

Lab 9: Autoencoders and Latent Spaces

University of Washington

EE 596/AMATH 563

Spring 2021

Outline

- Autoencoders Introduction
- Latent space representation
- Variational Autoencoders
- Example: Fully Connected MNIST Autoencoder
- Assignment: Autoencoder enhancements

Autoencoders Introduction

Decompositions as low-dimensional representations

- SVD, PCA, KLD are all used as low-dimensional representations of data
- Few dominant directions capture majority of useful information
- Can reconstruct original data using small number of directions and projecting back to original space

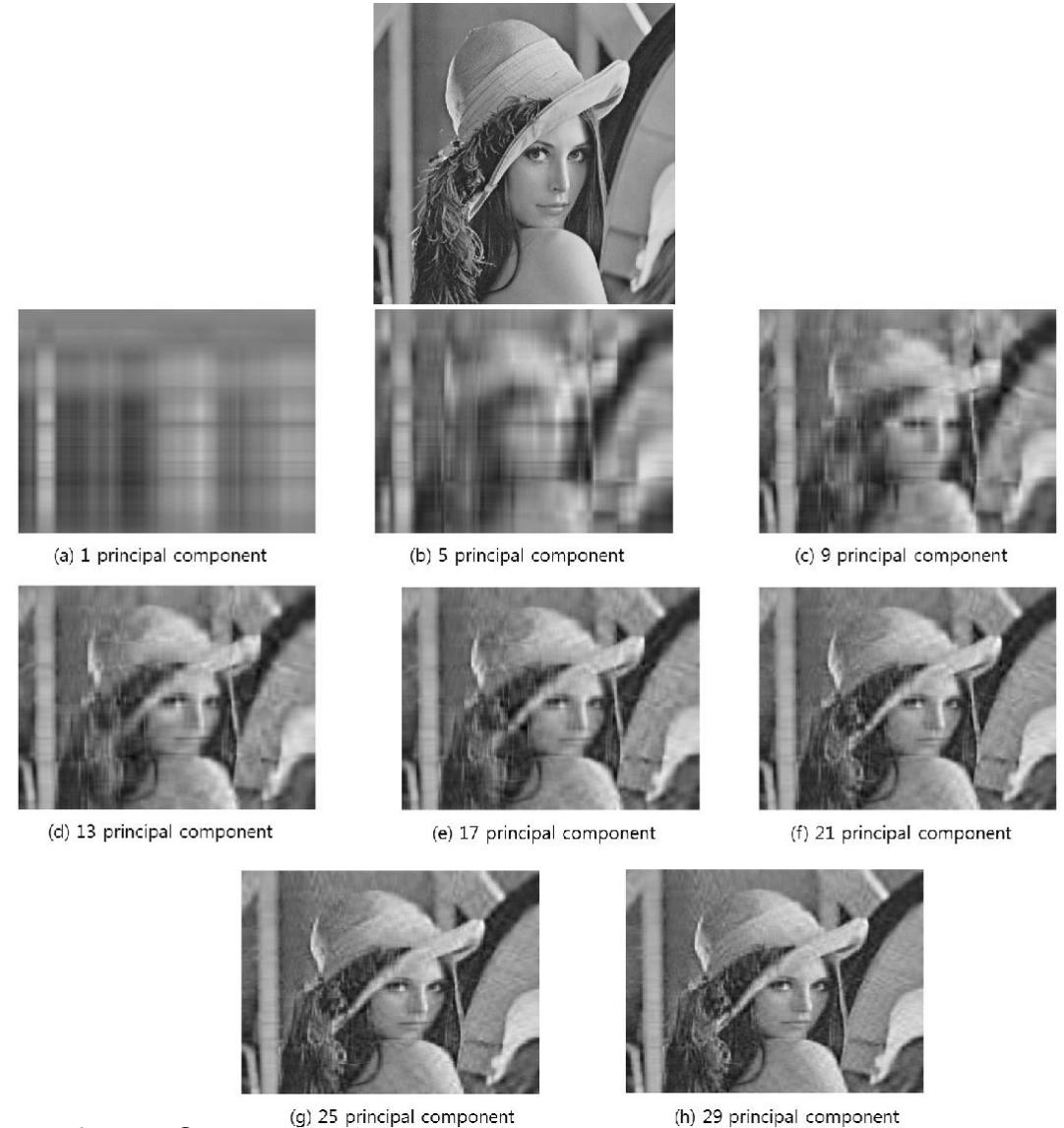
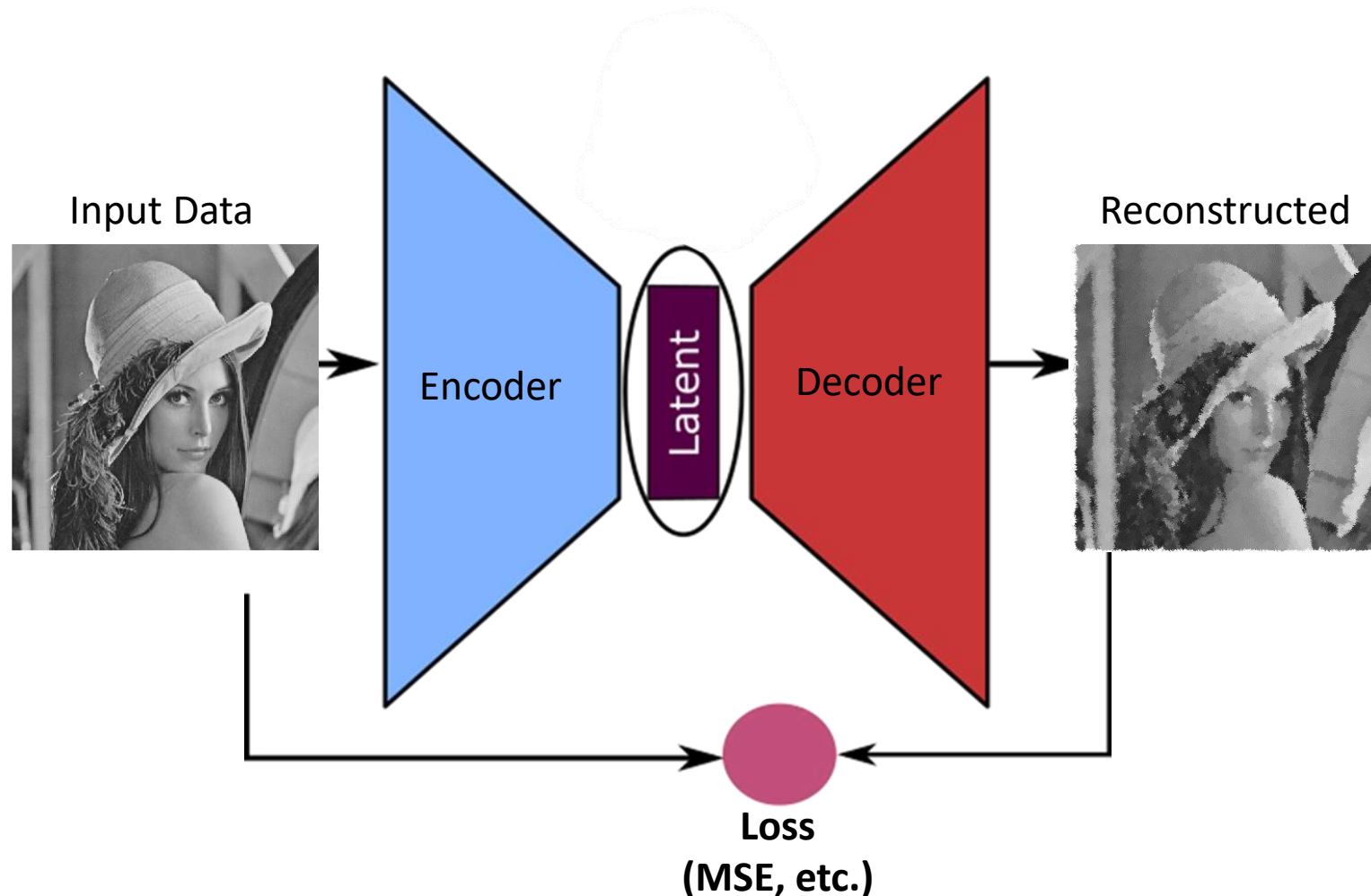


