Lab 9: Autoencoders and Latent Spaces

University of Washington

EE 596/AMATH 563

Spring 2021

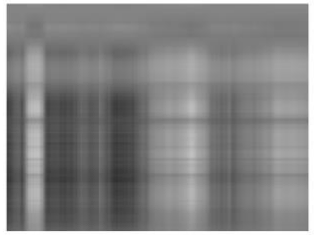
Outline

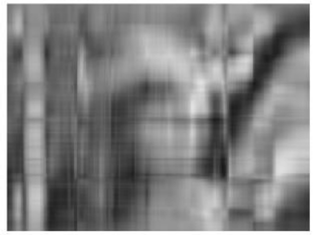
- Autoencoders Introduction
- Latent space analysis
- Variational Autoencoders
- Example:
- Assignment

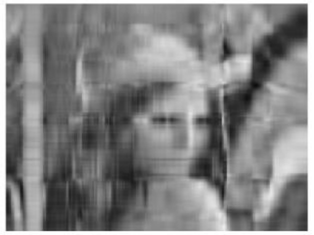
Autoencoders Introduction

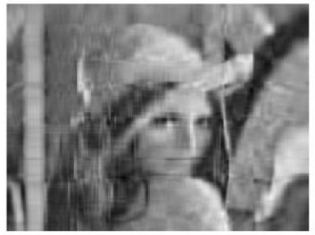
Decompositions as low-dimensional representations

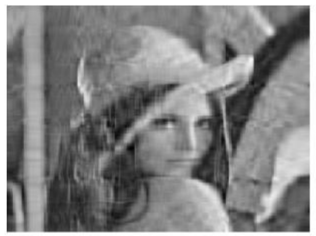
- SVD, PCA, KLD are all used as low-dimensional representations of data
- Few dominant directions capture majority of useful information
- Can reconstruct original data using small number of directions and projecting back to original space

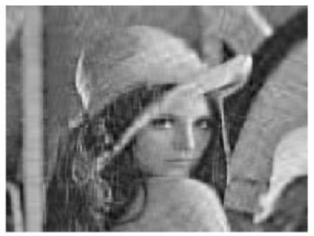

(a) 1 principal component

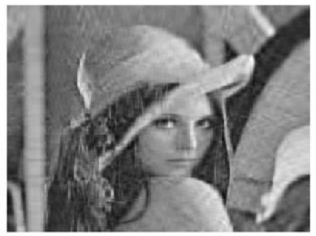

(b) 5 principal component


(c) 9 principal component


(d) 13 principal component


(e) 17 principal component


(f) 21 principal component


(g) 25 principal component

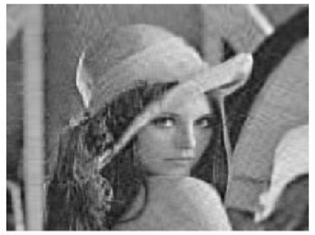
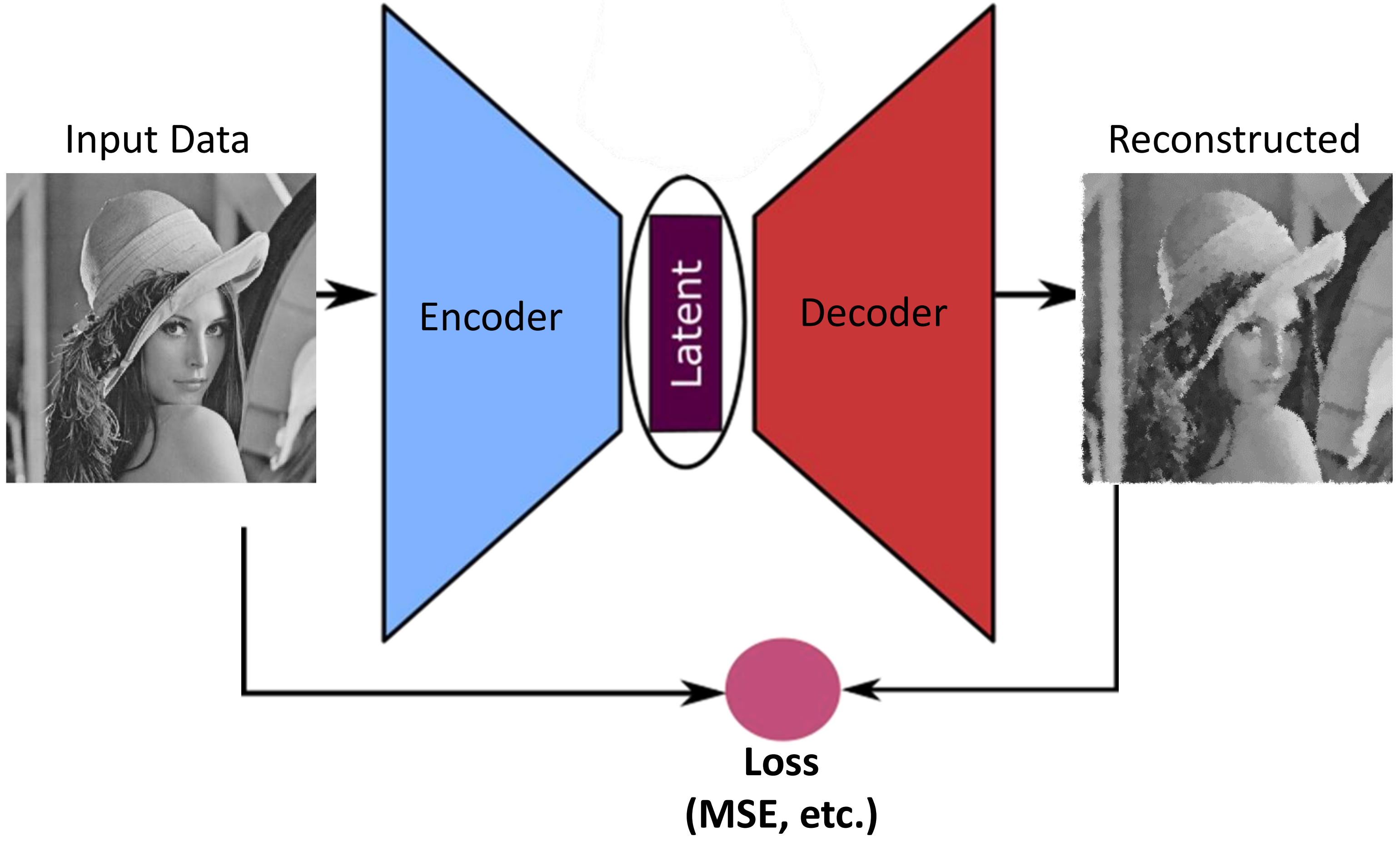

(h) 29 principal component

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

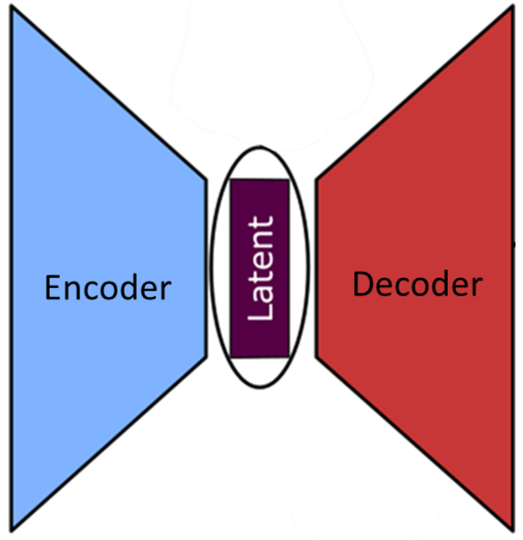
Autoencoders: Overview

- 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

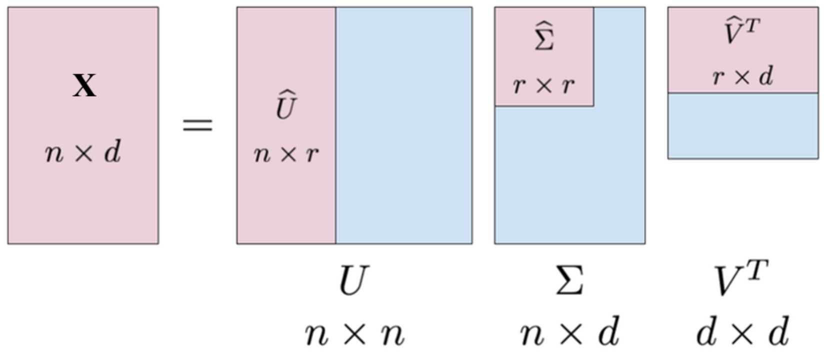


Autoencoders vs Classical Decompositions

- Autoencoders learned a fixed-sized representation
- Autoencoder decomposition is **nonlinear** and **learned iteratively**
- Encoder and decoder are not necessarily inverse – learned in parallel
- Latent space non necessarily orthogonal



