Autoencoders

Data Science in Electron Microscopy

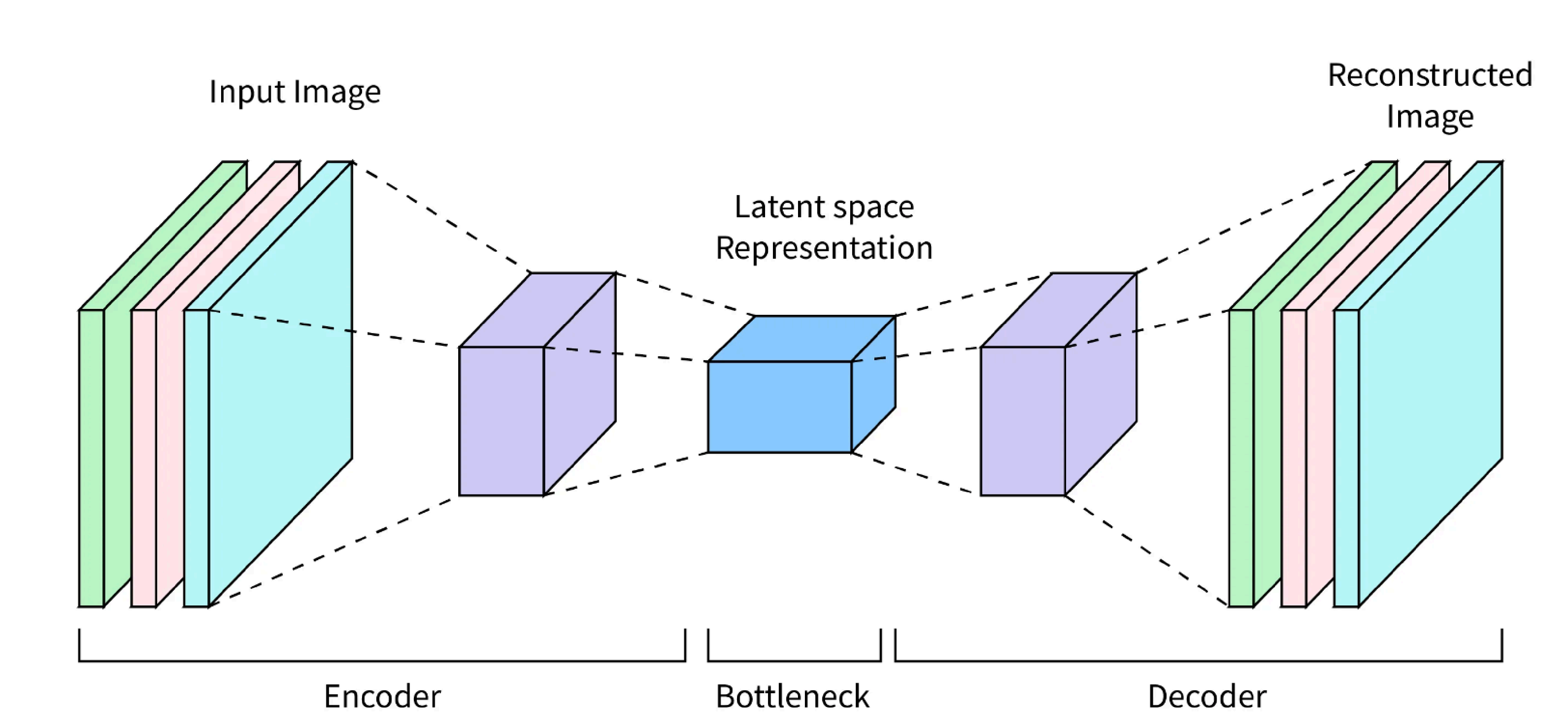
Philipp Pelz

2024

https://github.com/ECLIPSE-Lab/WS24_DataScienceForEM

What is an Autoencoder?

