Autoencoder

Industrial AI Lab.

Prof. Seungchul Lee

Generative Models

- Many applications such as image synthesis, denoising, super-resolution, speech synthesis, compression, etc. require to go beyond classification and regression, and model explicitly a high dimension signal.

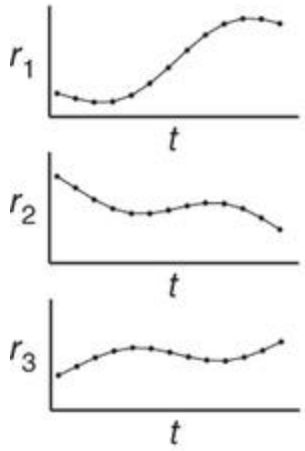
- This modeling consists of finding “meaningful degrees of freedom” that describe the signal, and are of lesser dimension.

Unsupervised Learning

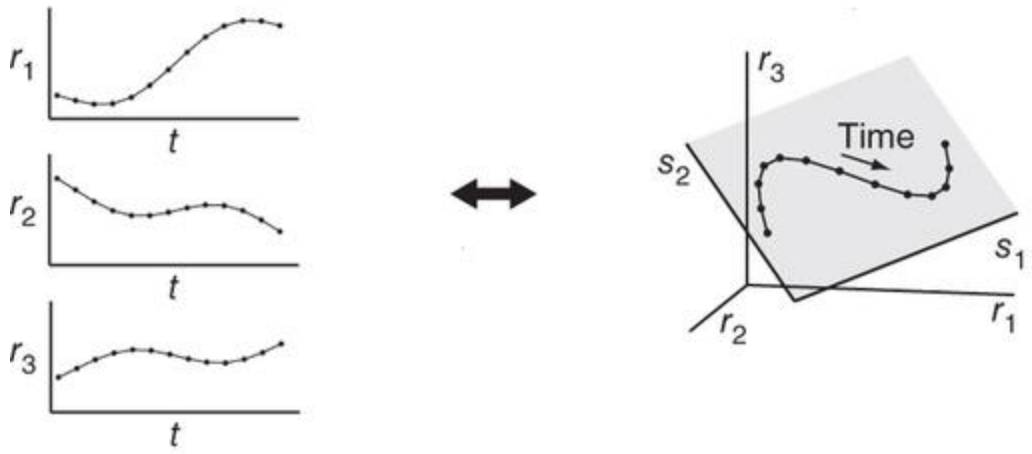
- Definition
  - Unsupervised learning refers to most attempts to extract information from a distribution that do not require human labor to annotate example
  - Main task is to find the ‘best’ representation of the data

- Dimension Reduction
  - Attempt to compress as much information as possible in a smaller representation
  - Preserve as much information as possible while obeying some constraint aimed at keeping the representation simpler

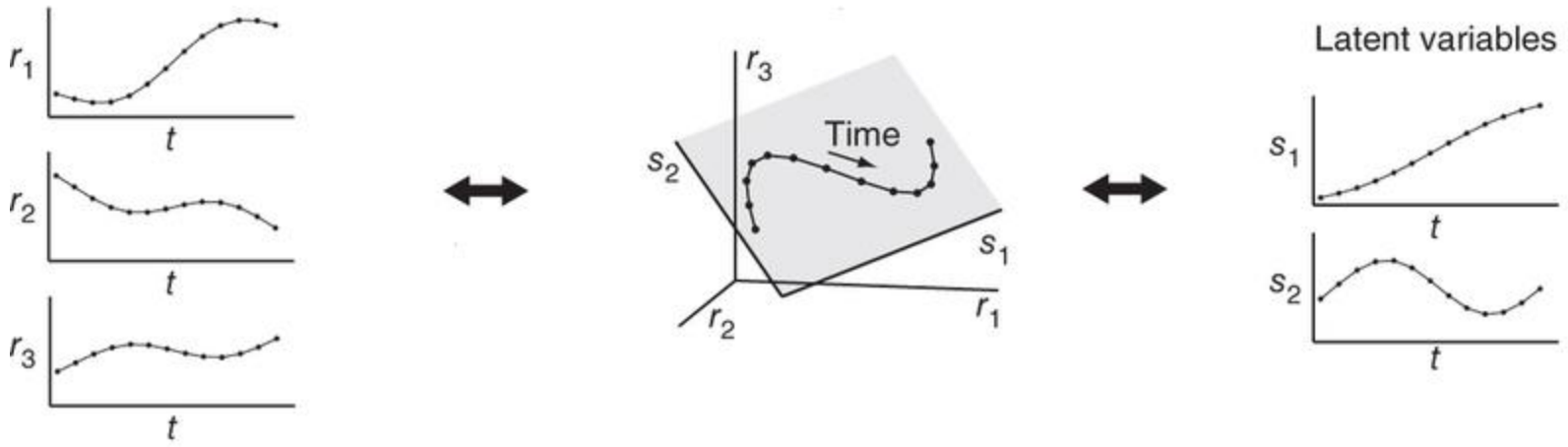
Unsupervised Learning



Unsupervised Learning

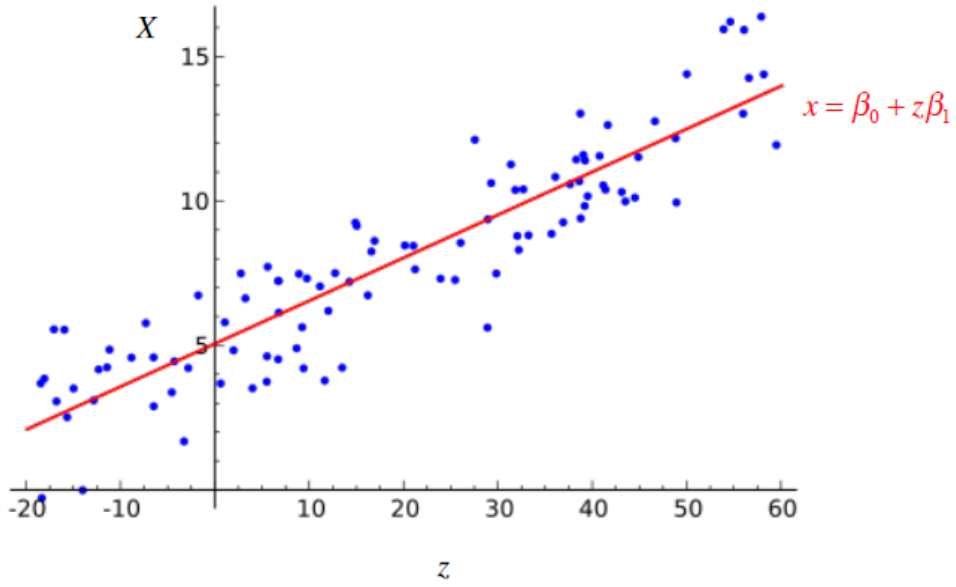


Unsupervised Learning



Recap: Linear Regression

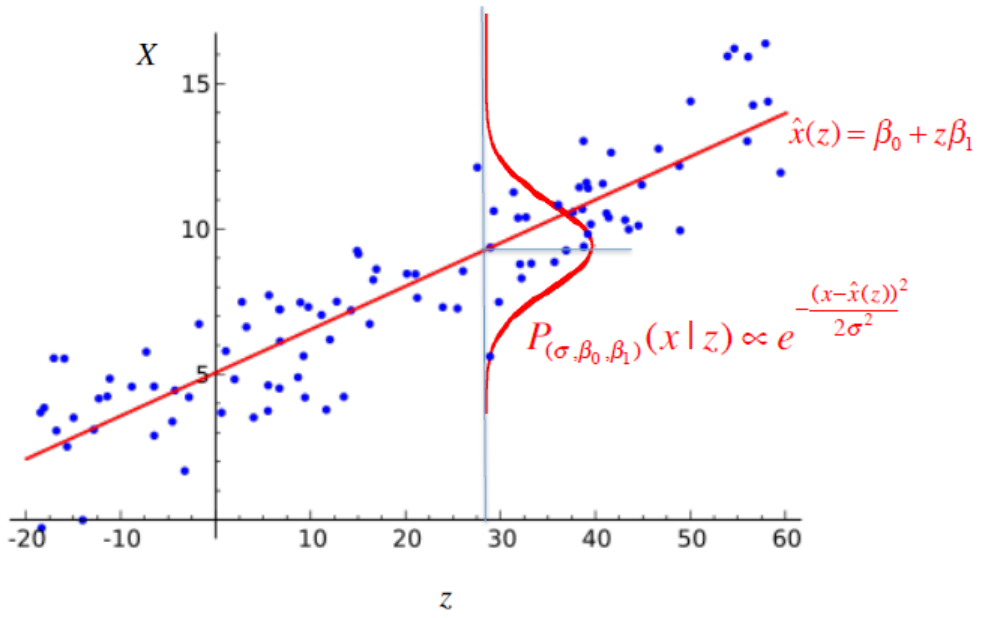
- Most people think of linear regression as points and a straight line:



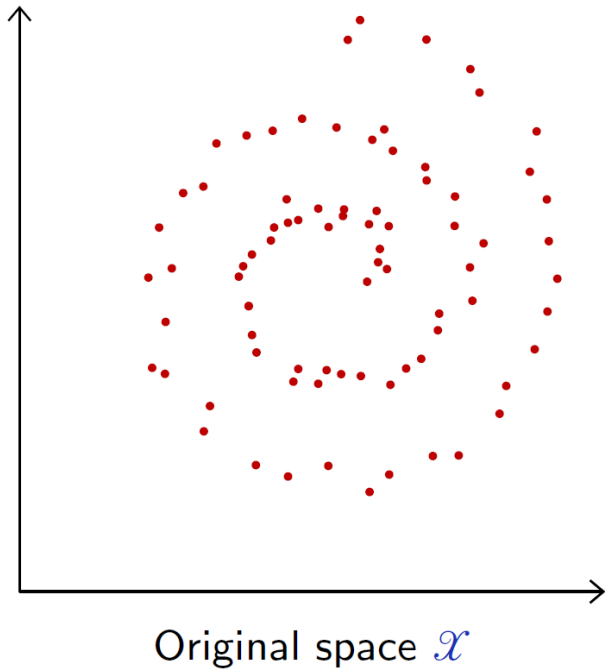
Recap: Linear Regression

- Statisticians additionally have $P_\theta(X|Z)$
- True model
  - May not be too complicated as opposed to original data
- Observed data = true model + error
- Benefits of having an error model:
  - How likely is a data point
  - Confidence bounds
  - Compare models

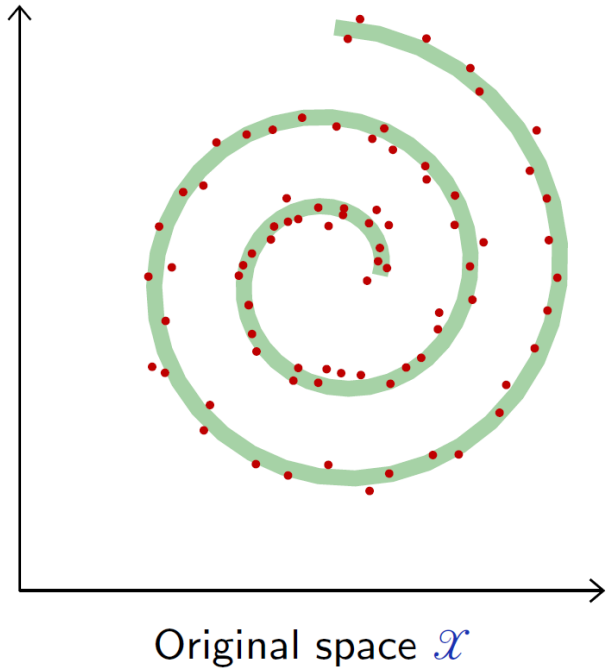
- Q: how to find an unseen true model (we never know the true model)



