Autoencoders

Aims:

1. Understanding autoencoders
2. Understanding convolutional autoencoders
3. Understanding variational autoencoders

What is an Autoencoder?

A type of neural network architecture designed to efficiently compress (encode) input data down to its essential features, then reconstruct (decode) the original input from this compressed representation.

- During training, the autoencoder learns which latent variables *-hidden or random variables that inform the way data is distributed-* can be used to most accurately reconstruct the original data

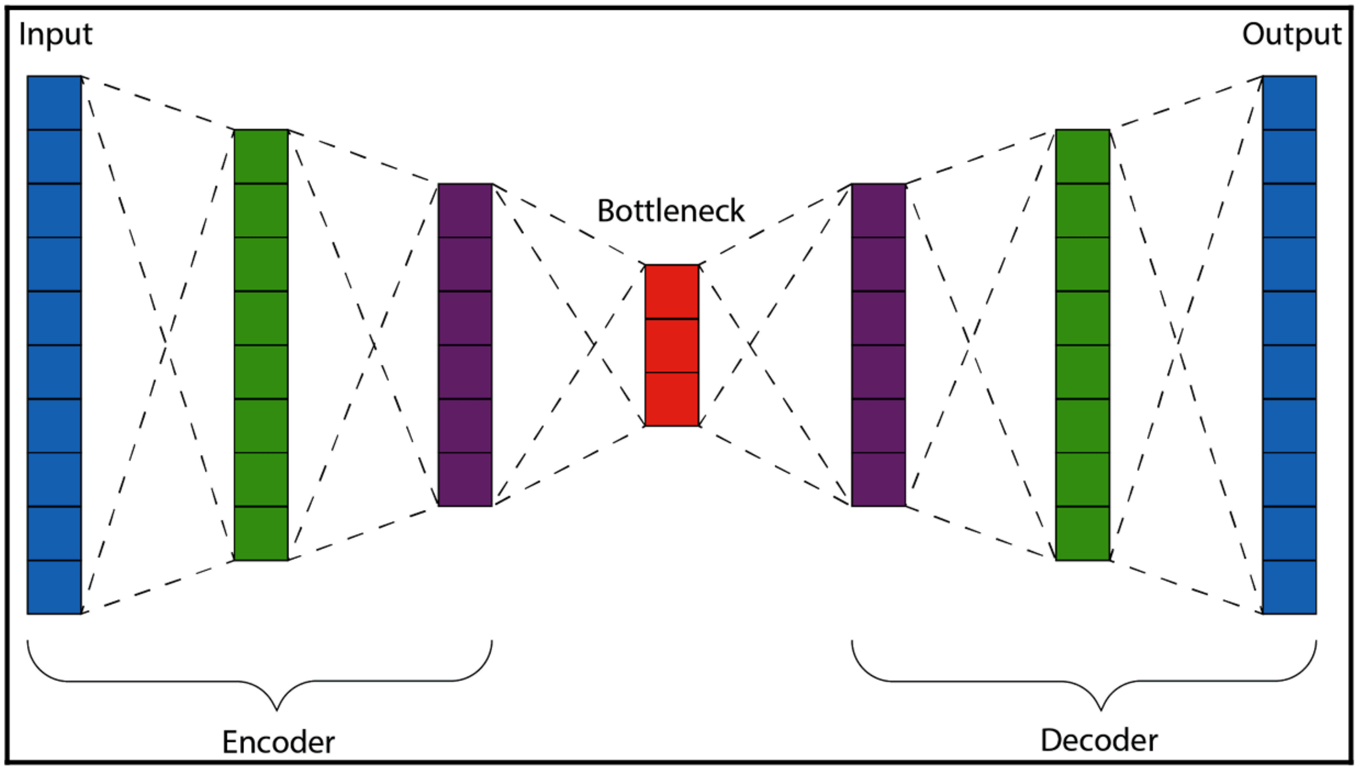
Why Autoencoders?

- **Unsupervised Learning**
- **Dimensionality Reduction**
- **Denoising (Cleaning Up Messy Data)**
- **Anomaly Detection (Finding the "Odd One Out")**

How does it Work?