Image Source:

https://www.projectrhea.org/rhea/index.php/PCA_Theory_Examples

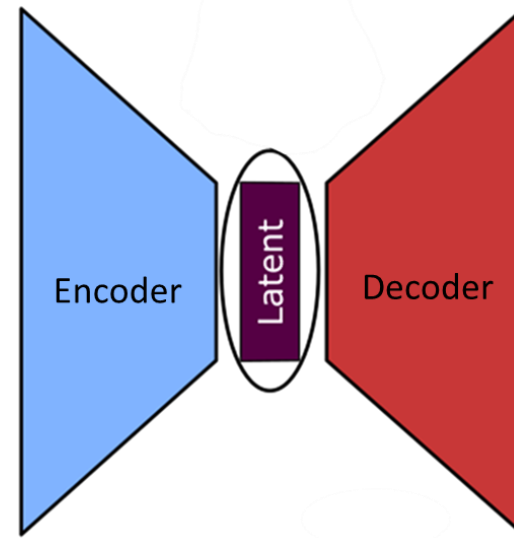
Autoencoders: Overview

- Input data is encoded into a low-dimensional latent space
- Decoder transforms back from latent space to original space
- Loss evaluated on difference between input and output
- Unsupervised learning



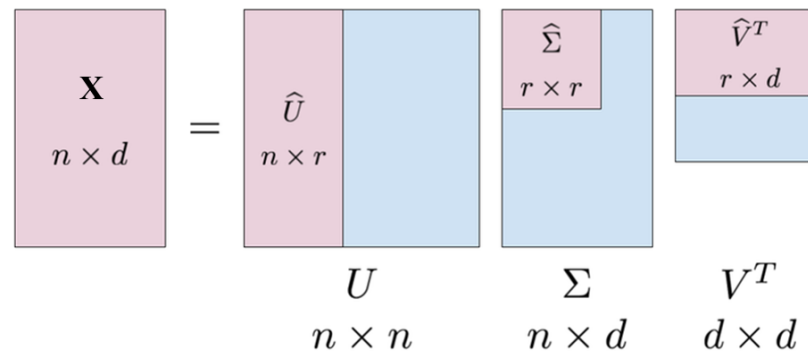
Autoencoders vs Classical Decompositions

- Autoencoders learned a fixed-sized representation
- Autoencoder decomposition is **nonlinear** and **learned iteratively**
- Latent space non necessarily orthogonal



Autoencoder

- Fixed-Dimension (hyperparameter)
- Non-linear
- Trained/learned
- Non-orthogonal basis