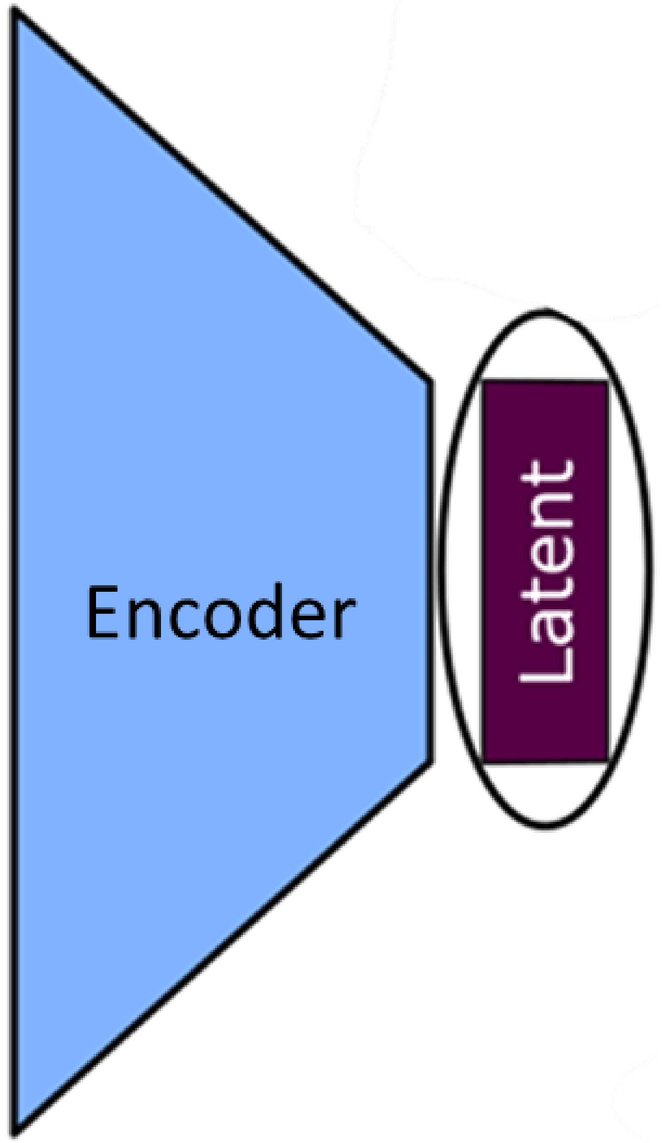
Autoencoder
- Fixed-Dimension (hyperparameter)
- Non-linear
- Trained/learned
- Non-orthogonal basis



