** Classifies images as real (from the dataset) or fake (from the generator).

3. Loss Functions:

- **Discriminator Loss:** Binary cross-entropy for distinguishing real vs fake images.
- **Generator Loss:** Binary cross-entropy to fool the discriminator into classifying generated images as real.

4. Train the GAN:

- Alternate between training the discriminator and the generator.

5. Generate synthetic images:

- Once trained, use the generator to create new samples for data augmentation.
-

PyTorch Code for a Simple GAN:

Here is a minimal implementation of a GAN for generating synthetic images from a small dataset (such as random shapes or simple categories):

1. Setup GAN Architecture:

```
python

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

# Create a synthetic dataset (e.g., random shapes or simple categories)
class SimpleDataset(Dataset):
    def __init__(self, num_samples=1000, image_size=28):
        self.num_samples = num_samples
        self.image_size = image_size
        self.transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))])

    def __len__(self):
        return self.num_samples

    def __getitem__(self, idx):
        # Generate a random image (e.g., random noise)
        noise = torch.randn(1, 1, self.image_size, self.image_size)
```

```
# Generate a random image (e.g., random shapes or categories)
image = np.random.rand(self.image_size, self.image_size) * 255 # Random image
return self.transform(image.astype(np.float32))

# Generator Network
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.fc = nn.Sequential(
            nn.Linear(100, 256), # Latent space size
            nn.ReLU(),
            nn.Linear(256, 512),
            nn.ReLU(),
            nn.Linear(512, 1024),
            nn.ReLU(),
            nn.Linear(1024, 28 * 28), # Output image size
            nn.Tanh() # To scale images between -1 and 1
        )

    def forward(self, z):
        return self.fc(z).view(-1, 1, 28, 28) # Reshape to image size

# Discriminator Network
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.fc = nn.Sequential(
            nn.Flatten(),
            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 probability of real/fake
        )

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

# Initialize the networks
generator = Generator()
discriminator = Discriminator()

# Loss and Optimizers
criterion = nn.BCELoss()
optimizer_g = optim.Adam(generator.parameters(), lr=0.0002, betas=(0.5, 0.999))
optimizer_d = optim.Adam(discriminator.parameters(), lr=0.0002, betas=(0.5, 0.999))

# Prepare the data
dataset = SimpleDataset(num_samples=1000)
dataloader = DataLoader(dataset, batch_size=64, shuffle=True)
```

2. Training the GAN:

```
python

# Training the GAN
num_epochs = 100
for epoch in range(num_epochs):
    for real_images in dataloader:
        batch_size = real_images.size(0)

        # Create labels for real and fake images
        real_labels = torch.ones(batch_size, 1)
        fake_labels = torch.zeros(batch_size, 1)

        # Train the Discriminator
        optimizer_d.zero_grad()
```



```
# Real images
output_real = discriminator(real_images)
d_loss_real = criterion(output_real, real_labels)

# Fake images
z = torch.randn(batch_size, 100) # Random noise
fake_images = generator(z)
output_fake = discriminator(fake_images.detach()) # Detach to avoid training generator
d_loss_fake = criterion(output_fake, fake_labels)

d_loss = d_loss_real + d_loss_fake
d_loss.backward()
optimizer_d.step()

# Train the Generator
optimizer_g.zero_grad()

output_fake = discriminator(fake_images)
g_loss = criterion(output_fake, real_labels) # We want fake to be classified as real
g_loss.backward()
optimizer_g.step()

# Print progress every few epochs
if epoch % 10 == 0:
    print(f"Epoch [{epoch}/{num_epochs}], d_loss: {d_loss.item()}, g_loss: {g_loss.item()}")
```

3. Generate Synthetic Data:

Once the GAN is trained, we can use the **generator** to generate synthetic data for augmentation:

```
python

# Generate new images
num_samples = 10
z = torch.randn(num_samples, 100) # Random noise
generated_images = generator(z)

# Convert and visualize the generated images
generated_images = generated_images.detach().numpy()
fig, axes = plt.subplots(1, num_samples, figsize=(15, 15))

for i, ax in enumerate(axes):
    ax.imshow(generated_images[i][0], cmap='gray')
    ax.axis('off')

plt.show()
```

4. Analysis of Generated Images:

After training, you can visually analyze the generated images and compare them with the original images from the dataset:

- **Quality:** Observe how well the generator has learned to produce images that resemble the original data (i.e., random shapes or categories).
- **Augmentation Use:** You can use these generated images as additional training data, potentially improving model performance on the original dataset.

Visualizing the Results:

- If the GAN successfully learns to model the data, the generated images should look similar to the dataset images.
- You may need to train for more epochs to improve the quality of generated samples.

Key Insights:

1. **GAN Training Challenges:** GANs can be tricky to train — they require fine-tuning of hyperparameters and balancing the generator and discriminator losses.
2. **Synthetic Data for Augmentation:** The generated samples can be used to augment the training set, especially when the original dataset is small or lacks variety.
3. **Visualization:** It's essential to compare generated samples to real ones and evaluate how well the generator learned the data distribution.

Would you like further clarification or help setting up synthetic datasets or different GAN architectures?



7.B) Train a simple classifier (e.g., a CNN) on a small dataset augmented with synthetic samples from a GAN. Compare the classifier's performance (accuracy, loss) on the augmented dataset versus the original dataset.

7B: Training a Simple Classifier with Augmented Data from GAN

In this task, you will train a simple **classifier (CNN)** on a small dataset that has been **augmented with synthetic samples generated by a GAN**. The goal is to compare the classifier's performance (accuracy and loss) on:

- The **original dataset**.
- The **augmented dataset** (original + synthetic data generated by GAN).

Steps to Implement the Solution:

1. **Create a small dataset** (e.g., random shapes, or simple categories).
2. **Train a GAN** to generate synthetic images for data augmentation.
3. **Use the generator** to create synthetic samples and augment the dataset.
4. **Train a simple CNN classifier** on both the original dataset and the augmented dataset.
5. **Compare performance** in terms of:
 - **Accuracy:** On validation/test data.
 - **Loss:** During training and validation.

Step-by-Step Solution:

1. Create a Simple Dataset

Let's assume we have a small dataset of images with simple categories (e.g., circles, squares, etc.). For simplicity, we'll generate random shapes.

```
python

import torch
import numpy as np
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms

# Generate synthetic dataset (random shapes)
class SimpleDataset(Dataset):
    def __init__(self, num_samples=1000, image_size=28):
        self.num_samples = num_samples
        self.image_size = image_size
        self.transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))])
        self.labels = np.random.randint(0, 2, num_samples) # Binary classes (0 or 1)

    def __len__(self):
        return self.num_samples

    def __getitem__(self, idx):
        # Create a random image (e.g., a circle or square) with random label
        image = np.random.rand(self.image_size, self.image_size) * 255
        label = self.labels[idx]
        return self.transform(image.astype(np.float32)), label

# Initialize dataset
dataset = SimpleDataset(num_samples=1000)
dataloader = DataLoader(dataset, batch_size=64, shuffle=True)
```

2. Train a Simple GAN for Data Augmentation

The GAN architecture is as we discussed in the previous part. After training the GAN, you will use the **generator** to produce synthetic data for augmentation.

```
python

# Assume the GAN is trained and generator is ready
def generate_synthetic_data(generator, num_samples=1000, batch_size=64):
    z = torch.randn(num_samples, 100) # Latent vector
    synthetic_data = generator(z)
    synthetic_labels = np.random.randint(0, 2, num_samples) # Labels for synthetic images
    return synthetic_data, synthetic_labels
```

3. Augment the Dataset with Synthetic Samples

Use the trained GAN generator to augment the dataset with synthetic images.

```
python

# Generate synthetic data
synthetic_images, synthetic_labels = generate_synthetic_data(generator)

# Convert the synthetic images and labels into a PyTorch Dataset
class SyntheticDataset(Dataset):
    def __init__(self, images, labels, transform=None):
        self.images = images
        self.labels = labels
        self.transform = transform
```

```
def __len__(self):
    return len(self.images)

def __getitem__(self, idx):
    image = self.images[idx]
    label = self.labels[idx]
    if self.transform:
        image = self.transform(image)
    return image, label

# Create DataLoader for augmented dataset
augmented_dataset = SyntheticDataset(synthetic_images, synthetic_labels, transform=transforms.ToTensor())
augmented_dataloader = DataLoader(augmented_dataset, batch_size=64, shuffle=True)
```

4. Train a Simple CNN Classifier

Now, define and train a simple CNN classifier using the original dataset and augmented dataset.

```
python

import torch.nn as nn
import torch.optim as optim

# Simple CNN Classifier
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
        self.fc1 = nn.Linear(64 * 7 * 7, 128)
        self.fc2 = nn.Linear(128, 2) # Binary output (0 or 1)
        self.relu = nn.ReLU()
        self.maxpool = nn.MaxPool2d(2)

    def forward(self, x):
        x = self.relu(self.conv1(x))
        x = self.maxpool(x)
        x = self.relu(self.conv2(x))
        x = self.maxpool(x)
        x = x.view(x.size(0), -1) # Flatten
        x = self.relu(self.fc1(x))
        x = self.fc2(x)
        return x

# Initialize model, loss, and optimizer
model = SimpleCNN().cuda()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Training function
def train_model(dataloader, epochs=5):
    model.train()
    total_loss = 0
    total_correct = 0
    total_samples = 0
    for epoch in range(epochs):
        for images, labels in dataloader:
            images, labels = images.cuda(), labels.cuda()

            optimizer.zero_grad()

            outputs = model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            total_loss += loss.item()
            _, predicted = torch.max(outputs, 1)
            total_correct += (predicted == labels).sum().item()
            total_samples += labels.size(0)
```

```

accuracy = total_correct / total_samples
print(f"Epoch {epoch+1}/{epochs}, Loss: {total_loss/len(data_loader)}, Accuracy: {accuracy*100}")
total_loss = 0
total_correct = 0
total_samples = 0

# Train on original dataset
train_model(data_loader)

# Train on augmented dataset (original + synthetic data)
augmented_data_loader = DataLoader(dataset + augmented_dataset, batch_size=64, shuffle=True)
train_model(augmented_data_loader)

```

5. Compare Performance:

- **Accuracy:** Compare the accuracy of the classifier on the original dataset and the augmented dataset.
- **Loss:** Monitor the loss during training to check if adding synthetic data helped reduce loss.

Analysis of Results:

1. Accuracy Comparison:

- If the GAN successfully generates relevant samples, the classifier trained on the augmented dataset should perform better than the one trained on the original dataset.
- The augmented dataset gives the classifier more variety, which can improve generalization and lead to higher accuracy.

2. Loss Comparison:

- Training with augmented data may lead to faster convergence due to the additional diverse samples, especially if the synthetic data fills gaps in the original dataset.