1. The **encoder** comprises layers that encode a compressed representation of the input data through dimensionality reduction.
2. The **bottleneck** (or “code”) contains the most compressed representation of the input/output in the network.
3. The **decoder** comprises hidden layers with a progressively larger number of nodes that decompress (or decode) the encoded representation of data, ultimately reconstructing the data back to its original form.

- Output is then compared to the ground truth. The difference between the output and ground truth is called the reconstruction error.



What is a Convolutional Autoencoder (CAE)?

A **Convolutional Autoencoder (CAE)** is a type of unsupervised neural network that is specifically designed for **image data**. Unlike traditional autoencoders that use fully connected layers, CAEs use **convolutional layers** to better capture the spatial structure of images.

The goal of a CAE is to:

- Learn a **compressed representation** (called the *latent vector*) of an input image,
- And then **reconstruct the image** from this representation as accurately as possible.

Applications:

- Image compression
- Denoising
- Feature extraction
- Visual data clustering

> The encoder compresses the image into a low-dimensional bottleneck representation, while the decoder reconstructs the original image from this

> compressed form.

CAE Architecture

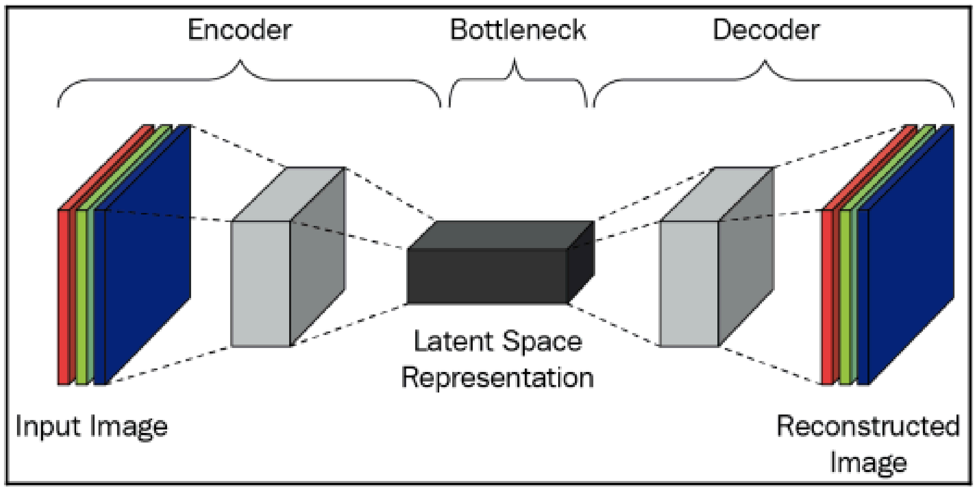
Encoder

- **Conv2D + ReLU:** Extracts spatial features from the image
- **MaxPooling:** Reduces spatial dimensions while preserving important features
- Repeated to reduce the image gradually
- Final output: **latent vector (bottleneck)**

Decoder

- **Upsampling or ConvTranspose2D:** Increases the spatial size of the image
- **Conv2D + ReLU:** Reconstructs image features layer by layer
- Final layer: **Tanh** or **Sigmoid** to produce the output image

> The bottleneck is a compressed summary of the original image.



Mathematical Representation

The CAE learns to minimize the difference between input image x and the reconstructed image $\hat{x}$.

Encoding:

$$z = f_{encoder}(x)$$

- `f_encoder` includes the convolution + pooling layers
- `z` is the latent vector (compressed image)

Decoding:

$$\hat{x} = f_{decoder}(z)$$

- `f_decoder` includes upsampling + convolution to reconstruct the image

---

Loss Function:

$$\mathcal{L}(x, \hat{x}) = \|x - \hat{x}\|^2$$

- Loss is calculated using Mean Squared Error (MSE)
- Lower loss = better reconstruction

Visualizing the Latent Space using t-SNE

To understand the structure of the latent space, we can project high-dimensional embeddings into a **2D plane** using **t-SNE**.