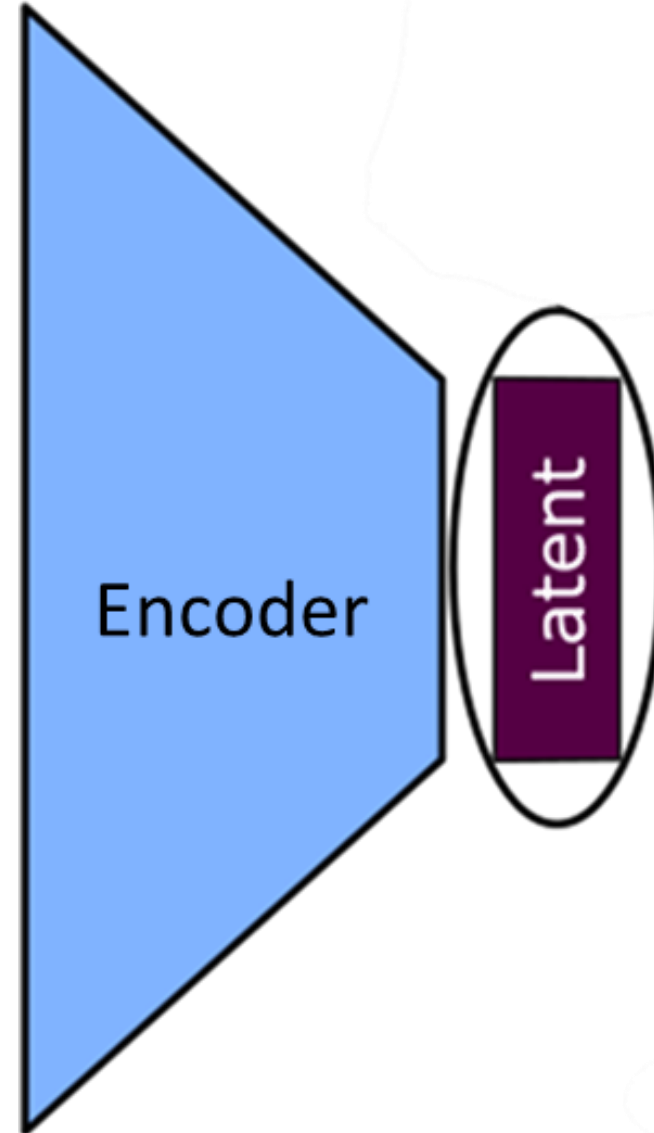


POD

- Full dimension (choose r after)
- Linear
- Deterministic
- Orthogonal basis

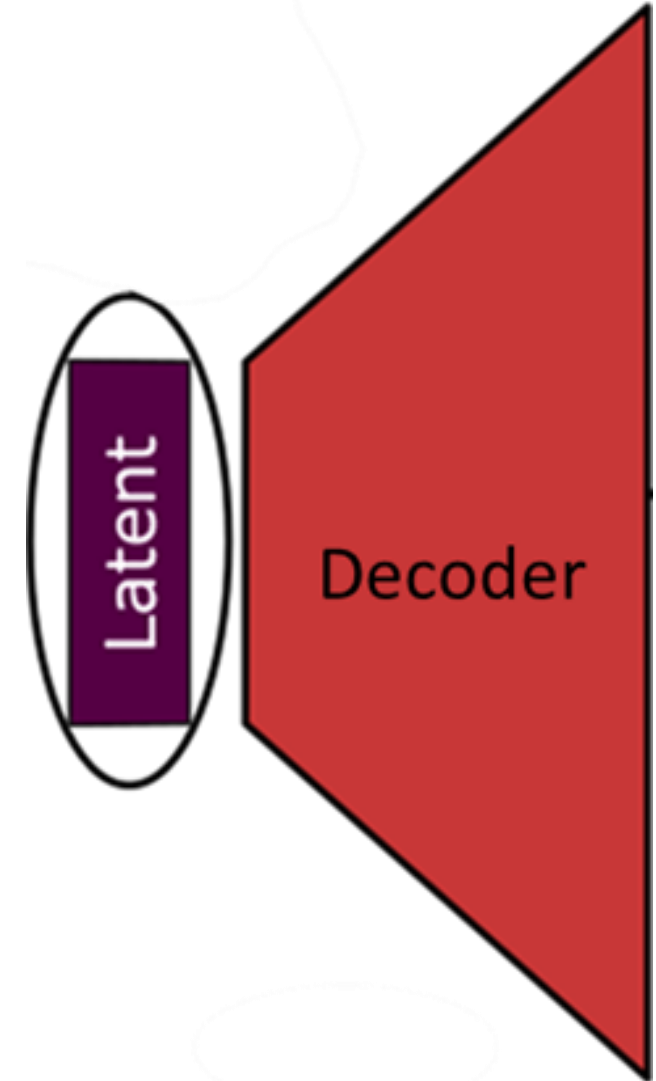
Autoencoder Structure: Encoder

- Encoder purpose: transform full-dimensional input to low-dimensional latent representation
- Architecture choice is flexible:
 - Fully connected
 - CNN
 - RNN



Autoencoder Structure: Decoder

- Decoder purpose: transform low-dimensional latent space to full-dimensional output
- Like encoder, architecture choice is flexible:
 - Fully connected
 - CNN
 - RNN
- Does not need to use same architecture type as Encoder



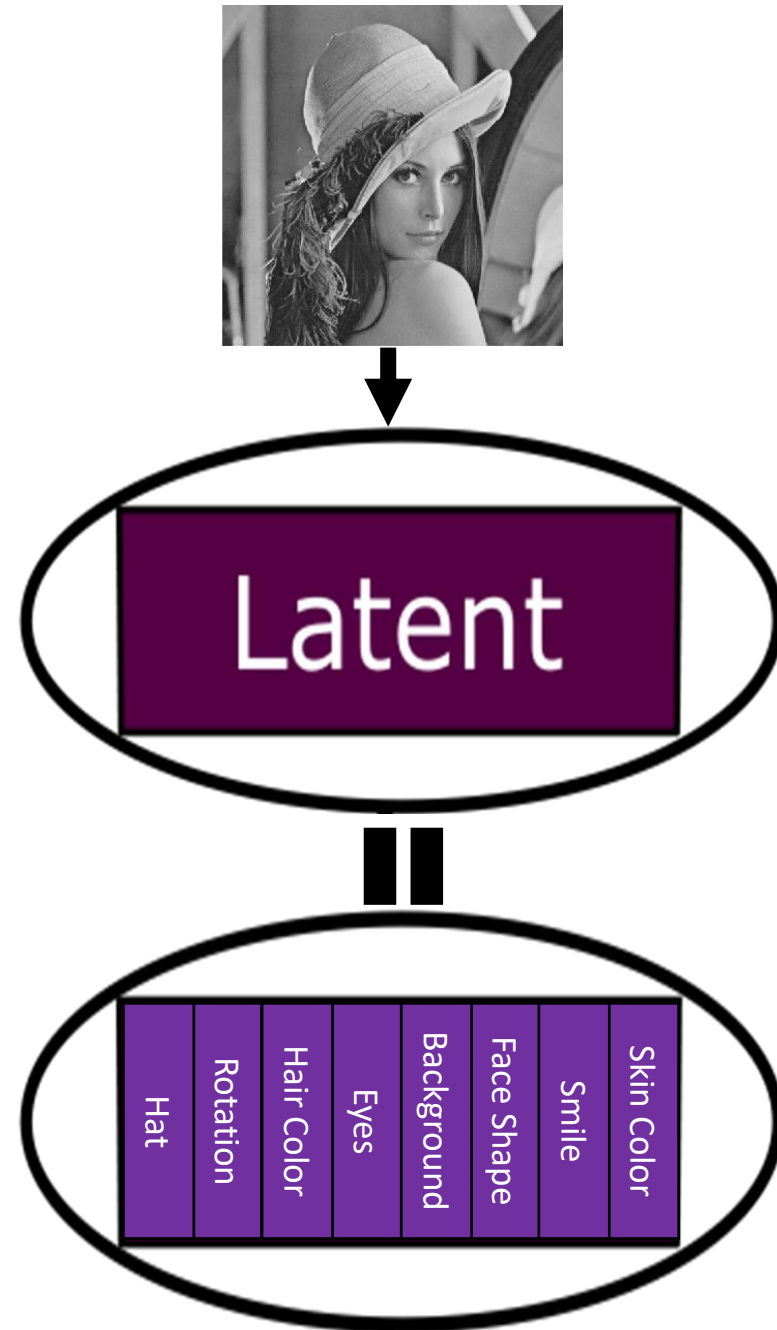
Sample Implementation in PyTorch

```
1 class Autoencoder(nn.Module):
2     def __init__(self, input_size, latent_size, p_drop = 0.2):
3         super(Autoencoder, self).__init__()
4
5         self.drop = nn.Dropout(p = p_drop)
6
7         #Define Encoder Layers
8         self.fc_e1 = nn.Linear(in_features = input_size, out_features = input_size//8)
9         self.fc_e2 = nn.Linear(in_features = input_size//8, out_features = latent_size)
10
11        #Define Decoder Layers
12        self.fc_d1 = nn.Linear(in_features = latent_size, out_features = input_size//4)
13        self.fc_d2 = nn.Linear(in_features = input_size//4, out_features = input_size)
14
15    def forward(self, input):
16        enc = self.drop(nn.Tanh()(self.fc_e1(input)))
17        latent = self.fc_e2(enc)
18        dec = self.drop(nn.Tanh()(self.fc_d1(latent)))
19        output = self.fc_d2(dec)
20        return output, latent
```