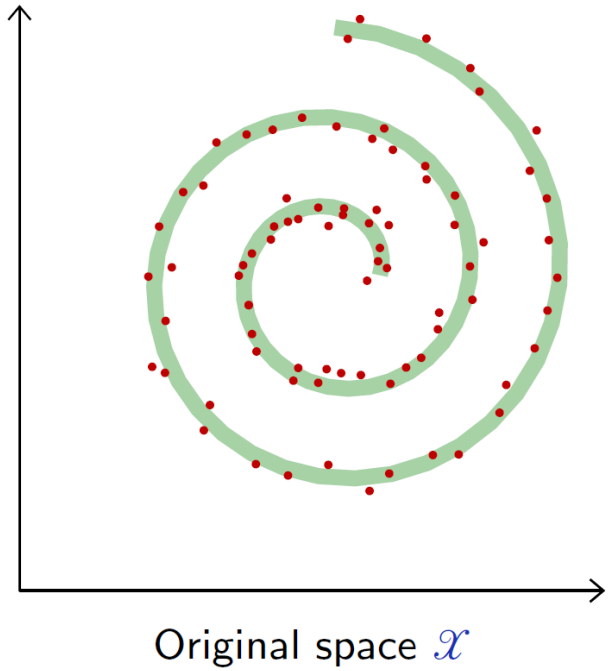
Original Space and Latent Space

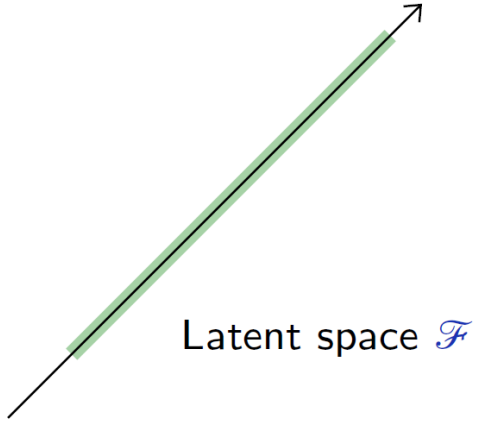


Original Space and Latent Space

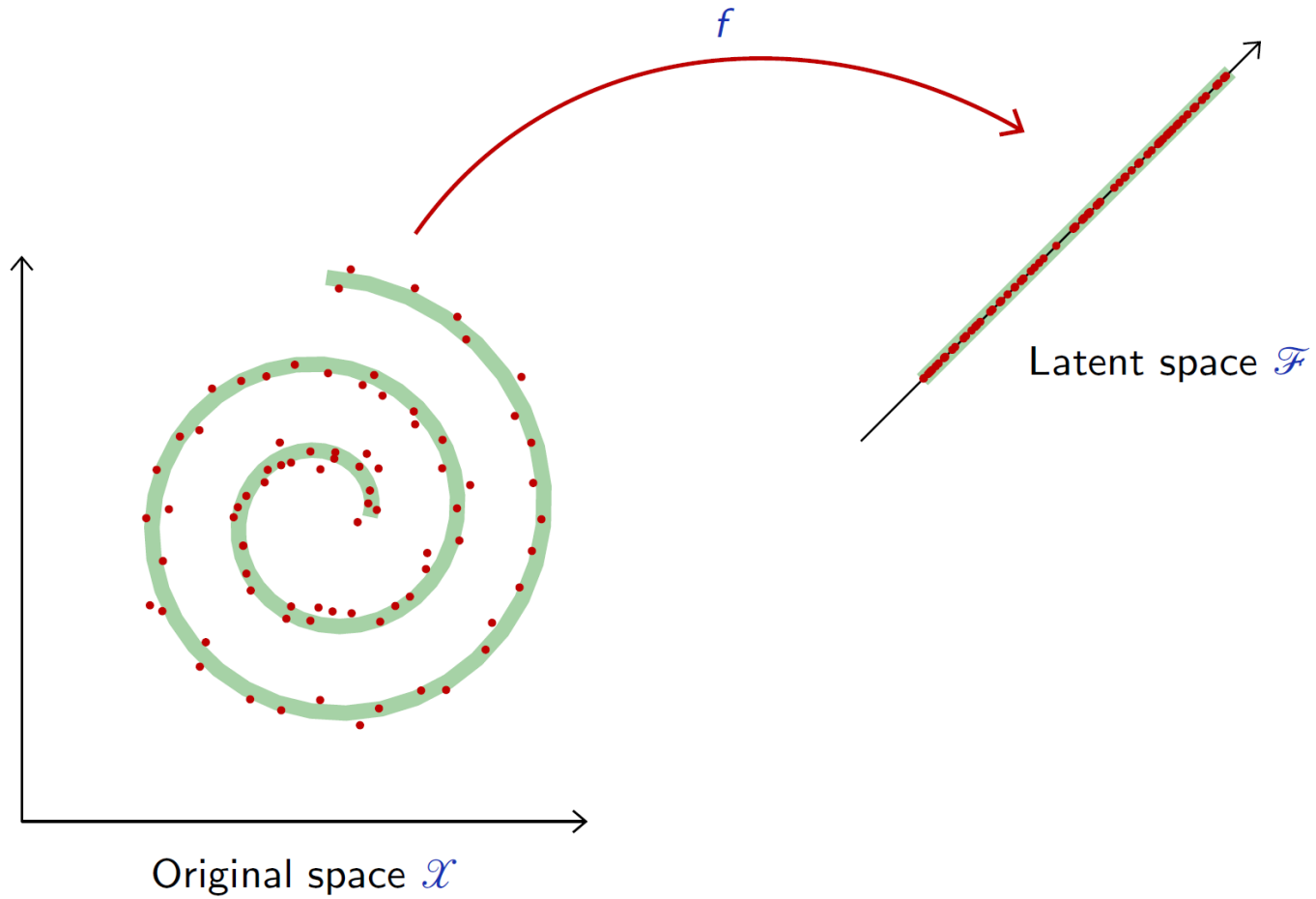


Original Space and Latent Space

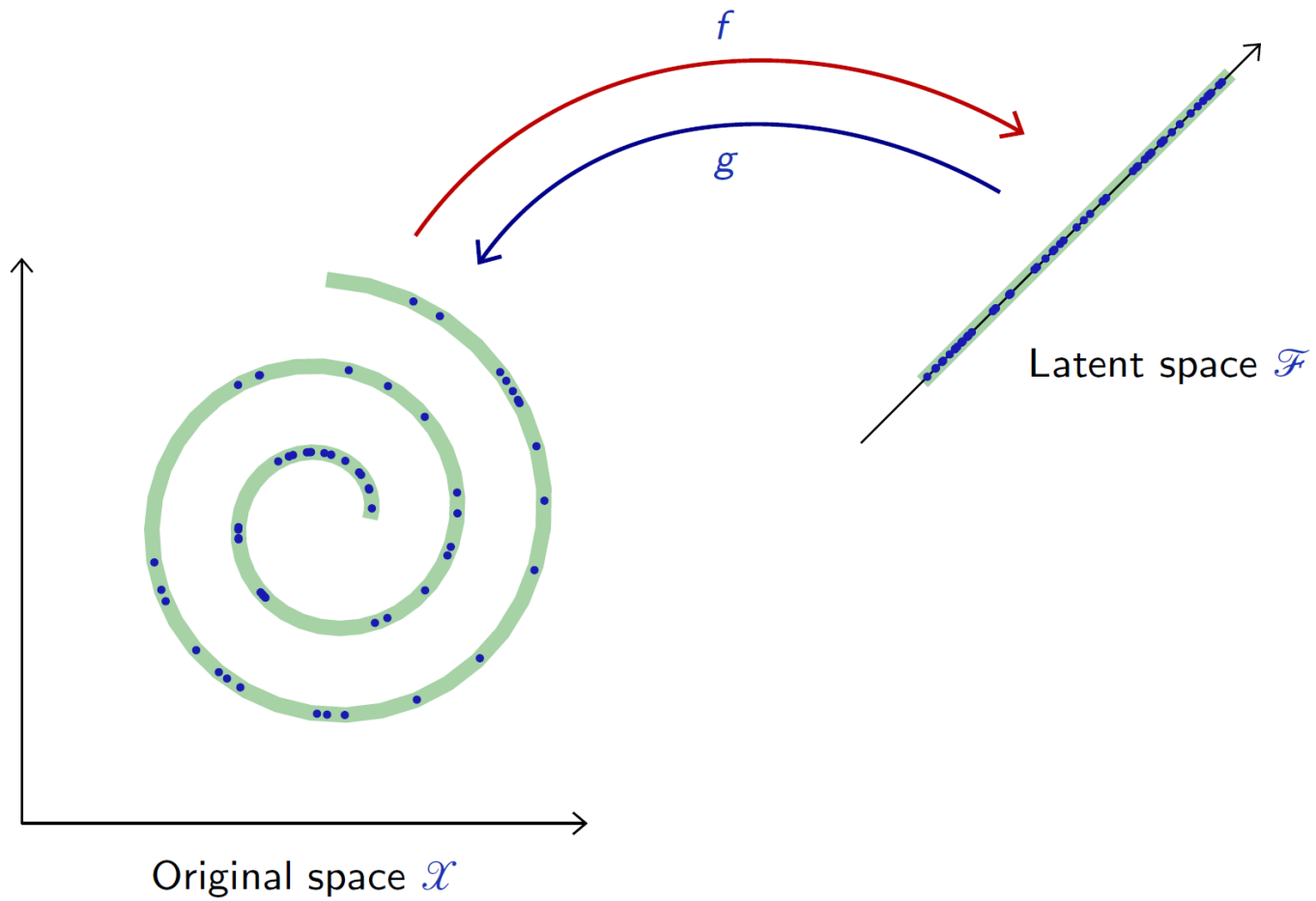




Original Space and Latent Space

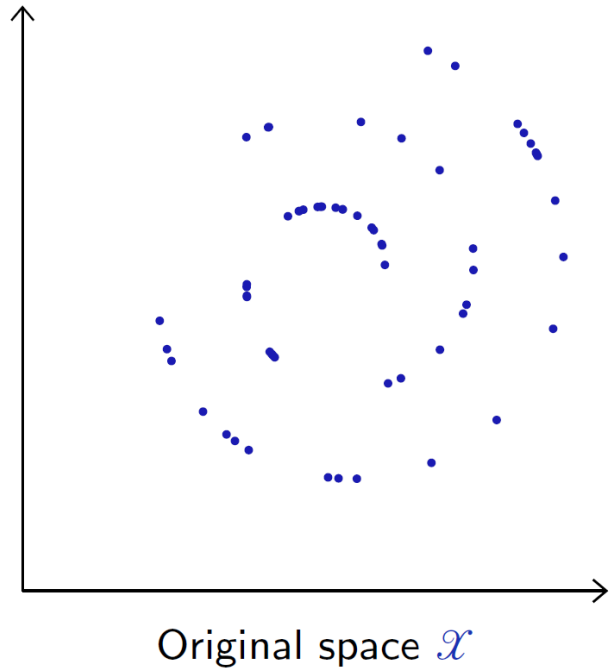


Original Space and Latent Space



Original Space and Latent Space

- We can generate data



Autoencoders

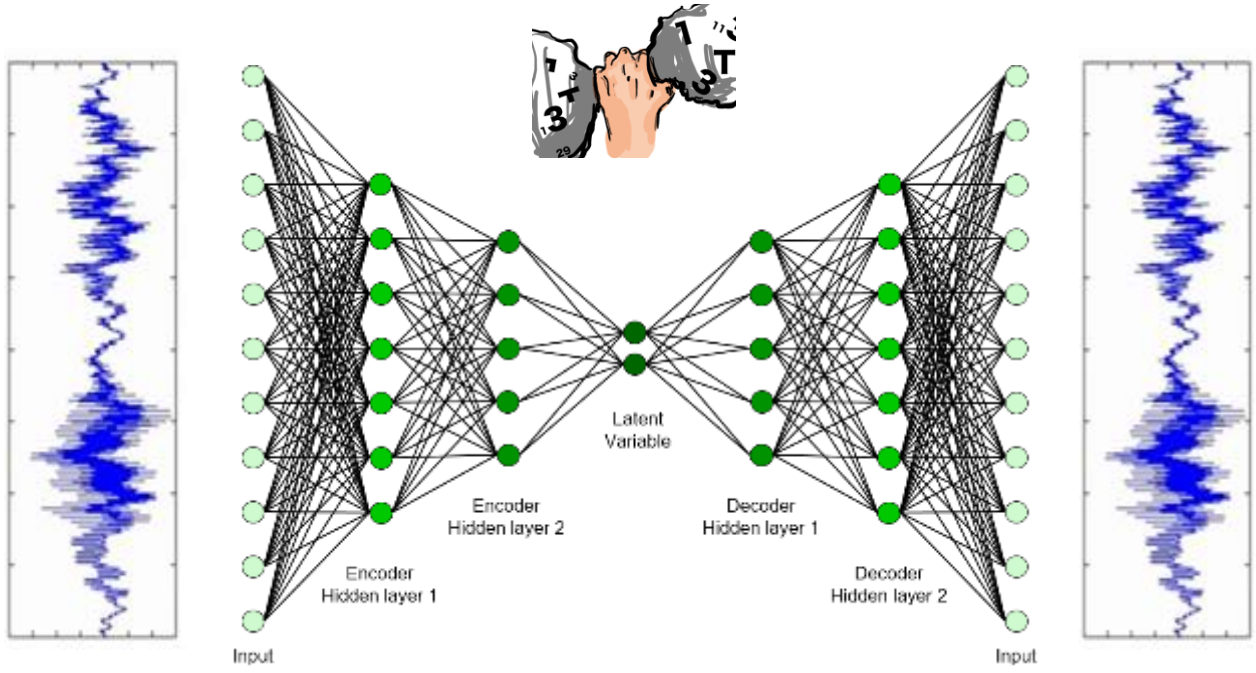
- It is like ‘deep learning version’ of unsupervised learning
- Definition
  - An autoencoder is a neural network that is trained to attempt to copy its input to its output
  - The network consists of two parts: an encoder and a decoder that produce a reconstruction
- Encoder and Decoder
  - Encoder function : $z = f(x)$
  - Decoder function : $x = g(z)$
  - We learn to set $g\big(f(x)\big) = x$

Autoencoder

- Dimension reduction
- Recover the input data

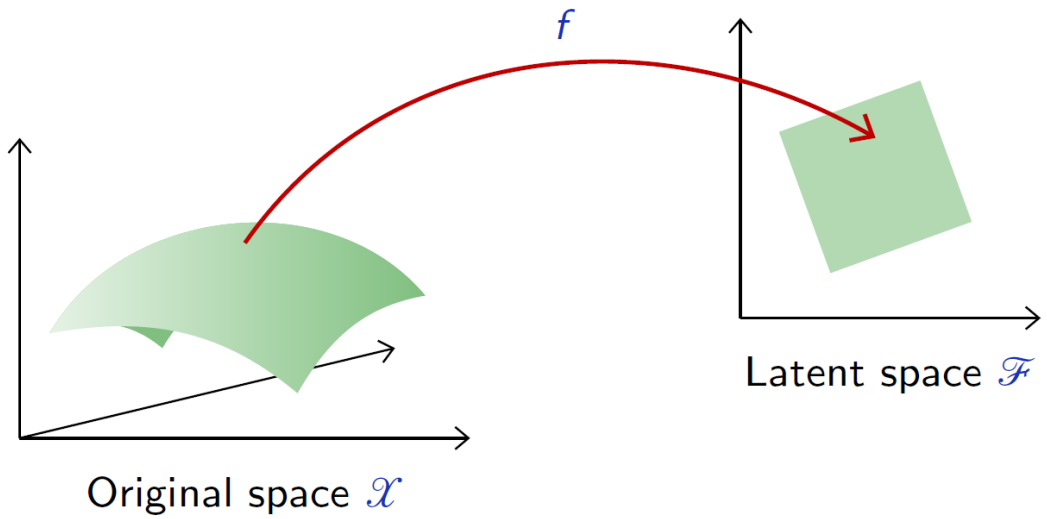
Autoencoder

- Dimension reduction
- Recover the input data
  - Learns an encoding of the inputs so as to recover the original input from the encodings as well as possible



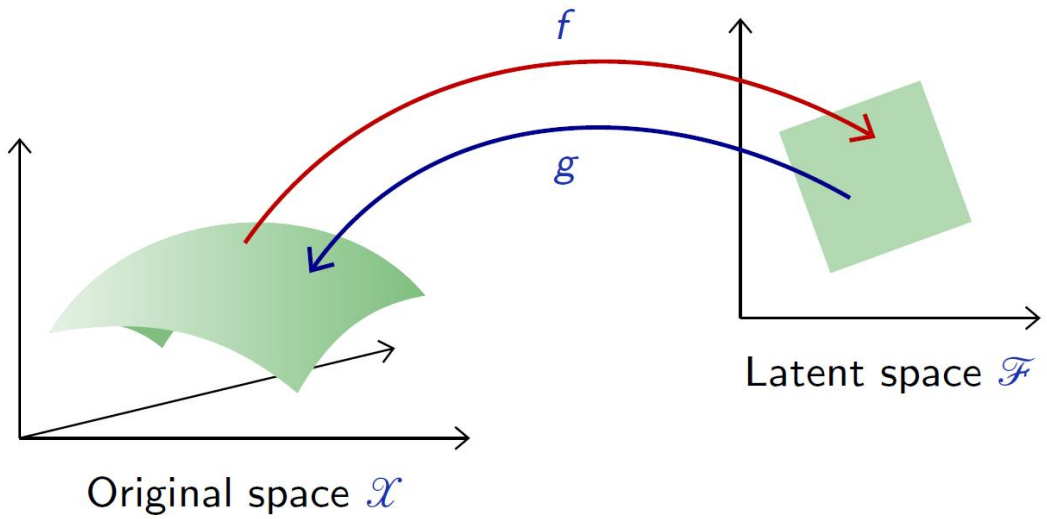
Autoencoder

