Experiment-8

Autoencoders for Representation Learning

Aim:

To implement and study a basic autoencoder and a denoising autoencoder for unsupervised Representation learning on the Fashion-MNIST dataset. The basic autoencoder reconstructs clean Images from clean inputs, while the denoising autoencoder reconstructs clean images from noisy inputs.

The experiment also compares their compression, reconstruction quality, robustness to noise, and Learned 2-D latent representations.

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

The model is trained to minimize reconstruction error, so it learns to preserve the most important information while discarding unnecessary detail. This makes autoencoders useful for dimensionality reduction, compression, denoising, and feature learning.

An autoencoder consists of two main parts:

- **Encoder:** converts the input into a latent-space representation
- **Decoder:** reconstructs the input from the latent representation

Types of Autoencoders:

Basic Autoencoder:

A basic autoencoder is trained on clean input images. The input passes through the encoder, which compresses it into a latent vector. The decoder then reconstructs the original image from that vector. The bottleneck layer forces the model to learn a compressed representation that captures the main structure of the input while discarding redundant information.

This type of autoencoder is useful when the goal is to learn compact features from clean data.

Denoising Autoencoder:

A denoising autoencoder is trained to reconstruct a clean image from a corrupted version of the same image. If the clean input is represented as x, then the noisy input can be written as:

$$\tilde{x} = x + n$$

where n is random noise. The noisy image $\tilde{x}$ is given as input to the model, but the target output remains the original clean image x. The model learns to remove the noise and recover the underlying structure of the image.

This helps the autoencoder learn features that are robust to corruption and useful for practical reconstruction tasks.

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

- **Input :** $\tilde{x}$ (noisy image)
- **Target :** x (clean image)
- **Loss :** $L\left(x, g\left(f(\tilde{x})\right)\right)$

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.

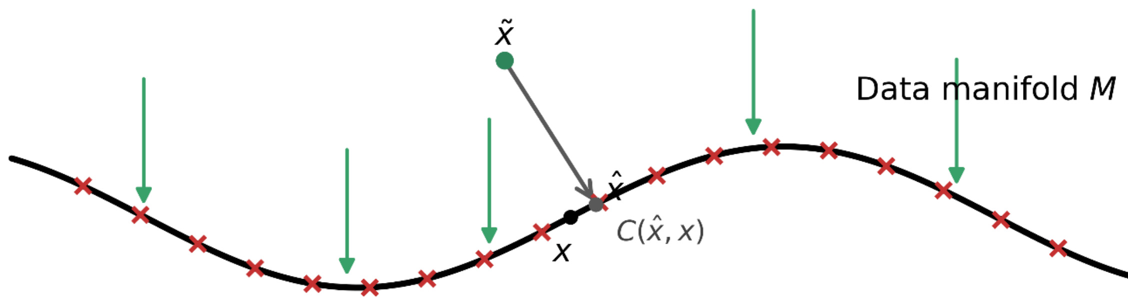


Figure 1 Geometric interpretation of a denoising autoencoder

This figure illustrates the manifold-learning interpretation of a denoising autoencoder (DAE). The black curve, denoted as the data manifold M, represents the low-dimensional structure on which clean data samples naturally lie. The red crosses correspond to clean training samples x distributed along this manifold. During training, a clean sample x is artificially corrupted to produce a noisy version $\tilde{x}$ (green point), which lies away from the manifold. The denoising autoencoder processes this corrupted input through an encoder–decoder mapping and produces a reconstruction $\hat{x}$ (gray point) that is pushed back toward the manifold and close to the original clean sample. The reconstruction error C($\hat{x}$, x) measures the discrepancy between the reconstructed sample and the true clean sample and is minimized during training. The green arrows along the manifold represent the learned denoising vector field, indicating that the autoencoder learns to move corrupted inputs toward regions of high data density on the manifold. Consequently, the DAE not only reconstructs clean samples from noisy observations but also learns the underlying geometric structure of the data distribution.

