Experiment-8

Autoencoders for Representation Learning

Aim:
To study and implement autoencoders for unsupervised representation learning by training both a basic and a denoising autoencoder on the Fashion-MNIST dataset (mapping noisy inputs to clean outputs), and to analyze unsupervised compression and reconstruction performance through reconstruction grids and a 2-D projection of the learned latent space.

Theory:

Introduction to Autoencoders

Autoencoders are a type of neural network used for unsupervised learning of efficient data representations. Unlike supervised learning methods that require labelled data, autoencoders learn useful features by attempting to reconstruct their input. The main idea is to compress the input into a lower-dimensional representation and then reconstruct the original input from this compressed form using an encoder–decoder architecture.

“High-dimensional data can be converted to low-dimensional codes by training a multilayer neural network with a small central layer to reconstruct high-dimensional input vectors."
- Hinton & Salakhutdinov, Science, 2006

An autoencoder consists of two main parts:

- **Encoder**: Compresses the input into a latent-space representation (bottleneck layer)
- **Decoder**: Reconstructs the input from the latent representation

  The network is trained to minimise the difference between the input and its reconstruction, forcing it to learn the most important features of the data.

Types of Autoencoders:

Basic Autoencoder:

A basic autoencoder learns to compress and reconstruct clean input data. The input image is passed through the encoder, compressed into a bottleneck (latent) representation, and then reconstructed by the decoder.
The bottleneck layer forces the network to learn a compressed representation that captures the essential features of the input while discarding redundant information.

Denoising Autoencoder:

“A denoising autoencoder is trained to reconstruct a clean 'repaired' input from a corrupted version of it."
- Vincent et al., ICML 2008

A denoising autoencoder is trained to reconstruct a clean “repaired” input from a corrupted version of it. In this case, noise is deliberately added to the input image, and the corrupted image is then fed into the autoencoder. The decoder attempts to reconstruct the original clean image rather than the noisy input.

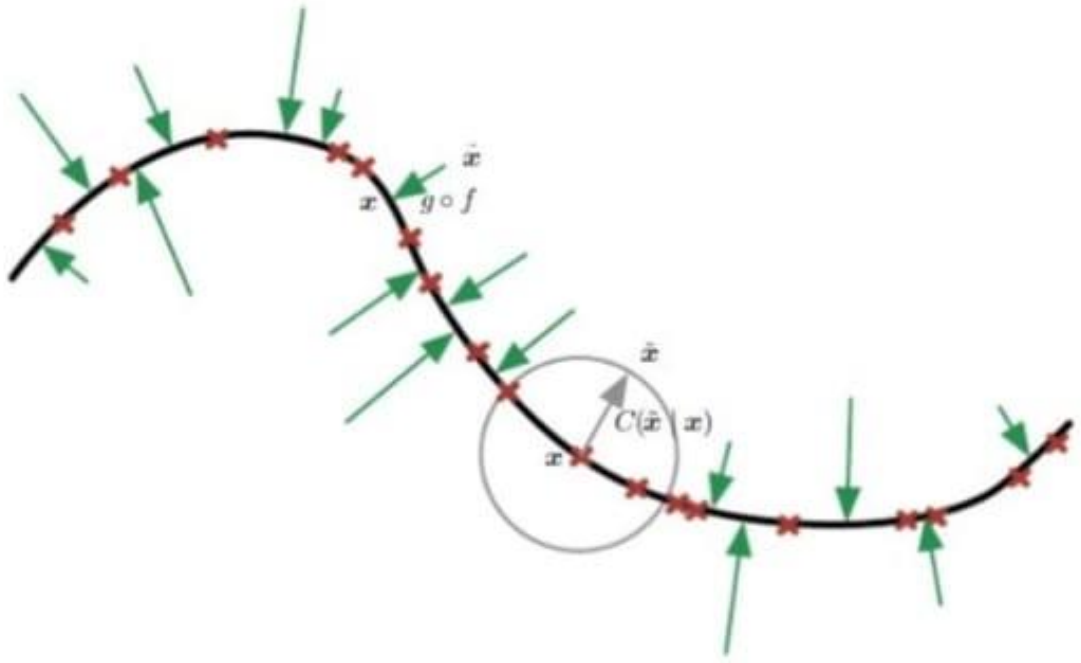


Figure 1- Denoising autoencoder

(Source: Deep Learning. Ian Goodfellow, Yoshua Bengio, and Aaron Courville, MIT Press.)

A denoising autoencoder is trained to map a corrupted data point x' back to the original data point x as shown in Figure 1. We illustrate training examples x as red crosses lying near a low-dimensional manifold illustrated with the bold black line. We illustrate the corruption process C (x' | x) with a gray circle of equiprobable corruption. A gray arrow demonstrates how one training example is transformed into one sample from this corruption process.