- Autoencoder combines an encoder f from the original space $\mathcal{X}$ to a latent space $\mathcal{F}$, and a decoder g to map back to $\mathcal{X}$, such that $g \circ f$ is [close to] the identity on the data



Autoencoder

- Autoencoder combines an encoder f from the original space $\mathcal{X}$ to a latent space $\mathcal{F}$, and a decoder g to map back to $\mathcal{X}$, such that $g \circ f$ is [close to] the identity on the data



- A proper autoencoder has to capture a "good" parametrization of the signal, and in particular the statistical dependencies between the signal components.

Autoencoder

Let q be the data distribution over $\mathcal{X}$. A good autoencoder could be characterized with the quadratic loss

$$\mathbb{E}_{X \sim q}\left[\|X - g \circ f(X)\|^2\right] \simeq 0.$$

Given two parametrized mappings $f(\cdot\,; w)$ and $g(\cdot\,; w)$, training consists of minimizing an empirical estimate of that loss

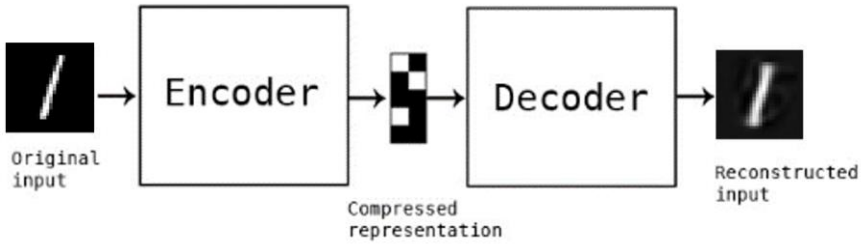
$$\hat{w}_f, \hat{w}_g = \underset{w_f, w_g}{\operatorname{argmin}} \frac{1}{N} \sum_{n=1}^{N} \|x_n - g(f(x_n; w_f); w_g)\|^2 .$$

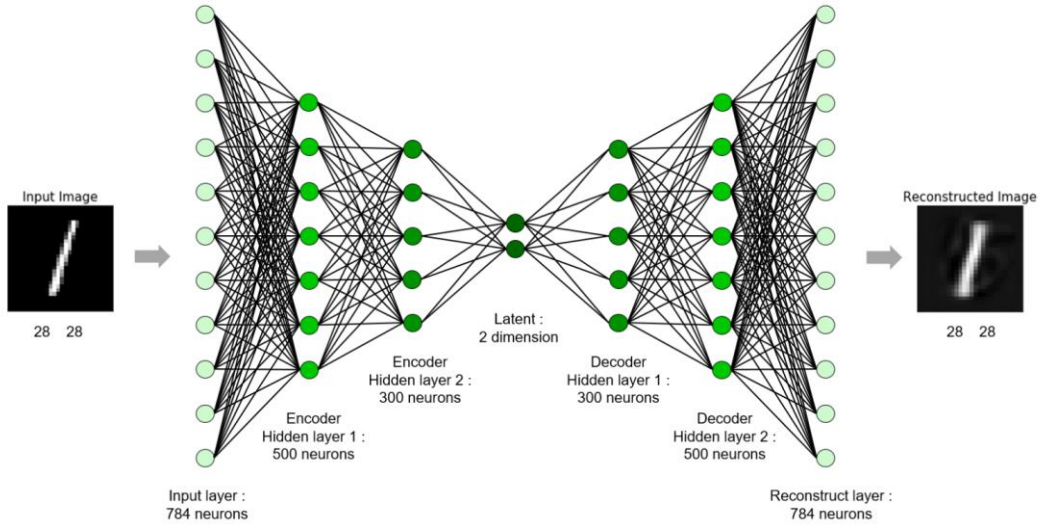
A simple example of such an autoencoder would be with both f and g linear, in which case the optimal solution is given by PCA. Better results can be achieved with more sophisticated classes of mappings, in particular deep architectures.

