Autoencoders

Introduction:

Autoencoders are a type of artificial neural network designed for unsupervised learning. They consist of an encoder network that maps the input data to a lower-dimensional representation (encoding), and a decoder network that reconstructs the original input from the encoding. The goal of autoencoders is to learn a compact representation of the input data.

Intuition:

The intuition behind autoencoders is to force the model to learn a compressed representation of the input data by minimizing the reconstruction error. The encoder learns to capture essential features, and the decoder reconstructs the input from this reduced representation. Autoencoders are used for various tasks such as dimensionality reduction, data denoising, and feature learning.

Algorithm:

1. **Encoder:**
   - The encoder network takes the input data and maps it to a lower-dimensional representation (encoding).
2. **Decoder:**
   - The decoder network takes the encoding and reconstructs the original input.
3. **Loss Function:**
   - The loss function measures the difference between the input and the reconstructed output. Common loss functions include mean squared error or binary cross-entropy.
4. **Training:**
   - The model is trained to minimize the reconstruction error by adjusting the weights in both the encoder and decoder.

Implementation in Python (MNIST Data):

Here's a simple example of implementing an autoencoder on the MNIST dataset using PyTorch.

Import libraries

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

Define the model

```python
# Define the autoencoder model
class Autoencoder(nn.Module):
    def __init__(self, encoding_dim):
        super(Autoencoder, self).__init__()
```

```
        self.encoder = nn.Sequential(
            nn.Linear(28 * 28, 512),
            nn.ReLU(),
            nn.Linear(512, 256),
            nn.ReLU(),
            nn.Linear(256, encoding_dim),
        )
        self.decoder = nn.Sequential(
            nn.Linear(encoding_dim, 256),
            nn.ReLU(),
            nn.Linear(256, 512),
            nn.ReLU(),
            nn.Linear(512, 28 * 28),
            nn.Sigmoid(),  # Sigmoid activation to ensure outputs are in [0, 1]
        )

    def forward(self, x):
        x = x.view(x.size(0), -1)  # Flatten the input
        encoded = self.encoder(x)
        decoded = self.decoder(encoded)
        return decoded
```

This code defines a simple autoencoder model using the PyTorch framework. An autoencoder is a neural network architecture designed for unsupervised learning of efficient data codings, typically used for dimensionality reduction or feature learning.

Here's a breakdown of the `Autoencoder` class:

1. **Initialization (`__init__`) Method:**
    - `encoding_dim`: The dimensionality of the latent space or encoding. This parameter determines the size of the compressed representation of the input data.
    - `encoder`: The encoder is defined as a sequential module consisting of linear layers with ReLU activation functions. It takes the flattened input (assumed to be images of size 28x28 pixels) and progressively reduces the dimensionality until reaching the specified encoding dimension.
    - `decoder`: The decoder is also defined as a sequential module. It takes the encoded representation and reconstructs the original input size. It mirrors the structure of the encoder but in reverse order, using ReLU activations for intermediate layers and a Sigmoid activation in the final layer to ensure the output values are in the range [0, 1].
2. **Forward (`forward`) Method:**
    - `x`: The input data, assumed to be images, is passed through the encoder to obtain the encoded representation (`encoded`). This representation is then passed through the decoder to reconstruct the original input (`decoded`).
    - The input is flattened (`view(x.size(0), -1)`) before being processed by the encoder. This is a common practice when working with image data.
    - The reconstructed output is returned by the forward method.

The purpose of training such an autoencoder is to learn a compressed representation of the input data in the encoding space. During training, the model aims to minimize the difference between the input and the reconstructed output. This process encourages the autoencoder to capture meaningful features in the data and generate a compact representation in the encoding space.

```
from torchsummary import summary

# Load the trained model
complex_autoencoder = Autoencoder(encoding_dim=2)
complex_autoencoder.load_state_dict(torch.load('autoencoder_model.pth'))

# Move the model to the CPU
complex_autoencoder.to('cuda:0')

# Print the summary of the encoder
summary(complex_autoencoder.encoder, (128, 1, 784))  # Assuming MNIST images (1 channel, 28x28)

# Print the summary of the decoder
summary(complex_autoencoder.decoder, (2,))  # Assuming encoding_dim is 2
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1          [-1, 128, 1, 512]         401,920
              ReLU-2          [-1, 128, 1, 512]               0
            Linear-3          [-1, 128, 1, 256]         131,328
              ReLU-4          [-1, 128, 1, 256]               0
            Linear-5            [-1, 128, 1, 2]             514
================================================================
Total params: 533,762
Trainable params: 533,762
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.38
Forward/backward pass size (MB): 1.50
Params size (MB): 2.04
Estimated Total Size (MB): 3.92
----------------------------------------------------------------
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1                   [-1, 256]             768
              ReLU-2                   [-1, 256]               0
            Linear-3                   [-1, 512]         131,584
              ReLU-4                   [-1, 512]               0
            Linear-5                   [-1, 784]         402,192
           Sigmoid-6                   [-1, 784]               0
================================================================
Total params: 534,544
Trainable params: 534,544
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.02
Params size (MB): 2.04
Estimated Total Size (MB): 2.06
----------------------------------------------------------------
```

