NormalizingFlowsFinal

September 1, 2024

1 Enhancing Variational Autoencoder with Normalizing Flows:

2 A Multivariate g-k Distribution and Levenberg-Marquardt Optimization Approach for MNIST Data Generation and Reconstruction

2.0.1 Overview

This project enhances a Variational Autoencoder (VAE) with Normalizing Flows using a Multivariate g-k Distribution and Levenberg-Marquardt (LM) optimization, applied to the MNIST dataset.

Key Innovations:

1. **Multivariate g-k Distribution**:
   - Integrated as the base distribution in Normalizing Flows, the g-k distribution allows for flexible modeling of the latent space, capturing complex data structures more effectively than a standard normal distribution.
2. **Levenberg-Marquardt Optimization**:
   - The g-k distribution parameters are optimized using LM, a precise method for non-linear parameter estimation. This optimization improves the flow's ability to model intricate patterns in the data, leading to better generation and reconstruction performance.

Implementation Highlights:

- **VAE Structure**:
  - **Encoder**: Compresses input images to a latent space.
  - **Normalizing Flows**: Transforms the latent space using invertible flows, with the g-k distribution as the base.
  - **Decoder**: Reconstructs images from the transformed latent space.
- **Training**:
  - The VAE is trained on MNIST, with the enhanced latent space enabling superior image generation.

This approach focuses on the advanced use of the Multivariate g-k Distribution and LM optimization, making the VAE more expressive and capable of generating realistic, complex data patterns.

2.0.2 Introduction to Normalizing Flows

Normalizing Flows are techniques used to transform a simple probability distribution into a more complex one through a series of invertible transformations, enhancing the model's ability to capture intricate data patterns.

Key Concepts:

- **Base Distribution**: Typically starts with a simple multivariate Gaussian distribution.
- **Flow Layers**: A sequence of invertible transformations applied to the base distribution to create a more complex distribution.
- **Invertibility and Jacobian**: Ensures transformations are reversible and computationally efficient.

2.0.3 Enhanced VAEs with Multivariate g-k Distribution and LM Optimization

In this approach, the base distribution in Normalizing Flows is replaced with a **Multivariate g-k Distribution**. The parameters of this distribution are optimized using **Levenberg-Marquardt (LM)**, leading to a more expressive latent space and improved generative performance in VAEs.

[1]:
```
### Data Preparation

# We start by loading and preprocessing the MNIST dataset.

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


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


train_dataset = datasets.MNIST(root='./data', train=True, transform=transform,␣
 ↪download=True)
test_dataset = datasets.MNIST(root='./data', train=False, transform=transform,␣
 ↪download=True)


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

[2]:
```
# Implementing the basic VAE model
# We start by defining the encoder and decoder components.


import torch.nn as nn
```

```
import torch.nn.functional as F

class VAE(nn.Module):
    def __init__(self, input_dim=784, hidden_dim=400, latent_dim=20):
        super(VAE, self).__init__()

        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc21 = nn.Linear(hidden_dim, latent_dim)
        self.fc22 = nn.Linear(hidden_dim, latent_dim)


        self.fc3 = nn.Linear(latent_dim, hidden_dim)
        self.fc4 = nn.Linear(hidden_dim, 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
```

[3]:
```
def loss_function(recon_x, x, mu, logvar):
    BCE = F.binary_cross_entropy(recon_x, x.view(-1, 784), reduction='sum')
    # KL
    KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
    return BCE + KLD

#
model = VAE()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
```

[4]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
```

```
from torch.utils.data import DataLoader
from torchvision.utils import save_image


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


transform = transforms.Compose([
    transforms.ToTensor(),
])

train_dataset = datasets.MNIST(root='./data', train=True, transform=transform,␣
 ↪download=True)
test_dataset = datasets.MNIST(root='./data', train=False, transform=transform,␣
 ↪download=True)

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)


batch_size = 64

data_loader = DataLoader(dataset=train_dataset, batch_size=batch_size,␣
 ↪shuffle=True)

class VAE(nn.Module):
    def __init__(self, input_dim=784, hidden_dim=400, latent_dim=20):
        super(VAE, self).__init__()

        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc21 = nn.Linear(hidden_dim, latent_dim)
        self.fc22 = nn.Linear(hidden_dim, latent_dim)


        self.fc3 = nn.Linear(latent_dim, hidden_dim)
        self.fc4 = nn.Linear(hidden_dim, 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))  #     [0, 1]

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


model = VAE().to(device)
optimizer = optim.Adam(model.parameters(), lr=1e-3)


def loss_function(recon_x, x, mu, logvar):

    BCE = F.binary_cross_entropy(recon_x, x.view(-1, 784), reduction='sum')
    KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
    return BCE + KLD

def train(epoch):
    model.train()
    train_loss = 0
    for batch_idx, (data, _) in enumerate(train_loader):
        data = data.to(device)
        optimizer.zero_grad()
        recon_batch, mu, logvar = model(data)
        loss = loss_function(recon_batch, data, mu, logvar)
        loss.backward()
        train_loss += loss.item()
        optimizer.step()
        if batch_idx % 100 == 0:
            print(f'Train Epoch: {epoch} [{batch_idx * len(data)}/
 ↪{len(train_loader.dataset)}]\tLoss: {loss.item() / len(data):.6f}')

    print(f'====> Epoch: {epoch} Average loss: {train_loss / len(train_loader.
 ↪dataset):.4f}')

def test(epoch):
    model.eval()
    test_loss = 0
    with torch.no_grad():
        for i, (data, _) in enumerate(test_loader):
            data = data.to(device)
            recon_batch, mu, logvar = model(data)
            test_loss += loss_function(recon_batch, data, mu, logvar).item()

            if i == 0:
```

```
                n = min(data.size(0), 8)
                comparison = torch.cat([data[:n], recon_batch.view(64, 1, 28,␣
↪28)[:n]])
                save_image(comparison.cpu(), 'reconstruction_' + str(epoch) + '.
↪png', nrow=n)

    test_loss /= len(test_loader.dataset)
    print(f'====> Test set loss: {test_loss:.4f}')


for epoch in range(1, 11):
    train(epoch)
    test(epoch)
```

```
Train Epoch: 1 [0/60000]        Loss: 551.919250
Train Epoch: 1 [6400/60000]     Loss: 182.184097
Train Epoch: 1 [12800/60000]    Loss: 160.768753
Train Epoch: 1 [19200/60000]    Loss: 146.986557
Train Epoch: 1 [25600/60000]    Loss: 134.226578
Train Epoch: 1 [32000/60000]    Loss: 127.335480
Train Epoch: 1 [38400/60000]    Loss: 132.698273
Train Epoch: 1 [44800/60000]    Loss: 126.019135
Train Epoch: 1 [51200/60000]    Loss: 123.994339
Train Epoch: 1 [57600/60000]    Loss: 120.055176
====> Epoch: 1 Average loss: 146.4137
====> Test set loss: 119.4359
Train Epoch: 2 [0/60000]        Loss: 117.034378
Train Epoch: 2 [6400/60000]     Loss: 114.082657
Train Epoch: 2 [12800/60000]    Loss: 111.612244
Train Epoch: 2 [19200/60000]    Loss: 116.158981
Train Epoch: 2 [25600/60000]    Loss: 117.521660
Train Epoch: 2 [32000/60000]    Loss: 117.935181
Train Epoch: 2 [38400/60000]    Loss: 114.167076
Train Epoch: 2 [44800/60000]    Loss: 113.666748
Train Epoch: 2 [51200/60000]    Loss: 113.663788
Train Epoch: 2 [57600/60000]    Loss: 111.847023
====> Epoch: 2 Average loss: 115.4172
====> Test set loss: 111.8258
Train Epoch: 3 [0/60000]        Loss: 113.436684
Train Epoch: 3 [6400/60000]     Loss: 109.977821
Train Epoch: 3 [12800/60000]    Loss: 116.638069
Train Epoch: 3 [19200/60000]    Loss: 112.055923
Train Epoch: 3 [25600/60000]    Loss: 111.278572
Train Epoch: 3 [32000/60000]    Loss: 107.828056
Train Epoch: 3 [38400/60000]    Loss: 101.563728
Train Epoch: 3 [44800/60000]    Loss: 111.164284
Train Epoch: 3 [51200/60000]    Loss: 115.798515
```

```
Train Epoch: 3 [57600/60000]    Loss: 110.774406
====> Epoch: 3 Average loss: 111.1608
====> Test set loss: 109.1689
Train Epoch: 4 [0/60000]        Loss: 104.054428
Train Epoch: 4 [6400/60000]     Loss: 113.087830
Train Epoch: 4 [12800/60000]    Loss: 106.848816
Train Epoch: 4 [19200/60000]    Loss: 105.746925
Train Epoch: 4 [25600/60000]    Loss: 111.881042
Train Epoch: 4 [32000/60000]    Loss: 116.088043
Train Epoch: 4 [38400/60000]    Loss: 115.689270
Train Epoch: 4 [44800/60000]    Loss: 107.331833
Train Epoch: 4 [51200/60000]    Loss: 105.795410
Train Epoch: 4 [57600/60000]    Loss: 108.316017
====> Epoch: 4 Average loss: 109.1728
====> Test set loss: 107.8859
Train Epoch: 5 [0/60000]        Loss: 105.699699
Train Epoch: 5 [6400/60000]     Loss: 109.980339
Train Epoch: 5 [12800/60000]    Loss: 111.590363
Train Epoch: 5 [19200/60000]    Loss: 108.682861
Train Epoch: 5 [25600/60000]    Loss: 111.743774
Train Epoch: 5 [32000/60000]    Loss: 110.864693
Train Epoch: 5 [38400/60000]    Loss: 106.427513
Train Epoch: 5 [44800/60000]    Loss: 107.434837
Train Epoch: 5 [51200/60000]    Loss: 111.152748
Train Epoch: 5 [57600/60000]    Loss: 105.850449
====> Epoch: 5 Average loss: 108.0194
====> Test set loss: 107.1133
Train Epoch: 6 [0/60000]        Loss: 103.628647
Train Epoch: 6 [6400/60000]     Loss: 103.281281
Train Epoch: 6 [12800/60000]    Loss: 105.837486
Train Epoch: 6 [19200/60000]    Loss: 108.514496
Train Epoch: 6 [25600/60000]    Loss: 106.206909
Train Epoch: 6 [32000/60000]    Loss: 107.776558
Train Epoch: 6 [38400/60000]    Loss: 103.632133
Train Epoch: 6 [44800/60000]    Loss: 107.543137
Train Epoch: 6 [51200/60000]    Loss: 105.774567
Train Epoch: 6 [57600/60000]    Loss: 104.187729
====> Epoch: 6 Average loss: 107.1999
====> Test set loss: 106.6534
Train Epoch: 7 [0/60000]        Loss: 101.738907
Train Epoch: 7 [6400/60000]     Loss: 106.827431
Train Epoch: 7 [12800/60000]    Loss: 108.391525
Train Epoch: 7 [19200/60000]    Loss: 105.876915
Train Epoch: 7 [25600/60000]    Loss: 109.297836
Train Epoch: 7 [32000/60000]    Loss: 110.194351
Train Epoch: 7 [38400/60000]    Loss: 108.250587
Train Epoch: 7 [44800/60000]    Loss: 100.728836
Train Epoch: 7 [51200/60000]    Loss: 111.466003
```

```
Train Epoch: 7 [57600/60000]    Loss: 102.639236
====> Epoch: 7 Average loss: 106.6182
====> Test set loss: 106.1118
Train Epoch: 8 [0/60000]        Loss: 109.441437
Train Epoch: 8 [6400/60000]     Loss: 108.864639
Train Epoch: 8 [12800/60000]    Loss: 106.313835
Train Epoch: 8 [19200/60000]    Loss: 107.764008
Train Epoch: 8 [25600/60000]    Loss: 105.121384
Train Epoch: 8 [32000/60000]    Loss: 100.524666
Train Epoch: 8 [38400/60000]    Loss: 104.180557
Train Epoch: 8 [44800/60000]    Loss: 109.969673
Train Epoch: 8 [51200/60000]    Loss: 106.098488
Train Epoch: 8 [57600/60000]    Loss: 104.331429
====> Epoch: 8 Average loss: 106.1324
====> Test set loss: 105.6353
Train Epoch: 9 [0/60000]        Loss: 103.444931
Train Epoch: 9 [6400/60000]     Loss: 104.519173
Train Epoch: 9 [12800/60000]    Loss: 111.503349
Train Epoch: 9 [19200/60000]    Loss: 101.432533
Train Epoch: 9 [25600/60000]    Loss: 110.798203
Train Epoch: 9 [32000/60000]    Loss: 108.348862
Train Epoch: 9 [38400/60000]    Loss: 106.928055
Train Epoch: 9 [44800/60000]    Loss: 97.816780
Train Epoch: 9 [51200/60000]    Loss: 108.421631
Train Epoch: 9 [57600/60000]    Loss: 100.129257
====> Epoch: 9 Average loss: 105.7254
====> Test set loss: 105.3206
Train Epoch: 10 [0/60000]       Loss: 103.752029
Train Epoch: 10 [6400/60000]    Loss: 108.575836
Train Epoch: 10 [12800/60000]   Loss: 106.196259
Train Epoch: 10 [19200/60000]   Loss: 106.441734
Train Epoch: 10 [25600/60000]   Loss: 107.783966
Train Epoch: 10 [32000/60000]   Loss: 106.886963
Train Epoch: 10 [38400/60000]   Loss: 104.403137
Train Epoch: 10 [44800/60000]   Loss: 112.906784
Train Epoch: 10 [51200/60000]   Loss: 108.084442
Train Epoch: 10 [57600/60000]   Loss: 104.700577
====> Epoch: 10 Average loss: 105.3731
====> Test set loss: 104.9960
```

During the training process, we observed the following:

1. **Gradual Decrease in Loss**: As training progresses, both the training loss and test loss decrease steadily. This indicates that the VAE model is effectively learning the data distribution, and its reconstruction ability is improving over time.
2. **Convergence Speed**: In the initial epochs, the loss decreases rapidly and then gradually stabilizes. This is a common pattern in model training, indicating that the model is progressively converging.

3. **Final Loss Values**: By the 10th epoch, the average loss on the training set is 105.62, while the average loss on the test set is 104.96. These values suggest that the model has reached a relatively stable state but may still have room for further optimization.

4. **Performance**: Although the model continues to improve, the rate of loss reduction slows down in the later epochs, indicating that the current model architecture might have some limitations in further reducing the loss on the MNIST dataset. This could be related to the latent space dimensions or the design of the encoder and decoder.

2.0.4 Planar Flows

We are introducing Planar Flows into the VAE model to enhance the flexibility and expressiveness of the latent space.

What are Planar Flows? Planar Flows are a simple yet effective type of Normalizing Flow. They apply a series of invertible linear transformations to the latent variables, making the latent space distribution more complex and better at capturing data distribution.

[5]:
```
import torch
import torch.nn as nn

class PlanarFlow(nn.Module):
    def __init__(self, latent_dim):
        super(PlanarFlow, self).__init__()
        self.u = nn.Parameter(torch.randn(latent_dim))
        self.w = nn.Parameter(torch.randn(latent_dim))
        self.b = nn.Parameter(torch.randn(1))

    def forward(self, z):
        # z shape: (batch_size, latent_dim)
        linear = torch.matmul(z, self.w) + self.b  # shape: (batch_size)
        activation = torch.tanh(linear)  # shape: (batch_size)

        # Add a dimension to u to enable broadcasting, i.e., (latent_dim,) ->␣
↪(1, latent_dim)
        z_new = z + self.u * activation.unsqueeze(-1)  # shape: (batch_size,␣
↪latent_dim)

        # Compute psi, which will have shape (batch_size, latent_dim)
        psi = (1 - torch.tanh(linear).pow(2)).unsqueeze(-1) * self.w  # shape:␣
↪(batch_size, latent_dim)

        # Compute the determinant of the Jacobian, shape: (batch_size,)
        det_jacobian = torch.abs(1 + torch.matmul(psi, self.u.unsqueeze(-1)).
↪squeeze(-1))

        return z_new, det_jacobian
```

[6]:
```
class VAEWithFlows(nn.Module):
    def __init__(self, input_dim=784, hidden_dim=400, latent_dim=20,␣
 ↪num_flows=2):
        super(VAEWithFlows, self).__init__()
        self.latent_dim = latent_dim
        self.num_flows = num_flows


        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc21 = nn.Linear(hidden_dim, latent_dim)
        self.fc22 = nn.Linear(hidden_dim, latent_dim)


        self.fc3 = nn.Linear(latent_dim, hidden_dim)
        self.fc4 = nn.Linear(hidden_dim, input_dim)

        # Planar Flows
        self.flows = nn.ModuleList([PlanarFlow(latent_dim) for _ in␣
 ↪range(num_flows)])

    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)
        z0 = mu + eps * std

        log_det_jacobian = 0
        z = z0


        for flow in self.flows:
            z, det_jacobian = flow(z)
            log_det_jacobian += torch.log(det_jacobian + 1e-6)

        return z, log_det_jacobian

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

[7]:
```
def loss_function_with_flows(recon_x, x, mu, logvar, log_det_jacobian):
    BCE = F.binary_cross_entropy(recon_x, x.view(-1, 784), reduction='sum')
    KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())


    KLD = KLD - log_det_jacobian.sum()

    return BCE + KLD
```

[8]:
```
model = VAEWithFlows().to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)

def train(epoch):
    model.train()
    train_loss = 0
    for batch_idx, (data, _) in enumerate(train_loader):
        data = data.to(device)
        optimizer.zero_grad()
        recon_batch, mu, logvar, log_det_jacobian = model(data)
        loss = loss_function_with_flows(recon_batch, data, mu, logvar,␣
 ↪log_det_jacobian)
        loss.backward()
        train_loss += loss.item()
        optimizer.step()
        if batch_idx % 100 == 0:
            print(f'Train Epoch: {epoch} [{batch_idx * len(data)}/
 ↪{len(train_loader.dataset)}]\tLoss: {loss.item() / len(data):.6f}')

    print(f'====> Epoch: {epoch} Average loss: {train_loss / len(train_loader.
 ↪dataset):.4f}')

def test(epoch):
    model.eval()
    test_loss = 0
    with torch.no_grad():
        for i, (data, _) in enumerate(test_loader):
            data = data.to(device)
            recon_batch, mu, logvar, log_det_jacobian = model(data)
            test_loss += loss_function_with_flows(recon_batch, data, mu,␣
 ↪logvar, log_det_jacobian).item()
```

```
            if i == 0:
                n = min(data.size(0), 8)
                comparison = torch.cat([data[:n], recon_batch.view(64, 1, 28,␣
↪28)[:n]])
                save_image(comparison.cpu(), 'reconstruction_with_flows_' +␣
↪str(epoch) + '.png', nrow=n)

    test_loss /= len(test_loader.dataset)
    print(f'====> Test set loss: {test_loss:.4f}')

for epoch in range(1, 11):
    train(epoch)
    test(epoch)
```

```
Train Epoch: 1 [0/60000]        Loss: 556.946838
Train Epoch: 1 [6400/60000]     Loss: 196.831177
Train Epoch: 1 [12800/60000]    Loss: 162.402649
Train Epoch: 1 [19200/60000]    Loss: 153.010284
Train Epoch: 1 [25600/60000]    Loss: 142.892868
Train Epoch: 1 [32000/60000]    Loss: 133.016876
Train Epoch: 1 [38400/60000]    Loss: 129.801010
Train Epoch: 1 [44800/60000]    Loss: 126.503456
Train Epoch: 1 [51200/60000]    Loss: 131.550217
Train Epoch: 1 [57600/60000]    Loss: 127.067749
====> Epoch: 1 Average loss: 150.5395
====> Test set loss: 123.3879
Train Epoch: 2 [0/60000]        Loss: 117.438263
Train Epoch: 2 [6400/60000]     Loss: 126.851456
Train Epoch: 2 [12800/60000]    Loss: 124.902420
Train Epoch: 2 [19200/60000]    Loss: 122.360931
Train Epoch: 2 [25600/60000]    Loss: 117.555168
Train Epoch: 2 [32000/60000]    Loss: 116.779045
Train Epoch: 2 [38400/60000]    Loss: 110.066292
Train Epoch: 2 [44800/60000]    Loss: 113.392258
Train Epoch: 2 [51200/60000]    Loss: 117.371048
Train Epoch: 2 [57600/60000]    Loss: 117.026505
====> Epoch: 2 Average loss: 119.2312
====> Test set loss: 114.0699
Train Epoch: 3 [0/60000]        Loss: 115.382057
Train Epoch: 3 [6400/60000]     Loss: 108.025475
Train Epoch: 3 [12800/60000]    Loss: 116.623398
Train Epoch: 3 [19200/60000]    Loss: 113.990570
Train Epoch: 3 [25600/60000]    Loss: 110.621033
Train Epoch: 3 [32000/60000]    Loss: 112.123047
Train Epoch: 3 [38400/60000]    Loss: 106.969559
Train Epoch: 3 [44800/60000]    Loss: 110.406799
Train Epoch: 3 [51200/60000]    Loss: 119.068817
```