Autoencoder with MNIST

Autoencoder with TensorFlow

- MNIST example
- Use only (1, 5, 6) digits to visualize in 2-D





Import Libraries and Load MNIST Data

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
%matplotlib inline
```

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
```

- Only use (1, 5, 6) digits to visualize latent space in 2-D

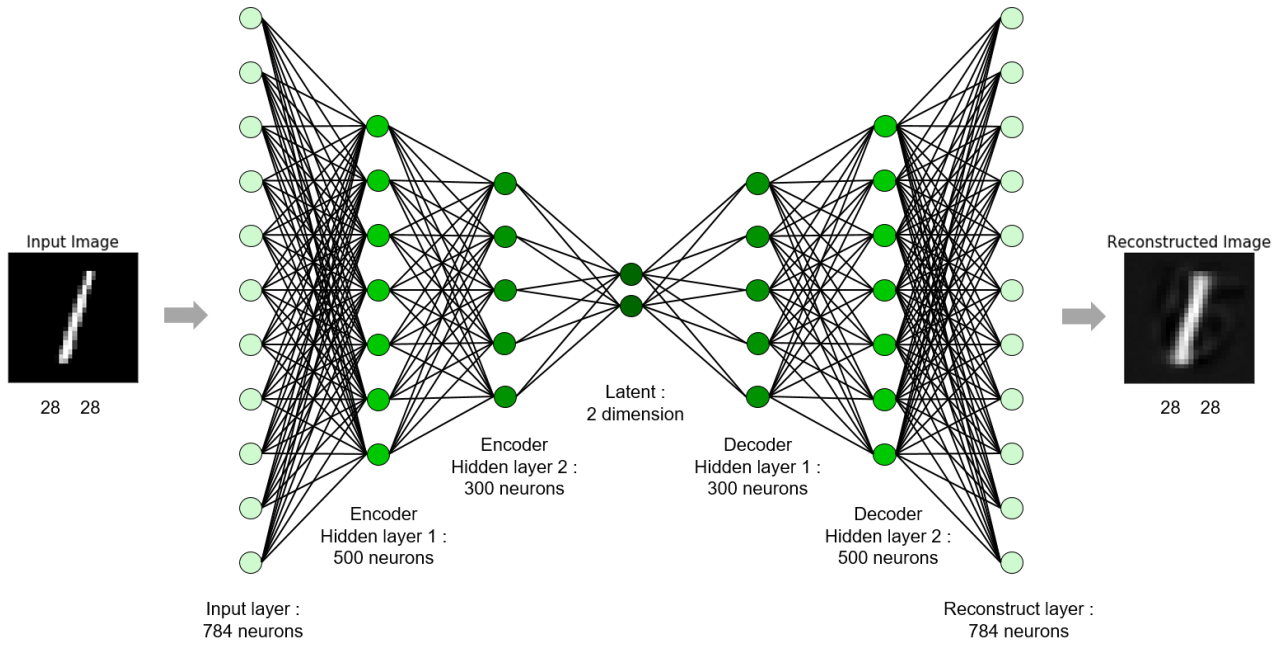
```
train_idx = ((np.argmax(mnist.train.labels, 1) == 1) | \
             (np.argmax(mnist.train.labels, 1) == 5) | \
             (np.argmax(mnist.train.labels, 1) == 6))
test_idx = ((np.argmax(mnist.test.labels, 1) == 1) | \
            (np.argmax(mnist.test.labels, 1) == 5) | \
            (np.argmax(mnist.test.labels, 1) == 6))

train_imgs   = mnist.train.images[train_idx]
train_labels = mnist.train.labels[train_idx]
test_imgs    = mnist.test.images[test_idx]
test_labels  = mnist.test.labels[test_idx]
n_train      = train_imgs.shape[0]
n_test       = test_imgs.shape[0]

print ("The number of training images : {}, shape : {}".format(n_train, train_imgs.shape))
print ("The number of testing images : {}, shape : {}".format(n_test, test_imgs.shape))
```

```
The number of training images : 16583, shape : (16583, 784)
The number of testing images : 2985, shape : (2985, 784)
```

Structure of Autoencoder

- Input shape and latent variable shape
- Encoder shape
- Decoder shape

```
# Shape of input and latent variable

n_input = 28*28

# Encoder structure
n_encoder1 = 500
n_encoder2 = 300

n_latent = 2

# Decoder structure
n_decoder1 = 300
n_decoder2 = 500
```



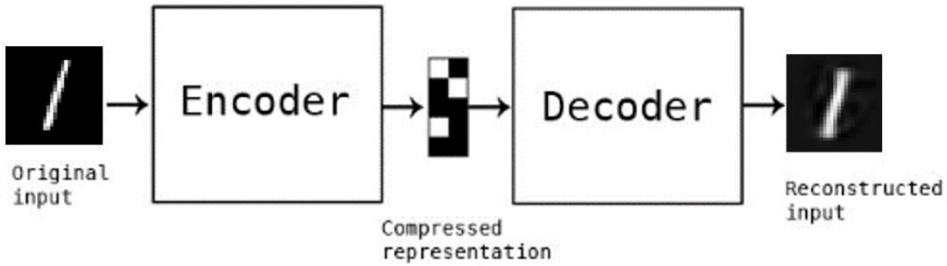
Weights & Biases, and Placeholder

- Define weights and biases for encoder and decoder, separately
- Based on the pre-defined layer size
- Initialize with normal distribution of $\mu = 0$ and $\sigma = 0.1$

```
weights = {
    'encoder1' : tf.Variable(tf.random_normal([n_input, n_encoder1], stddev=0.1)),
    'encoder2' : tf.Variable(tf.random_normal([n_encoder1, n_encoder2], stddev=0.1)),
    'latent' : tf.Variable(tf.random_normal([n_encoder2, n_latent], stddev=0.1)),
    'decoder1' : tf.Variable(tf.random_normal([n_latent, n_decoder1], stddev=0.1)),
    'decoder2' : tf.Variable(tf.random_normal([n_decoder1, n_decoder2], stddev=0.1)),
    'reconst' : tf.Variable(tf.random_normal([n_decoder2, n_input], stddev=0.1))
}

biases = {
    'encoder1' : tf.Variable(tf.random_normal([n_encoder1], stddev=0.1)),
    'encoder2' : tf.Variable(tf.random_normal([n_encoder2], stddev=0.1)),
    'latent' : tf.Variable(tf.random_normal([n_latent], stddev=0.1)),
    'decoder1' : tf.Variable(tf.random_normal([n_decoder1], stddev=0.1)),
    'decoder2' : tf.Variable(tf.random_normal([n_decoder2], stddev=0.1)),
    'reconst' : tf.Variable(tf.random_normal([n_input], stddev=0.1))
}
```

```
x = tf.placeholder(tf.float32, [None, n_input])
```

Build a Model




```python
def encoder(x, weights, biases):
    encoder1 = tf.add(tf.matmul(x, weights['encoder1']), biases['encoder1'])
    encoder1 = tf.nn.tanh(encoder1)

    encoder2 = tf.add(tf.matmul(encoder1, weights['encoder2']), biases['encoder2'])
    encoder2 = tf.nn.tanh(encoder2)

    latent = tf.add(tf.matmul(encoder2, weights['latent']), biases['latent'])

    return latent
```

```python
def decoder(latent, weights, biases):
    decoder1 = tf.add(tf.matmul(latent, weights['decoder1']), biases['decoder1'])
    decoder1 = tf.nn.tanh(decoder1)

    decoder2 = tf.add(tf.matmul(decoder1, weights['decoder2']), biases['decoder2'])
    decoder2 = tf.nn.tanh(decoder2)

    reconst = tf.add(tf.matmul(decoder2, weights['reconst']), biases['reconst'])

    return reconst
```

Loss and Optimizer

- Loss
  - Squared loss

$$\frac{1}{m}\sum_{i=1}^{m}(t_i - y_i)^2$$

- Optimizer
  - AdamOptimizer: the most popular optimizer

```
LR = 0.0001

latent = encoder(x, weights, biases)
reconst = decoder(latent, weights, biases)
loss = tf.square(tf.subtract(x, reconst))
loss = tf.reduce_mean(loss)

optm = tf.train.AdamOptimizer(LR).minimize(loss)
```

Define Optimization Configuration and Batch Maker

- Define parameters for training autoencoder
  - n_batch: batch size for mini-batch gradient descent
  - n_iter: the number of iterations steps
  - n_prt: check loss for every n_prt iteration

```
n_batch = 50
n_iter = 2500
n_prt = 250
```

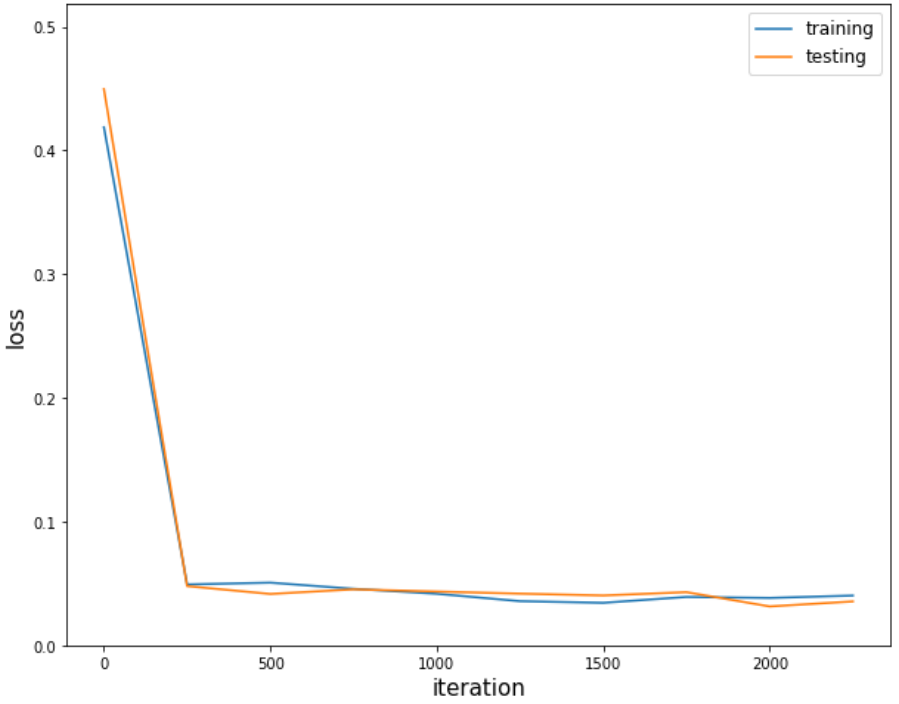
- Batch maker

```
def train_batch_maker(batch_size):
    random_idx = np.random.randint(n_train, size = batch_size)
    return train_imgs[random_idx], train_labels[random_idx]
```

```
def test_batch_maker(batch_size):
    random_idx = np.random.randint(n_test, size = batch_size)
    return test_imgs[random_idx], test_labels[random_idx]
```