3. Possible Improvements:

- **Better Generator:** Improving the GAN's quality (e.g., using a DCGAN or WGAN) could lead to better synthetic samples, which would improve the classifier's performance.
- **Data Augmentation Techniques:** In addition to GAN-generated samples, consider using traditional data augmentation (e.g., rotation, flipping) for even better generalization.

Key Insights:

- **Data Augmentation with GANs:** GANs can significantly enhance small datasets by providing synthetic samples, which can help improve the performance of classifiers.
- **Model Performance:** By adding synthetic data, the classifier has a larger and more varied training set, which may improve its ability to generalize to new, unseen data.
- **Synthetic Data Quality:** The quality of the synthetic data from the GAN is crucial. Poorly generated images will not help improve the model and could even harm performance.

Would you like to try this setup with a specific dataset or adjust any parameters? Let me know if you'd like help implementing specific parts!



8)

A) Train a Variational Autoencoder (VAE) on a small dataset of images and generate additional samples. Visualize the quality of the generated data and examine how well it complements the original data distribution.

8A: Training a Variational Autoencoder (VAE) on a Small Dataset of Images

In this task, you will train a **Variational Autoencoder (VAE)** on a small dataset of images and then use it to generate additional samples. You will also **visualize** the generated data and examine how well it complements the original data distribution.

Steps to Implement the VAE:

1. **Dataset Preparation:** Select a small dataset of images (e.g., 20-50 images).
2. **Define the VAE architecture:**
 - **Encoder:** Maps the input data to a probabilistic latent space (mean and variance).
 - **Decoder:** Reconstructs the data from the latent space.
3. **Loss Function:**
 - **Reconstruction Loss:** Measures how well the decoder reconstructs the original images.
 - **KL Divergence:** Ensures the latent space follows a normal distribution (regularization).
4. **Train the VAE:** Use the combined loss function to train the model.
5. **Generate Samples:** After training, sample from the latent space and generate new data.
6. **Visualize:** Compare the original and generated samples.

PyTorch Code for a VAE:

1. Define the VAE Architecture

```
python

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

# Define the Encoder Network
class Encoder(nn.Module):
    def __init__(self, latent_dim=2):
        super(Encoder, self).__init__()
        self.fc1 = nn.Linear(28*28, 400)
        self.fc2_mu = nn.Linear(400, latent_dim)
        self.fc2_logvar = nn.Linear(400, latent_dim)

    def forward(self, x):
        x = torch.flatten(x, start_dim=1) # Flatten the input image
        h = torch.relu(self.fc1(x))
```

```

        mu = self.fc2_mu(h)
        logvar = self.fc2_logvar(h)
        return mu, logvar

# Define the Decoder Network
class Decoder(nn.Module):
    def __init__(self, latent_dim=2):
        super(Decoder, self).__init__()
        self.fc1 = nn.Linear(latent_dim, 400)
        self.fc2 = nn.Linear(400, 28*28)

    def forward(self, z):
        h = torch.relu(self.fc1(z))
        x_reconstructed = torch.sigmoid(self.fc2(h))
        return x_reconstructed.view(-1, 1, 28, 28) # Reshape to image size

# Define the VAE Network
class VAE(nn.Module):
    def __init__(self, latent_dim=2):
        super(VAE, self).__init__()
        self.encoder = Encoder(latent_dim)
        self.decoder = Decoder(latent_dim)

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

    def forward(self, x):
        mu, logvar = self.encoder(x)
        z = self.reparameterize(mu, logvar)
        x_reconstructed = self.decoder(z)
        return x_reconstructed, mu, logvar

    def loss_function(self, recon_x, x, mu, logvar):
        BCE = nn.functional.binary_cross_entropy(recon_x.view(-1, 28*28), x.view(-1, 28*28), reduce:
        # KL divergence
        # Compute the KL divergence between q(z|x) and p(z)
        # where p(z) is a standard normal distribution
        # q(z|x) is the distribution parameterized by mu and logvar
        # logvar is the logarithm of the variance
        # We want the latent variable distribution to be close to the standard normal distribution
        # D_KL = 0.5 * sum(1 + log(sigma^2) - mu^2 - sigma^2)
        # where mu, sigma are the mean and std of the latent space
        # logvar = log(sigma^2)
        MSE_loss = BCE
        KL_loss = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
        return MSE_loss + KL_loss

```

2. Prepare Dataset (e.g., MNIST)

For simplicity, we will use **MNIST** as our small dataset. You can replace it with your custom dataset if needed.

```

python

from torchvision import datasets

# Load MNIST dataset
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

train_dataset = datasets.MNIST(root='./data', train=True, download=True, transform=transform)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)

```

3. Train the VAE

```
python

# Initialize model, optimizer, and device
latent_dim = 2
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
vae = VAE(latent_dim).to(device)
optimizer = optim.Adam(vae.parameters(), lr=1e-3)

# Training loop
epochs = 20
for epoch in range(epochs):
    vae.train()
    train_loss = 0
    for batch_idx, (data, _) in enumerate(train_loader):
        data = data.to(device)

        optimizer.zero_grad()

        # Forward pass
        recon_batch, mu, logvar = vae(data)

        # Compute loss
        loss = vae.loss_function(recon_batch, data, mu, logvar)
        loss.backward()

        train_loss += loss.item()
        optimizer.step()

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

4. Generate New Samples from Latent Space

After training, you can sample points from the latent space and decode them to generate new images.

```
python

def generate_samples(model, num_samples=10):
    model.eval()
    with torch.no_grad():
        z = torch.randn(num_samples, latent_dim).to(device) # Latent space samples
        generated_images = model.decoder(z)
    return generated_images

# Generate new samples
generated_images = generate_samples(vae)

# Visualize the generated samples
def visualize_images(images):
    images = images.cpu().numpy()
    fig, axes = plt.subplots(1, len(images), figsize=(15, 5))
    for i, ax in enumerate(axes):
        ax.imshow(images[i][0], cmap='gray')
        ax.axis('off')
    plt.show()

visualize_images(generated_images)
```

5. Analyze the Quality of Generated Data

- **Visualization:** Visualize the original and generated images side-by-side to evaluate how well the VAE has learned to model the data distribution.

- **Quality Check:** If the VAE is well-trained, the generated samples should look like meaningful variations of the original data.
- **Latent Space Exploration:** You can also explore the latent space by sampling along specific axes to see how the model captures different variations of the data (e.g., changing features like size or orientation of digits).

Key Insights and Analysis:

1. Quality of Generated Data:

- The generated samples should resemble the training data if the VAE has learned the data distribution well.
- If the generated samples are blurry or unrealistic, it could indicate that the VAE has not fully captured the data distribution and needs further training or architectural improvements.

2. Latent Space Exploration:

- By sampling from different points in the latent space, you can observe how the VAE interpolates between different images. The smoothness of this transition can indicate the model's ability to generalize and understand the underlying structure of the data.

3. Complementing the Original Data:

- You can use the generated data to augment the original dataset, especially if the original dataset is small. This should ideally improve the performance of classifiers trained on the augmented dataset, as the synthetic samples provide additional diversity.

Conclusion:

The VAE can be an effective method for generating additional data that complements a small original dataset. By learning a compact latent space, it can produce realistic new samples, which can be useful for data augmentation. The visualization of generated samples allows you to evaluate how well the model has learned the data distribution and identify potential areas for improvement.

Would you like to adjust the architecture, try with a different dataset, or explore latent space visualization? Let me know!



8. B) Train a classifier (e.g., CNN) using the dataset augmented with samples generated from a VAE or GAN. Compare the performance of the classifier trained on augmented data against one trained on the original dataset.

8B: Training a Classifier on Augmented Data from VAE or GAN

In this task, you will train a **classifier (CNN)** on a dataset that has been augmented with **synthetic samples generated by a VAE or GAN**. The goal is to compare the performance of the classifier trained on:

1. Original dataset.

2. **Augmented dataset** (original + synthetic data generated by VAE or GAN).

You will compare:

- **Accuracy.**
- **Loss.**
- The **quality** of the generated samples.

Steps to Implement the Solution:

1. **Dataset Preparation:** Use the same dataset as before (e.g., MNIST or your custom dataset).
2. **Train the VAE or GAN** to generate synthetic data (we already discussed this in the previous steps).
3. **Augment the dataset** with synthetic samples generated by the VAE or GAN.
4. **Define and train the CNN classifier** on both:
 - The **original dataset**.
 - The **augmented dataset**.
5. **Compare the classifier's performance** in terms of accuracy, loss, and other metrics.

Step-by-Step Solution:

1. Train a VAE (or GAN) and Generate Synthetic Data

You can use the **VAE** or **GAN** trained earlier to generate synthetic data. For simplicity, let's assume that we have already trained a VAE and generated new samples.

```
python

# Generate new samples using the trained VAE (already defined in previous step)
generated_images = generate_samples(vae, num_samples=1000) # 1000 synthetic samples
generated_labels = np.random.randint(0, 2, 1000) # Random labels for synthetic data (for binary classification)

# Convert the synthetic images into a PyTorch Dataset
class SyntheticDataset(Dataset):
    def __init__(self, images, labels, transform=None):
        self.images = images
        self.labels = labels
        self.transform = transform

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

    def __getitem__(self, idx):
        image = self.images[idx]
        label = self.labels[idx]
        if self.transform:
            image = self.transform(image)
        return image, label

# Create DataLoader for augmented dataset (original + synthetic data)
augmented_dataset = SyntheticDataset(generated_images, generated_labels, transform=transforms.ToTensor())
augmented_dataloader = DataLoader(augmented_dataset, batch_size=64, shuffle=True)
```

2. Define and Train the CNN Classifier

Now, we will define a **simple CNN classifier** and train it on both the **original dataset** and the **augmented dataset**.