- Neural network architecture that learns to:
  - Compress (encode) data into a lower-dimensional representation
  - Reconstruct (decode) the original data from this representation
- Trained to minimize reconstruction error
- Learns efficient data representations unsupervised



Basic Autoencoder Architecture

Autoencoder Components

- **Encoder**: Compresses input into latent representation
- **Latent Space**: Compressed representation of the data
- **Decoder**: Reconstructs input from latent representation
- Training objective: minimize difference between input and output

```python
import torch
import torch.nn as nn

class ConvAutoencoder(nn.Module):
    def __init__(self):
        super().__init__()

        # Encoder
        self.encoder = nn.Sequential(
            nn.Conv2d(1, 16, 3, stride=2, padding=1),  # [B, 1, 28, 28] -> [B, 16, 14, 14]
            nn.ReLU(),
            nn.Conv2d(16, 32, 3, stride=2, padding=1), # [B, 16, 14, 14] -> [B, 32, 7, 7]
            nn.ReLU(),
            nn.Conv2d(32, 64, 7)                       # [B, 32, 7, 7] -> [B, 64, 1, 1]
        )

        # Decoder
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(64, 32, 7)               # [B, 64, 1, 1] -> [B, 32, 7, 7]
```

Training an Autoencoder

```python
def train_autoencoder(model, train_loader, num_epochs=10):
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    model = model.to(device)
    criterion = nn.MSELoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)

    for epoch in range(num_epochs):
        for data in train_loader:
            img = data[0].to(device)

            # Forward pass
            output = model(img)
            loss = criterion(output, img)

            # Backward pass
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
```

Applications of Autoencoders

- **Dimensionality Reduction**
  - Alternative to PCA
  - Can capture non-linear relationships
- **Denoising**
  - Train to reconstruct clean data from noisy input
  - Useful for image restoration
- **Feature Learning**
  - Learn meaningful representations for downstream tasks
  - Transfer learning

Variations of Autoencoders

- **Denoising Autoencoders**
  - Add noise to input during training
  - Learn to recover original data
- **Variational Autoencoders (VAE)**
  - Learn probabilistic encodings
  - Generate new samples
- **Sparse Autoencoders**
  - Add sparsity constraints to latent representation
  - Learn more efficient encodings

Example: Denoising Autoencoder

```
def add_noise(img, noise_factor=0.3):
    noisy = img + noise_factor * torch.randn(*img.shape)
    return torch.clamp(noisy, 0., 1.)

def train_denoising_autoencoder(model, train_loader, num_epochs=10):
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    model = model.to(device)
    criterion = nn.MSELoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)

    for epoch in range(num_epochs):
        for data in train_loader:
            img = data[0].to(device)
            noisy_img = add_noise(img)

            # Forward pass
            output = model(noisy_img)
            loss = criterion(output, img)  # Compare with clean image
```

Practical Tips for Autoencoders

- Choose appropriate architecture for your data type
  - CNNs for images
  - RNNs for sequences
  - Dense layers for tabular data
- Consider:
  - Latent space dimension
  - Depth of encoder/decoder
  - Loss function
  - Regularization techniques
- Common issues:
  - Overfitting
  - Underfitting
  - Mode collapse (in VAEs)
  - Reconstruction quality vs. compression trade-off

Variational Autoencoders (VAEs)

- Extension of traditional autoencoders that learns a **probabilistic** latent representation
- Instead of encoding to fixed points, encodes to probability distributions
- Enables:
  - Principled generation of new samples
  - Meaningful latent space interpolation
  - Better regularization of the latent space

VAE vs. Traditional Autoencoder

Traditional Autoencoder

- Deterministic encoding
- Point-wise latent representation
- No guarantee of continuous latent space
- Focus on reconstruction

Variational Autoencoder

- Probabilistic encoding
- Distribution-based latent representation
- Continuous, structured latent space
- Balance between reconstruction and regularization

VAE Mathematics

Instead of encoding input x to a point, VAE encodes to parameters of a distribution:

- Encoder outputs μ and $\log\sigma^2$ for each latent dimension
- Latent vector is sampled: $z = \mu + \sigma \odot \epsilon$, where $\epsilon \sim \mathcal{N}(0, I)$

The VAE loss has two terms:

$$\mathcal{L}_{\text{VAE}} = \mathcal{L}_{\text{reconstruction}} + \beta \cdot \mathcal{L}_{\text{KL}}$$

where:

$$\mathcal{L}_{\text{KL}} = \frac{1}{2} \sum_{j=1}^{J} (\mu_j^2 + \sigma_j^2 - \log(\sigma_j^2) - 1)$$

VAE Implementation

```python
class ConvVAE(nn.Module):
    def __init__(self, latent_dim=32):
        super().__init__()

        # Encoder
        self.encoder = nn.Sequential(
            nn.Conv2d(1, 32, 3, stride=2, padding=1),  # 28x28 -> 14x14
            nn.ReLU(),
            nn.Conv2d(32, 64, 3, stride=2, padding=1), # 14x14 -> 7x7
            nn.ReLU(),
            nn.Flatten(),
            nn.Linear(64 * 7 * 7, 256)
        )

        # Latent space
        self.fc_mu = nn.Linear(256, latent_dim)
        self.fc_var = nn.Linear(256, latent_dim)

        # Decoder
```

Training a VAE

```
def vae_loss(recon_x, x, mu, log_var, beta=1.0):
    # Reconstruction loss (binary cross entropy)
    BCE = F.binary_cross_entropy(recon_x, x, reduction='sum')

    # KL divergence loss
    KLD = -0.5 * torch.sum(1 + log_var - mu.pow(2) - log_var.exp())

    return BCE + beta * KLD
```

```python
def train_vae(model, train_loader, num_epochs=10):
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    model = model.to(device)
    optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)

    for epoch in range(num_epochs):
        for data in train_loader:
            img = data[0].to(device)

            # Forward pass
            recon_batch, mu, log_var = model(img)
            loss = vae_loss(recon_batch, img, mu, log_var)

            # Backward pass
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
```

VAE Latent Space Properties

- **Continuous**: Similar points in latent space decode to similar images
- **Structured**: Enforced by KL divergence term
- **Meaningful**: Can perform interpolation and arithmetic in latent space

VAE Latent Space Visualization

Generating New Samples with VAE

```
def generate_samples(model, num_samples=1):
    with torch.no_grad():
        # Sample from standard normal distribution
        z = torch.randn(num_samples, model.latent_dim).to(device)
        # Decode the samples
        samples = model.decode(z)
    return samples

def interpolate(model, img1, img2, steps=10):
    # Encode both images
    mu1, _ = model.encode(img1)
    mu2, _ = model.encode(img2)

    # Create interpolation points
    alphas = torch.linspace(0, 1, steps)
    interpolated = []

    with torch.no_grad():
        for alpha in alphas:
```

Key Differences Summary

1. **Latent Space**
   - Vanilla: Discrete, potentially discontinuous
   - VAE: Continuous, probabilistic
2. **Loss Function**
   - Vanilla: Only reconstruction loss
   - VAE: Reconstruction + KL divergence loss
3. **Generation Capabilities**
   - Vanilla: Limited/unreliable
   - VAE: Principled generation of new samples
4. **Training Stability**
   - Vanilla: Can be unstable
   - VAE: More stable due to regularization