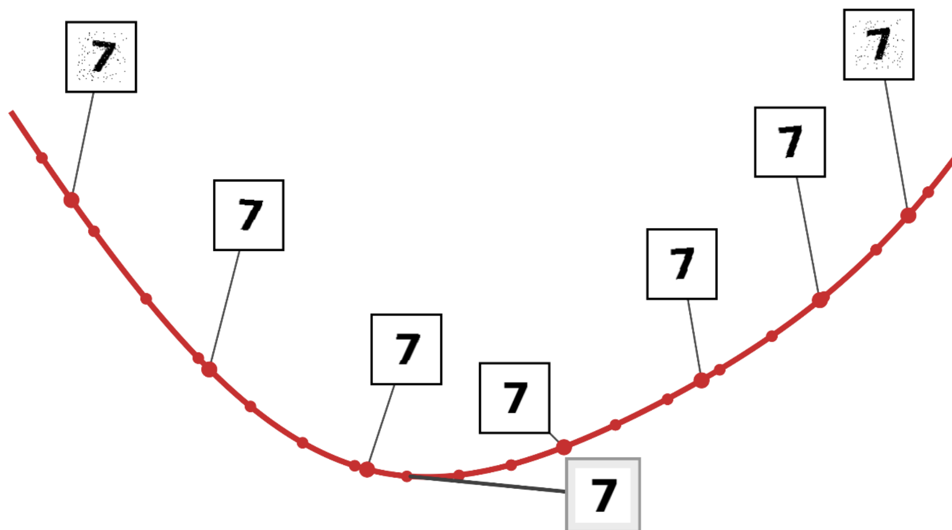


Figure 2 Denoising autoencoder reconstruction on the learned manifold of handwritten digit 7

Figure 2 illustrates the learned latent manifold of handwritten digit 7 representations in a denoising autoencoder. The red curve represents a low-dimensional manifold in latent space, where each point corresponds to a latent code whose decoded image is shown in the associated box. The smoother and cleaner digit images near the centre indicate regions of higher data density, while the noisier images toward the ends represent less representative latent states. The arrow depicts the denoising process, where corrupted representations are guided toward the manifold, resulting in a cleaner reconstructed digit that preserves the essential characteristics of the original input.

Latent Space Representation:

The latent space (bottleneck layer) is the compressed representation learned by the encoder. It is the

The latent space (bottleneck layer) is the compressed representation learned by the encoder. It is the stacked layers in the encoder its depth that enable autoencoders to learn hierarchical representations of data, where early layers capture low-level features (such as edges and textures) and deeper layers capture increasingly abstract patterns. The bottleneck itself plays a separate but equally important role: by constraining the dimensionality, it forces the network to retain only the most essential information and discard redundancy.

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 as a scatter plot 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.

Fashion-MNIST Dataset

Fashion-MNIST is a dataset of grayscale images representing 10 different fashion categories. Each image is 28×28 pixels. The 10 classes are:

1. T-shirt/top
2. Trouser
3. Pullover
4. Dress
5. Coat
6. Sandal
7. Shirt
8. Sneaker
9. Bag
10. Ankle boot

This dataset is commonly used for testing machine learning algorithms because it is more challenging than standard MNIST digits while maintaining the same image format.

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:

[9]:
```
# 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/datasets/organizations/zalando-research/fashionmnist/t10k-labels-idx1-ubyte
/kaggle/input/datasets/organizations/zalando-research/fashionmnist/t10k-images-idx3-ubyte
/kaggle/input/datasets/organizations/zalando-research/fashionmnist/fashion-mnist_test.csv
/kaggle/input/datasets/organizations/zalando-research/fashionmnist/fashion-mnist_train.csv
/kaggle/input/datasets/organizations/zalando-research/fashionmnist/train-labels-idx1-ubyte
/kaggle/input/datasets/organizations/zalando-research/fashionmnist/train-images-idx3-ubyte
```

Step 1: Import Libraries

[12]:
```
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
```

Step 2: Add dataset

[13]:
```
import os
import shutil


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

src = "/kaggle/input/datasets/organizations/zalando-research/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
```

Step 3 :Load Dataset

[20]:
```
from torchvision import datasets, transforms
from torch.utils.data import DataLoader


transform = transforms.ToTensor()

train_data = datasets.FashionMNIST(
```