```
Train Epoch: 3 [57600/60000]    Loss: 109.565742
====> Epoch: 3 Average loss: 112.8788
====> Test set loss: 110.0252
Train Epoch: 4 [0/60000]        Loss: 107.801819
Train Epoch: 4 [6400/60000]     Loss: 106.599533
Train Epoch: 4 [12800/60000]    Loss: 113.195885
Train Epoch: 4 [19200/60000]    Loss: 109.285591
Train Epoch: 4 [25600/60000]    Loss: 112.686188
Train Epoch: 4 [32000/60000]    Loss: 103.254128
Train Epoch: 4 [38400/60000]    Loss: 107.956390
Train Epoch: 4 [44800/60000]    Loss: 114.667099
Train Epoch: 4 [51200/60000]    Loss: 104.335541
Train Epoch: 4 [57600/60000]    Loss: 102.767899
====> Epoch: 4 Average loss: 109.3005
====> Test set loss: 107.2092
Train Epoch: 5 [0/60000]        Loss: 107.941963
Train Epoch: 5 [6400/60000]     Loss: 106.589745
Train Epoch: 5 [12800/60000]    Loss: 107.095276
Train Epoch: 5 [19200/60000]    Loss: 107.577019
Train Epoch: 5 [25600/60000]    Loss: 111.295303
Train Epoch: 5 [32000/60000]    Loss: 110.598343
Train Epoch: 5 [38400/60000]    Loss: 108.069534
Train Epoch: 5 [44800/60000]    Loss: 104.231277
Train Epoch: 5 [51200/60000]    Loss: 106.946075
Train Epoch: 5 [57600/60000]    Loss: 103.621399
====> Epoch: 5 Average loss: 107.1554
====> Test set loss: 105.7634
Train Epoch: 6 [0/60000]        Loss: 106.768494
Train Epoch: 6 [6400/60000]     Loss: 109.946899
Train Epoch: 6 [12800/60000]    Loss: 105.051102
Train Epoch: 6 [19200/60000]    Loss: 104.902290
Train Epoch: 6 [25600/60000]    Loss: 99.521812
Train Epoch: 6 [32000/60000]    Loss: 110.650818
Train Epoch: 6 [38400/60000]    Loss: 108.857925
Train Epoch: 6 [44800/60000]    Loss: 102.431229
Train Epoch: 6 [51200/60000]    Loss: 100.170464
Train Epoch: 6 [57600/60000]    Loss: 104.804916
====> Epoch: 6 Average loss: 105.7350
====> Test set loss: 104.4674
Train Epoch: 7 [0/60000]        Loss: 104.887192
Train Epoch: 7 [6400/60000]     Loss: 100.655975
Train Epoch: 7 [12800/60000]    Loss: 98.495056
Train Epoch: 7 [19200/60000]    Loss: 106.359848
Train Epoch: 7 [25600/60000]    Loss: 104.719810
Train Epoch: 7 [32000/60000]    Loss: 104.474014
Train Epoch: 7 [38400/60000]    Loss: 109.240303
Train Epoch: 7 [44800/60000]    Loss: 101.162491
Train Epoch: 7 [51200/60000]    Loss: 109.285683
```

```
Train Epoch: 7 [57600/60000]    Loss: 104.148903
====> Epoch: 7 Average loss: 104.7413
====> Test set loss: 103.8603
Train Epoch: 8 [0/60000]        Loss: 109.676201
Train Epoch: 8 [6400/60000]     Loss: 104.920609
Train Epoch: 8 [12800/60000]    Loss: 98.264046
Train Epoch: 8 [19200/60000]    Loss: 105.062134
Train Epoch: 8 [25600/60000]    Loss: 100.268745
Train Epoch: 8 [32000/60000]    Loss: 97.970490
Train Epoch: 8 [38400/60000]    Loss: 103.641556
Train Epoch: 8 [44800/60000]    Loss: 106.670769
Train Epoch: 8 [51200/60000]    Loss: 107.257339
Train Epoch: 8 [57600/60000]    Loss: 102.967819
====> Epoch: 8 Average loss: 103.9211
====> Test set loss: 103.0108
Train Epoch: 9 [0/60000]        Loss: 109.330444
Train Epoch: 9 [6400/60000]     Loss: 100.687614
Train Epoch: 9 [12800/60000]    Loss: 106.326492
Train Epoch: 9 [19200/60000]    Loss: 106.605026
Train Epoch: 9 [25600/60000]    Loss: 97.284012
Train Epoch: 9 [32000/60000]    Loss: 104.188423
Train Epoch: 9 [38400/60000]    Loss: 102.945663
Train Epoch: 9 [44800/60000]    Loss: 106.897697
Train Epoch: 9 [51200/60000]    Loss: 104.220490
Train Epoch: 9 [57600/60000]    Loss: 102.940277
====> Epoch: 9 Average loss: 103.2698
====> Test set loss: 102.4828
Train Epoch: 10 [0/60000]       Loss: 104.927757
Train Epoch: 10 [6400/60000]    Loss: 96.868744
Train Epoch: 10 [12800/60000]   Loss: 100.284195
Train Epoch: 10 [19200/60000]   Loss: 101.415451
Train Epoch: 10 [25600/60000]   Loss: 101.656357
Train Epoch: 10 [32000/60000]   Loss: 101.800552
Train Epoch: 10 [38400/60000]   Loss: 100.305908
Train Epoch: 10 [44800/60000]   Loss: 100.187973
Train Epoch: 10 [51200/60000]   Loss: 101.634964
Train Epoch: 10 [57600/60000]   Loss: 98.215576
====> Epoch: 10 Average loss: 102.7517
====> Test set loss: 102.1331
```

2.0.5 Key Points on Introducing Normalizing Flows

- **Flow Quantity and Type**: Too few flows may not enhance the model; too many can cause instability. Planar Flows might be too simple—consider using RealNVP or Glow.
- **Optimization Issues**: Numerical instability from Jacobian calculations; requires careful tuning of learning rate and parameter initialization.
- **Latent Space and Complexity**: High-dimensional latent space increases model complexity,

making optimization difficult.

- **Dataset Simplicity**: On simple datasets like MNIST, Planar Flows may show limited benefits.

2.0.6 Recommendations

- **Adjust Flow Layers**: Use 2-5 flow layers and monitor results.
- **Explore Different Flows**: Try RealNVP or Radial Flows.
- **Tune Hyperparameters**: Lower the learning rate and adjust initialization.
- **Debug Incrementally**: Add flows step-by-step and visualize latent space changes.

2.0.7 Example Adjustment:

Reduce flow layers to 2 and decrease the learning rate to `1e-4`.

[9]:
```
class VAEWithFlows(nn.Module):
    def __init__(self, input_dim=784, hidden_dim=400, latent_dim=20,␣
 ↪num_flows=2):
        super(VAEWithFlows, self).__init__()
        self.latent_dim = latent_dim
        self.num_flows = num_flows


        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc21 = nn.Linear(hidden_dim, latent_dim)
        self.fc22 = nn.Linear(hidden_dim, latent_dim)


        self.fc3 = nn.Linear(latent_dim, hidden_dim)
        self.fc4 = nn.Linear(hidden_dim, input_dim)

        # Planar Flows
        self.flows = nn.ModuleList([PlanarFlow(latent_dim) for _ in␣
 ↪range(num_flows)])

    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)
        z0 = mu + eps * std

        log_det_jacobian = 0
        z = z0
```

```
        for flow in self.flows:
            z, det_jacobian = flow(z)
            log_det_jacobian += torch.log(det_jacobian + 1e-6)

        return z, log_det_jacobian

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



optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)


for epoch in range(1, 11):
    train(epoch)
    test(epoch)
```

```
Train Epoch: 1 [0/60000]        Loss: 108.970695
Train Epoch: 1 [6400/60000]     Loss: 96.966873
Train Epoch: 1 [12800/60000]    Loss: 99.417160
Train Epoch: 1 [19200/60000]    Loss: 108.783051
Train Epoch: 1 [25600/60000]    Loss: 101.173264
Train Epoch: 1 [32000/60000]    Loss: 96.929878
Train Epoch: 1 [38400/60000]    Loss: 100.307625
Train Epoch: 1 [44800/60000]    Loss: 104.952232
Train Epoch: 1 [51200/60000]    Loss: 102.790001
Train Epoch: 1 [57600/60000]    Loss: 100.911514
====> Epoch: 1 Average loss: 100.8352
====> Test set loss: 100.4292
Train Epoch: 2 [0/60000]        Loss: 99.368614
Train Epoch: 2 [6400/60000]     Loss: 101.848206
Train Epoch: 2 [12800/60000]    Loss: 102.184288
Train Epoch: 2 [19200/60000]    Loss: 103.012375
Train Epoch: 2 [25600/60000]    Loss: 99.054466
Train Epoch: 2 [32000/60000]    Loss: 96.746216
Train Epoch: 2 [38400/60000]    Loss: 100.770004
Train Epoch: 2 [44800/60000]    Loss: 97.793381
Train Epoch: 2 [51200/60000]    Loss: 98.234146
```

```
Train Epoch: 2 [57600/60000]    Loss: 96.082634
====> Epoch: 2 Average loss: 100.5479
====> Test set loss: 100.2379
Train Epoch: 3 [0/60000]        Loss: 101.912964
Train Epoch: 3 [6400/60000]     Loss: 99.520050
Train Epoch: 3 [12800/60000]    Loss: 94.996780
Train Epoch: 3 [19200/60000]    Loss: 101.873367
Train Epoch: 3 [25600/60000]    Loss: 102.823135
Train Epoch: 3 [32000/60000]    Loss: 104.092651
Train Epoch: 3 [38400/60000]    Loss: 95.973297
Train Epoch: 3 [44800/60000]    Loss: 97.674530
Train Epoch: 3 [51200/60000]    Loss: 99.274628
Train Epoch: 3 [57600/60000]    Loss: 98.477104
====> Epoch: 3 Average loss: 100.4291
====> Test set loss: 100.2028
Train Epoch: 4 [0/60000]        Loss: 100.542000
Train Epoch: 4 [6400/60000]     Loss: 100.172150
Train Epoch: 4 [12800/60000]    Loss: 105.120476
Train Epoch: 4 [19200/60000]    Loss: 99.481110
Train Epoch: 4 [25600/60000]    Loss: 98.112511
Train Epoch: 4 [32000/60000]    Loss: 99.576424
Train Epoch: 4 [38400/60000]    Loss: 92.990288
Train Epoch: 4 [44800/60000]    Loss: 97.850029
Train Epoch: 4 [51200/60000]    Loss: 98.976753
Train Epoch: 4 [57600/60000]    Loss: 106.096382
====> Epoch: 4 Average loss: 100.3881
====> Test set loss: 100.1308
Train Epoch: 5 [0/60000]        Loss: 92.071899
Train Epoch: 5 [6400/60000]     Loss: 95.853951
Train Epoch: 5 [12800/60000]    Loss: 97.642525
Train Epoch: 5 [19200/60000]    Loss: 97.680038
Train Epoch: 5 [25600/60000]    Loss: 99.213478
Train Epoch: 5 [32000/60000]    Loss: 96.312363
Train Epoch: 5 [38400/60000]    Loss: 103.351585
Train Epoch: 5 [44800/60000]    Loss: 96.226082
Train Epoch: 5 [51200/60000]    Loss: 98.462624
Train Epoch: 5 [57600/60000]    Loss: 102.726624
====> Epoch: 5 Average loss: 100.2933
====> Test set loss: 100.0404
Train Epoch: 6 [0/60000]        Loss: 103.948547
Train Epoch: 6 [6400/60000]     Loss: 100.949104
Train Epoch: 6 [12800/60000]    Loss: 99.942261
Train Epoch: 6 [19200/60000]    Loss: 99.857437
Train Epoch: 6 [25600/60000]    Loss: 99.641685
Train Epoch: 6 [32000/60000]    Loss: 104.971573
Train Epoch: 6 [38400/60000]    Loss: 96.607880
Train Epoch: 6 [44800/60000]    Loss: 101.415695
Train Epoch: 6 [51200/60000]    Loss: 99.901917
```

```
Train Epoch: 6 [57600/60000]    Loss: 109.574684
====> Epoch: 6 Average loss: 100.2139
====> Test set loss: 100.0030
Train Epoch: 7 [0/60000]        Loss: 98.976929
Train Epoch: 7 [6400/60000]     Loss: 101.972466
Train Epoch: 7 [12800/60000]    Loss: 98.293327
Train Epoch: 7 [19200/60000]    Loss: 98.376633
Train Epoch: 7 [25600/60000]    Loss: 100.403687
Train Epoch: 7 [32000/60000]    Loss: 103.103516
Train Epoch: 7 [38400/60000]    Loss: 99.818733
Train Epoch: 7 [44800/60000]    Loss: 101.364365
Train Epoch: 7 [51200/60000]    Loss: 97.915863
Train Epoch: 7 [57600/60000]    Loss: 98.247116
====> Epoch: 7 Average loss: 100.1447
====> Test set loss: 100.0205
Train Epoch: 8 [0/60000]        Loss: 102.192108
Train Epoch: 8 [6400/60000]     Loss: 100.247055
Train Epoch: 8 [12800/60000]    Loss: 100.321686
Train Epoch: 8 [19200/60000]    Loss: 101.854553
Train Epoch: 8 [25600/60000]    Loss: 101.253853
Train Epoch: 8 [32000/60000]    Loss: 97.376038
Train Epoch: 8 [38400/60000]    Loss: 102.805428
Train Epoch: 8 [44800/60000]    Loss: 103.457466
Train Epoch: 8 [51200/60000]    Loss: 96.180656
Train Epoch: 8 [57600/60000]    Loss: 94.424225
====> Epoch: 8 Average loss: 100.0897
====> Test set loss: 99.9165
Train Epoch: 9 [0/60000]        Loss: 94.239388
Train Epoch: 9 [6400/60000]     Loss: 101.788406
Train Epoch: 9 [12800/60000]    Loss: 100.479416
Train Epoch: 9 [19200/60000]    Loss: 104.938370
Train Epoch: 9 [25600/60000]    Loss: 104.765564
Train Epoch: 9 [32000/60000]    Loss: 97.631172
Train Epoch: 9 [38400/60000]    Loss: 102.504517
Train Epoch: 9 [44800/60000]    Loss: 102.987328
Train Epoch: 9 [51200/60000]    Loss: 99.074532
Train Epoch: 9 [57600/60000]    Loss: 96.581345
====> Epoch: 9 Average loss: 100.0531
====> Test set loss: 99.8900
Train Epoch: 10 [0/60000]       Loss: 100.069214
Train Epoch: 10 [6400/60000]    Loss: 101.746597
Train Epoch: 10 [12800/60000]   Loss: 103.116196
Train Epoch: 10 [19200/60000]   Loss: 109.222870
Train Epoch: 10 [25600/60000]   Loss: 98.201706
Train Epoch: 10 [32000/60000]   Loss: 100.829651
Train Epoch: 10 [38400/60000]   Loss: 102.449371
Train Epoch: 10 [44800/60000]   Loss: 103.482803
Train Epoch: 10 [51200/60000]   Loss: 96.879311
```

```
Train Epoch: 10 [57600/60000]   Loss: 99.116119
====> Epoch: 10 Average loss: 99.9592
====> Test set loss: 99.7775
```

2.0.8 RealNVP (Real-valued Non-Volume Preserving) Flow: Key Points

RealNVP is a type of Normalizing Flow that allows for complex, non-linear transformations of data while keeping the computation of the Jacobian determinant tractable. It is particularly useful in generative modeling, where the goal is to transform a simple distribution (like a Gaussian) into a complex data distribution.

Key Concepts:

1. **Affine Coupling Layers**:
   - RealNVP utilizes *affine coupling layers*, which split the input data into two parts: one part remains unchanged, and the other part is transformed conditionally based on the unchanged part. This transformation typically involves scaling and translation operations.
2. **Invertibility**:
   - The transformation is designed to be easily invertible. This means that given the output, the input can be exactly recovered, which is crucial for both density estimation and sampling.
3. **Efficient Jacobian Computation**:
   - A major advantage of RealNVP is that the Jacobian of the transformation is triangular, making its determinant easy to compute as the product of the diagonal elements. This efficiency is key to applying RealNVP in high-dimensional settings.
4. **Complexity**:
   - Despite the simplicity of each individual transformation, stacking multiple affine coupling layers allows RealNVP to model very complex, high-dimensional distributions.
5. **Application in Generative Models**:
   - RealNVP is often used in Variational Autoencoders (VAEs) and other generative models to improve the expressiveness of the latent space, leading to better quality in generated samples.

RealNVP's ability to balance expressiveness with computational efficiency makes it a powerful tool in modern generative modeling.

[ ]:

[10]:
```python
class RealNVPFlow(nn.Module):
    def __init__(self, input_dim, hidden_dim=256):
        super(RealNVPFlow, self).__init__()
        self.input_dim = input_dim

        #      s(x)   t(x)
        self.s = nn.Sequential(
            nn.Linear(input_dim // 2, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
```

```
            nn.ReLU(),
            nn.Linear(hidden_dim, input_dim // 2),
            nn.Tanh()
        )

        self.t = nn.Sequential(
            nn.Linear(input_dim // 2, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, input_dim // 2)
        )

    def forward(self, z):
        z1, z2 = z.chunk(2, dim=-1)
        s = self.s(z2)
        t = self.t(z2)
        z1 = z1 * torch.exp(s) + t
        log_det_jacobian = s.sum(dim=-1)
        return torch.cat([z1, z2], dim=-1), log_det_jacobian


class VAEWithRealNVP(nn.Module):
    def __init__(self, input_dim=784, hidden_dim=400, latent_dim=20,␣
 ↪num_flows=4):
        super(VAEWithRealNVP, self).__init__()
        self.latent_dim = latent_dim
        self.num_flows = num_flows


        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc21 = nn.Linear(hidden_dim, latent_dim)
        self.fc22 = nn.Linear(hidden_dim, latent_dim)


        self.fc3 = nn.Linear(latent_dim, hidden_dim)
        self.fc4 = nn.Linear(hidden_dim, input_dim)

        # RealNVP Flows
        self.flows = nn.ModuleList([RealNVPFlow(latent_dim) for _ in␣
 ↪range(num_flows)])

    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)
        z0 = mu + eps * std

        log_det_jacobian = 0
        z = z0

        for flow in self.flows:
            z, det_jacobian = flow(z)
            log_det_jacobian += det_jacobian

        return z, log_det_jacobian

    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, log_det_jacobian = self.reparameterize(mu, logvar)
        return self.decode(z), mu, logvar, log_det_jacobian
    def generate(self, z):
        return self.decode(z)
```

[11]:
```
model = VAEWithRealNVP().to(device)


optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)


for epoch in range(1, 21):
    train(epoch)
    test(epoch)
```

```
Train Epoch: 1 [0/60000]        Loss: 551.756714
Train Epoch: 1 [6400/60000]     Loss: 197.183884
Train Epoch: 1 [12800/60000]    Loss: 185.453918
Train Epoch: 1 [19200/60000]    Loss: 155.786942
Train Epoch: 1 [25600/60000]    Loss: 142.300949
Train Epoch: 1 [32000/60000]    Loss: 118.001137
Train Epoch: 1 [38400/60000]    Loss: 110.788528
Train Epoch: 1 [44800/60000]    Loss: 104.693710
Train Epoch: 1 [51200/60000]    Loss: 112.641640
Train Epoch: 1 [57600/60000]    Loss: 104.319725
====> Epoch: 1 Average loss: 153.4320
====> Test set loss: 101.0385
Train Epoch: 2 [0/60000]        Loss: 103.739388
Train Epoch: 2 [6400/60000]     Loss: 99.047760
```

```
Train Epoch: 2 [12800/60000]    Loss: 98.032654
Train Epoch: 2 [19200/60000]    Loss: 87.395500
Train Epoch: 2 [25600/60000]    Loss: 97.936234
Train Epoch: 2 [32000/60000]    Loss: 90.205414
Train Epoch: 2 [38400/60000]    Loss: 100.069115
Train Epoch: 2 [44800/60000]    Loss: 85.286331
Train Epoch: 2 [51200/60000]    Loss: 102.036392
Train Epoch: 2 [57600/60000]    Loss: 99.483765
====> Epoch: 2 Average loss: 95.7020
====> Test set loss: 90.0949
Train Epoch: 3 [0/60000]        Loss: 90.146347
Train Epoch: 3 [6400/60000]     Loss: 98.167053
Train Epoch: 3 [12800/60000]    Loss: 88.786018
Train Epoch: 3 [19200/60000]    Loss: 91.922897
Train Epoch: 3 [25600/60000]    Loss: 87.879845
Train Epoch: 3 [32000/60000]    Loss: 98.232658
Train Epoch: 3 [38400/60000]    Loss: 92.567497
Train Epoch: 3 [44800/60000]    Loss: 77.174553
Train Epoch: 3 [51200/60000]    Loss: 92.535332
Train Epoch: 3 [57600/60000]    Loss: 89.905716
====> Epoch: 3 Average loss: 88.1987
====> Test set loss: 85.1761
Train Epoch: 4 [0/60000]        Loss: 83.373878
Train Epoch: 4 [6400/60000]     Loss: 87.055344
Train Epoch: 4 [12800/60000]    Loss: 79.175407
Train Epoch: 4 [19200/60000]    Loss: 90.048676
Train Epoch: 4 [25600/60000]    Loss: 80.082336
Train Epoch: 4 [32000/60000]    Loss: 86.779060
Train Epoch: 4 [38400/60000]    Loss: 82.935371
Train Epoch: 4 [44800/60000]    Loss: 82.226067
Train Epoch: 4 [51200/60000]    Loss: 78.933868
Train Epoch: 4 [57600/60000]    Loss: 82.116127
====> Epoch: 4 Average loss: 84.3082
====> Test set loss: 82.1300
Train Epoch: 5 [0/60000]        Loss: 87.049187
Train Epoch: 5 [6400/60000]     Loss: 83.498672
Train Epoch: 5 [12800/60000]    Loss: 83.078346
Train Epoch: 5 [19200/60000]    Loss: 76.693062
Train Epoch: 5 [25600/60000]    Loss: 87.979401
Train Epoch: 5 [32000/60000]    Loss: 79.942703
Train Epoch: 5 [38400/60000]    Loss: 88.626625
Train Epoch: 5 [44800/60000]    Loss: 83.851570
Train Epoch: 5 [51200/60000]    Loss: 85.909439
Train Epoch: 5 [57600/60000]    Loss: 85.255554
====> Epoch: 5 Average loss: 81.7602
====> Test set loss: 79.9964
Train Epoch: 6 [0/60000]        Loss: 75.037903
Train Epoch: 6 [6400/60000]     Loss: 82.717148
```

