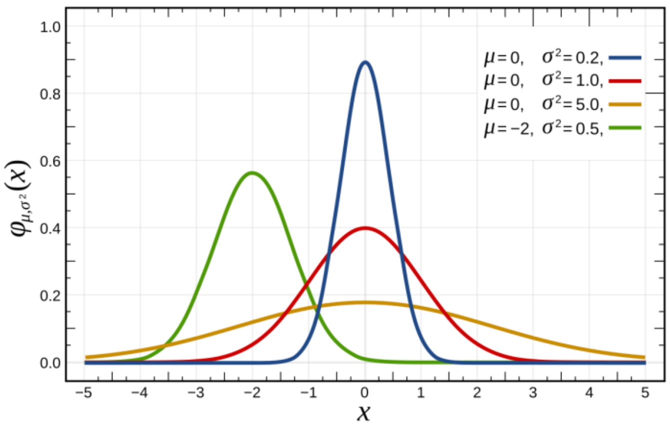
Advanced Machine Learning
Generative Model

Yu Wang
Assistant Professor
Department of Computer Science
University of Oregon

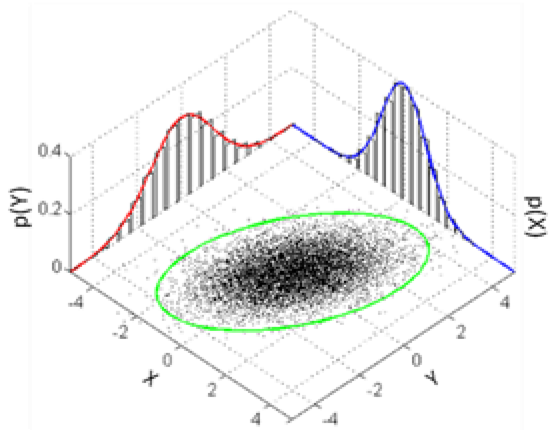
Summary

1D Gaussian Distribution

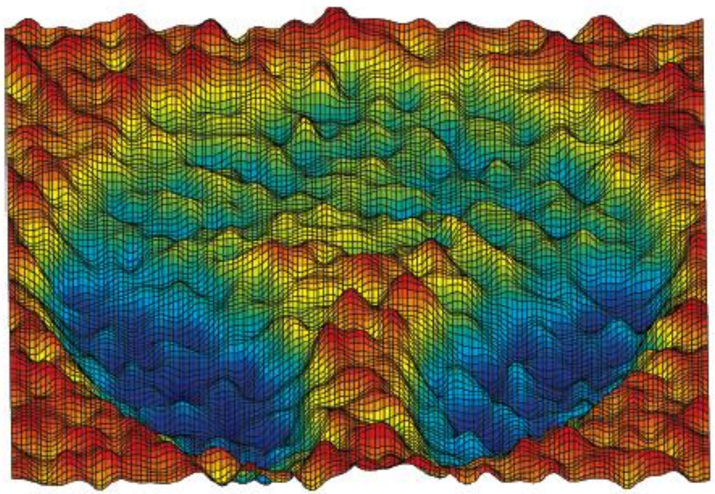


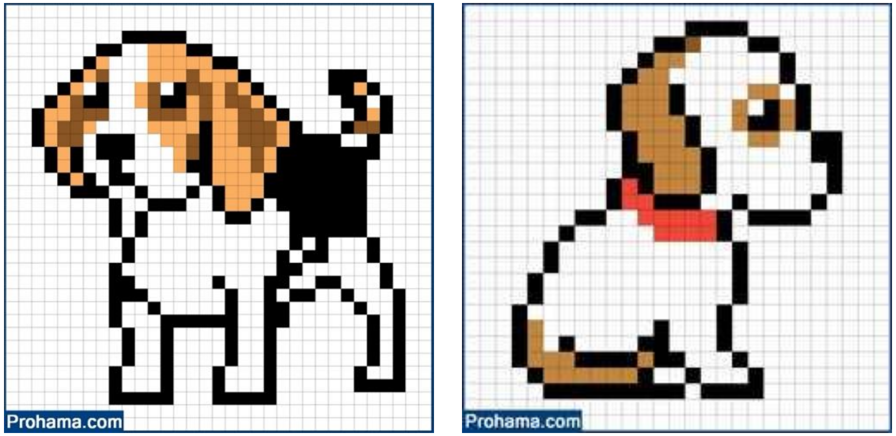
$\mathbb{R}$

2D Gaussian Distribution



$\mathbb{R}^2$





$\mathbb{R}^{256\times256}$

Summary

Probability distribution of the **objective** based on the **observed data**

- **Machine Learning Methods**

$$\{x_i\}_{i=1}^{N} \xrightarrow{\text{Good Model}} P(x) \xrightarrow{\text{Good Data}} x$$

  - Gaussian Kernel Density Estimation
  - Gaussian Mixture Models

Using existing function to estimate what you do not know that can best fit your observation

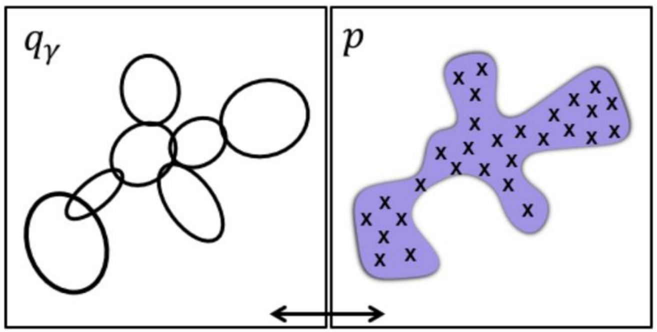
- **Deep Learning Methods**
  - Auto-Encoder (AE)
  - Variational AE (LLM is actually a VAE)
  - Generative Adversarial Network
  - Diffusion Model

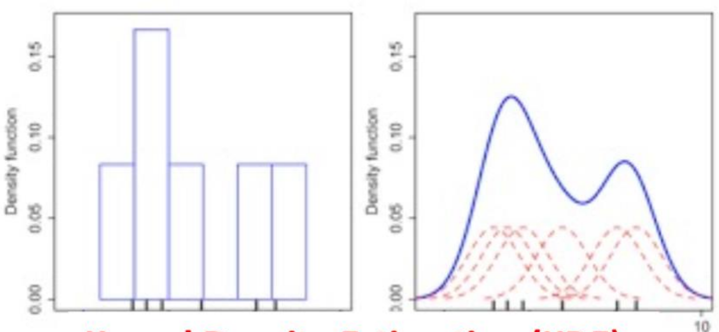
Using learnable function to estimate what you do not know that can best fit your observation

Summary

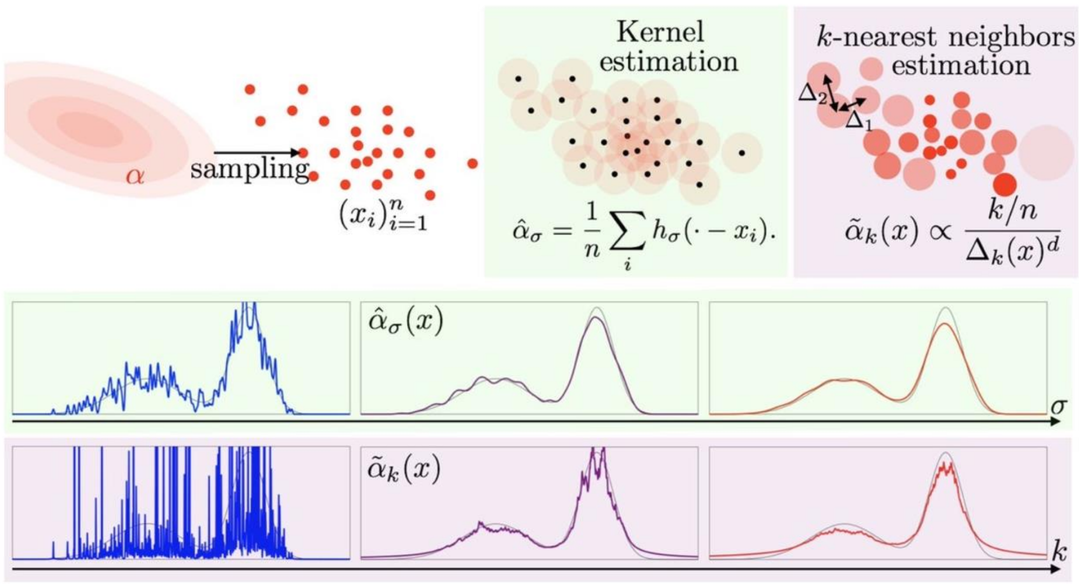
Using existing function to estimate what you do not know that can best fit your observation

Gaussian Mixture Models





Kernel Density Estimation (KDE)