```
python

import torch.nn as nn
import torch.optim as optim

# Simple CNN model
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
        self.fc1 = nn.Linear(64 * 7 * 7, 128)
        self.fc2 = nn.Linear(128, 2) # Binary output (0 or 1)
        self.relu = nn.ReLU()
        self.maxpool = nn.MaxPool2d(2)

    def forward(self, x):
        x = self.relu(self.conv1(x))
        x = self.maxpool(x)
        x = self.relu(self.conv2(x))
        x = self.maxpool(x)
        x = x.view(x.size(0), -1) # Flatten
        x = self.relu(self.fc1(x))
        x = self.fc2(x)
        return x

# Initialize model, loss function, and optimizer
model = SimpleCNN().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

3. Train on the Original Dataset

```
python

# Train the CNN on the original dataset (e.g., MNIST)
def train_model(dataloader, model, criterion, optimizer, epochs=10):
    model.train()
    for epoch in range(epochs):
        running_loss = 0.0
        correct = 0
        total = 0
        for images, labels in dataloader:
            images, labels = images.to(device), labels.to(device)

            optimizer.zero_grad()

            # Forward pass
            outputs = model(images)
            loss = criterion(outputs, labels)

            loss.backward()
            optimizer.step()

            running_loss += loss.item()

            # Accuracy calculation
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        epoch_loss = running_loss / len(dataloader.dataset)
        epoch_accuracy = 100 * correct / total
        print(f"Epoch [{epoch+1}/{epochs}], Loss: {epoch_loss:.4f}, Accuracy: {epoch_accuracy:.2f}%")

# Initialize DataLoader for the original dataset (e.g., MNIST)
```

```
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)

# Train on the original dataset
train_model(train_loader, model, criterion, optimizer)
```

4. Train on the Augmented Dataset

To train on the **augmented dataset** (original + synthetic data), we combine both the original data and the generated data into a new DataLoader.

```
python

# Combine the original and augmented datasets
combined_dataset = torch.utils.data.ConcatDataset([train_dataset, augmented_dataset])
combined_dataloader = DataLoader(combined_dataset, batch_size=64, shuffle=True)

# Train the model on the combined dataset (original + synthetic)
train_model(combined_dataloader, model, criterion, optimizer)
```

5. Compare Performance

Now that you have trained the model on both the original and augmented datasets, you can compare the performance of the model in terms of:

- **Accuracy:** Measure how well the classifier performs on the test set (e.g., test accuracy).
- **Loss:** Monitor the loss during training to check how well the model is learning from both real and synthetic data.

```
python

# Evaluate the model on the test dataset
def evaluate_model(test_loader, model):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in test_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    accuracy = 100 * correct / total
    print(f"Test Accuracy: {accuracy:.2f}%")

# Initialize DataLoader for the test dataset (e.g., MNIST test set)
test_dataset = datasets.MNIST(root='./data', train=False, download=True, transform=transform)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)

# Evaluate the model on original dataset
evaluate_model(test_loader, model)

# Evaluate the model on augmented dataset (original + synthetic data)
evaluate_model(test_loader, model)
```

Analysis of Results:

1. Accuracy:

- If the synthetic samples from the VAE or GAN are of high quality, the classifier trained on the augmented dataset should have higher accuracy than the one trained on the original dataset alone.
- The augmented dataset provides more diversity and better generalization, which can help the classifier perform better on unseen data.

2. Loss:

- Monitor the loss curves to see if the augmented data leads to faster convergence. If the synthetic data is realistic and complements the original data, the model may learn more efficiently.

3. Synthetic Data Quality:

- The quality of the generated data plays a crucial role. If the generated samples do not closely resemble real data, they may negatively impact the classifier's performance by introducing noise or irrelevant patterns.
- Visual inspection of synthetic samples can help identify if there are significant issues with the generated data.

Key Insights:

- **Data Augmentation:** Using synthetic data from GANs or VAEs can help improve the performance of classifiers, especially in scenarios with small datasets.
- **Model Performance:** Training on an augmented dataset can enhance the model's ability to generalize, as it exposes the classifier to a wider range of input variations.
- **Quality of Synthetic Data:** The success of data augmentation with generative models relies heavily on the quality of the synthetic samples. Poorly generated data can hurt performance, while high-quality synthetic data can provide substantial benefits.

Would you like to adjust any parameters, experiment with a different dataset, or explore further details on the augmentation techniques? Let me know how you'd like to proceed!



9)
A) Develop a program to evaluate the realism and diversity of video sequences generated by a GAN. Utilize qualitative evaluation methods, such as visual inspection, to assess the quality of the generated frames.

9A: Evaluating the Realism and Diversity of Video Sequences Generated by a GAN

In this task, we'll focus on **evaluating the realism and diversity** of **video sequences** generated by a **Generative Adversarial Network (GAN)**. To assess the quality of the generated video frames, we'll use qualitative evaluation methods like **visual inspection**, **frame-by-frame analysis**, and **diversity checks**.

The evaluation can be broken down into several steps:

1. **Generate Video Sequences with a GAN:** We'll first train a GAN model that generates video frames, or you can use an already trained GAN model.
2. **Visual Inspection:** The first step in evaluating video quality is looking at the video frames generated by the GAN. This helps in assessing if the generated frames look realistic or if they exhibit artifacts like distortion.
3. **Diversity Assessment:** The second step is evaluating the diversity of the video sequence. This ensures that the GAN is not just generating repetitive or identical frames.
4. **Comparison:** Optionally, we can compare the generated video sequences with real video sequences to gauge how realistic and diverse the generated sequences are.

Steps to Implement the Program

1. **Generate Video Sequences:** For simplicity, we will assume that the GAN model has already been trained, and we will use it to generate video frames.
2. **Visual Inspection of Generated Frames:** Displaying the frames allows us to observe the realism of the generated video sequences.
3. **Diversity Assessment:** By analyzing the similarity between consecutive frames, we can check if the video sequence is diverse or too repetitive.
4. **Implementation with Python Code:** We will use **PyTorch** or **TensorFlow** to generate frames and **OpenCV** or **Matplotlib** for displaying and evaluating the video frames.

Implementation:

1. **Define the GAN Model to Generate Video Frames** (using a simple GAN as an example):

```
python

import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import matplotlib.pyplot as plt
import cv2

# Example GAN architecture for generating frames (simplified)
class Generator(nn.Module):
    def __init__(self, z_dim=100):
        super(Generator, self).__init__()
        self.fc1 = nn.Linear(z_dim, 128)
        self.fc2 = nn.Linear(128, 256)
        self.fc3 = nn.Linear(256, 512)
        self.fc4 = nn.Linear(512, 28*28) # Assuming 28x28 frame size

    def forward(self, z):
        z = torch.relu(self.fc1(z))
        z = torch.relu(self.fc2(z))
        z = torch.relu(self.fc3(z))
        return torch.sigmoid(self.fc4(z)).view(-1, 1, 28, 28) # Reshape to 28x28 image

# Initialize the generator
z_dim = 100
generator = Generator(z_dim)
```

2. **Generate Video Frames:** We will generate video sequences (e.g., 30 frames of size 28x28).

```
python

def generate_video(generator, num_frames=30, z_dim=100):
    generator.eval()
    with torch.no_grad():
        video_frames = []
        for _ in range(num_frames):
            z = torch.randn(1, z_dim) # Random noise vector
            frame = generator(z) # Generate a frame
            video_frames.append(frame.squeeze().cpu().numpy()) # Convert to numpy array
        return video_frames

# Generate a sequence of 30 frames
generated_video_frames = generate_video(generator)
```

3. **Visual Inspection of Generated Frames:** Display the generated frames to qualitatively assess their realism.

```
python

def visualize_video_frames(video_frames):
    fig, axes = plt.subplots(5, 6, figsize=(12, 10)) # Assuming 30 frames in a grid
    axes = axes.flatten()
    for i, ax in enumerate(axes):
        ax.imshow(video_frames[i], cmap='gray')
        ax.axis('off')
    plt.show()

# Visualize the first few frames of the generated video
visualize_video_frames(generated_video_frames[:30])
```

4. **Diversity Assessment:** To assess the diversity of the generated frames, we can compute a **pairwise similarity** between consecutive frames using a metric like **Mean Squared Error (MSE)** or **Structural Similarity Index (SSIM)**.

```
python

import skimage.metrics

def calculate_frame_diversity(video_frames):
    diversity_scores = []
    for i in range(1, len(video_frames)):
        frame1 = video_frames[i - 1].flatten()
        frame2 = video_frames[i].flatten()
        # Using Mean Squared Error to compute similarity between consecutive frames
        mse = np.mean((frame1 - frame2) ** 2)
        diversity_scores.append(mse)
    return diversity_scores

# Calculate the diversity of the generated video sequence
diversity_scores = calculate_frame_diversity(generated_video_frames)
print(f"Average Diversity (MSE): {np.mean(diversity_scores)}")
```

- **Interpretation:** A lower MSE value indicates that consecutive frames are more similar (less diverse), while a higher MSE indicates greater diversity between frames.

5. **Optionally, Compare with Real Video Frames:** If you have access to a real video dataset, you can perform a similar evaluation to compare the realism and diversity of the generated video sequences with real videos.

```
python

def load_real_video(video_path):
    cap = cv2.VideoCapture(video_path)
    frames = []
    while(cap.isOpened()):
        ret, frame = cap.read()
        if not ret:
            break
        frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY) # Convert to grayscale
        frame = cv2.resize(frame, (28, 28)) # Resize to 28x28
        frames.append(frame)
    cap.release()
    return frames

# Example of comparing with a real video (path to a video file)
real_video_frames = load_real_video("real_video.mp4")

# Visualize real video frames (first 30 frames)
visualize_video_frames(real_video_frames[:30])
```

Key Evaluation Metrics:

1. Visual Inspection:

- **Realism:** Look at the generated frames. Do they look realistic? Are there any visual artifacts, such as blurriness, noise, or unnatural structures?
- **Consistency:** Are the frames consistent over time, or are there sudden jumps or discontinuities in the video?

2. Diversity Assessment:

- **Frame Similarity:** Calculate the **Mean Squared Error (MSE)** or **SSIM** between consecutive frames. A higher value indicates more diversity, while a lower value indicates similarity.
- **Diversity Score:** Track the diversity score across all frames and look for patterns of consistent diversity or lack thereof.

3. Comparison with Real Videos (Optional):

- **Realism:** How does the generated video sequence compare to real video sequences in terms of appearance, coherence, and smoothness?
- **Temporal Consistency:** Are there any noticeable artifacts when comparing generated and real videos frame by frame?

Conclusion:

- **Realism** is assessed through **visual inspection**, checking if the frames generated look natural and free from artifacts.
- **Diversity** is assessed through **MSE** or **SSIM** scores between consecutive frames to ensure the generated sequence doesn't become overly repetitive.
- For a more in-depth evaluation, you can compare the **generated video** with **real video sequences** to assess how closely the GAN can mimic real-world video dynamics.

This approach will help you assess the quality and diversity of the video sequences produced by the GAN.

Would you like to explore using a more advanced GAN model like **VideoGAN** or improve the diversity evaluation further? Let me know!



9. B) Use the trained VAE to reconstruct both normal and anomalous instances from a small test dataset. Calculate the reconstruction error for each instance and identify potential anomalies based on high reconstruction errors. Visualize the reconstruction error for both normal and anomalous data.

9B: Anomaly Detection with VAE Reconstruction Error

In this task, we'll use a **Variational Autoencoder (VAE)** to detect **anomalous instances** in a dataset by leveraging the **reconstruction error**. The idea is that the VAE will perform well on **normal** data (i.e., data similar to the training distribution) and poorly on **anomalous** data (i.e., data that differs significantly from the training distribution). We will reconstruct both **normal and anomalous instances** from a test dataset, calculate the reconstruction error for each instance, and use this to identify potential anomalies.