The training process for a denoising autoencoder can be written as:

- Input: $x' = x + n \cdot x$ where **n** is random noise
- Target: x (clean image)
- Loss: $L\left(x,\ g\left(f\left(x'\right)\right)\right)$

The reconstruction loss is computed by comparing the decoder's output with the original clean image. This forces the network to learn features that are resilient to noise and capture the underlying structure of the data.

Latent Space Representation:

Latent space (bottleneck layer) is the compressed representation learned by the encoder. This compression enables autoencoders to learn hierarchical representations of data.

For visualisation purposes, a 2-dimensional latent space is often used. When the latent dimension is 2, the encoded representations of input images can be directly plotted to observe how the autoencoder organises different patterns in the data.

Similar fashion items tend to cluster together in the learned latent space, indicating that the autoencoder has learned meaningful and discriminative representations.

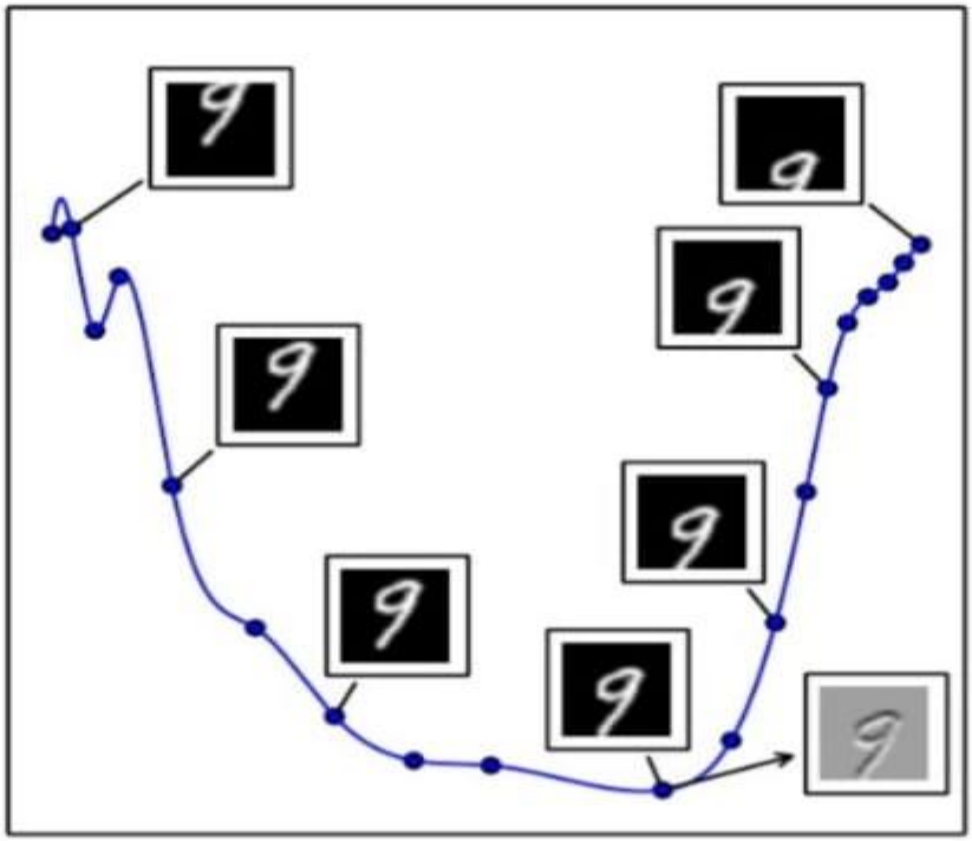


Figure 2- Reconstructed Image After Noise Removal

(Source: Deep Learning. Ian Goodfellow, Yoshua Bengio, and Aaron Courville, MIT Press.)

The amount of vertical translation defines a coordinate along a one-dimensional manifold that traces a curved path through image space. The above plot as shown in Figure 2 shows a few points along this manifold. For visualization, we have projected the manifold into two-dimensional space using PCA. An n-dimensional manifold has an n-dimensional tangent plane at every point. This tangent plane touches the manifold exactly at that point and is oriented parallel to the surface at that point.

Merits of Autoencoders

- **Unsupervised Learning**: No labelled data required for training
- **Dimensionality Reduction**: Learns compact representations of high-dimensional data
- **Noise Reduction**: Denoising autoencoders can remove noise from corrupted data

- **Feature Learning**: Automatically discovers useful features without manual engineering
- **Data Compression**: Can be used for efficient data storage and transmission

Demerits of Autoencoders

- **Reconstruction Quality**: May not perfectly reconstruct complex images
- **Training Complexity**: Requires careful tuning of architecture and hyperparameters
- **Computational Cost**: Deep autoencoders require significant training time
- **Task-Specific**: Representations learned may not transfer well to other tasks

Code & Result:

[ ]:
```
# This Python 3 environment comes with many helpful analytics libraries installed
# It is defined by the kaggle/python Docker image: https://github.com/kaggle/docker-python
# For example, here's several helpful packages to load

import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)

# Input data files are available in the read-only "../input/" directory
# For example, running this (by clicking run or pressing Shift+Enter) will list all files under the input directory

import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname,  filename))

 # You can write up to 20GB to the current directory (/kaggle/working/) that gets preserved as output when you create a version using "Save & Run All"
# You can also write temporary files to /kaggle/temp/, but they won't be saved outside of the current session
```

```
/kaggle/input/fashionmnist/t10k-labels-idx1-ubyte
/kaggle/input/fashionmnist/t10k-images-idx3-ubyte
/kaggle/input/fashionmnist/fashion-mnist_test.csv
/kaggle/input/fashionmnist/fashion-mnist_train.csv
/kaggle/input/fashionmnist/train-labels-idx1-ubyte
/kaggle/input/fashionmnist/train-images-idx3-ubyte
```

1 IMPORT LIBRARIES

[ ]:
```
# ================================================================================
# IMPORT LIBRARIES
# ================================================================================
import torch
import torch.nn as nn
```

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

2 ADD DATA

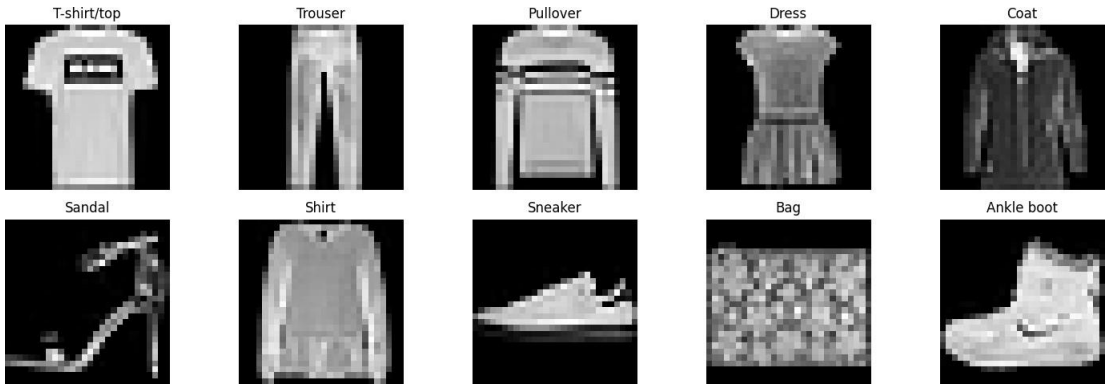
[ ]:
```
import os
import shutil

base = "/kaggle/working/FashionMNIST/raw"
os.makedirs(base, exist_ok=True)

src = "/kaggle/input/fashionmnist"

files = [
    "train-images-idx3-ubyte",
    "train-labels-idx1-ubyte",
    "t10k-images-idx3-ubyte",
    "t10k-labels-idx1-ubyte"
]

for f in files:
    shutil.copy(os.path.join(src, f), os.path.join(base, f))

print("Files copied successfully")
```

```
Files copied successfully
```

3 SHOW DATA

[ ]:
```
import matplotlib.pyplot as plt
import numpy as np

# Define class names for FashionMNIST
classes = [
    'T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
    'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot'
]

def display_one_from_each_class(dataset):
    found_classes = {}
    idx = 0

    # Loop until we find one of each (0-9)
```

```
    while len(found_classes) < 10:
        img, label = dataset[idx]
        if label not in found_classes:
            found_classes[label] = img
        idx += 1

    # Plotting
    plt.figure(figsize=(15, 5))
    for label, img in sorted(found_classes.items()):
        plt.subplot(2, 5, label + 1)
        # Convert tensor to numpy and remove channel dim for grayscale
        plt.imshow(img.squeeze(),    cmap='gray')
        plt.title(classes[label])
        plt.axis('off')

    plt.tight_layout()
    plt.show()

# Execute the function
display_one_from_each_class(train_data)
```



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

transform = transforms.ToTensor()

train_data = datasets.FashionMNIST(
    root="/kaggle/working",
    train=True,
    download=False,
    transform=transform
)
```

```
test_data = datasets.FashionMNIST(
    root="/kaggle/working",
    train=False,
    download=False,
    transform=transform
)