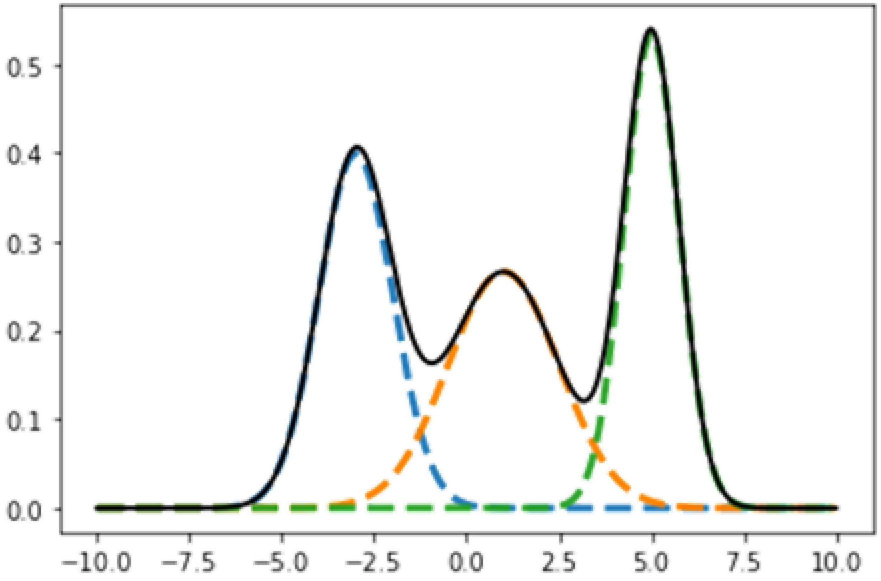


KNN Density Estimation

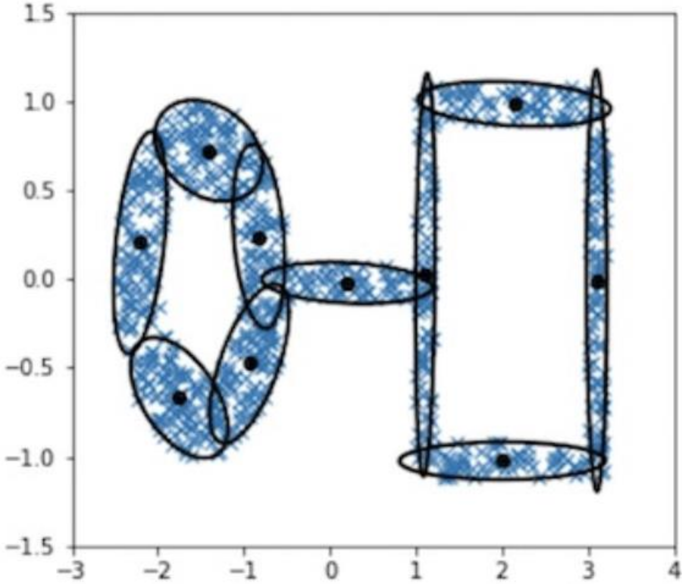
What is the problem?

Problem?

Using existing function to estimate what you do not know that can best fit your observation



Three bumps but I just give you two different gaussian

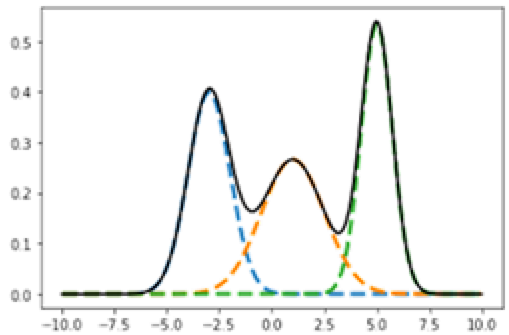


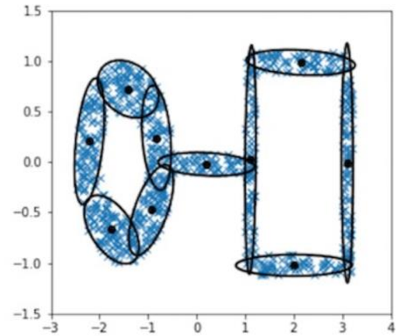
I just give you two different gaussian.

All you know for modeling what you do not know is fixed. But how do you know those fixed things is able to model the unknown thing?

Problem?

Using **existing function** to **estimate what you do not know** that **can best fit your observation**





$\mathbb{R}^1, \mathbb{R}^2$

$\downarrow$

$\mathbb{R}^{256\times256}$

What you have is some low-dimensional data

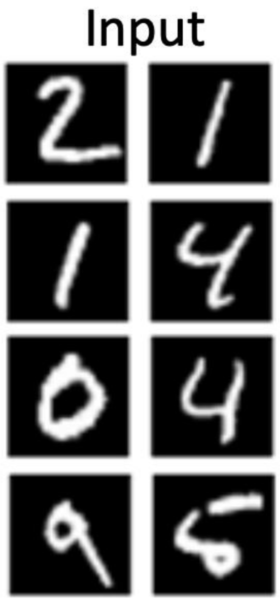
But what you want to model is some high-dimensional data, how it could be?

Problem?

What we want: model any data distribution

How to transform any data distribution to low dimensional data?

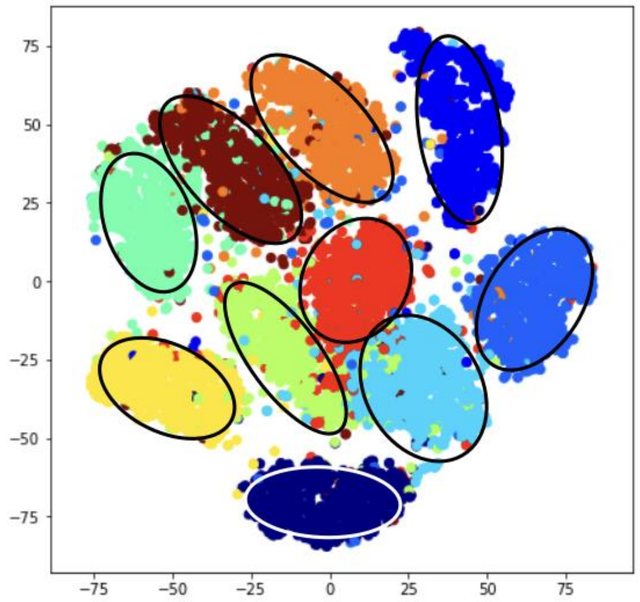
What we have: kernel density estimation to estimate low dimensional PDF



Someway to transform

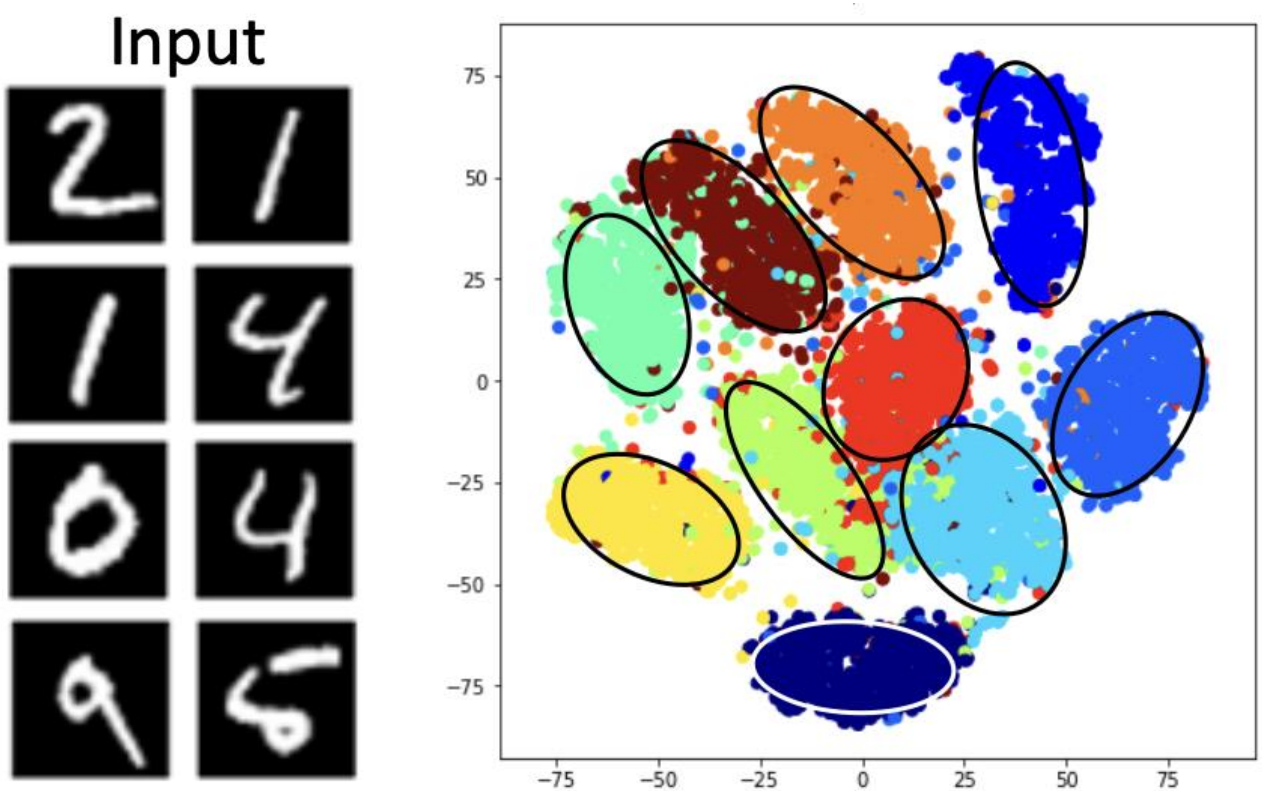
Transform back

Kernel Density Estimation



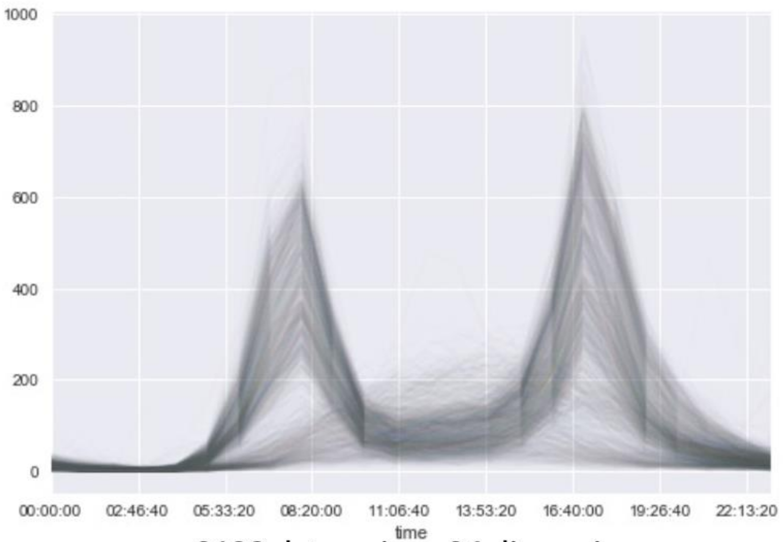
Observation

A key assumption: high-dimensional data lies on the low-dimensional manifold space