```
Train Epoch: 6 [12800/60000]    Loss: 78.224106
Train Epoch: 6 [19200/60000]    Loss: 78.752655
Train Epoch: 6 [25600/60000]    Loss: 86.960312
Train Epoch: 6 [32000/60000]    Loss: 83.773796
Train Epoch: 6 [38400/60000]    Loss: 83.061050
Train Epoch: 6 [44800/60000]    Loss: 82.571426
Train Epoch: 6 [51200/60000]    Loss: 84.534698
Train Epoch: 6 [57600/60000]    Loss: 85.419586
====> Epoch: 6 Average loss: 79.9172
====> Test set loss: 78.7322
Train Epoch: 7 [0/60000]        Loss: 81.335571
Train Epoch: 7 [6400/60000]     Loss: 86.707253
Train Epoch: 7 [12800/60000]    Loss: 73.339455
Train Epoch: 7 [19200/60000]    Loss: 81.922371
Train Epoch: 7 [25600/60000]    Loss: 78.770020
Train Epoch: 7 [32000/60000]    Loss: 79.392929
Train Epoch: 7 [38400/60000]    Loss: 69.641815
Train Epoch: 7 [44800/60000]    Loss: 74.152260
Train Epoch: 7 [51200/60000]    Loss: 84.147385
Train Epoch: 7 [57600/60000]    Loss: 81.352310
====> Epoch: 7 Average loss: 78.5317
====> Test set loss: 77.3837
Train Epoch: 8 [0/60000]        Loss: 86.128189
Train Epoch: 8 [6400/60000]     Loss: 81.599869
Train Epoch: 8 [12800/60000]    Loss: 68.548401
Train Epoch: 8 [19200/60000]    Loss: 72.827957
Train Epoch: 8 [25600/60000]    Loss: 80.178284
Train Epoch: 8 [32000/60000]    Loss: 74.195251
Train Epoch: 8 [38400/60000]    Loss: 73.222862
Train Epoch: 8 [44800/60000]    Loss: 75.741219
Train Epoch: 8 [51200/60000]    Loss: 68.513596
Train Epoch: 8 [57600/60000]    Loss: 78.336708
====> Epoch: 8 Average loss: 77.4449
====> Test set loss: 76.6189
Train Epoch: 9 [0/60000]        Loss: 78.107712
Train Epoch: 9 [6400/60000]     Loss: 77.153732
Train Epoch: 9 [12800/60000]    Loss: 80.386993
Train Epoch: 9 [19200/60000]    Loss: 79.284439
Train Epoch: 9 [25600/60000]    Loss: 76.154205
Train Epoch: 9 [32000/60000]    Loss: 81.156052
Train Epoch: 9 [38400/60000]    Loss: 77.683403
Train Epoch: 9 [44800/60000]    Loss: 74.097809
Train Epoch: 9 [51200/60000]    Loss: 72.305847
Train Epoch: 9 [57600/60000]    Loss: 78.783981
====> Epoch: 9 Average loss: 76.4566
====> Test set loss: 75.6532
Train Epoch: 10 [0/60000]       Loss: 78.064987
Train Epoch: 10 [6400/60000]    Loss: 82.048218
```

```
Train Epoch: 10 [12800/60000]   Loss: 71.297211
Train Epoch: 10 [19200/60000]   Loss: 76.313980
Train Epoch: 10 [25600/60000]   Loss: 76.824059
Train Epoch: 10 [32000/60000]   Loss: 81.752075
Train Epoch: 10 [38400/60000]   Loss: 77.203346
Train Epoch: 10 [44800/60000]   Loss: 73.637474
Train Epoch: 10 [51200/60000]   Loss: 80.989220
Train Epoch: 10 [57600/60000]   Loss: 74.291817
====> Epoch: 10 Average loss: 75.7266
====> Test set loss: 75.1791
Train Epoch: 11 [0/60000]       Loss: 81.585602
Train Epoch: 11 [6400/60000]    Loss: 75.711540
Train Epoch: 11 [12800/60000]   Loss: 76.101776
Train Epoch: 11 [19200/60000]   Loss: 73.047882
Train Epoch: 11 [25600/60000]   Loss: 73.072433
Train Epoch: 11 [32000/60000]   Loss: 75.694496
Train Epoch: 11 [38400/60000]   Loss: 74.458862
Train Epoch: 11 [44800/60000]   Loss: 79.741867
Train Epoch: 11 [51200/60000]   Loss: 69.525116
Train Epoch: 11 [57600/60000]   Loss: 78.452942
====> Epoch: 11 Average loss: 75.0154
====> Test set loss: 74.3609
Train Epoch: 12 [0/60000]       Loss: 78.385925
Train Epoch: 12 [6400/60000]    Loss: 72.761597
Train Epoch: 12 [12800/60000]   Loss: 72.591393
Train Epoch: 12 [19200/60000]   Loss: 75.291664
Train Epoch: 12 [25600/60000]   Loss: 77.313446
Train Epoch: 12 [32000/60000]   Loss: 70.997513
Train Epoch: 12 [38400/60000]   Loss: 73.634369
Train Epoch: 12 [44800/60000]   Loss: 71.980301
Train Epoch: 12 [51200/60000]   Loss: 71.638618
Train Epoch: 12 [57600/60000]   Loss: 72.352081
====> Epoch: 12 Average loss: 74.3803
====> Test set loss: 73.9742
Train Epoch: 13 [0/60000]       Loss: 74.040451
Train Epoch: 13 [6400/60000]    Loss: 72.079208
Train Epoch: 13 [12800/60000]   Loss: 72.358231
Train Epoch: 13 [19200/60000]   Loss: 78.307556
Train Epoch: 13 [25600/60000]   Loss: 72.312889
Train Epoch: 13 [32000/60000]   Loss: 72.537346
Train Epoch: 13 [38400/60000]   Loss: 69.642830
Train Epoch: 13 [44800/60000]   Loss: 74.637100
Train Epoch: 13 [51200/60000]   Loss: 76.438431
Train Epoch: 13 [57600/60000]   Loss: 75.166145
====> Epoch: 13 Average loss: 73.8628
====> Test set loss: 73.5870
Train Epoch: 14 [0/60000]       Loss: 71.037666
Train Epoch: 14 [6400/60000]    Loss: 69.710197
```

```
Train Epoch: 14 [12800/60000]   Loss: 77.722633
Train Epoch: 14 [19200/60000]   Loss: 71.351677
Train Epoch: 14 [25600/60000]   Loss: 77.945671
Train Epoch: 14 [32000/60000]   Loss: 80.104820
Train Epoch: 14 [38400/60000]   Loss: 69.318024
Train Epoch: 14 [44800/60000]   Loss: 72.534485
Train Epoch: 14 [51200/60000]   Loss: 69.857033
Train Epoch: 14 [57600/60000]   Loss: 74.110733
====> Epoch: 14 Average loss: 73.3888
====> Test set loss: 73.0152
Train Epoch: 15 [0/60000]       Loss: 66.049034
Train Epoch: 15 [6400/60000]    Loss: 73.404701
Train Epoch: 15 [12800/60000]   Loss: 75.803108
Train Epoch: 15 [19200/60000]   Loss: 72.853638
Train Epoch: 15 [25600/60000]   Loss: 77.642159
Train Epoch: 15 [32000/60000]   Loss: 71.843613
Train Epoch: 15 [38400/60000]   Loss: 70.052444
Train Epoch: 15 [44800/60000]   Loss: 74.069122
Train Epoch: 15 [51200/60000]   Loss: 72.857605
Train Epoch: 15 [57600/60000]   Loss: 67.223396
====> Epoch: 15 Average loss: 72.9582
====> Test set loss: 72.6762
Train Epoch: 16 [0/60000]       Loss: 66.102554
Train Epoch: 16 [6400/60000]    Loss: 77.322372
Train Epoch: 16 [12800/60000]   Loss: 77.480843
Train Epoch: 16 [19200/60000]   Loss: 68.280106
Train Epoch: 16 [25600/60000]   Loss: 75.477455
Train Epoch: 16 [32000/60000]   Loss: 72.714272
Train Epoch: 16 [38400/60000]   Loss: 75.143394
Train Epoch: 16 [44800/60000]   Loss: 71.397766
Train Epoch: 16 [51200/60000]   Loss: 71.671265
Train Epoch: 16 [57600/60000]   Loss: 71.551392
====> Epoch: 16 Average loss: 72.5366
====> Test set loss: 72.2397
Train Epoch: 17 [0/60000]       Loss: 76.392326
Train Epoch: 17 [6400/60000]    Loss: 77.339188
Train Epoch: 17 [12800/60000]   Loss: 73.555244
Train Epoch: 17 [19200/60000]   Loss: 81.549316
Train Epoch: 17 [25600/60000]   Loss: 71.233780
Train Epoch: 17 [32000/60000]   Loss: 71.538918
Train Epoch: 17 [38400/60000]   Loss: 71.460487
Train Epoch: 17 [44800/60000]   Loss: 68.403320
Train Epoch: 17 [51200/60000]   Loss: 71.029716
Train Epoch: 17 [57600/60000]   Loss: 65.840134
====> Epoch: 17 Average loss: 72.1952
====> Test set loss: 71.9793
Train Epoch: 18 [0/60000]       Loss: 75.502213
Train Epoch: 18 [6400/60000]    Loss: 73.839310
```

```
Train Epoch: 18 [12800/60000]   Loss: 74.992119
Train Epoch: 18 [19200/60000]   Loss: 73.201019
Train Epoch: 18 [25600/60000]   Loss: 68.710732
Train Epoch: 18 [32000/60000]   Loss: 68.146294
Train Epoch: 18 [38400/60000]   Loss: 73.385796
Train Epoch: 18 [44800/60000]   Loss: 68.954475
Train Epoch: 18 [51200/60000]   Loss: 66.669319
Train Epoch: 18 [57600/60000]   Loss: 71.626976
====> Epoch: 18 Average loss: 71.8235
====> Test set loss: 71.5028
Train Epoch: 19 [0/60000]       Loss: 68.570160
Train Epoch: 19 [6400/60000]    Loss: 64.234276
Train Epoch: 19 [12800/60000]   Loss: 74.848740
Train Epoch: 19 [19200/60000]   Loss: 68.486122
Train Epoch: 19 [25600/60000]   Loss: 77.457397
Train Epoch: 19 [32000/60000]   Loss: 72.375473
Train Epoch: 19 [38400/60000]   Loss: 66.836563
Train Epoch: 19 [44800/60000]   Loss: 74.704353
Train Epoch: 19 [51200/60000]   Loss: 72.107086
Train Epoch: 19 [57600/60000]   Loss: 64.773590
====> Epoch: 19 Average loss: 71.4978
====> Test set loss: 71.5658
Train Epoch: 20 [0/60000]       Loss: 66.934860
Train Epoch: 20 [6400/60000]    Loss: 70.811432
Train Epoch: 20 [12800/60000]   Loss: 70.686981
Train Epoch: 20 [19200/60000]   Loss: 68.395294
Train Epoch: 20 [25600/60000]   Loss: 65.428879
Train Epoch: 20 [32000/60000]   Loss: 70.251091
Train Epoch: 20 [38400/60000]   Loss: 76.756287
Train Epoch: 20 [44800/60000]   Loss: 76.113640
Train Epoch: 20 [51200/60000]   Loss: 67.048058
Train Epoch: 20 [57600/60000]   Loss: 72.799965
====> Epoch: 20 Average loss: 71.1110
====> Test set loss: 71.0677
```

[12]:
```python
def vae_loss_function(recon_x, x, mu, logvar, log_det_jacobian):
    BCE = F.binary_cross_entropy(recon_x, x.view(-1, 784), reduction='sum')
    KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp()) -␣
 ↪log_det_jacobian.sum()
    return BCE + KLD

def train_vae_with_realnvpflow(model, data_loader, num_epochs=10):
    model.train()
    optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)

    for epoch in range(num_epochs):
        train_loss = 0
```

```
        for i, (data, _) in enumerate(data_loader):
            data = data.to(device)
            optimizer.zero_grad()
            recon_batch, mu, logvar, log_det_jacobian = model(data)
            loss = vae_loss_function(recon_batch, data, mu, logvar,␣
 ↪log_det_jacobian)
            loss.backward()
            train_loss += loss.item()
            optimizer.step()

        print(f"====> Epoch: {epoch} Average loss: {train_loss /␣
 ↪len(data_loader.dataset):.4f}")
```

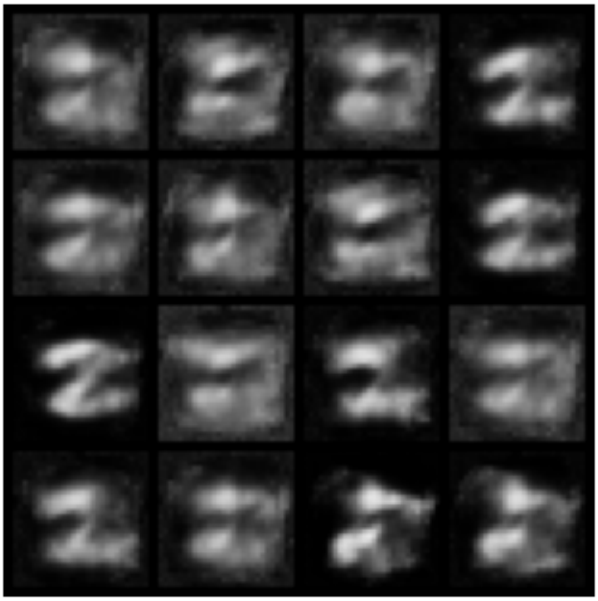
[13]:
```
import matplotlib.pyplot as plt
from torchvision.utils import save_image, make_grid
import numpy as np

from sklearn.decomposition import PCA
from sklearn.manifold import TSNE

def visualize_samples_and_latent_space(model, data_loader, num_samples=16,␣
 ↪use_pca=True):
    model.eval()
    with torch.no_grad():

        z = torch.randn(num_samples, model.latent_dim).to(device)
        samples = model.generate(z).cpu()


        plt.figure(figsize=(4, 4))
        grid_img = make_grid(samples.view(num_samples, 1, 28, 28), nrow=4,␣
 ↪padding=2)
        plt.imshow(grid_img.permute(1, 2, 0).numpy(), cmap='gray')
        plt.axis('off')
        plt.show()


        latents = []
        labels = []
        for data, label in data_loader:
            data = data.to(device)
            mu, logvar = model.encode(data.view(-1, 784))
            z, _ = model.reparameterize(mu, logvar)
            latents.append(z.cpu().numpy())
            labels.append(label.numpy())

        latents = np.concatenate(latents)
```

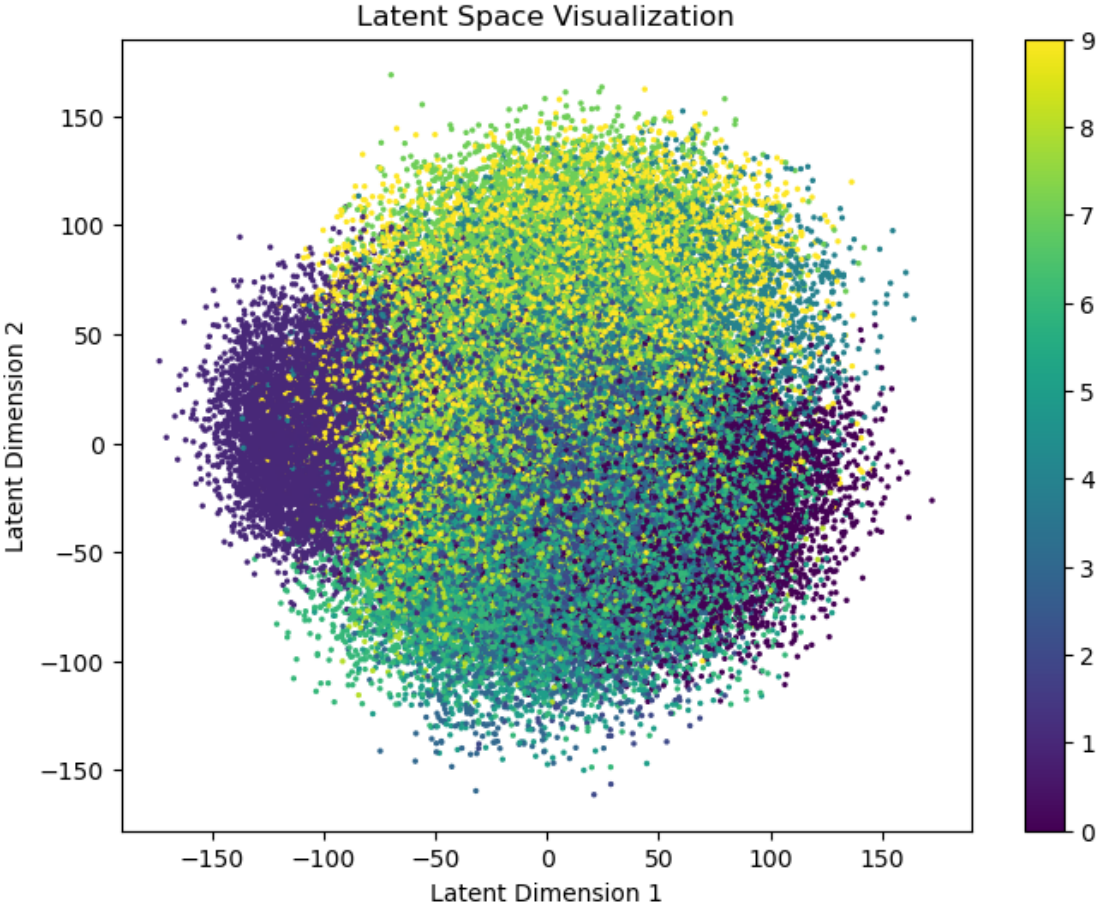
```
        labels = np.concatenate(labels)


        if model.latent_dim > 2:
            if use_pca:
                latents_2d = PCA(n_components=2).fit_transform(latents)
            else:
                latents_2d = TSNE(n_components=2).fit_transform(latents)
        else:
            latents_2d = latents

        plt.figure(figsize=(8, 6))
        plt.scatter(latents_2d[:, 0], latents_2d[:, 1], c=labels,␣
↪cmap='viridis', s=2)
        plt.colorbar()
        plt.xlabel('Latent Dimension 1')
        plt.ylabel('Latent Dimension 2')
        plt.title('Latent Space Visualization')
        plt.show()
```

[14]:
```
vae_with_realnvp = VAEWithRealNVP().to(device)
train_vae_with_realnvpflow(vae_with_realnvp, data_loader, num_epochs=10)
```

```
====> Epoch: 0 Average loss: 104.6459
====> Epoch: 1 Average loss: 78.2237
====> Epoch: 2 Average loss: 74.5231
====> Epoch: 3 Average loss: 72.8711
====> Epoch: 4 Average loss: 71.7555
====> Epoch: 5 Average loss: 70.7489
====> Epoch: 6 Average loss: 70.0991
====> Epoch: 7 Average loss: 69.4540
====> Epoch: 8 Average loss: 68.9480
====> Epoch: 9 Average loss: 68.5396
```

[15]:
```
visualize_samples_and_latent_space(vae_with_realnvp, data_loader)
```




Latent Space Visualization
Latent Dimension 2
Latent Dimension 1
150
100
50
0
−50
−100
−150
−150
−100
−50
0
50
100
150
9
8
7
6
5
4
3
2
1
0

2.0.9 Analysis Based on the plot:

1. Sample Quality

- **Observation**: Generated images are blurry with unclear details, making digits hard to recognize.
- **Issues**:
  - **Undertraining**: Insufficient training may lead to poor sample quality.
  - **VAE Limitations**: VAE may struggle with generating high-quality samples, especially with complex data.
  - **RealNVP Adjustment**: Inadequate tuning of RealNVP may fail to capture the true data distribution.
- **Suggestions**:
  - Extend training duration.
  - Fine-tune hyperparameters (e.g., learning rate, latent dimensions).
  - Experiment with other flow models like Planar Flow or MAF.

2. Latent Space Structure

- **Observation**: Data points are evenly distributed but lack clear clustering; labels are mixed without distinct separation.
- **Issues**:
  - **Misaligned Distribution**: RealNVP may not properly align latent space, causing poor category separation.
  - **Overextended Space**: Excessive spread in latent space may degrade sample quality.
- **Suggestions**:
  - Adjust dimensionality reduction techniques (e.g., t-SNE, PCA).
  - Increase regularization for a more compact latent space.
  - Modify encoder and flow structure for better alignment with data distribution.

2.0.10 Summary

- **Sample Quality**: Currently poor; needs more training or model adjustments.
- **Latent Space**: Lacks clear separation; requires better structure alignment.

To improve results, consider fine-tuning hyperparameters, extending training, or switching flow models for enhanced sample quality and latent space structure.

2.0.11 Why RealNVP is Effective

- **Expressive Power**: Captures complex distributions through invertible affine transformations, fitting high-dimensional data better.
- **Stable Training**: Simplifies Jacobian calculation, ensuring stable and efficient training.
- **Latent Space**: Enhances latent space flexibility, leading to more realistic samples.
- **Scalability**: Easily expands with more layers without increasing complexity.

2.0.12 Summary

RealNVP significantly boosts model performance, improving sample quality and stability, especially in complex, high-dimensional tasks.

3 Combined Flow: Key Points

Combined Flow enhances a VAE by combining **RealNVP** and **Planar Flows** to increase the flexibility and expressiveness of the latent space.

Key Concepts:

1. **Hybrid Approach**:
   - Integrates **RealNVP** for global transformations and **Planar Flows** for local adjustments in the latent space, providing a balance of structure and flexibility.
2. **Sequential Processing**:
   - Applies flows in a sequence, first transforming with RealNVP for robust shaping, then fine-tuning with Planar Flows.
3. **Improved Expressiveness**:
   - This combination allows the model to better capture complex data distributions, enhancing the VAE's ability to generate diverse and accurate samples.

The Combined Flow approach makes the VAE more adaptable to complex data patterns, leading to superior generative performance.

[16]:
```
class CombinedFlow(nn.Module):
    def __init__(self, latent_dim, hidden_dim=256, num_realnvp_flows=2,␣
 ↪num_planar_flows=2):
        super(CombinedFlow, self).__init__()
        self.realnvp_flows = nn.ModuleList([RealNVPFlow(latent_dim, hidden_dim)␣
 ↪for _ in range(num_realnvp_flows)])
        self.planar_flows = nn.ModuleList([PlanarFlow(latent_dim) for _ in␣
 ↪range(num_planar_flows)])

    def forward(self, z):
        log_det_jacobian = 0


        for flow in self.realnvp_flows:
            z, det_jacobian = flow(z)
            log_det_jacobian += det_jacobian


        for flow in self.planar_flows:
            z, det_jacobian = flow(z)
            log_det_jacobian += torch.log(det_jacobian + 1e-6)

        return z, log_det_jacobian
```

```
class VAEWithCombinedFlow(nn.Module):
    def __init__(self, input_dim=784, hidden_dim=400, latent_dim=50,␣
 ↪num_realnvp_flows=2, num_planar_flows=2):
        super(VAEWithCombinedFlow, self).__init__()
        self.latent_dim = latent_dim