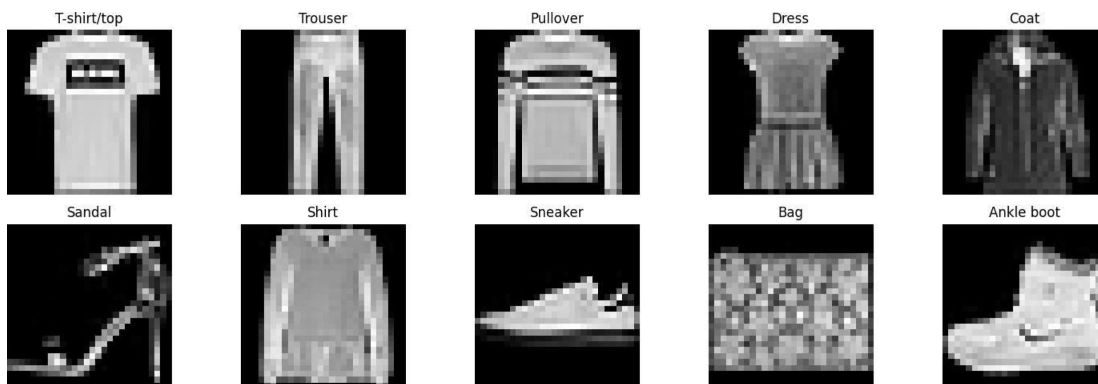
```
    root="/kaggle/working",
    train=True,
    download=False,
    transform=transform
)

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)

print("Dataset  loaded  successfully!")
print(f"Training samples: {len(train_data)} | Test samples: {len(test_data)}")
```

```
Dataset loaded successfully!
Training samples: 60000 | Test samples: 10000
```

Step 4: Show Dataset Sample

[21]:
```
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)
```



Step 5 : Model Architecture

[22]:
```
# 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

print("Model architecture defined successfully!")
```

Model architecture defined successfully!

Step 6: Basic AE Training

[23]:

```
# BASIC AUTOENCODER TRAINING
# Trains on clean images only — no noise added (clean → clean)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
basic_model = Autoencoder().to(device)

criterion_mse = nn.MSELoss()
criterion_l1 = nn.L1Loss()
optimizer_basic = optim.Adam(basic_model.parameters(), lr=0.001)
scheduler_basic = optim.lr_scheduler.ReduceLROnPlateau(optimizer_basic,
 ↪mode='min', patience=5)

epochs = 100
```

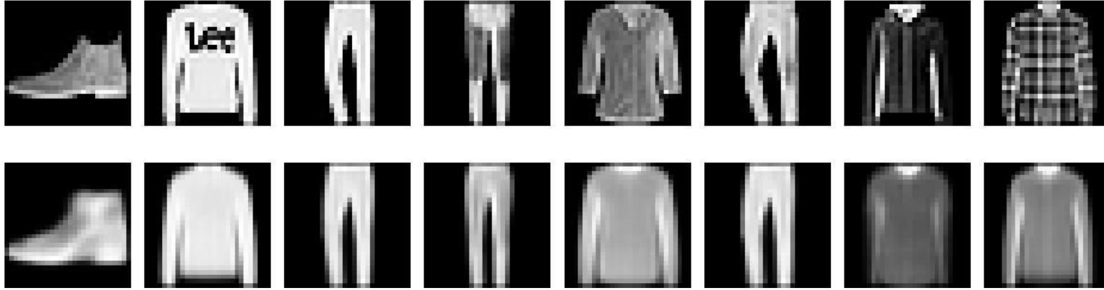
```python
best_loss_basic = float('inf')

for epoch in range(epochs):
    basic_model.train()
    total_loss = 0

    for images, _ in train_loader:
        images = images.to(device)
        outputs, _ = basic_model(images)

        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_basic.zero_grad()
        loss.backward()
        torch.nn.utils.clip_grad_norm_(basic_model.parameters(), max_norm=1.0)
        optimizer_basic.step()
        total_loss += loss.item()

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

    if avg_loss < best_loss_basic:
        best_loss_basic = avg_loss

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

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

```
Epoch [5/100] Loss: 0.0536
Epoch [10/100] Loss: 0.0507
Epoch [15/100] Loss: 0.0506
Epoch [20/100] Loss: 0.0499
Epoch [25/100] Loss: 0.0464
Epoch [30/100] Loss: 0.0458
Epoch [35/100] Loss: 0.0455
Epoch [40/100] Loss: 0.0451
Epoch [45/100] Loss: 0.0450
Epoch [50/100] Loss: 0.0449
Epoch [55/100] Loss: 0.0447
Epoch [60/100] Loss: 0.0445
Epoch [65/100] Loss: 0.0445
Epoch [70/100] Loss: 0.0446
Epoch [75/100] Loss: 0.0445
Epoch [80/100] Loss: 0.0445
```

```
Epoch [85/100] Loss: 0.0445
Epoch [90/100] Loss: 0.0445
Epoch [95/100] Loss: 0.0444
Epoch [100/100] Loss: 0.0444