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

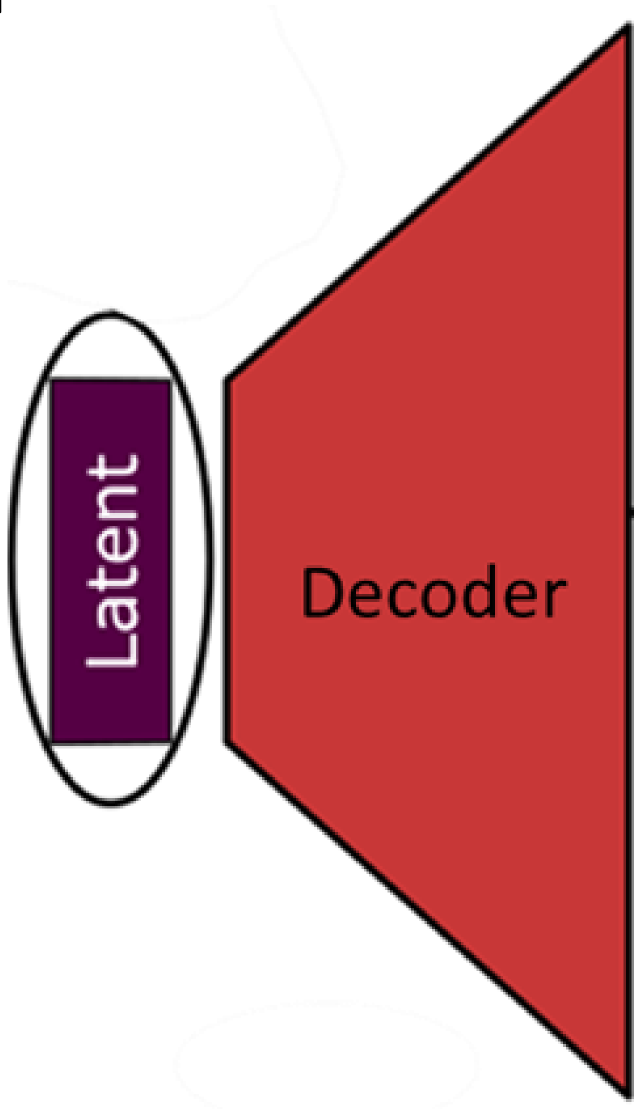
Autoencoder Structure: Encoder

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

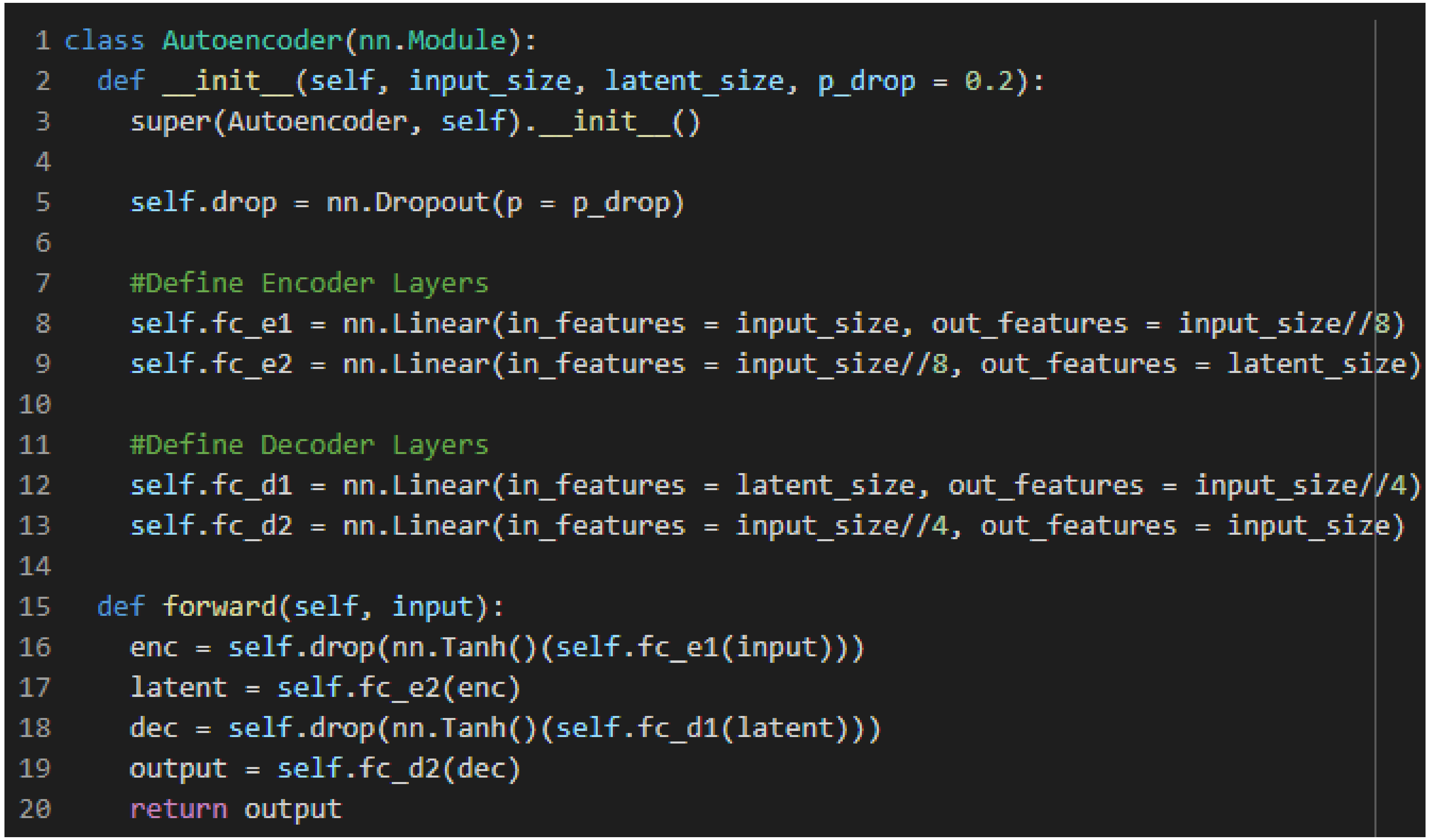


Autoencoder Structure: Decoder

- 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

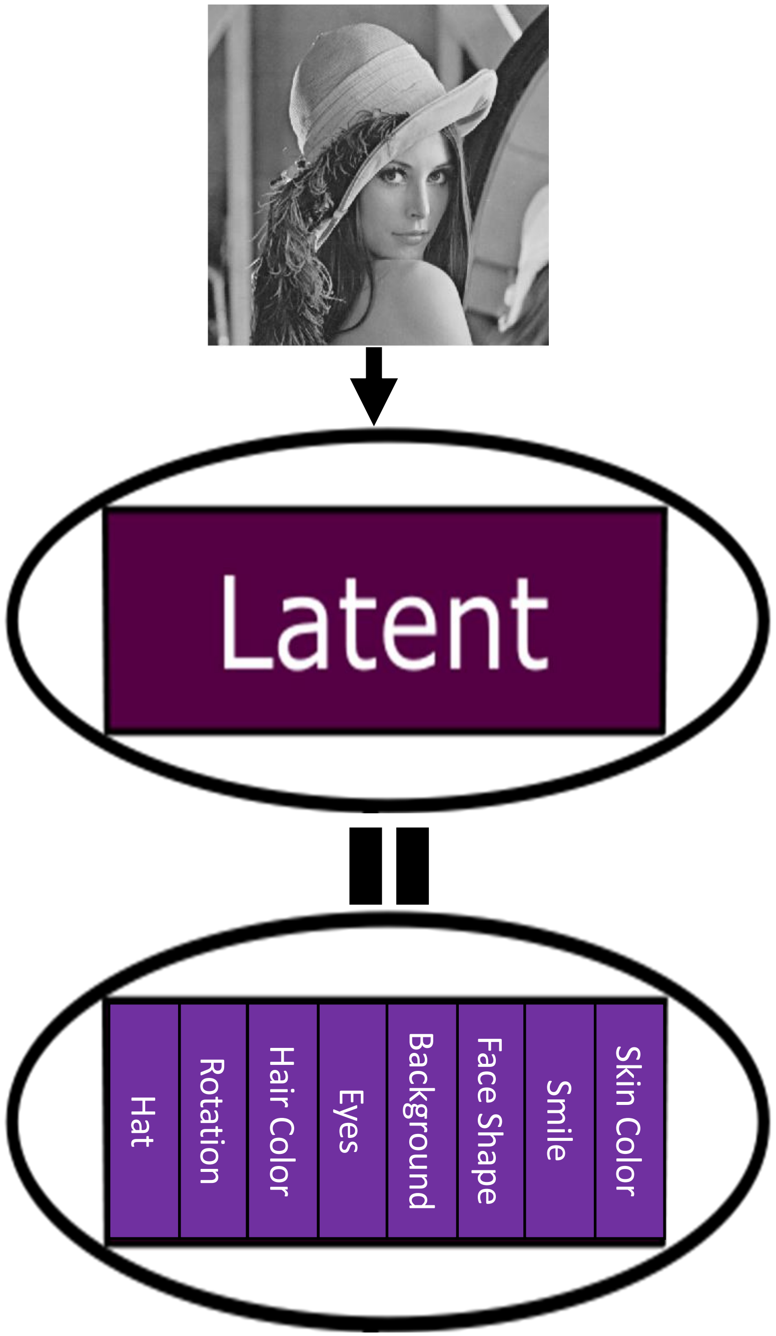


```python
class Autoencoder(nn.Module):
  def __init__(self, input_size, latent_size, p_drop = 0.2):
    super(Autoencoder, self).__init__()

    self.drop = nn.Dropout(p = p_drop)

    #Define Encoder Layers
    self.fc_e1 = nn.Linear(in_features = input_size, out_features = input_size//8)
    self.fc_e2 = nn.Linear(in_features = input_size//8, out_features = latent_size)

    #Define Decoder Layers
    self.fc_d1 = nn.Linear(in_features = latent_size, out_features = input_size//4)
    self.fc_d2 = nn.Linear(in_features = input_size//4, out_features = input_size)

  def forward(self, input):
    enc = self.drop(nn.Tanh()(self.fc_e1(input)))
    latent = self.fc_e2(enc)
    dec = self.drop(nn.Tanh()(self.fc_d1(latent)))
    output = self.fc_d2(dec)
    return output
```