        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc21 = nn.Linear(hidden_dim, latent_dim)
        self.fc22 = nn.Linear(hidden_dim, latent_dim)


        self.fc3 = nn.Linear(latent_dim, hidden_dim)
        self.fc4 = nn.Linear(hidden_dim, input_dim)


        self.flow = CombinedFlow(latent_dim, hidden_dim, num_realnvp_flows,␣
 ↪num_planar_flows)

    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)
        z0 = mu + eps * std


        z, log_det_jacobian = self.flow(z0)

        return z, log_det_jacobian

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

    def generate(self, z):
        return self.decode(z)
```

[17]:
```
model = VAEWithCombinedFlow().to(device)


optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)


for epoch in range(1, 21):
    train(epoch)
    test(epoch)
```

```
Train Epoch: 1 [0/60000]        Loss: 554.683228
Train Epoch: 1 [6400/60000]     Loss: 184.731567
Train Epoch: 1 [12800/60000]    Loss: 171.809280
Train Epoch: 1 [19200/60000]    Loss: 150.241943
Train Epoch: 1 [25600/60000]    Loss: 137.676071
Train Epoch: 1 [32000/60000]    Loss: 131.328018
Train Epoch: 1 [38400/60000]    Loss: 110.952324
Train Epoch: 1 [44800/60000]    Loss: 123.687164
Train Epoch: 1 [51200/60000]    Loss: 121.474014
Train Epoch: 1 [57600/60000]    Loss: 113.123947
====> Epoch: 1 Average loss: 155.9750
====> Test set loss: 114.4605
Train Epoch: 2 [0/60000]        Loss: 112.696976
Train Epoch: 2 [6400/60000]     Loss: 110.119186
Train Epoch: 2 [12800/60000]    Loss: 110.233643
Train Epoch: 2 [19200/60000]    Loss: 107.310501
Train Epoch: 2 [25600/60000]    Loss: 107.445679
Train Epoch: 2 [32000/60000]    Loss: 109.807159
Train Epoch: 2 [38400/60000]    Loss: 104.083588
Train Epoch: 2 [44800/60000]    Loss: 106.503647
Train Epoch: 2 [51200/60000]    Loss: 102.831657
Train Epoch: 2 [57600/60000]    Loss: 104.082207
====> Epoch: 2 Average loss: 106.9139
====> Test set loss: 100.1970
Train Epoch: 3 [0/60000]        Loss: 106.538780
Train Epoch: 3 [6400/60000]     Loss: 105.090790
Train Epoch: 3 [12800/60000]    Loss: 95.988937
Train Epoch: 3 [19200/60000]    Loss: 96.323090
Train Epoch: 3 [25600/60000]    Loss: 96.161621
Train Epoch: 3 [32000/60000]    Loss: 103.012619
Train Epoch: 3 [38400/60000]    Loss: 88.866241
Train Epoch: 3 [44800/60000]    Loss: 99.646896
Train Epoch: 3 [51200/60000]    Loss: 89.380394
Train Epoch: 3 [57600/60000]    Loss: 94.285294
====> Epoch: 3 Average loss: 97.4473
```

```
====> Test set loss: 92.9738
Train Epoch: 4 [0/60000]        Loss: 97.023659
Train Epoch: 4 [6400/60000]     Loss: 98.314636
Train Epoch: 4 [12800/60000]    Loss: 95.619331
Train Epoch: 4 [19200/60000]    Loss: 96.575546
Train Epoch: 4 [25600/60000]    Loss: 88.507828
Train Epoch: 4 [32000/60000]    Loss: 83.951447
Train Epoch: 4 [38400/60000]    Loss: 88.534752
Train Epoch: 4 [44800/60000]    Loss: 96.937416
Train Epoch: 4 [51200/60000]    Loss: 98.214996
Train Epoch: 4 [57600/60000]    Loss: 96.955933
====> Epoch: 4 Average loss: 92.1485
====> Test set loss: 89.8049
Train Epoch: 5 [0/60000]        Loss: 99.415001
Train Epoch: 5 [6400/60000]     Loss: 98.247856
Train Epoch: 5 [12800/60000]    Loss: 82.017876
Train Epoch: 5 [19200/60000]    Loss: 81.730202
Train Epoch: 5 [25600/60000]    Loss: 85.305161
Train Epoch: 5 [32000/60000]    Loss: 87.244423
Train Epoch: 5 [38400/60000]    Loss: 99.490639
Train Epoch: 5 [44800/60000]    Loss: 97.741051
Train Epoch: 5 [51200/60000]    Loss: 91.218254
Train Epoch: 5 [57600/60000]    Loss: 93.872620
====> Epoch: 5 Average loss: 90.2391
====> Test set loss: 87.4474
Train Epoch: 6 [0/60000]        Loss: 87.471054
Train Epoch: 6 [6400/60000]     Loss: 86.367889
Train Epoch: 6 [12800/60000]    Loss: 90.747528
Train Epoch: 6 [19200/60000]    Loss: 81.581001
Train Epoch: 6 [25600/60000]    Loss: 91.844986
Train Epoch: 6 [32000/60000]    Loss: 91.289772
Train Epoch: 6 [38400/60000]    Loss: 84.526520
Train Epoch: 6 [44800/60000]    Loss: 75.065208
Train Epoch: 6 [51200/60000]    Loss: 81.812294
Train Epoch: 6 [57600/60000]    Loss: 82.383621
====> Epoch: 6 Average loss: 85.0950
====> Test set loss: 83.2832
Train Epoch: 7 [0/60000]        Loss: 78.759384
Train Epoch: 7 [6400/60000]     Loss: 88.426918
Train Epoch: 7 [12800/60000]    Loss: 87.214722
Train Epoch: 7 [19200/60000]    Loss: 94.188202
Train Epoch: 7 [25600/60000]    Loss: 91.327354
Train Epoch: 7 [32000/60000]    Loss: 93.709488
Train Epoch: 7 [38400/60000]    Loss: 86.414078
Train Epoch: 7 [44800/60000]    Loss: 90.132980
Train Epoch: 7 [51200/60000]    Loss: 82.892159
Train Epoch: 7 [57600/60000]    Loss: 78.056946
====> Epoch: 7 Average loss: 86.5070
```

```
====> Test set loss: 83.3568
Train Epoch: 8 [0/60000]        Loss: 81.268845
Train Epoch: 8 [6400/60000]     Loss: 86.355415
Train Epoch: 8 [12800/60000]    Loss: 88.578873
Train Epoch: 8 [19200/60000]    Loss: 76.330254
Train Epoch: 8 [25600/60000]    Loss: 77.311584
Train Epoch: 8 [32000/60000]    Loss: 83.957970
Train Epoch: 8 [38400/60000]    Loss: 85.357002
Train Epoch: 8 [44800/60000]    Loss: 84.136040
Train Epoch: 8 [51200/60000]    Loss: 80.685333
Train Epoch: 8 [57600/60000]    Loss: 81.577484
====> Epoch: 8 Average loss: 81.8842
====> Test set loss: 79.3167
Train Epoch: 9 [0/60000]        Loss: 76.688774
Train Epoch: 9 [6400/60000]     Loss: 82.693024
Train Epoch: 9 [12800/60000]    Loss: 77.646164
Train Epoch: 9 [19200/60000]    Loss: 87.219864
Train Epoch: 9 [25600/60000]    Loss: 72.522102
Train Epoch: 9 [32000/60000]    Loss: 73.026901
Train Epoch: 9 [38400/60000]    Loss: 85.154419
Train Epoch: 9 [44800/60000]    Loss: 79.866486
Train Epoch: 9 [51200/60000]    Loss: 75.896805
Train Epoch: 9 [57600/60000]    Loss: 83.892303
====> Epoch: 9 Average loss: 80.0564
====> Test set loss: 78.1605
Train Epoch: 10 [0/60000]       Loss: 80.098328
Train Epoch: 10 [6400/60000]    Loss: 75.751251
Train Epoch: 10 [12800/60000]   Loss: 72.295181
Train Epoch: 10 [19200/60000]   Loss: 77.422722
Train Epoch: 10 [25600/60000]   Loss: 74.601151
Train Epoch: 10 [32000/60000]   Loss: 69.187622
Train Epoch: 10 [38400/60000]   Loss: 79.912384
Train Epoch: 10 [44800/60000]   Loss: 80.334618
Train Epoch: 10 [51200/60000]   Loss: 76.856216
Train Epoch: 10 [57600/60000]   Loss: 77.151398
====> Epoch: 10 Average loss: 78.5615
====> Test set loss: 77.6566
Train Epoch: 11 [0/60000]       Loss: 81.982285
Train Epoch: 11 [6400/60000]    Loss: 73.234245
Train Epoch: 11 [12800/60000]   Loss: 82.668327
Train Epoch: 11 [19200/60000]   Loss: 75.824127
Train Epoch: 11 [25600/60000]   Loss: 69.550858
Train Epoch: 11 [32000/60000]   Loss: 81.478050
Train Epoch: 11 [38400/60000]   Loss: 84.464844
Train Epoch: 11 [44800/60000]   Loss: 75.130302
Train Epoch: 11 [51200/60000]   Loss: 78.777954
Train Epoch: 11 [57600/60000]   Loss: 74.127014
====> Epoch: 11 Average loss: 77.6663
```

```
====> Test set loss: 76.0430
Train Epoch: 12 [0/60000]       Loss: 76.567261
Train Epoch: 12 [6400/60000]    Loss: 74.724785
Train Epoch: 12 [12800/60000]   Loss: 80.968399
Train Epoch: 12 [19200/60000]   Loss: 73.646362
Train Epoch: 12 [25600/60000]   Loss: 72.110519
Train Epoch: 12 [32000/60000]   Loss: 84.050995
Train Epoch: 12 [38400/60000]   Loss: 80.911552
Train Epoch: 12 [44800/60000]   Loss: 75.999695
Train Epoch: 12 [51200/60000]   Loss: 78.087769
Train Epoch: 12 [57600/60000]   Loss: 79.404480
====> Epoch: 12 Average loss: 76.3535
====> Test set loss: 74.5274
Train Epoch: 13 [0/60000]       Loss: 72.040390
Train Epoch: 13 [6400/60000]    Loss: 68.971397
Train Epoch: 13 [12800/60000]   Loss: 77.156784
Train Epoch: 13 [19200/60000]   Loss: 67.430946
Train Epoch: 13 [25600/60000]   Loss: 74.697800
Train Epoch: 13 [32000/60000]   Loss: 78.527344
Train Epoch: 13 [38400/60000]   Loss: 76.572273
Train Epoch: 13 [44800/60000]   Loss: 74.659531
Train Epoch: 13 [51200/60000]   Loss: 78.167976
Train Epoch: 13 [57600/60000]   Loss: 68.736206
====> Epoch: 13 Average loss: 76.1422
====> Test set loss: 74.2728
Train Epoch: 14 [0/60000]       Loss: 75.671280
Train Epoch: 14 [6400/60000]    Loss: 75.000229
Train Epoch: 14 [12800/60000]   Loss: 71.657639
Train Epoch: 14 [19200/60000]   Loss: 81.340576
Train Epoch: 14 [25600/60000]   Loss: 77.844017
Train Epoch: 14 [32000/60000]   Loss: 72.911522
Train Epoch: 14 [38400/60000]   Loss: 77.727493
Train Epoch: 14 [44800/60000]   Loss: 72.610031
Train Epoch: 14 [51200/60000]   Loss: 73.870567
Train Epoch: 14 [57600/60000]   Loss: 82.159706
====> Epoch: 14 Average loss: 75.3801
====> Test set loss: 72.7241
Train Epoch: 15 [0/60000]       Loss: 70.901443
Train Epoch: 15 [6400/60000]    Loss: 78.227707
Train Epoch: 15 [12800/60000]   Loss: 74.152496
Train Epoch: 15 [19200/60000]   Loss: 75.872437
Train Epoch: 15 [25600/60000]   Loss: 76.167267
Train Epoch: 15 [32000/60000]   Loss: 75.325653
Train Epoch: 15 [38400/60000]   Loss: 69.597435
Train Epoch: 15 [44800/60000]   Loss: 73.933090
Train Epoch: 15 [51200/60000]   Loss: 70.040131
Train Epoch: 15 [57600/60000]   Loss: 76.997375
====> Epoch: 15 Average loss: 73.9142
```

```
====> Test set loss: 71.5064
Train Epoch: 16 [0/60000]       Loss: 73.070847
Train Epoch: 16 [6400/60000]    Loss: 70.700195
Train Epoch: 16 [12800/60000]   Loss: 76.554169
Train Epoch: 16 [19200/60000]   Loss: 75.926964
Train Epoch: 16 [25600/60000]   Loss: 70.198898
Train Epoch: 16 [32000/60000]   Loss: 72.402946
Train Epoch: 16 [38400/60000]   Loss: 72.849472
Train Epoch: 16 [44800/60000]   Loss: 74.073830
Train Epoch: 16 [51200/60000]   Loss: 87.790138
Train Epoch: 16 [57600/60000]   Loss: 86.250793
====> Epoch: 16 Average loss: 74.8463
====> Test set loss: 79.2954
Train Epoch: 17 [0/60000]       Loss: 80.237434
Train Epoch: 17 [6400/60000]    Loss: 77.384048
Train Epoch: 17 [12800/60000]   Loss: 73.700539
Train Epoch: 17 [19200/60000]   Loss: 71.976204
Train Epoch: 17 [25600/60000]   Loss: 71.395805
Train Epoch: 17 [32000/60000]   Loss: 72.496597
Train Epoch: 17 [38400/60000]   Loss: 72.667999
Train Epoch: 17 [44800/60000]   Loss: 63.992615
Train Epoch: 17 [51200/60000]   Loss: 75.043274
Train Epoch: 17 [57600/60000]   Loss: 68.377281
====> Epoch: 17 Average loss: 73.6142
====> Test set loss: 70.5415
Train Epoch: 18 [0/60000]       Loss: 72.964272
Train Epoch: 18 [6400/60000]    Loss: 68.088898
Train Epoch: 18 [12800/60000]   Loss: 62.519028
Train Epoch: 18 [19200/60000]   Loss: 69.428642
Train Epoch: 18 [25600/60000]   Loss: 69.540756
Train Epoch: 18 [32000/60000]   Loss: 66.446060
Train Epoch: 18 [38400/60000]   Loss: 63.148476
Train Epoch: 18 [44800/60000]   Loss: 78.982994
Train Epoch: 18 [51200/60000]   Loss: 78.143791
Train Epoch: 18 [57600/60000]   Loss: 74.094345
====> Epoch: 18 Average loss: 72.3236
====> Test set loss: 70.1435
Train Epoch: 19 [0/60000]       Loss: 74.255028
Train Epoch: 19 [6400/60000]    Loss: 77.392998
Train Epoch: 19 [12800/60000]   Loss: 67.859283
Train Epoch: 19 [19200/60000]   Loss: 66.390167
Train Epoch: 19 [25600/60000]   Loss: 69.555641
Train Epoch: 19 [32000/60000]   Loss: 65.826126
Train Epoch: 19 [38400/60000]   Loss: 72.179581
Train Epoch: 19 [44800/60000]   Loss: 73.133675
Train Epoch: 19 [51200/60000]   Loss: 71.185532
Train Epoch: 19 [57600/60000]   Loss: 70.920891
====> Epoch: 19 Average loss: 70.2878
```

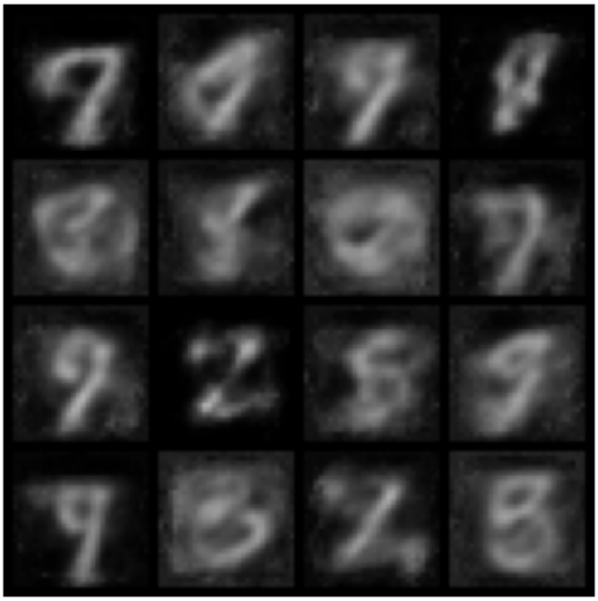
```
====> Test set loss: 68.2532
Train Epoch: 20 [0/60000]       Loss: 67.096878
Train Epoch: 20 [6400/60000]    Loss: 73.898651
Train Epoch: 20 [12800/60000]   Loss: 60.570786
Train Epoch: 20 [19200/60000]   Loss: 71.570930
Train Epoch: 20 [25600/60000]   Loss: 73.037628
Train Epoch: 20 [32000/60000]   Loss: 65.029144
Train Epoch: 20 [38400/60000]   Loss: 66.659821
Train Epoch: 20 [44800/60000]   Loss: 62.536995
Train Epoch: 20 [51200/60000]   Loss: 69.492996
Train Epoch: 20 [57600/60000]   Loss: 65.896751
====> Epoch: 20 Average loss: 69.8619
====> Test set loss: 68.4026
```

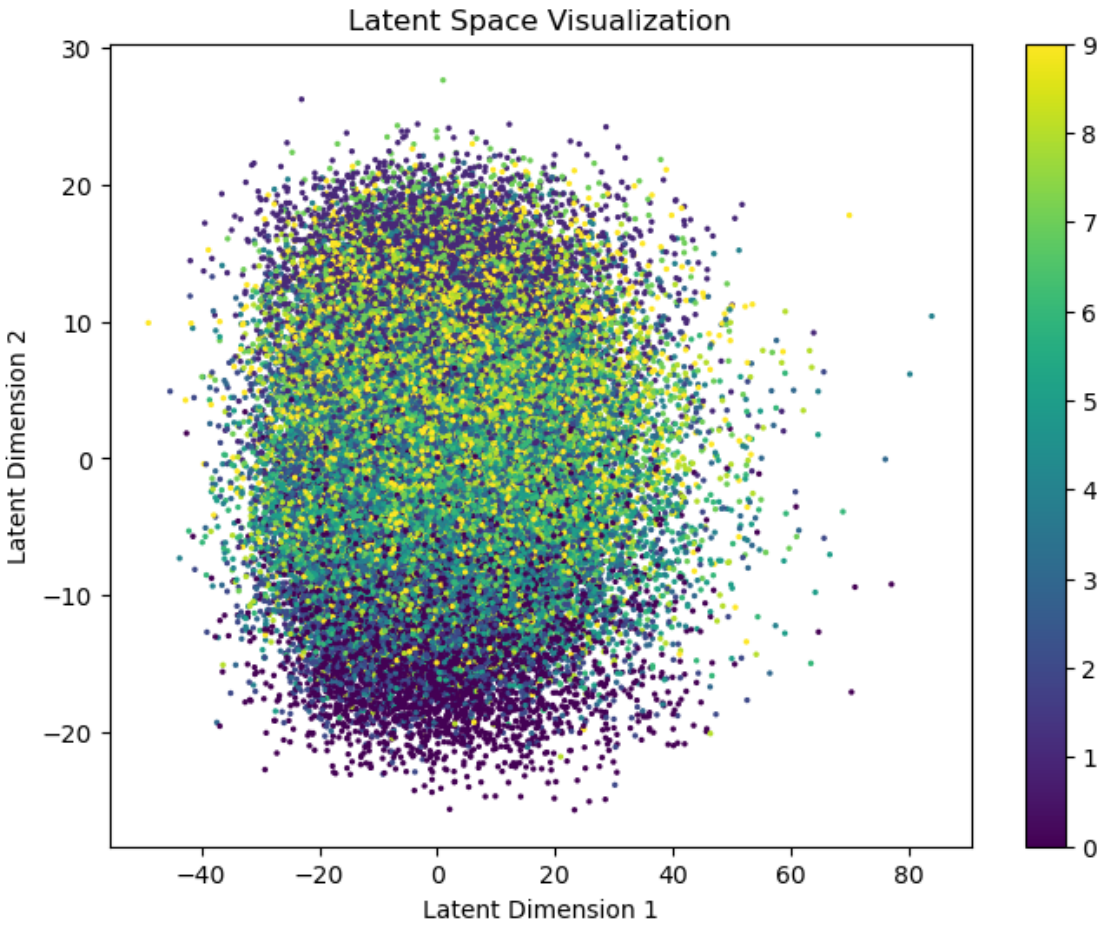
[18]:
```
def train_vae_with_combined_flow(model, data_loader, num_epochs=10):
    model.train()
    optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)

    for epoch in range(num_epochs):
        train_loss = 0
        for i, (data, _) in enumerate(data_loader):
            data = data.to(device)
            optimizer.zero_grad()
            recon_batch, mu, logvar, log_det_jacobian = model(data)
            loss = vae_loss_function(recon_batch, data, mu, logvar,␣
 ↪log_det_jacobian)
            loss.backward()
            train_loss += loss.item()
            optimizer.step()

        print(f"====> Epoch: {epoch} Average loss: {train_loss /␣
 ↪len(data_loader.dataset):.4f}")
```

[19]:
```
input_dim = 784
hidden_dim = 400
latent_dim = 50
num_realnvp_flows = 2
num_planar_flows = 2
num_epochs = 10


vae_with_combined_flow = VAEWithCombinedFlow(input_dim=input_dim,␣
 ↪hidden_dim=hidden_dim, latent_dim=latent_dim,␣
 ↪num_realnvp_flows=num_realnvp_flows, num_planar_flows=num_planar_flows).
 ↪to(device)
train_vae_with_combined_flow(vae_with_combined_flow, data_loader,␣
 ↪num_epochs=num_epochs)
```