Latent Space Representation

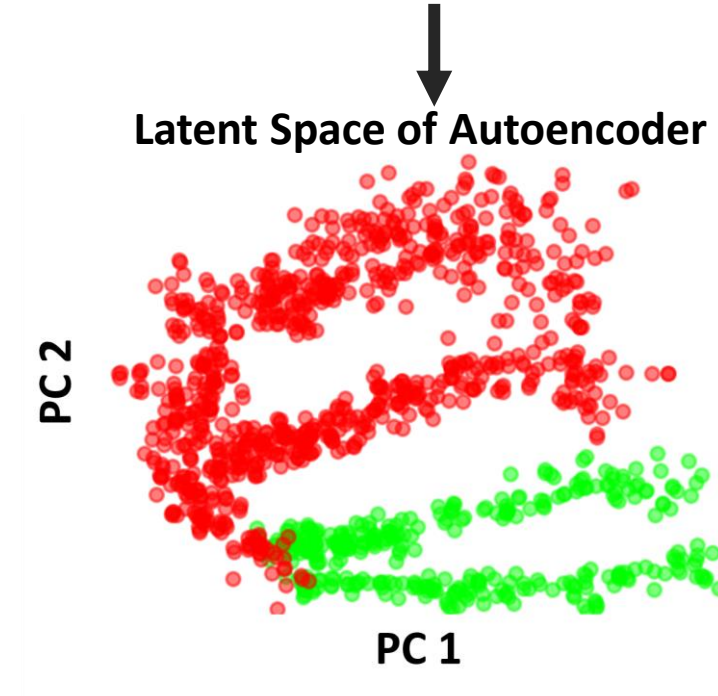
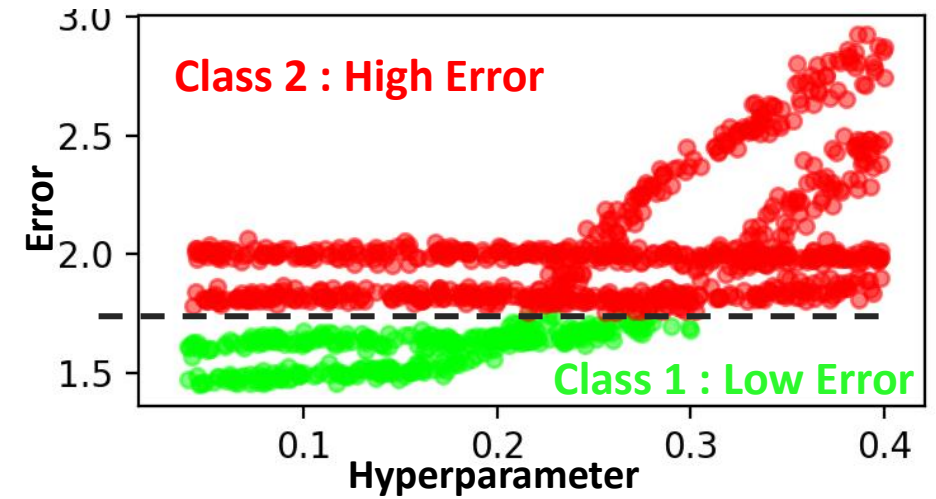
Latent Space Features

- Vectors in latent space not necessarily orthogonal
- Unlike POD, directions are not ordered
- All dimensions used to create faithful reconstruction
- Directions can encode particular features in the data



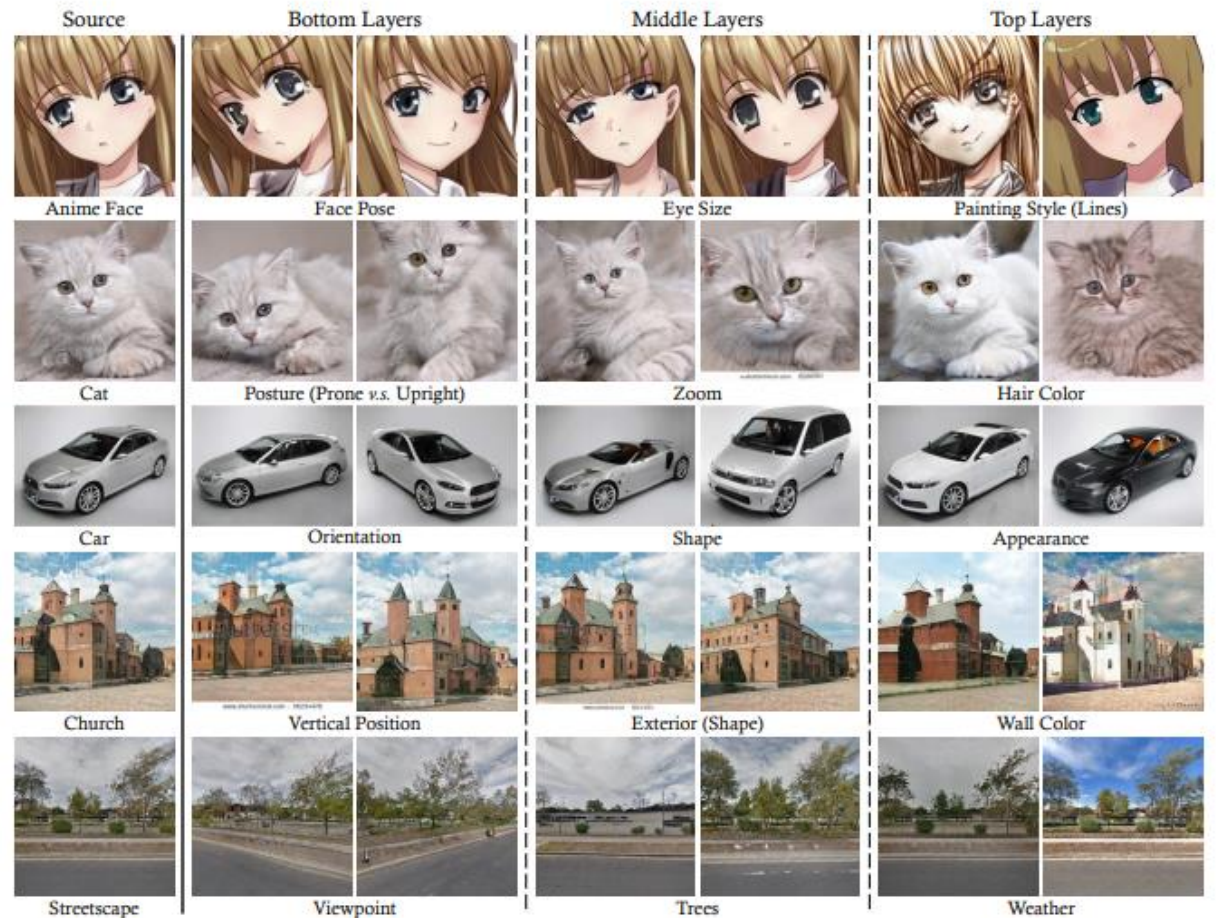
Clustering in Latent Space

- Projection onto the latent space can yield clusters
- Since unordered, you may need to take decomposition of latent space to visualize



Latent Space Manipulations

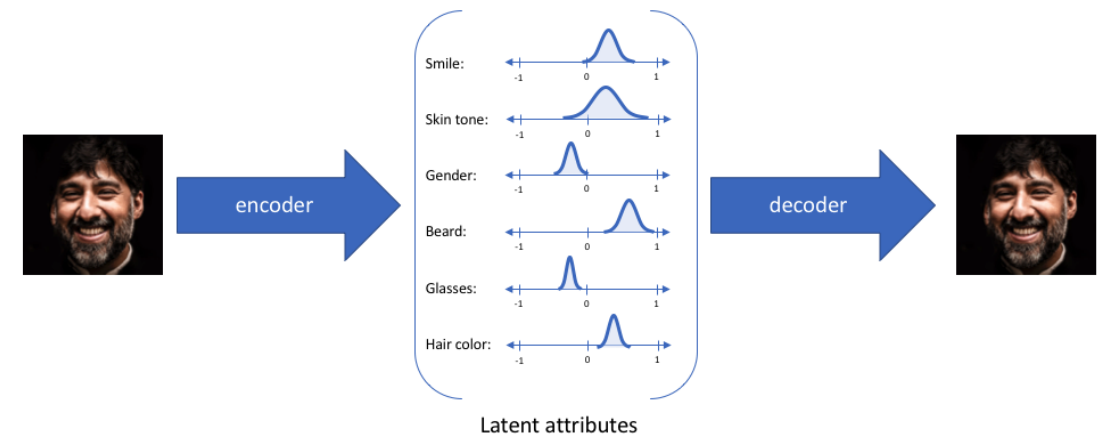
- Can identify dominant directions in latent space by taking decomposition (eigen, SVD, etc.)
- Aided by semantic factorization
 - Interpretable feature representation
- For generative models (such as GANs), can manipulate specific directions to control desired features