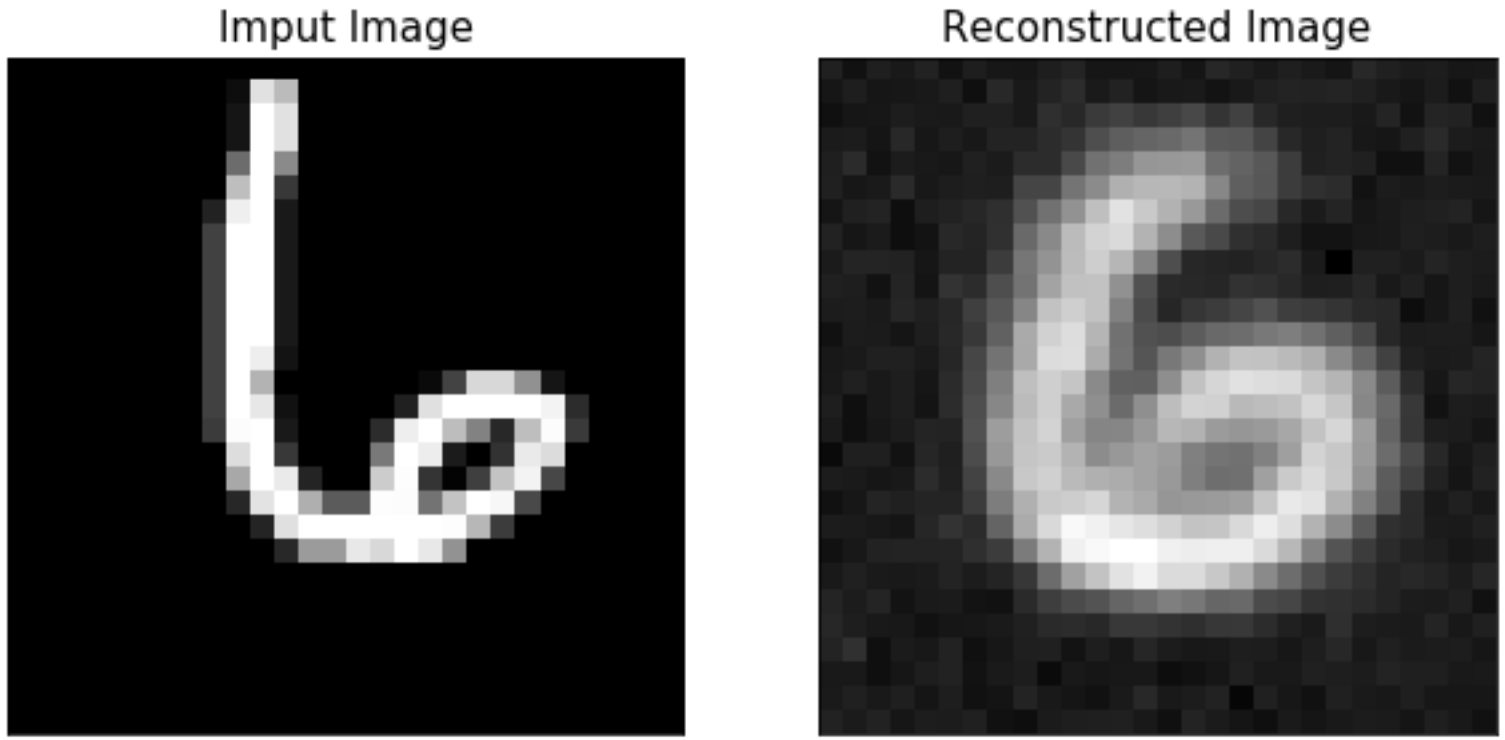
Optimization

```
sess = tf.Session()
init = tf.global_variables_initializer()
sess.run(init)

loss_record_train = []
loss_record_test = []
for epoch in range(n_iter):
    train_x, _ = train_batch_maker(n_batch)
    sess.run(optm, feed_dict = {x : train_x})

    if epoch % n_prt == 0:
        test_x, _ = test_batch_maker(n_batch)
        c1 = sess.run(loss, feed_dict = {x: train_x})
        c2 = sess.run(loss, feed_dict = {x: test_x})
        loss_record_train.append(c1)
        loss_record_test.append(c2)
        print ("Iter : {}".format(epoch))
        print ("Cost : {}".format(c1))
```



Test or Evaluation

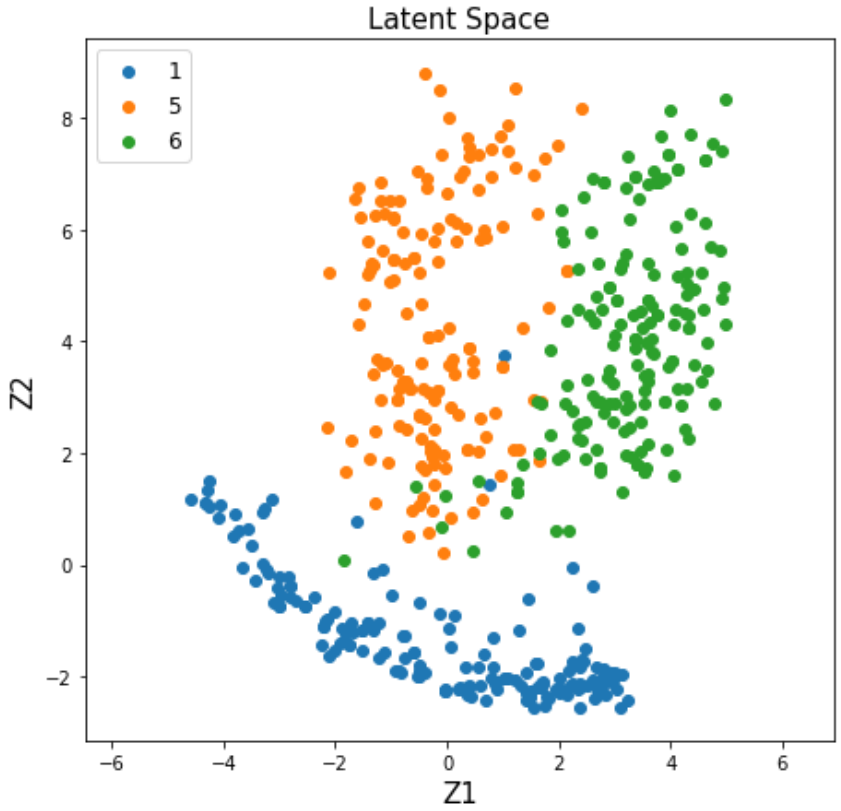
```
test_x, _ = test_batch_maker(1)
x_reconst = sess.run(reconst, feed_dict = {x: test_x})
```



Distribution in Latent Space

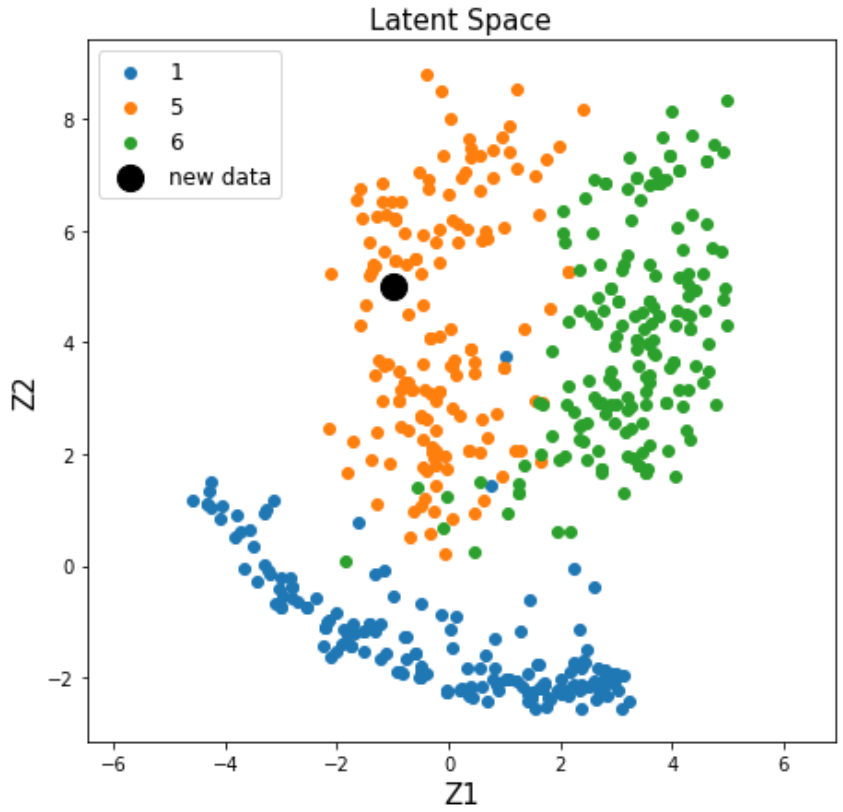
- Make a projection of 784-dim image onto 2-dim latent space

```
test_x, test_y = test_batch_maker(500)
test_y = np.argmax(test_y, axis = 1)
test_latent = sess.run(latent, feed_dict = {x: test_x})
```