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

4 AUTOENCODER MODEL

[ ]:
```
# ============================================================================
# Architecture: Deeper network with BatchNorm, Dropout, and 2D latent space
# Progression: 784 -> 512 -> 256 -> 128 -> 64 -> 32 -> 16 -> 8 -> 4 -> 2
# ============================================================================
class Autoencoder(nn.Module):
    def __init__(self):
        super().__init__()

        # Deeper encoder: Compressing from 784 down to 2
        self.encoder = nn.Sequential(
            nn.Flatten(),
            nn.Linear(784, 512),
            nn.BatchNorm1d(512),
            nn.ReLU(),
            nn.Dropout(0.1),

            nn.Linear(512, 256),
            nn.BatchNorm1d(256),
            nn.ReLU(),

            nn.Linear(256, 128),
            nn.BatchNorm1d(128),
            nn.ReLU(),

            nn.Linear(128, 64),
            nn.BatchNorm1d(64),
            nn.ReLU(),

            nn.Linear(64, 32),
            nn.BatchNorm1d(32),
            nn.ReLU(),

            nn.Linear(32, 16),
            nn.BatchNorm1d(16),
```

```
            nn.ReLU(),

            nn.Linear(16, 8),
            nn.BatchNorm1d(8),
            nn.ReLU(),

            nn.Linear(8, 4),
            nn.BatchNorm1d(4),
            nn.ReLU(),

            nn.Linear(4, 2)  # Latent space (2D for visualization)
        )

        # Deeper decoder: Reconstructing from 2 back to 784
        self.decoder = nn.Sequential(
            nn.Linear(2, 4),
            nn.BatchNorm1d(4),
            nn.ReLU(),

            nn.Linear(4, 8),
            nn.BatchNorm1d(8),
            nn.ReLU(),

            nn.Linear(8, 16),
            nn.BatchNorm1d(16),
            nn.ReLU(),

            nn.Linear(16, 32),
            nn.BatchNorm1d(32),
            nn.ReLU(),

            nn.Linear(32, 64),
            nn.BatchNorm1d(64),
            nn.ReLU(),

            nn.Linear(64, 128),
            nn.BatchNorm1d(128),
            nn.ReLU(),

            nn.Linear(128, 256),
            nn.BatchNorm1d(256),
            nn.ReLU(),

            nn.Linear(256, 512),
            nn.BatchNorm1d(512),
            nn.ReLU(),
```

```
            nn.Linear(512, 784),
            nn.Sigmoid() # Use Sigmoid for pixel values between [0, 1]
        )

    def forward(self, x):
        z = self.encoder(x)
        out = self.decoder(z)
        # Reshape output back to image dimensions if necessary for your loss function
        # out = out.view(-1, 1, 28, 28)
        return out, z
```

5 TRAINING SETUP

[ ]:
```
# ================================================================================
# SETUP TRAINING CONFIGURATION
# Initialize model, loss functions, optimizer, and learning rate scheduler
# ================================================================================
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = Autoencoder().to(device)

# Use MSE + L1 loss for better detail preservation
criterion_mse = nn.MSELoss()
criterion_l1 = nn.L1Loss()

# Improved optimizer with scheduler
optimizer = torch.optim.AdamW(model.parameters(), lr=0.001, weight_decay=1e-5)
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
    optimizer, mode='min', factor=0.5, patience=5
)
```

6 NOISE ADDITION FUNCTION

[ ]:
```
# ================================================================================
# NOISE ADDITION FUNCTION
# Adds Gaussian noise to images for denoising task
# Reduced noise factor (0.25 vs 0.4) for better reconstruction quality
# ================================================================================
def add_noise(x, noise_factor=0.25):  # Reduced from 0.4 to 0.25
    noisy = x + noise_factor * torch.randn_like(x)
    return torch.clamp(noisy, 0., 1.)
```