Here's a step-by-step breakdown of how we can implement this:

Steps to Implement the Solution:

1. **Train a VAE:** We will assume the VAE has already been trained on a normal dataset, such as MNIST (if using image data).
2. **Reconstruct Data:** We will pass both **normal** and **anomalous** data through the VAE and calculate the **reconstruction error**.
3. **Reconstruction Error Calculation:** The reconstruction error is typically calculated as the **Mean Squared Error (MSE)** between the original input and the reconstructed output.
4. **Anomaly Detection:** Instances with high reconstruction errors will be flagged as anomalies.
5. **Visualization:** We will visualize the reconstruction errors and inspect how well the VAE can separate normal and anomalous data.

Step-by-Step Solution

1. Define the VAE Model (assuming it's already trained)

Let's first define a simple VAE model for reconstruction. If you already have a trained VAE, you can skip the training step.

```
python

import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
```

```
from torch.utils.data import DataLoader
import numpy as np
import matplotlib.pyplot as plt

# VAE Model Definition (Encoder and Decoder)
class VAE(nn.Module):
    def __init__(self, input_dim=784, latent_dim=2):
        super(VAE, self).__init__()
        self.fc1 = nn.Linear(input_dim, 400)
        self.fc21 = nn.Linear(400, latent_dim) # mean of z
        self.fc22 = nn.Linear(400, latent_dim) # log variance of z
        self.fc3 = nn.Linear(latent_dim, 400)
        self.fc4 = nn.Linear(400, input_dim)

    def encode(self, x):
        h1 = F.relu(self.fc1(x))
        return self.fc21(h1), self.fc22(h1)

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

    def decode(self, z):
        h3 = F.relu(self.fc3(z))
        return torch.sigmoid(self.fc4(h3))

    def forward(self, x):
        mu, logvar = self.encode(x.view(-1, 784))
        z = self.reparameterize(mu, logvar)
        return self.decode(z), mu, logvar

# Initialize the VAE model
vae = VAE(latent_dim=2).to(device)
```

2. Reconstruct Data and Calculate Reconstruction Error

To calculate the **reconstruction error**, we will compare the original data (both normal and anomalous) with the reconstructed data using **Mean Squared Error (MSE)**.

```
python

# Reconstruction Error Calculation
def calculate_reconstruction_error(original_data, reconstructed_data):
    return torch.mean((original_data - reconstructed_data) ** 2, dim=1).cpu().numpy()

# Function to get reconstruction errors for normal and anomalous data
def get_reconstruction_errors(model, data_loader):
    model.eval()
    reconstruction_errors = []
    with torch.no_grad():
        for data in data_loader:
            data = data[0].view(-1, 784).to(device)
            reconstructed, _, _ = model(data)
            error = calculate_reconstruction_error(data, reconstructed)
            reconstruction_errors.extend(error)
    return np.array(reconstruction_errors)
```

3. Test Dataset with Normal and Anomalous Data

For the **test dataset**, we will assume you have two sets of data:

- **Normal Data:** This data is from the same distribution as the training set (e.g., normal MNIST digits).

- **Anomalous Data:** This data is different or generated to be anomalous (e.g., random noise or altered data points).

For the sake of the example, we'll create synthetic **anomalous data** that doesn't belong to the MNIST distribution.

```
python

# Assuming you have a test dataset
normal_test_data = torch.randn(100, 1, 28, 28) # Normal data (e.g., from MNIST)
anomalous_test_data = torch.randn(100, 1, 28, 28) + 3 # Anomalous data (e.g., outliers)

# Convert to DataLoader format
normal_test_loader = DataLoader(normal_test_data, batch_size=32, shuffle=False)
anomalous_test_loader = DataLoader(anomalous_test_data, batch_size=32, shuffle=False)
```

4. Calculate Reconstruction Errors for Both Normal and Anomalous Data

Now that we have the normal and anomalous test data loaders, we can calculate the **reconstruction errors**.

```
python

# Calculate reconstruction errors for normal and anomalous data
normal_reconstruction_errors = get_reconstruction_errors(vae, normal_test_loader)
anomalous_reconstruction_errors = get_reconstruction_errors(vae, anomalous_test_loader)

# Visualize the errors
plt.hist(normal_reconstruction_errors, bins=30, alpha=0.5, label='Normal Data')
plt.hist(anomalous_reconstruction_errors, bins=30, alpha=0.5, label='Anomalous Data')
plt.xlabel('Reconstruction Error')
plt.ylabel('Frequency')
plt.title('Reconstruction Error Distribution for Normal and Anomalous Data')
plt.legend()
plt.show()
```

5. Anomaly Detection: Threshold for Anomalies

We can set a **threshold** for the reconstruction error above which we consider the instance as anomalous. This threshold can be chosen based on the histogram distribution or using statistical methods like **percentiles**.

```
python

# Calculate threshold (for example, using the 95th percentile of normal reconstruction errors)
threshold = np.percentile(normal_reconstruction_errors, 95)

# Classify data points as normal or anomalous based on reconstruction error
normal_anomalies = normal_reconstruction_errors > threshold
anomalous_anomalies = anomalous_reconstruction_errors > threshold

# Count the number of anomalies
normal_anomaly_count = np.sum(normal_anomalies)
anomalous_anomaly_count = np.sum(anomalous_anomalies)

print(f"Anomalies in normal data: {normal_anomaly_count}")
print(f"Anomalies in anomalous data: {anomalous_anomaly_count}")
```

6. Visualize Individual Reconstructions

To further inspect how the VAE handles specific instances, you can visualize the **original data** and its **reconstruction** for both normal and anomalous instances.

```
python

def visualize_reconstruction(data, reconstructed, num_samples=5):
    fig, axes = plt.subplots(2, num_samples, figsize=(12, 5))
    for i in range(num_samples):
        axes[0, i].imshow(data[i].view(28, 28).cpu().numpy(), cmap='gray')
        axes[0, i].axis('off')
        axes[1, i].imshow(reconstructed[i].view(28, 28).cpu().numpy(), cmap='gray')
        axes[1, i].axis('off')
    plt.show()

# Visualize some normal data and its reconstruction
normal_data_batch = next(iter(normal_test_loader))
normal_data = normal_data_batch[0].view(-1, 784).to(device)
reconstructed_normal, _, _ = vae(normal_data)
visualize_reconstruction(normal_data, reconstructed_normal)

# Visualize some anomalous data and its reconstruction
anomalous_data_batch = next(iter(anomalous_test_loader))
anomalous_data = anomalous_data_batch[0].view(-1, 784).to(device)
reconstructed_anomalous, _, _ = vae(anomalous_data)
visualize_reconstruction(anomalous_data, reconstructed_anomalous)
```

Key Insights:

1. **Reconstruction Error:** The main idea here is that the VAE will have a higher reconstruction error for anomalous data because it has not been trained on such data. This high error can be used as a signal for anomaly detection.
2. **Threshold Selection:** You can experiment with the **percentile** or **statistical methods** to find an optimal threshold for classifying anomalies. The choice of threshold depends on the desired sensitivity and specificity of the anomaly detection system.
3. **Visualization:** Visualizing the reconstruction errors helps in identifying which instances have high errors, and comparing normal vs anomalous data gives you a better understanding of how the VAE perceives anomalies.

Would you like to extend this with other anomaly detection methods or dive deeper into tuning the VAE for anomaly detection? Let me know how you'd like to proceed!



10)

A) Train a VAE on normal data. For each test instance, calculate the Euclidean distance between its latent representation and the normal data's latent representations. Classify instances as anomalous if the distance exceeds a threshold.

10A: Anomaly Detection using VAE Latent Space with Euclidean Distance

In this task, we will use the **latent space** of a **Variational Autoencoder (VAE)** for anomaly detection. After training the VAE on **normal data**, we will:

1. **Extract the latent representations** of both the training data and the test instances.
2. **Calculate the Euclidean distance** between the latent representation of each test instance and the latent representations of the training data.
3. **Classify test instances as anomalous** if their latent representation exceeds a specified **distance threshold** from the training latent space.

The basic assumption is that the VAE will learn a compact and well-structured latent space for the **normal data**. Anomalous instances, which are far from the training distribution, will have large distances in this space.

Steps to Implement the Solution:

1. **Train a VAE:** Train a VAE on the normal data (e.g., MNIST).
2. **Extract Latent Representations:** For both the training and test datasets, extract the latent representations from the VAE's encoder.
3. **Calculate Euclidean Distance:** For each test instance, calculate the **Euclidean distance** between its latent representation and the latent representations of the normal training data.
4. **Anomaly Detection:** Classify an instance as anomalous if the distance exceeds a certain threshold.

Implementation

1. Define the VAE Model (Encoder and Decoder)

We'll assume the VAE model is already defined and trained on **normal data**.

```
python

import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import numpy as np

# VAE Model Definition (Encoder and Decoder)
class VAE(nn.Module):
    def __init__(self, input_dim=784, latent_dim=2):
        super(VAE, self).__init__()
        self.fc1 = nn.Linear(input_dim, 400)
        self.fc21 = nn.Linear(400, latent_dim) # mean of z
        self.fc22 = nn.Linear(400, latent_dim) # log variance of z
        self.fc3 = nn.Linear(latent_dim, 400)
        self.fc4 = nn.Linear(400, input_dim)

    def encode(self, x):
        h1 = F.relu(self.fc1(x))
        return self.fc21(h1), self.fc22(h1)

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

    def decode(self, z):
        h3 = F.relu(self.fc3(z))
        return torch.sigmoid(self.fc4(h3))

    def forward(self, x):
```

```
mu, logvar = self.encode(x.view(-1, 784))
z = self.reparameterize(mu, logvar)
return self.decode(z), mu, logvar

# Initialize the VAE model (assuming it's already trained)
vae = VAE(latent_dim=2).to(device)
```

2. Train the VAE on Normal Data

We will assume that the VAE has already been trained on normal data. If you're not familiar with this step, it typically involves using an image dataset like MNIST and training the VAE to reconstruct the data.

```
python

# Placeholder for training the VAE (assuming this part is already done)
# You would typically train your VAE on normal data like MNIST.
# For simplicity, we're assuming the VAE is trained here.
```

3. Extract Latent Representations

Once the model is trained, we can pass the normal data through the encoder to extract the latent space representations. We will use these representations to compute the **Euclidean distance**.

```
python

def extract_latent_representations(data_loader, model):
    latent_representations = []
    model.eval()
    with torch.no_grad():
        for data in data_loader:
            data = data[0].view(-1, 784).to(device)
            mu, logvar = model.encode(data)
            latent_representations.append(mu.cpu().numpy())
    return np.vstack(latent_representations)

# Example: Extract latent representations for the training data
normal_train_loader = torch.utils.data.DataLoader(normal_train_data, batch_size=32, shuffle=False)
latent_normal_train = extract_latent_representations(normal_train_loader, vae)
```

