Here's a complete implementation of a **Variational Autoencoder (VAE)** for Fashion-MNIST reconstruction in PyTorch, with visualization:

```
# File: 03_VAE_FashionMNIST_Reconstruction.py
# Topic: VAE implementation for Fashion-MNIST reconstruction using
PyTorch

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision
from torchvision import transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
import numpy as np
from torch.utils.tensorboard import SummaryWriter
import os

# Set random seed for reproducibility
torch.manual_seed(42)
np.random.seed(42)

# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Hyperparameters
latent_dim = 20
batch_size = 128
epochs = 30
learning_rate = 1e-3
image_size = 28 * 28  # Fashion-MNIST dimensions

# Create output directory
os.makedirs('vae_results', exist_ok=True)
os.makedirs('vae_results/reconstructions', exist_ok=True)
writer = SummaryWriter('vae_results/runs')

# Fashion-MNIST dataset
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))  # Scale to [-1, 1]
])

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

# VAE Model
class VAE(nn.Module):
```

PROF

```
    def __init__(self):
        super(VAE, self).__init__()

        # Encoder
        self.fc1 = nn.Linear(image_size, 400)
        self.fc21 = nn.Linear(400, latent_dim)  # μ (mean)
        self.fc22 = nn.Linear(400, latent_dim)  # logvar

        # Decoder
        self.fc3 = nn.Linear(latent_dim, 400)
        self.fc4 = nn.Linear(400, image_size)

    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.tanh(self.fc4(h3))  # Tanh to match input range
[-1,1]

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

# Loss function
def loss_function(recon_x, x, mu, logvar):
    BCE = F.mse_loss(recon_x, x.view(-1, image_size), reduction='sum')
    KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
    return BCE + KLD

# Initialize model and optimizer
model = VAE().to(device)
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

# Fixed input for visualization
fixed_input = next(iter(train_loader))[0][:32].to(device)

def save_reconstructions(epoch, original, reconstructed):
    """Save original and reconstructed images for comparison"""
    original = original.cpu().numpy()
    reconstructed = reconstructed.cpu().numpy()

    plt.figure(figsize=(10, 4))
    for i in range(8):  # Display first 8 samples
        # Original images
        ax = plt.subplot(2, 8, i + 1)
```

```
            plt.imshow(original[i].reshape(28, 28), cmap='gray')
            plt.axis('off')

            # Reconstructed images
            ax = plt.subplot(2, 8, i + 8 + 1)
            plt.imshow(reconstructed[i].reshape(28, 28), cmap='gray')
            plt.axis('off')

        plt.savefig(f'vae_results/reconstructions/epoch_{epoch:03d}.png')
        plt.close()

    # Training loop
    for epoch in range(epochs):
        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)} '
                      f'({100. * batch_idx /
    len(train_loader):.0f}%)]\tLoss: {loss.item() / len(data):.6f}')

        # Validation and visualization
        model.eval()
        with torch.no_grad():
            # Calculate validation loss
            val_loss = 0
            for data, _ in train_loader:  # Using same data for simplicity
                data = data.to(device)
                recon, mu, logvar = model(data)
                val_loss += loss_function(recon, data, mu, logvar).item()

            # Generate reconstructions
            recon_fixed, _, _ = model(fixed_input)

            # Save reconstructions
            save_reconstructions(epoch, fixed_input, recon_fixed)

            # TensorBoard logging
            writer.add_scalar('Training Loss', train_loss /
    len(train_loader.dataset), epoch)
            writer.add_scalar('Validation Loss', val_loss /
    len(train_loader.dataset), epoch)
```

```
        # Save sample to TensorBoard
        comparison = torch.cat([
            fixed_input[:8],
            recon_fixed.view(-1, 1, 28, 28)[:8]
        ])
        writer.add_images(
            'Original vs Reconstructed',
            (comparison + 1) / 2,  # Scale to [0,1]
            epoch
        )

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

# Save model
torch.save(model.state_dict(), 'vae_results/vae_fashionmnist.pth')
writer.close()

# Generate new samples
with torch.no_grad():
    sample = torch.randn(64, latent_dim).to(device)
    generated = model.decode(sample).cpu()

    plt.figure(figsize=(10, 10))
    for i in range(64):
        plt.subplot(8, 8, i + 1)
        plt.imshow(generated[i].view(28, 28), cmap='gray')
        plt.axis('off')
    plt.savefig('vae_results/generated_samples.png')
    plt.close()
```

Key Components Explained:

1. **VAE Architecture**:
   - **Encoder**: Maps input to latent space (μ and logvar)
   - **Reparameterization Trick**: Enables backpropagation through random sampling
   - **Decoder**: Reconstructs input from latent space
2. **Loss Function**:
   - **Reconstruction Loss (MSE)**: Measures pixel-wise difference
   - **KL Divergence**: Regularizes latent space to be close to standard normal
3. **Fashion-MNIST Specifics**:
   - Input images normalized to [-1, 1] range
   - Tanh activation in decoder matches input range
   - 28x28 grayscale images
4. **Visualization Features**:

PROF

- Saves original vs reconstructed comparisons each epoch
- Generates new samples from random latent vectors
- TensorBoard integration for loss tracking

Expected Output:

```
vae_results/
├── reconstructions/
│   ├── epoch_000.png
│   ├── epoch_001.png
│   └── ...
├── runs/ (TensorBoard logs)
├── generated_samples.png
└── vae_fashionmnist.pth
```

Monitoring Training:

```
tensorboard --logdir vae_results/runs
```

Common Improvements:

1. **For Better Quality**:

```
# Replace fc layers with convolutional layers
class ConvVAE(nn.Module):
    def __init__(self):
        super().__init__()
        # Encoder
        self.encoder = nn.Sequential(
            nn.Conv2d(1, 32, 3, stride=2, padding=1),
            nn.ReLU(),
            nn.Conv2d(32, 64, 3, stride=2, padding=1),
            nn.ReLU()
        )
        # Latent space
        self.fc_mu = nn.Linear(64*7*7, latent_dim)
        self.fc_logvar = nn.Linear(64*7*7, latent_dim)
        # Decoder
        self.decoder_input = nn.Linear(latent_dim, 64*7*7)
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(64, 32, 3, stride=2, padding=1,
output_padding=1),
            nn.ReLU(),
            nn.ConvTranspose2d(32, 1, 3, stride=2, padding=1,
output_padding=1),
```

```
            nn.Tanh()
        )
```

2. **For Different Objectives**:
    - Use BCE loss instead of MSE for sharper images
    - Add perceptual loss with a pre-trained network
3. **For Advanced Use**:
    - Implement β-VAE for disentangled representations
    - Add classifier guidance for conditional generation

Would you like me to modify any aspect of this implementation or add specific features like:

- Latent space interpolation visualization?
- Anomaly detection capabilities?
- Quantitative evaluation metrics (FID, etc.)?