7 TRAINING

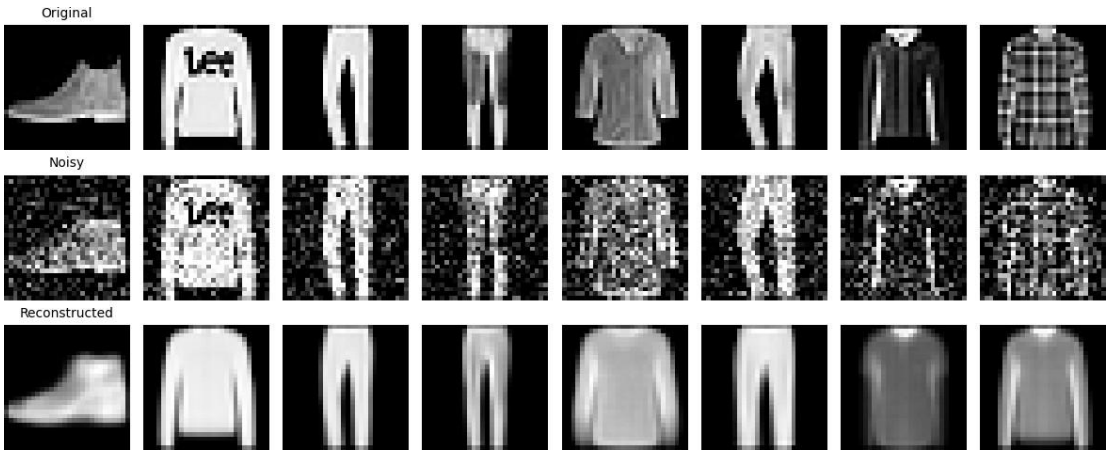
[ ]:
```
# ============================================================================
# TRAINING LOOP
# Train for 100 epochs with combined MSE+L1 loss, gradient clipping, and
# automatic best model checkpointing
# ============================================================================
epochs = 100  # Increased epochs
best_loss = float('inf')

print("Starting training...")
for epoch in range(epochs):
    model.train()
    total_loss = 0

    for images, _ in train_loader:
        images = images.to(device)
        noisy = add_noise(images)

        outputs, _ = model(noisy)

        # Combined loss for better quality
        loss_mse = criterion_mse(outputs, images.view(images.size(0), -1))
        loss_l1 = criterion_l1(outputs, images.view(images.size(0), -1))
        loss = 0.7 * loss_mse + 0.3 * loss_l1

        optimizer.zero_grad()
        loss.backward()

        # Gradient clipping for stability
        torch.nn.utils.clip_grad_norm_(model.parameters(),   max_norm=1.0)o

        optimizer.step()
        total_loss += loss.item()

    avg_loss = total_loss / len(train_loader)
    scheduler.step(avg_loss)

    # Save best model
    if avg_loss < best_loss:
        best_loss = avg_loss
        torch.save(model.state_dict(), 'best_autoencoder.pth')

    if (epoch + 1) % 5 == 0:
        print(f"Epoch [{epoch+1}/{epochs}] Loss: {avg_loss:.4f}")

print(f"\nTraining completed! Best Loss: {best_loss:.4f}")
```

```
Starting training...
Epoch [5/100] Loss: 0.0542
Epoch [10/100] Loss: 0.0520
Epoch [15/100] Loss: 0.0510
Epoch [20/100] Loss: 0.0487
Epoch [25/100] Loss: 0.0477
Epoch [30/100] Loss: 0.0496
Epoch [35/100] Loss: 0.0470
Epoch [40/100] Loss: 0.0482
Epoch [45/100] Loss: 0.0467
Epoch [50/100] Loss: 0.0461
Epoch [55/100] Loss: 0.0458
Epoch [60/100] Loss: 0.0457
Epoch [65/100] Loss: 0.0457
Epoch [70/100] Loss: 0.0448
Epoch [75/100] Loss: 0.0450
Epoch [80/100] Loss: 0.0448
Epoch [85/100] Loss: 0.0447
Epoch [90/100] Loss: 0.0445
Epoch [95/100] Loss: 0.0446
Epoch [100/100] Loss: 0.0445

Training completed! Best Loss: 0.0445
```

8 VISUALIZATION 1: BASIC RECONSTRUCTION

[ ]:
```python
# ============================================================================
# VISUALIZATION 1: BASIC RECONSTRUCTION
# Display original, noisy, and reconstructed images side-by-side
# Shows 8 sample images from test set
# ============================================================================
model.eval()
images, _ = next(iter(test_loader))
images = images.to(device)
noisy = add_noise(images)

with torch.no_grad():
    reconstructed, _ = model(noisy)

n = 8
plt.figure(figsize=(12, 5))
for i in range(n):
    # Original
    plt.subplot(3, n, i+1)
    plt.imshow(images[i].cpu().squeeze(), cmap='gray')
    plt.axis('off')
    if i == 0:
```

```
        plt.title('Original',    fontsize=10)

    # Noisy
    plt.subplot(3, n, i+1+n)
    plt.imshow(noisy[i].cpu().squeeze(), cmap='gray')
    plt.axis('off')
    if i == 0:
        plt.title('Noisy',   fontsize=10)

    # Reconstructed
    plt.subplot(3, n, i+1+2*n)
    plt.imshow(reconstructed[i].view(28,28).cpu(), cmap='gray')
    plt.axis('off')
    if i == 0:
        plt.title('Reconstructed',  fontsize=10)

plt.tight_layout()
plt.savefig('improved_reconstruction.png', dpi=150, bbox_inches='tight')
plt.show()

print("\nReconstruction visualization saved as 'improved.png'")
```