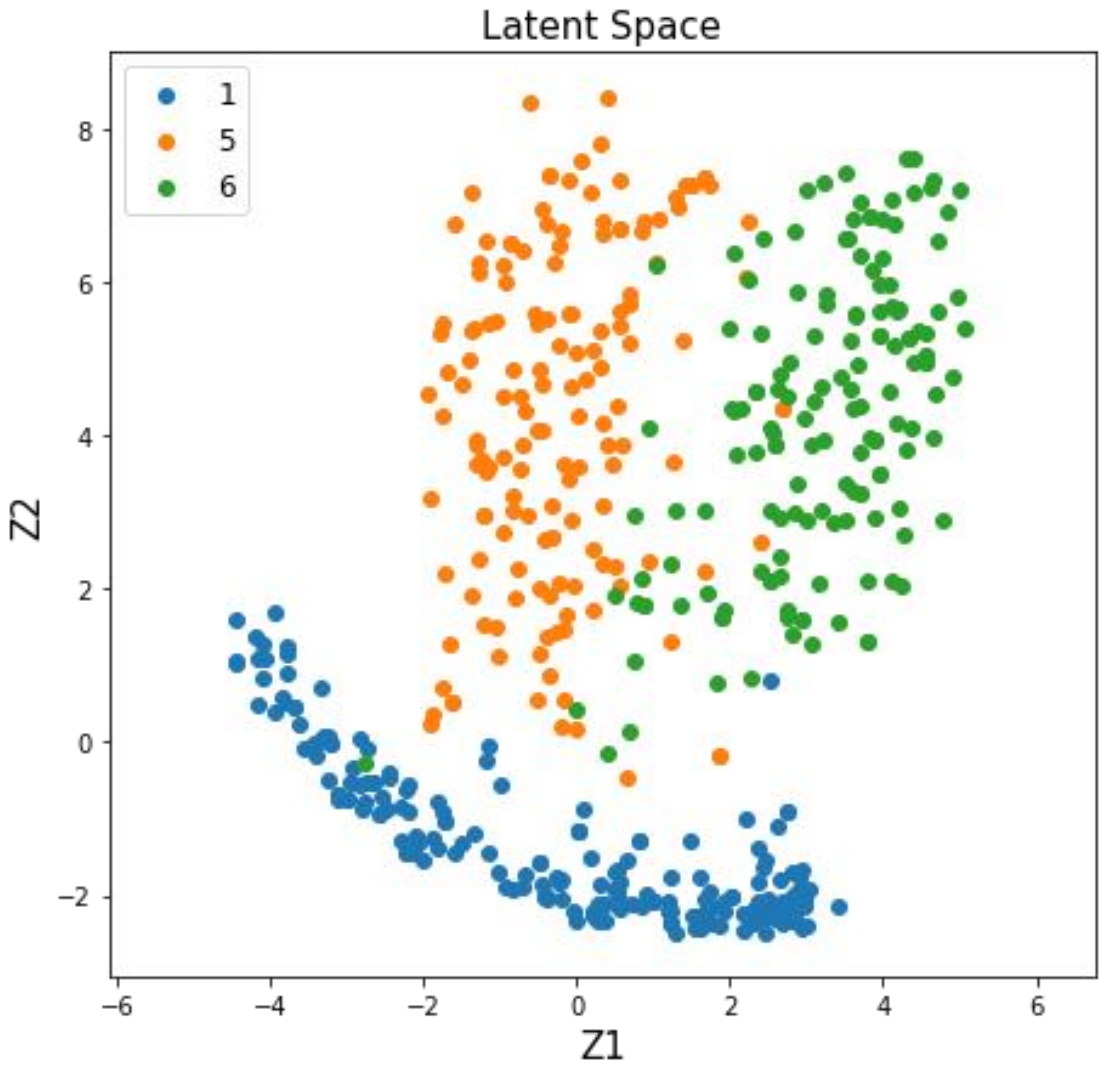


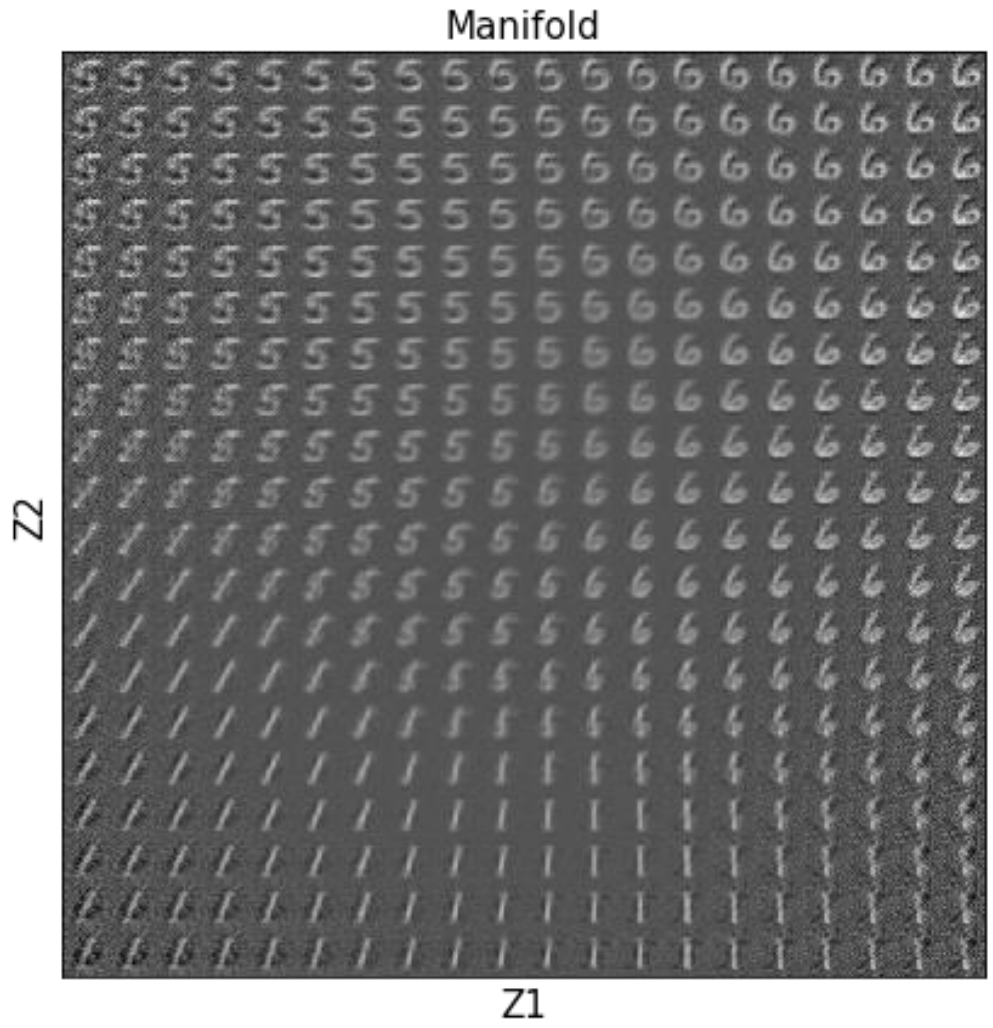
Data Generation

```
new_data = np.array([[-1, 5]])

latent_input = tf.placeholder(tf.float32, [None, n_latent])
reconst = decoder(latent_input, weights, biases)
fake_image = sess.run(reconst, feed_dict = {latent_input: new_data})
```



Visualization

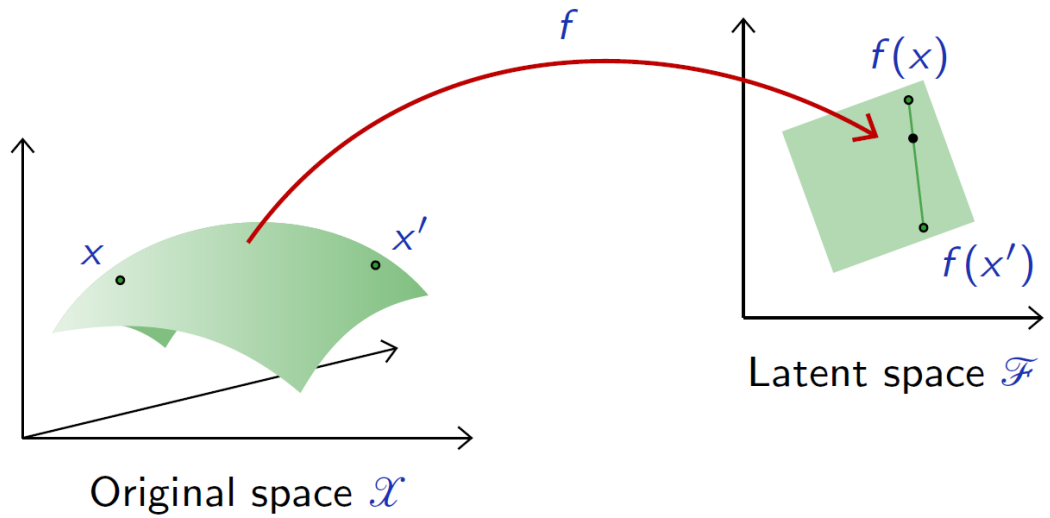




Autoencoder as Generative Model

Latent Representation

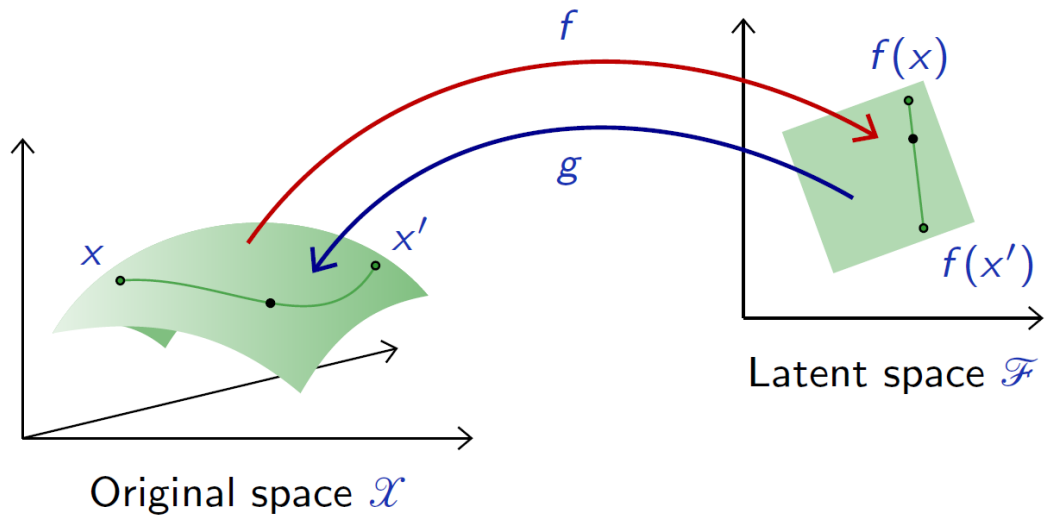
- To get an intuition of the latent representation, we can pick two samples x and x' at random and interpolate samples along the line in the latent space



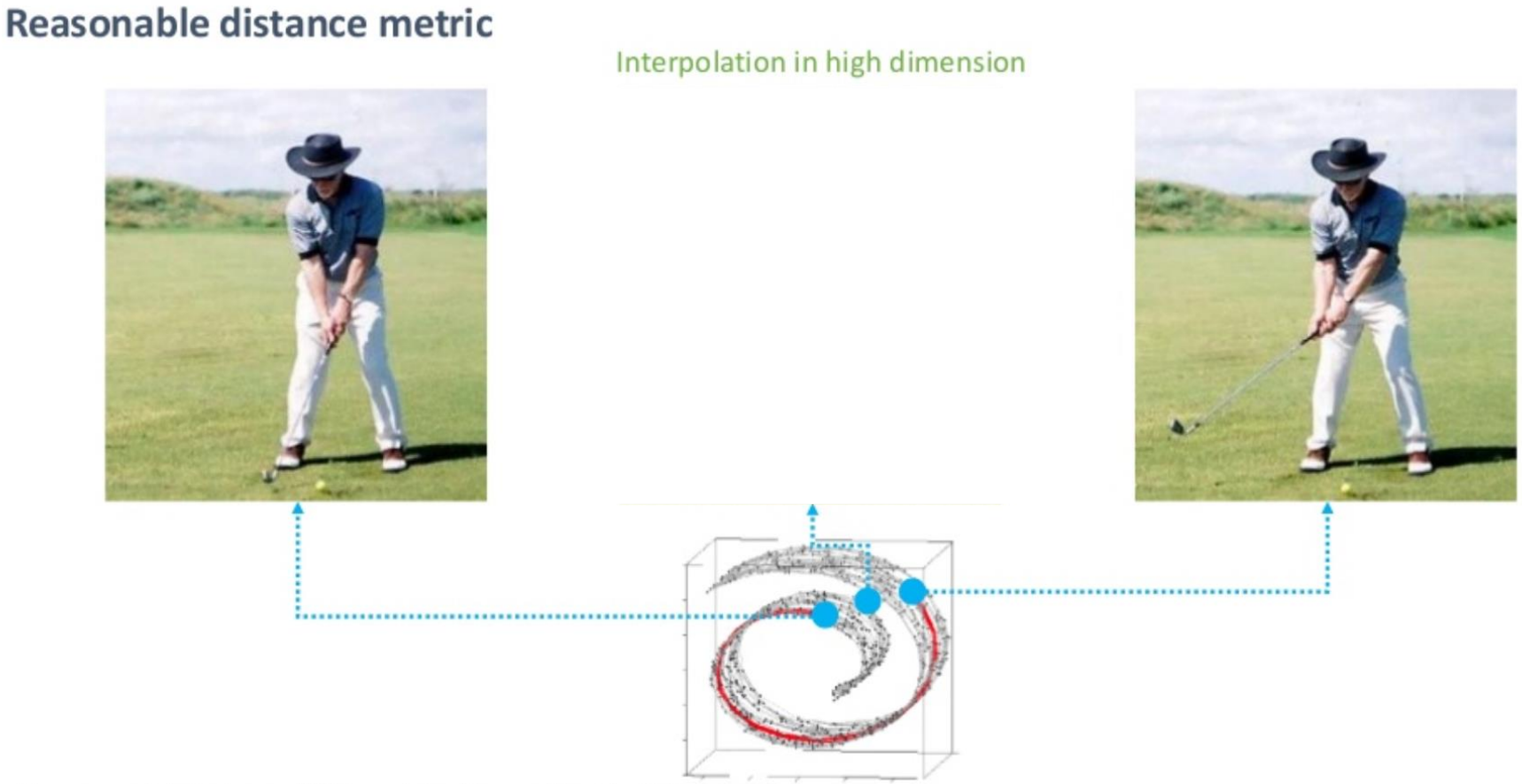
Latent Representation

- To get an intuition of the latent representation, we can pick two samples x and x' at random and interpolate samples along the line in the latent space