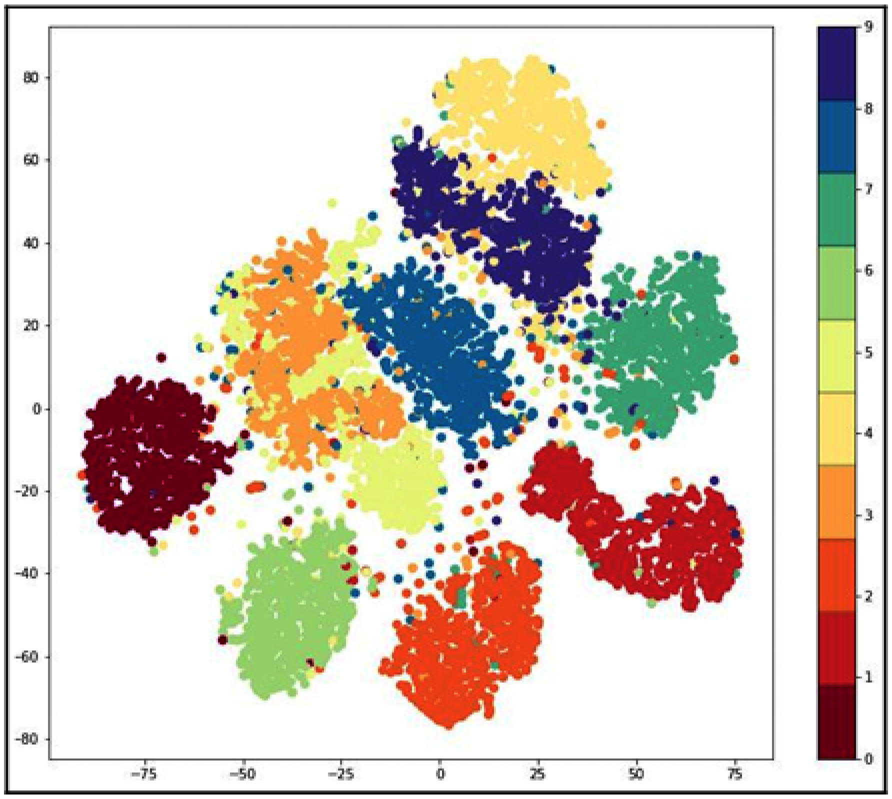
Why t-SNE?

- Latent vectors (64-d) are too large to visualize
- **t-SNE** reduces them to 2D while preserving structure
- Similar images → plotted close together
- Dissimilar images → plotted far apart