```
====> Epoch: 0 Average loss: 98.7750
====> Epoch: 1 Average loss: 66.8329
====> Epoch: 2 Average loss: 62.8472
====> Epoch: 3 Average loss: 59.6244
====> Epoch: 4 Average loss: 58.3282
====> Epoch: 5 Average loss: 61.2480
====> Epoch: 6 Average loss: 59.2590
====> Epoch: 7 Average loss: 56.5406
====> Epoch: 8 Average loss: 55.9999
====> Epoch: 9 Average loss: 56.4312
```

[20]:
```
visualize_samples_and_latent_space(vae_with_combined_flow, data_loader)
```





3.0.1 Key Points

1. Sample Quality

- **Blurry Images**: Generated images are unclear with poor detail.
- **Undertraining**: Model likely needs more training.
- **Complexity Issues**: CombinedFlow may be too complex, affecting stability.

2. Latent Space Structure

- **Poor Category Separation**: Latent space lacks distinct clusters.
- **Balancing Issues**: KL divergence and flow contributions may not be well-balanced.

3.0.2 Suggestions

- **Increase Training**: Extend epochs for better learning.
- **Adjust Flow Model**: Simplify or reconfigure flows for improved stability and separation.
- **Enhance Regularization**: Strengthen KL divergence to improve latent space clarity.

3.0.3 Introducing Adversarial Training with VAE-GAN

VAE-GAN is a powerful approach that combines the strengths of a Variational Autoencoder (VAE) with a Generative Adversarial Network (GAN). This hybrid model leverages the generative capabilities of the VAE and the adversarial training of the GAN to enhance the quality, realism, and diversity of generated samples. By integrating these two models, VAE-GAN aims to overcome the limitations of each individual model and produce more accurate and high-fidelity outputs.

VAE-GAN Architecture

1. **Variational Autoencoder (VAE)**:
   - **Encoder**: Maps input data to a latent space by learning the mean and variance of a Gaussian distribution. This probabilistic mapping enables the model to capture complex data distributions and generate diverse samples.
   - **Latent Space**: The VAE samples from the latent space to generate new data points. This space represents the underlying features learned from the input data.
   - **Decoder**: Reconstructs the input data from the sampled latent variables, generating data that resembles the original input.
2. **Generative Adversarial Network (GAN)**:
   - **Generator (VAE)**: In the VAE-GAN setup, the VAE's decoder serves as the generator. It generates new samples from the latent space.
   - **Discriminator**: The GAN's discriminator evaluates the authenticity of the samples by distinguishing between real data and the data generated by the VAE. It provides feedback to the generator, pushing it to produce more realistic samples.
   - **Adversarial Training**: The VAE and discriminator are trained in a competitive manner. The discriminator tries to correctly identify real and fake samples, while the VAE adjusts its parameters to generate samples that can fool the discriminator.

Implementation Strategy

1. **Use VAE as the Generator**:
   - The VAE generates samples by sampling from the learned latent space. The decoder reconstructs these samples, aiming to produce data that closely resembles the training data.
2. **GAN Discriminator for Distribution Evaluation**:
   - The GAN's discriminator is trained to evaluate the differences between real and generated samples. It assesses how closely the generated samples match the true data distribution, providing a measure of sample quality.
3. **Adversarial Training for Realism**:
   - Adversarial training is employed to make the VAE-generated samples more realistic. The discriminator's feedback is used to refine the VAE, improving the overall quality of the generated data.

Benefits of VAE-GAN

- **Improved Sample Quality**: By combining VAE's generative capabilities with GAN's adversarial training, VAE-GAN produces higher quality samples with sharper details and fewer artifacts.

- **Diverse Sample Generation**: The VAE's latent space sampling ensures diversity in the generated data, while the GAN's discriminator ensures that this diversity does not come at the cost of realism.
- **Overcoming Limitations**: VAE-GAN mitigates common issues in VAEs, such as blurry outputs, by incorporating the GAN's ability to generate sharp and realistic images. It also addresses the mode collapse issue in GANs by utilizing VAE's structured latent space.

Implementing the Discriminator First, we implement a discriminator to distinguish between real samples and generated samples. We can use Spectral Normalization to enhance the stability of the discriminator.

[21]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F

class SpectralNorm(nn.Module):
    def __init__(self, module, name='weight', power_iterations=1):
        super(SpectralNorm, self).__init__()
        self.module = module
        self.name = name
        self.power_iterations = power_iterations
        if not self._made_params():
            self._make_params()

    def _update_u_v(self):
        u = getattr(self.module, self.name + "_u")
        v = getattr(self.module, self.name + "_v")
        w = getattr(self.module, self.name + "_bar")

        for _ in range(self.power_iterations):
            v = F.normalize(torch.matmul(w.view(w.size(0), -1).t(), u), dim=0,␣
 ↪eps=1e-12)
            u = F.normalize(torch.matmul(w.view(w.size(0), -1), v), dim=0,␣
 ↪eps=1e-12)

        sigma = torch.dot(u, torch.matmul(w.view(w.size(0), -1), v))
        setattr(self.module, self.name, nn.Parameter(w / sigma.expand_as(w)))

    def _made_params(self):
        try:
            u = getattr(self.module, self.name + "_u")
            v = getattr(self.module, self.name + "_v")
            w = getattr(self.module, self.name + "_bar")
            return True
        except AttributeError:
            return False

    def _make_params(self):
```

```
        w = getattr(self.module, self.name)

        height = w.data.shape[0]
        width = w.view(height, -1).data.shape[1]

        u = F.normalize(w.new_empty(height).normal_(0, 1), dim=0, eps=1e-12)
        v = F.normalize(w.new_empty(width).normal_(0, 1), dim=0, eps=1e-12)
        w_bar = w.data


        self.module.register_buffer(self.name + "_u", u)
        self.module.register_buffer(self.name + "_v", v)
        self.module.register_parameter(self.name + "_bar", nn.Parameter(w_bar))

        self._update_u_v()

    def forward(self, *args):
        self._update_u_v()
        return self.module.forward(*args)


class DiscriminatorWithSpectralNorm(nn.Module):
    def __init__(self, input_dim=784, hidden_dim=256):
        super(DiscriminatorWithSpectralNorm, self).__init__()
        self.fc1 = SpectralNorm(nn.Linear(input_dim, hidden_dim))
        self.fc2 = SpectralNorm(nn.Linear(hidden_dim, hidden_dim))
        self.fc3 = SpectralNorm(nn.Linear(hidden_dim, 1))

    def forward(self, x):
        x = x.view(-1, 784)  #       (batch_size, 784)
        x = F.leaky_relu(self.fc1(x), 0.2)
        x = F.leaky_relu(self.fc2(x), 0.2)
        return torch.sigmoid(self.fc3(x))
```

Modified generator (VAEWithCombinedFlow) VAEWithCombinedFlow is used as a generator in which the flow has been augmented. We use it as a generator part for adversarial training of GAN.

Implementing a VAE-GAN Hybrid Model To combine VAE with GAN, we need a combinatorial model that handles both the reconstruction of VAE and the adversarial loss of GAN.

[22]:
```
class VAEGAN(nn.Module):
    def __init__(self, vae, discriminator):
        super(VAEGAN, self).__init__()
        self.vae = vae
        self.discriminator = discriminator

    def forward(self, x):
        recon_x, mu, logvar, log_det_jacobian = self.vae(x)
```

```
        return recon_x, mu, logvar, log_det_jacobian

    def generate(self, z):
        return self.vae.decode(z)
```

Define the loss function We need a combined loss function that takes into account the reconstruction error of the VAE, the KL dispersion, and the adversarial loss of the GAN.

[23]:
```
def vae_gan_loss(recon_x, x, mu, logvar, log_det_jacobian, real_output,␣
 ↪fake_output):
    BCE = F.binary_cross_entropy(recon_x, x.view(-1, 784), reduction='sum')
    KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
    KLD = KLD - log_det_jacobian.sum()

    #
    adv_loss = F.binary_cross_entropy(fake_output, torch.ones_like(fake_output))

    return BCE + KLD + adv_loss

def discriminator_loss(real_output, fake_output):
    real_loss = F.binary_cross_entropy(real_output, torch.
 ↪ones_like(real_output))
    fake_loss = F.binary_cross_entropy(fake_output, torch.
 ↪zeros_like(fake_output))
    return real_loss + fake_loss
```

Training the VAE-GAN model Next, a training loop is defined in which both the VAE and the discriminator are optimized.

[24]:
```
latent_dim = 50
vae = VAEWithCombinedFlow(input_dim=784, hidden_dim=400, latent_dim=latent_dim,␣
 ↪num_realnvp_flows=2, num_planar_flows=2).to(device)
discriminator = DiscriminatorWithSpectralNorm(input_dim=784, hidden_dim=256).
 ↪to(device)
vae_gan = VAEGAN(vae, discriminator).to(device)

optimizer_g = torch.optim.Adam(vae_gan.vae.parameters(), lr=1e-4, betas=(0.5, 0.
 ↪999))
optimizer_d = torch.optim.Adam(vae_gan.discriminator.parameters(), lr=1e-4,␣
 ↪betas=(0.5, 0.999))

batch_size = 64

num_epochs = 20
```

[25]:
```
data_loader = DataLoader(dataset=train_dataset, batch_size=batch_size,␣
 ↪shuffle=True)
```

```
for epoch in range(num_epochs):
    for i, (real_images, _) in enumerate(data_loader):
        real_images = real_images.view(-1, 784).to(device)
        batch_size = real_images.size(0)


        recon_images, mu, logvar, log_det_jacobian = vae_gan(real_images)


        z = torch.randn(batch_size, latent_dim).to(device)
        fake_images = vae_gan.generate(z)


        real_output = vae_gan.discriminator(real_images)
        fake_output = vae_gan.discriminator(fake_images)


        d_loss = discriminator_loss(real_output, fake_output)


        optimizer_d.zero_grad()
        d_loss.backward()
        optimizer_d.step()


        fake_output = vae_gan.discriminator(recon_images)
        g_loss = vae_gan_loss(recon_images, real_images, mu, logvar,␣
 ↪log_det_jacobian, real_output, fake_output)


        optimizer_g.zero_grad()
        g_loss.backward()
        optimizer_g.step()

    print(f"Epoch [{epoch}/{num_epochs}], d_loss: {d_loss.item()}, g_loss:␣
 ↪{g_loss.item()}")
```

```
Epoch [0/20], d_loss: 1.1445033550262451, g_loss: 3973.83056640625
Epoch [1/20], d_loss: 1.0391623973846436, g_loss: 3057.28759765625
Epoch [2/20], d_loss: 1.008840799331665, g_loss: 2957.677978515625
Epoch [3/20], d_loss: 0.9954007863998413, g_loss: 2772.764892578125
Epoch [4/20], d_loss: 0.9905545115470886, g_loss: 2512.4111328125
Epoch [5/20], d_loss: 1.001739263534546, g_loss: 2468.214111328125
Epoch [6/20], d_loss: 1.0247597694396973, g_loss: 2452.9541015625
Epoch [7/20], d_loss: 1.0499069690704346, g_loss: 2650.5126953125
Epoch [8/20], d_loss: 1.0331026315689087, g_loss: 2039.690185546875
```

```
Epoch [9/20], d_loss: 1.053390622138977, g_loss: 2079.548828125
Epoch [10/20], d_loss: 1.0526208877563477, g_loss: 2269.3388671875
Epoch [11/20], d_loss: 1.0442800521850586, g_loss: 2239.904052734375
Epoch [12/20], d_loss: 1.0597752332687378, g_loss: 2249.018310546875
Epoch [13/20], d_loss: 1.0615637302398682, g_loss: 1974.9710693359375
Epoch [14/20], d_loss: 1.0663193464279175, g_loss: 1846.298828125
Epoch [15/20], d_loss: 1.0616655349731445, g_loss: 1943.9842529296875
Epoch [16/20], d_loss: 1.1132010221481323, g_loss: 1994.5111083984375
Epoch [17/20], d_loss: 1.0743820667266846, g_loss: 1673.5137939453125
Epoch [18/20], d_loss: 1.0929055213928223, g_loss: 1944.426513671875
Epoch [19/20], d_loss: 1.0973994731903076, g_loss: 2003.361083984375
```

[26]:
```python
def vae_loss_function(recon_x, x, mu, logvar, log_det_jacobian):
    BCE = F.binary_cross_entropy(recon_x, x.view(-1, 784), reduction='sum')
    KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
    KLD = KLD - log_det_jacobian.sum()
    return BCE + KLD
```

[27]:
```python
def train_vae(vae, data_loader, num_epochs):
    vae.train()
    optimizer = torch.optim.Adam(vae.parameters(), lr=1e-3)

    for epoch in range(num_epochs):
        train_loss = 0
        for i, (data, _) in enumerate(data_loader):
            data = data.to(device)
            optimizer.zero_grad()
            recon_batch, mu, logvar, log_det_jacobian = vae(data)
            loss = vae_loss_function(recon_batch, data, mu, logvar,␣
 ↪log_det_jacobian)
            loss.backward()
            train_loss += loss.item()
            optimizer.step()

        print(f"====> Epoch: {epoch} Average loss: {train_loss /␣
 ↪len(data_loader.dataset):.4f}")
        test_vae(vae)

def test_vae(vae):
    vae.eval()
    test_loss = 0
    with torch.no_grad():
        for data, _ in data_loader:
            data = data.to(device)
            recon_batch, mu, logvar, log_det_jacobian = vae(data)
            test_loss += vae_loss_function(recon_batch, data, mu, logvar,␣
 ↪log_det_jacobian).item()
```

```
    test_loss /= len(data_loader.dataset)
    print(f"====> Test set loss: {test_loss:.4f}")
    return test_loss
```

[28]:
```
def train_vae_gan(vae_gan, data_loader, num_epochs):
    vae_gan.train()
    optimizer_g = torch.optim.Adam(vae_gan.vae.parameters(), lr=1e-4, betas=(0.
 ↪5, 0.999))
    optimizer_d = torch.optim.Adam(vae_gan.discriminator.parameters(), lr=1e-4,␣
 ↪betas=(0.5, 0.999))

    for epoch in range(num_epochs):
        train_loss_g = 0
        train_loss_d = 0
        for i, (real_images, _) in enumerate(data_loader):
            real_images = real_images.to(device)
            batch_size = real_images.size(0)


            recon_images, mu, logvar, log_det_jacobian = vae_gan(real_images)


            z = torch.randn(batch_size, latent_dim).to(device)
            fake_images = vae_gan.generate(z)


            real_output = vae_gan.discriminator(real_images.view(-1, 784))
            fake_output = vae_gan.discriminator(fake_images.view(-1, 784))


            d_loss = discriminator_loss(real_output, fake_output)
            train_loss_d += d_loss.item()


            optimizer_d.zero_grad()
            d_loss.backward()
            optimizer_d.step()


            fake_output = vae_gan.discriminator(recon_images.view(-1, 784))
            g_loss = vae_gan_loss_function(recon_images, real_images, mu,␣
 ↪logvar, log_det_jacobian, real_output, fake_output)
            train_loss_g += g_loss.item()


            optimizer_g.zero_grad()
            g_loss.backward()
```

```
            optimizer_g.step()

        print(f"Epoch [{epoch}/{num_epochs}], d_loss: {train_loss_d /␣
 ↪len(data_loader):.4f}, g_loss: {train_loss_g / len(data_loader):.4f}")
```

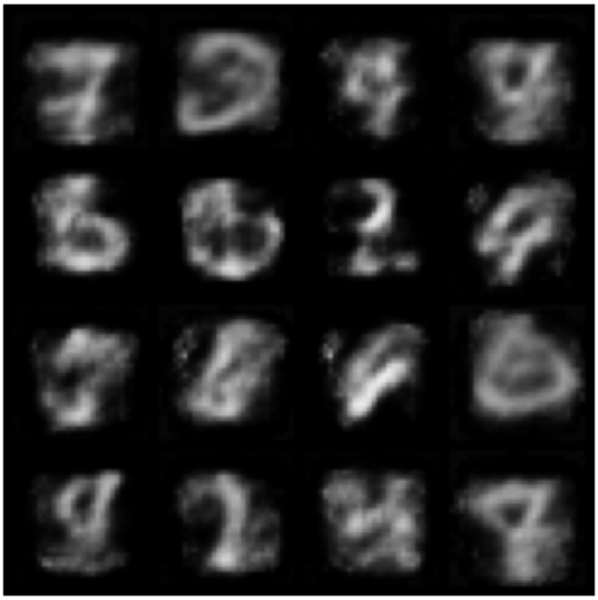
[29]:
```
import matplotlib.pyplot as plt
def generate_samples(model, num_samples=16):
    model.eval()
    with torch.no_grad():
        z = torch.randn(num_samples, latent_dim).to(device)
        samples = model.generate(z).cpu()
        save_image(samples.view(num_samples, 1, 28, 28), 'generated_samples.
 ↪png', nrow=4)


    plt.figure(figsize=(4, 4))
    plt.axis('off')
    plt.imshow(samples.view(-1, 28).numpy(), cmap='gray')
    plt.show()
```