Training completed! Best Loss: 0.0444
```

Step 7: Basic AE Reconstruction

[24]:
```
# VISUALIZATION: BASIC RECONSTRUCTION
# Display original and reconstructed images side-by-side (no noise — clean
↪input only)

basic_model.eval()
images, _ = next(iter(test_loader))
images = images.to(device)

with torch.no_grad():
    reconstructed_basic, _ = basic_model(images)

# Display 8 samples
n = 8
plt.figure(figsize=(12, 4))

for i in range(n):
    # Original
    plt.subplot(2, n, i+1)
    plt.imshow(images[i].cpu().squeeze(), cmap='gray')
    plt.axis('off')

    # Reconstructed
    plt.subplot(2, n, i+1+n)
    plt.imshow(reconstructed_basic[i].view(28,28).cpu(), cmap='gray')
    plt.axis('off')

plt.tight_layout()
plt.show()

print("Basic reconstruction visualization completed!")
```



Basic reconstruction visualization completed!

Step 8: Denoising AE Training

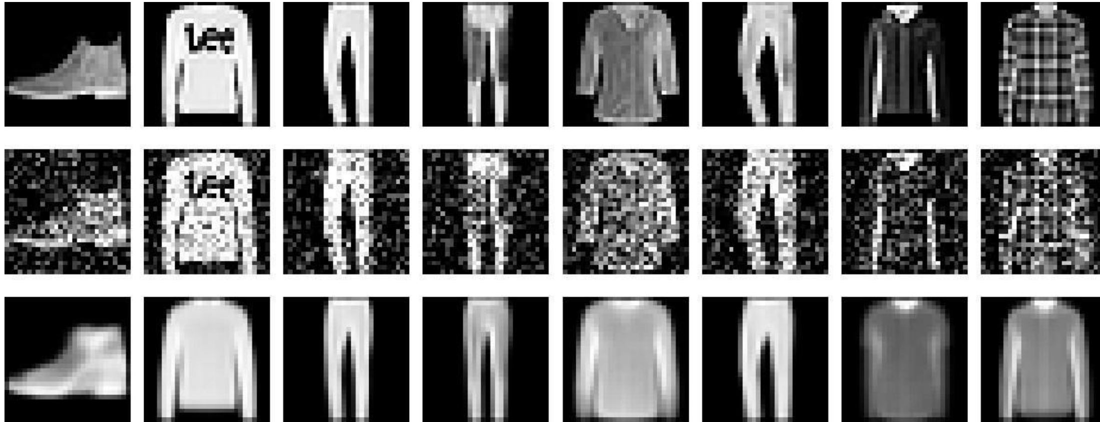
[25]:
```
# DENOISING AUTOENCODER TRAINING
# Uses combined MSE + L1 loss with gradient clipping
# Trains on noisy images, reconstructs clean ones (noisy → clean)

def add_noise(x, noise_factor=0.25):
    noisy = x + noise_factor * torch.randn_like(x)
    return torch.clamp(noisy, 0., 1.)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = Autoencoder().to(device)

criterion_mse = nn.MSELoss()
criterion_l1 = nn.L1Loss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min',
 patience=5)

epochs = 100
best_loss = float('inf')

print("Starting denoising autoencoder 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)
```

```
        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()
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
        optimizer.step()

        total_loss += loss.item()

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

    if avg_loss < best_loss:
        best_loss = avg_loss

    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 denoising autoencoder training...
Epoch [5/100] Loss: 0.0534
Epoch [10/100] Loss: 0.0535
Epoch [15/100] Loss: 0.0485
Epoch [20/100] Loss: 0.0488
Epoch [25/100] Loss: 0.0468
Epoch [30/100] Loss: 0.0476
Epoch [35/100] Loss: 0.0461
Epoch [40/100] Loss: 0.0459
Epoch [45/100] Loss: 0.0455
Epoch [50/100] Loss: 0.0452
Epoch [55/100] Loss: 0.0450
Epoch [60/100] Loss: 0.0448
Epoch [65/100] Loss: 0.0446
Epoch [70/100] Loss: 0.0445
Epoch [75/100] Loss: 0.0444
Epoch [80/100] Loss: 0.0444
Epoch [85/100] Loss: 0.0441
Epoch [90/100] Loss: 0.0441
Epoch [95/100] Loss: 0.0442

