```
# Import Libraries
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.transforms as transforms
from torchvision import datasets
from torch.utils.data import DataLoader
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report
import matplotlib.pyplot as plt
```

```
# 1. Load MNIST Dataset
transform = transforms.Compose([transforms.ToTensor()])
train_data = datasets.MNIST(root='./data', train=True, download=True, transform=transform)
test_data = datasets.MNIST(root='./data', train=False, transform=transform)

train_loader = DataLoader(train_data, batch_size=128, shuffle=True)
test_loader = DataLoader(test_data, batch_size=64, shuffle=False)
```

```
100%|██████████| 9.91M/9.91M [00:00<00:00, 18.0MB/s]
100%|██████████| 28.9k/28.9k [00:00<00:00, 474kB/s]
100%|██████████| 1.65M/1.65M [00:00<00:00, 4.38MB/s]
100%|██████████| 4.54k/4.54k [00:00<00:00, 8.01MB/s]
```

```
# 2. Define Autoencoder
class Autoencoder(nn.Module):
    def __init__(self):
        super(Autoencoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Linear(28*28, 128),
            nn.ReLU(True),
            nn.Linear(128, 64),
            nn.ReLU(True),
            nn.Linear(64, 12),
            nn.ReLU(True),
            nn.Linear(12, 3)  # latent space (3D)
        )
        self.decoder = nn.Sequential(
            nn.Linear(3, 12),
            nn.ReLU(True),
            nn.Linear(12, 64),
            nn.ReLU(True),
            nn.Linear(64, 128),
            nn.ReLU(True),
            nn.Linear(128, 28*28),
            nn.Sigmoid()
        )

    def forward(self, x):
        x = x.view(-1, 28*28)
        encoded = self.encoder(x)
        decoded = self.decoder(encoded)
        return encoded, decoded
```

```
# 3. Initialize Model, Loss Function, Optimizer
model = Autoencoder()
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.005)
```

```
# 4. Train the Autoencoder
num_epochs = 10
for epoch in range(num_epochs):
    for data, _ in train_loader:
        optimizer.zero_grad()
        encoded, decoded = model(data)
        loss = criterion(decoded, data.view(-1, 28*28))
        loss.backward()
        optimizer.step()
    print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.6f}')
print("Training complete.")
```

```
Epoch [1/10], Loss: 0.044633
Epoch [2/10], Loss: 0.036677
Epoch [3/10], Loss: 0.034153
Epoch [4/10], Loss: 0.034089
```

```
Epoch [5/10], Loss: 0.034890
Epoch [6/10], Loss: 0.036899
Epoch [7/10], Loss: 0.037067
Epoch [8/10], Loss: 0.037141
Epoch [9/10], Loss: 0.034065
Epoch [10/10], Loss: 0.032605
Training complete.
```

```
# 5. Extract Compressed Features for Classification
def get_latent_features(data_loader):
    features = []
    labels = []
    with torch.no_grad():
        for data, target in data_loader:
            encoded, _ = model(data)
            features.append(encoded)
            labels.append(target)
    return torch.cat(features).numpy(), torch.cat(labels).numpy()


train_features, train_labels = get_latent_features(train_loader)
test_features, test_labels = get_latent_features(test_loader)

print(f"Compressed feature shape: {train_features.shape}")
```

```
Compressed feature shape: (60000, 3)
```

```
# 6. Train a Classifier on Encoded Features
clf = LogisticRegression(max_iter=500)
clf.fit(train_features, train_labels)
```

```
▾ LogisticRegression ⓘ ?
LogisticRegression(max_iter=500)
```

```
# 7. Predict and Generate Classification Report
pred_labels = clf.predict(test_features)
report = classification_report(test_labels, pred_labels, digits=4)
print("Classification Report:\n", report)
```

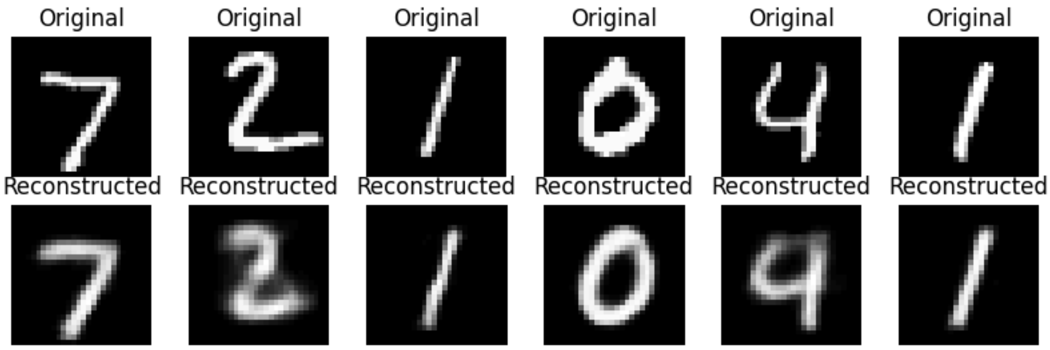
```
Classification Report:
               precision    recall  f1-score   support

           0     0.9519    0.9490    0.9504       980
           1     0.9426    0.9692    0.9557      1135
           2     0.8175    0.6986    0.7534      1032
           3     0.6288    0.5871    0.6073      1010
           4     0.6875    0.4817    0.5665       982
           5     0.6042    0.5135    0.5552       892
           6     0.8322    0.8956    0.8627       958
           7     0.8750    0.8716    0.8733      1028
           8     0.6165    0.7608    0.6811       974
           9     0.5572    0.7334    0.6333      1009

    accuracy                         0.7510     10000
   macro avg     0.7513    0.7460    0.7439     10000
weighted avg     0.7555    0.7510    0.7485     10000
```

```
# 8. Visualize Reconstruction
dataiter = iter(test_loader)
images, _ = next(dataiter)
with torch.no_grad():
    _, reconstructed = model(images)

f, axarr = plt.subplots(2, 6, figsize=(10, 3))
for i in range(6):
    axarr[0][i].imshow(images[i].view(28, 28), cmap='gray')
    axarr[0][i].set_title("Original")
    axarr[0][i].axis('off')
    axarr[1][i].imshow(reconstructed[i].view(28, 28), cmap='gray')
    axarr[1][i].set_title("Reconstructed")
    axarr[1][i].axis('off')
plt.show()
```


Original
Original
Original
Original
Original
Original
Reconstructed
Reconstructed
Reconstructed
Reconstructed
Reconstructed
Reconstructed