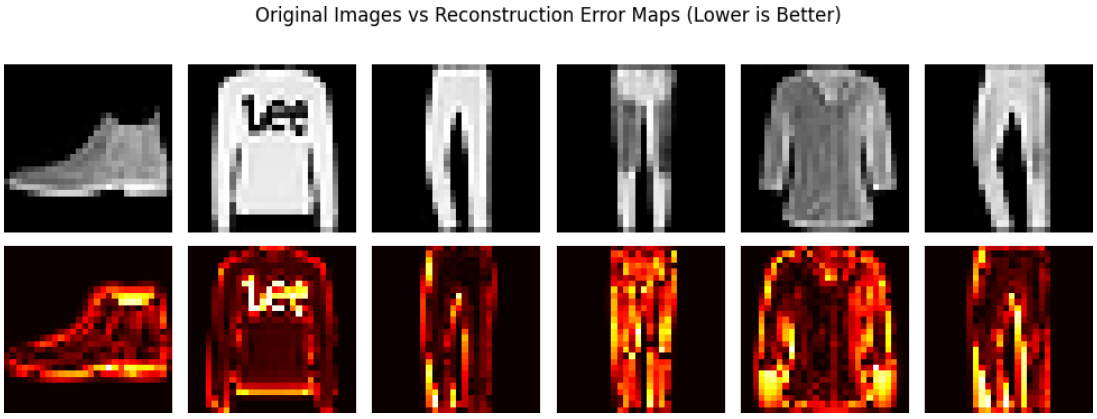



```
Reconstruction visualization saved as 'improved.png'
```

9 VISUALIZATION 2: ERROR MAPS

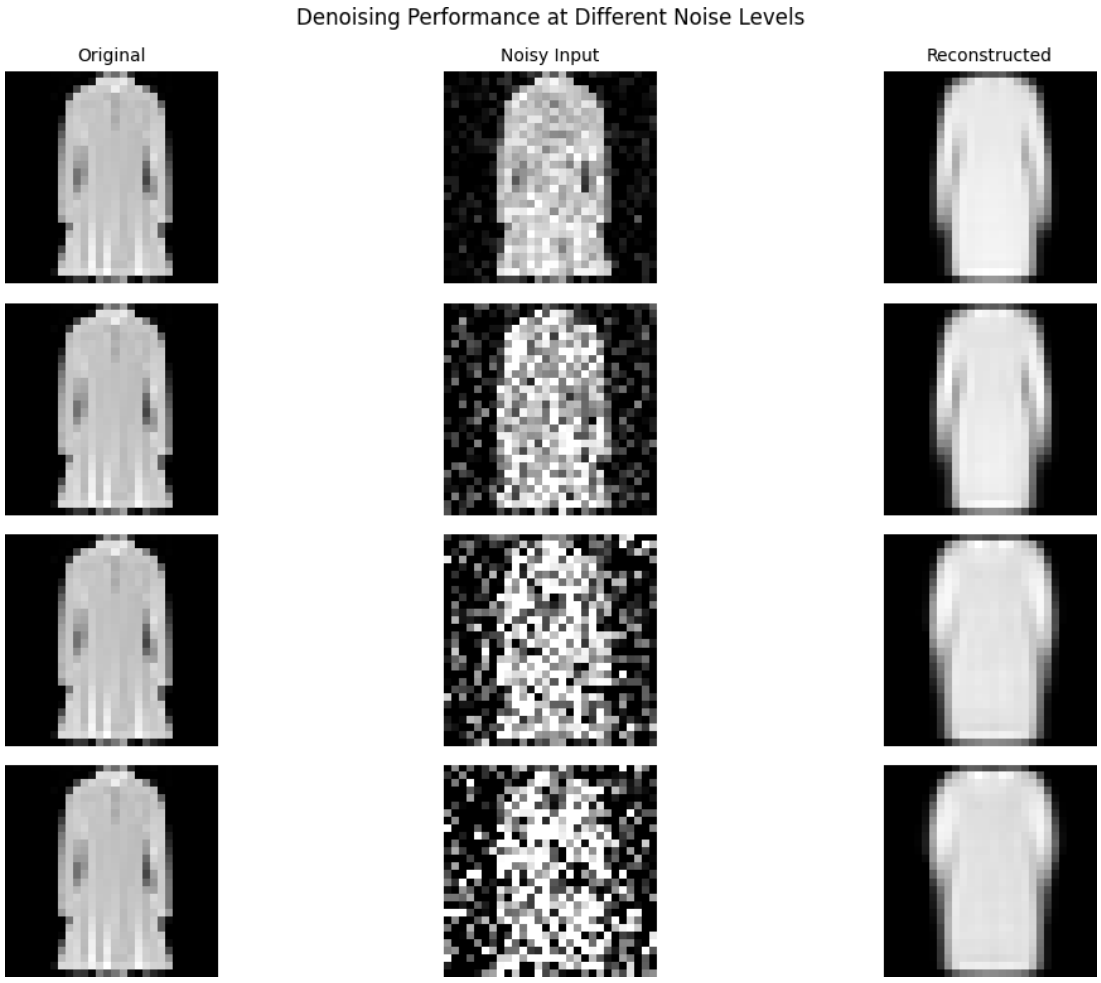
[ ]:
```
# ============================================================================
# VISUALIZATION 2: ERROR MAPS
# Heat maps showing pixel-wise reconstruction errors
# Brighter areas = higher error, Darker areas = better reconstruction
```

```python
# ============================================================================
model.eval()
images, _ = next(iter(test_loader))
images = images.to(device)
noisy = add_noise(images)

with torch.no_grad():
    reconstructed, _ = model(noisy)

reconstructed = reconstructed.view(-1, 1, 28, 28)
error = torch.abs(images - reconstructed)

plt.figure(figsize=(10, 4))
for i in range(6):
    # Original
    plt.subplot(2, 6, i+1)
    plt.imshow(images[i].cpu().squeeze(), cmap='gray')
    plt.axis('off')
    if i == 0:
        plt.ylabel('Original', fontsize=10)

    # Error map
    plt.subplot(2, 6, i+7)
    plt.imshow(error[i].cpu().squeeze(), cmap='hot')
    plt.axis('off')
    if i == 0:
        plt.ylabel('Error Map', fontsize=10)

plt.suptitle("Original Images vs Reconstruction Error Maps (Lower is Better)",
  fontsize=12)
plt.tight_layout()
plt.savefig('error_maps.png', dpi=150, bbox_inches='tight')
plt.show()
```