Epoch [100/100] Loss: 0.0439

Training completed! Best Loss: 0.0439
```

Step 9: Denoising Reconstruction

[26]:
```
# VISUALIZATION: BASIC RECONSTRUCTION
# Display original, noisy, and reconstructed images side-by-side

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

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

# Display 8 samples
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')

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

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

plt.tight_layout()
plt.show()

print("Reconstruction visualization completed!")
```

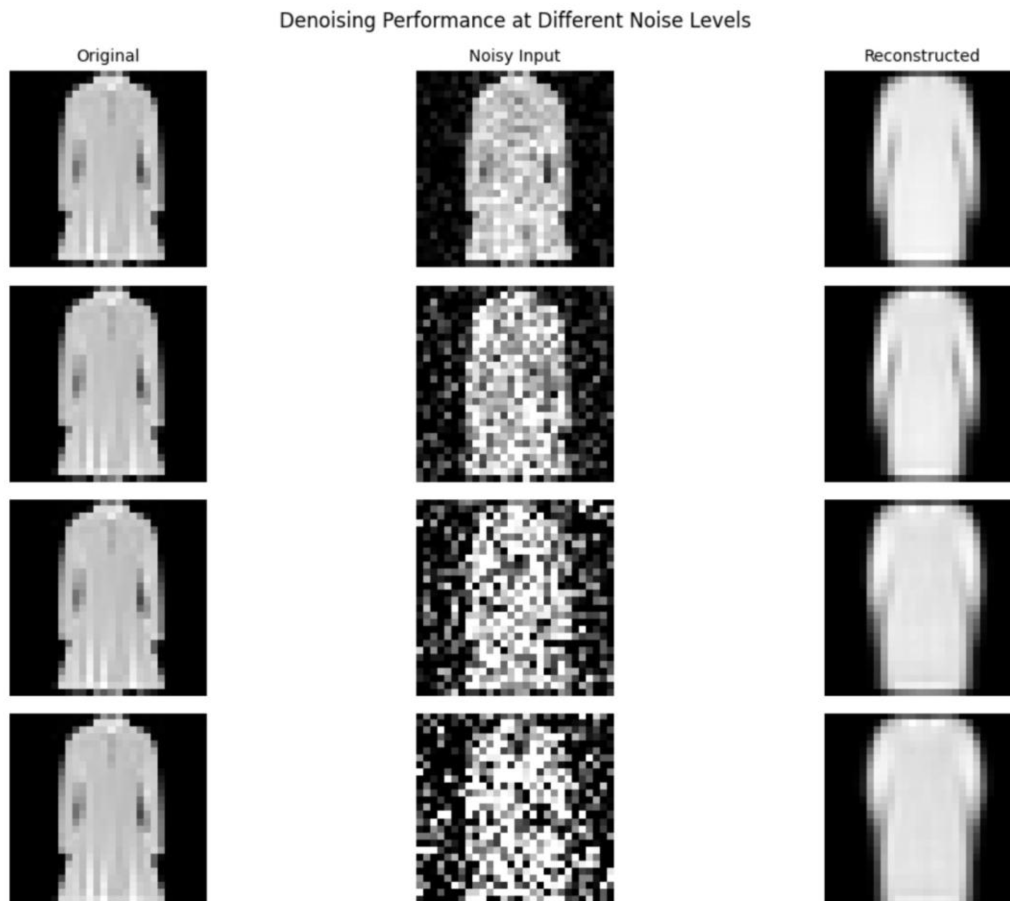


Reconstruction visualization completed!

Step 10: Noise Robustness

[27]:
```
# ============================================================================
# VISUALIZATION : 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)
```

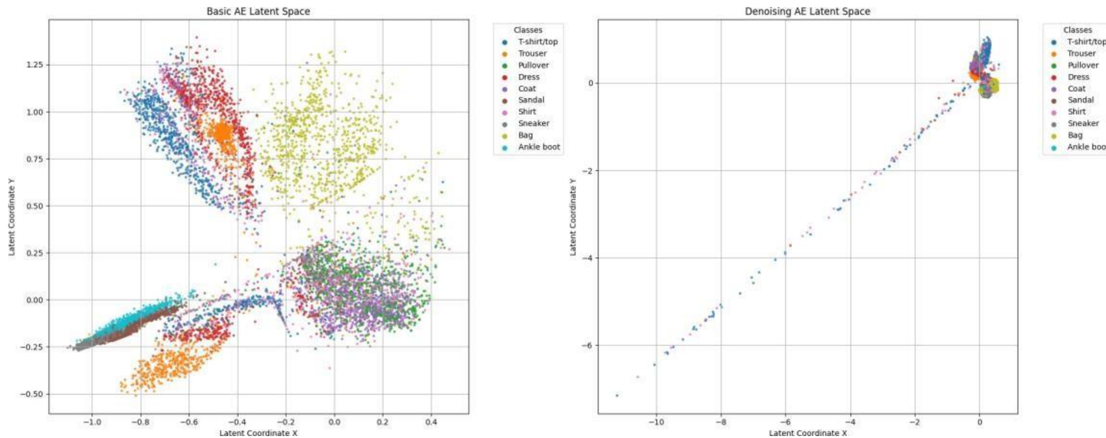
```
        plt.imshow(noisy_test[img_idx].cpu().squeeze(), cmap='gray')
        plt.axis('off')
        if i == 0:
            plt.title('Noisy Input', fontsize=10)