Latent Space Representation

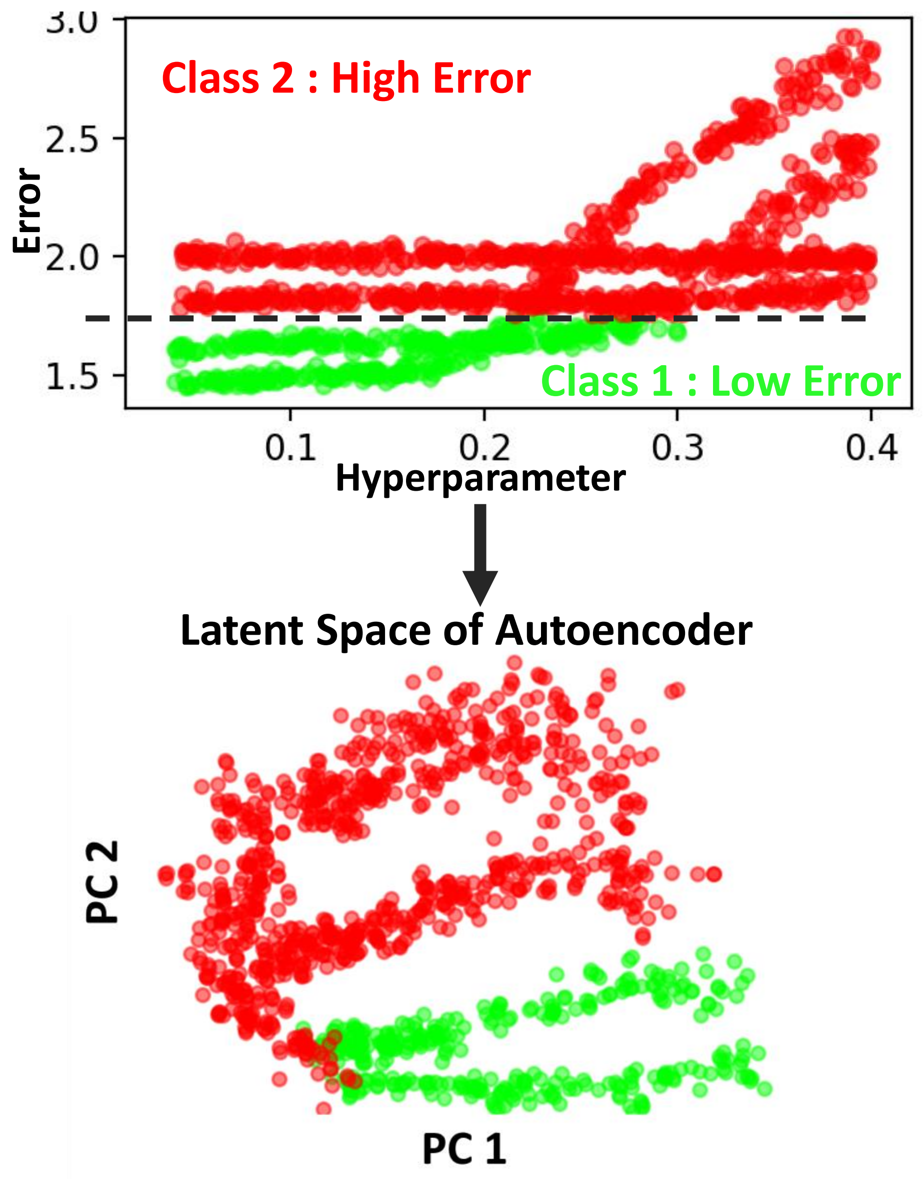
Latent Space Features

- 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



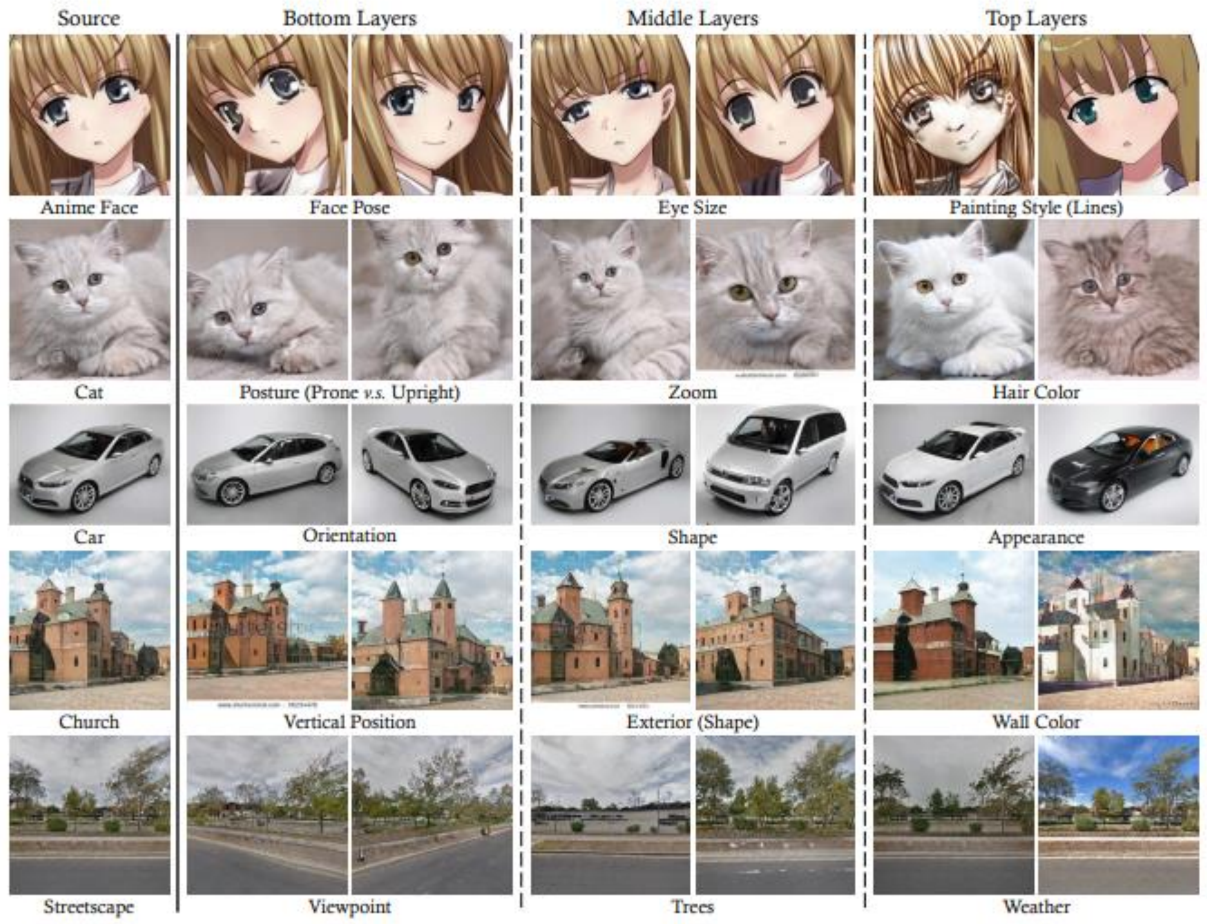
Clustering in Latent Space

- 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

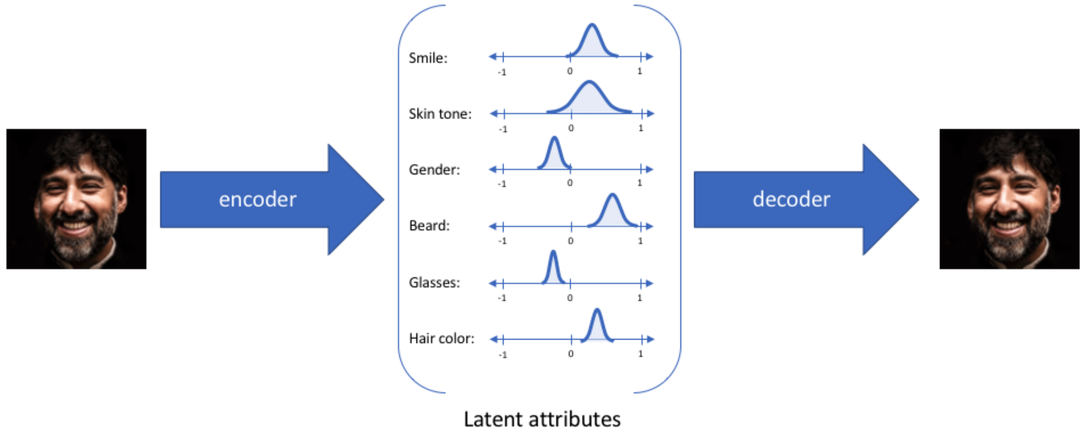


Source: https://arxiv.org/pdf/2007.06600.pdf

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



Source: https://www.jeremyjordan.me/variational-autoencoders/

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:

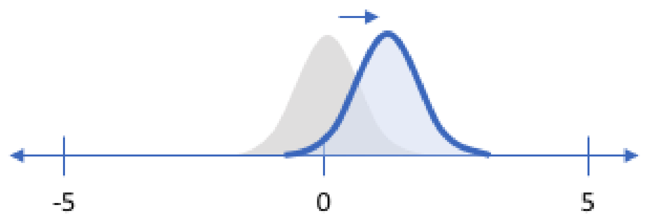
$$\mathcal{D}\left[Q(z)\|P(z|X)\right] = E_{z\sim Q}\left[\log Q(z) - \log P(z|X)\right]$$

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

Source: https://arxiv.org/pdf/1606.05908.pdf

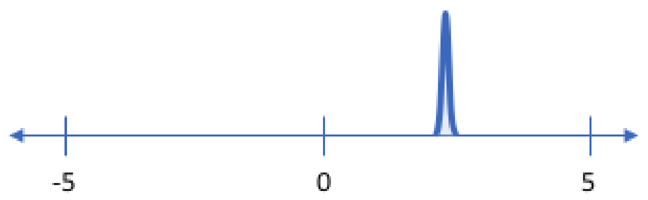
Mixed Losses: MSE and KL Divergence

Penalizing reconstruction loss encourages the distribution to describe the input



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

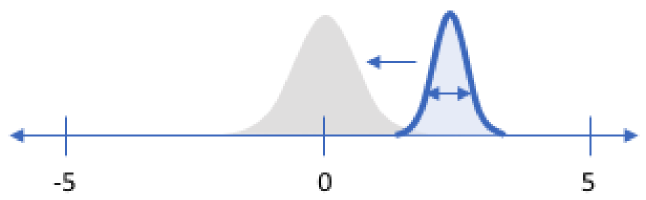
Without regularization, our network can "cheat" by learning narrow distributions