10 VISUALIZATION 3: NOISE ROBUSTNESS TEST

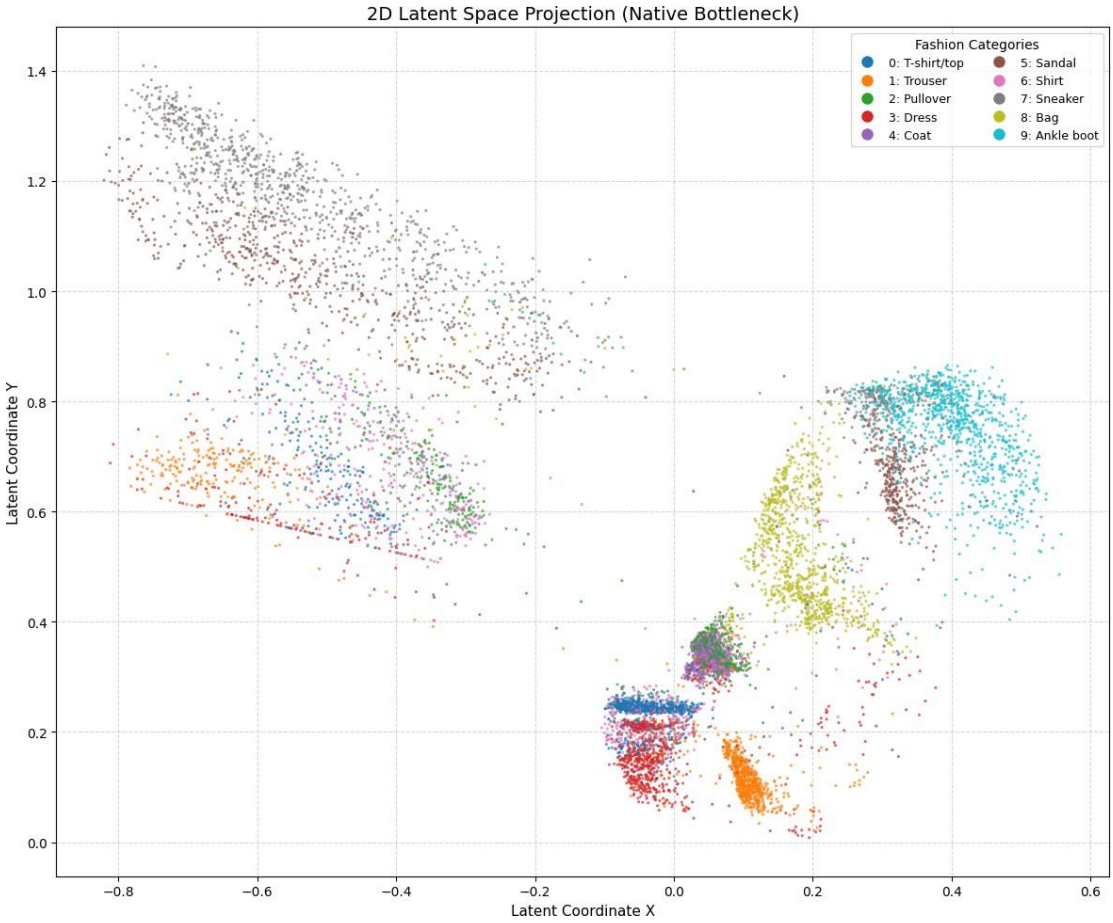
[ ]:
```
# ============================================================================
# VISUALIZATION 3: NOISE ROBUSTNESS TEST
# Tests model performance at different noise levels (0.1 to 0.6)
# Shows how well the model handles varying amounts of noise
# ============================================================================
import random
img_idx = random.randint(0, images.size(0) - 1)
noise_levels = [0.1, 0.25, 0.4, 0.6]

plt.figure(figsize=(12, 8))
for i, nf in enumerate(noise_levels):
    noisy_test = add_noise(images, nf)
    with torch.no_grad():
        recon, _ = model(noisy_test)

    # Original
    plt.subplot(len(noise_levels), 3, i*3 + 1)
    plt.imshow(images[img_idx].cpu().squeeze(), cmap='gray')
    plt.axis('off')
    if i == 0:
        plt.title('Original', fontsize=10)
    plt.ylabel(f'Noise={nf}', fontsize=9)

    # Noisy
    plt.subplot(len(noise_levels), 3, i*3 + 2)
    plt.imshow(noisy_test[img_idx].cpu().squeeze(), cmap='gray')
    plt.axis('off')
    if i == 0:
        plt.title('Noisy Input', fontsize=10)

    # Reconstructed
    plt.subplot(len(noise_levels), 3, i*3 + 3)
    plt.imshow(recon[img_idx].view(28,28).cpu(), cmap='gray')
    plt.axis('off')
    if i == 0:
        plt.title('Reconstructed', fontsize=10)

plt.suptitle('Denoising Performance at Different Noise Levels', fontsize=12)
plt.tight_layout()
plt.savefig('noise_comparison.png', dpi=150, bbox_inches='tight')
plt.show()
```