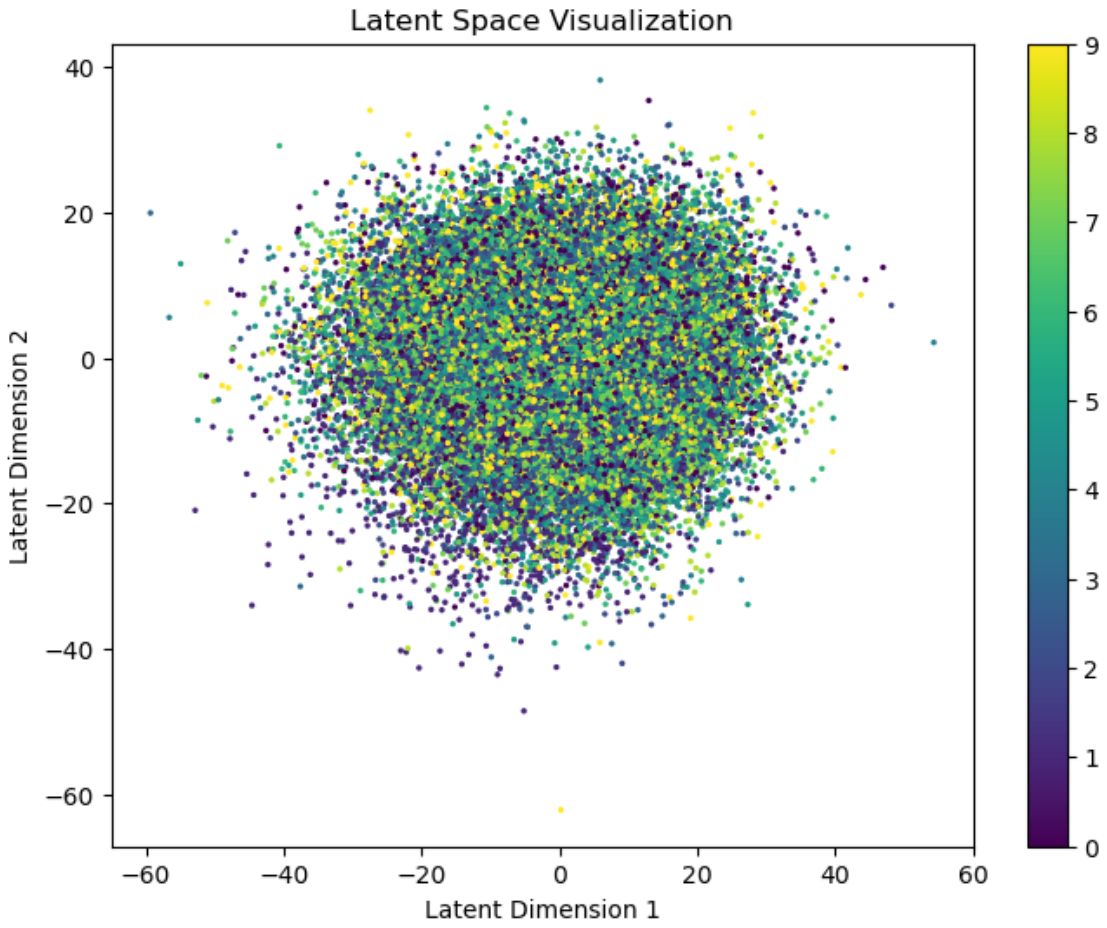
[30]:
```
# Initializing and Training a VAE Model

vae = VAEWithCombinedFlow().to(device)
train_vae(vae, data_loader, num_epochs)
```

```
====> Epoch: 0 Average loss: 91.7329
====> Test set loss: 68.6578
====> Epoch: 1 Average loss: 64.8647
====> Test set loss: 61.8764
====> Epoch: 2 Average loss: 61.2163
====> Test set loss: 59.7085
====> Epoch: 3 Average loss: 58.9891
====> Test set loss: 58.2526
====> Epoch: 4 Average loss: 58.4252
====> Test set loss: 57.6530
====> Epoch: 5 Average loss: 58.1854
====> Test set loss: 56.7840
====> Epoch: 6 Average loss: 56.7914
====> Test set loss: 56.4034
====> Epoch: 7 Average loss: 56.2514
====> Test set loss: 55.9732
====> Epoch: 8 Average loss: 55.8531
====> Test set loss: 55.5329
====> Epoch: 9 Average loss: 55.5975
====> Test set loss: 55.4575
====> Epoch: 10 Average loss: 55.2392
====> Test set loss: 55.0394
====> Epoch: 11 Average loss: 54.9960
```

```
====> Test set loss: 54.6947
====> Epoch: 12 Average loss: 54.9550
====> Test set loss: 54.7900
====> Epoch: 13 Average loss: 54.6072
====> Test set loss: 54.1867
====> Epoch: 14 Average loss: 54.4700
====> Test set loss: 54.1064
====> Epoch: 15 Average loss: 54.3396
====> Test set loss: 54.1945
====> Epoch: 16 Average loss: 54.1721
====> Test set loss: 54.0679
====> Epoch: 17 Average loss: 55.0234
====> Test set loss: 54.8609
====> Epoch: 18 Average loss: 54.2472
====> Test set loss: 54.1380
====> Epoch: 19 Average loss: 53.9772
====> Test set loss: 53.7722
```

[31]:
```
visualize_samples_and_latent_space(vae, train_loader)
```





[32]:
```
def vae_gan_loss_function(recon_x, x, mu, logvar, log_det_jacobian,␣
 ↪real_output, fake_output):
    VAE_loss = vae_loss_function(recon_x, x, mu, logvar, log_det_jacobian)
    adv_loss = F.binary_cross_entropy(fake_output, torch.ones_like(fake_output))
    return VAE_loss + adv_loss

def discriminator_loss(real_output, fake_output):
    real_loss = F.binary_cross_entropy(real_output, torch.
 ↪ones_like(real_output))
    fake_loss = F.binary_cross_entropy(fake_output, torch.
 ↪zeros_like(fake_output))
    return real_loss + fake_loss
```

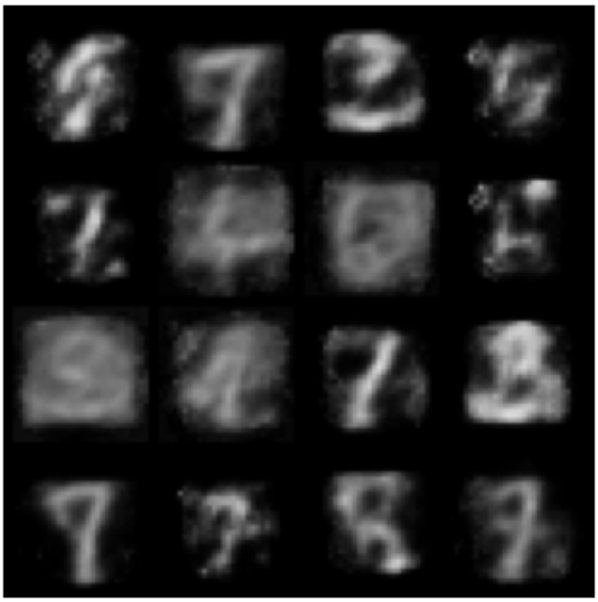
[33]:
```
discriminator = DiscriminatorWithSpectralNorm().to(device)
vae_gan = VAEGAN(vae, discriminator).to(device)

train_vae_gan(vae_gan, data_loader, num_epochs)
```

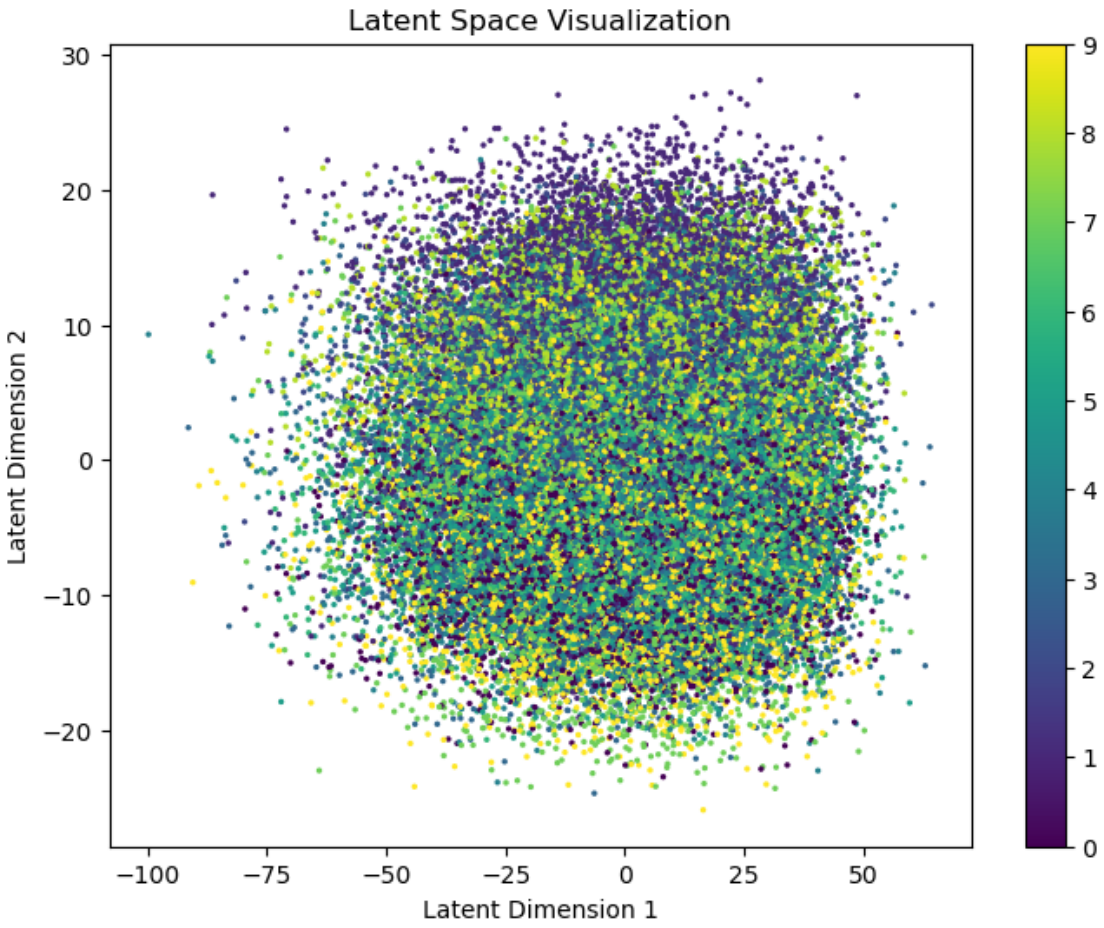
```
Epoch [0/20], d_loss: 1.3313, g_loss: 3297.3393
Epoch [1/20], d_loss: 1.2736, g_loss: 3266.9220
Epoch [2/20], d_loss: 1.2468, g_loss: 3257.5763
Epoch [3/20], d_loss: 1.2326, g_loss: 3250.6384
Epoch [4/20], d_loss: 1.2242, g_loss: 3245.9573
Epoch [5/20], d_loss: 1.2194, g_loss: 3239.6213
Epoch [6/20], d_loss: 1.2165, g_loss: 3237.9134
Epoch [7/20], d_loss: 1.2138, g_loss: 3234.7144
Epoch [8/20], d_loss: 1.2123, g_loss: 3233.8741
Epoch [9/20], d_loss: 1.2124, g_loss: 3233.8619
Epoch [10/20], d_loss: 1.2108, g_loss: 3230.4439
Epoch [11/20], d_loss: 1.2100, g_loss: 3226.6941
Epoch [12/20], d_loss: 1.2093, g_loss: 3226.1228
Epoch [13/20], d_loss: 1.2083, g_loss: 3223.7569
Epoch [14/20], d_loss: 1.2077, g_loss: 3220.5416
Epoch [15/20], d_loss: 1.2073, g_loss: 3219.8003
Epoch [16/20], d_loss: 1.2072, g_loss: 3218.8582
Epoch [17/20], d_loss: 1.2065, g_loss: 3216.2781
Epoch [18/20], d_loss: 1.2062, g_loss: 3215.3682
Epoch [19/20], d_loss: 1.2064, g_loss: 3214.3161
```

[34]:
```python
def visualize_samples_and_latent_space(model, data_loader, num_samples=16,␣
↪use_pca=True):
    model.eval()
    with torch.no_grad():
        #
        latent_dim = model.vae.latent_dim  #   VAEGAN   VAE   latent_dim
        z = torch.randn(num_samples, latent_dim).to(device)
        samples = model.vae.generate(z).cpu()  #   VAE   generate

        #
        plt.figure(figsize=(4, 4))
        grid_img = make_grid(samples.view(num_samples, 1, 28, 28), nrow=4,␣
↪padding=2)
        plt.imshow(grid_img.permute(1, 2, 0).numpy(), cmap='gray')
        plt.axis('off')
        plt.show()

        #
        latents = []
        labels = []
        for data, label in data_loader:
            data = data.to(device)
            mu, logvar = model.vae.encode(data.view(-1, 784))  #   VAE   encode␣
↪
            z, _ = model.vae.reparameterize(mu, logvar)  #   VAE ␣
↪reparameterize
```

```
            latents.append(z.cpu().numpy())
            labels.append(label.numpy())

        latents = np.concatenate(latents)
        labels = np.concatenate(labels)

        #       > 2
        if latent_dim > 2:
            if use_pca:
                latents_2d = PCA(n_components=2).fit_transform(latents)
            else:
                latents_2d = TSNE(n_components=2).fit_transform(latents)
        else:
            latents_2d = latents  #   latent_dim    2

        plt.figure(figsize=(8, 6))
        plt.scatter(latents_2d[:, 0], latents_2d[:, 1], c=labels,␣
↪cmap='viridis', s=2)
        plt.colorbar()
        plt.xlabel('Latent Dimension 1')
        plt.ylabel('Latent Dimension 2')
        plt.title('Latent Space Visualization')
        plt.show()
```

[35]:
```
visualize_samples_and_latent_space(vae_gan, train_loader)
```





3.0.4 Key Findings

1. Sample Quality

- **Blurry and Distorted**: Generated samples are unclear and often distorted, with poor boundary definition.
- **Potential Issues**: Insufficient training and imbalance between adversarial and reconstruction losses.
- **Suggestions**:
  - Extend training time.
  - Rebalance loss weights.
  - Enhance discriminator complexity.

2. Latent Space Visualization

- **Weak Category Separation**: Data points are evenly distributed but lack clear category separation.

- **Potential Issues**: Inadequate latent space representation and disruption from adversarial training.
- **Suggestions**:
  - Increase KL divergence weight.
  - Add flow model complexity.
  - Apply stronger regularization.

3. Next Steps

- **Improve Sample Quality**: Continue training, adjust loss weights, and enhance the discriminator.
- **Refine Latent Space**: Increase model complexity to achieve better category separation and consistency.

4 Report: Overview of MultivariateGKDistribution

MultivariateGKDistribution is a specialized parameterized distribution model designed to extend the univariate g-and-k distribution for multivariate scenarios. It is particularly effective in capturing the complexities of real-world data, such as asymmetry and heavy-tailed characteristics.

Key Parameters:

- **Location (a)**: Centers the distribution.
- **Scale (b)**: Determines the distribution's spread.
- **Skewness (g)**: Controls the asymmetry of the distribution.
- **Kurtosis (k)**: Governs the thickness of the distribution's tails, impacting its heavy-tailed nature.

Mathematical Foundation: The distribution is defined by applying a nonlinear transformation to a standard normal variable `u`, producing a variable `z` whose distribution is shaped by the parameters `a`, `b`, `g`, and `k`. This flexibility allows the model to represent a wide range of distributional forms, from symmetric normal distributions to skewed and heavy-tailed distributions.

[36]:
```
import torch
import torch.nn as nn

class MultivariateGKDistribution(nn.Module):
    def __init__(self, dim, a=0, b=1, g=0, k=0):
        super(MultivariateGKDistribution, self).__init__()
        self.dim = dim
        self.a = nn.Parameter(torch.tensor(a, dtype=torch.float32).repeat(dim))
        self.b = nn.Parameter(torch.tensor(b, dtype=torch.float32).repeat(dim))
        self.g = nn.Parameter(torch.tensor(g, dtype=torch.float32).repeat(dim))
        self.k = nn.Parameter(torch.tensor(k, dtype=torch.float32).repeat(dim))

    def forward(self, u):
```

```
        z = self.a + self.b * (1 + 0.8 * (1 - torch.exp(-self.g * u)) / (1 +␣
↪torch.exp(-self.g * u))) * (1 + u**2)**self.k * u
        return z

    def sample(self, num_samples):
        u = torch.randn(num_samples, self.dim, dtype=torch.float32)
        return self.forward(u)
```

[37]:
```
from scipy.optimize import least_squares

def levenberg_marquardt_loss(params, model, target_distribution):
    dim = model.dim
    a, b, g, k = params[:dim], params[dim:2*dim], params[2*dim:3*dim],␣
↪params[3*dim:]
    model.a.data = torch.tensor(a, dtype=torch.float32)
    model.b.data = torch.tensor(b, dtype=torch.float32)
    model.g.data = torch.tensor(g, dtype=torch.float32)
    model.k.data = torch.tensor(k, dtype=torch.float32)

    samples = model.sample(len(target_distribution))
    loss = torch.mean((samples - target_distribution) ** 2).item()
    return loss

def optimize_gk_parameters(model, target_distribution):
    initial_params = torch.cat([model.a.data, model.b.data, model.g.data, model.
↪k.data]).numpy()
    result = least_squares(levenberg_marquardt_loss, initial_params,␣
↪args=(model, target_distribution), method='lm')
    optimized_params = result.x

    model.a.data = torch.tensor(optimized_params[:model.dim], dtype=torch.
↪float32)
    model.b.data = torch.tensor(optimized_params[model.dim:2*model.dim],␣
↪dtype=torch.float32)
    model.g.data = torch.tensor(optimized_params[2*model.dim:3*model.dim],␣
↪dtype=torch.float32)
    model.k.data = torch.tensor(optimized_params[3*model.dim:], dtype=torch.
↪float32)
```

[38]:
```
class FlowLayer(nn.Module):
    def __init__(self, latent_dim):
        super(FlowLayer, self).__init__()
        self.u = nn.Parameter(torch.randn(latent_dim, dtype=torch.float32))
        self.w = nn.Parameter(torch.randn(latent_dim, 1, dtype=torch.float32)) ␣
↪# Adjusted to have a singleton dimension for broadcasting
        self.b = nn.Parameter(torch.zeros(1, dtype=torch.float32))
```

```
    def forward(self, z):
        linear = torch.matmul(z, self.w) + self.b
        activation = torch.tanh(linear)
        z_new = z + self.u.unsqueeze(0) * activation  # Broadcast u

        psi = (1 - activation**2) * self.w.squeeze(1)  # Squeeze w to match the␣
↪psi calculation
        det_jacobian = torch.abs(1 + torch.matmul(psi, self.u))

        log_det_jacobian = torch.log(det_jacobian + 1e-6).float()

        return z_new, log_det_jacobian

class NormalizingFlow(nn.Module):
    def __init__(self, latent_dim, num_flows=2, base_dist=None):
        super(NormalizingFlow, self).__init__()
        self.flows = nn.ModuleList([FlowLayer(latent_dim) for _ in␣
↪range(num_flows)])
        self.base_dist = base_dist if base_dist is not None else␣
↪MultivariateGKDistribution(latent_dim)

    def forward(self, z):
        log_det_jacobian = torch.zeros(z.size(0), dtype=torch.float32, device=z.
↪device)
        for flow in self.flows:
            z, log_det = flow(z)
            log_det_jacobian += log_det
        return z, log_det_jacobian

    def sample(self, num_samples):
        z0 = self.base_dist.sample(num_samples)
        z, _ = self.forward(z0)
        return z
```

[39]:
```
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

class VAEWithFlow(nn.Module):
    def __init__(self, input_dim=784, hidden_dim=400, latent_dim=50,␣
↪num_flows=2):
        super(VAEWithFlow, self).__init__()
        self.latent_dim = latent_dim
        self.encoder = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
```

```
            nn.Linear(hidden_dim, latent_dim * 2)  # For mean and log variance
        )
        self.decoder = nn.Sequential(
            nn.Linear(latent_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, input_dim),
            nn.Sigmoid()  # Apply sigmoid to ensure output is in range [0, 1]
        )
        self.flow = NormalizingFlow(latent_dim, num_flows=num_flows)

    def encode(self, x):
        x = x.view(-1, 784)  # Flatten the input image to match the input_dim␣
 ↪of 784
        params = self.encoder(x)
        mu, logvar = params[:, :self.latent_dim], params[:, self.latent_dim:]
        return mu, logvar

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

    def decode(self, z):
        return self.decoder(z)  # Keep it as [batch_size, 784], values in [0, 1]

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

    def generate(self, z):
        return self.decode(z)  # Use the decoder to generate images from z


    # Define the MNIST dataset with the correct transformation
transform = transforms.Compose([
    transforms.ToTensor(),  # Converts the image to a tensor with pixel values␣
 ↪between 0 and 1
])

train_dataset = datasets.MNIST(root='./data', train=True, transform=transform,␣
 ↪download=True)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
# Initialize the VAE model with Normalizing Flow
```

```
vae_with_flow = VAEWithFlow(input_dim=784, hidden_dim=400, latent_dim=50,␣
 ↪num_flows=4).to(device)

# Define the optimizer
optimizer = optim.Adam(vae_with_flow.parameters(), lr=1e-3)

def loss_function(recon_x, x, mu, logvar, log_det_jacobian):
    # Flatten the input to match the output of the decoder
    x = x.view(-1, 784)  # Flatten target to [batch_size, 784]

    BCE = nn.functional.binary_cross_entropy(recon_x, x, reduction='sum')
    KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
    return BCE + KLD - torch.sum(log_det_jacobian)

for epoch in range(1, 11):
    train_loss = 0
    vae_with_flow.train()
    for batch_idx, (data, _) in enumerate(train_loader):
        data = data.to(device)
        optimizer.zero_grad()
        recon_batch, mu, logvar, log_det_jacobian = vae_with_flow(data)


        loss = loss_function(recon_batch, data, mu, logvar, log_det_jacobian)
        loss.backward()
        train_loss += loss.item()
        optimizer.step()

    print(f'Epoch {epoch}, Loss: {train_loss / len(train_loader.dataset):.6f}')
```