Variational Autoencoders (VAE)

Latent Space as a Distribution

- Latent Space of VAEs learn the mean and standard deviation of a multi-dimensional Gaussian
- Input to decoder is a SAMPLE from the distribution defined by these values
- Loss function uses KL divergence in addition to MSE reconstruction loss



KL Divergence for VAE

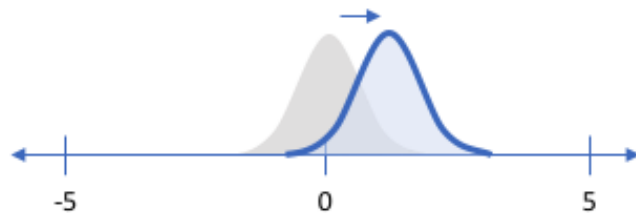
- Encoder generates a probability distribution $Q(z)$ as a function of input X (represents z most likely to produce X)
- Decoder generates conditional probability $P(X/z)$
- For Gaussian Q , KL divergence becomes:

$$\mathcal{D} [Q(z) \| P(z|X)] = E_{z \sim Q} [\log Q(z) - \log P(z|X)]$$

$$\mathcal{D}[\mathcal{N}(\mu(X), \Sigma(X)) \| \mathcal{N}(0, I)] = \frac{1}{2} \left(\text{tr}(\Sigma(X)) + (\mu(X))^\top (\mu(X)) - k - \log \det(\Sigma(X)) \right)$$

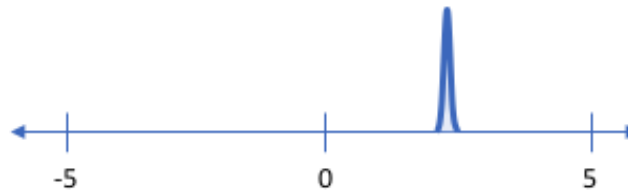
Mixed Losses: MSE and KL Divergence

Penalizing reconstruction loss encourages the distribution to describe the input



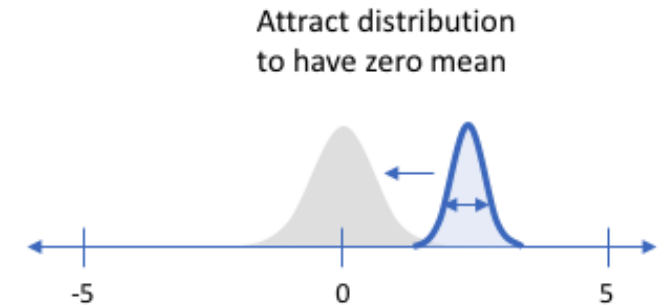
Our distribution deviates from the prior to describe some characteristic of the data

Without regularization, our network can “cheat” by learning narrow distributions



With a small enough variance, this distribution is effectively only representing a single value

Penalizing KL divergence acts as a regularizing force



Ensure sufficient variance to yield a smooth latent space

Variational Autoencoder Enhancements

- There are many variations to VAE structure which change the loss function/constraints



Beta-VAE



DIP VAE

Reconstructed Faces
from VAE variants

- **beta-VAE:** Learns interpretable factorization of data generative factors
(<https://openreview.net/forum?id=Sy2fzU9gl>)
- **Deep Feature Consistent VAE:** enforces deep feature consistency to preserve spatial correlations of data
(<https://arxiv.org/abs/1610.00291>)
- **DIP VAE:** Introduces an additional regularizer to encourage disentanglement of latent space
(<https://arxiv.org/abs/1711.00848>)

Example: Fully Connected AE on MNIST

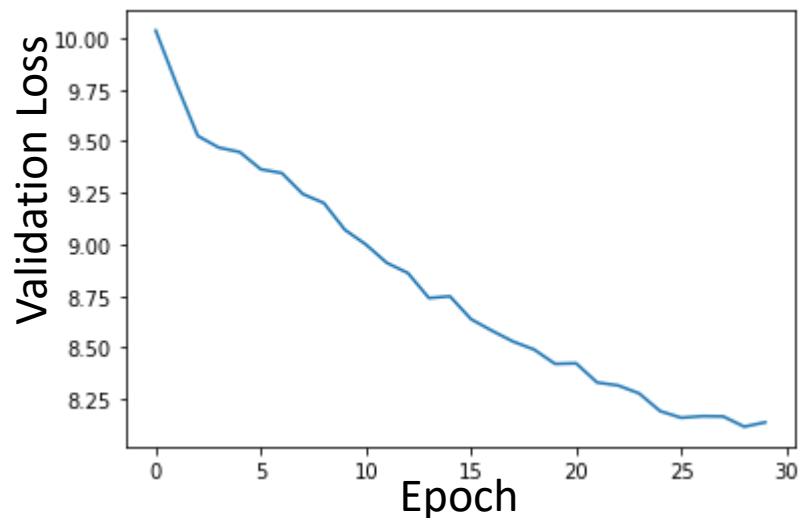
Autoencoder Setup and hyperparameters

- Same autoencoder setup as in previous section, except it now also returns the latent values
- Used latent size of 32
- Input size for MNIST is 784 (when flattened)

```
1 class Autoencoder(nn.Module):
2     def __init__(self, input_size, latent_size, p_drop = 0.2):
3         super(Autoencoder, self).__init__()
4
5         self.drop = nn.Dropout(p = p_drop)
6
7         #Define Encoder Layers
8         self.fc_e1 = nn.Linear(in_features = input_size, out_features = input_size//8)
9         self.fc_e2 = nn.Linear(in_features = input_size//8, out_features = latent_size)
10
11        #Define Decoder Layers
12        self.fc_d1 = nn.Linear(in_features = latent_size, out_features = input_size//4)
13        self.fc_d2 = nn.Linear(in_features = input_size//4, out_features = input_size)
14
15    def forward(self, input):
16        enc = self.drop(nn.Tanh()(self.fc_e1(input)))
17        latent = self.fc_e2(enc)
18        dec = self.drop(nn.Tanh()(self.fc_d1(latent)))
19        output = self.fc_d2(dec)
20        return output, latent
```