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

Penalizing KL divergence acts as a regularizing force

Attract distribution to have zero mean



Ensure sufficient variance to yield a smooth latent space

Source: https://www.jeremyjordan.me/variational-autoencoders/

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

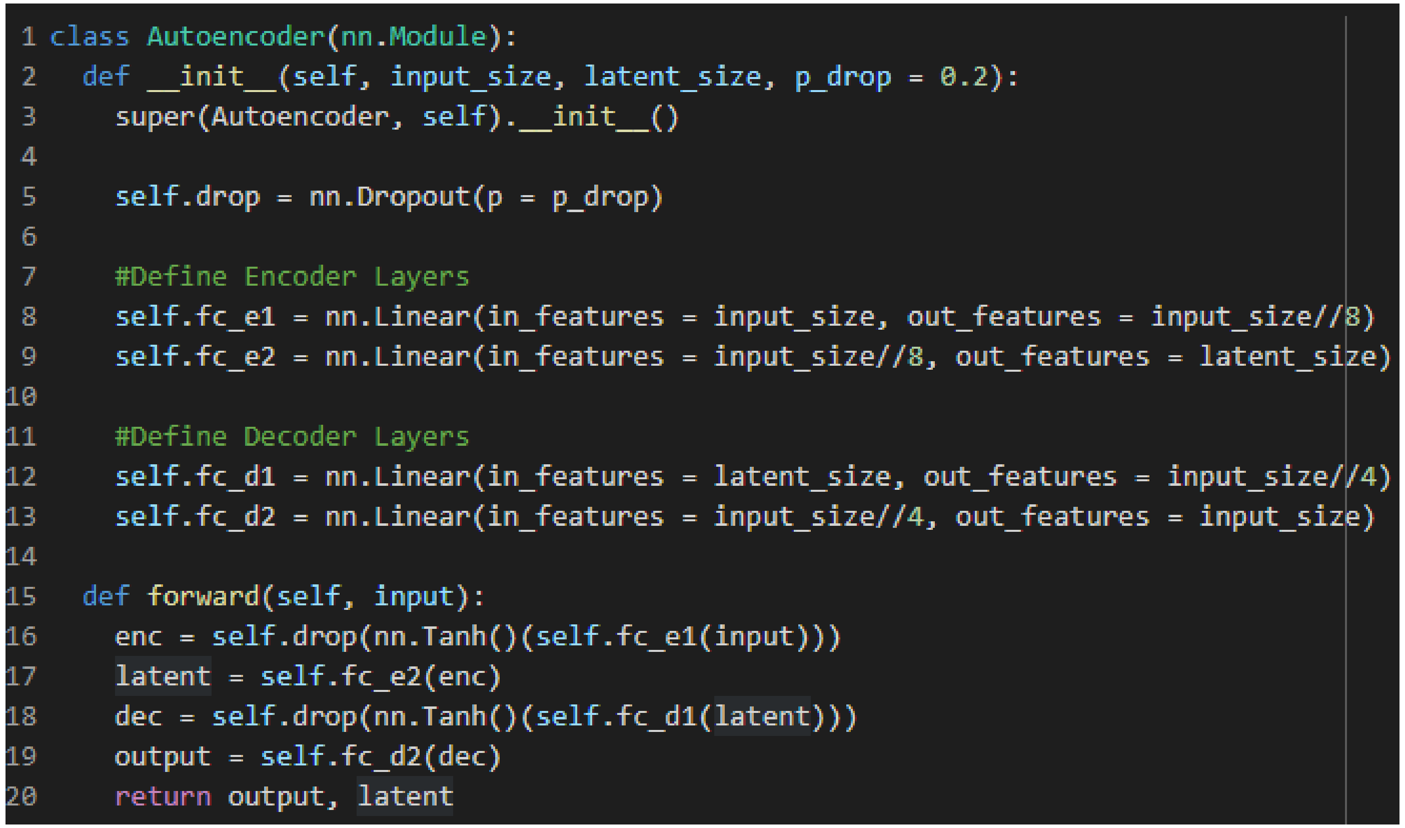
- **beta-VAE**: Learns interpretable factorization of data generative factors (https://openreview.net/forum?id=Sy2fzU9gl)
- **Deep Feature Consistent VAE**: enforces deep feature consistency to preserve spatial correlations of data (https://arxiv.org/abs/1610.00291)
- **DIP VAE:** Introduces an additional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

Image Source: https://github.com/AntixK/PyTorch-VAE

Example: Fully Connected AE on MNIST

Autoencoder Setup and hyperparameters

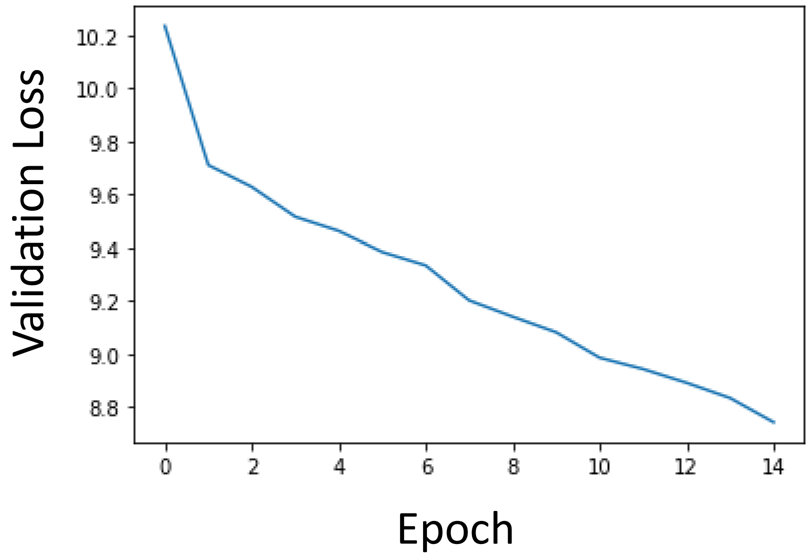
- Same autoencoder setup as in previous section, except it now also returns the latent values
- Used latent size of 32
- Input size for MNIST is 784 (when flattened)