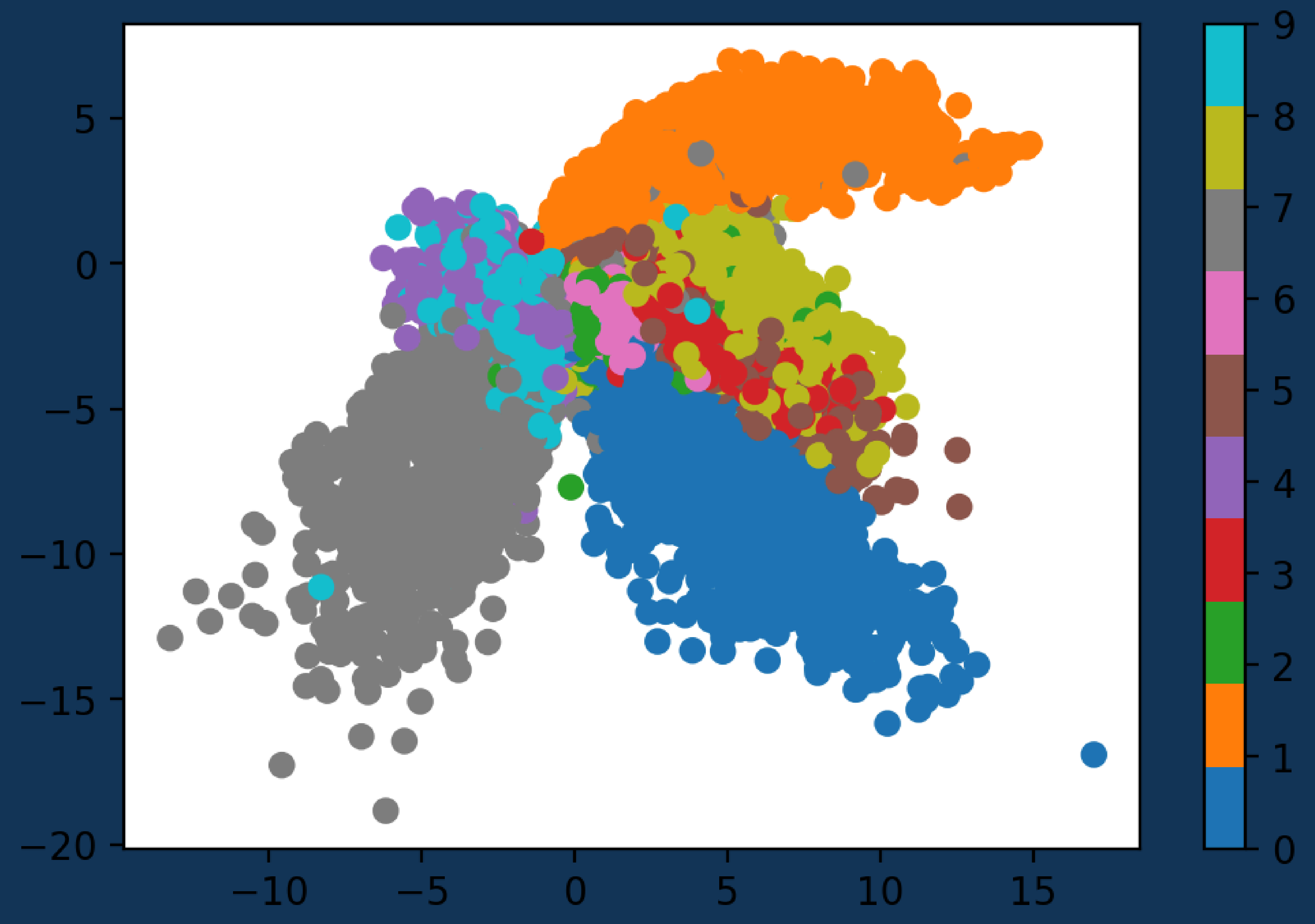
Example training a VAE

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


def add_noise(x, noise_factor=0.3):
    noisy = x + noise_factor * torch.randn_like(x)
    return torch.clamp(noisy, 0., 1.)

def train_epoch(model, dataloader, optimizer, device, noise_factor=0.3):
    model.train()
    train_loss = 0

    for batch_idx, (data, _) in enumerate(dataloader):
        data = data.to(device)
        noisy_data = add_noise(data, noise_factor)
```