Autoencoder Training

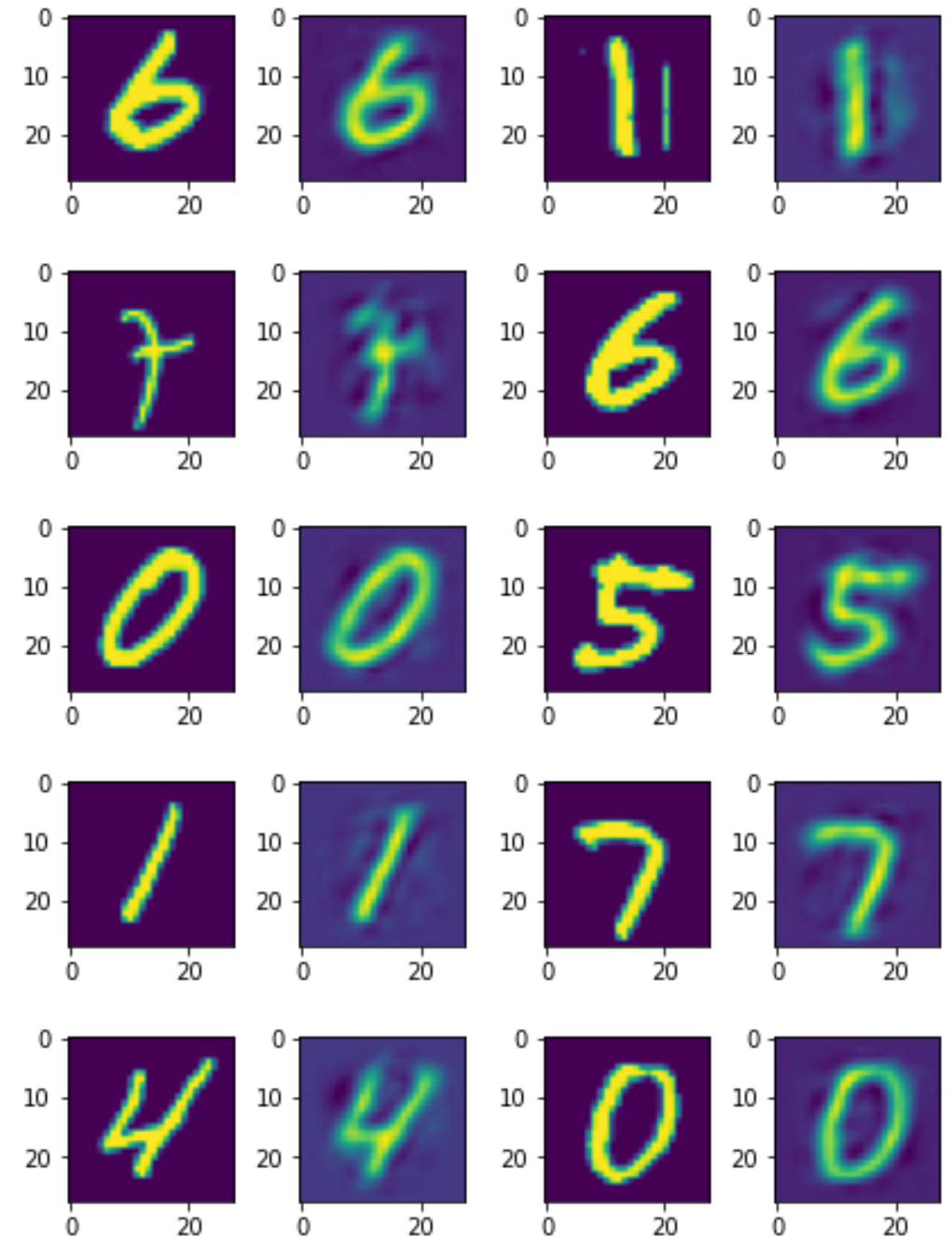
- We use the inputs as the targets for the autoencoder
- `MSELoss()` is the loss function
- Adam optimizer, `lr = 0.001`, 30 epochs



```
1 from tqdm import tqdm
2 train_losses = []
3 val_losses = []
4 for epoch in tqdm(range(max_epochs)):
5     model.train()
6     for i, sample in enumerate(train_dataloader):
7         optimizer.zero_grad()
8         data, label = sample
9         data_in = data.view(data.shape[0], input_size)
10        output = model(data_in)[0]
11        loss = criterion(data_in, output)
12        loss.backward()
13        train_losses.append(loss.item())
14        optimizer.step()
15    model.eval()
16    vl = 0
17    with torch.no_grad():
18        for sample in val_dataloader:
19            data, label = sample
20            data_in = data.view(data.shape[0], input_size)
21            output = model(data_in)[0]
22            loss = criterion(data_in, output)
23            vl += loss.item()
24    val_losses.append(vl)
```

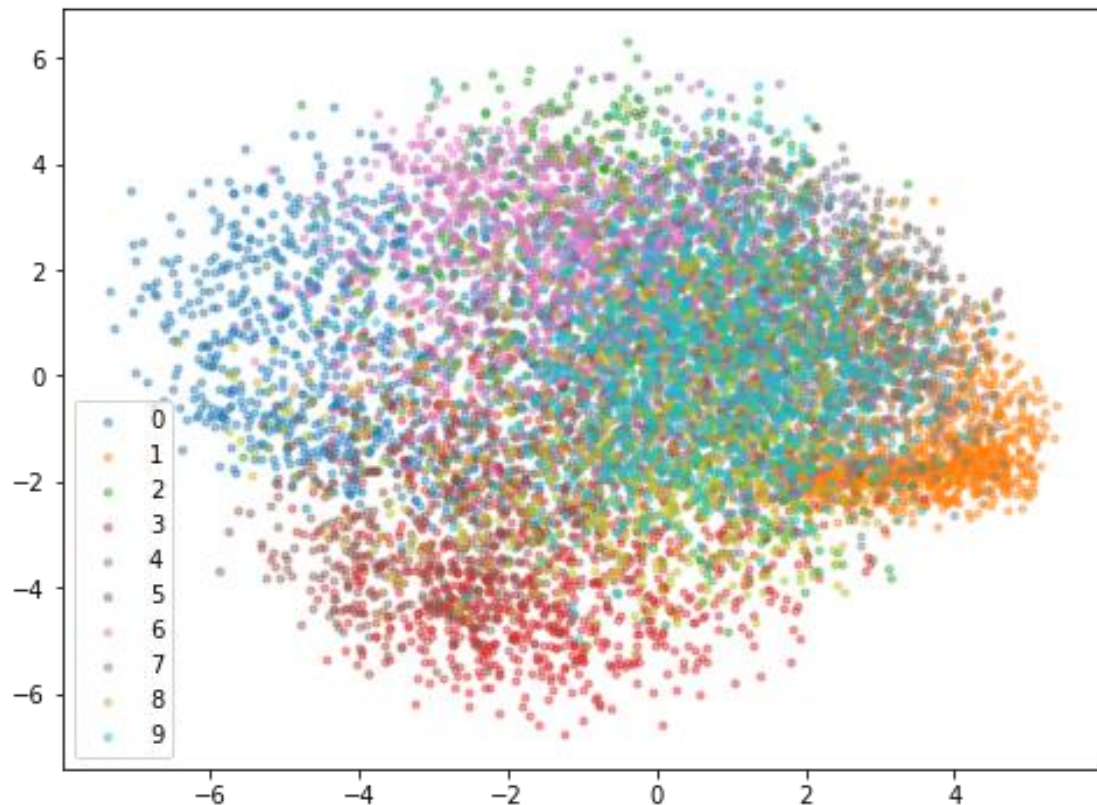
Reconstruction Results

- Successfully recaptured the pixels which pixels were had the highest values (yellow)
- Background contains noise (not purple)
- Shape of numbers is still clearly retained and identifiable



Clustering from Latent Space

- Numbers in test set were somewhat separated in latent space.