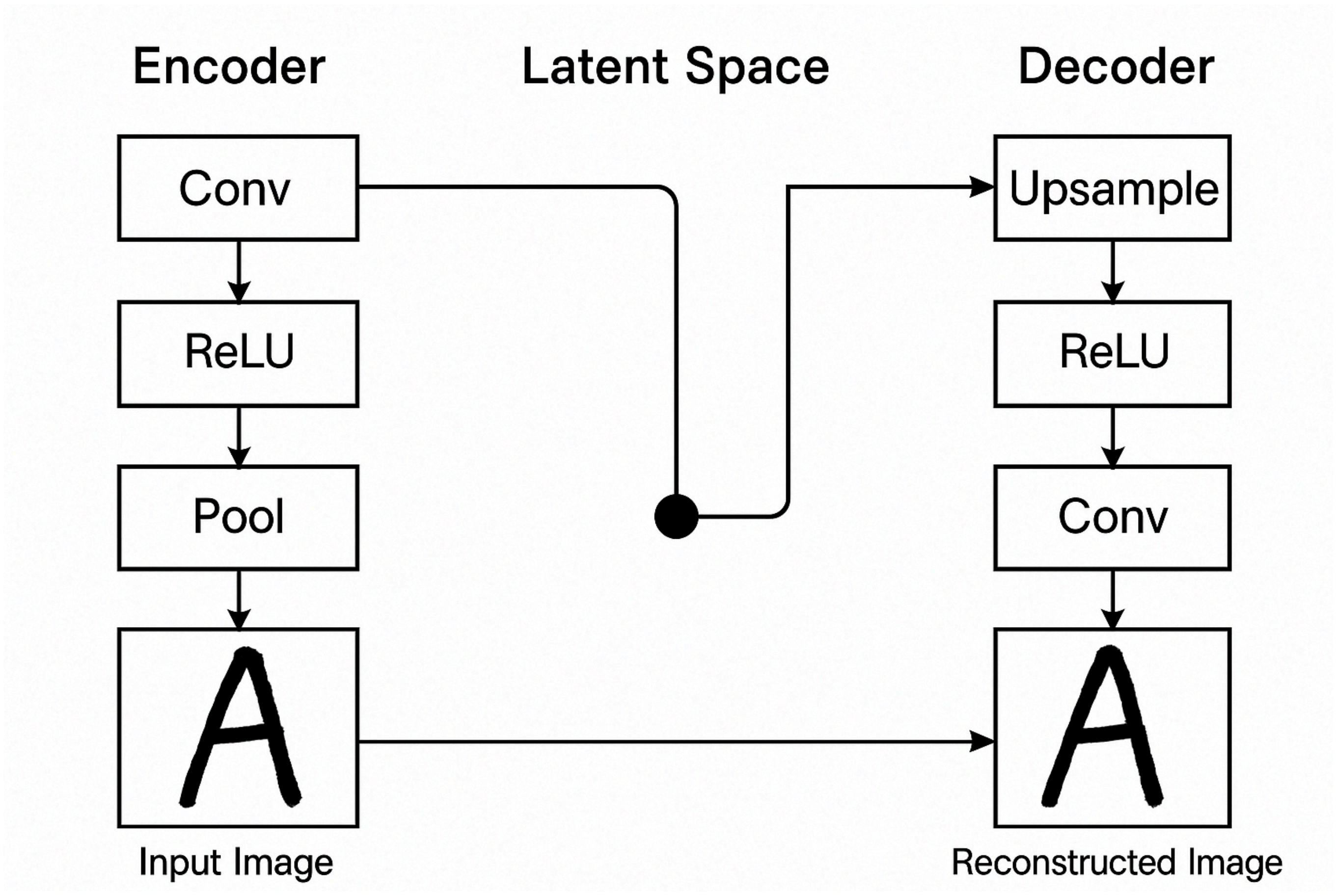
Goal:

- Visualize clustering of similar image embeddings
- Validate that the encoder captures image similarity effectively

> Example: All embeddings of the letter "A" form one cluster, while "Z" appears in another.



80
60
40
20
0
-20
-40
-60
-80
-75
-50
-25
0
25
50
75
0
1
2
3
4
5
6
7
8
9



Encoder
Latent Space
Decoder
Conv
ReLU
Pool
Upsample
ReLU
Conv
Input Image
Reconstructed Image

Key Takeaways A Convolutional Autoencoder (CAE) uses convolutional layers to effectively compress and reconstruct image data.

The encoder extracts essential features and reduces dimensionality, resulting in a compact latent representation.

The decoder rebuilds the image from the latent vector using upsampling and convolutional operations.

The latent space can be visualized using techniques like t-SNE, which helps reveal image similarity and feature clustering.

- CAEs are powerful tools for unsupervised feature learning, image reconstruction, and visual data understanding.

Variational Autoencoder (VAE)

What is a Variational Autoencoder (VAE)?

A Variational Autoencoder (VAE) is a type of neural network that builds upon the standard Autoencoder. Its purpose goes beyond just compressing data (e.g., images) and reconstructing it—it also enables the generation of new, realistic samples.

For instance, if we input a handwritten digit like "7" into a standard Autoencoder, it compresses it into a fixed latent vector and reconstructs it accurately. However, it struggles to generate new images effectively. Sampling a random point from its latent space might produce a blurry or nonsensical image because the latent space is not structured.

VAE addresses this by adopting a probabilistic approach. Instead of outputting a fixed latent vector, it learns a probability distribution (typically Gaussian) in the latent space. This makes the latent space continuous and organized, allowing for the generation of new, realistic images by sampling from the distribution.

Main Differences Between VAE and Standard Autoencoder

Here’s a comparison table highlighting the key differences:

Aspect	Standard Autoencoder	Variational Autoencoder (VAE)
Goal	Compress data and reconstruct it accurately	Compress data, reconstruct it, and generate new realistic
Latent Space	Fixed latent vector, possibly disorganized	Probability distribution (mean and standard deviation), org
Generation	Inefficient at generating new data	Efficient at generating new data via sampling

Aspect	Standard Autoencoder	Variational Autoencoder (VAE)
Encoder Output	Produces a fixed latent vector	Produces two vectors: mean and standard deviation
Loss Function	Reconstruction loss (e.g., MSE)	Reconstruction loss + KL Divergence

Why VAE Solves the Shortcomings of Standard Autoencoders?

