Variational Autoencoders (VAEs) with PyTorch

Project Overview

This project demonstrates the implementation of a Variational Autoencoder (VAE) using PyTorch. VAEs are generative models that learn to represent high-dimensional data in a lower-dimensional latent space, enabling the generation of new, similar data samples. This implementation includes data preparation, model architecture, training procedures, and visualization of results.

Prerequisites

Required Libraries

- Python 3.7 or later
- `torch`: For building and training the neural network.
- `numpy`: For numerical computations.
- `matplotlib`: For data visualization.
- `scikit-learn`: For dataset splitting.

Installation

Install the necessary libraries using pip:

```
pip install torch numpy matplotlib scikit-learn
```

Dataset Preparation

For demonstration purposes, we'll generate a synthetic dataset:

```
import numpy as np
from sklearn.model_selection import train_test_split
import torch
from torch.utils.data import DataLoader, TensorDataset

# Generating synthetic data
X = np.random.randn(1000, 20)

# Splitting data into training and test sets
X_train, X_test = train_test_split(X, test_size=0.2, random_state=42)

# Creating DataLoader for training
train_dataset = TensorDataset(torch.Tensor(X_train))
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
```

Model Architecture

The VAE consists of an encoder, a reparameterization step, and a decoder:

```
import torch.nn as nn

class VAE(nn.Module):
    def __init__(self, input_dim, latent_dim):
        super(VAE, self).__init__()
        # Encoder layers
        self.fc1 = nn.Linear(input_dim, 512)
        self.fc2_mean = nn.Linear(512, latent_dim)
        self.fc2_logvar = nn.Linear(512, latent_dim)

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

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

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

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

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

Training the Model

Define the loss function and training loop:

```
 import torch.optim as optim

# Initialize VAE
vae = VAE(input_dim=20, latent_dim=5)
optimizer = optim.Adam(vae.parameters(), lr=1e-3)

# Loss function
def loss_function(recon_x, x, mu, logvar):
    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

# Training loop
epochs = 10
vae.train()
for epoch in range(epochs):
    train_loss = 0
    for data in train_loader:
        data = data[0]
        optimizer.zero_grad()
        recon_batch, mu, logvar = vae(data)
        loss = loss_function(recon_batch, data, mu, logvar)
        loss.backward()
        train_loss += loss.item()
        optimizer.step()
    print(f"Epoch {epoch+1}, Loss: {train_loss / len(train_loader.dataset):.4f}")
```

Evaluating the Model

Visualize the reconstruction performance:

```
 import matplotlib.pyplot as plt

vae.eval()
with torch.no_grad():
    sample = torch.Tensor(X_test)
    reconstructed, _, _ = vae(sample)

    # Plot original vs reconstructed for a sample
    plt.figure(figsize=(8, 4))
    plt.subplot(1, 2, 1)
    plt.plot(X_test[0], label='Original')
    plt.title('Original Data')
    plt.subplot(1, 2, 2)
    plt.plot(reconstructed[0].numpy(), label='Reconstructed')
    plt.title('Reconstructed Data')
    plt.show()
```

References

For further reading and tutorials on VAEs with PyTorch:

- A simple tutorial of Variational AutoEncoders with PyTorch
- Variational AutoEncoders (VAE) with PyTorch
- Modern PyTorch Techniques for VAEs: A Comprehensive Tutorial