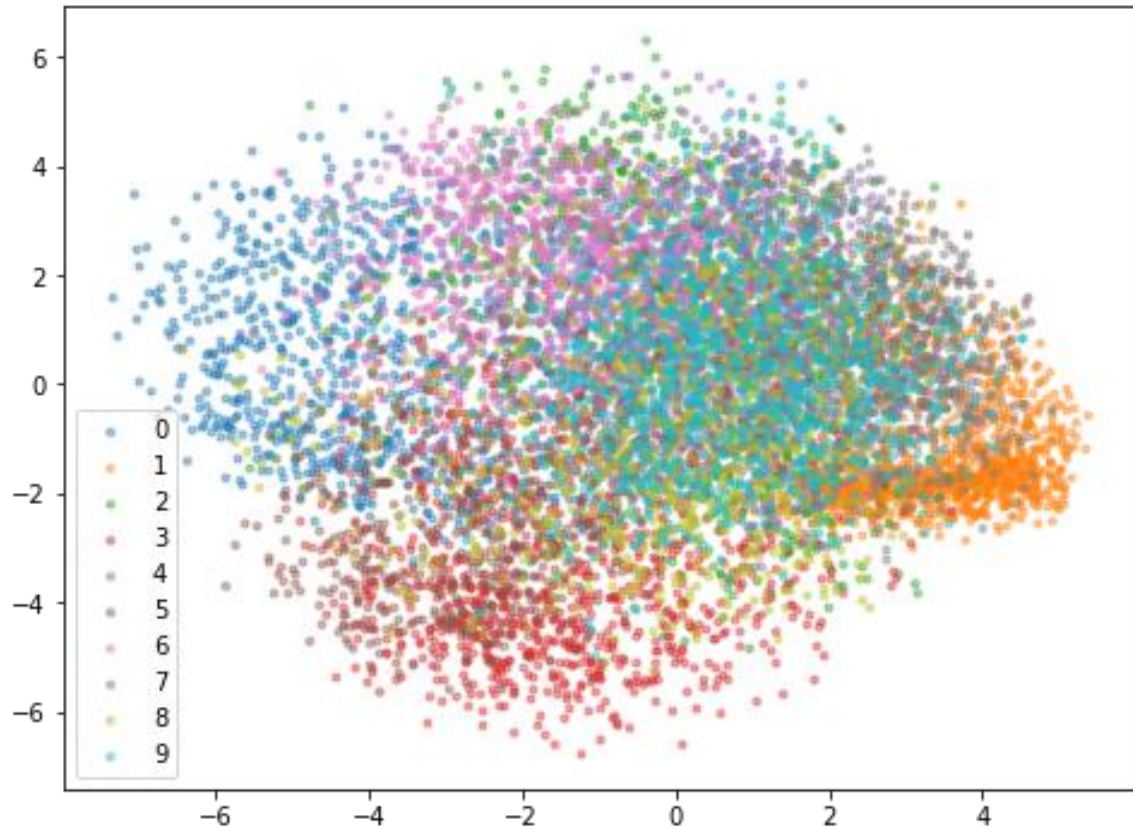


```
1 test_dataloader = torch.utils.data.DataLoader(test_set,
2                                             batch_size = 10000, shuffle = False)
3 for sample in test_dataloader:
4     data = sample[0]
5     data_in = data.view(data.shape[0], -1)
6     latents = model(data_in)[1]
7     labels = sample[1]
8 U, S, V = torch.pca_lowrank(latents)
```

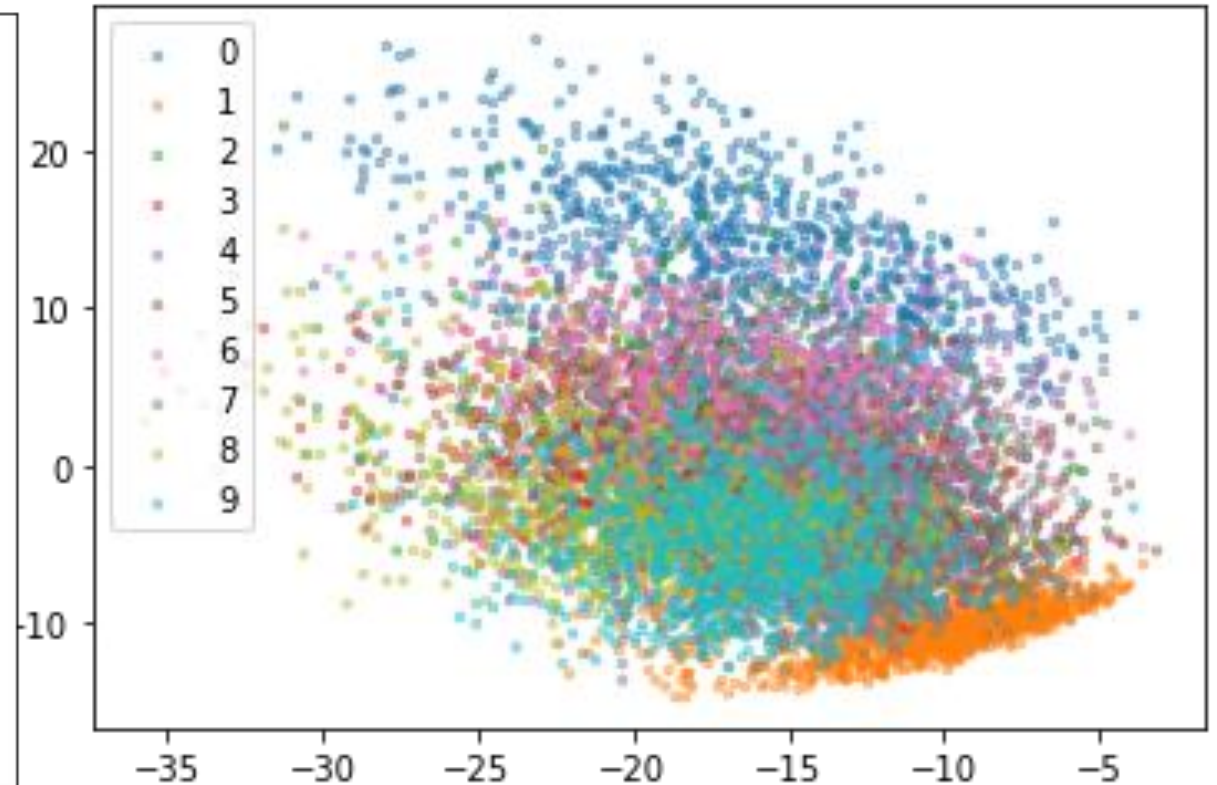
```
1 plt.figure(figsize = (8,6))
2 for i in range(10):
3     mask = labels == i
4     Y = (U[mask]@torch.diag(S)).detach()
5     plt.scatter(Y[:,0], Y[:,1], label = i, alpha = 0.4, s = 10)
6     plt.legend()
```