VAE (Variational Autoencoder) addresses the limitations of standard autoencoders by imposing structure on the latent space. In a standard autoencoder:

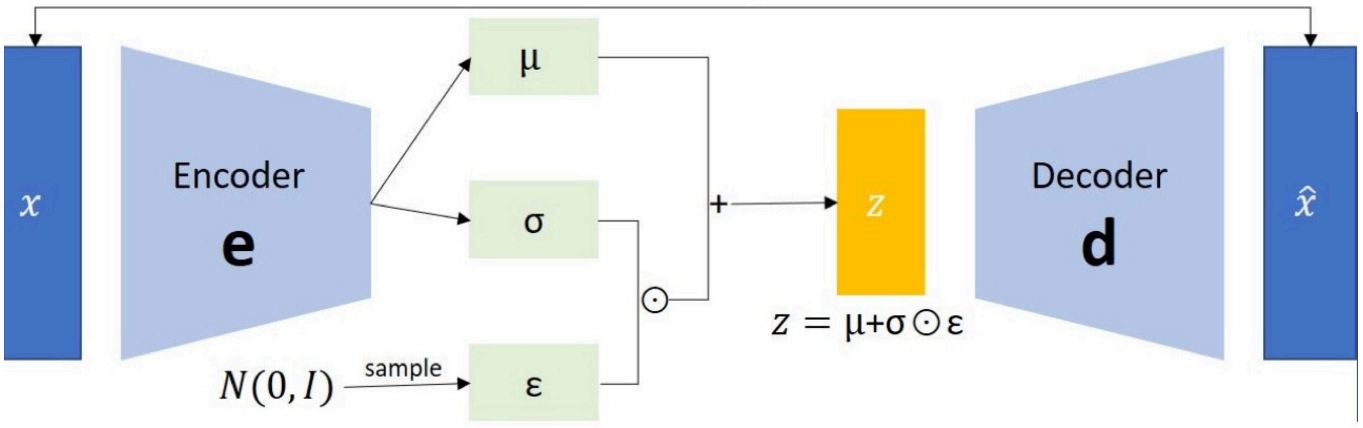
- Points in the latent space are scattered irregularly
- Large empty gaps exist between data clusters
- Random sampling often produces distorted or unrealistic images

The VAE solves these problems through its unique approach:

It enforces a specific distribution (e.g., Gaussian) on the latent space The space becomes continuous and fully covered *without major gaps* Random sampling reliably generates realistic and meaningful images

This structured approach enables high-quality generation of diverse, semantically valid new data samples. The key innovation is the probabilistic encoding that organizes the latent space in a predictable, well-behaved manner.

Variational Autoencoder Architecture



How Does the Encoder Work in VAE?

Unlike the standard Autoencoder, where the Encoder outputs a single latent vector, the VAE Encoder generates two vectors for each input:

- **Mean Vector (μ):** Represents the central point of the latent representation.
- **Standard Deviation Vector (σ):** Indicates the variance around this point.

VAE Workflow:

1. **Encoder:** Calculates the mean (μ) and standard deviation (σ) vectors for the input image.
2. **Sampling:** Generates a random vector (ε) from a standard normal distribution (mean 0, standard deviation 1), then adjusts it using the reparameterization trick to produce the latent representation (z):

$$\mathrm{z} = \mu + \sigma \odot \epsilon$$

3. **Decoder:** Reconstructs the image from the latent representation (z).

This randomness ensures the latent space is structured and suitable for generating new samples.

Loss Function in VAE

The VAE loss function combines two components:

1. **Reconstruction Loss**
   Measures the difference between the original image (x) and the reconstructed image (\hat{x}).
   Typically uses Mean Squared Error (MSE):

$$\text{Reconstruction Loss} = \frac{1}{N}\sum_{i=1}^{N}(x_i - \hat{x}_i)^2$$

2. **KL Divergence Loss**
   Quantifies the deviation between the learned distribution and standard normal distribution:

$$\sum_{i=1}^{n}\sigma_i^2 + \mu_i^2 - \log(\sigma_i) - 1$$

3. **Total Loss**

$$\text{Total Loss} = \text{Reconstruction Loss} + \text{KLD}$$

Why Both Components?