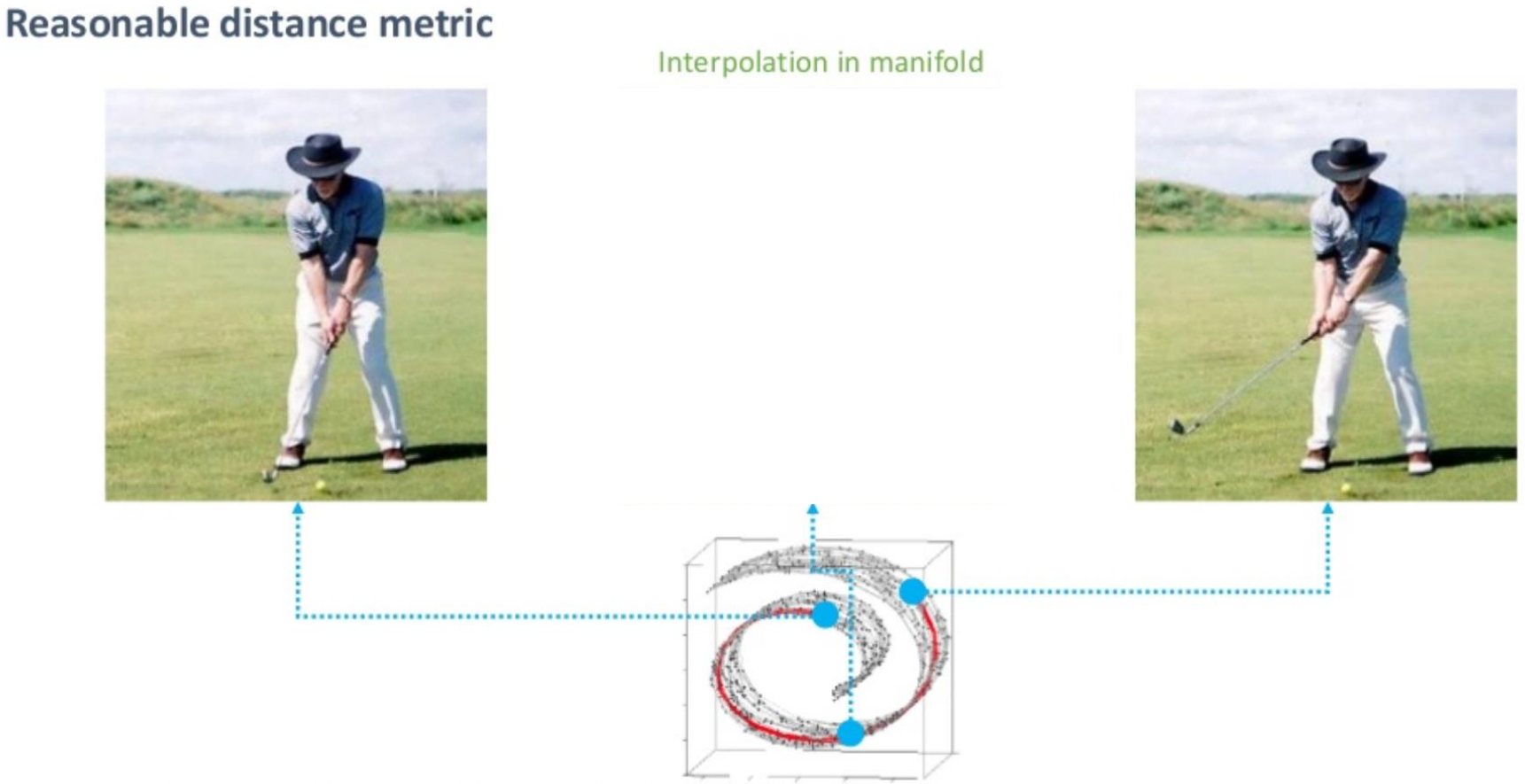
$$\forall x, x' \in \mathcal{X}^2,\ \alpha \in [0,1],\quad \xi(x, x', \alpha) = g((1-\alpha)f(x) + \alpha f(x')).$$



Interpolation in High Dimension

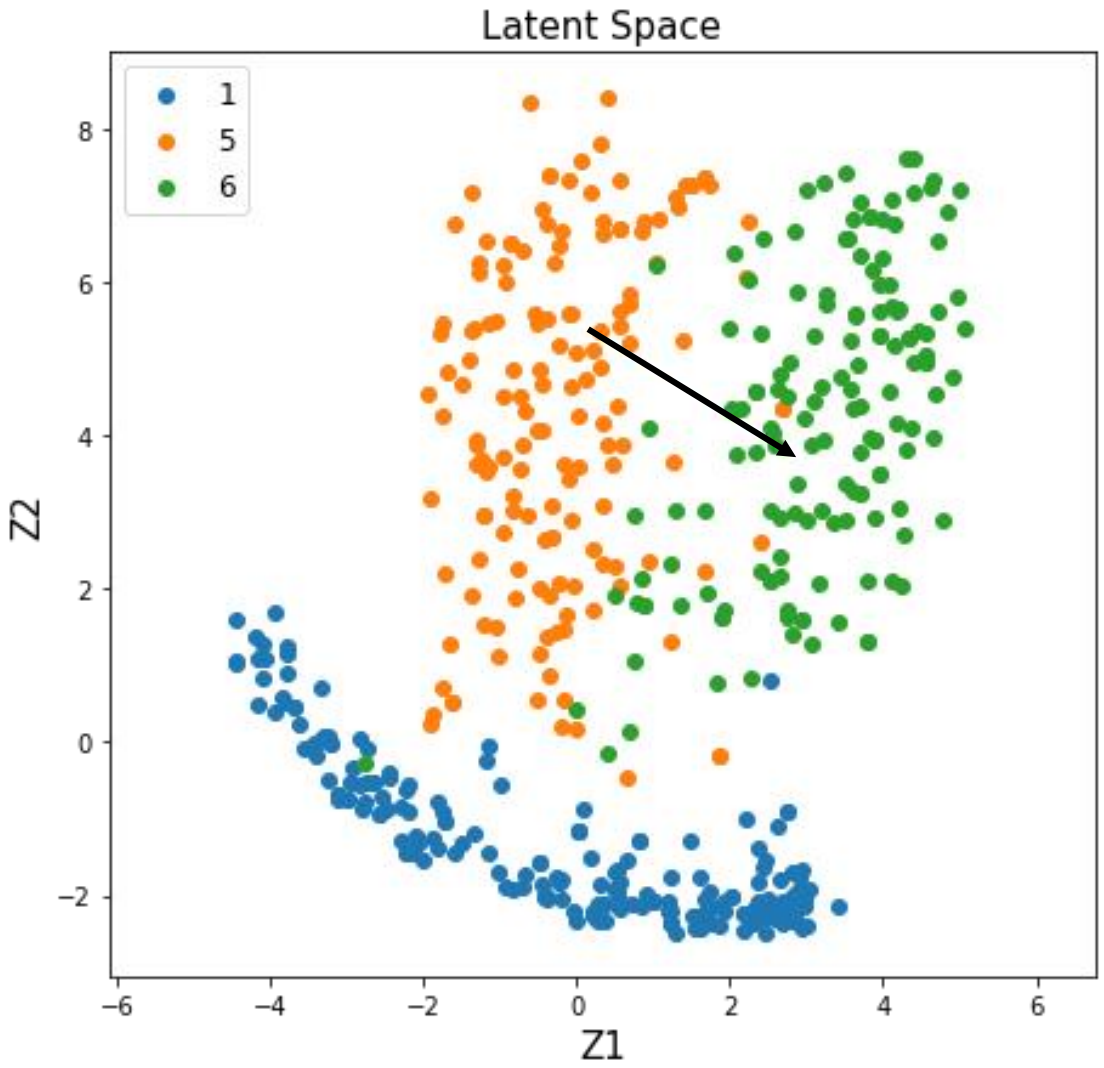


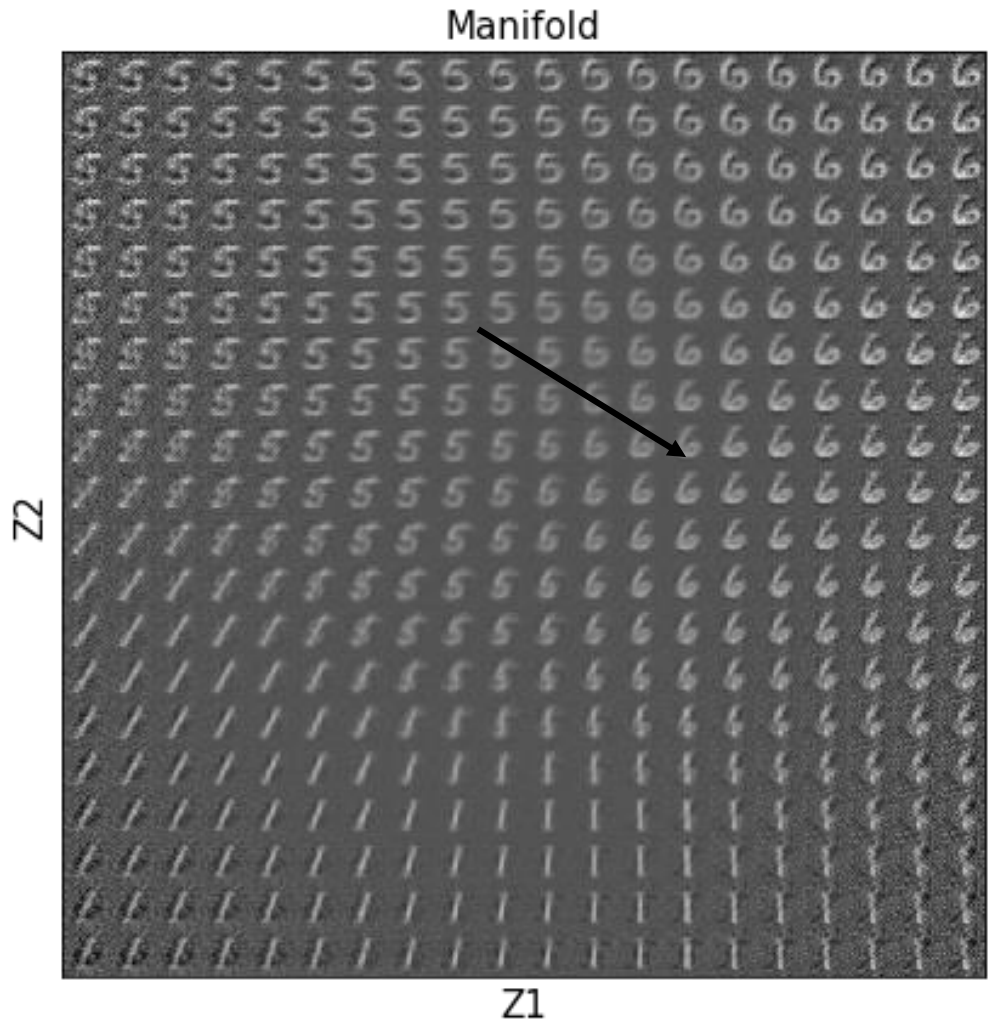
Interpolation in Manifold



https://www.cs.cmu.edu/~efros/courses/AP06/presentations/ThompsonDimensionalityReduction.pdf

MNIST Example





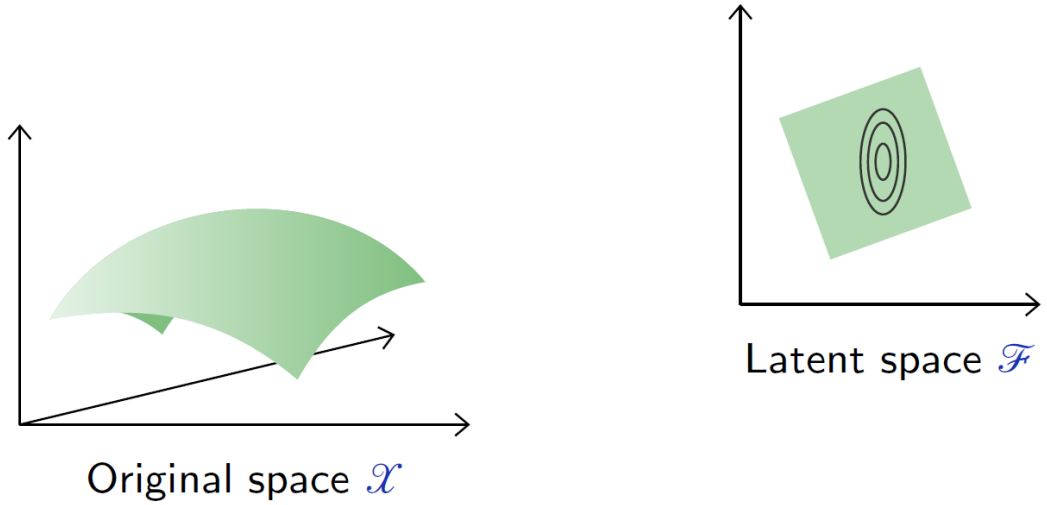
Generative Capabilities

- We can assess the generative capabilities of the decoder g by introducing a [simple] density model q^Z over the latent space $\mathcal{F}$, sample there, and map the samples into the image space $\mathcal{X}$ with g.