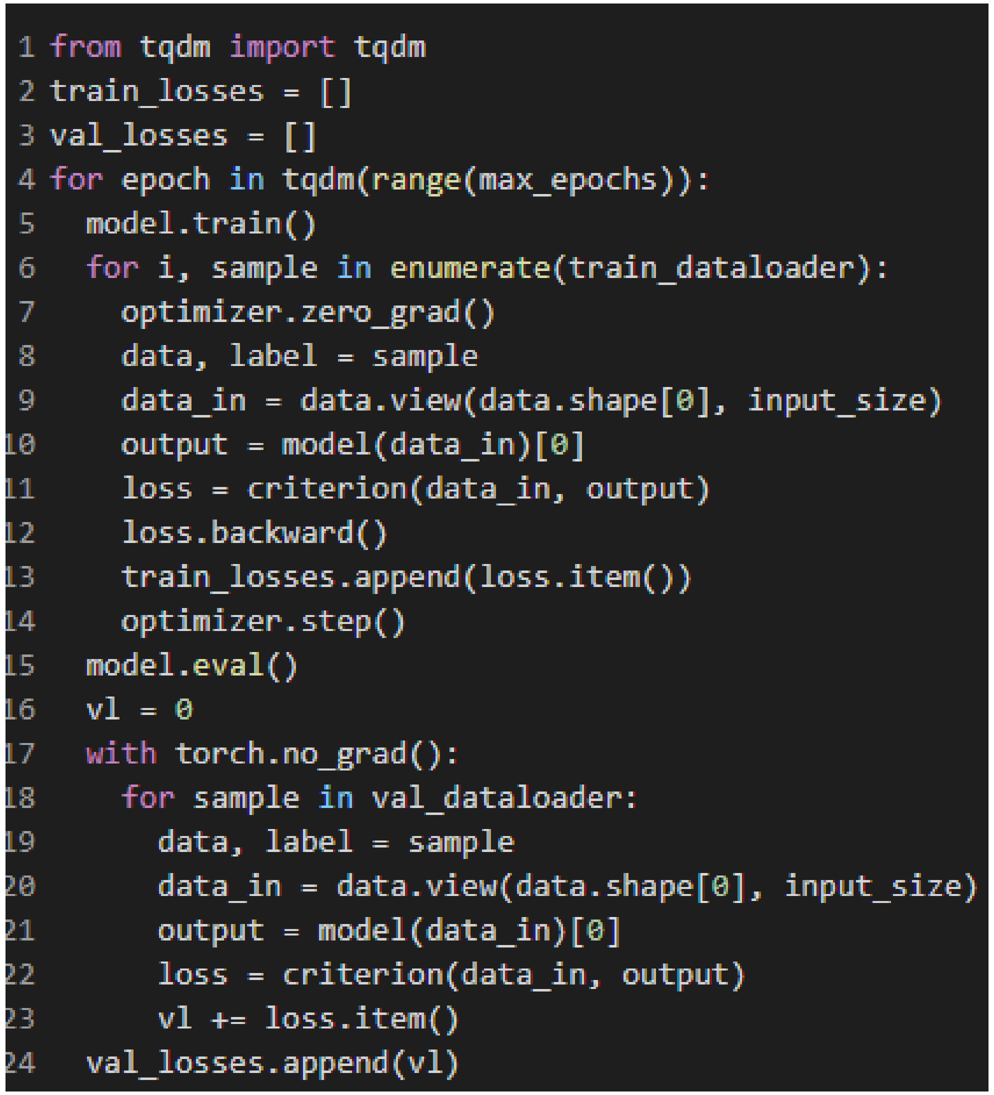


```python
class Autoencoder(nn.Module):
  def __init__(self, input_size, latent_size, p_drop = 0.2):
    super(Autoencoder, self).__init__()

    self.drop = nn.Dropout(p = p_drop)

    #Define Encoder Layers
    self.fc_e1 = nn.Linear(in_features = input_size, out_features = input_size//8)
    self.fc_e2 = nn.Linear(in_features = input_size//8, out_features = latent_size)

    #Define Decoder Layers
    self.fc_d1 = nn.Linear(in_features = latent_size, out_features = input_size//4)
    self.fc_d2 = nn.Linear(in_features = input_size//4, out_features = input_size)

  def forward(self, input):
    enc = self.drop(nn.Tanh()(self.fc_e1(input)))
    latent = self.fc_e2(enc)
    dec = self.drop(nn.Tanh()(self.fc_d1(latent)))
    output = self.fc_d2(dec)
    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

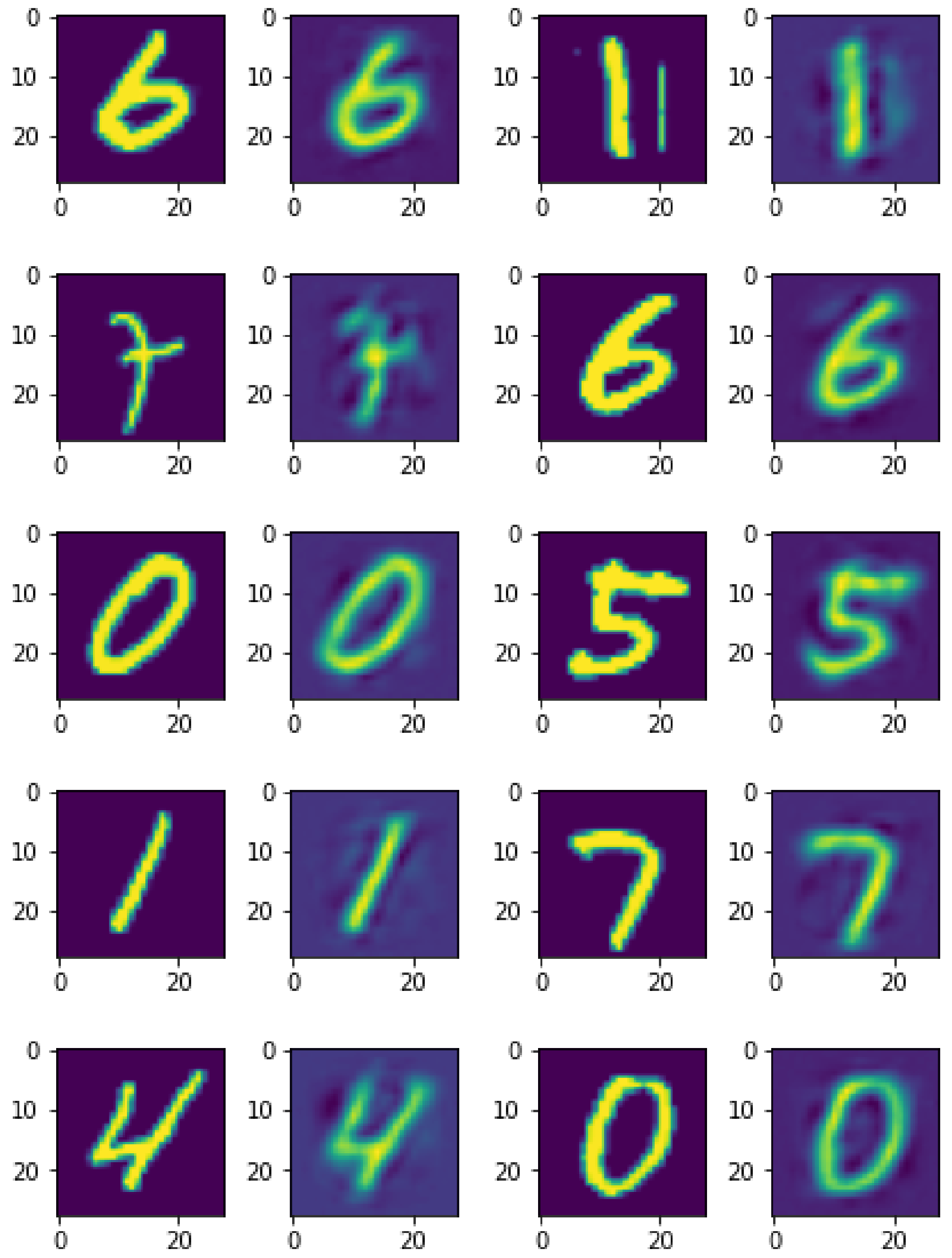