        # 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()

print("Noise robustness test completed!")
```



Noise robustness test completed!

Step 11: Latent Space

[33]:
```
def get_latents(mdl, loader):
    latents, labels = [], []
    with torch.no_grad():
        for images, lbls in loader:
            images = images.to(device)
            _, z = mdl(images)
            latents.append(z.cpu())
            labels.append(lbls)
    return torch.cat(latents).numpy(), torch.cat(labels).numpy()

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

basic_latents, labels = get_latents(basic_model, test_loader)
denoise_latents, _ = get_latents(model, test_loader)

fig, axes = plt.subplots(1, 2, figsize=(20, 8))

for ax, latents, title in zip(axes,
                              [basic_latents, denoise_latents],
                              ["Basic AE Latent Space", "Denoising AE Latent␣
↪Space"]):
    scatter = ax.scatter(latents[:, 0], latents[:, 1], c=labels, cmap='tab10',␣
↪s=5, alpha=0.7)
    handles = [ax.scatter([], [], c=[scatter.cmap(scatter.norm(i))],␣
↪label=class_names[i], s=30)
               for i in range(len(class_names))]
    ax.legend(handles=handles, title="Classes", bbox_to_anchor=(1.05, 1),␣
↪loc='upper left')
    ax.set_xlabel("Latent Coordinate X")
    ax.set_ylabel("Latent Coordinate Y")
    ax.set_title(title)
    ax.grid(True)

plt.tight_layout()
plt.show()

print("Latent space visualization completed!")
```



Latent space visualization completed!

Step 12: Quantitative Evaluation

[34]:
```
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()

def evaluate_model(mdl, loader, use_noise=False):
    mdl.eval()
    total_mse = total_ssim = total_psnr = total_samples = 0

    with torch.no_grad():
        for images, _ in loader:
            images = images.to(device)
            inputs = add_noise(images) if use_noise else images
            outputs, _ = mdl(inputs)
            outputs = outputs.view_as(images)

            for i in range(images.size(0)):
                mse = ((outputs[i] - images[i]) ** 2).mean().item()
                total_mse += mse
                # Safe PSNR: clamp MSE to avoid log(0)
                total_psnr += 20 * np.log10(1.0 / np.sqrt(max(mse, 1e-10)))
                total_ssim += calculate_ssim(images[i].squeeze(), outputs[i].
↪squeeze())
```

```python
                total_samples += 1

    return {
        "MSE": total_mse / total_samples,
        "PSNR": total_psnr / total_samples,
        "SSIM": total_ssim / total_samples
    }

# Evaluate both models
basic_metrics = evaluate_model(basic_model, test_loader, use_noise=False)
denoise_metrics = evaluate_model(model, test_loader, use_noise=True)

print("Quantitative Evaluation Results:")
print(f"{'Metric':<10} {'Basic AE':>12} {'Denoising AE':>14}")
print("-" * 38)
for metric in ["MSE", "PSNR", "SSIM"]:
    unit = " dB" if metric == "PSNR" else ""
    print(f"{metric:<10}{basic_metrics[metric]:>11.4f}{unit}
    ↪{denoise_metrics[metric]:>11.4f}{unit}")
```

```
Quantitative Evaluation Results:
Metric          Basic AE    Denoising AE
--------------------------------------
MSE              0.0269        0.0263
PSNR            16.3590 dB     16.4855 dB
SSIM             0.8225        0.8244
```