4. Calculate Euclidean Distance for Test Instances

Now, for each **test instance**, we calculate its latent representation and compare it with all the latent representations from the training data using the **Euclidean distance**.

```
python

from scipy.spatial.distance import cdist

def calculate_anomalies(test_data_loader, latent_train_data, model, threshold=1.0):
    anomalies = []
    model.eval()
    with torch.no_grad():
        for data in test_data_loader:
            data = data[0].view(-1, 784).to(device)
            mu, _ = model.encode(data)
            # Calculate the Euclidean distance between test instance and all training latent representations
            distances = cdist(mu.cpu().numpy(), latent_train_data, 'euclidean')
            # For each test instance, find the minimum distance to any training latent representation
            min_distances = distances.min(axis=1)
            anomalies.extend(min_distances)
```

```
# Classify instances as anomalous if their distance exceeds the threshold
anomaly_labels = [1 if dist > threshold else 0 for dist in anomalies] # 1 for anomaly, 0 for normal
return anomalies, anomaly_labels

# Example: Test data loader (anomalous and normal test data)
test_loader = torch.utils.data.DataLoader(test_data, batch_size=32, shuffle=False)

# Calculate anomalies with a threshold (e.g., 1.0 for Euclidean distance)
anomalies, anomaly_labels = calculate_anomalies(test_loader, latent_normal_train, vae, threshold=1.0)
```

5. Visualize Anomalies

We can visualize the **Euclidean distances** of test instances and highlight the anomalous ones.

```
python

import matplotlib.pyplot as plt

# Visualizing Euclidean distances and anomalies
def visualize_anomalies(anomalies, anomaly_labels):
    plt.figure(figsize=(10, 6))
    plt.hist(anomalies, bins=30, alpha=0.7, color='g', label='Normal Instances')
    plt.scatter(np.array(range(len(anomalies)))[np.array(anomaly_labels) == 1],
                np.array(anomalies)[np.array(anomaly_labels) == 1],
                color='r', label='Anomalous Instances', alpha=0.7)
    plt.xlabel('Euclidean Distance to Normal Latent Space')
    plt.ylabel('Frequency')
    plt.legend()
    plt.title('Anomaly Detection using VAE Latent Space')
    plt.show()

# Visualize the results
visualize_anomalies(anomalies, anomaly_labels)
```

6. Evaluation

To evaluate how well the model classifies normal vs anomalous instances, we can compare the **anomaly labels** against the true labels if available.

```
python

from sklearn.metrics import classification_report

# Assuming you have true labels for the test data
true_labels = np.array([...]) # 0 for normal, 1 for anomalous

# Print classification report (precision, recall, f1-score)
print(classification_report(true_labels, anomaly_labels))
```

Summary of Steps:

1. **Train the VAE** on normal data.
2. **Extract the latent representations** of both the training data and the test instances from the VAE encoder.
3. **Calculate the Euclidean distance** between the test instance's latent representation and the latent representations of the normal data.
4. **Classify instances as anomalous** if their Euclidean distance exceeds a predefined threshold.

5. **Visualize the anomaly detection results**, plotting the distances and highlighting anomalous instances.

Conclusion:

This approach uses the **latent space** learned by the VAE to detect anomalies by measuring the **Euclidean distance** between test instances and the normal data's latent representations. This method is effective when normal data is well-represented in the latent space, and anomalies can be detected as outliers.

Would you like to further refine this method, tune the threshold, or explore another anomaly detection technique? Let me know!



10, B) Implement a PyTorch Encoder that maps a musical sequence (length L) to a latent space (dimension D) and a reparameterize function to sample from the resulting Gaussian distribution.

10B: Implementing a PyTorch Encoder for a Musical Sequence to Latent Space with Reparameterization

In this task, we will create a **PyTorch Encoder** that maps a **musical sequence** (length L) into a latent space of dimension **D**. The encoder will learn a distribution over the latent space (in the form of a **Gaussian distribution**), and we will implement a **reparameterization** trick to sample from this latent distribution.

This setup is useful for tasks like **Variational Autoencoders (VAE)** applied to music generation or anomaly detection in musical sequences, where we want to learn a meaningful representation of musical data in a lower-dimensional latent space.

Steps to Implement the Solution:

1. **Encoder Architecture:** We'll implement an encoder that takes in a sequence of musical data (such as MIDI or raw audio data), processes it through a few layers (like **LSTM** or **GRU**), and produces two outputs: the **mean** and **log variance** of a Gaussian distribution in latent space.
2. **Reparameterization Trick:** This is used to allow backpropagation through the latent variable sampling. The idea is to sample from the Gaussian distribution defined by the mean and variance by using the **reparameterization trick**:

$$z = \mu + \sigma \cdot \epsilon$$

where:

- μ is the mean of the latent distribution
- σ is the standard deviation (exponent of log variance)
- ϵ is random noise sampled from a standard normal distribution.

3. **Latent Space:** After reparameterization, we will sample from the learned Gaussian distribution and obtain a point in the latent space.

Implementation in PyTorch

Let's define an **Encoder** using **LSTM** layers (since musical sequences are sequential), and implement the reparameterization trick.

Step-by-Step Code

```
python

import torch
import torch.nn as nn
import torch.nn.functional as F

class MusicVAEEncoder(nn.Module):
    def __init__(self, input_dim, hidden_dim, latent_dim):
        super(MusicVAEEncoder, self).__init__()

        # LSTM Layer for encoding the musical sequence
        self.lstm = nn.LSTM(input_dim, hidden_dim, batch_first=True)

        # Fully connected layers for the mean and log variance of the latent space
        self.fc_mu = nn.Linear(hidden_dim, latent_dim)
        self.fc_logvar = nn.Linear(hidden_dim, latent_dim)

    def forward(self, x):
        """
        Forward pass through the encoder.

        Parameters:
        - x: Musical sequence of shape (batch_size, sequence_length, input_dim)

        Returns:
        - mu: Mean of the latent distribution (batch_size, latent_dim)
        - logvar: Log variance of the latent distribution (batch_size, latent_dim)
        """
        # LSTM Layer outputs
        lstm_out, (h_n, c_n) = self.lstm(x) # h_n: (num_layers, batch_size, hidden_dim)

        # The final hidden state for the sequence is used to generate the latent variables
        final_hidden_state = h_n[-1] # Take the last hidden state (for many-to-one case)

        # Mean and Log Variance for the latent space
        mu = self.fc_mu(final_hidden_state) # Mean
        logvar = self.fc_logvar(final_hidden_state) # Log variance

        return mu, logvar

    def reparameterize(mu, logvar):
        """
        Reparameterization trick: Sample from the Gaussian distribution defined by mu and logvar.

        Parameters:
        - mu: Mean of the latent distribution (batch_size, latent_dim)
        - logvar: Log variance of the latent distribution (batch_size, latent_dim)

        Returns:
        - z: Sampled latent vector (batch_size, latent_dim)
        """
        std = torch.exp(0.5 * logvar) # Standard deviation
        eps = torch.randn_like(std) # Random noise
        z = mu + eps * std # Reparameterization trick
        return z
```

Explanation of the Code:

1. Encoder (LSTM):

- We use an **LSTM** layer to process the musical sequence. The LSTM takes in the musical sequence (of length L and input dimension $input_dim$) and outputs the hidden states for each timestep.
- For simplicity, we use the last hidden state (`h_n[-1]`) as a representation of the entire sequence. This is a common approach for many-to-one sequence-to-latent mapping (e.g., for VAE).

2. Latent Variables (Mean and Log Variance):

- The final hidden state from the LSTM is passed through two fully connected layers (`fc_mu` and `fc_logvar`) to produce the **mean** and **log variance** of the latent Gaussian distribution.

3. Reparameterization Trick:

- The `reparameterize()` function samples from the latent space by using the **reparameterization trick**, which allows for backpropagation through the sampling process.
- This is done by sampling from a standard normal distribution and scaling it by the standard deviation (calculated from `logvar`), then shifting by the mean (`mu`).

Usage Example:

Let's define some parameters for our input and use the encoder to process a musical sequence.

```
python

# Parameters
input_dim = 128 # Input dimension, e.g., number of MIDI notes (for simplicity, assume 128)
hidden_dim = 256 # Hidden dimension in LSTM
latent_dim = 10 # Latent space dimension

# Create the Encoder model
encoder = MusicVAEEncoder(input_dim, hidden_dim, latent_dim)

# Example musical sequence (batch_size, sequence_length, input_dim)
# Let's assume a sequence of 50 time steps and 128 possible MIDI values for each time step.
batch_size = 32
sequence_length = 50
music_sequence = torch.randn(batch_size, sequence_length, input_dim) # Random sequence as example

# Forward pass through the encoder
mu, logvar = encoder(music_sequence)

# Reparameterization to sample latent vector z
z = reparameterize(mu, logvar)

# Print the results
print(f"Mean (mu): {mu.shape}")
print(f"Log Variance (logvar): {logvar.shape}")
print(f"Sampled Latent Vector (z): {z.shape}")
```

Explanation of the Example:

- 1. Input Sequence:** A batch of musical sequences with `batch_size = 32`, `sequence_length = 50` (50 timesteps), and `input_dim = 128` (which could represent 128 MIDI notes or other features).
- 2. Output:** The encoder outputs the **mean (mu)** and **log variance (logvar)** of the latent space for each sequence in the batch. These are used to sample latent vectors `z`.
- 3. Reparameterization:** Using the `reparameterize()` function, we sample a latent vector `z` from the Gaussian distribution defined by `mu` and `logvar`.

Further Notes:

- This is just the **encoder** part of the VAE model. To complete the VAE, you would need to define a **decoder** that takes in the latent vectors `z` and reconstructs the musical sequence (or other desired output).
- The **latent dimension** (`latent_dim`) can be adjusted based on how compact or expressive you want the latent space to be.
- The **input dimension** (`input_dim`) might vary depending on the features you extract from the music (e.g., MIDI note values, one-hot encoding of notes, spectrogram features, etc.).

Conclusion:

This implementation provides an **encoder** for mapping musical sequences into a latent space using an **LSTM** network and the **reparameterization trick** to sample from the resulting Gaussian distribution. This framework is useful for applications like music generation, style transfer, or anomaly detection in musical data.

If you need help in setting up the decoder or other parts of the system, feel free to ask!



11)

A) Implement a GAN to generate target domain samples from source domain data. Train and visualize the generated target domain samples.