The `Autoencoder` class in the provided code has three layers in both the encoder and the decoder. Let's break down the layers:

Encoder:

1. Linear layer with input size 28 * 28 and output size 512.
2. ReLU activation function.
3. Linear layer with input size 512 and output size 256.
4. ReLU activation function.
5. Linear layer with input size 256 and output size equal to the specified `encoding_dim`.

Decoder:

1. Linear layer with input size equal to the specified `encoding_dim` and output size 256.
2. ReLU activation function.
3. Linear layer with input size 256 and output size 512.
4. ReLU activation function.
5. Linear layer with input size 512 and output size 28 * 28.
6. Sigmoid activation function.

Therefore, both the encoder and the decoder consist of five layers. This architecture is a common choice for a basic autoencoder. The encoder progressively reduces the dimensionality of the input, while the decoder reconstructs the original input size. The ReLU activation functions introduce non-linearity, and the Sigmoid activation in the final layer of the decoder ensures that the reconstructed values are in the range [0, 1].

Training function

```
# Function to train the autoencoder
def train_autoencoder(model, dataloader, criterion, optimizer, num_epochs=10):
    for epoch in range(num_epochs):
        for data in dataloader:
            inputs, _ = data

            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, inputs.view(inputs.size(0), -1))
            loss.backward()
            optimizer.step()

        print(f'Epoch {epoch+1}/{num_epochs}, Loss: {loss.item()}')
```

Train the model

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

# Define hyperparameters
encoding_dim = 2
batch_size = 128
learning_rate = 0.001
num_epochs = 20

# Load MNIST dataset
```

```python
transform = transforms.Compose([transforms.ToTensor()])
train_dataset = datasets.MNIST(root='./data', train=True, download=True, transform=transform)
train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True)

# Initialize the autoencoder model, criterion, and optimizer
autoencoder_model = Autoencoder(encoding_dim=encoding_dim)
criterion = nn.MSELoss()  # Mean Squared Error Loss
optimizer = optim.Adam(autoencoder_model.parameters(), lr=learning_rate)