```
Using device: cuda
Training completed and model saved!
```

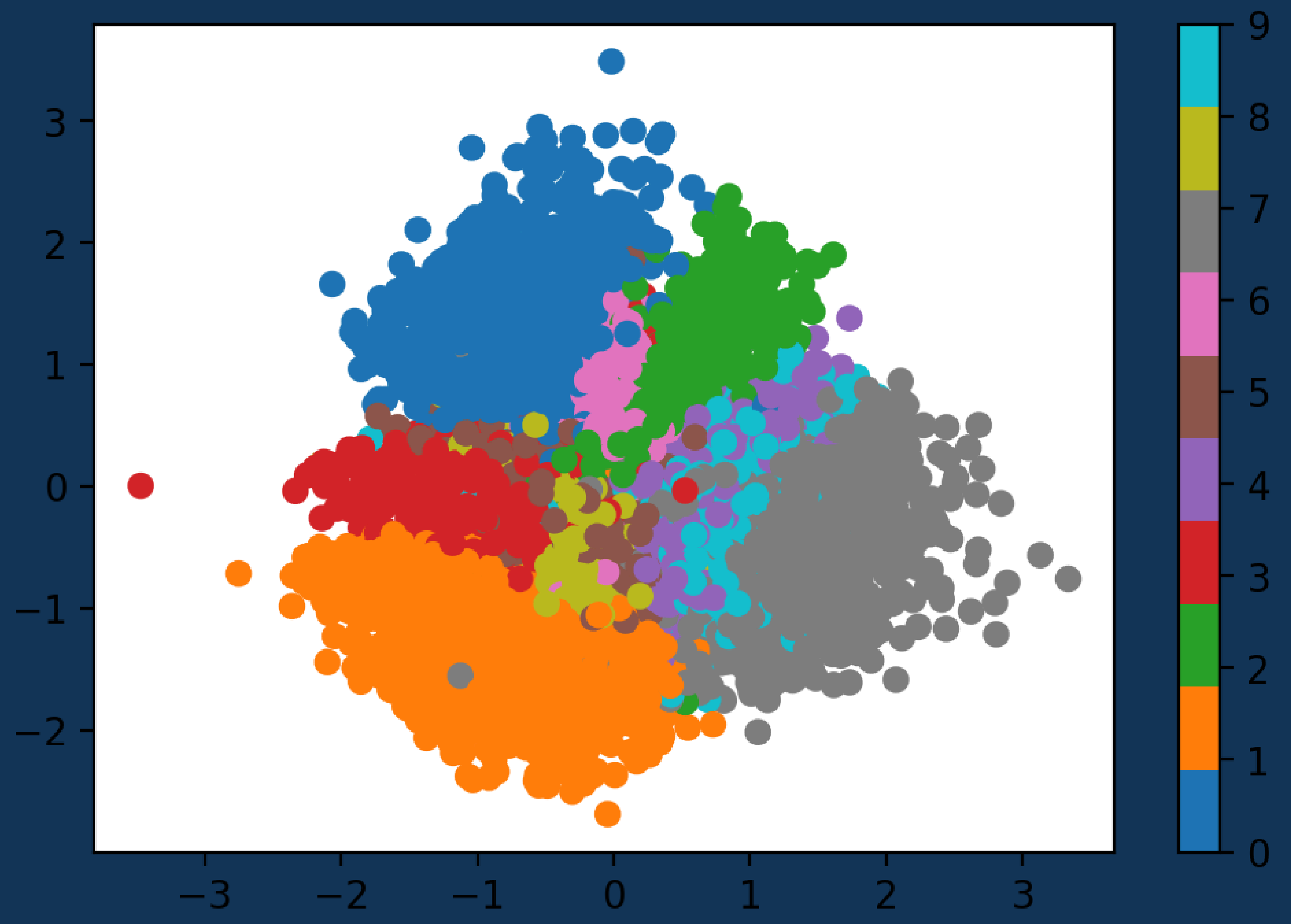
VAE vs. Traditional Autoencoder

Traditional Autoencoder



Vanilla AE Latent Space Visualization

Variational Autoencoder



VAE Latent Space Visualization