11A: Implementing a GAN to Generate Target Domain Samples from Source Domain Data

In this task, we will implement a **Generative Adversarial Network (GAN)** that learns to generate **target domain samples** from **source domain data**. Essentially, this setup aims to map source domain data to the distribution of the target domain using GANs. This is similar to **domain adaptation** or **image-to-image translation** tasks.

We will:

1. **Define a Generator** that learns to map source domain data to the target domain.
2. **Define a Discriminator** that distinguishes between generated target samples and real target samples.
3. **Train the GAN** to generate convincing target domain samples from source domain data.
4. **Visualize the generated samples** to see how well the model maps the source domain to the target domain.

Steps:

1. **Prepare Data:** For simplicity, assume we have access to two datasets: a source domain and a target domain. Each of these datasets consists of images (e.g., source domain = photos, target domain = sketches).

2. **Define the Generator:** The generator should take in a **source domain image** (or a random noise vector, in some cases) and output a **generated target domain image**.
3. **Define the Discriminator:** The discriminator will classify whether a given image is a **real target domain image** or a **fake (generated) target domain image**.
4. **Train the GAN:** We will train the generator and discriminator in tandem. The generator will aim to fool the discriminator, and the discriminator will attempt to distinguish real target domain images from generated ones.
5. **Visualize the Results:** Once the GAN is trained, we'll visualize the generated target domain samples.

Step-by-Step Code Implementation

1. Define the Generator and Discriminator

We will use simple fully connected layers for this implementation. However, for more complex image-to-image translation tasks (like photo-to-sketch), we might use **Convolutional Neural Networks (CNNs)**. Below is a basic example using fully connected layers.

```
python

import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt

# Define the Generator
class Generator(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(Generator, self).__init__()
        self.fc1 = nn.Linear(input_dim, 128)
        self.fc2 = nn.Linear(128, 256)
        self.fc3 = nn.Linear(256, output_dim)
        self.relu = nn.ReLU()
        self.tanh = nn.Tanh() # To output images in the [-1, 1] range

    def forward(self, z):
        x = self.relu(self.fc1(z))
        x = self.relu(self.fc2(x))
        x = self.tanh(self.fc3(x))
        return x

# Define the Discriminator
class Discriminator(nn.Module):
    def __init__(self, input_dim):
        super(Discriminator, self).__init__()
        self.fc1 = nn.Linear(input_dim, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 1)
        self.relu = nn.ReLU()
        self.sigmoid = nn.Sigmoid() # For binary classification (real or fake)

    def forward(self, x):
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.fc3(x)
        x = self.sigmoid(x)
        return x
```

2. Training the GAN

To train the GAN, we need to alternate between training the discriminator and the generator. The discriminator is trained to classify real and fake images, while the generator is trained to generate images that can fool the discriminator.

python

```
# Training the GAN
def train_gan(generator, discriminator, real_data, num_epochs=1000, batch_size=64, learning_rate=0.001):
    criterion = nn.BCELoss()
    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))

    # Labels for real and fake data
    real_label = 1
    fake_label = 0

    for epoch in range(num_epochs):
        for i, data in enumerate(real_data):
            # Create real labels (real images)
            real_images = data
            real_labels = torch.full((batch_size,), real_label)

            # Generate fake images using the generator
            z = torch.randn(batch_size, 100) # Random noise
            fake_images = generator(z)
            fake_labels = torch.full((batch_size,), fake_label)

            # --- Train the Discriminator ---
            optimizer_d.zero_grad()

            # Real images loss
            output_real = discriminator(real_images)
            loss_real = criterion(output_real.view(-1), real_labels)

            # Fake images loss
            output_fake = discriminator(fake_images.detach())
            loss_fake = criterion(output_fake.view(-1), fake_labels)

            # Total discriminator loss
            d_loss = loss_real + loss_fake
            d_loss.backward()
            optimizer_d.step()

            # --- Train the Generator ---
            optimizer_g.zero_grad()

            # We want to fool the discriminator, so use real labels for fake images
            output_fake_for_g = discriminator(fake_images)
            g_loss = criterion(output_fake_for_g.view(-1), real_labels)

            g_loss.backward()
            optimizer_g.step()

        # Print the losses every 100 epochs
        if epoch % 100 == 0:
            print(f"Epoch [{epoch}/{num_epochs}] | D Loss: {d_loss.item()} | G Loss: {g_loss.item()}")

        # Generate and visualize samples every few epochs
        if epoch % 200 == 0:
            generate_samples(generator)

# Function to visualize generated samples
def generate_samples(generator, num_samples=5, noise_dim=100):
    z = torch.randn(num_samples, noise_dim)
    generated_images = generator(z)
    generated_images = generated_images.detach().numpy()

    # Visualizing the generated samples
    fig, axes = plt.subplots(1, num_samples, figsize=(15, 5))
    for i in range(num_samples):
        axes[i].imshow(generated_images[i].reshape(28, 28), cmap='gray') # Assuming 28x28 images
        axes[i].axis('off')
```

```
plt.show()

# Example: Set the input and output dimensions
input_dim = 100 # Latent space dimension (random noise)
output_dim = 28 * 28 # Assuming target domain images are 28x28 (for example, MNIST)

# Create the models
generator = Generator(input_dim, output_dim)
discriminator = Discriminator(output_dim)

# Dummy dataset for the source domain (using random data for simplicity)
# Replace this with your actual source and target domain data
real_data = torch.randn(1000, output_dim) # Replace with actual target domain data

# Train the GAN
train_gan(generator, discriminator, real_data, num_epochs=1000)
```

3. Visualize the Results

After training, you can visualize the generated images using the `generate_samples()` function, which will display **generated target domain samples**.

Explanation:

1. **Generator:** The generator takes in random noise `z` as input and generates an image in the target domain. It consists of three fully connected layers with ReLU activations, and the final output is passed through a `tanh()` to map the pixel values to the range `[-1, 1]`.
2. **Discriminator:** The discriminator takes an image as input and outputs a probability that the image is real (from the target domain) or fake (generated by the generator). It consists of three fully connected layers with ReLU activations and uses a `sigmoid()` activation for the final output.

3. Training Loop:

- The discriminator is trained to distinguish between **real target images** and **fake target images**.
 - The generator is trained to **fool the discriminator**, i.e., generate fake images that are classified as real.
 - The **binary cross-entropy loss (BCE)** is used for both the generator and discriminator.
4. **Visualization:** Every 200 epochs, we generate and display a few target domain samples to visualize the progress of the generator.

Conclusion:

This implementation trains a **GAN** that generates **target domain images** from **source domain data**. After training, you can visualize the output images and observe how well the generator has learned to map source data to the target domain. You can further improve the model by using more complex architectures (e.g., **DCGAN** for image generation) and fine-tuning the training process.

If you need any adjustments, feel free to ask!



11. B) Write a program to extend a basic GAN model to generate video sequences (e.g., 5-10 frames of simple motion). Train the model and generate a sample video sequence. Display or save the

generated frames as output.

11B: Extending a Basic GAN to Generate Video Sequences

In this task, we will extend the basic **GAN model** to generate **video sequences**. Specifically, we want to generate a sequence of frames (e.g., 5-10 frames) that exhibit simple motion, such as a moving object or pattern. This can be useful for tasks like **video generation**, **motion modeling**, and **video prediction**.

Approach:

1. **Video Representation:** In a video, each frame is essentially an image, but a sequence of frames represents temporal changes. A video can be represented as a 4D tensor: `(batch_size, num_frames, height, width, channels)`.
2. **Generator and Discriminator Modifications:**
 - The **Generator** will take in random noise (or a latent vector) and output a sequence of frames instead of a single frame.
 - The **Discriminator** will classify whether the sequence of frames is real (from the training dataset) or fake (generated by the GAN).
3. **Model Architecture:** We'll modify the GAN's **Generator** and **Discriminator** to handle video data. The generator will produce multiple frames, while the discriminator will classify a sequence of frames.
4. **Training:** The GAN will be trained to generate sequences of frames that resemble real video sequences. The generator will learn to fool the discriminator, and the discriminator will distinguish between real and fake sequences.
5. **Visualization:** After training, we can generate a video sequence and visualize it.

Step-by-Step Code Implementation

We'll define a simple **3D Convolutional GAN** (3D-CGAN) model to handle video sequences.

1. Define the Generator and Discriminator

```
python

import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
import numpy as np

# Define the Generator for video sequences (using 3D convolutions)
class VideoGenerator(nn.Module):
    def __init__(self, input_dim, num_frames, image_size, channels):
        super(VideoGenerator, self).__init__()
        self.num_frames = num_frames
        self.image_size = image_size
        self.channels = channels

        # Fully connected layers to project latent vector to 3D tensor
        self.fc1 = nn.Linear(input_dim, 256)
        self.fc2 = nn.Linear(256, 512)
        self.fc3 = nn.Linear(512, 1024)
```

```
# Reshape the output into a shape that can be processed by 3D Conv layers
self.deconv1 = nn.ConvTranspose3d(1024, 512, kernel_size=4, stride=2, padding=1)
self.deconv2 = nn.ConvTranspose3d(512, 256, kernel_size=4, stride=2, padding=1)
self.deconv3 = nn.ConvTranspose3d(256, channels, kernel_size=4, stride=2, padding=1)

self.relu = nn.ReLU()
self.tanh = nn.Tanh() # Output video frames in [-1, 1] range

def forward(self, z):
    x = self.relu(self.fc1(z))
    x = self.relu(self.fc2(x))
    x = self.relu(self.fc3(x))
    x = x.view(-1, 1024, 1, 1, 1) # Reshape to match 3D Conv input

    x = self.relu(self.deconv1(x))
    x = self.relu(self.deconv2(x))
    x = self.tanh(self.deconv3(x))

    return x

# Define the Discriminator for video sequences (using 3D convolutions)
class VideoDiscriminator(nn.Module):
    def __init__(self, num_frames, image_size, channels):
        super(VideoDiscriminator, self).__init__()
        self.num_frames = num_frames
        self.image_size = image_size
        self.channels = channels

        self.conv1 = nn.Conv3d(channels, 64, kernel_size=4, stride=2, padding=1)
        self.conv2 = nn.Conv3d(64, 128, kernel_size=4, stride=2, padding=1)
        self.conv3 = nn.Conv3d(128, 256, kernel_size=4, stride=2, padding=1)
        self.conv4 = nn.Conv3d(256, 512, kernel_size=4, stride=2, padding=1)

        self.fc = nn.Linear(512 * (image_size // 16) * (image_size // 16) * num_frames, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.relu(self.conv1(x))
        x = self.relu(self.conv2(x))
        x = self.relu(self.conv3(x))
        x = self.relu(self.conv4(x))

        x = x.view(x.size(0), -1) # Flatten for fully connected layer
        x = self.fc(x)

        return self.sigmoid(x)
```