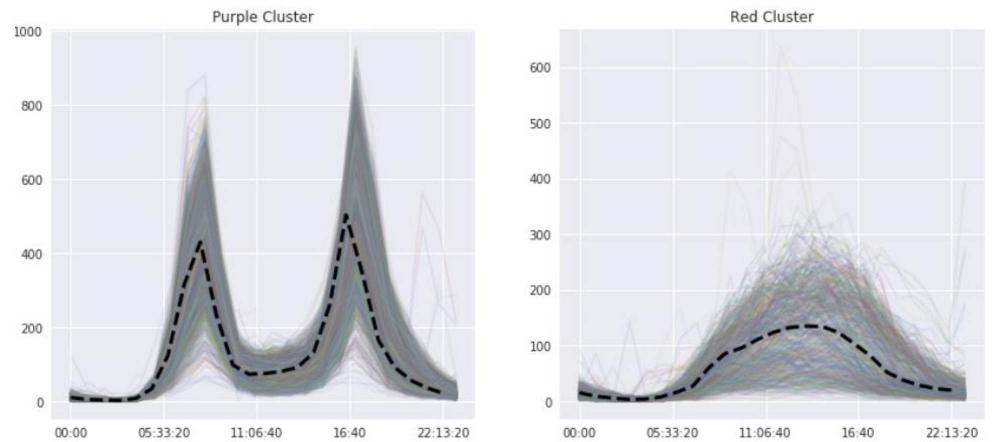


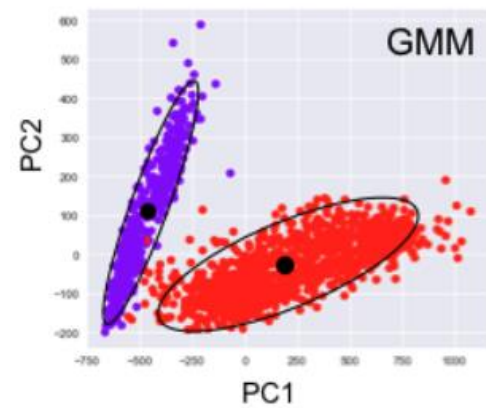
Observation

A key assumption: high-dimensional data lies on the low-dimensional manifold space



2128 data points, 24 dimensions





Summary

Probability distribution of the **objective** based on the **observed data**

$\{x_i\}_{i=1}^{N} \xrightarrow{\text{Good Model}} P(x) \xrightarrow{\text{Good Data}} x$

- **Machine Learning Methods**
  - Gaussian Kernel Density Estimation
  - Gaussian Mixture Models

Using existing function to estimate what you do not know that can best fit your observation

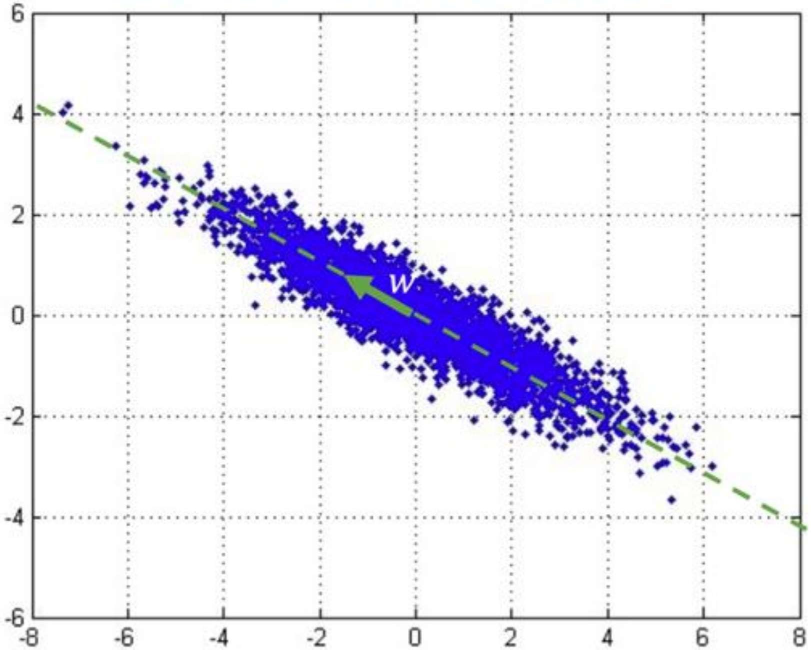
PCA Dimensional Reduction

- **Deep Learning Methods**
  - Auto-Encoder (AE)
  - Variational AE (LLM is actually a VAE)
  - Generative Adversarial Network
  - Diffusion Model

Using learnable function to estimate what you do not know that can best fit your observation

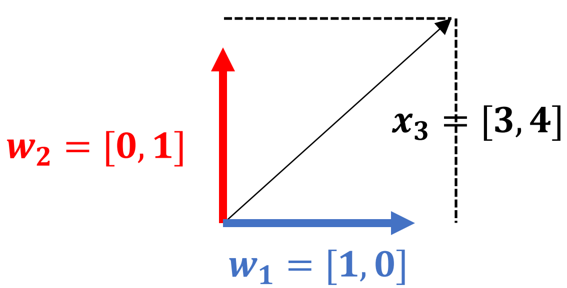
PCA - Variance Maximization

2D Gaussian dataset



What would be a good reduction?

- Find w such that it maximizes the variance of the projected data
- Find w such that it minimizes the reconstruction error

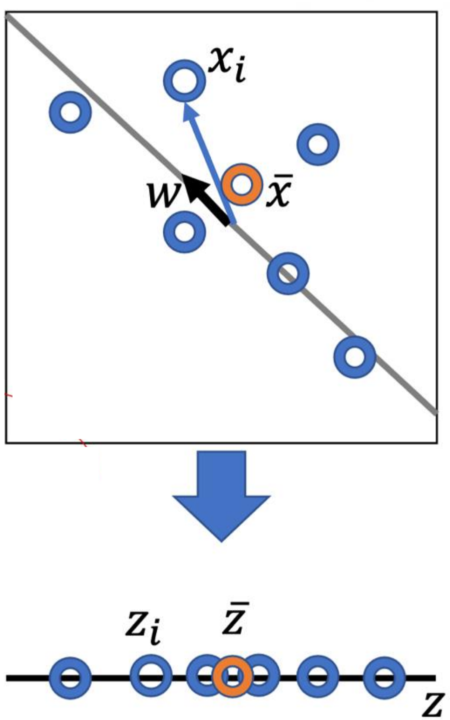


$$\cos\theta = \frac{w_1^T x_3}{|w_1|_2 |x_3|_2}$$

$$\cos\theta |x_3|_2 = \frac{w_1^T x_3}{|w_1|_2} = w_1^T x_3$$

PCA - Variance Maximization

Find w such that it maximizes the variance of the projected data



$$var = \frac{1}{N}\sum_{i=1}^{N}(w^T(x_i - \bar{x}))^2 = \frac{1}{N}\sum_{i=1}^{N} w^T(x_i - \bar{x})(x_i - \bar{x})^T w$$

$$= w^T\left(\frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})(x_i - \bar{x})^T\right)w$$

$$= w^T S w$$

Therefore maximizing the variance can be written as:

Constrained Optimization

$$\max_w \; w^T S w \qquad s.t. \; w^T w = 1 \quad \xrightarrow{\text{KKT Condition}} \quad Sw = \lambda w$$

$$L(w, \alpha) = w^T S w + \alpha(w^T w - 1)$$

$$\nabla_w L(w, \alpha) = 2Sw + 2\alpha w = 0$$

$$\nabla_w L(w, \alpha) = Sw - \lambda w = 0 \qquad\qquad Sw = \lambda w$$

PCA - Variance Maximization

Some characteristics of the eigenvectors:

- $\|v_i\| = 1$
- $v_i^T v_j = 0, \ \ \forall i \neq j$

Covariance matrix is a real and symmetric matrix (in fact it is PSD) therefore it can be uniquely decomposed via:

$$S = \sum_i \lambda_i v_i v_i^T$$

Multiply both sides by v_k:

$$S v_k = \lambda_k v_k$$

Therefore w should be the first eigenvector of S

$$Sw = \lambda w$$

$$\boldsymbol{\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n}$$

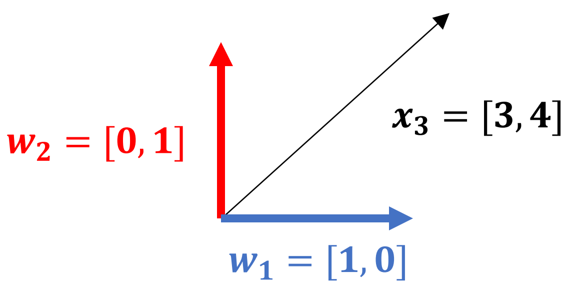
$$\boldsymbol{w_1 \geq w_2 \geq \cdots \geq w_n}$$

$$\boldsymbol{w_1^T x_1 = \sigma_1}$$

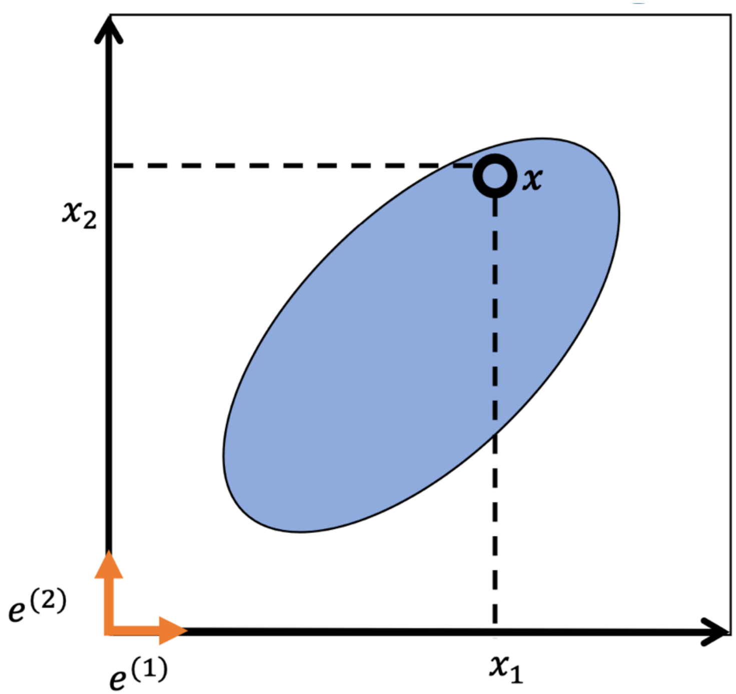
$$\boldsymbol{w_2^T x_1 = \sigma_2}$$

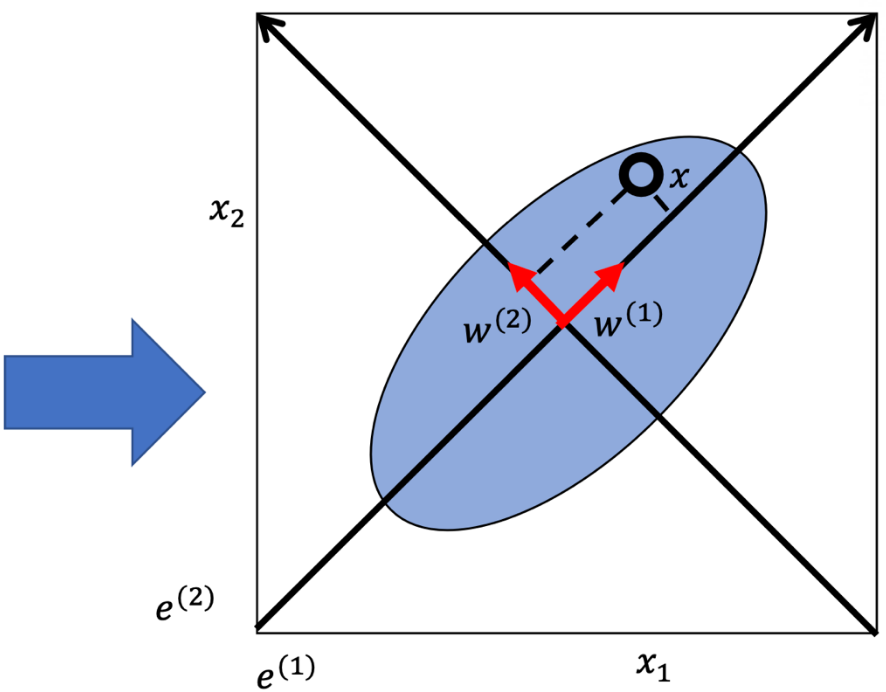
$$\vdots$$

$$\boldsymbol{w_n^T x_1 = \sigma_n}$$



PCA - Variance Maximization





$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \Rightarrow x = x_1 \underbrace{\begin{bmatrix} 1 \\ 0 \end{bmatrix}}_{e^{(1)}} + x_2 \underbrace{\begin{bmatrix} 0 \\ 1 \end{bmatrix}}_{e^{(2)}}$$

$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \Rightarrow x = \bar{x} + \left(x^T w^{(1)}\right) w^{(1)} + \left(x^T w^{(2)}\right) w^{(2)}$$

PCA - Variance Maximization

[11]:

```python
import torch
import numpy as np
import matplotlib.pyplot as plt
from torchvision import datasets, transforms

# Load MNIST dataset
transform = transforms.ToTensor()
mnist_data = datasets.MNIST(root='./data', train=True, download=True, transform=transform)
images = mnist_data.data.float()
labels = mnist_data.targets
```

[13]:

```python
# Flatten images to vectors of size 784
N, H, W = images.shape  # N=60000, H=28, W=28
X = images.view(N, H*W)  # shape: (60000, 784)

# Normalize data
mean_image = X.mean(dim=0)
X_centered = X - mean_image

# Compute covariance matrix
cov = (X_centered.T @ X_centered) / (N-1)

# Eigen-decomposition
eigenvalues, eigenvectors = torch.linalg.eigh(cov)
# Sort eigenvalues and eigenvectors in descending order
eigenvalues, indices = torch.sort(eigenvalues, descending=True)
eigenvectors = eigenvectors[:, indices]
```

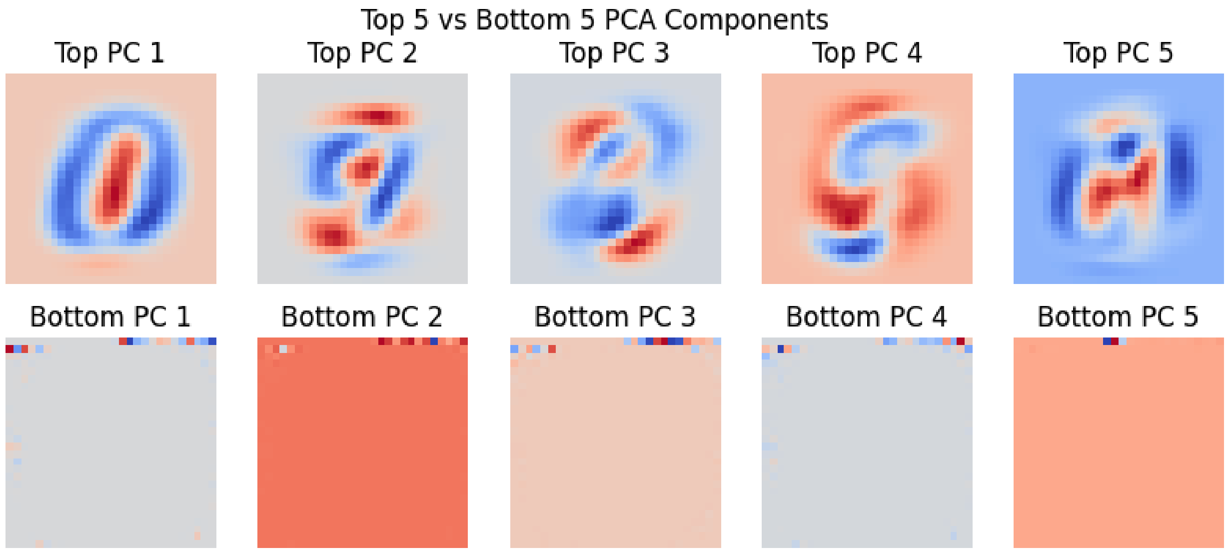
[14]:

```python
### Visualization 1: Original MNIST Dataset
fig, axes = plt.subplots(1, 10, figsize=(10, 2))
for i in range(10):
    axes[i].imshow(X[i].reshape(28, 28), cmap='gray')
    axes[i].axis('off')
plt.suptitle('Original MNIST Samples')
plt.show()
```

Original MNIST Samples

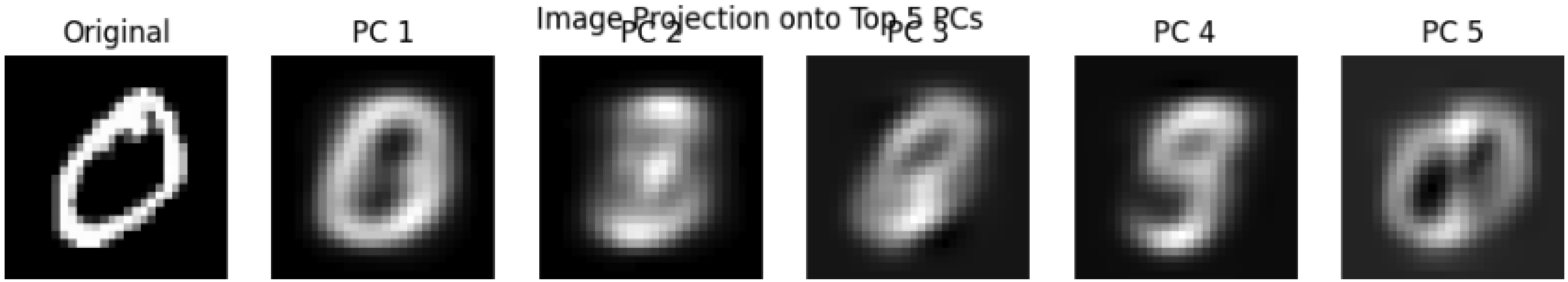
PCA - Variance Maximization

```
### Visualization 2: Top 5 and Bottom 5 PCA Components
fig, axes = plt.subplots(2, 5, figsize=(10, 4))
for i in range(5):
    top_comp = eigenvectors[:, i].reshape(28, 28)
    bottom_comp = eigenvectors[:, -i-1].reshape(28, 28)

    axes[0, i].imshow(top_comp, cmap='coolwarm')
    axes[0, i].set_title(f'Top PC {i+1}')
    axes[0, i].axis('off')

    axes[1, i].imshow(bottom_comp, cmap='coolwarm')
    axes[1, i].set_title(f'Bottom PC {i+1}')
    axes[1, i].axis('off')
plt.suptitle('Top 5 vs Bottom 5 PCA Components')
plt.show()
```