```
from tqdm import tqdm
train_losses = []
val_losses = []
for epoch in tqdm(range(max_epochs)):
  model.train()
  for i, sample in enumerate(train_dataloader):
    optimizer.zero_grad()
    data, label = sample
    data_in = data.view(data.shape[0], input_size)
    output = model(data_in)[0]
    loss = criterion(data_in, output)
    loss.backward()
    train_losses.append(loss.item())
    optimizer.step()
  model.eval()
  vl = 0
  with torch.no_grad():
    for sample in val_dataloader:
      data, label = sample
      data_in = data.view(data.shape[0], input_size)
      output = model(data_in)[0]
      loss = criterion(data_in, output)
      vl += loss.item()
  val_losses.append(vl)
```

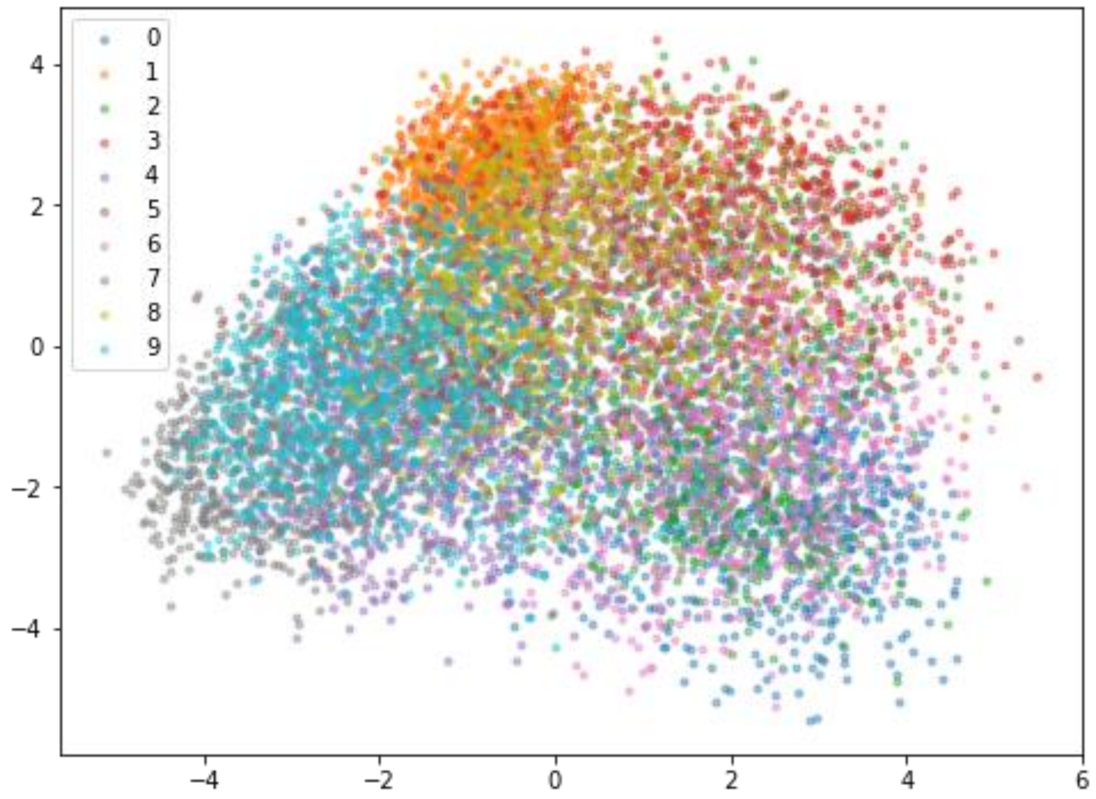
Reconstruction Results

- 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 not separated in latent space.



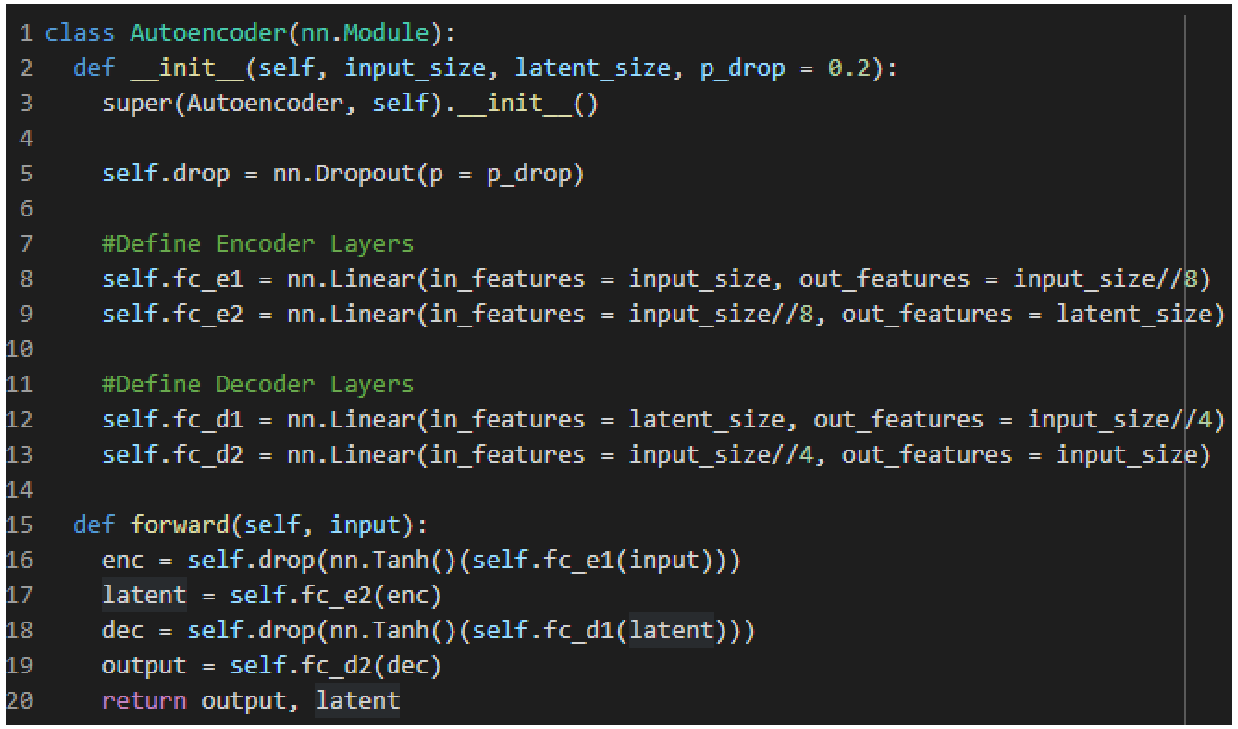
```python
test_dataloader = torch.utils.data.DataLoader(test_set,
                                              batch_size = 10000, shuffle = False)
for sample in test_dataloader:
  data = sample[0]
  data_in = data.view(data.shape[0], -1)
  latents = model(data_in)[1]
  labels = sample[1]
U, S, V =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

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

Assignment: Enhanced Autoencoder on MNIST

Your Task

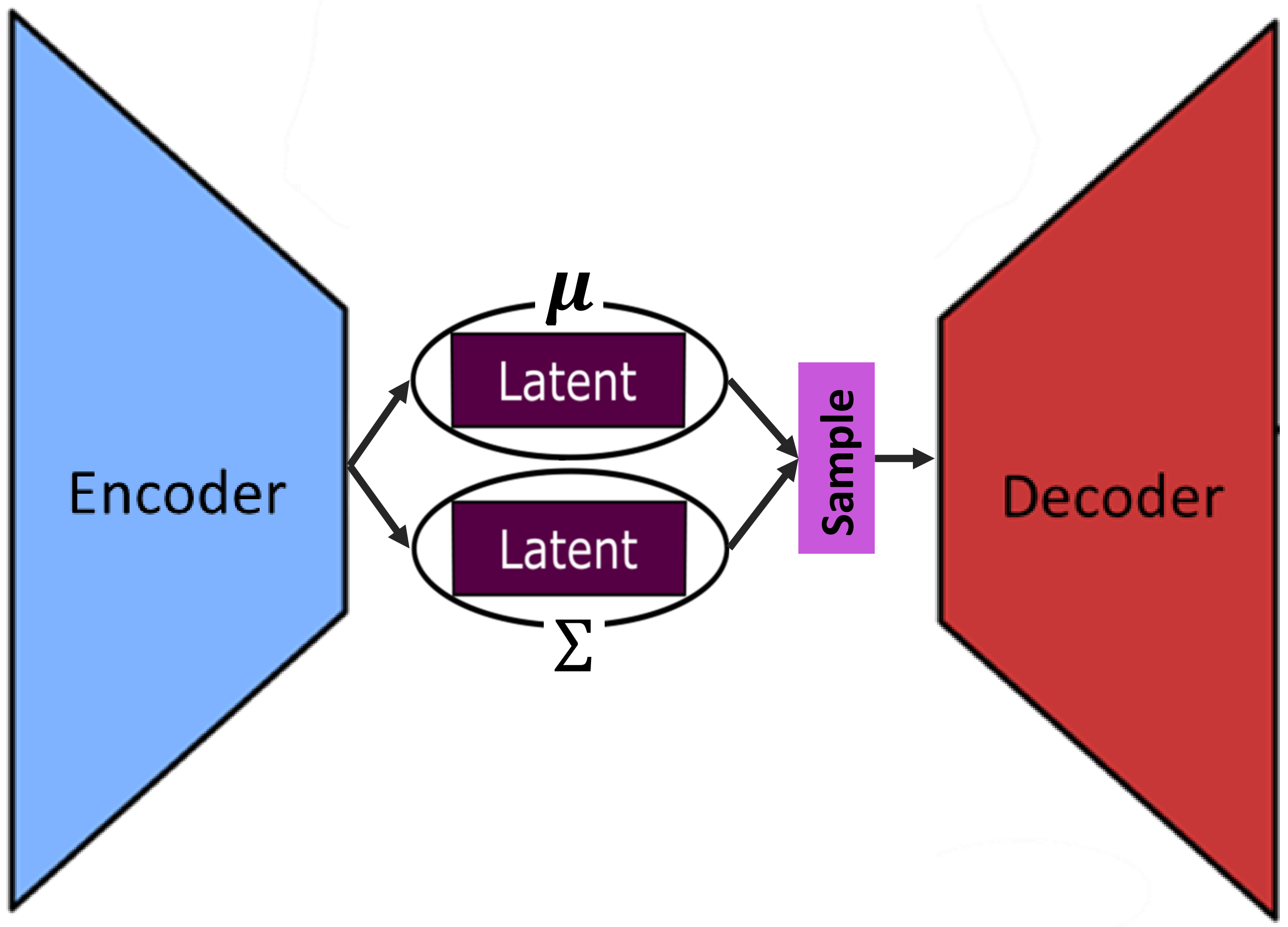
- Modify my autoencoder class in at least one of the following ways (EC for both):
  1. Change the architecture of the encoder and/or decoder from fully connected to either CNN or RNN
  2. Implement a variational autoencoder (see next slides)
- Plot examples of input and reconstructed images
- Show clustering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



```python
class Autoencoder(nn.Module):
  def __init__(self, input_size, latent_size, p_drop = 0.2):
    super(Autoencoder, self).__init__()

    self.drop = nn.Dropout(p = p_drop)

    #Define Encoder Layers
    self.fc_e1 = nn.Linear(in_features = input_size, out_features = input_size//8)
    self.fc_e2 = nn.Linear(in_features = input_size//8, out_features = latent_size)

    #Define Decoder Layers
    self.fc_d1 = nn.Linear(in_features = latent_size, out_features = input_size//4)
    self.fc_d2 = nn.Linear(in_features = input_size//4, out_features = input_size)

  def forward(self, input):
    enc = self.drop(nn.Tanh()(self.fc_e1(input)))
    latent = self.fc_e2(enc)
    dec = self.drop(nn.Tanh()(self.fc_d1(latent)))
    output = self.fc_d2(dec)
    return output, latent
```