2. Training the GAN

Now that we've defined the generator and discriminator, we need to train the GAN. The training process involves alternating between training the discriminator and the generator.

```
python

# Training the GAN to generate video sequences
def train_gan(generator, discriminator, num_epochs=1000, batch_size=16, learning_rate=0.0002, noise_criterion = nn.BCELoss()):
    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))

    # Labels for real and fake data
    real_label = 1
    fake_label = 0

    for epoch in range(num_epochs):
        for i in range(batch_size):
            # Create real video sequences (dummy data for simplicity)
            real_video = torch.randn(batch_size, num_frames, channels, image_size, image_size) # Real video
            real_labels = torch.full((batch_size,), real_label)
```



```
# Generate fake video sequences using the generator
z = torch.randn(batch_size, noise_dim)
fake_video = generator(z)
fake_labels = torch.full((batch_size,), fake_label)

# --- Train the Discriminator ---
optimizer_d.zero_grad()

# Real video loss
output_real = discriminator(real_video)
loss_real = criterion(output_real.view(-1), real_labels)

# Fake video loss
output_fake = discriminator(fake_video.detach())
loss_fake = criterion(output_fake.view(-1), fake_labels)

# Total discriminator loss
d_loss = loss_real + loss_fake
d_loss.backward()
optimizer_d.step()

# --- Train the Generator ---
optimizer_g.zero_grad()

# We want to fool the discriminator, so use real labels for fake video sequences
output_fake_for_g = discriminator(fake_video)
g_loss = criterion(output_fake_for_g.view(-1), real_labels)

g_loss.backward()
optimizer_g.step()

# Print the losses every 100 epochs
if epoch % 100 == 0:
    print(f"Epoch [{epoch}/{num_epochs}] | D Loss: {d_loss.item()} | G Loss: {g_loss.item()}")

# Generate and visualize video samples every 200 epochs
if epoch % 200 == 0:
    generate_video_samples(generator, num_samples=1)

# Function to visualize the generated video sequences
def generate_video_samples(generator, num_samples=1, noise_dim=100, num_frames=10, image_size=64, channels=3):
    z = torch.randn(num_samples, noise_dim)
    generated_video = generator(z)

    # Visualizing the generated video frames
    fig, axes = plt.subplots(1, num_frames, figsize=(15, 5))
    for i in range(num_frames):
        axes[i].imshow(generated_video[0, i].detach().numpy().transpose(1, 2, 0)) # Transpose to HWC
        axes[i].axis('off')
    plt.show()
```

3. Running the Training Loop

```
python

# Set the parameters
input_dim = 100 # Latent vector dimension
num_frames = 10 # Number of frames in the video
image_size = 64 # Size of each frame
channels = 3 # RGB channels
batch_size = 16

# Create the models
generator = VideoGenerator(input_dim, num_frames, image_size, channels)
discriminator = VideoDiscriminator(num_frames, image_size, channels)

# Train the GAN
train_gan(generator, discriminator, num_epochs=1000, batch_size=batch_size, noise_dim=input_dim, num_frames=num_frames)
```

Explanation:

1. Generator:

- The generator takes in a latent vector of random noise and processes it through several fully connected layers. It is reshaped and passed through 3D transpose convolution layers to generate a sequence of video frames.

2. Discriminator:

- The discriminator processes the video sequence through several 3D convolution layers and classifies it as real or fake.

3. Training:

- During training, the discriminator is trained to distinguish real video sequences from fake sequences, while the generator is trained to generate realistic sequences that fool the discriminator.

4. Visualization:

- After training, the `generate_video_samples()` function generates a video sequence (e.g., 10 frames) and visualizes the frames.

Conclusion:

This code implements a basic **3D GAN** model that generates **video sequences** by extending the traditional GAN architecture to handle temporal data. The generator creates sequences of frames, and the discriminator classifies whether the sequence is real or fake. After training, you can visualize the generated video sequences and see how well the model has learned to generate motion.

If you need to refine the model or use real video datasets, you can modify the data pipeline accordingly.



12)

A) Write a program to generate novel music sequences by sampling from the latent space of the trained VAE. Convert the generated latent vectors back into music sequences (e.g., MIDI or simple waveforms) and play or visualize the output.

To generate novel music sequences using a Variational Autoencoder (VAE), we need to train the VAE on a music dataset, sample from the latent space, and convert the generated latent vectors back into music sequences. This process involves a few key steps:

- 1. Train a VAE on Music Data:** We first need to train a VAE using a music dataset, such as MIDI files. The VAE will learn to encode music sequences into a latent space and reconstruct them back.
- 2. Sampling from the Latent Space:** After training, we can sample from the latent space of the VAE to generate new music sequences.
- 3. Convert Latent Vectors to Music Sequences:** We need to convert the latent vectors (the generated music) back into a playable format such as MIDI or waveform data.
- 4. Play or Visualize the Output:** Finally, we'll need to either play the generated music or visualize it (for example, using a piano roll or a similar representation).

Step-by-Step Code Implementation

1. Install Necessary Libraries

You will need the following libraries:

- `torch`: for building the VAE model
- `mido`: for MIDI handling
- `pretty_midi`: for converting MIDI data into a usable format
- `pydub` or `librosa`: for playing and handling audio if necessary

```
bash

pip install torch mido pretty_midi pydub librosa
```

2. Define the VAE Architecture

We'll define the VAE model that can process music data. We will use a simple encoder-decoder architecture. The encoder will map music sequences (e.g., piano roll, or note sequences) into a latent space, and the decoder will generate music from these latent vectors.

```
python

import torch
import torch.nn as nn
import torch.optim as optim
import pretty_midi
import numpy as np
import random

# Define VAE architecture
class MusicVAE(nn.Module):
    def __init__(self, input_dim, latent_dim):
        super(MusicVAE, self).__init__()
        self.latent_dim = latent_dim

        # Encoder
        self.fc1 = nn.Linear(input_dim, 512)
        self.fc2_mean = nn.Linear(512, latent_dim)
        self.fc2_logvar = nn.Linear(512, latent_dim)

        # Decoder
        self.fc3 = nn.Linear(latent_dim, 512)
        self.fc4 = nn.Linear(512, input_dim)

    def encode(self, x):
        h1 = torch.relu(self.fc1(x))
        return self.fc2_mean(h1), self.fc2_logvar(h1)

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

    def decode(self, z):
        h3 = torch.relu(self.fc3(z))
        return torch.sigmoid(self.fc4(h3))

    def forward(self, x):
        mu, logvar = self.encode(x)
```

```
z = self.reparameterize(mu, logvar)
return self.decode(z), mu, logvar
```

3. Define the Loss Function

The VAE loss is composed of two parts:

- **Reconstruction loss:** Measures how well the decoder reconstructs the input.
- **KL divergence:** Measures how much the learned latent distribution diverges from a standard normal distribution.

python

```
def vae_loss(reconstructed_x, x, mu, logvar):
    reconstruction_loss = nn.BCELoss(reduction='sum')(reconstructed_x, x)
    kl_loss = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
    return reconstruction_loss + kl_loss
```

4. Training the VAE

We will train the VAE on a music dataset (e.g., piano rolls converted into numerical sequences). Below is a training loop.

python

```
def train_vae(model, train_loader, epochs=100, lr=0.001):
    optimizer = optim.Adam(model.parameters(), lr=lr)
    for epoch in range(epochs):
        model.train()
        total_loss = 0
        for data in train_loader:
            data = data.view(-1, input_dim) # Flatten the music data (piano roll)
            optimizer.zero_grad()
            reconstructed_x, mu, logvar = model(data)
            loss = vae_loss(reconstructed_x, data, mu, logvar)
            loss.backward()
            optimizer.step()
            total_loss += loss.item()

        print(f"Epoch {epoch+1}, Loss: {total_loss/len(train_loader)}")
```

5. Sampling from the Latent Space

Once the VAE is trained, we can sample latent vectors from the latent space and decode them back into music sequences.

python

```
def generate_music(model, num_samples=1):
    model.eval()
    with torch.no_grad():
        z = torch.randn(num_samples, model.latent_dim) # Sample from standard normal distribution
        generated_data = model.decode(z)
    return generated_data
```

6. Convert Latent Vectors to Music Sequences (MIDI)

We need to convert the generated latent vectors back into a format that can be played as music (e.g., MIDI). Here we assume the model generates piano rolls (binary matrices) where each row corresponds to a time step and each column corresponds to a note.

```
python

def piano_roll_to_midi(piano_roll, output_filename="generated_music.mid"):
    midi = pretty_midi.PrettyMIDI()
    piano_program = pretty_midi.program_to_instrument_name(0)
    piano = pretty_midi.Instrument(program=piano_program)

    for i, time_step in enumerate(piano_roll):
        for note_idx, note_on in enumerate(time_step):
            if note_on:
                note = pretty_midi.Note(velocity=100, pitch=note_idx + 60, start=i * 0.1, end=(i + 1) * 0.1)
                piano.notes.append(note)

    midi.instruments.append(piano)
    midi.write(output_filename)

# Example usage:
generated_music = generate_music(model)
generated_piano_roll = generated_music[0].detach().numpy() # Convert tensor to numpy array

# Convert to MIDI
piano_roll_to_midi(generated_piano_roll)
```

7. Play the Generated Music

To play the generated music, you can use `pydub` or `librosa` to load and play the MIDI file.

```
python

from pydub import AudioSegment
from pydub.playback import play

def play_midi(midi_filename="generated_music.mid"):
    midi_audio = AudioSegment.from_file(midi_filename, format="mid")
    play(midi_audio)

# Play the generated music
play_midi("generated_music.mid")
```

Complete Example

Here's a complete workflow:

```
python

# Initialize and train the VAE
input_dim = 128 # Example input dimension (e.g., 128 notes)
latent_dim = 10 # Latent space dimension
model = MusicVAE(input_dim=input_dim, latent_dim=latent_dim)

# Assuming train_loader is a DataLoader that loads music data as sequences
train_vae(model, train_loader)

# Generate music from the trained VAE
generated_music = generate_music(model)
generated_piano_roll = generated_music[0].detach().numpy()

# Convert the generated music to MIDI
piano_roll_to_midi(generated_piano_roll)
```

```
# Play the generated MIDI
play_midi("generated_music.mid")
```

Explanation:

1. **MusicVAE:** The VAE model consists of an encoder that learns a latent space representation of music and a decoder that reconstructs music from the latent space. The latent space captures the underlying structure of music.
2. **vae_loss:** The VAE loss consists of a reconstruction term (measuring how well the model recreates the input) and a KL divergence term (which regularizes the latent space).
3. **Training:** The model is trained using a data loader containing music sequences. You would train the model with music data such as piano rolls.
4. **Music Generation:** Once the model is trained, we can generate new music by sampling from the latent space and decoding the latent vectors back into music sequences.
5. **MIDI Conversion:** The generated music is represented as a piano roll (a binary matrix), which is then converted to MIDI format and saved as a `.mid` file.
6. **Playing Music:** The MIDI file can be played back using `pydub` or any MIDI player.