- **Reconstruction Loss:** Ensures the output closely matches the input.
- **KL Divergence:** Regularizes the latent space to resemble a normal distribution, facilitating generation.

Focusing only on KL Divergence would force (μ = 0) and (σ = 1), sacrificing reconstruction quality. Ignoring it would leave the latent space disorganized, hindering generation. The balance is key.

Applications

- ### **Image Clustering**:

  Similar images have similar embeddings in the bottleneck layer

  t-SNE can be used to visualize these similarities in 2D

- ### **Image Generation**:

  VAEs can generate new images by sampling from the latent space

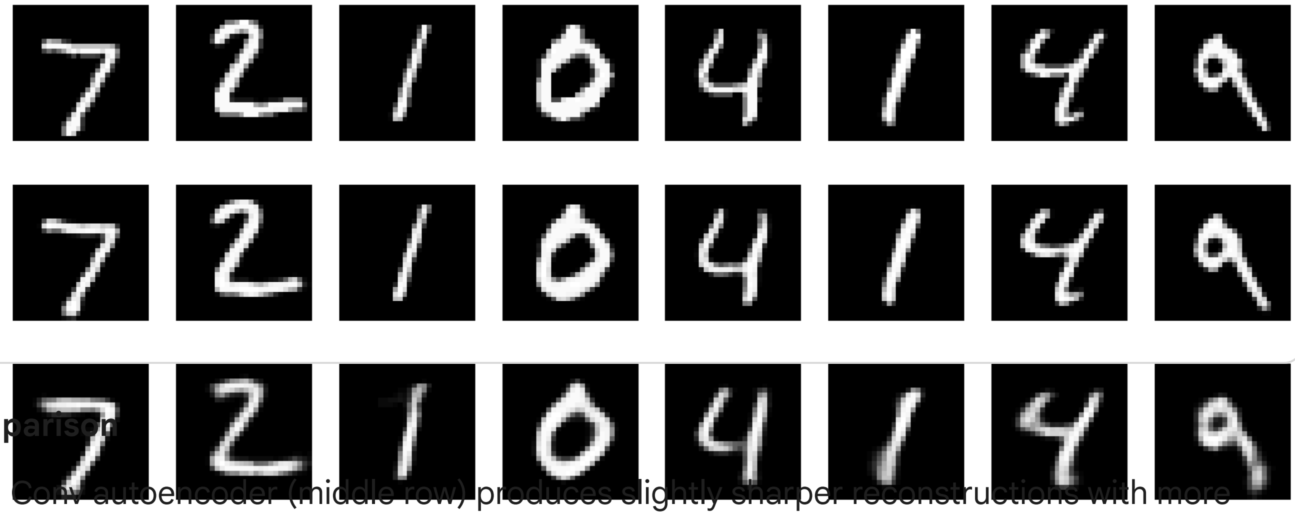
  Interpolation between different points in latent space creates smooth transitions

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

# Device configuration
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

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

batch_size = 128
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

#  Autoencoder Architecture
class ConvAutoencoder(nn.Module):
    def __init__(self):
        super().__init__()
        self.encoder = nn.Sequential(
            nn.Conv2d(1, 16, 3, stride=2, padding=1),
            nn.ReLU(),
            nn.Conv2d(16, 32, 3, stride=2, padding=1),
            nn.ReLU(),
            nn.Conv2d(32, 64, 3, padding=1),
            nn.ReLU()
        )
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(64, 32, 3, padding=1),
            nn.ReLU(),
            nn.ConvTranspose2d(32, 16, 3, stride=2, padding=1, output_padding=1),
            nn.ReLU(),
            nn.ConvTranspose2d(16, 1, 3, stride=2, padding=1, output_padding=1),
            nn.Sigmoid()
        )

    def forward(self, x):
        encoded = self.encoder(x)
        decoded = self.decoder(encoded)
        return decoded