VAE Implementation

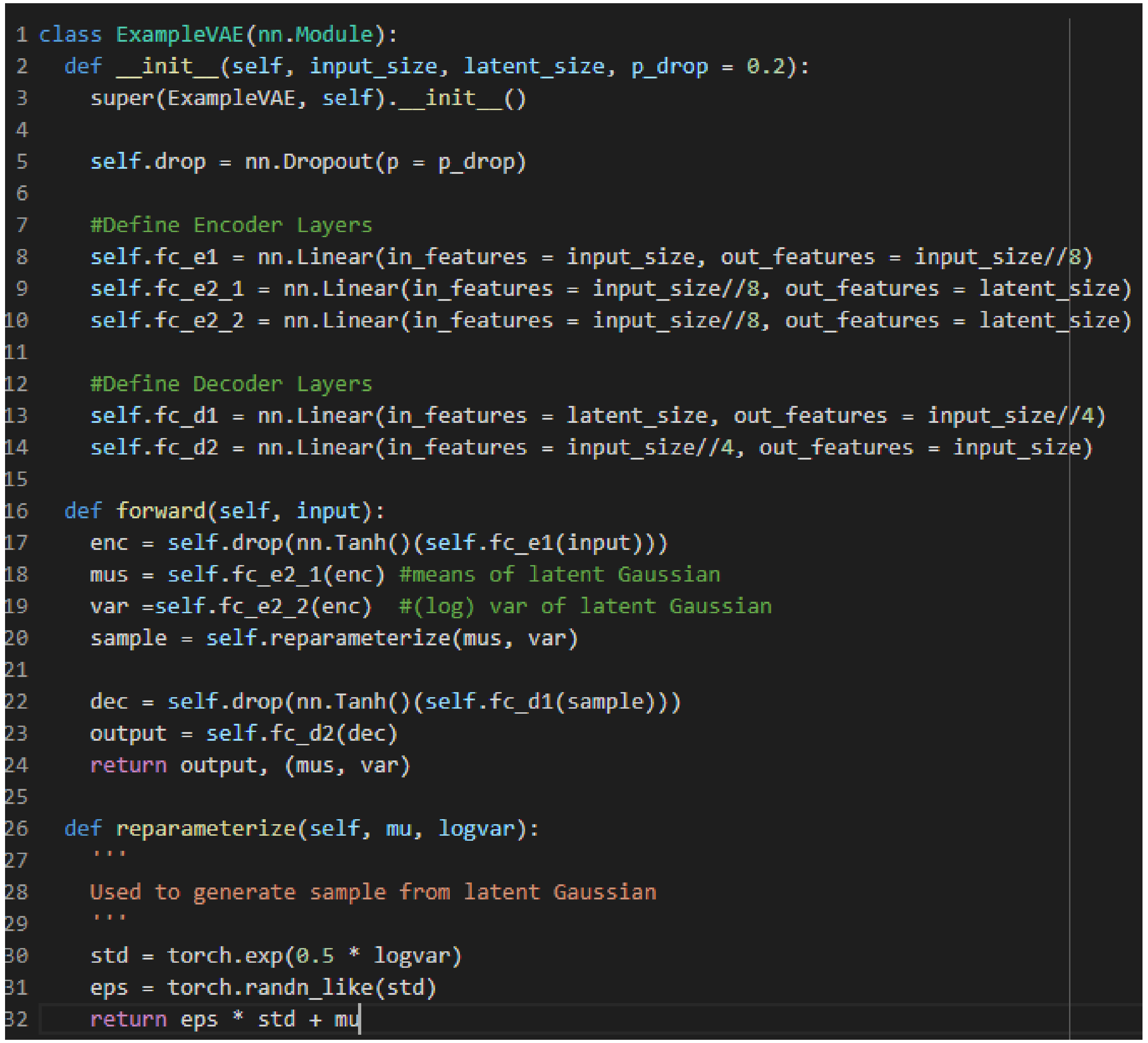
- For the VAE, if you have a latent size of *n*, you will have two separate latent spaces of size *n* which will represent:
    1. The distribution means (**μ**)
    2. The distribution variances (**Σ**)



VAE Implementation

- Previous AE can be changed to VAE by adding another parallel layer in encoder and adding "reparametrize" function

- For loss function, implement KL divergence using standard normal as your prior (should be able to do this manually)



```python
class ExampleVAE(nn.Module):
  def __init__(self, input_size, latent_size, p_drop = 0.2):
    super(ExampleVAE, self).__init__()

    self.drop = nn.Dropout(p = p_drop)

    #Define Encoder Layers
    self.fc_e1 = nn.Linear(in_features = input_size, out_features = input_size//8)
    self.fc_e2_1 = nn.Linear(in_features = input_size//8, out_features = latent_size)
    self.fc_e2_2 = nn.Linear(in_features = input_size//8, out_features = latent_size)

    #Define Decoder Layers
    self.fc_d1 = nn.Linear(in_features = latent_size, out_features = input_size//4)
    self.fc_d2 = nn.Linear(in_features = input_size//4, out_features = input_size)

  def forward(self, input):
    enc = self.drop(nn.Tanh()(self.fc_e1(input)))
    mus = self.fc_e2_1(enc) #means of latent Gaussian
    var =self.fc_e2_2(enc)  #(log) var of latent Gaussian
    sample = self.reparameterize(mus, var)

    dec = self.drop(nn.Tanh()(self.fc_d1(sample)))
    output = self.fc_d2(dec)
    return output, (mus, var)

  def reparameterize(self, mu, logvar):
    '''
    Used to generate sample from latent Gaussian
    '''
    std = torch.exp(0.5 * logvar)
    eps = torch.randn_like(std)
    return eps * std + mu
```