PCA - Variance Maximization

```
### Visualization 4: Projecting a Specific Image onto PC1–5
sample_idx = 1  # Choose an image
test_image = X[sample_idx]
fig, axes = plt.subplots(1, 6, figsize=(12, 2))
axes[0].imshow(test_image.reshape(28, 28), cmap='gray')
axes[0].set_title('Original')
axes[0].axis('off')

for i in range(5):
    weight = torch.dot(test_image - mean_image, eigenvectors[:, i])
    recon = weight * eigenvectors[:, i] + mean_image
    axes[i+1].imshow(recon.reshape(28, 28), cmap='gray')
    axes[i+1].set_title(f'PC {i+1}')
    axes[i+1].axis('off')
plt.suptitle('Image Projection onto Top 5 PCs')
plt.show()
```



PCA - Variance Maximization

Input x

=

Mean $\bar{x}$

+ 341.6*

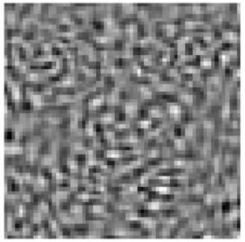
$w^{(1)}$

- 12.7*

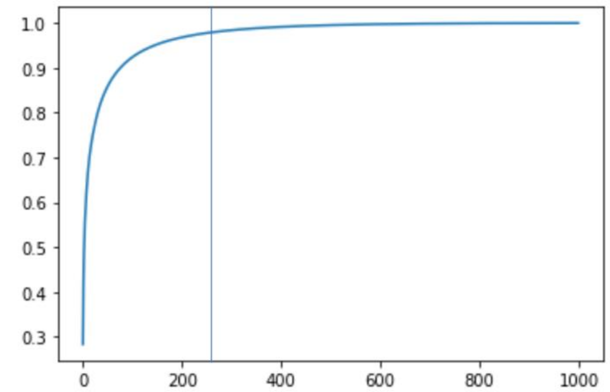
$w^{(2)}$

+... + 12.2*

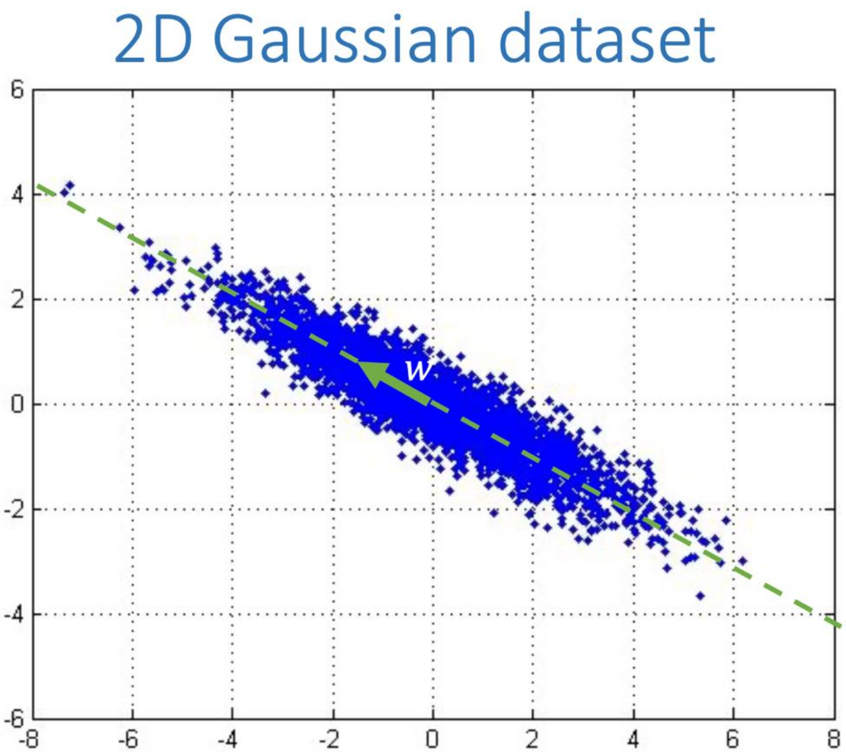
$w^{(1000)}$



Reconstruction as a function of number of PC components



Reconstruction Loss for PCA



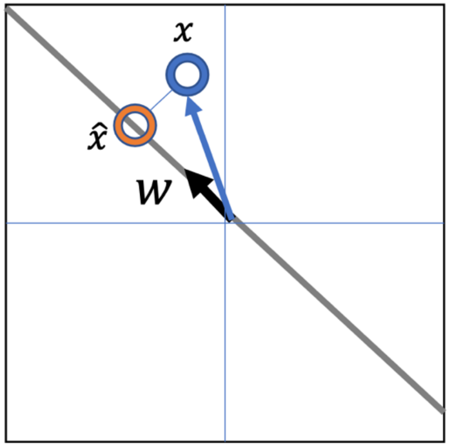
Reduction to 1D

What would be a good reduction?

- Find w such that it maximizes the variance of the projected data
- **Find w such that it minimizes the reconstruction error**

Reconstruction Loss for PCA

Recall how to project a vector x onto the subspace spanned by the unit vector w (i.e., onto a line).



$$\hat{x} = ww^T x \qquad ||\boldsymbol{w}|| = \mathbf{1}$$

Now lets write the reconstruction error:

$$Reconstruction\ Error = \frac{1}{N}\sum_{i=1}^{N} ||\hat{x}^{(i)} - x^{(i)}||^2 = \frac{1}{N}\sum_{i=1}^{N} ||ww^T x^{(i)} - x^{(i)}||^2$$

$x^{(i)} \in \mathbb{R}^{2*1}$ $\qquad$ $w^T \in \mathbb{R}^{1*2}$ $\qquad$ $w \in \mathbb{R}^{2*1}$

Reconstruction Loss for PCA

Minimize the reconstruction error:

$$\min_W \frac{1}{N} \sum_{i=1}^{N} \left\| WW^T x^{(i)} - x^{(i)} \right\|^2$$

This can be expanded as:

$$= \frac{1}{N} \sum_{i=1}^{N} \left(WW^T x^{(i)} - x^{(i)} \right)^T \left(WW^T x^{(i)} - x^{(i)} \right)$$

Expanding the quadratic form:

$$= \frac{1}{N} \sum_{i=1}^{N} \left[(x^{(i)})^T WW^T WW^T x^{(i)} - 2(x^{(i)})^T WW^T x^{(i)} + (x^{(i)})^T x^{(i)} \right]$$

Assuming WW^T is an orthogonal projection matrix (so $WW^T WW^T = WW^T$), we simplify:

$$= \frac{1}{N} \sum_{i=1}^{N} \left[(x^{(i)})^T WW^T x^{(i)} - 2(x^{(i)})^T WW^T x^{(i)} + (x^{(i)})^T x^{(i)} \right]$$

$$= \frac{1}{N} \sum_{i=1}^{N} \left[-(x^{(i)})^T WW^T x^{(i)} + (x^{(i)})^T x^{(i)} \right]$$

$$= \frac{1}{N} \sum_{i=1}^{N} (x^{(i)})^T x^{(i)} - \frac{1}{N} \sum_{i=1}^{N} (x^{(i)})^T WW^T x^{(i)}$$

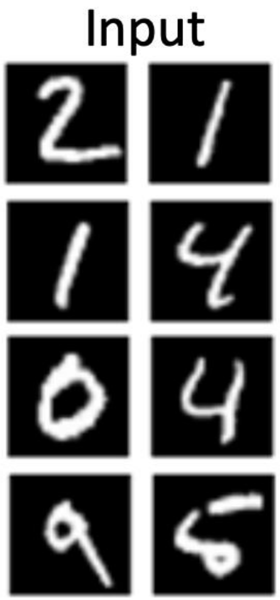
$$\max_W \frac{1}{N} \sum_{i=1}^{N} (x^{(i)})^T WW^T x^{(i)}$$

PCA to bridge GMM and High-Dimension Data

What we want: model any data distribution

How to transform any data distribution to low dimensional data?

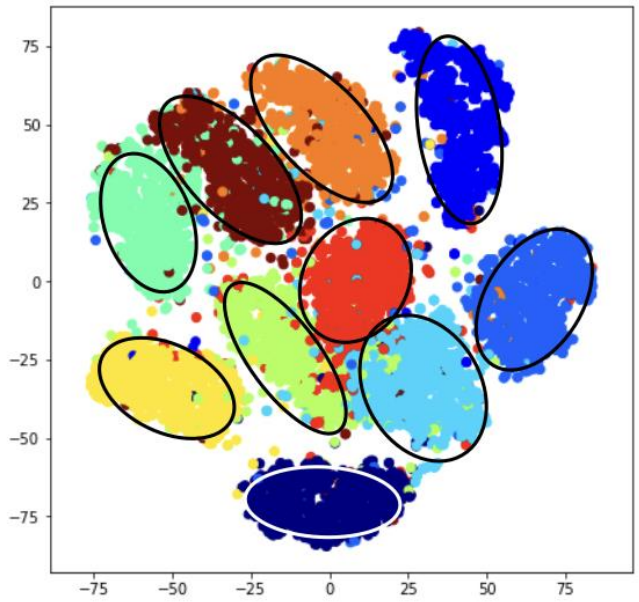
What we have: kernel density estimation to estimate low dimensional PDF