We can for instance use a Gaussian model with diagonal covariance matrix.

$$f(X) \sim \mathcal{N}(\hat{m}, \hat{\Delta})$$

where $\hat{m}$ is a vector and $\hat{\Delta}$ a diagonal matrix, both estimated on training data.



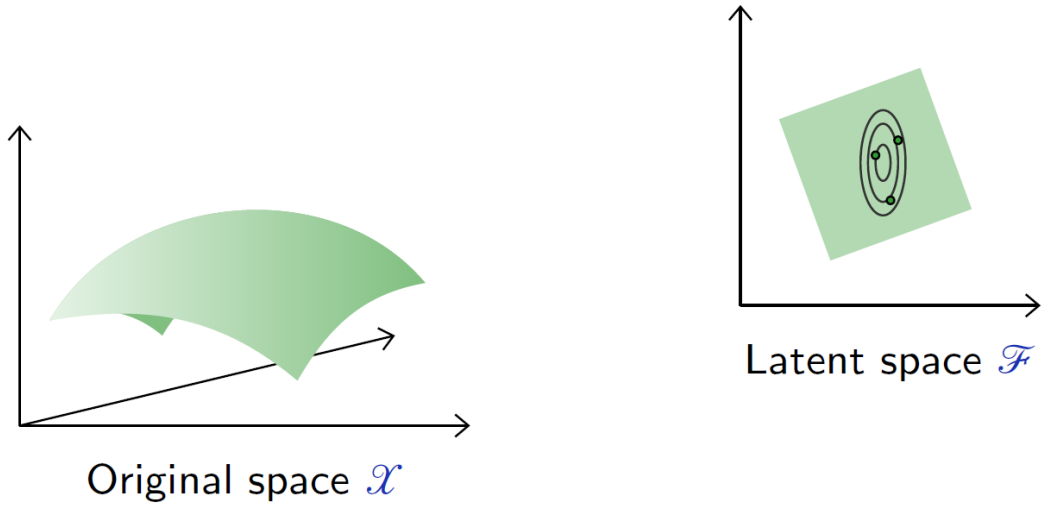
Generative Capabilities

- We can assess the generative capabilities of the decoder g by introducing a [simple] density model q^Z over the latent space $\mathcal{F}$, sample there, and map the samples into the image space $\mathcal{X}$ with g.

We can for instance use a Gaussian model with diagonal covariance matrix.

$$f(X) \sim \mathcal{N}(\hat{m}, \hat{\Delta})$$

where $\hat{m}$ is a vector and $\hat{\Delta}$ a diagonal matrix, both estimated on training data.



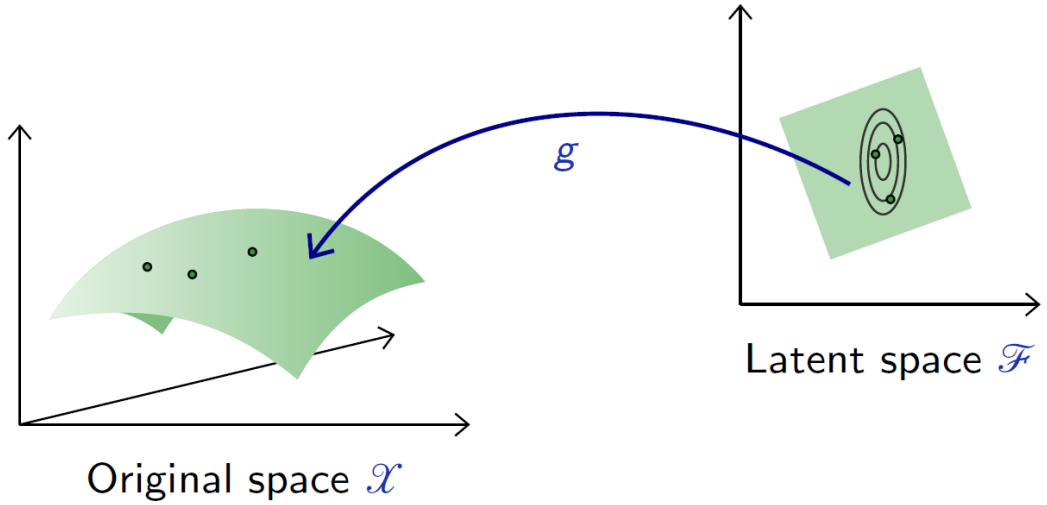
Generative Capabilities

- We can assess the generative capabilities of the decoder g by introducing a [simple] density model q^Z over the latent space $\mathcal{F}$, sample there, and map the samples into the image space $\mathcal{X}$ with g.

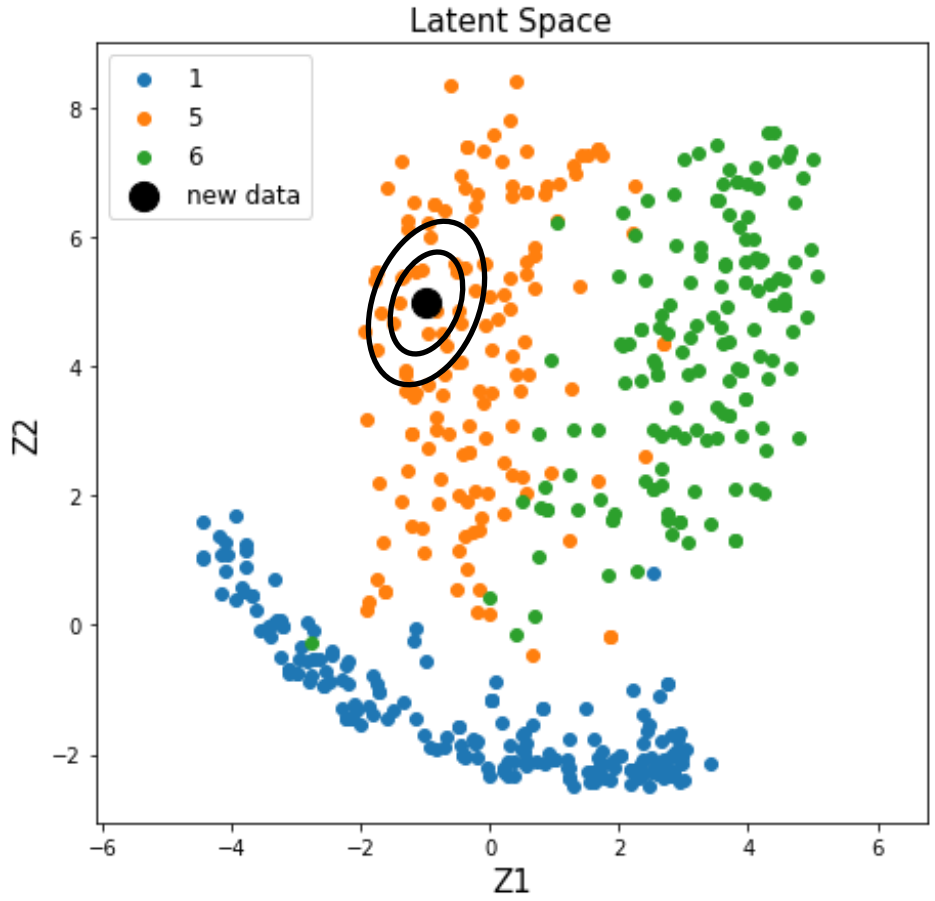
We can for instance use a Gaussian model with diagonal covariance matrix.

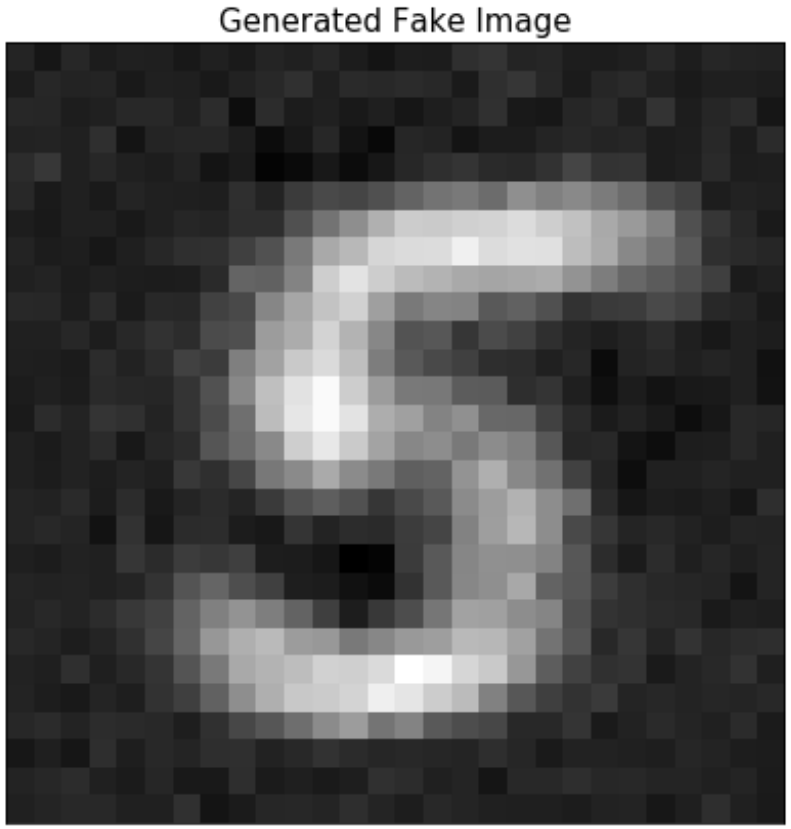
$$f(X) \sim \mathcal{N}(\hat{m}, \hat{\Delta})$$

where $\hat{m}$ is a vector and $\hat{\Delta}$ a diagonal matrix, both estimated on training data.



MNIST Example





Generative Models

- It generates something that makes sense.
- These results are unsatisfying, because the density model used on the latent space $\mathcal{F}$ is too simple and inadequate.
- Building a “good” model amounts to our original problem of modeling an empirical distribution, although it may now be in a lower dimension space.
- This is a motivation to VAE or GAN.