# VAE Architecture
class ConvVAE(nn.Module):
    def __init__(self, latent_dim=128):
        super().__init__()
        self.encoder_conv = nn.Sequential(
            nn.Conv2d(1, 16, 3, stride=2, padding=1),
            nn.ReLU(),
            nn.Conv2d(16, 32, 3, stride=2, padding=1),
            nn.ReLU(),
            nn.Conv2d(32, 64, 3, padding=1),
            nn.ReLU()
        )
        self.flatten_size = 64 * 7 * 7
        self.fc_mu = nn.Linear(self.flatten_size, latent_dim)
        self.fc_logvar = nn.Linear(self.flatten_size, latent_dim)
        self.fc_decoder = nn.Linear(latent_dim, self.flatten_size)
        self.decoder_conv = nn.Sequential(
            nn.ConvTranspose2d(64, 32, 3, padding=1),
            nn.ReLU(),
            nn.ConvTranspose2d(32, 16, 3, stride=2, padding=1, output_padding=1),
            nn.ReLU(),
            nn.ConvTranspose2d(16, 1, 3, stride=2, padding=1, output_padding=1),
            nn.Sigmoid()
        )

    def encode(self, x):
        x = self.encoder_conv(x).view(-1, self.flatten_size)
```

```
            return self.fc_mu(x), self.fc_logvar(x)

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

        def decode(self, z):
            x = self.fc_decoder(z).view(-1, 64, 7, 7)
            return self.decoder_conv(x)

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

    #  Training Loop
    autoencoder = ConvAutoencoder().to(device)
    vae = ConvVAE().to(device)

    ae_optimizer = optim.Adam(autoencoder.parameters(), lr=1e-3)
    vae_optimizer = optim.Adam(vae.parameters(), lr=1e-3)

    # Training parameters
    epochs = 5

    # Train both models
    for epoch in range(1, epochs + 1):
        # Train Autoencoder
        autoencoder.train()
        for batch_idx, (data, _) in enumerate(train_loader):
            data = data.to(device)
            ae_optimizer.zero_grad()
            recon = autoencoder(data)
            loss = F.binary_cross_entropy(recon, data)
            loss.backward()
            ae_optimizer.step()

        # Train VAE
        vae.train()
        for batch_idx, (data, _) in enumerate(train_loader):
            data = data.to(device)
            vae_optimizer.zero_grad()
            recon, mu, logvar = vae(data)
            loss = F.binary_cross_entropy(recon, data, reduction='sum') - 0.5 * torch.s
            loss.backward()
            vae_optimizer.step()

        print(f'Epoch {epoch}/{epochs} completed')

    # Visualization
```

```
def visualize():
    autoencoder.eval()
    vae.eval()
    with torch.no_grad():
        test_images, _ = next(iter(test_loader))
        test_images = test_images.to(device)

        # Get reconstructions
        ae_recon = autoencoder(test_images[:8])
        vae_recon, _, _ = vae(test_images[:8])

        # Prepare plots
        fig, axs = plt.subplots(3, 8, figsize=(20, 8))

        for i in range(8):
            # Original
            axs[0, i].imshow(test_images[i].cpu().squeeze(), cmap='gray')
            axs[0, i].axis('off')

            # Autoencoder
            axs[1, i].imshow(ae_recon[i].cpu().squeeze(), cmap='gray')
            axs[1, i].axis('off')

            # VAE
            axs[2, i].imshow(vae_recon[i].cpu().squeeze(), cmap='gray')
            axs[2, i].axis('off')

        axs[0, 0].set_ylabel('Original')
        axs[1, 0].set_ylabel('Autoencoder')
        axs[2, 0].set_ylabel('VAE')
        plt.show()

visualize()
```

```
Using device: cpu
Epoch 1/5 completed
Epoch 2/5 completed
Epoch 3/5 completed
Epoch 4/5 completed
Epoch 5/5 completed
```



Comparison

- Conv autoencoder (middle row) produces slightly sharper reconstructions with more defined edges
- Better for exact reconstructions when that's the primary goal

---

- Conv VAE (bottom row) reconstructions are slightly softer/smoother
- Better for generative tasks and learning meaningful representations