Someway to transform

Transform back

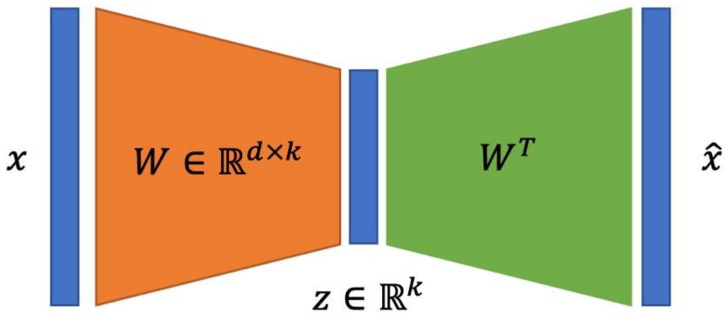
Kernel Density Estimation



PCA to bridge GMM and High-Dimension Data

[Code Demo]

From PCA to Auto-Encoder



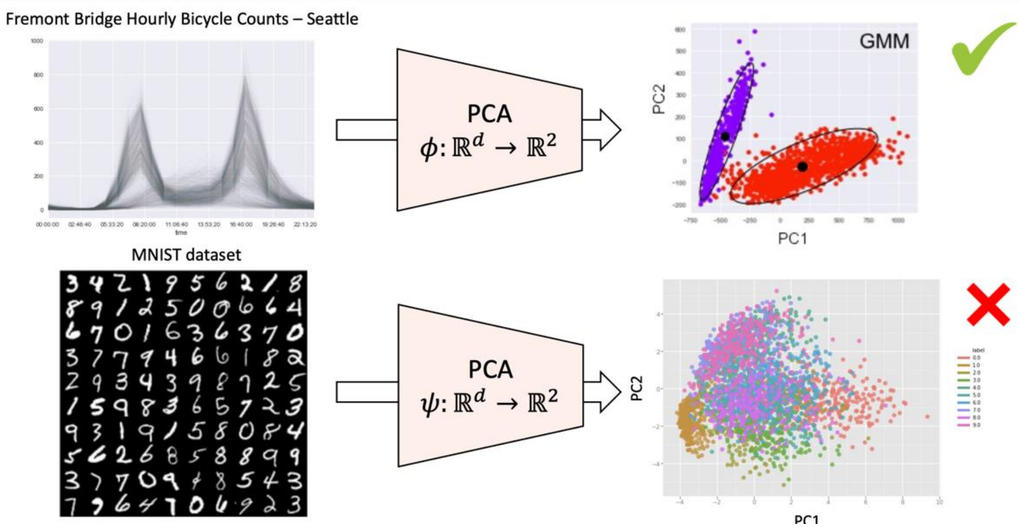
PCA:

- Forward transform: $z = W^T x$
- Inverse transform: $\hat{x} = Wz$

Linear dimensionality Reduction

$$\min_W \mathbb{E}_x[\|x - \hat{x}\|^2] = \mathbb{E}_x[\| x - WW^T x\|^2]$$
$$s.t.\quad W^T W = I_{k\times k}$$

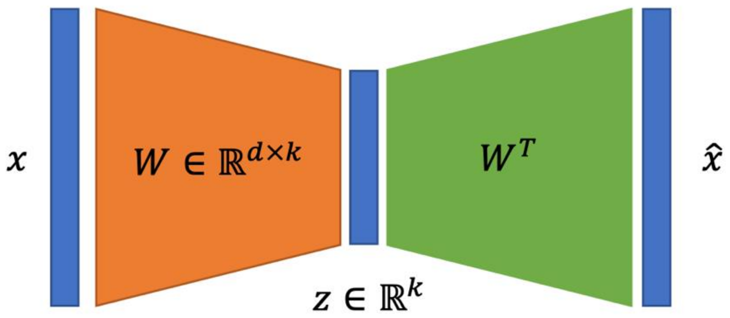
High-dimensional data often lives on non-linear manifolds that cannot be captured by linear models such as PCA



Can we add nonlinearity?

Yes, then it becomes neural network!

From PCA to Auto-Encoder

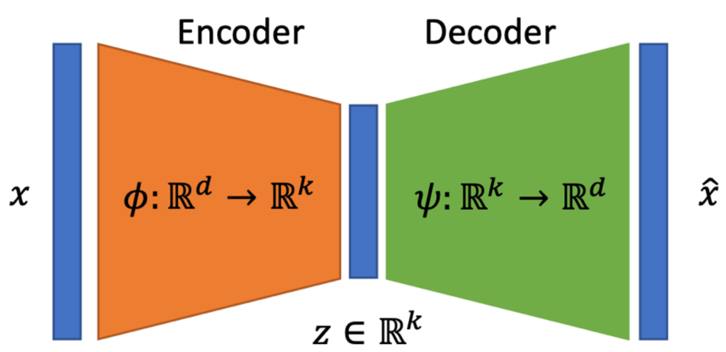


PCA:

- Forward transform: $z = W^T x$
- Inverse transform: $\hat{x} = Wz$

Linear dimensionality Reduction

$$\min_W \mathbb{E}_x[\|x - \hat{x}\|^2] = \mathbb{E}_x[\| x - WW^T x\|^2]$$
$$s.t. \quad W^T W = I_{k \times k}$$



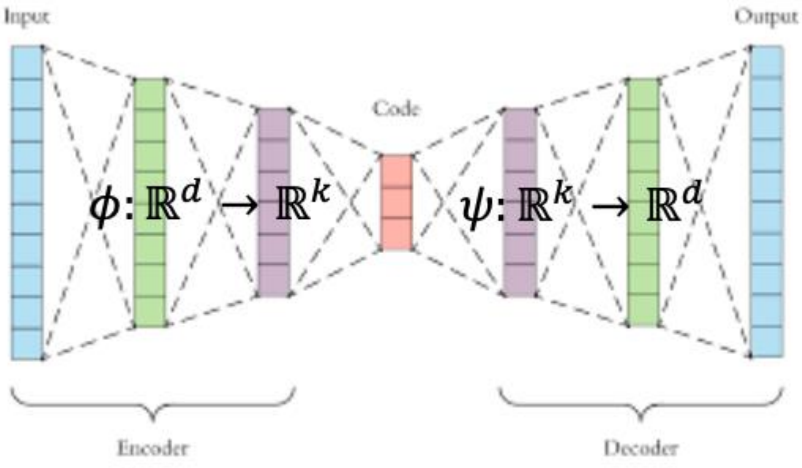
AE:

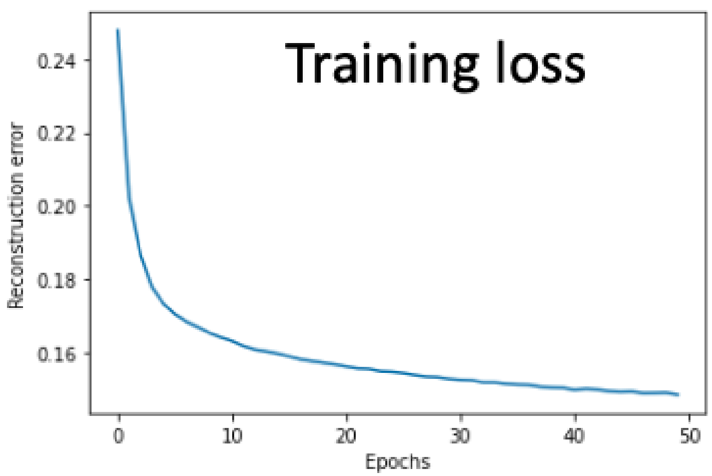
- Forward transform: $z = \phi(x)$
- Inverse transform: $\hat{x} = \psi(z)$

Nonlinear dimensionality Reduction

$$\min_{\phi,\psi} \mathbb{E}_x[\|x - \hat{x}\|^2] = \mathbb{E}_x\left[\left\| x - \psi(\phi(x))\right\|^2\right]$$

Auto-Encoder

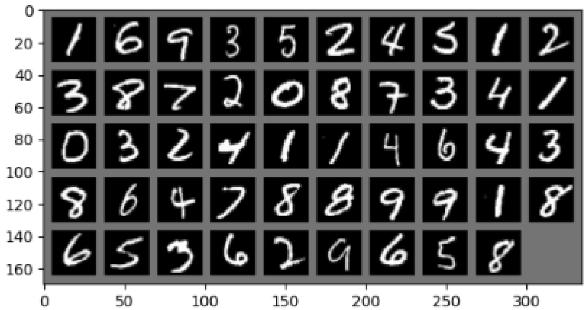




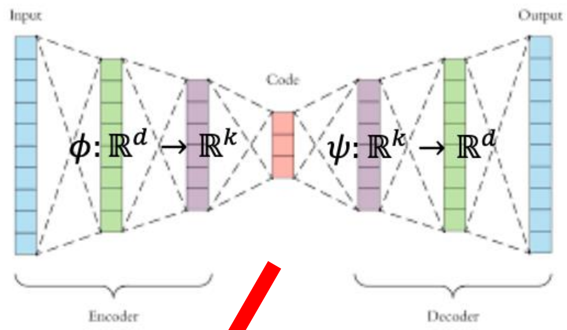

```python
class MLP_AE(nn.Module):
  def __init__(self,architecture=[784,128,64,2],activation='LeakyReLU'):
        super(MLP_AE, self).__init__()
        self.architecture=architecture
        if activation=='LeakyReLU':
            self.activation=nn.LeakyReLU()
        elif activation=='ReLU':
            self.activation=nn.ReLU()
        elif activation=='Sigmoid':
            self.activation=nn.Sigmoid()
        else:
            print('Activation not defined, reverting to default!')
            self.activation=nn.LeakyReLU()
        # Defining $\phi$
        arch=[]
        for i in range(1,len(architecture)):
            arch.append(nn.Linear(architecture[i-1],architecture[i]))
            if i!=len(architecture)-1:
                arch.append(self.activation)
        self.encoder=nn.Sequential(*arch)
        # Defining $\psi$
        arch=[]
        for i in range(len(architecture)-1,0,-1):
            arch.append(nn.Linear(architecture[i],architecture[i-1]))
            if i!=1:
                arch.append(self.activation)
        self.decoder=nn.Sequential(*arch)

  def encode(self,f):
        assert f.shape[1]==self.architecture[0]
        return self.encoder(f)

  def decode(self,fhat):
        assert fhat.shape[1]==self.architecture[-1]
        return self.decoder(fhat)

  def forward(self,x):
    return self.decode(self.encode(x))
```