Comparison to SVD

Autoencoder Latent Space Clustering



SVD Clustering



Assignment: Enhanced Autoencoder on MNIST

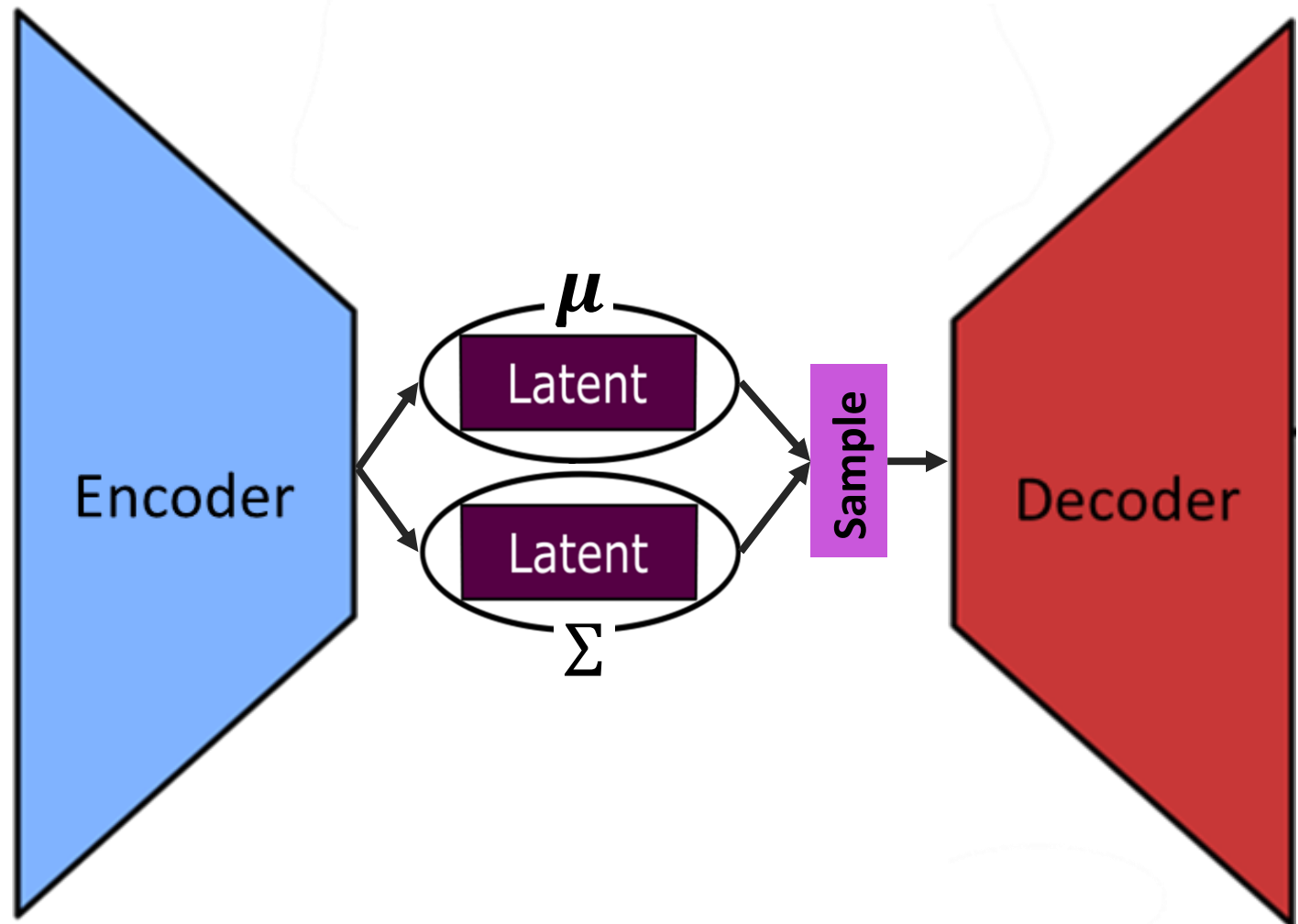
Your Task

- Modify my autoencoder class in at least one of the following ways (EC for both):
 1. Change the architecture of the encoder and/or decoder from fully connected to either CNN or RNN
 2. Implement a variational autoencoder (see next slides)
- Plot examples of input and reconstructed images
- Show clustering on test set
 - Does your method achieve better clustering than the FC autoencoder?
 - How does it compare to SVD?

```
1 class Autoencoder(nn.Module):
2     def __init__(self, input_size, latent_size, p_drop = 0.2):
3         super(Autoencoder, self).__init__()
4
5         self.drop = nn.Dropout(p = p_drop)
6
7         #Define Encoder Layers
8         self.fc_e1 = nn.Linear(in_features = input_size, out_features = input_size//8)
9         self.fc_e2 = nn.Linear(in_features = input_size//8, out_features = latent_size)
10
11        #Define Decoder Layers
12        self.fc_d1 = nn.Linear(in_features = latent_size, out_features = input_size//4)
13        self.fc_d2 = nn.Linear(in_features = input_size//4, out_features = input_size)
14
15    def forward(self, input):
16        enc = self.drop(nn.Tanh()(self.fc_e1(input)))
17        latent = self.fc_e2(enc)
18        dec = self.drop(nn.Tanh()(self.fc_d1(latent)))
19        output = self.fc_d2(dec)
20        return output, latent
```

VAE Implementation

- For the VAE, if you have a latent size of n , you will have two separate latent spaces of size n which will represent:
 1. The distribution means (μ)
 2. The distribution variances (Σ)



VAE Implementation

- Previous AE can be changed to VAE by adding another parallel layer in encoder and adding “reparameterize” function
- For loss function, implement KL divergence using standard normal as your prior (should be able to do this manually)

```
1 class ExampleVAE(nn.Module):
2     def __init__(self, input_size, latent_size, p_drop = 0.2):
3         super(ExampleVAE, self).__init__()
4
5         self.drop = nn.Dropout(p = p_drop)
6
7         #Define Encoder Layers
8         self.fc_e1 = nn.Linear(in_features = input_size, out_features = input_size//8)
9         self.fc_e2_1 = nn.Linear(in_features = input_size//8, out_features = latent_size)
10        self.fc_e2_2 = nn.Linear(in_features = input_size//8, out_features = latent_size)
11
12        #Define Decoder Layers
13        self.fc_d1 = nn.Linear(in_features = latent_size, out_features = input_size//4)
14        self.fc_d2 = nn.Linear(in_features = input_size//4, out_features = input_size)
15
16    def forward(self, input):
17        enc = self.drop(nn.Tanh()(self.fc_e1(input)))
18        mus = self.fc_e2_1(enc) #means of latent Gaussian
19        var = self.fc_e2_2(enc) #(log) var of latent Gaussian
20        sample = self.reparameterize(mus, var)
21
22        dec = self.drop(nn.Tanh()(self.fc_d1(sample)))
23        output = self.fc_d2(dec)
24        return output, (mus, var)
25
26    def reparameterize(self, mu, logvar):
27        ...
28        Used to generate sample from latent Gaussian
29        ...
30        std = torch.exp(0.5 * logvar)
31        eps = torch.randn_like(std)
32        return eps * std + mu
```