Conclusion:

This approach uses a VAE to generate novel music sequences by sampling from the latent space and converting the samples into MIDI format for playback. The music generation process involves training a VAE on a dataset of music, sampling from the learned latent space, and decoding the samples into music sequences.



12. B) Write a program to implement a basic Variational Autoencoder (VAE) architecture using TensorFlow. Train the VAE model on a small set of MIDI or audio samples. Use the trained model to generate new latent space representations.

To implement a basic **Variational Autoencoder (VAE)** in **TensorFlow** and train it on MIDI or audio samples, we need to follow these steps:

1. **Prepare the Dataset:** We'll use a set of MIDI or audio samples. For simplicity, let's assume we're using a dataset of piano rolls (a binary matrix representing the presence/absence of notes at each time step).
2. **VAE Model Architecture:** We'll define the VAE model with an encoder and decoder. The encoder maps input data to a latent space (mean and log variance), and the decoder reconstructs the input from a sampled latent vector.
3. **Training the VAE:** We'll train the model using the reconstruction loss and the Kullback-Leibler (KL) divergence.
4. **Generate New Latent Representations:** After training, we'll sample latent vectors from the latent space and generate new data from those latent vectors.

Here's the code implementation for this task:

1. Install Necessary Libraries

First, install TensorFlow and any required libraries for MIDI processing (if necessary):

```
bash

pip install tensorflow pretty_midi
```

2. Define the VAE Architecture

Here, we'll define a simple VAE with fully connected layers for both the encoder and decoder. We'll assume that the input data is a piano roll (a 128-dimensional vector per time step).

```
python

import tensorflow as tf
from tensorflow.keras import layers, models
import numpy as np
import pretty_midi
import random

# Define the VAE model
class VAE(tf.keras.Model):
    def __init__(self, latent_dim):
        super(VAE, self).__init__()
        self.latent_dim = latent_dim

    # Encoder (input -> latent space)
    self.encoder = models.Sequential([
        layers.InputLayer(input_shape=(128,)), # Assuming input is a 128-dimensional piano roll
        layers.Dense(512, activation='relu'),
        layers.Dense(256, activation='relu'),
        layers.Dense(latent_dim + latent_dim) # latent_dim mean + latent_dim logvar
    ])

    # Decoder (latent space -> reconstructed input)
    self.decoder = models.Sequential([
        layers.InputLayer(input_shape=(latent_dim,)),
        layers.Dense(256, activation='relu'),
        layers.Dense(512, activation='relu'),
        layers.Dense(128, activation='sigmoid') # Output should match the input dimension
    ])

    def encode(self, x):
        z_mean_log_var = self.encoder(x)
        z_mean = z_mean_log_var[:, :self.latent_dim]
        z_log_var = z_mean_log_var[:, self.latent_dim:]
        return z_mean, z_log_var

    def reparameterize(self, z_mean, z_log_var):
        epsilon = tf.random.normal(shape=tf.shape(z_mean))
        return z_mean + tf.exp(0.5 * z_log_var) * epsilon

    def decode(self, z):
        return self.decoder(z)

    def call(self, x):
        z_mean, z_log_var = self.encode(x)
        z = self.reparameterize(z_mean, z_log_var)
        reconstructed_x = self.decode(z)
        return reconstructed_x, z_mean, z_log_var
```

3. Define the Loss Function

The VAE loss consists of two parts:

- **Reconstruction loss:** Measures how well the decoder reconstructs the input.
- **KL divergence:** Measures how much the latent distribution differs from a standard normal distribution.

```
python

def vae_loss(reconstructed_x, x, z_mean, z_log_var):
    reconstruction_loss = tf.reduce_mean(tf.reduce_sum(
        tf.keras.losses.binary_crossentropy(x, reconstructed_x), axis=1))

    kl_loss = -0.5 * tf.reduce_mean(tf.reduce_sum(
        1 + z_log_var - tf.square(z_mean) - tf.exp(z_log_var), axis=1))

    return reconstruction_loss + kl_loss
```

4. Prepare the Dataset

Assume you have MIDI files, and you want to convert them into piano rolls (binary matrices where each row represents a time step and each column represents a note). We'll use `pretty_midi` to extract MIDI data and convert it into a piano roll.

```
python

def midi_to_piano_roll(midi_file, resolution=16):
    midi_data = pretty_midi.PrettyMIDI(midi_file)
    piano_roll = midi_data.get_piano_roll(fs=resolution)
    piano_roll = piano_roll > 0 # Binary piano roll (on/off notes)
    return piano_roll.T # Transpose to shape [time_steps, num_notes]

def load_data(midi_files, resolution=16):
    piano_rolls = []
    for midi_file in midi_files:
        piano_roll = midi_to_piano_roll(midi_file, resolution)
        piano_rolls.append(piano_roll)

    # Pad sequences to have consistent shape
    max_length = max([piano_roll.shape[0] for piano_roll in piano_rolls])
    padded_piano_rolls = [np.pad(pr, ((0, max_length - pr.shape[0]), (0, 0)), mode='constant') for pr in piano_rolls]

    return np.array(padded_piano_rolls)

# Example usage
midi_files = ["path_to_midi_1.mid", "path_to_midi_2.mid"]
data = load_data(midi_files)
```

5. Train the VAE

Now we can set up the training process. We'll use `Adam` optimizer and train the VAE on the MIDI data.

```
python

# Parameters
latent_dim = 10 # Latent space dimension
epochs = 50
batch_size = 2

# Initialize the VAE model
vae = VAE(latent_dim)
vae.compile(optimizer='adam', loss=None)
```



```
# Train the model
vae.fit(data, data, epochs=epochs, batch_size=batch_size)
```

6. Generate New Latent Representations

After training, we can sample new latent vectors from a standard normal distribution and decode them into new music sequences.

```
python

def generate_latent_samples(vae, num_samples=5):
    z_samples = tf.random.normal(shape=(num_samples, latent_dim))
    generated_data = vae.decode(z_samples)
    return generated_data

# Generate new music from latent space
generated_music = generate_latent_samples(vae, num_samples=5)

# Visualize or save the generated music
# Convert the generated music back to MIDI (assuming it's in piano roll format)
def piano_roll_to_midi(piano_roll, output_filename="generated_music.mid"):
    midi = pretty_midi.PrettyMIDI()
    piano_program = pretty_midi.program_to_instrument_name(0)
    piano = pretty_midi.Instrument(program=piano_program)

    for i, time_step in enumerate(piano_roll):
        for note_idx, note_on in enumerate(time_step):
            if note_on:
                note = pretty_midi.Note(velocity=100, pitch=note_idx + 60, start=i * 0.1, end=(i + 1) * 0.1)
                piano.notes.append(note)

    midi.instruments.append(piano)
    midi.write(output_filename)

# Example: Save the generated music to a MIDI file
generated_piano_roll = generated_music[0].numpy()
piano_roll_to_midi(generated_piano_roll, output_filename="generated_music.mid")
```

7. Play or Visualize the Generated Music

After generating new MIDI sequences from the VAE's latent space, you can save the sequences to `.mid` files and play them or visualize them using any MIDI player or visualization tool.

Complete Example

Here's the full example in a concise form:

```
python

import tensorflow as tf
from tensorflow.keras import layers, models
import pretty_midi
import numpy as np

# Define the VAE model
class VAE(tf.keras.Model):
    def __init__(self, latent_dim):
        super(VAE, self).__init__()
        self.latent_dim = latent_dim
        self.encoder = models.Sequential([
            layers.InputLayer(input_shape=(128,)),
            layers.Dense(512, activation='relu'),
            layers.Dense(256, activation='relu'),
            layers.Dense(latent_dim + latent_dim)
```

```

    ])

    self.decoder = models.Sequential([
        layers.InputLayer(input_shape=(latent_dim,)),
        layers.Dense(256, activation='relu'),
        layers.Dense(512, activation='relu'),
        layers.Dense(128, activation='sigmoid')
    ])

    def encode(self, x):
        z_mean_log_var = self.encoder(x)
        z_mean = z_mean_log_var[:, :self.latent_dim]
        z_log_var = z_mean_log_var[:, self.latent_dim:]
        return z_mean, z_log_var

    def reparameterize(self, z_mean, z_log_var):
        epsilon = tf.random.normal(shape=tf.shape(z_mean))
        return z_mean + tf.exp(0.5 * z_log_var) * epsilon

    def decode(self, z):
        return self.decoder(z)

    def call(self, x):
        z_mean, z_log_var = self.encode(x)
        z = self.reparameterize(z_mean, z_log_var)
        reconstructed_x = self.decode(z)
        return reconstructed_x, z_mean, z_log_var

# Loss function
def vae_loss(reconstructed_x, x, z_mean, z_log_var):
    reconstruction_loss = tf.reduce_mean(tf.reduce_sum(
        tf.keras.losses.binary_crossentropy(x, reconstructed_x), axis=1))

    kl_loss = -0.5 * tf.reduce_mean(tf.reduce_sum(
        1 + z_log_var - tf.square(z_mean) - tf.exp(z_log_var), axis=1))

    return reconstruction_loss + kl_loss

# Load Data (assuming a set of MIDI files)
def midi_to_piano_roll(midi_file, resolution=16):
    midi_data = pretty_midi.PrettyMIDI(midi_file)
    piano_roll = midi_data.get_piano_roll(fs=resolution)
    piano_roll = piano_roll > 0
    return piano_roll.T

def load_data(midi_files, resolution=16):
    piano_rolls = []
    for midi_file in midi_files:
        piano_roll = midi_to_piano_roll(midi_file, resolution)
        piano_rolls.append(piano_roll)

    max_length = max([piano_roll.shape[0] for piano_roll in piano_rolls])
    padded_piano_rolls = [np.pad(pr, ((0, max_length - pr.shape[0]), (0, 0)), mode='constant') for pr in piano_rolls]
    return np.array(padded_piano_rolls)

# Training VAE
latent_dim = 10
vae = VAE(latent_dim)
vae.compile(optimizer='adam', loss=None)

# Example: Train VAE on data
midi_files = ["path_to_midi_1.mid", "path_to_midi_2.mid"]
data = load_data(midi_files)
vae.fit(data, data, epochs=50, batch_size=2)

# Generate new music sequences from latent space
generated_music = generate_latent_samples(vae, num_samples=5)

```

Conclusion:

This implementation creates a **Variational Autoencoder (VAE)** model in **TensorFlow**, trains it on a set of MIDI piano roll data, and uses it to generate new latent space representations, which are then decoded into new music sequences. You can save these sequences as MIDI files and play or visualize them using MIDI players or music software.