Auto-Encoder

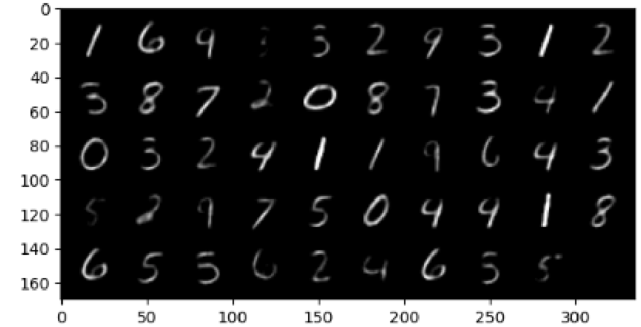
Input

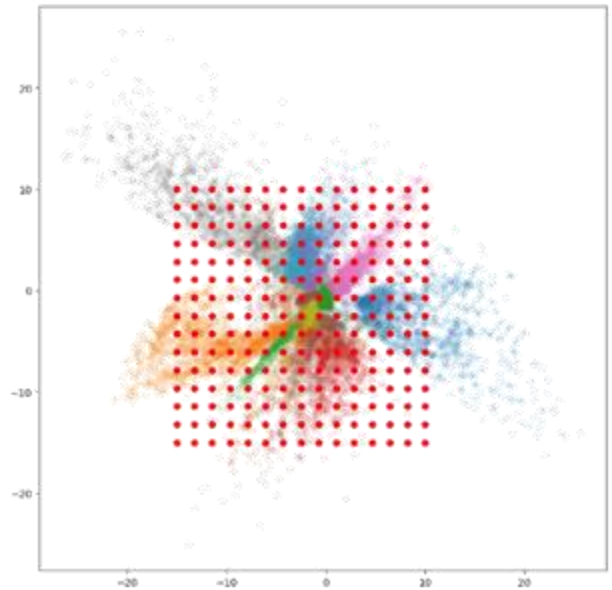


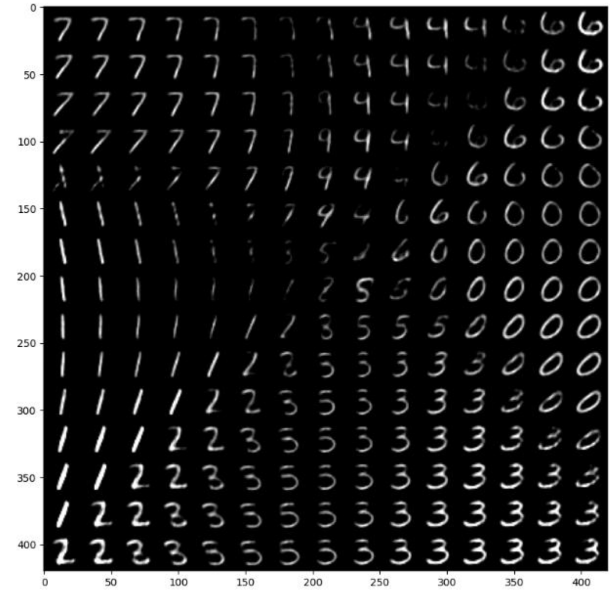
AE



Output

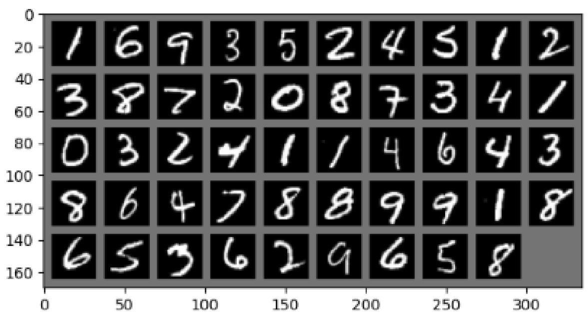




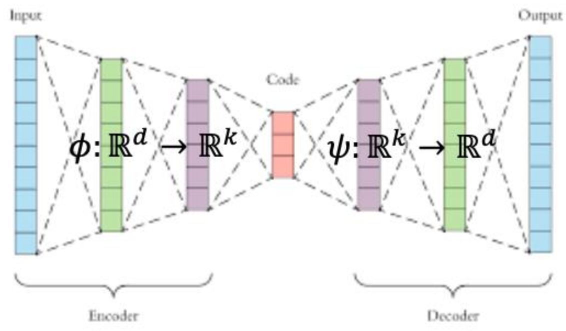


Auto-Encoder

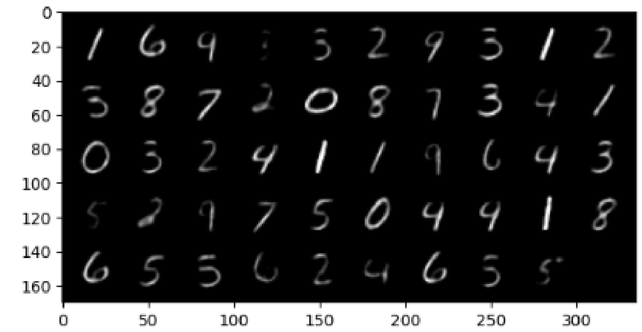
Input



AE

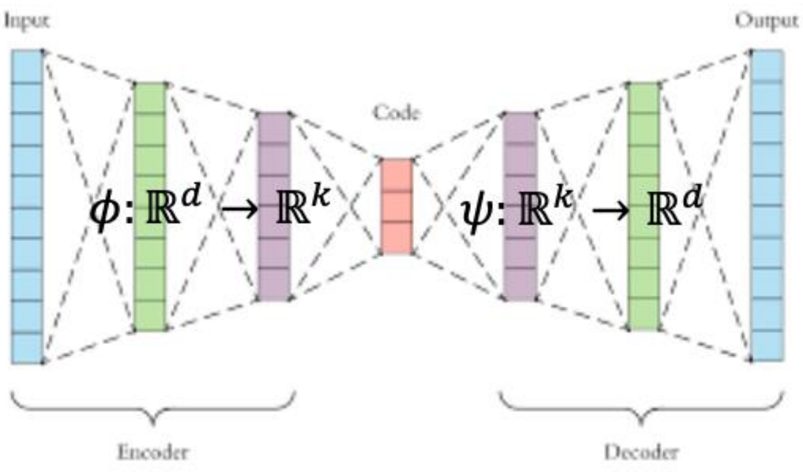


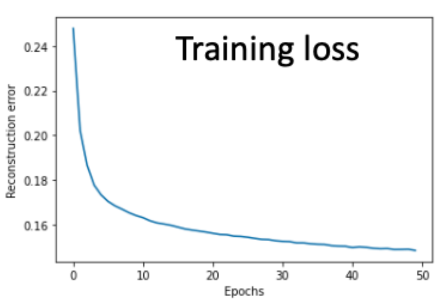
Output

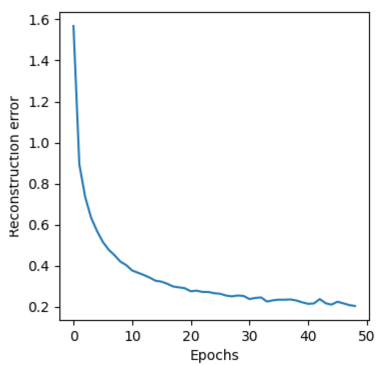


Any problem with this architecture?

Class-supervised Auto-Encoder



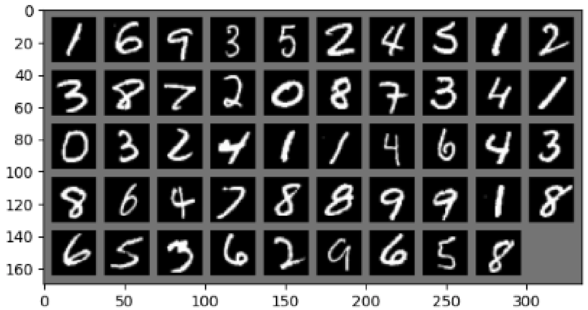




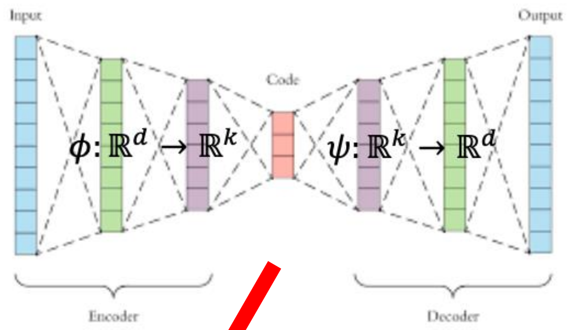

```python
class MLP_Discriminant_AE(nn.Module):
  def __init__(self,architecture=[784,128,64,2],nclasses=10,activation='LeakyReLU'):
        super(MLP_Discriminant_AE, self).__init__()
        self.architecture=architecture
        if activation=='LeakyReLU':
            self.activation=nn.LeakyReLU()
        elif activation=='ReLU':
            self.activation=nn.ReLU()
        elif activation=='Sigmoid':
            self.activation=nn.Sigmoid()
        else:
            print('Activation not defined, reverting to default!')
            self.activation=nn.LeakyReLU()
        # Defining $\phi$
        arch=[]
        for i in range(1,len(architecture)):
            arch.append(nn.Linear(architecture[i-1],architecture[i]))
            if i!=len(architecture)-1:
                arch.append(self.activation)
        self.encoder=nn.Sequential(*arch)
        # Defining $\psi$
        arch=[]
        for i in range(len(architecture)-1,0,-1):
            arch.append(nn.Linear(architecture[i],architecture[i-1]))
            if i!=1:
                arch.append(self.activation)
        self.decoder=nn.Sequential(*arch)
        self.classifier=nn.Linear(architecture[-1],nclasses)
        self.nclasses=nclasses

  def encode(self,f):
        assert f.shape[1]==self.architecture[0]
        return self.encoder(f)

  def decode(self,fhat):
        assert fhat.shape[1]==self.architecture[-1]
        return self.decoder(fhat)

  def forward(self,x):
    z=self.encode(x)
    return self.decode(z),self.classifier(z)
```