# Train the autoencoder
train_autoencoder(autoencoder_model, train_loader, criterion, optimizer, num_epochs=num_epochs)
```

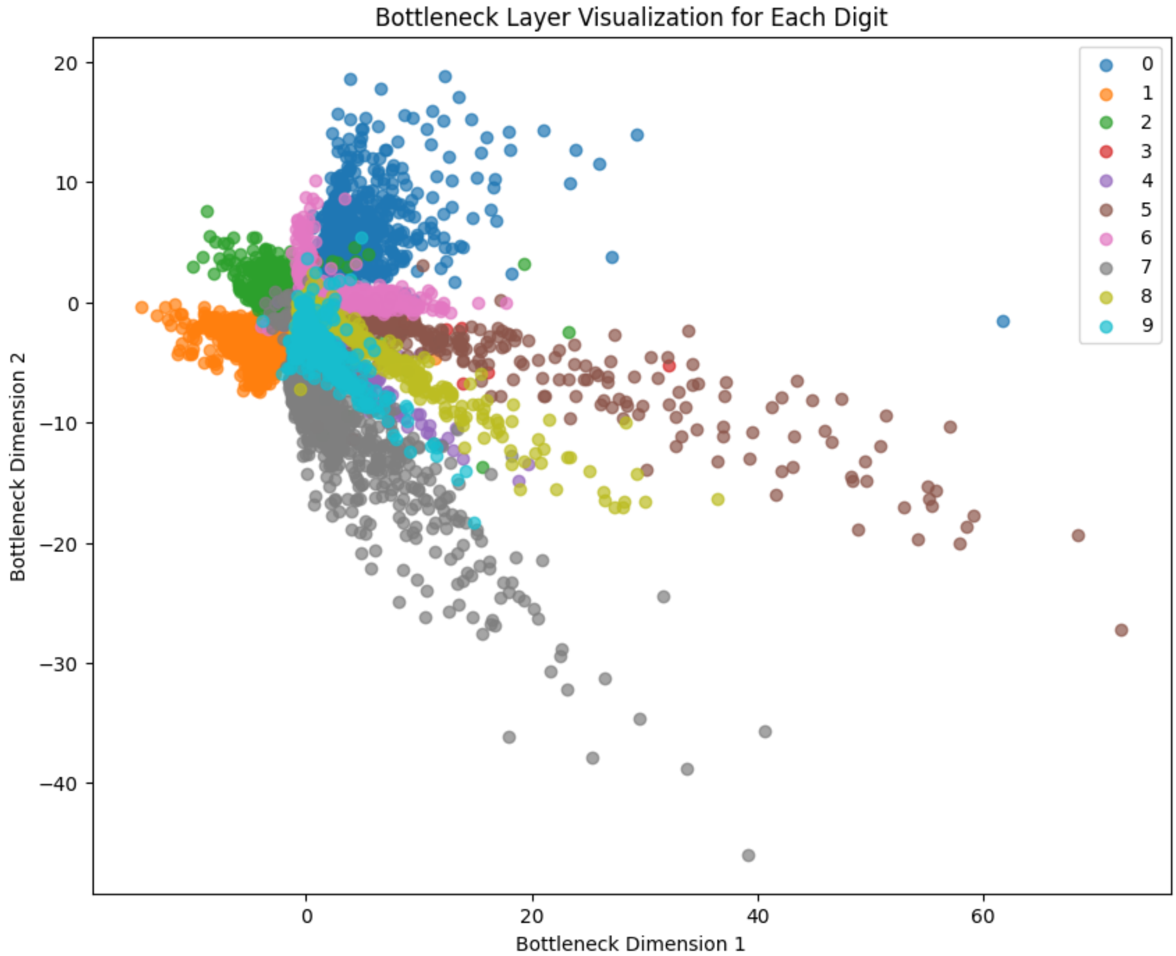
Save the model

```python
torch.save(autoencoder_model.state_dict(), 'autoencoder_model.pth')
print("Trained model saved.")
```

Visualize embeddings

```python
# Function to visualize the bottleneck layer for each digit
def visualize_bottleneck(encoder, dataloader):
    encoder.eval()
    all_embeddings = []
    all_labels = []

    with torch.no_grad():
        for data in dataloader:
            inputs, labels = data
            embeddings = encoder(inputs.view(inputs.size(0), -1)).detach().numpy()
            all_embeddings.append(torch.from_numpy(embeddings))  # Convert to PyTorch tensor
            all_labels.append(labels)

    all_embeddings = torch.cat(all_embeddings, dim=0)
    all_labels = torch.cat(all_labels, dim=0).numpy()  # Convert to NumPy array

    # Plot 2D representations, color-coded by digit label
    plt.figure(figsize=(10, 8))
    print(all_labels)
    for digit in range(10):
        digit_indices = (all_labels == digit)
        plt.scatter(all_embeddings[digit_indices, 0], all_embeddings[digit_indices, 1], label=str(digit), alpha=0.7)

    plt.title('Bottleneck Layer Visualization for Each Digit')
    plt.xlabel('Bottleneck Dimension 1')
    plt.ylabel('Bottleneck Dimension 2')
    plt.legend()
```

```
    plt.show()

# Load the trained autoencoder
autoencoder_model = Autoencoder(encoding_dim=2)  # Assuming you have a trained Autoencoder
autoencoder_model.load_state_dict(torch.load('autoencoder_model.pth'))  # Load the saved model

# Load MNIST dataset
transform = transforms.Compose([transforms.ToTensor()])
test_dataset = datasets.MNIST(root='./data', train=False, download=True, transform=transform)
test_loader = DataLoader(dataset=test_dataset, batch_size=len(test_dataset), shuffle=False)

# Visualize the bottleneck layer for each digit
visualize_bottleneck(autoencoder_model.encoder, test_loader)
```



In this example:

- The autoencoder consists of a 5 fully connected layers for both the encoder and decoder.
- The model is trained to minimize the MSE loss between the input and the reconstructed output.
- After training, the autoencoder is used to encode and decode the test set, and the results are visualized.

Feel free to adjust the architecture, hyperparameters, and visualize more samples based on your needs.