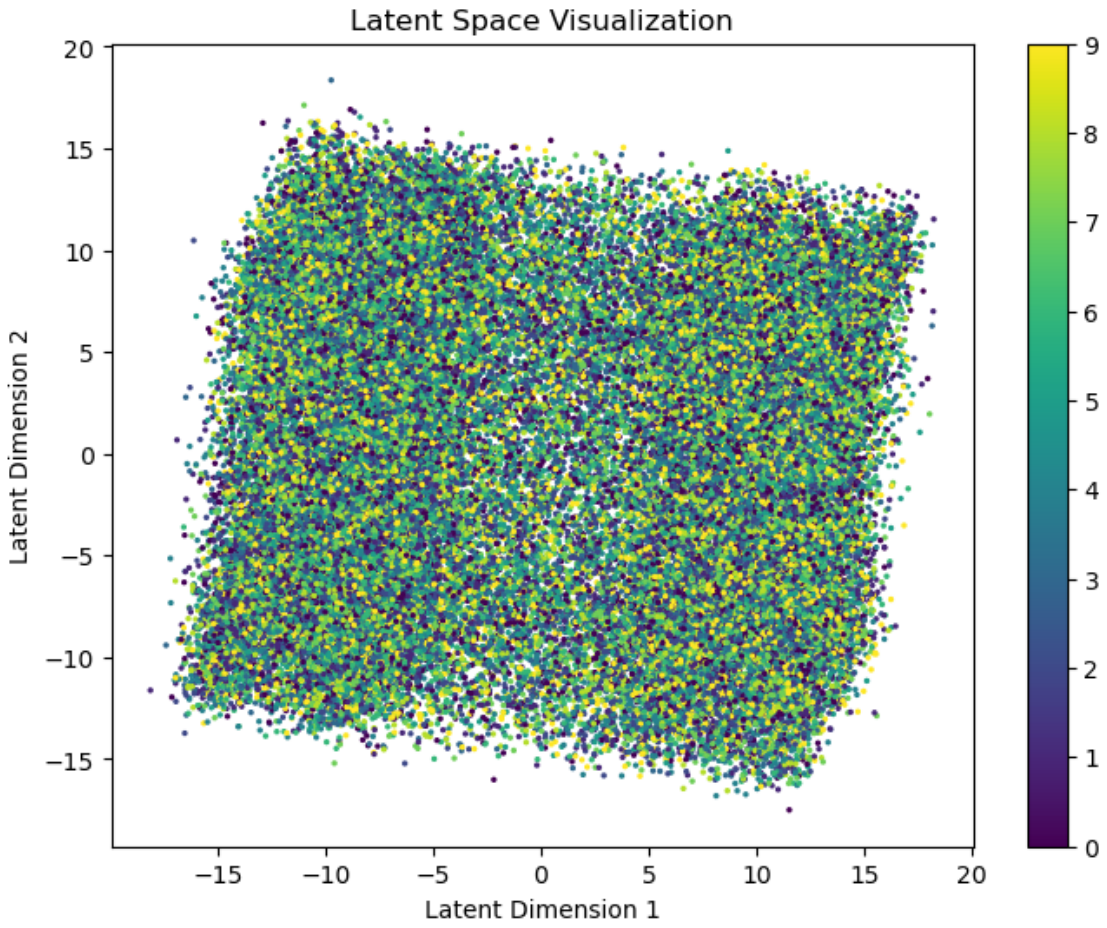
```
Epoch 1, Loss: 165.520616
Epoch 2, Loss: 128.902781
Epoch 3, Loss: 117.573229
Epoch 4, Loss: 112.241660
Epoch 5, Loss: 109.010816
Epoch 6, Loss: 106.742873
Epoch 7, Loss: 105.046126
Epoch 8, Loss: 103.527843
Epoch 9, Loss: 102.124691
Epoch 10, Loss: 100.774069
```

[40]:
```
import numpy as np
import matplotlib.pyplot as plt
from torchvision.utils import make_grid
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE
```

```
def visualize_samples_and_latent_space(model, data_loader, num_samples=16,␣
 ↪use_pca=True):
    model.eval()
    latent_dim = model.latent_dim  #

    with torch.no_grad():
        #
        z = torch.randn(num_samples, latent_dim).to(device)
        samples = model.generate(z).cpu()


        plt.figure(figsize=(4, 4))
        grid_img = make_grid(samples.view(num_samples, 1, 28, 28), nrow=4,␣
 ↪padding=2)
        plt.imshow(grid_img.permute(1, 2, 0).numpy(), cmap='gray')
        plt.axis('off')
        plt.show()


        latents = []
        labels = []
        for data, label in data_loader:
            data = data.to(device)
            mu, logvar = model.encode(data.view(-1, 784))
            z, _ = model.reparameterize(mu, logvar)
            latents.append(z.cpu().numpy())
            labels.append(label.numpy())

        latents = np.concatenate(latents)
        labels = np.concatenate(labels)


        if latent_dim > 2:
            if use_pca:
                latents_2d = PCA(n_components=2).fit_transform(latents)
            else:
                latents_2d = TSNE(n_components=2).fit_transform(latents)
        else:
            latents_2d = latents

        plt.figure(figsize=(8, 6))
        plt.scatter(latents_2d[:, 0], latents_2d[:, 1], c=labels,␣
 ↪cmap='viridis', s=2)
        plt.colorbar()
        plt.xlabel('Latent Dimension 1')
        plt.ylabel('Latent Dimension 2')
        plt.title('Latent Space Visualization')
```

```
        plt.show()
```

[41]:
```
visualize_samples_and_latent_space(vae_with_flow, train_loader)
```



[42]:
```
import torch
import torch.nn as nn

class MultivariateGKDistribution(nn.Module):
    def __init__(self, dim, a=0, b=1, g=0, k=0):
        super(MultivariateGKDistribution, self).__init__()
        self.dim = dim
        self.a = nn.Parameter(torch.tensor(a, dtype=torch.float32).repeat(dim))
        self.b = nn.Parameter(torch.tensor(b, dtype=torch.float32).repeat(dim))
        self.g = nn.Parameter(torch.tensor(g, dtype=torch.float32).repeat(dim))
        self.k = nn.Parameter(torch.tensor(k, dtype=torch.float32).repeat(dim))

    def forward(self, u):
        z = self.a + self.b * (1 + 0.8 * (1 - torch.exp(-self.g * u)) / (1 +␣
 ↪torch.exp(-self.g * u))) * (1 + u**2)**self.k * u
        return z
```

```
    def sample(self, num_samples):
        u = torch.randn(num_samples, self.dim, dtype=torch.float32)
        return self.forward(u)
from scipy.optimize import least_squares

def levenberg_marquardt_loss(params, model, target_distribution):
    dim = model.dim
    a, b, g, k = params[:dim], params[dim:2*dim], params[2*dim:3*dim],␣
 ↪params[3*dim:]
    model.a.data = torch.tensor(a, dtype=torch.float32)
    model.b.data = torch.tensor(b, dtype=torch.float32)
    model.g.data = torch.tensor(g, dtype=torch.float32)
    model.k.data = torch.tensor(k, dtype=torch.float32)

    samples = model.sample(len(target_distribution))
    loss = torch.mean((samples - target_distribution) ** 2).item()
    return loss

def optimize_gk_parameters(model, target_distribution):
    initial_params = torch.cat([model.a.data, model.b.data, model.g.data, model.
 ↪k.data]).numpy()
    result = least_squares(levenberg_marquardt_loss, initial_params,␣
 ↪args=(model, target_distribution), method='lm')
    optimized_params = result.x

    model.a.data = torch.tensor(optimized_params[:model.dim], dtype=torch.
 ↪float32)
    model.b.data = torch.tensor(optimized_params[model.dim:2*model.dim],␣
 ↪dtype=torch.float32)
    model.g.data = torch.tensor(optimized_params[2*model.dim:3*model.dim],␣
 ↪dtype=torch.float32)
    model.k.data = torch.tensor(optimized_params[3*model.dim:], dtype=torch.
 ↪float32)
class FlowLayer(nn.Module):
    def __init__(self, latent_dim):
        super(FlowLayer, self).__init__()
        self.u = nn.Parameter(torch.randn(latent_dim, dtype=torch.float32))
        self.w = nn.Parameter(torch.randn(latent_dim, 1, dtype=torch.float32)) ␣
 ↪# Adjusted to have a singleton dimension for broadcasting
        self.b = nn.Parameter(torch.zeros(1, dtype=torch.float32))

    def forward(self, z):
        linear = torch.matmul(z, self.w) + self.b
        activation = torch.tanh(linear)
        z_new = z + self.u.unsqueeze(0) * activation  # Broadcast u
```

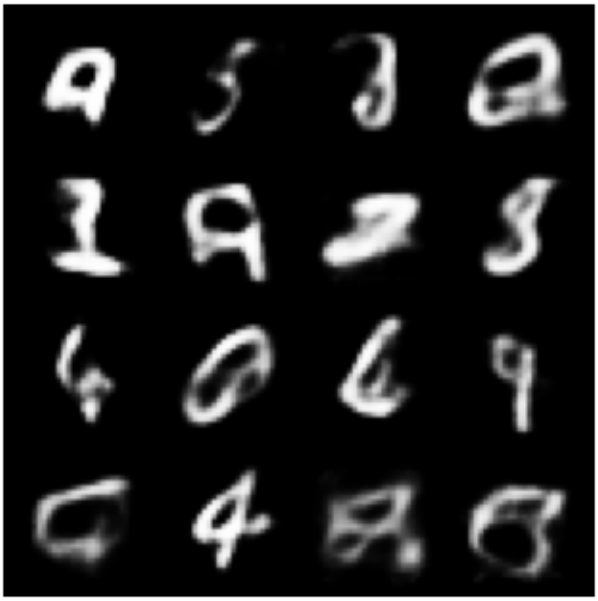
```
        psi = (1 - activation**2) * self.w.squeeze(1)  # Squeeze w to match the␣
 ↪psi calculation
        det_jacobian = torch.abs(1 + torch.matmul(psi, self.u))

        log_det_jacobian = torch.log(det_jacobian + 1e-6).float()

        return z_new, log_det_jacobian

class NormalizingFlow(nn.Module):
    def __init__(self, latent_dim, num_flows=2, base_dist=None):
        super(NormalizingFlow, self).__init__()
        self.flows = nn.ModuleList([FlowLayer(latent_dim) for _ in␣
 ↪range(num_flows)])
        self.base_dist = base_dist if base_dist is not None else␣
 ↪MultivariateGKDistribution(latent_dim)

    def forward(self, z):
        log_det_jacobian = torch.zeros(z.size(0), dtype=torch.float32, device=z.
 ↪device)
        for flow in self.flows:
            z, log_det = flow(z)
            log_det_jacobian += log_det
        return z, log_det_jacobian

    def sample(self, num_samples):
        z0 = self.base_dist.sample(num_samples)
        z, _ = self.forward(z0)
        return z
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

class VAEWithFlow(nn.Module):
    def __init__(self, input_dim=784, hidden_dim=400, latent_dim=50,␣
 ↪num_flows=2):
        super(VAEWithFlow, self).__init__()
        self.latent_dim = latent_dim
        self.encoder = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, latent_dim * 2)  # For mean and log variance
        )
        self.decoder = nn.Sequential(
            nn.Linear(latent_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, input_dim),
            nn.Sigmoid()  # Apply sigmoid to ensure output is in range [0, 1]
```

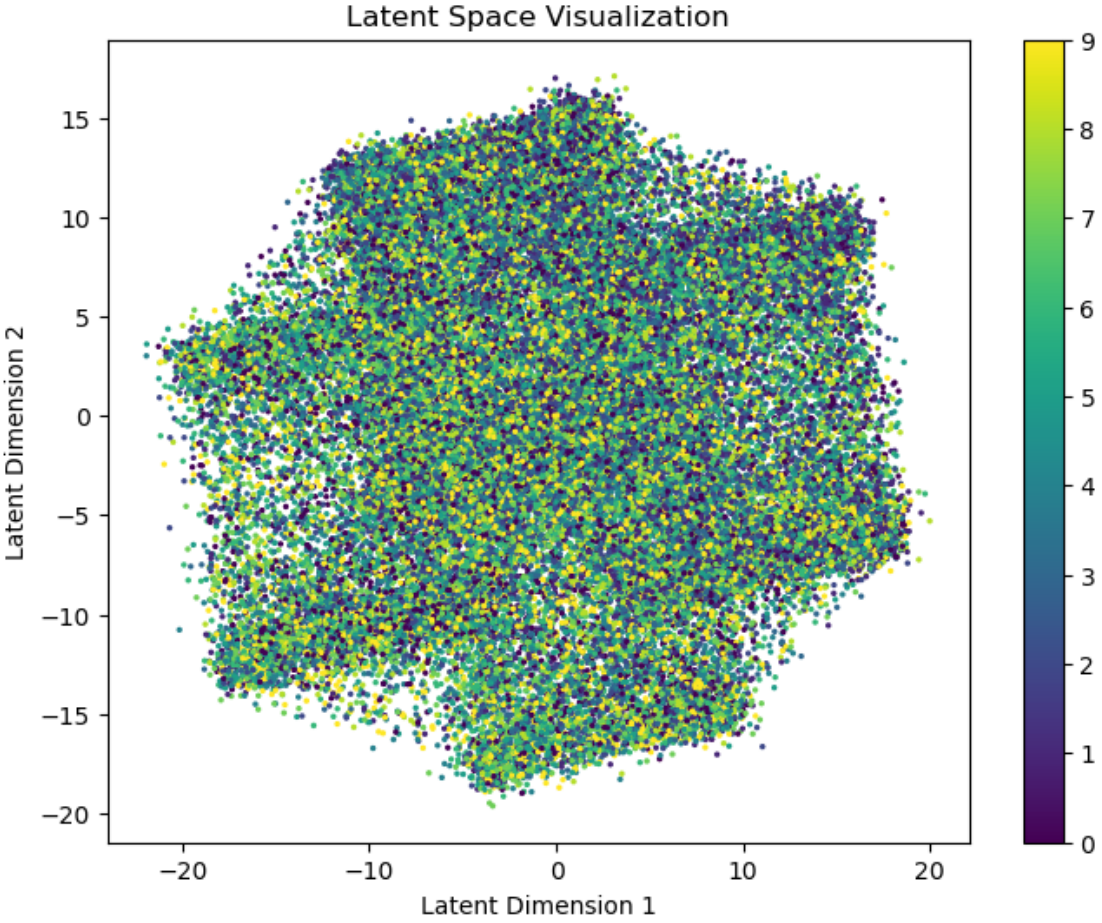
```
        )
        self.flow = NormalizingFlow(latent_dim, num_flows=num_flows)

    def encode(self, x):
        x = x.view(-1, 784)  # Flatten the input image to match the input_dim␣
 ↪of 784
        params = self.encoder(x)
        mu, logvar = params[:, :self.latent_dim], params[:, self.latent_dim:]
        return mu, logvar

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

    def decode(self, z):
        return self.decoder(z)  # Keep it as [batch_size, 784], values in [0, 1]

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

    def generate(self, z):
        return self.decode(z)  # Use the decoder to generate images from z


    # Define the MNIST dataset with the correct transformation
transform = transforms.Compose([
    transforms.ToTensor(),  # Converts the image to a tensor with pixel values␣
 ↪between 0 and 1
])

train_dataset = datasets.MNIST(root='./data', train=True, transform=transform,␣
 ↪download=True)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
# Initialize the VAE model with Normalizing Flow
vae_with_flow = VAEWithFlow(input_dim=784, hidden_dim=400, latent_dim=50,␣
 ↪num_flows=4).to(device)

# Define the optimizer
optimizer = optim.Adam(vae_with_flow.parameters(), lr=1e-3)

def loss_function(recon_x, x, mu, logvar, log_det_jacobian):
```

```
    # Flatten the input to match the output of the decoder
    x = x.view(-1, 784)  # Flatten target to [batch_size, 784]

    BCE = nn.functional.binary_cross_entropy(recon_x, x, reduction='sum')
    KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
    return BCE + KLD - torch.sum(log_det_jacobian)

for epoch in range(1, 11):
    train_loss = 0
    vae_with_flow.train()

    for batch_idx, (data, _) in enumerate(train_loader):
        data = data.to(device)
        optimizer.zero_grad()
        recon_batch, mu, logvar, log_det_jacobian = vae_with_flow(data)


        loss = loss_function(recon_batch, data, mu, logvar, log_det_jacobian)
        loss.backward()
        train_loss += loss.item()
        optimizer.step()

    print(f'Epoch {epoch}, Loss: {train_loss / len(train_loader.dataset):.6f}')
```

```
Epoch 1, Loss: 170.042920
Epoch 2, Loss: 133.808491
Epoch 3, Loss: 121.290844
Epoch 4, Loss: 114.870641
Epoch 5, Loss: 111.416504
Epoch 6, Loss: 109.153973
Epoch 7, Loss: 107.344833
Epoch 8, Loss: 105.460621
Epoch 9, Loss: 103.956071
Epoch 10, Loss: 102.798736
```

[43]:
```
# Example usage after training:
visualize_samples_and_latent_space(vae_with_flow, train_loader)
```




Latent Space Visualization
Latent Dimension 1
Latent Dimension 2
15
10
5
0
−5
−10
−15
−20
−20
−10
0
10
20
9
8
7
6
5
4
3
2
1
0

[44]:
```
# Potential spatial distribution generated using training data
latent_samples = []
vae_with_flow.eval()
with torch.no_grad():
    for data, _ in train_loader:
        data = data.to(device)
        mu, logvar = vae_with_flow.encode(data)
        z = vae_with_flow.reparameterize(mu, logvar)[0]
        latent_samples.append(z.cpu())

target_distribution = torch.cat(latent_samples, dim=0)
```

[45]:
```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
from scipy.optimize import least_squares

# Multivariate g-k Distribution
class MultivariateGKDistribution(nn.Module):
    def __init__(self, dim, a=0, b=1, g=0, k=0):
        super(MultivariateGKDistribution, self).__init__()
        self.dim = dim
        self.a = nn.Parameter(torch.tensor(a, dtype=torch.float32).repeat(dim))
        self.b = nn.Parameter(torch.tensor(b, dtype=torch.float32).repeat(dim))
        self.g = nn.Parameter(torch.tensor(g, dtype=torch.float32).repeat(dim))
        self.k = nn.Parameter(torch.tensor(k, dtype=torch.float32).repeat(dim))

    def forward(self, u):
        z = self.a + self.b * (1 + 0.8 * (1 - torch.exp(-self.g * u)) / (1 +␣
 ↪torch.exp(-self.g * u))) * (1 + u**2)**self.k * u
        return z

    def sample(self, num_samples):
        u = torch.randn(num_samples, self.dim, dtype=torch.float32)
        return self.forward(u)

# Levenberg-Marquardt Loss Function
def levenberg_marquardt_loss(params, model, target_distribution):
    dim = model.dim
    a, b, g, k = params[:dim], params[dim:2*dim], params[2*dim:3*dim],␣
 ↪params[3*dim:]
    model.a.data = torch.tensor(a, dtype=torch.float32)
```

```
    model.b.data = torch.tensor(b, dtype=torch.float32)
    model.g.data = torch.tensor(g, dtype=torch.float32)
    model.k.data = torch.tensor(k, dtype=torch.float32)

    samples = model.sample(len(target_distribution))
    loss = torch.mean((samples - target_distribution) ** 2).item()
    return loss

def optimize_gk_parameters(model, target_distribution):
    initial_params = torch.cat([model.a.data, model.b.data, model.g.data, model.
 ↪k.data]).numpy()

    #   'trf'    'lm'   'trf'   'lm'
    result = least_squares(levenberg_marquardt_loss, initial_params,␣
 ↪args=(model, target_distribution), method='trf')

    optimized_params = result.x

    model.a.data = torch.tensor(optimized_params[:model.dim], dtype=torch.
 ↪float32)
    model.b.data = torch.tensor(optimized_params[model.dim:2*model.dim],␣
 ↪dtype=torch.float32)
    model.g.data = torch.tensor(optimized_params[2*model.dim:3*model.dim],␣
 ↪dtype=torch.float32)
    model.k.data = torch.tensor(optimized_params[3*model.dim:], dtype=torch.
 ↪float32)



# Flow Layer and Normalizing Flow
class FlowLayer(nn.Module):
    def __init__(self, latent_dim):
        super(FlowLayer, self).__init__()
        self.u = nn.Parameter(torch.randn(latent_dim, dtype=torch.float32))
        self.w = nn.Parameter(torch.randn(latent_dim, 1, dtype=torch.float32))
        self.b = nn.Parameter(torch.zeros(1, dtype=torch.float32))

    def forward(self, z):
        linear = torch.matmul(z, self.w) + self.b
        activation = torch.tanh(linear)
        z_new = z + self.u.unsqueeze(0) * activation

        psi = (1 - activation**2) * self.w.squeeze(1)
        det_jacobian = torch.abs(1 + torch.matmul(psi, self.u))

        log_det_jacobian = torch.log(det_jacobian + 1e-6).float()
```

```
        return z_new, log_det_jacobian

class NormalizingFlow(nn.Module):
    def __init__(self, latent_dim, num_flows=2, base_dist=None):
        super(NormalizingFlow, self).__init__()
        self.flows = nn.ModuleList([FlowLayer(latent_dim) for _ in␣
 ↪range(num_flows)])
        self.base_dist = base_dist if base_dist is not None else␣
 ↪MultivariateGKDistribution(latent_dim)

    def forward(self, z):
        log_det_jacobian = torch.zeros(z.size(0), dtype=torch.float32, device=z.
 ↪device)
        for flow in self.flows:
            z, log_det = flow(z)
            log_det_jacobian += log_det
        return z, log_det_jacobian

    def sample(self, num_samples):
        z0 = self.base_dist.sample(num_samples)
        z, _ = self.forward(z0)
        return z

# VAE with Normalizing Flow and g-k Distribution
class VAEWithFlow(nn.Module):
    def __init__(self, input_dim=784, hidden_dim=400, latent_dim=50,␣
 ↪num_flows=2):
        super(VAEWithFlow, self).__init__()
        self.latent_dim = latent_dim
        self.encoder = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, latent_dim * 2)  # For mean and log variance
        )
        self.decoder = nn.Sequential(
            nn.Linear(latent_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, input_dim),
            nn.Sigmoid()
        )
        self.flow = NormalizingFlow(latent_dim, num_flows=num_flows)

    def encode(self, x):
        x = x.view(-1, 784)
        params = self.encoder(x)
        mu, logvar = params[:, :self.latent_dim], params[:, self.latent_dim:]
```

```
        return mu, logvar

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

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

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

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

# Define the MNIST dataset with the correct transformation
transform = transforms.Compose([
    transforms.ToTensor(),
])

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

# Initialize the VAE model with Normalizing Flow and g-k distribution
vae_with_flow = VAEWithFlow(input_dim=784, hidden_dim=400, latent_dim=50,␣
 ↪num_flows=4).to(device)

# Levenberg-Marquardt optimization of g-k distribution before training
# Here, we assume target_distribution is some pre-defined target data you wish␣
 ↪to fit the g-k distribution to
# Replace with actual target distribution data
optimize_gk_parameters(vae_with_flow.flow.base_dist, target_distribution)

# Define the optimizer
optimizer = optim.Adam(vae_with_flow.parameters(), lr=1e-3)

def loss_function(recon_x, x, mu, logvar, log_det_jacobian):
    x = x.view(-1, 784)
    BCE = nn.functional.binary_cross_entropy(recon_x, x, reduction='sum')
    KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
```

```
    return BCE + KLD - torch.sum(log_det_jacobian)

for epoch in range(1, 11):
    train_loss = 0
    vae_with_flow.train()
    for batch_idx, (data, _) in enumerate(train_loader):
        data = data.to(device)
        optimizer.zero_grad()
        recon_batch, mu, logvar, log_det_jacobian = vae_with_flow(data)
        loss = loss_function(recon_batch, data, mu, logvar, log_det_jacobian)
        loss.backward()
        train_loss += loss.item()
        optimizer.step()

    print(f'Epoch {epoch}, Loss: {train_loss / len(train_loader.dataset):.6f}')
```