Class-supervised Auto-Encoder

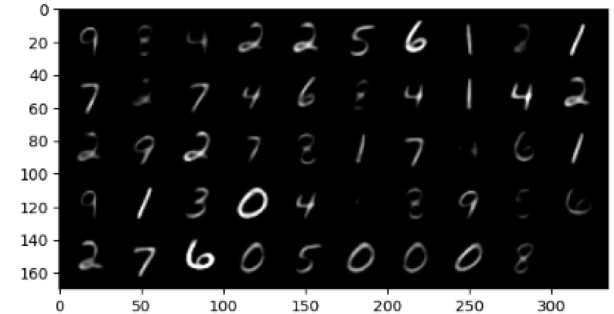
Input

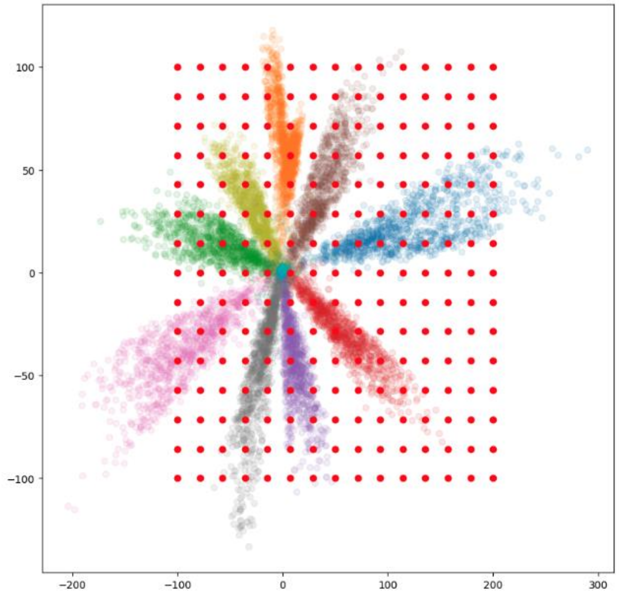


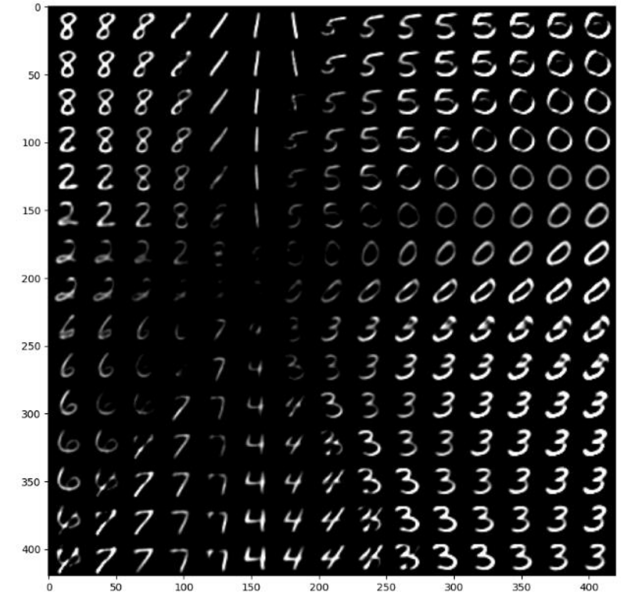
AE



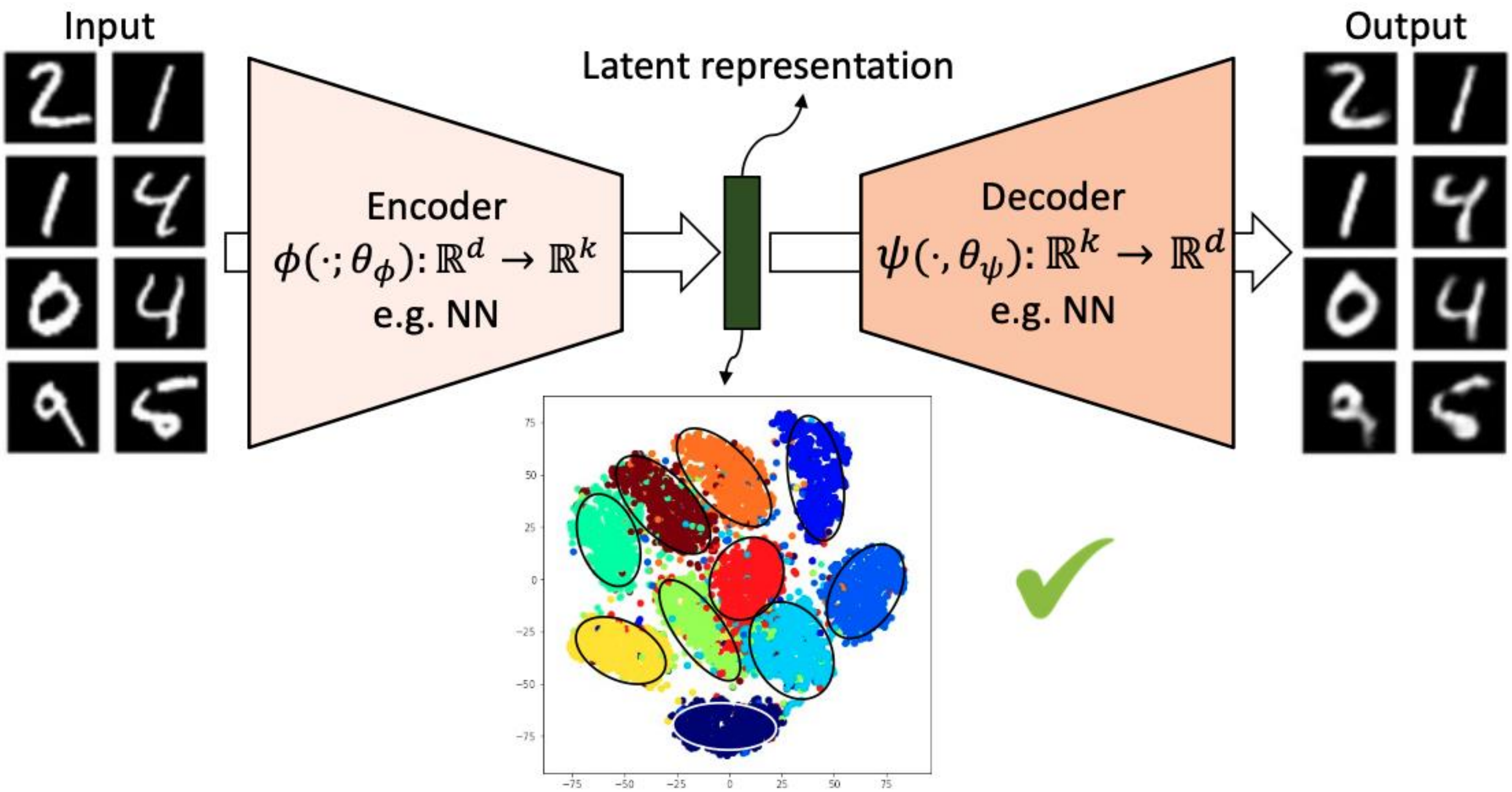
Output







Problem with Auto-Encoder

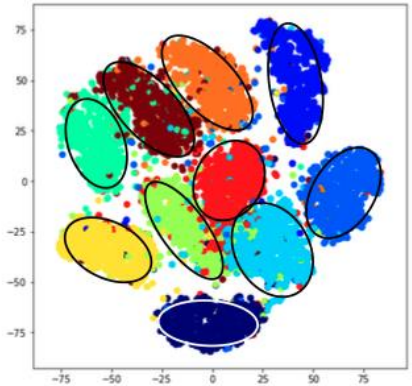


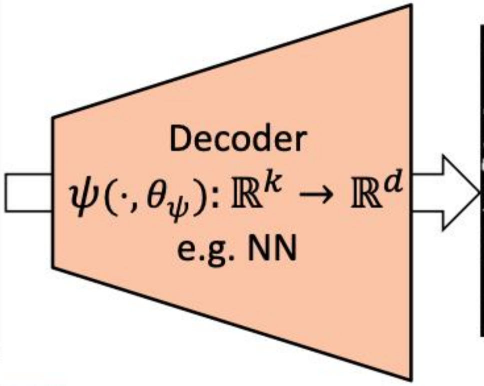
Need to estimate the latent distribution post-hoc!

Problem with Auto-Encoder

Deep Auto-Encoders (AE) could provide pseudo-invertible nonlinear dimensionality reduction

Sample GMM in the latent space





Synthesized images

SW-GMM

The encoder in the AE captures the nonlinear variations in the dataset enabling GMM modeling in the latent space while the decoder enables generative modeling.

Kolouri, S., Rohde, G. K., and Hoffmann, H., "Sliced Wasserstein Distances for Learning Gaussian Mixture Models", CVPR'18.