11 VISUALIZATION 4: LATENT SPACE PROJECTION

[ ]:
```
# ============================================================================
# VISUALIZATION 4: LATENT SPACE PROJECTION
# 2D visualization of the NATIVE 2D latent space showing class clusters
# Each color represents a different FashionMNIST class
# ============================================================================
model.eval()
latents = []
labels = []

with torch.no_grad():
    for images, lbls in test_loader:
        images = images.to(device)
        # Your model returns: (reconstruction, latent_z)
```

```
            _, z = model(images)
            latents.append(z.cpu())
            labels.append(lbls)

# Concatenate all batches into single arrays
latents = torch.cat(latents).numpy()
labels = torch.cat(labels).numpy()

from matplotlib.lines import Line2D

class_names = [
    "T-shirt/top", "Trouser", "Pullover", "Dress", "Coat",
    "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"
]

plt.figure(figsize=(12, 10))
# Since latents is already (N, 2), we use index 0 and 1 directly
scatter = plt.scatter(
    latents[:, 0],
    latents[:, 1],
    c=labels,
    cmap='tab10',
    s=5,           # Increased size slightly for better visibility
    alpha=0.7,     # Adjusted alpha for overlap
    edgecolors='none'
)

plt.xlabel("Latent Coordinate X", fontsize=11)
plt.ylabel("Latent Coordinate Y", fontsize=11)
plt.title("2D Latent Space Projection (Native Bottleneck)", fontsize=14)

# Create custom legend
legend_elements = [
    Line2D([0], [0], marker='o', color='w',
           label=f"{i}: {class_names[i]}",
           markerfacecolor=scatter.cmap(scatter.norm(i)),
           markersize=10)
    for i in range(10)
]

plt.legend(handles=legend_elements, title="Fashion Categories",
           loc="upper right", fontsize=9, ncol=2, frameon=True)

plt.grid(True, linestyle='--', alpha=0.5)
plt.tight_layout()
plt.savefig('latent_space_2d.png', dpi=150, bbox_inches='tight')
plt.show()
```



[ ]:
```
# ================================================================================
# QUANTITATIVE EVALUATION
# ================================================================================
model.eval()
total_mse = total_ssim = total_psnr = total_samples = 0

def calculate_ssim(img1, img2):
    C1, C2 = 0.01 ** 2, 0.03 ** 2
    mu1, mu2 = img1.mean(), img2.mean()
    sigma1_sq = ((img1 - mu1) ** 2).mean()
    sigma2_sq = ((img2 - mu2) ** 2).mean()
    sigma12 = ((img1 - mu1) * (img2 - mu2)).mean()
    return (((2 * mu1 * mu2 + C1) * (2 * sigma12 + C2)) /
            ((mu1 ** 2 + mu2 ** 2 + C1) * (sigma1_sq + sigma2_sq + C2))).item()

with torch.no_grad():
    for images, _ in test_loader:
        images = images.to(device)
```

```
        noisy = add_noise(images)
        outputs, _ = model(noisy)
        outputs = outputs.view_as(images)    # Reshape to match images

        mse = ((outputs - images) ** 2).mean(dim=[1, 2, 3])
        total_mse += mse.sum().item()
        total_psnr += sum(20 * torch.log10(1.0 / torch.sqrt(m)) for m in mse).
  item()
        total_ssim += sum(calculate_ssim(images[i].squeeze(), outputs[i].
  squeeze())
                          for i in range(images.size(0)))
        total_samples += images.size(0)

print(f"\nMSE: {total_mse/total_samples:.6f} | PSNR: {total_psnr/total_samples:.
  2f} dB | SSIM: {total_ssim/total_samples:.4f}")
```

MSE: 0.026800 | PSNR: 16.33 dB | SSIM: 0.8210