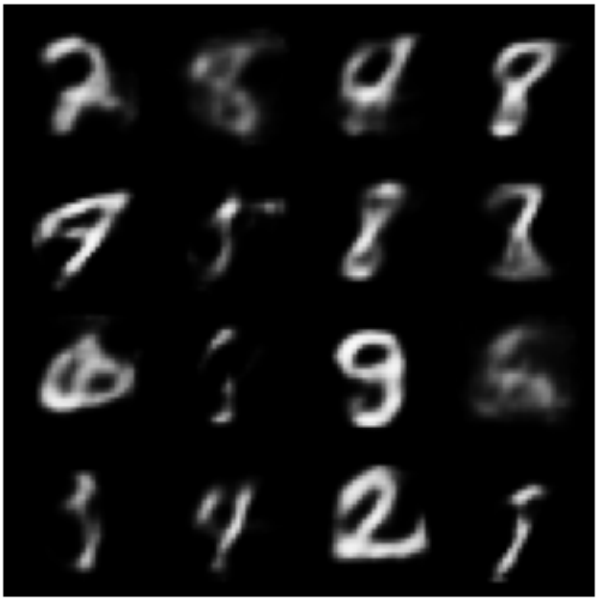
```
Epoch 1, Loss: 179.468242
Epoch 2, Loss: 155.490709
Epoch 3, Loss: 143.635647
Epoch 4, Loss: 135.820261
Epoch 5, Loss: 140.466578
Epoch 6, Loss: 140.380552
Epoch 7, Loss: 138.572246
Epoch 8, Loss: 133.961569
Epoch 9, Loss: 133.757182
Epoch 10, Loss: 144.803902
```

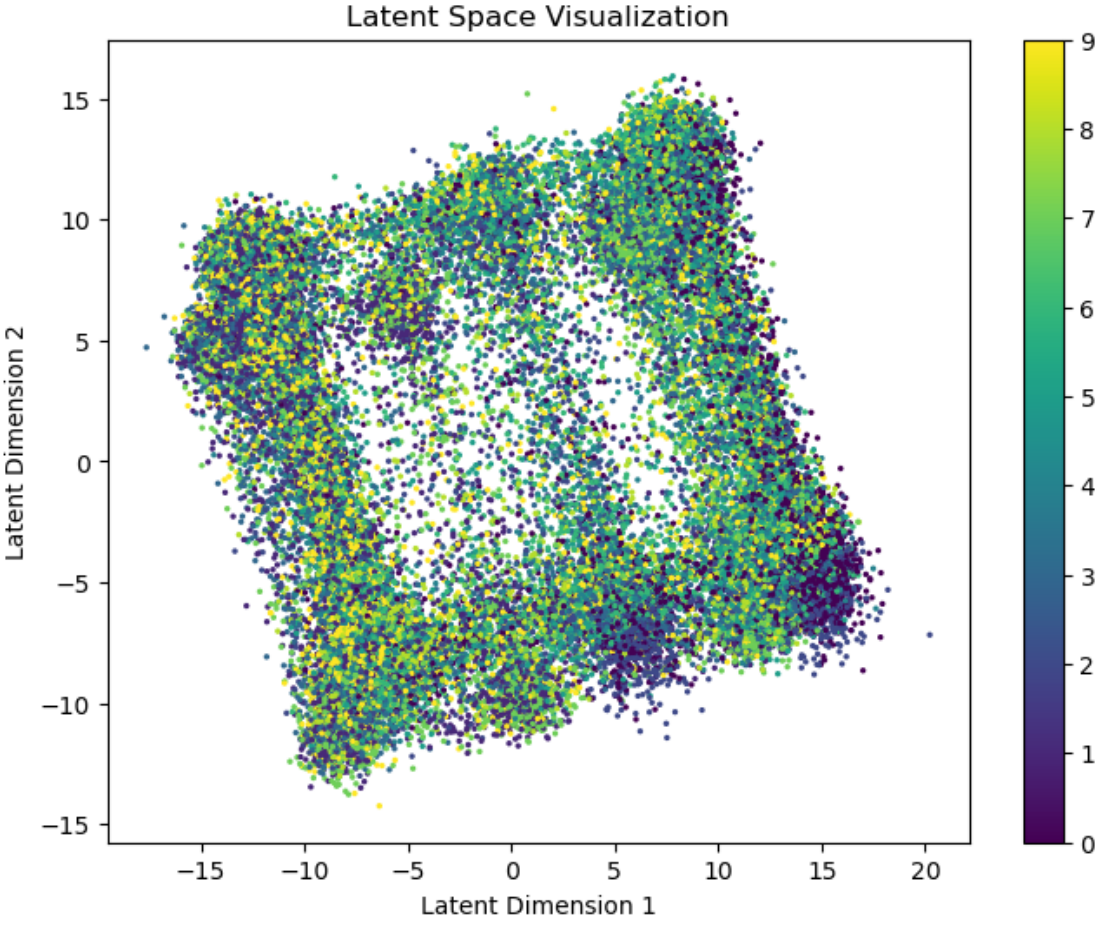
[46]:
```
# Define the MNIST dataset with the correct transformation
transform = transforms.Compose([
    transforms.ToTensor(),
])

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

# Initialize the VAE model with Normalizing Flow and g-k distribution
vae_with_flow = VAEWithFlow(input_dim=784, hidden_dim=400, latent_dim=50,␣
 ↪num_flows=4).to(device)

# Levenberg-Marquardt optimization of g-k distribution before training
# Here, we assume target_distribution is some pre-defined target data you wish␣
 ↪to fit the g-k distribution to

optimize_gk_parameters(vae_with_flow.flow.base_dist, target_distribution)

# Define the optimizer
```

```
optimizer = optim.Adam(vae_with_flow.parameters(), lr=1e-3)

def loss_function(recon_x, x, mu, logvar, log_det_jacobian):
    x = x.view(-1, 784)
    BCE = nn.functional.binary_cross_entropy(recon_x, x, reduction='sum')
    KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
    return BCE + KLD - torch.sum(log_det_jacobian)

for epoch in range(1, 11):
    train_loss = 0
    vae_with_flow.train()
    for batch_idx, (data, _) in enumerate(train_loader):
        data = data.to(device)
        optimizer.zero_grad()
        recon_batch, mu, logvar, log_det_jacobian = vae_with_flow(data)
        loss = loss_function(recon_batch, data, mu, logvar, log_det_jacobian)
        loss.backward()
        train_loss += loss.item()
        optimizer.step()

    print(f'Epoch {epoch}, Loss: {train_loss / len(train_loader.dataset):.6f}')
```

```
Epoch 1, Loss: 179.615487
Epoch 2, Loss: 151.809655
Epoch 3, Loss: 146.450001
Epoch 4, Loss: 142.862890
Epoch 5, Loss: 140.388594
Epoch 6, Loss: 153.785822
Epoch 7, Loss: 139.039761
Epoch 8, Loss: 134.541962
Epoch 9, Loss: 136.854830
Epoch 10, Loss: 131.666251
```

[47]:
```
#
visualize_samples_and_latent_space(vae_with_flow, train_loader)
```




Latent Space Visualization
Latent Dimension 1
Latent Dimension 2
15
10
5
0
−5
−10
−15
−15
−10
−5
0
5
10
15
20
9
8
7
6
5
4
3
2
1
0

4.0.1 Advantages of This Plot:

1. **Clear Boundaries**:
   - Distinct classification boundaries suggest well-defined regions for different digits in the latent space, aiding clear differentiation during generation.
2. **Concentrated Regions**:
   - The data points are more concentrated, indicating deeper recognition of specific digits, which supports generating distinct, well-separated digits.
3. **Better Separation**:
   - The clear boundaries and dispersed distribution help the model better identify and generate each digit.

4.0.2 Features of the Previous Plot:

1. **Broader Coverage**:
   - The previous plot shows an even distribution over a larger latent space, which might help generate diverse samples but could reduce classification clarity.
2. **Higher Ambiguity**:
   - The lack of distinct boundaries in the previous plot may lead to less clear differentiation between digits.

4.0.3 Conclusion:

This plot better aligns with our goal of clearly distinguishing digits, with superior classification boundaries and concentrated regions, making it more effective for our needs.

[51]:
```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
from scipy.optimize import least_squares

# Multivariate g-k Distribution
class MultivariateGKDistribution(nn.Module):
    def __init__(self, dim, a=0, b=1, g=0, k=0):
        super(MultivariateGKDistribution, self).__init__()
        self.dim = dim
        self.a = nn.Parameter(torch.tensor(a, dtype=torch.float32).repeat(dim))
        self.b = nn.Parameter(torch.tensor(b, dtype=torch.float32).repeat(dim))
        self.g = nn.Parameter(torch.tensor(g, dtype=torch.float32).repeat(dim))
        self.k = nn.Parameter(torch.tensor(k, dtype=torch.float32).repeat(dim))

    def forward(self, u):
        z = self.a + self.b * (1 + 0.8 * (1 - torch.exp(-self.g * u)) / (1 +␣
 ↪torch.exp(-self.g * u))) * (1 + u**2)**self.k * u
```

```
        return z

    def sample(self, num_samples):
        u = torch.randn(num_samples, self.dim, dtype=torch.float32)
        return self.forward(u)

# Levenberg-Marquardt Loss Function
def levenberg_marquardt_loss(params, model, target_distribution):
    dim = model.dim
    a, b, g, k = params[:dim], params[dim:2*dim], params[2*dim:3*dim],␣
 ↪params[3*dim:]
    model.a.data = torch.tensor(a, dtype=torch.float32)
    model.b.data = torch.tensor(b, dtype=torch.float32)
    model.g.data = torch.tensor(g, dtype=torch.float32)
    model.k.data = torch.tensor(k, dtype=torch.float32)

    samples = model.sample(len(target_distribution))
    loss = torch.mean((samples - target_distribution) ** 2).item()
    return loss

def optimize_gk_parameters(model, target_distribution):
    initial_params = torch.cat([model.a.data, model.b.data, model.g.data, model.
 ↪k.data]).numpy()

    result = least_squares(levenberg_marquardt_loss, initial_params,␣
 ↪args=(model, target_distribution), method='trf')

    optimized_params = result.x

    model.a.data = torch.tensor(optimized_params[:model.dim], dtype=torch.
 ↪float32)
    model.b.data = torch.tensor(optimized_params[model.dim:2*model.dim],␣
 ↪dtype=torch.float32)
    model.g.data = torch.tensor(optimized_params[2*model.dim:3*model.dim],␣
 ↪dtype=torch.float32)
    model.k.data = torch.tensor(optimized_params[3*model.dim:], dtype=torch.
 ↪float32)

# Flow Layers and Normalizing Flow
class FlowLayer(nn.Module):
    def __init__(self, latent_dim):
        super(FlowLayer, self).__init__()
        self.u = nn.Parameter(torch.randn(latent_dim, dtype=torch.float32))
        self.w = nn.Parameter(torch.randn(latent_dim, 1, dtype=torch.float32))
        self.b = nn.Parameter(torch.zeros(1, dtype=torch.float32))
```

```
    def forward(self, z):
        linear = torch.matmul(z, self.w) + self.b
        activation = torch.tanh(linear)
        z_new = z + self.u.unsqueeze(0) * activation

        psi = (1 - activation**2) * self.w.squeeze(1)
        det_jacobian = torch.abs(1 + torch.matmul(psi, self.u))

        log_det_jacobian = torch.log(det_jacobian + 1e-6).float()

        return z_new, log_det_jacobian

# Introducing Radial Flow for more complex transformations
class RadialFlow(nn.Module):
    def __init__(self, latent_dim):
        super(RadialFlow, self).__init__()
        self.z0 = nn.Parameter(torch.randn(1, latent_dim, dtype=torch.float32))
        self.alpha = nn.Parameter(torch.randn(1, dtype=torch.float32))
        self.beta = nn.Parameter(torch.randn(1, dtype=torch.float32))

    def forward(self, z):
        r = torch.norm(z - self.z0, dim=1, keepdim=True)
        h = 1 / (self.alpha + r)
        beta_h = self.beta * h
        z_new = z + beta_h * (z - self.z0)
        log_det_jacobian = torch.log(1 + beta_h * (1 - beta_h * r)).sum(dim=1)
        return z_new, log_det_jacobian

class NormalizingFlow(nn.Module):
    def __init__(self, latent_dim, num_flows=2, base_dist=None):
        super(NormalizingFlow, self).__init__()
        #
        self.flows = nn.ModuleList(
            [FlowLayer(latent_dim) if i % 2 == 0 else RadialFlow(latent_dim)␣
 ↪for i in range(num_flows)]
        )
        self.base_dist = base_dist if base_dist is not None else␣
 ↪MultivariateGKDistribution(latent_dim)

    def forward(self, z):
        log_det_jacobian = torch.zeros(z.size(0), dtype=torch.float32, device=z.
 ↪device)
        for flow in self.flows:
            z, log_det = flow(z)
            log_det_jacobian += log_det
        return z, log_det_jacobian
```

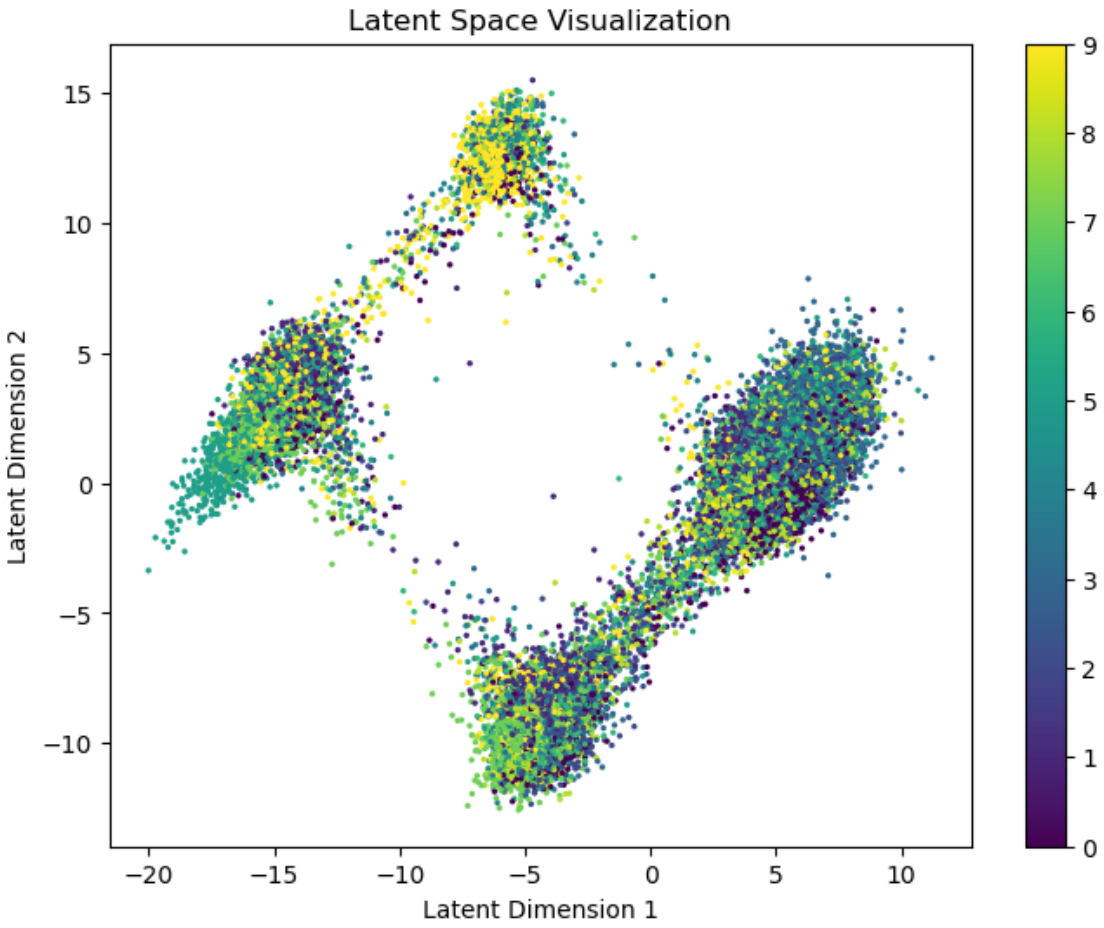
```
    def sample(self, num_samples):
        z0 = self.base_dist.sample(num_samples)
        z, _ = self.forward(z0)
        return z

# VAE with Complex Normalizing Flow and g-k Distribution
class VAEWithFlow(nn.Module):
    def __init__(self, input_dim=784, hidden_dim=400, latent_dim=50,␣
 ↪num_flows=6):  #   num_flows
        super(VAEWithFlow, self).__init__()
        self.latent_dim = latent_dim
        self.encoder = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, latent_dim * 2)  # For mean and log variance
        )
        self.decoder = nn.Sequential(
            nn.Linear(latent_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, input_dim),
            nn.Sigmoid()
        )
        self.flow = NormalizingFlow(latent_dim, num_flows=num_flows)  #

    def encode(self, x):
        x = x.view(-1, 784)
        params = self.encoder(x)
        mu, logvar = params[:, :self.latent_dim], params[:, self.latent_dim:]
        return mu, logvar

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

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

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

    def generate(self, z):
```

```
        return self.decode(z)

# Define the MNIST dataset with the correct transformation
transform = transforms.Compose([
    transforms.ToTensor(),
])

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

# Initialize the VAE model with Complex Normalizing Flow and g-k distribution
vae_with_flow = VAEWithFlow(input_dim=784, hidden_dim=400, latent_dim=50,␣
 ↪num_flows=6).to(device)

# Levenberg-Marquardt optimization of g-k distribution before training
# Here, we assume target_distribution is some pre-defined target data you wish␣
 ↪to fit the g-k distribution to
# Replace with actual target distribution data
optimize_gk_parameters(vae_with_flow.flow.base_dist, target_distribution)

# Define the optimizer
optimizer = optim.Adam(vae_with_flow.parameters(), lr=1e-3)

def loss_function(recon_x, x, mu, logvar, log_det_jacobian):
    x = x.view(-1, 784)
    BCE = nn.functional.binary_cross_entropy(recon_x, x, reduction='sum')
    KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
    return BCE + KLD - torch.sum(log_det_jacobian)

for epoch in range(1, 11):
    train_loss = 0
    vae_with_flow.train()
    for batch_idx, (data, _) in enumerate(train_loader):
        data = data.to(device)
        optimizer.zero_grad()
        recon_batch, mu, logvar, log_det_jacobian = vae_with_flow(data)
        loss = loss_function(recon_batch, data, mu, logvar, log_det_jacobian)
        loss.backward()
        train_loss += loss.item()
        optimizer.step()

    print(f'Epoch {epoch}, Loss: {train_loss / len(train_loader.dataset):.6f}')
```

```
Epoch 1, Loss: 187.225034
Epoch 2, Loss: 159.982319
Epoch 3, Loss: 157.670301
```

```
Epoch 4, Loss: 142.617740
Epoch 5, Loss: 147.140170
Epoch 6, Loss: 144.306917
Epoch 7, Loss: 151.251292
Epoch 8, Loss: 145.453654
Epoch 9, Loss: 142.763340
Epoch 10, Loss: 152.582088
```

[52]:
```
#
visualize_samples_and_latent_space(vae_with_flow, train_loader)
```



4.0.4 Advantages:

1. **Clearer Classification**:
   - The data points in this new plot are distinctly grouped into three clusters, indicating that the model effectively separates different categories (likely different digits) in the latent space. This clustered distribution aids in clearer classification of different digits.
2. **Higher Separation in Latent Space**:
   - The noticeable gaps between clusters suggest a higher degree of separation between different categories. This distribution is beneficial for accurately distinguishing and recognizing different categories during the generation process.
3. **Tight Cluster Distribution**:
   - The data points within each cluster are tightly packed, indicating the model's high consistency in generating samples of the same category, which helps reduce confusion between categories during generation.

4.0.5 Comparison with Previous Plots:

- **Compared to the Previous Plots**:

  - The new plot shows more distinct classification, with clearer clusters and gaps between them. This suggests that the new model may better differentiate between different digits in the generation task.
- **Latent Space Coverage**:
  - While the new plot has a less extensive coverage of the latent space compared to the previous plots, it offers more explicit and clear classification. Therefore, if the goal is to clearly distinguish between different categories of digits, the new result might be better.

4.0.6 Conclusion:

The new plot performs the best, showing clearer category separation and tighter intra-category grouping in the latent space. This is highly advantageous for accurately distinguishing different digits in the generation task, making the algorithm behind the new plot more effective. Thank you!

[ ]: