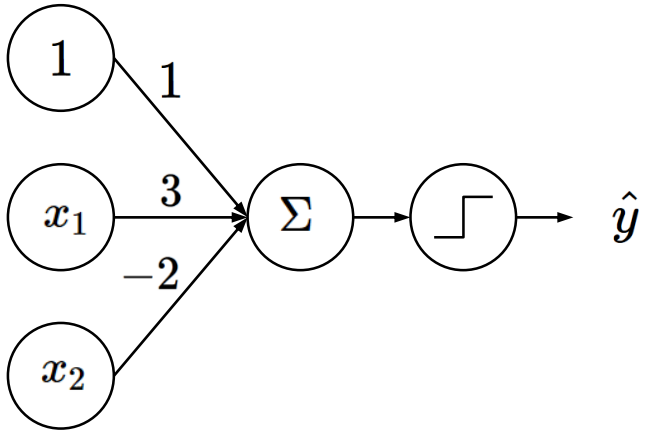
ANN & AE

Industrial AI Lab.

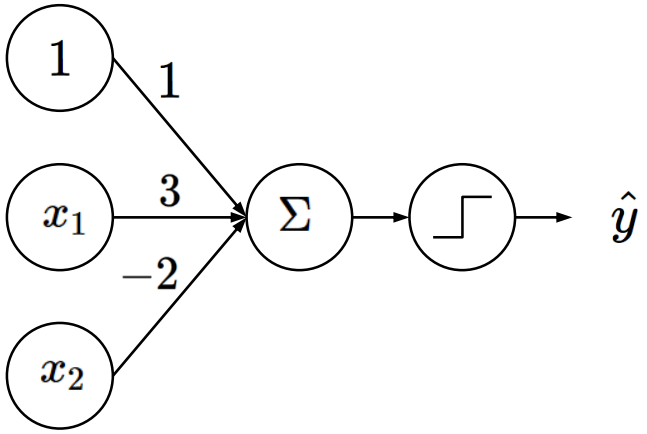
Prof. Seungchul Lee

Perceptron: Example

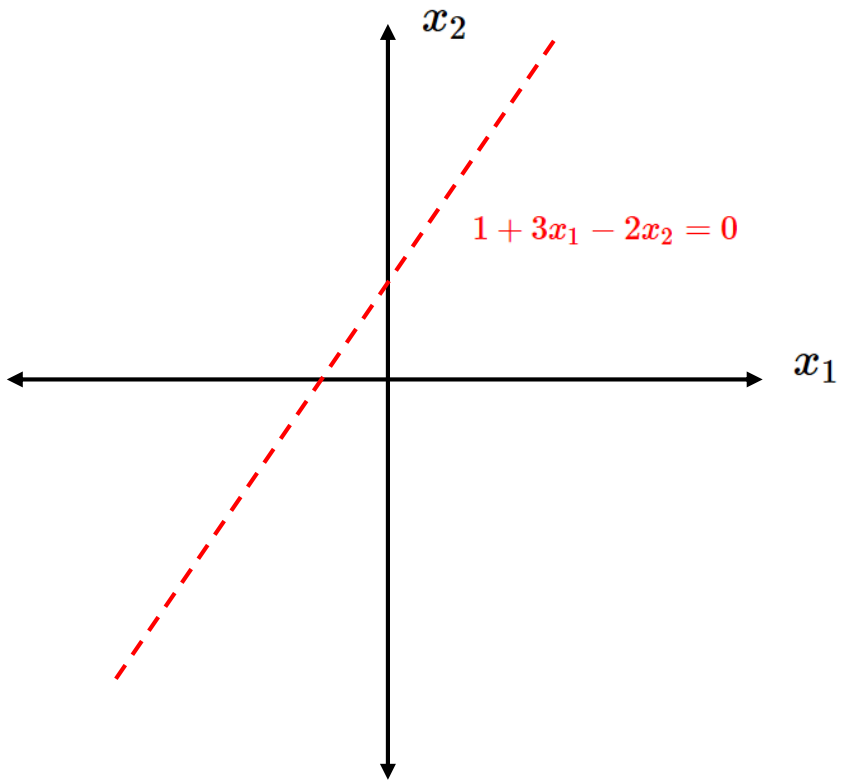


$$
\begin{aligned}
\hat{y} &= g\left(\omega_0 + X^T \omega\right) \\
&= g\left(1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 3 \\ -2 \end{bmatrix}\right) \\
&= g\left(1 + 3x_1 - 2x_2\right)
\end{aligned}
$$

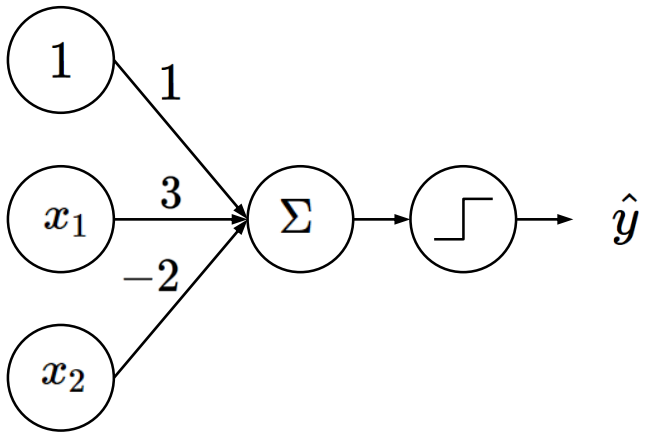
Perceptron: Example



$$\hat{y} = g(1 + 3x_1 - 2x_2)$$

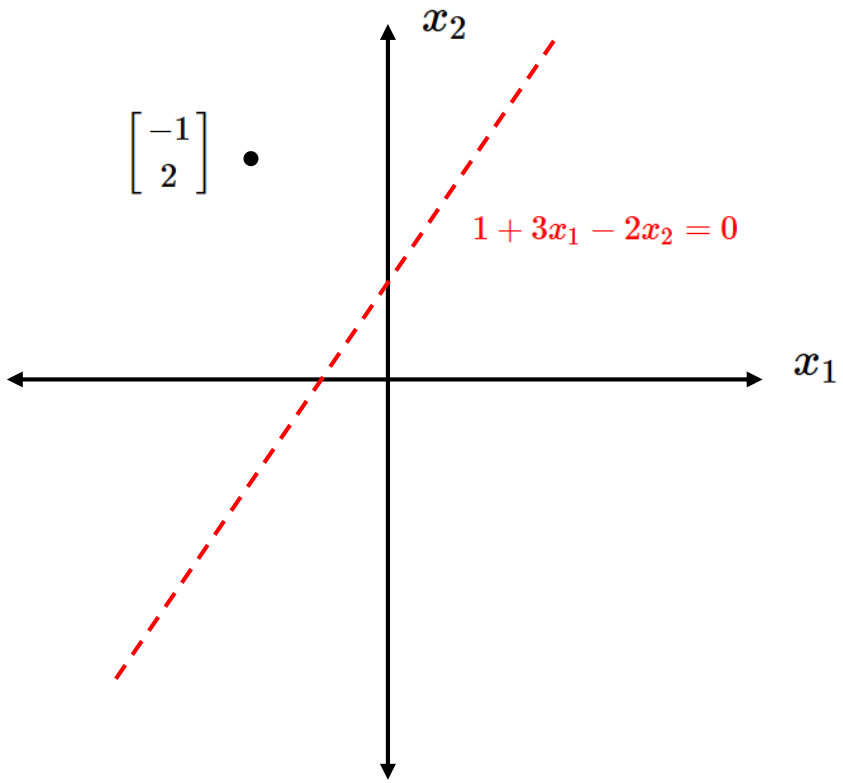


Perceptron: Example

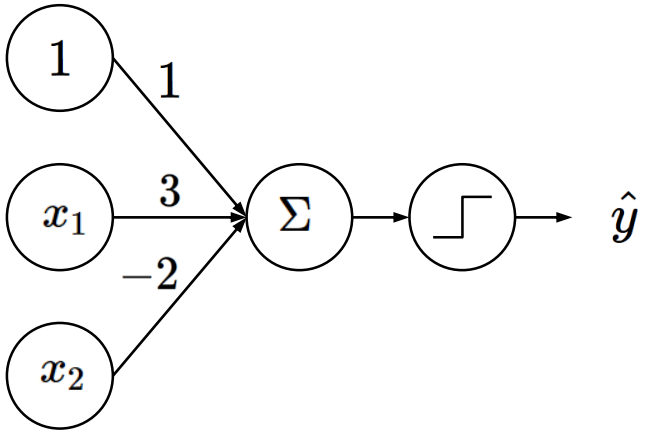


$$\hat{y} = g(1 + 3 \times (-1) - 2 \times 2) = g(-6) = -1$$

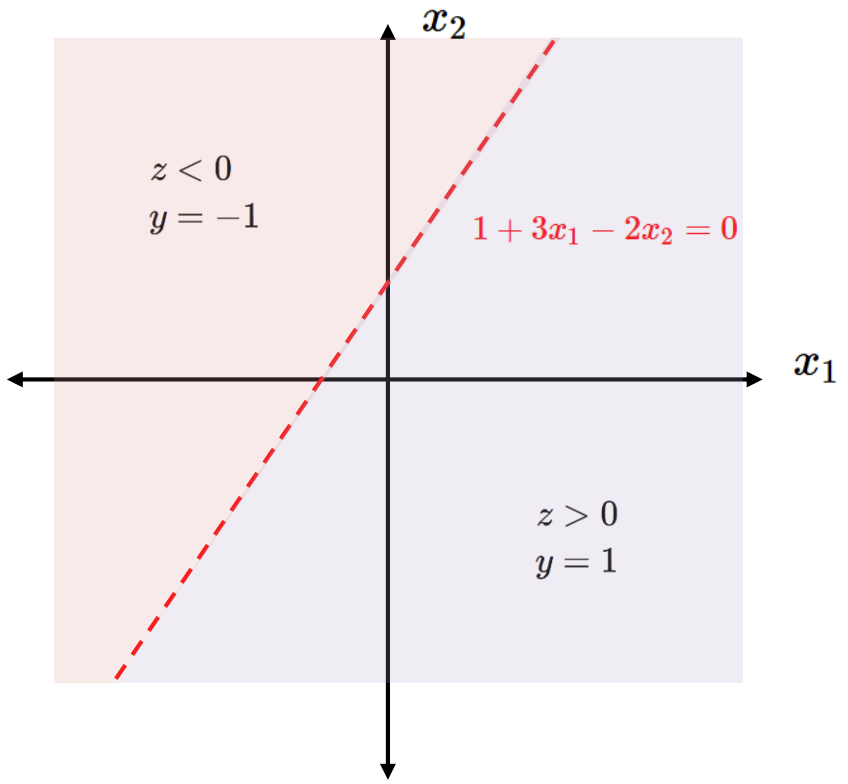
$$\hat{y} = g(1 + 3x_1 - 2x_2)$$



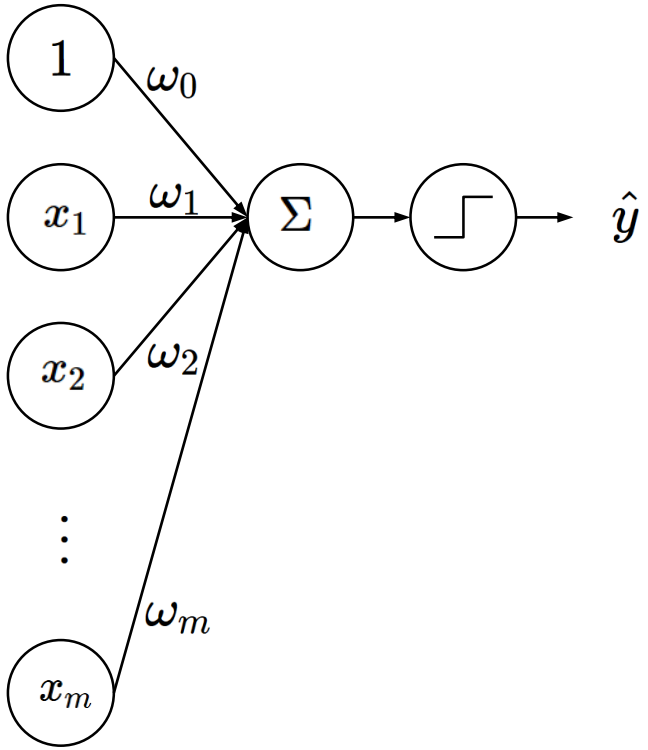
Perceptron: Example



$$\hat{y} = g(1 + 3x_1 - 2x_2)$$



Perceptron: Forward Propagation



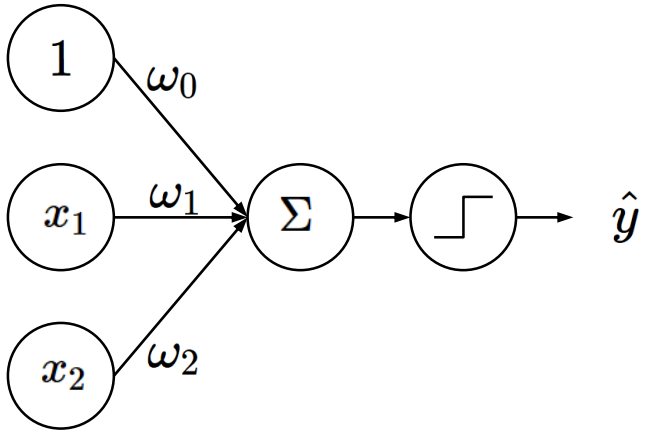
$$\hat{y} = g\left(\omega_0 + X^T\omega\right)$$

$$= g\left(\omega_0 + \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}^T \begin{bmatrix} \omega_1 \\ \vdots \\ \omega_m \end{bmatrix}\right)$$

From Perceptron to MLP

Artificial Neural Networks: Perceptron

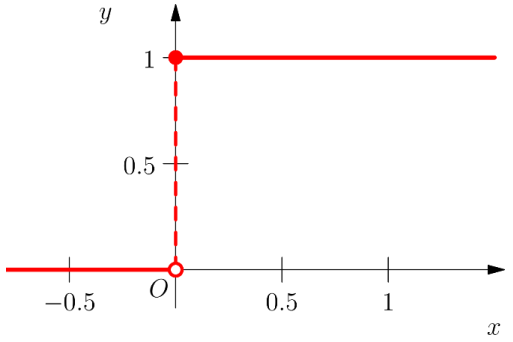
- Perceptron for $h(\theta)$ or $h(\omega)$
  - Neurons compute the weighted sum of their inputs
  - A neuron is activated or fired when the sum a is positive



$$a = \omega_0 + \omega_1 x_1 + \omega_2 x_2$$

$$\hat{y} = g(a) = \begin{cases} 1 & a > 0 \\ 0 & \text{otherwise} \end{cases}$$

- A step function is not differentiable
- One neuron is often not enough
  - One hyperplane

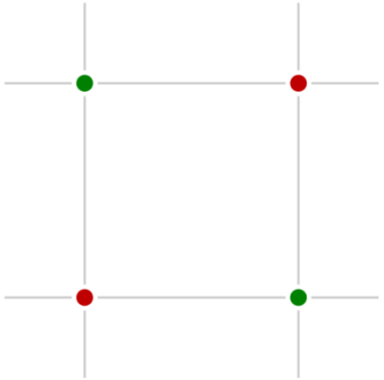


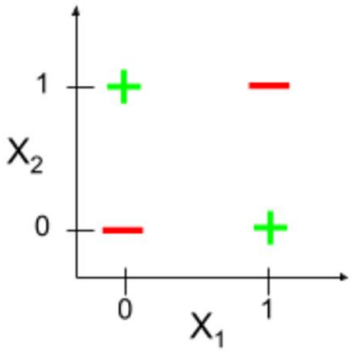
Here, a step function is illustrated instead of a sign function

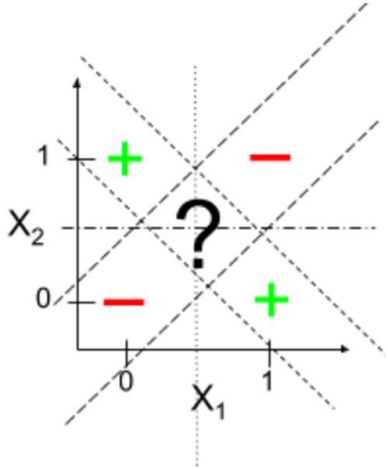
XOR Problem

- Minsky-Papert Controversy on XOR
  - Not linearly separable
  - Limitation of perceptron

x_1	x_2	x_1 **XOR** x_2
0	0	0
0	1	1
1	0	1
1	1	0



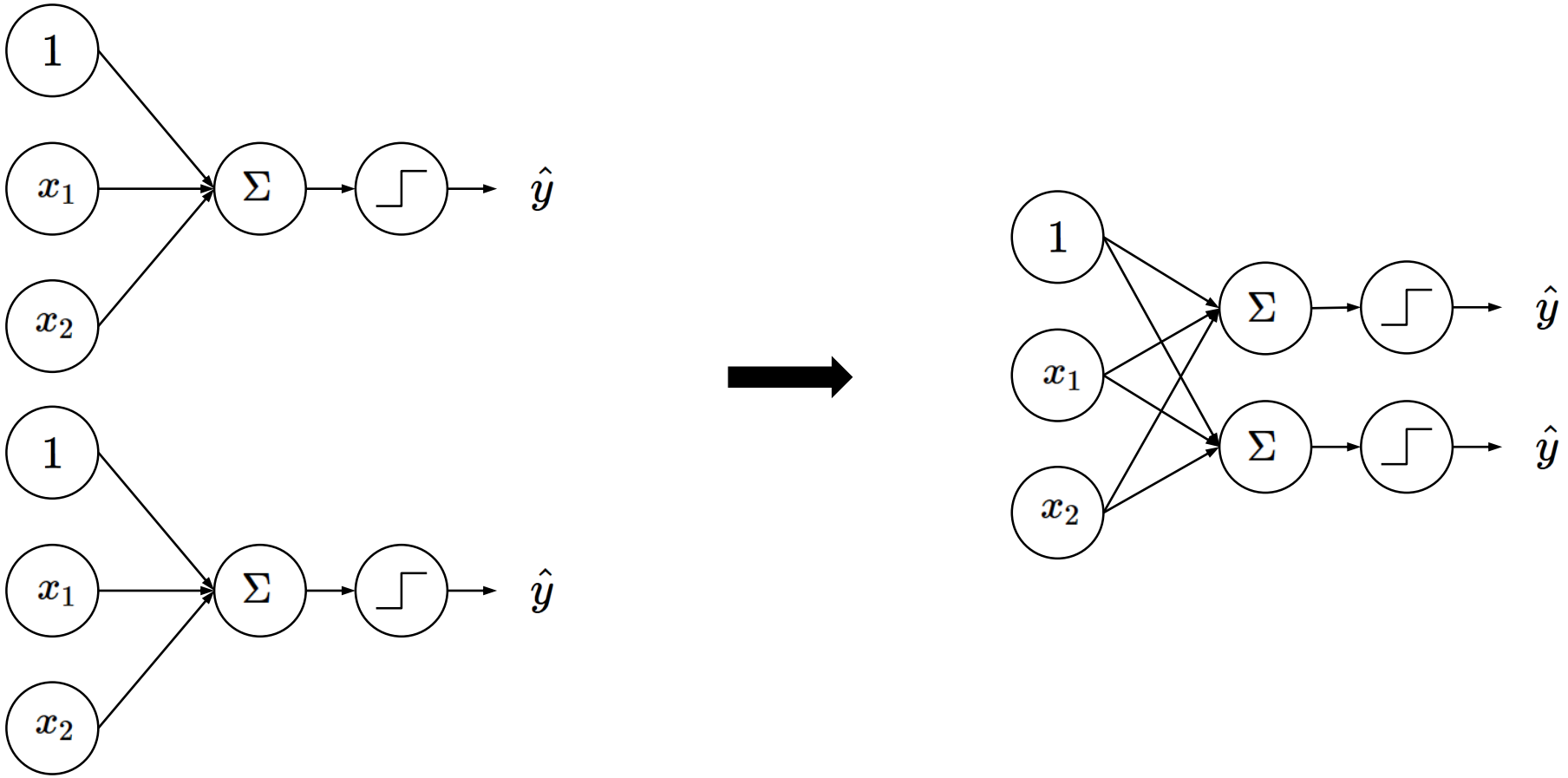




- Single neuron = one linear classification boundary

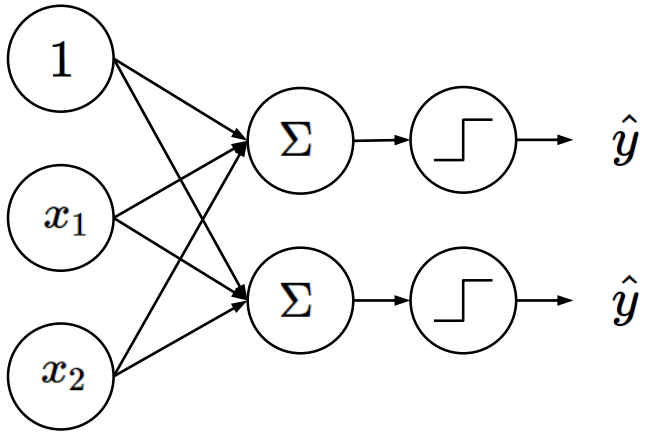
Artificial Neural Networks: MLP

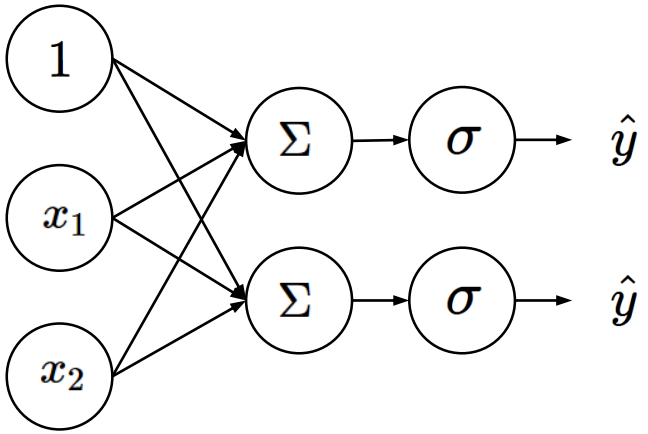
- Multi-layer Perceptron (MLP) = Artificial Neural Networks (ANN)
  - Multi neurons = multiple linear classification boundaries

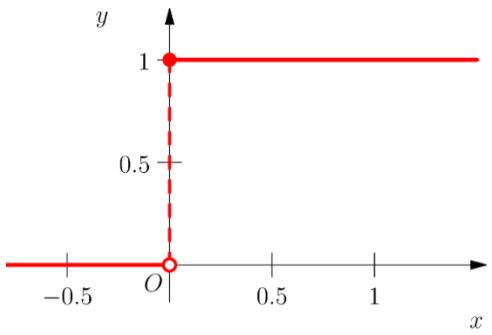


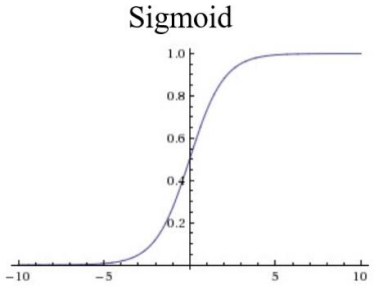
Artificial Neural Networks: Activation Function

- Differentiable nonlinear activation function







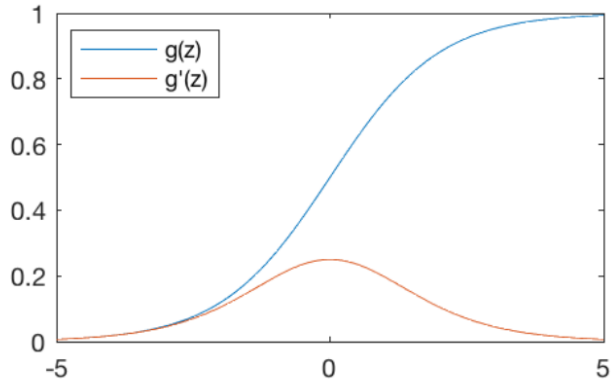


$$g(x) = \frac{1}{1 + e^{-x}}$$

Common Activation Functions

Discuss later

Sigmoid Function

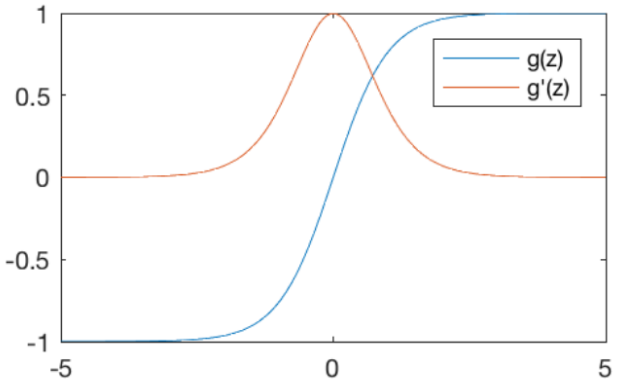


$$g(z) = \frac{1}{1 + e^{-z}}$$

$$g'(z) = g(z)(1 - g(z))$$

`tf.nn.sigmoid(z)`

Hyperbolic Tangent

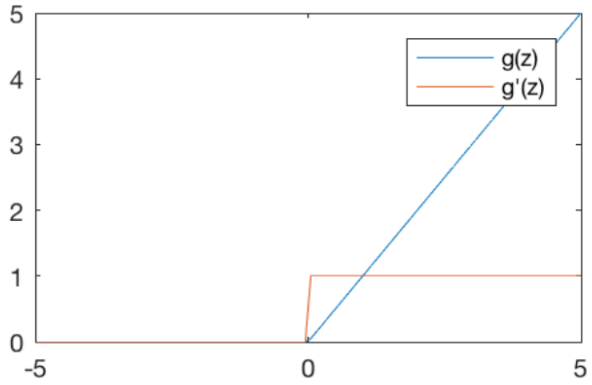


$$g(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

$$g'(z) = 1 - g(z)^2$$

`tf.nn.tanh(z)`

Rectified Linear Unit (ReLU)



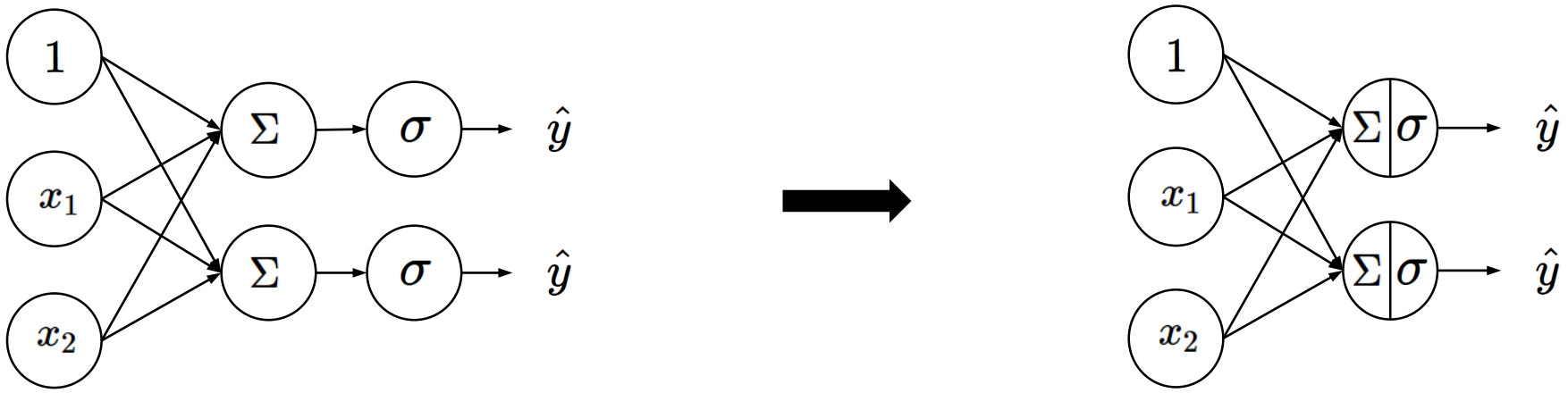
$$g(z) = \max(0, z)$$

$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

`tf.nn.relu(z)`

Artificial Neural Networks

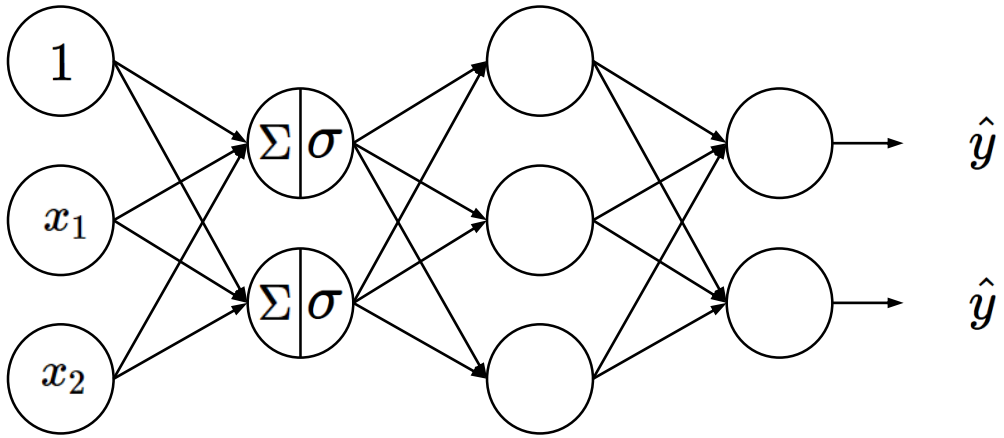
- In a compact representation



Artificial Neural Networks

- A single layer is not enough to be able to represent complex relationship between input and output

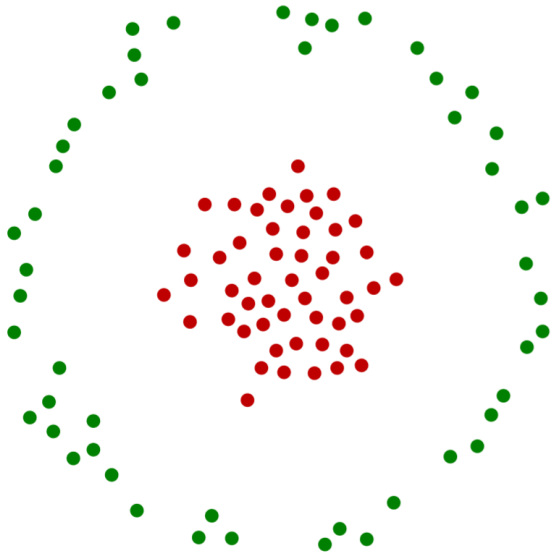
  $\Longrightarrow$ perceptron with many layers and units

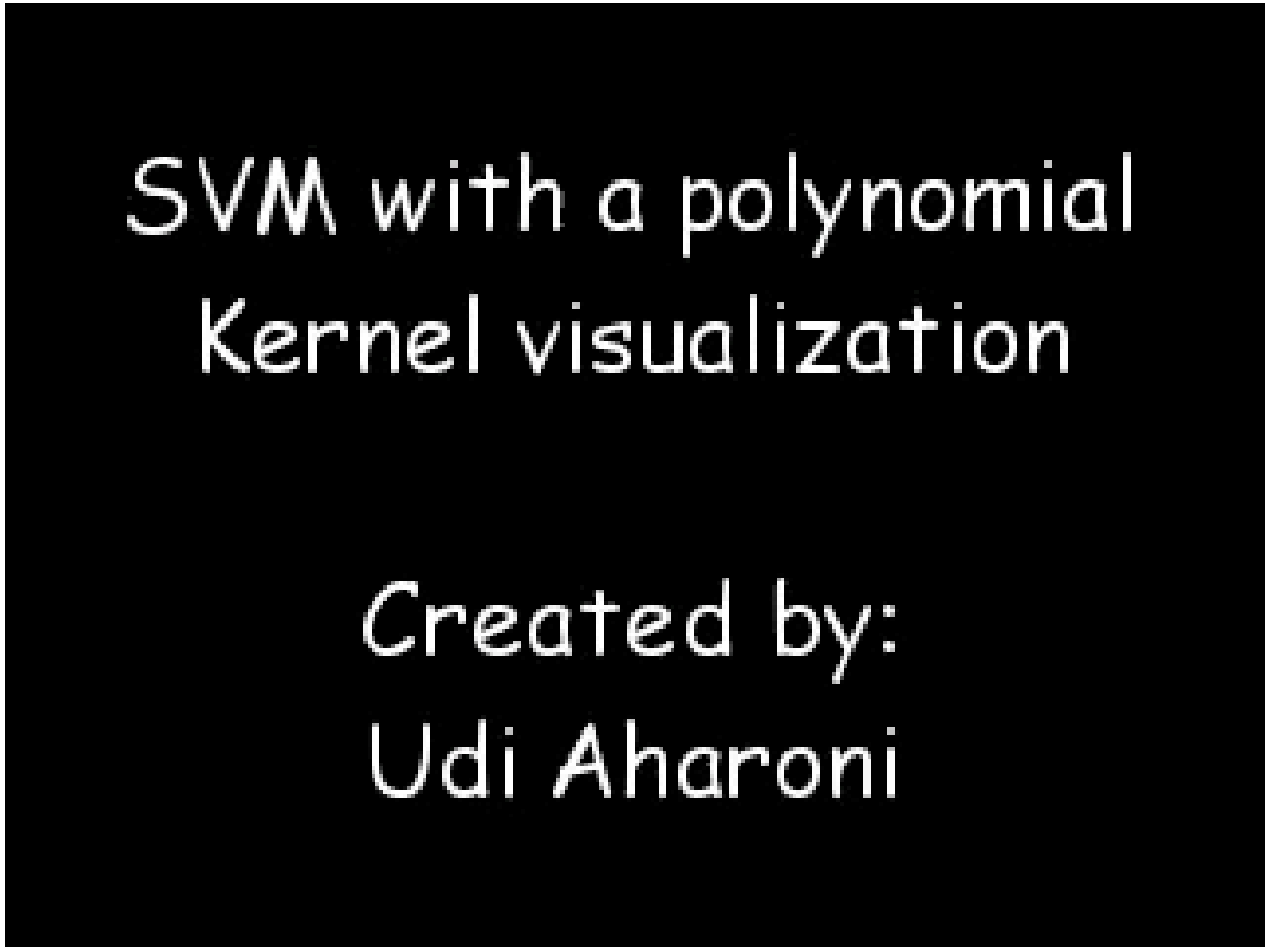


- Multi-layer perceptron
  - Features of features
  - Mapping of mappings

Another Perspective:
ANN as Kernel Learning

Nonlinear Classification

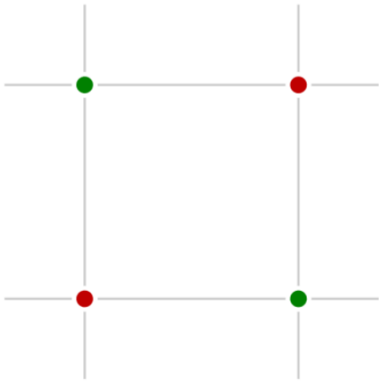




https://www.youtube.com/watch?v=3liCbRZPrZA

XOR Problem

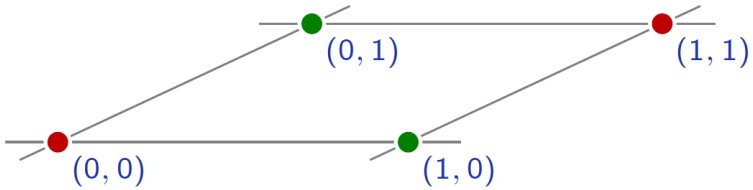
- The main weakness of linear predictors is their lack of capacity.
- For classification, the populations have to be linearly separable.



“xor”

Nonlinear Mapping

- The XOR example can be solved by pre-processing the data to make the two populations linearly separable.

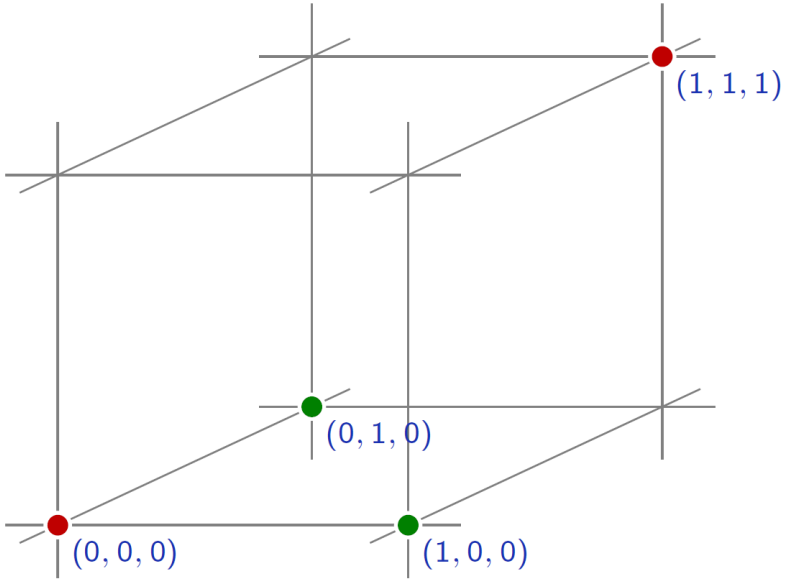


Source: Dr. Francois Fleuret at EPFL

Nonlinear Mapping

- The XOR example can be solved by pre-processing the data to make the two populations linearly separable.

$$\phi : (x_u, x_v) \rightarrow (x_u, x_v, x_u x_v)$$

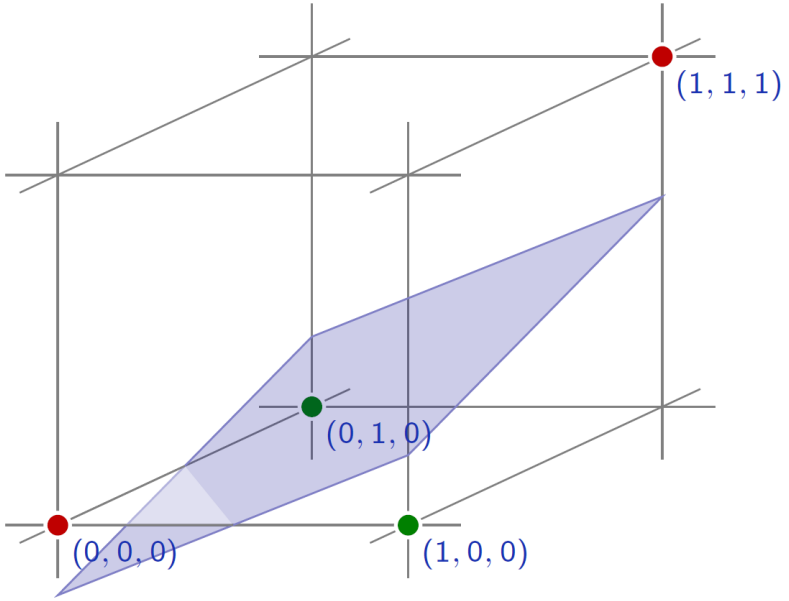


Source: Dr. Francois Fleuret at EPFL

Nonlinear Mapping

- The XOR example can be solved by pre-processing the data to make the two populations linearly separable.

$$\phi : (x_u, x_v) \rightarrow (x_u, x_v, x_u x_v)$$



Source: Dr. Francois Fleuret at EPFL

Kernel

- Often we want to capture nonlinear patterns in the data
  - nonlinear regression: input and output relationship may not be linear
  - nonlinear classification: classes may not be separable by a linear boundary

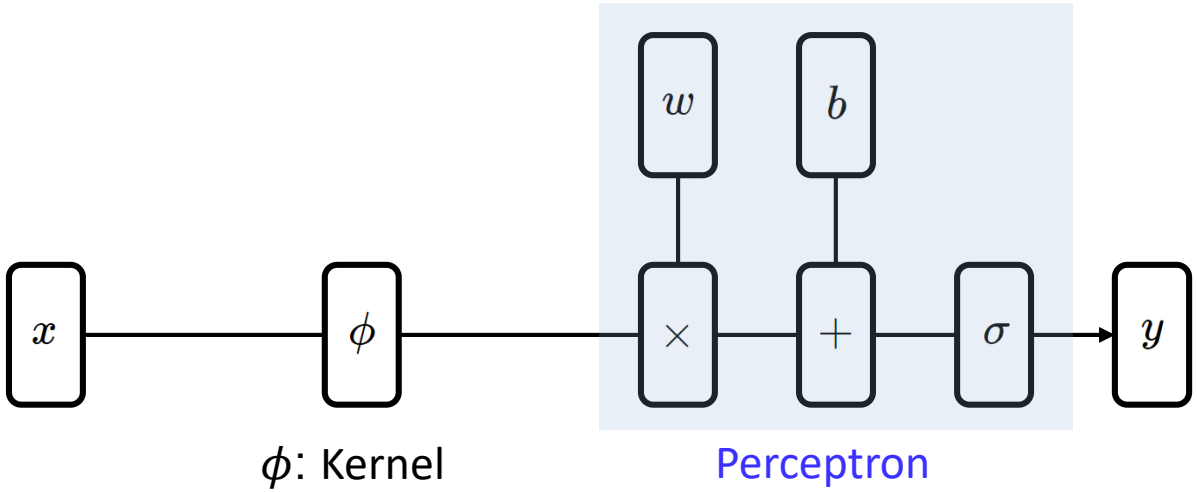
- Linear models (e.g. linear regression, linear SVM) are not just rich enough
  - by mapping data to higher dimensions where it exhibits linear patterns
  - apply the linear model in the new input feature space
  - mapping = changing the feature representation

- Kernels: make linear model work in nonlinear settings

Kernel + Neuron

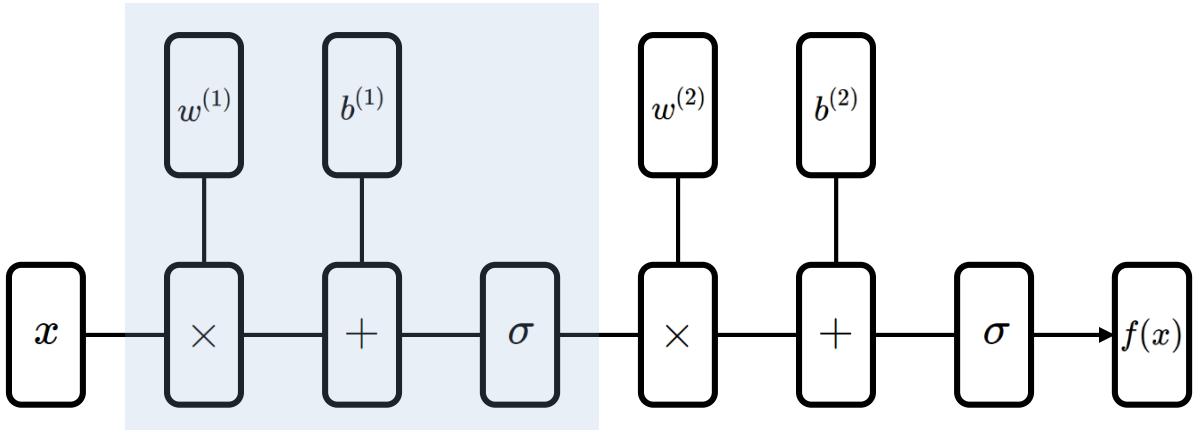
- Nonlinear mapping + neuron

$$\phi : (x_u, x_v) \rightarrow (x_u, x_v, x_u x_v)$$



Neuron + Neuron

- Nonlinear mapping can be represented by another neurons

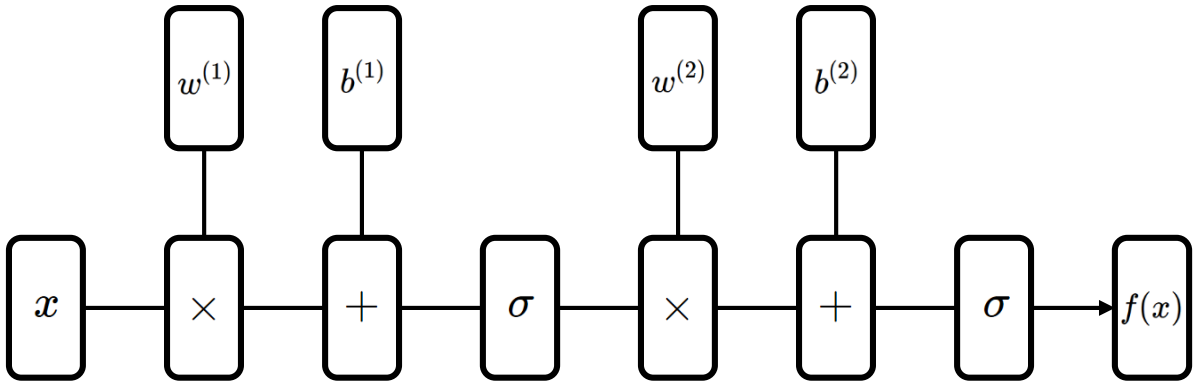


ϕ: Kernel

- Nonlinear Kernel
  - Nonlinear activation functions

Multi Layer Perceptron

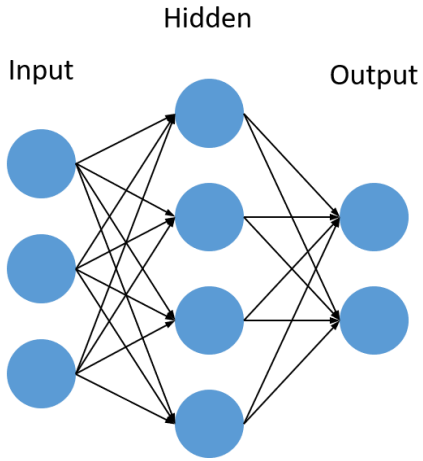
- Nonlinear mapping can be represented by another neurons
- We can generalize an MLP



ϕ: Kernel

Summary

- Universal function approximator
- Universal function classifier
- Parameterized

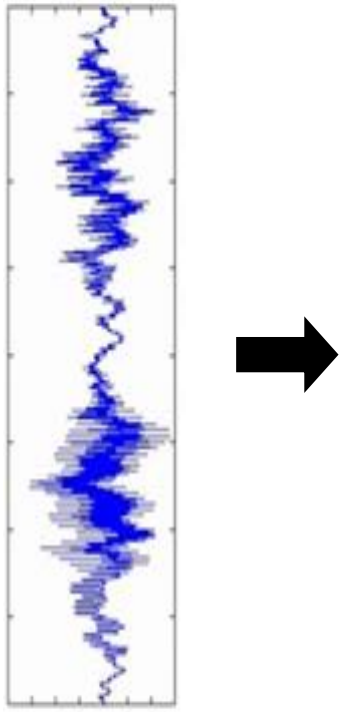


$$x \rightarrow \boxed{\hat{y} = f_{\omega_1,\cdots,\omega_k}(x)} \rightarrow y$$

Artificial Neural Networks

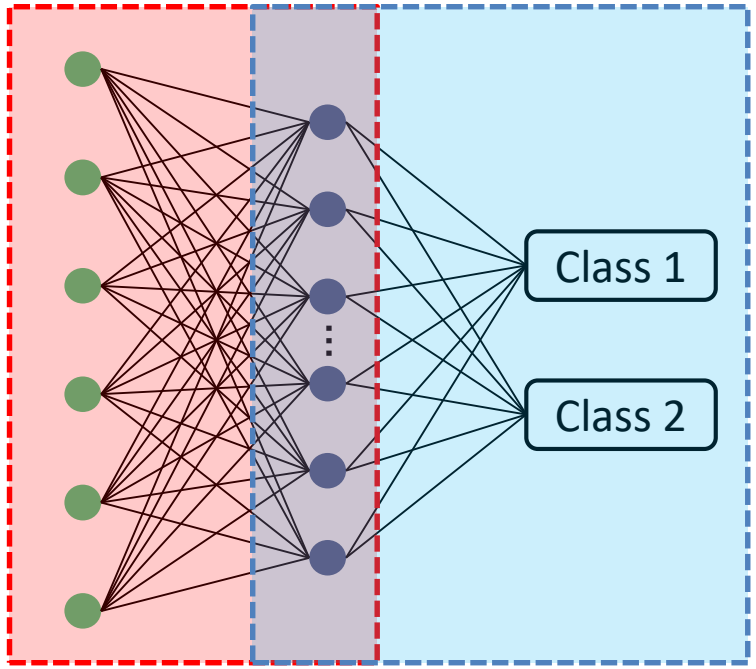
- Complex/Nonlinear universal function approximator
  - Linearly connected networks
  - Simple nonlinear neurons

Input



Output

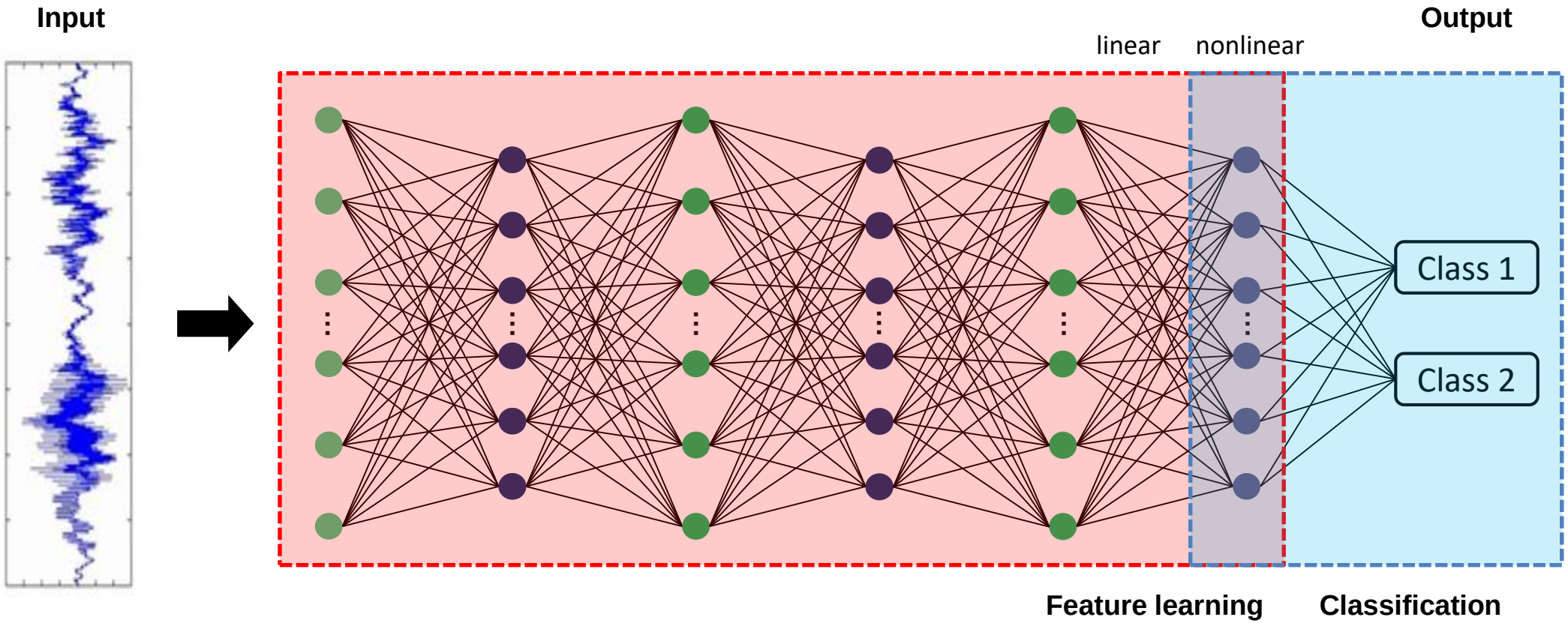
linear nonlinear



Feature learning **Classification**

Deep Artificial Neural Networks

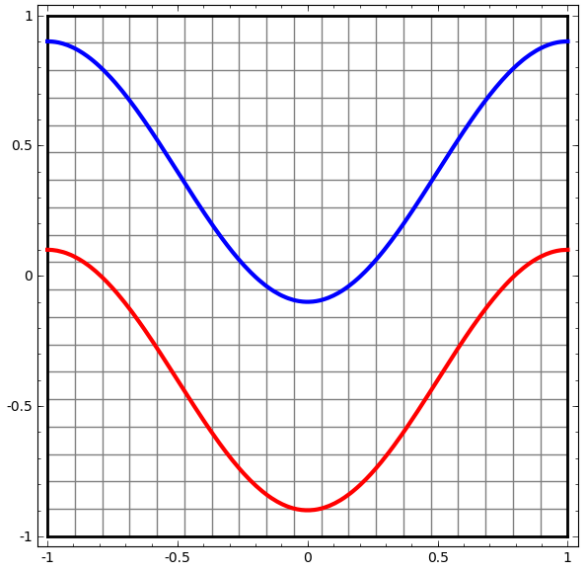
- Complex/Nonlinear universal function approximator
  - Linearly connected networks
  - Simple nonlinear neurons

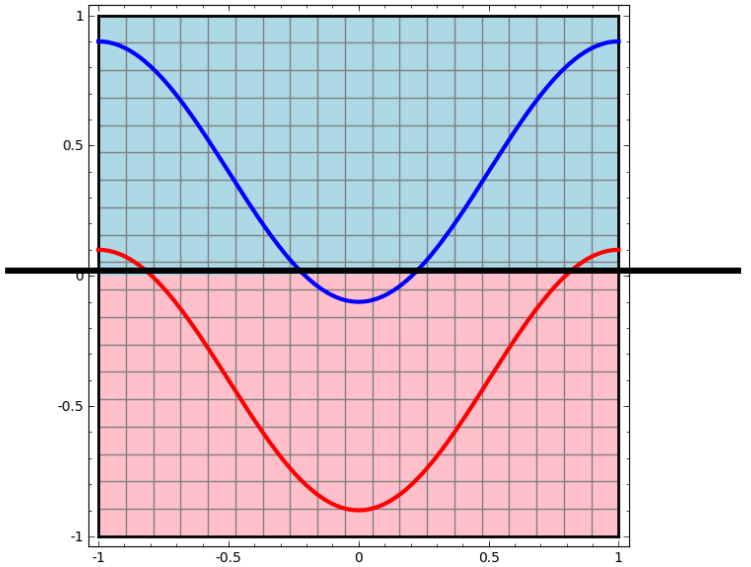


Looking at Parameters

Example: Linear Classifier

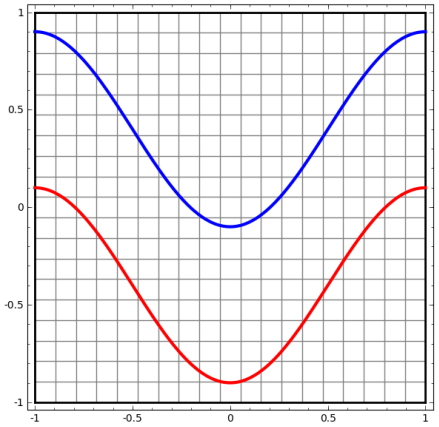
- Perceptron tries to separate the two classes of data by dividing them with a line

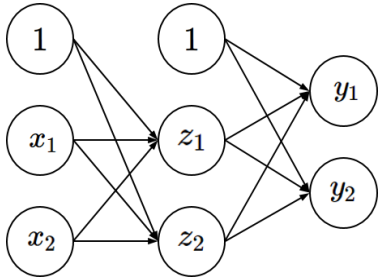


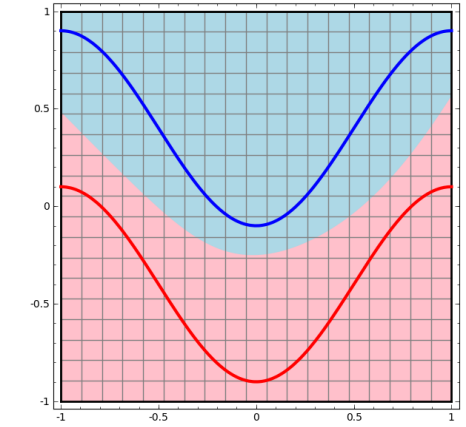


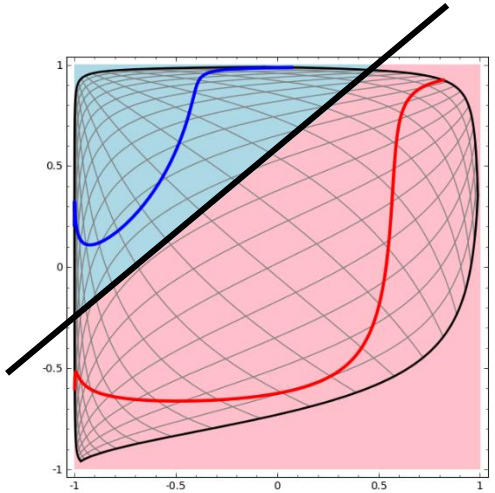
Example: Neural Networks

- The hidden layer learns a representation so that the data gets linearly separable

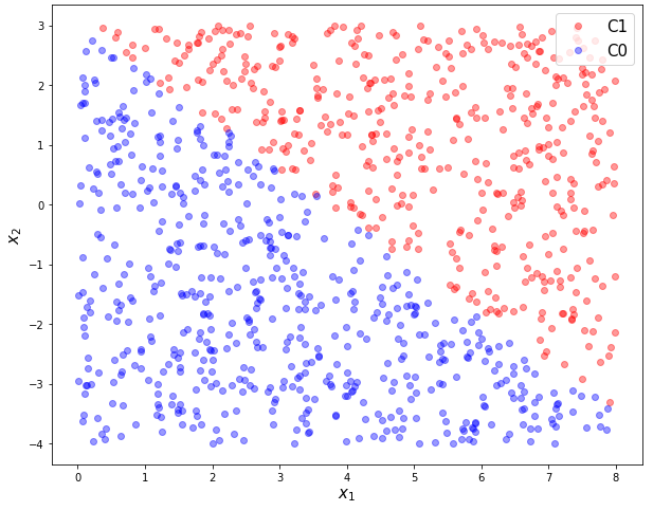




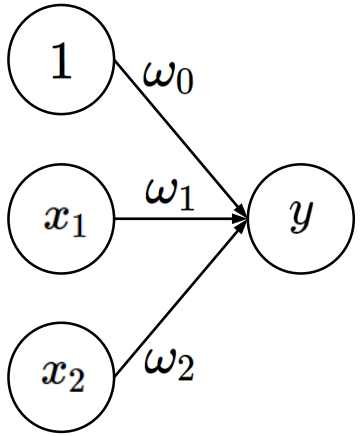


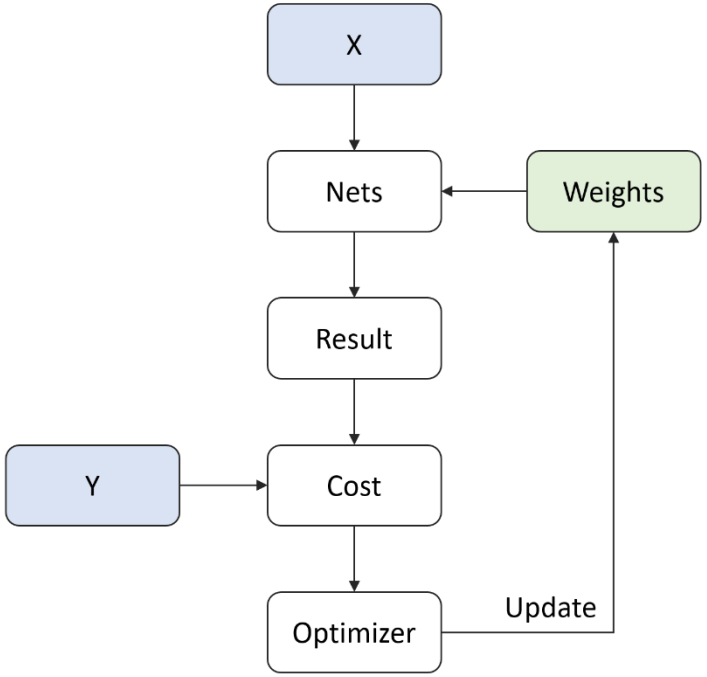


Logistic Regression in a Form of Neural Network



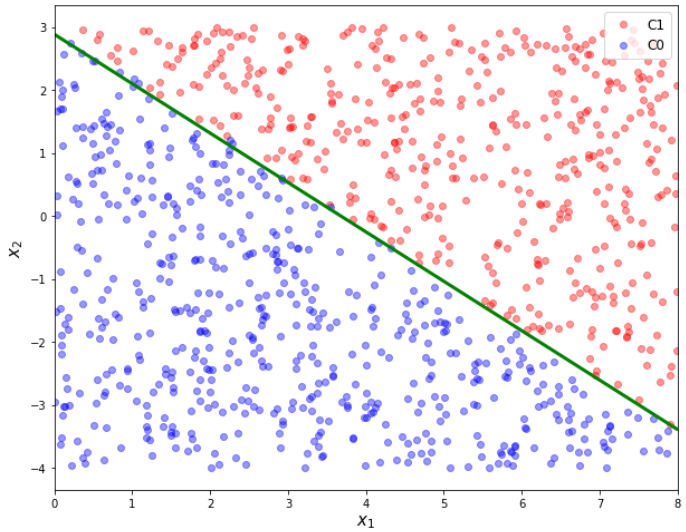
$$y = \sigma\left(\omega_0 + \omega_1 x_1 + \omega_2 x_2\right)$$



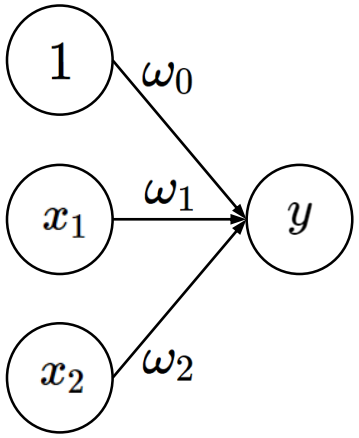


Logistic Regression in a Form of Neural Network

- Neural network convention

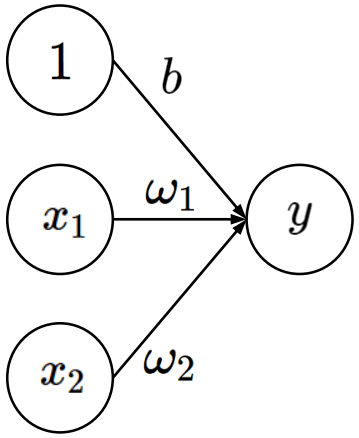


$$y = \sigma\left(\omega_0 + \omega_1 x_1 + \omega_2 x_2\right)$$

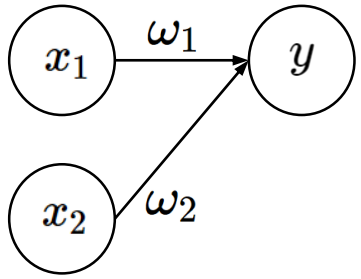


$\omega_0 \longrightarrow b$

$$y = \sigma\left(b + \omega_1 x_1 + \omega_2 x_2\right)$$



Do not indicate bias units

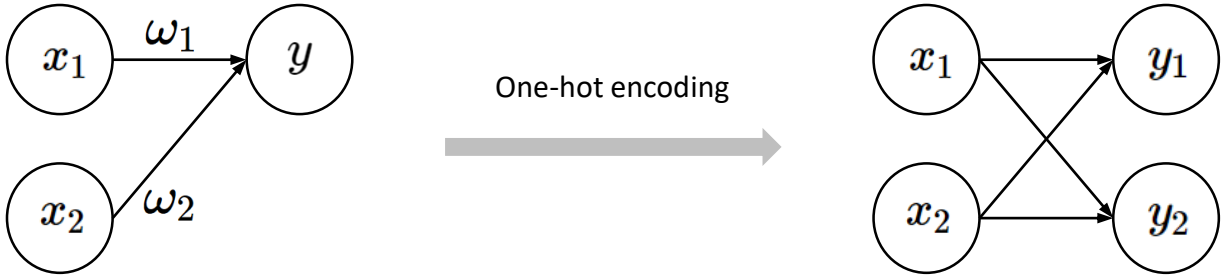



```
n_input = 2
n_output = 1
```

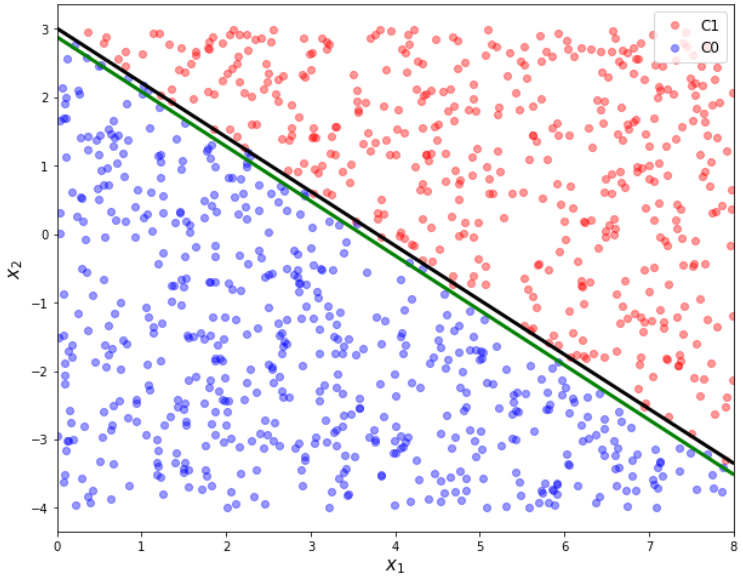
Logistic Regression in a Form of Neural Network

- One-hot encoding
  - One-hot encoding is a conventional practice for a multi-class classification

$$y^{(i)} \in \{1, 0\} \quad \Longrightarrow \quad y^{(i)} \in \{[0, 1], [1, 0]\}$$

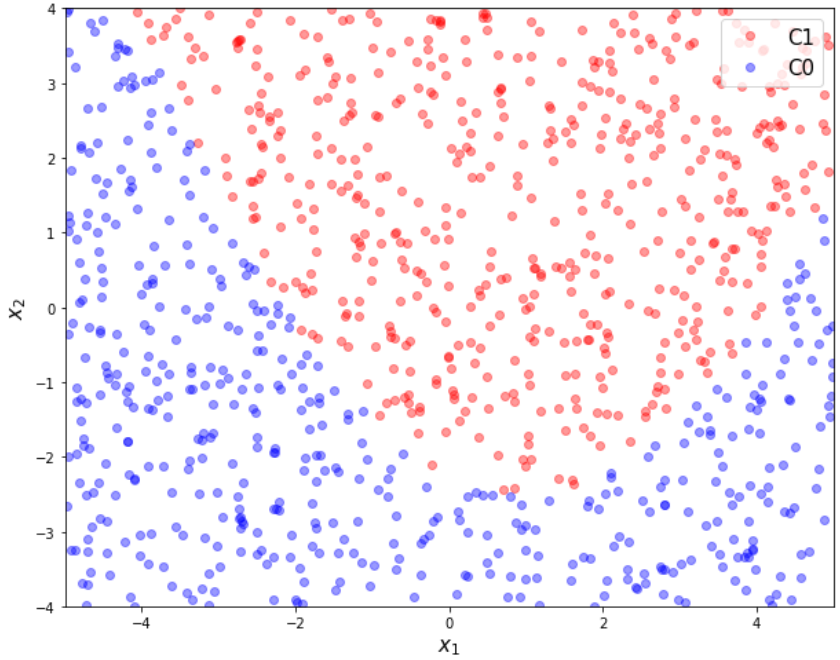


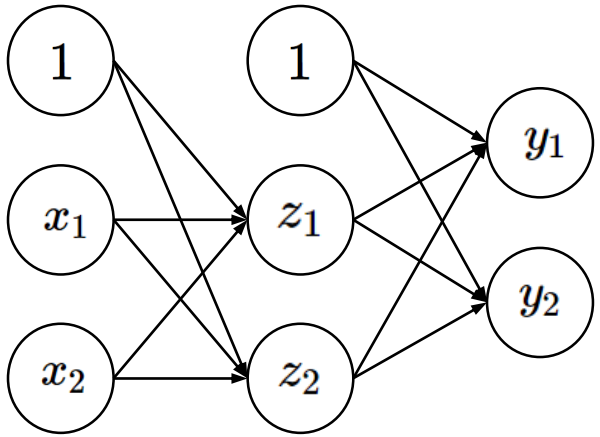

```
n_input = 2
n_output = 2
```



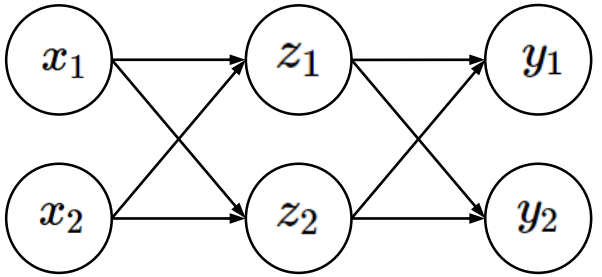
Nonlinearly Distributed Data

- Example to understand network's behavior
  - Include a hidden layer





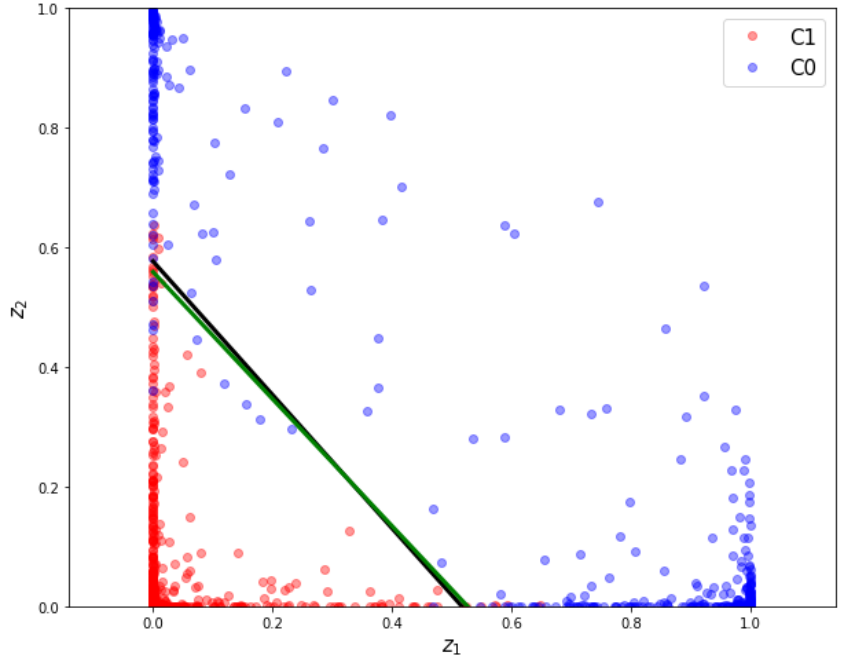
Do not include bias units

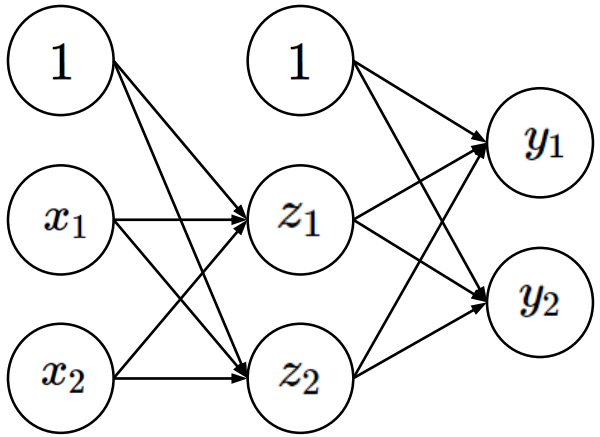



```
n_input = 2
n_hidden = 2
n_output = 2
```

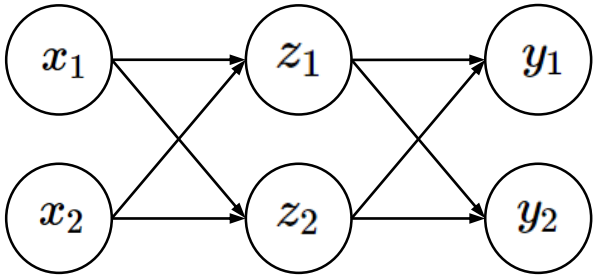
Multi Layers

- z space





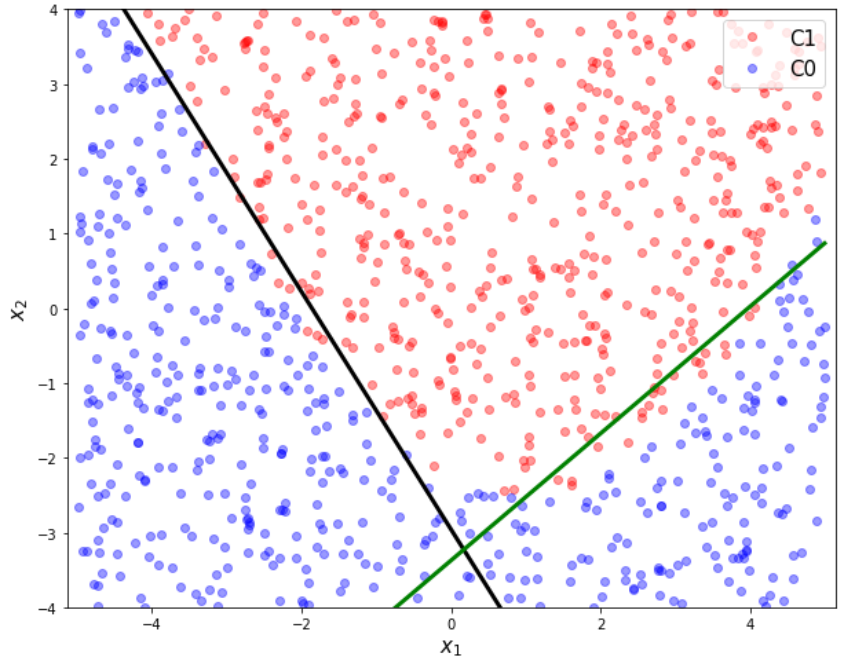
Do not include bias units

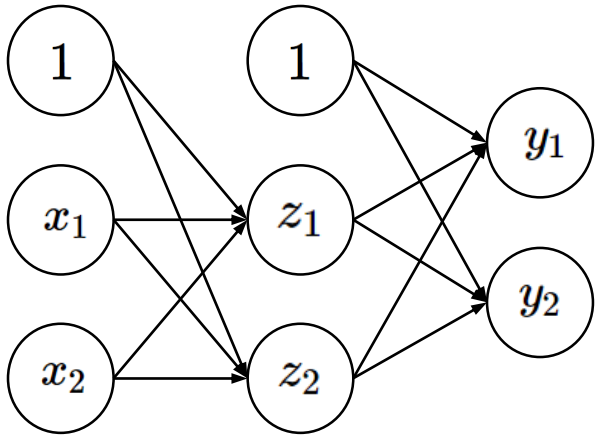



```
n_input = 2
n_hidden = 2
n_output = 2
```

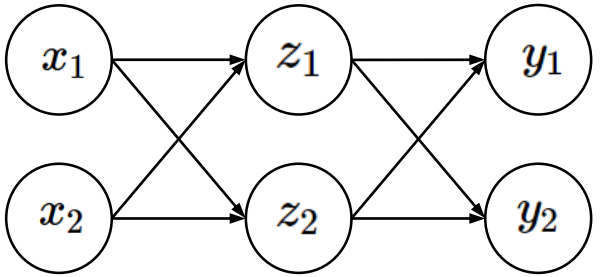
Multi Layers

- x space





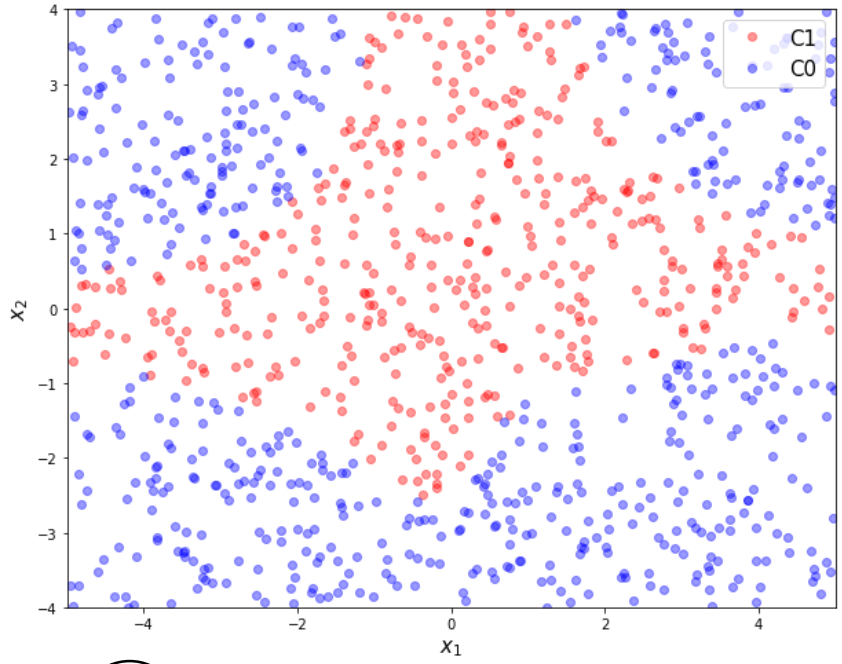
Do not include bias units

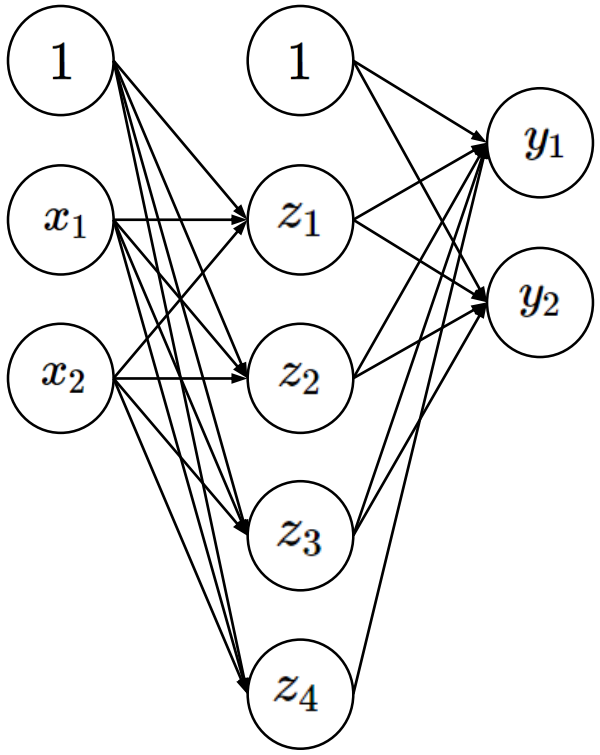



```
n_input = 2
n_hidden = 2
n_output = 2
```

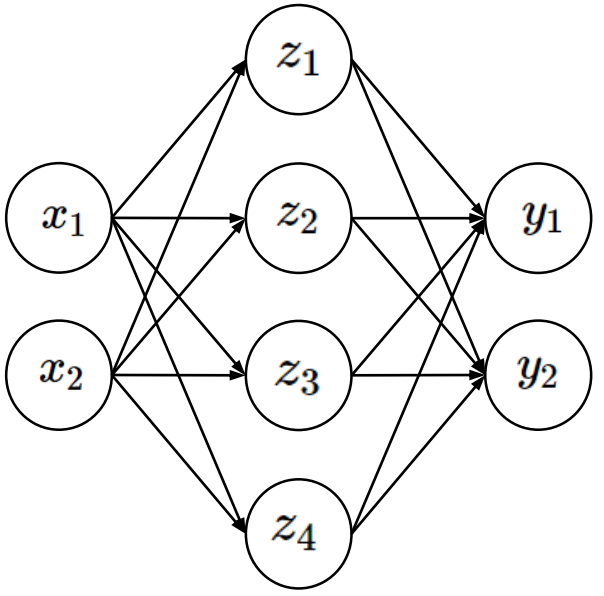
Nonlinearly Distributed Data

- More neurons in hidden layer





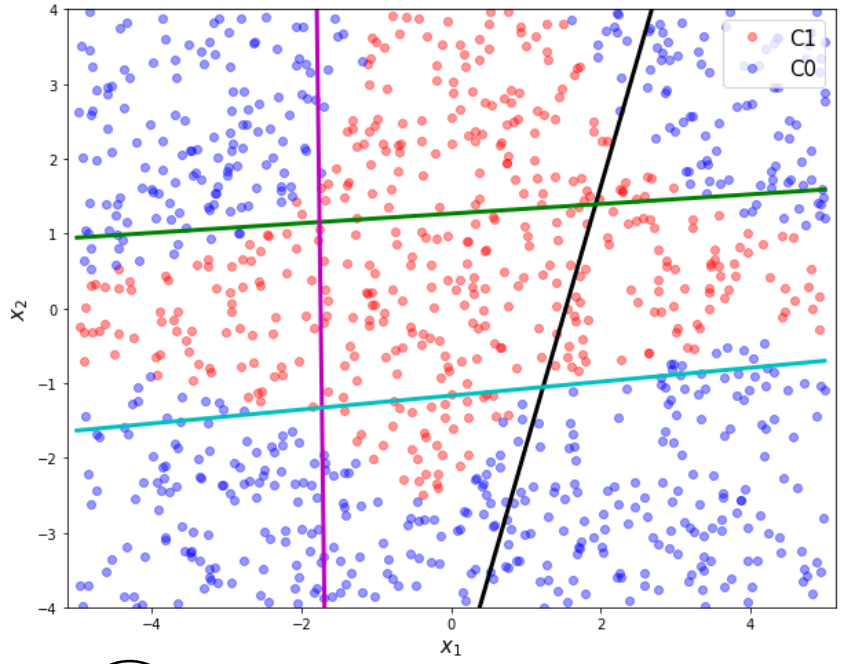
Do not include bias units

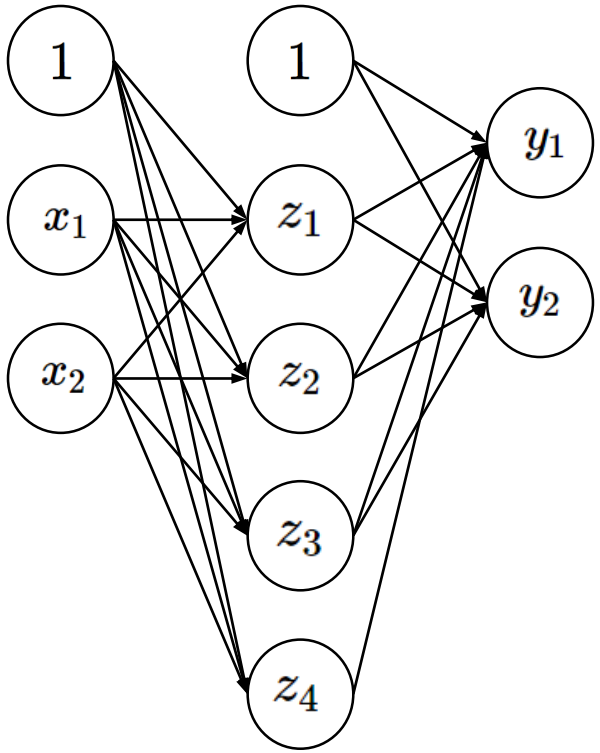



```
n_input = 2
n_hidden = 4
n_output = 2
```

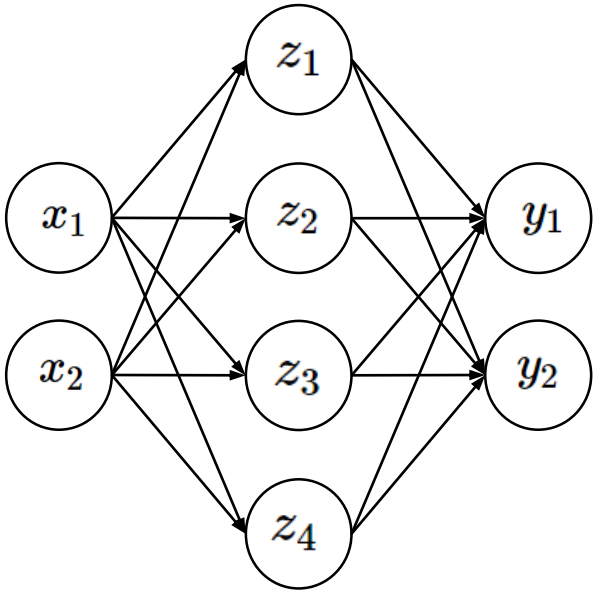
Multi Layers

- Multiple linear classification boundaries





Do not include bias units




```
n_input = 2
n_hidden = 4
n_output = 2
```

Training

Training Neural Networks: Optimization

- Learning or estimating weights and biases of multi-layer perceptron from training data

- 3 key components
  - objective function $f(\cdot)$
  - decision variable or unknown ω
  - constraints $g(\cdot)$

- In mathematical expression

$$\min_{\omega} \quad f(\omega)$$

Training Neural Networks: Loss Function

- Measures error between target values and predictions

$$\min_{\omega} \sum_{i=1}^{m} \ell\left(h_{\omega}\left(x^{(i)}\right), y^{(i)}\right)$$

- Example
  - Squared loss (for regression):

$$\frac{1}{m} \sum_{i=1}^{m} \left(h_{\omega}\left(x^{(i)}\right) - y^{(i)}\right)^2$$

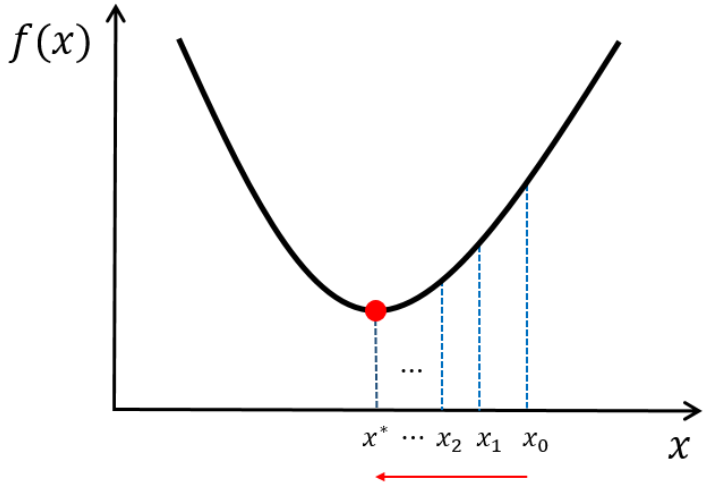
  - Cross entropy (for classification):

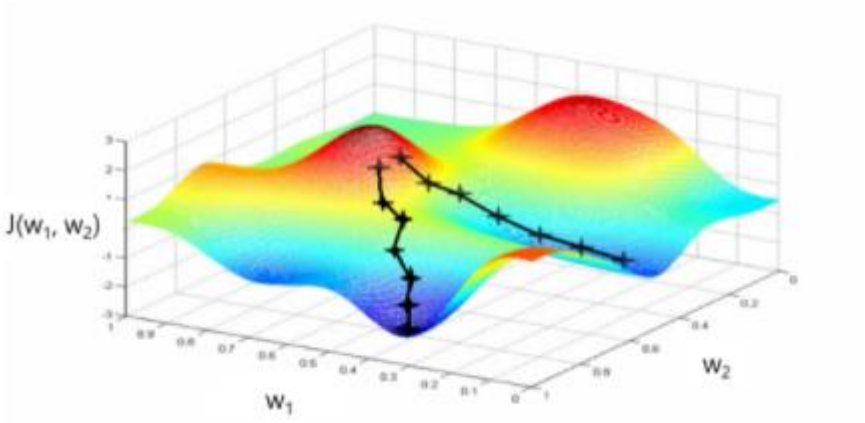
$$-\frac{1}{m} \sum_{i=1}^{m} y^{(i)} \log\left(h_{\omega}\left(x^{(i)}\right)\right) + \left(1 - y^{(i)}\right) \log\left(1 - h_{\omega}\left(x^{(i)}\right)\right)$$

Training Neural Networks: Gradient Descent

- Negative gradients points directly downhill of the cost function
- We can decrease the cost by moving in the direction of the negative gradient (α is a learning rate)

$$\omega \Leftarrow \omega - \alpha \nabla_{\omega} \ell \left(h_{\omega} \left(x^{(i)} \right), y^{(i)} \right)$$

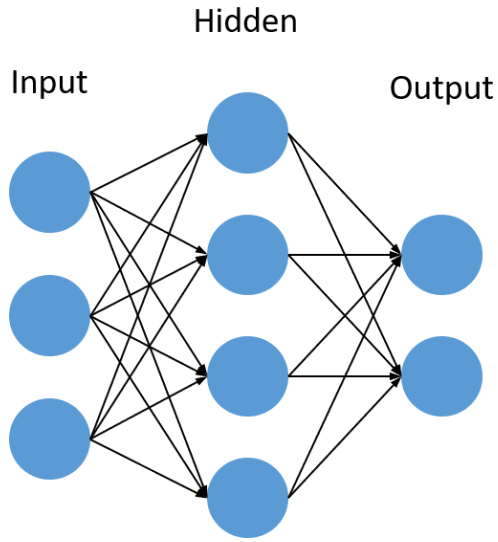


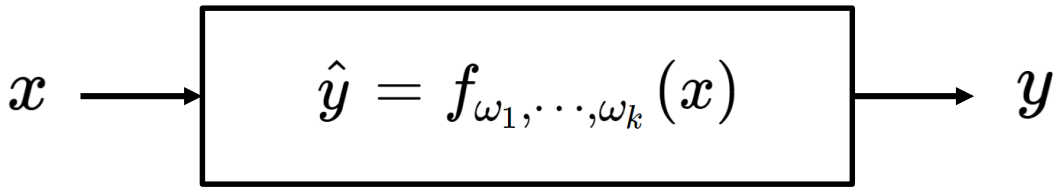


Gradients in ANN

- Learning weights and biases from data using gradient descent
- $\frac{\partial \ell}{\partial \omega}$: too many computations are required for all ω

- Structural constraint of NN:
  - Composition of functions
  - Chain rule
  - Dynamic programming

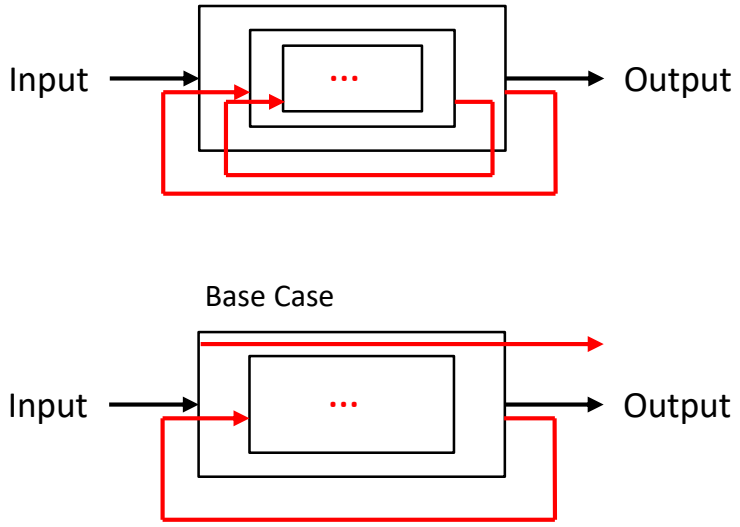




Dynamic Programming

Recursive Algorithm

- One of the central ideas of computer science
- Depends on solutions to smaller instances of the same problem (= sub-problem)
- Function to call itself (it is impossible in the real world)
- Factorial example
  - $n! = n \cdot (n-1) \cdots 2 \cdot 1$



Dynamic Programming

- Dynamic Programming: general, powerful algorithm design technique

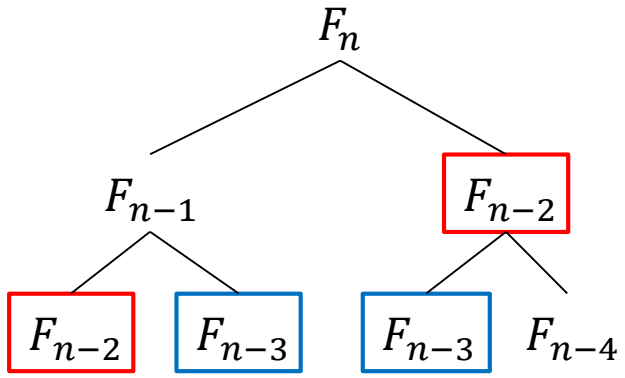
- Fibonacci numbers:

$$F_1 = F_2 = 1$$
$$F_n = F_{n-1} + F_{n-2}$$

Naïve Recursive Algorithm

```
fib(n) :
        if n ≤ 2 :  f = 1
        else :  f = fib(n − 1) + fib(n − 2)
        return f
```

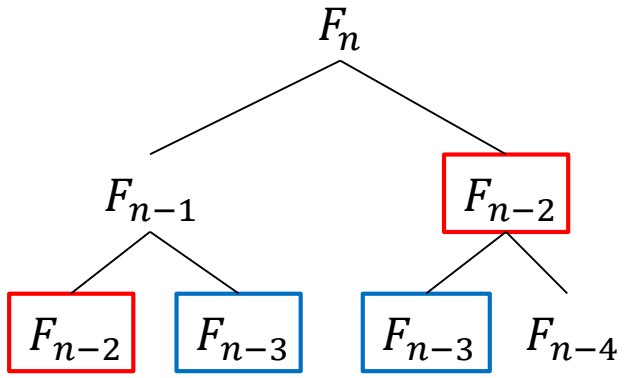
- It works. Is it good?



Memorized Recursive Algorithm

```
memo = [ ]
    fib(n) :
        if n in memo : return memo[n]

        if n ≤ 2 : f = 1
        else : f = fib(n − 1) + fib(n − 2)

        memo[n] = f
        return f
```

- Benefit?
  - fib(n) only recurses the first time it's called



Dynamic Programming Algorithm

- Memorize (remember) & re-use solutions to subproblems that helps solve the problem

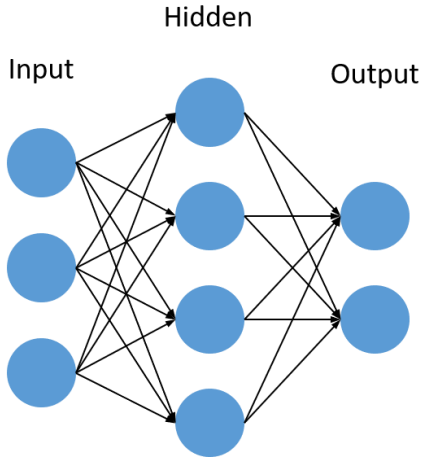
- DP ≈ recursion + memorization

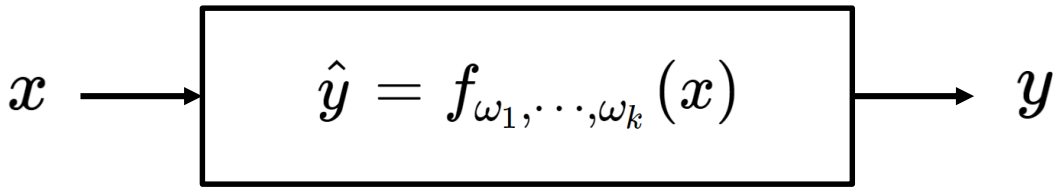
Backpropagation

Gradients in ANN

- Learning weights and biases from data using gradient descent
- $\frac{\partial \ell}{\partial \omega}$: too many computations are required for all ω

- Structural constraint of NN:
  - Composition of functions
  - Chain rule
  - Dynamic programming





Training Neural Networks: Backpropagation Learning

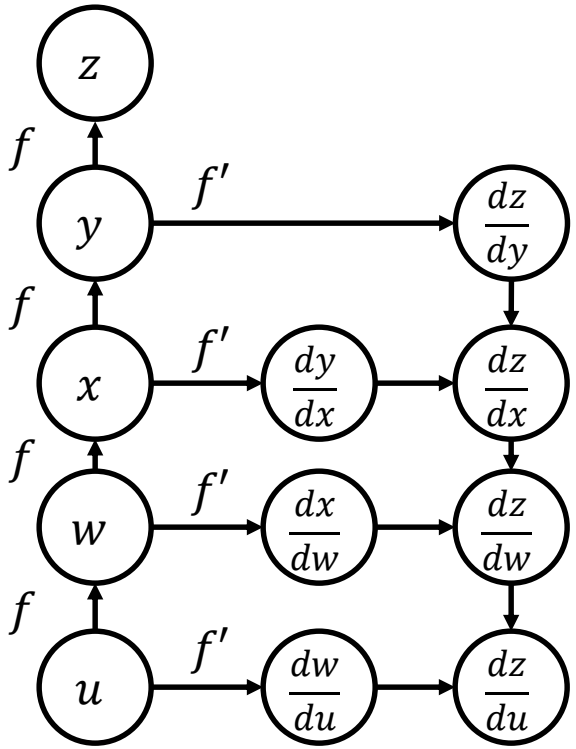
- Forward propagation
  - the initial information propagates up to the hidden units at each layer and finally produces output

- Backpropagation
  - allows the information from the cost to flow backwards through the network in order to compute the gradients

Backpropagation

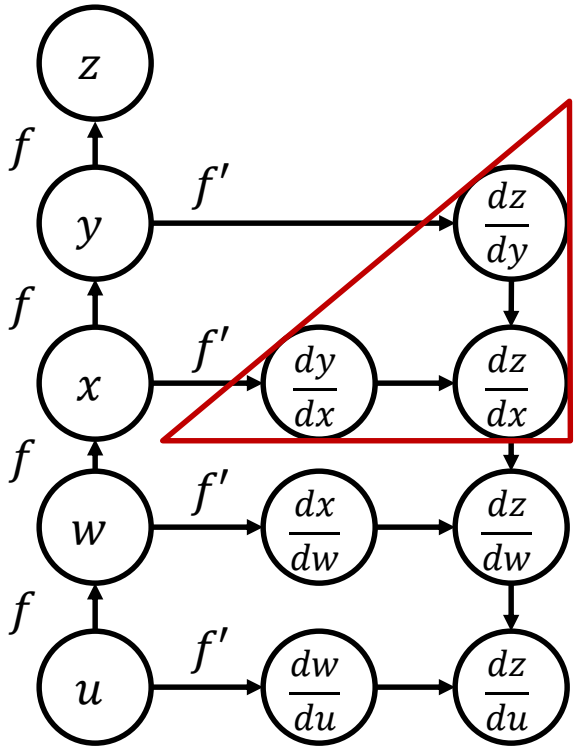
- Chain Rule
  - Computing the derivative of the composition of functions
    - $f\big(g(x)\big)' = f'\big(g(x)\big)g'(x)$
    - $\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$
    - $\frac{dz}{dw} = \left(\frac{dz}{dy} \cdot \frac{dy}{dx}\right) \cdot \frac{dx}{dw}$
    - $\frac{dz}{du} = \left(\frac{dz}{dy} \cdot \frac{dy}{dx} \cdot \frac{dx}{dw}\right) \cdot \frac{dw}{du}$

- Backpropagation
  - Update weights recursively



Backpropagation

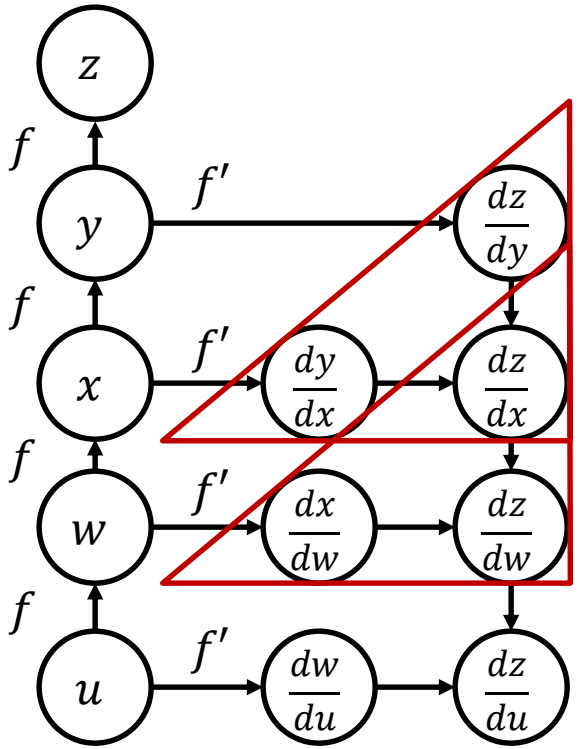
- Chain Rule
  - Computing the derivative of the composition of functions
    - $f\big(g(x)\big)' = f'\big(g(x)\big)g'(x)$
    - $\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$
    - $\frac{dz}{dw} = \left(\frac{dz}{dy} \cdot \frac{dy}{dx}\right) \cdot \frac{dx}{dw}$
    - $\frac{dz}{du} = \left(\frac{dz}{dy} \cdot \frac{dy}{dx} \cdot \frac{dx}{dw}\right) \cdot \frac{dw}{du}$
- Backpropagation
  - Update weights recursively



Backpropagation

- Chain Rule
  - Computing the derivative of the composition of functions
    - $f\big(g(x)\big)' = f'\big(g(x)\big)g'(x)$
    - $\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$
    - $\frac{dz}{dw} = \left(\frac{dz}{dy} \cdot \frac{dy}{dx}\right) \cdot \frac{dx}{dw}$
    - $\frac{dz}{du} = \left(\frac{dz}{dy} \cdot \frac{dy}{dx} \cdot \frac{dx}{dw}\right) \cdot \frac{dw}{du}$

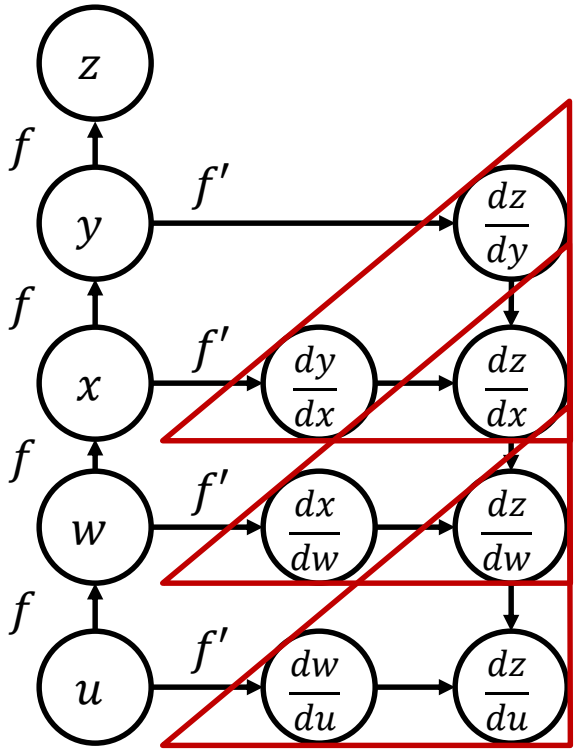
- Backpropagation
  - Update weights recursively



Backpropagation

- Chain Rule
  - Computing the derivative of the composition of functions
    - $f\big(g(x)\big)' = f'\big(g(x)\big)g'(x)$
    - $\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$
    - $\frac{dz}{dw} = \left(\frac{dz}{dy} \cdot \frac{dy}{dx}\right) \cdot \frac{dx}{dw}$
    - $\frac{dz}{du} = \left(\frac{dz}{dy} \cdot \frac{dy}{dx} \cdot \frac{dx}{dw}\right) \cdot \frac{dw}{du}$

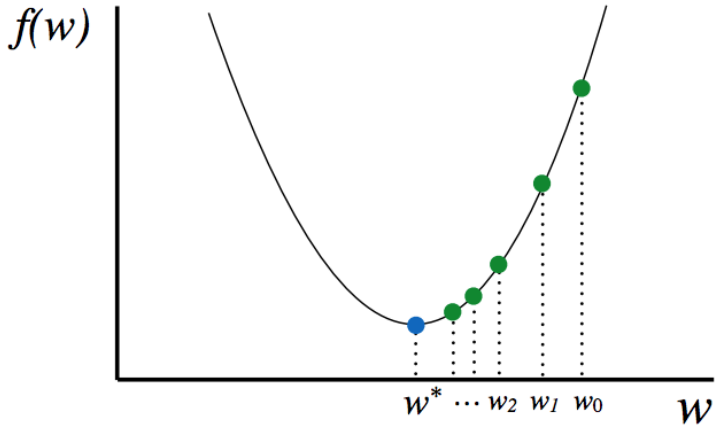
- Backpropagation
  - Update weights recursively with memory



Training Neural Networks with TensorFlow

- Optimization procedure

Start at a random point
Repeat
    Determine a descent direction
    Choose a step size
    Update
Until stopping criterion is satisfied

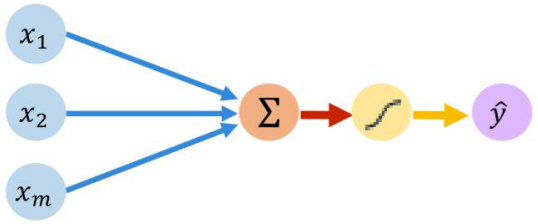


- It is not easy to numerically compute gradients in network in general.
  - The good news: people have already done all the "hard work" of developing numerical solvers (or libraries)
  - There are a wide range of tools → We will use the TensorFlow

Core Foundation Review

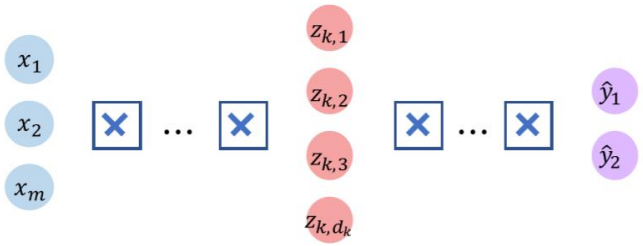
The Perceptron

- Structural building blocks
- Nonlinear activation functions



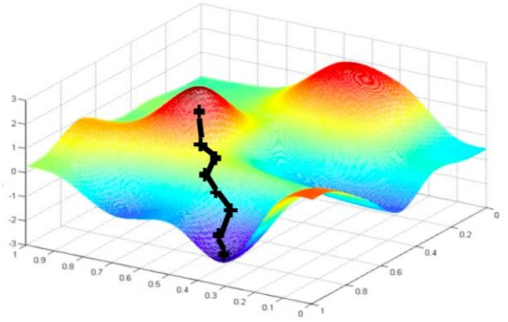
Neural Networks

- Stacking Perceptrons to form neural networks
- Optimization through backpropagation



Training in Practice

- Adaptive learning
- Batching
- Regularization



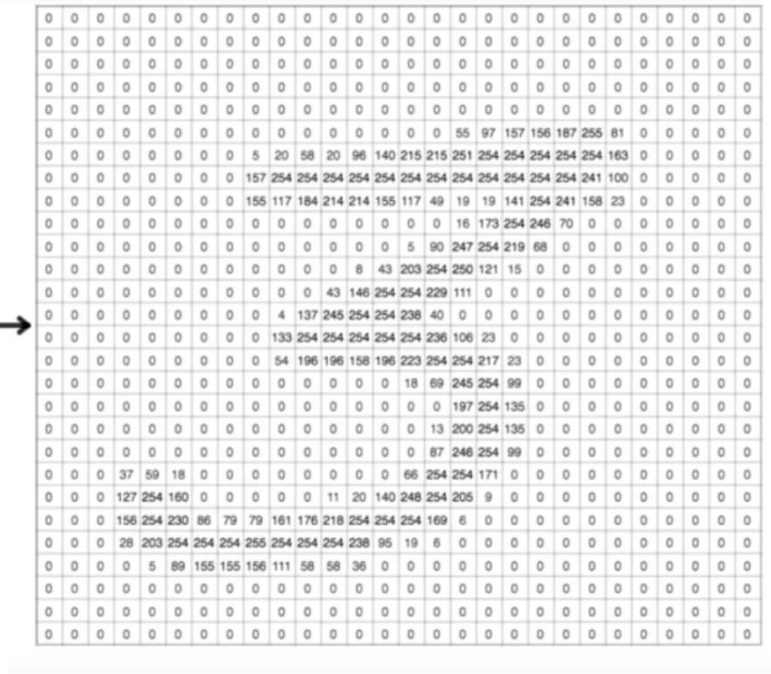
Source: 6.S191 Intro. to Deep Learning at MIT

MNIST

MNIST database

- Mixed National Institute of Standards and Technology database
- Handwritten digit database
- 28×28 gray scaled image
- Flattened matrix into a vector of $28 \times 28 = 784$

28 x 28
= 784 pixels

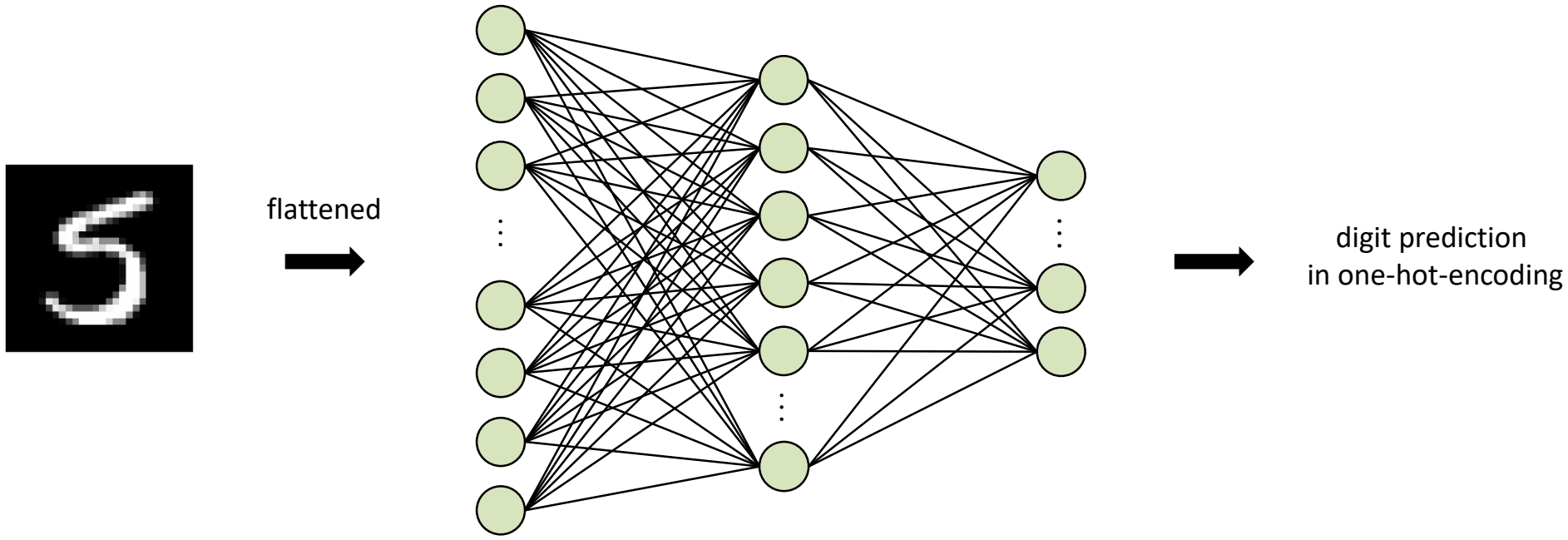


pixel 1
pixel 2
pixel 3
pixel 4
pixel 5
pixel 6
pixel 7
pixel 8
pixel 9
pixel 10
pixel 11
pixel 12
pixel 13
pixel 14
pixel 15
pixel 16
pixel 17
pixel 18
pixel 19

pixel 784

ANN in TensorFlow: MNIST

Our Network Model

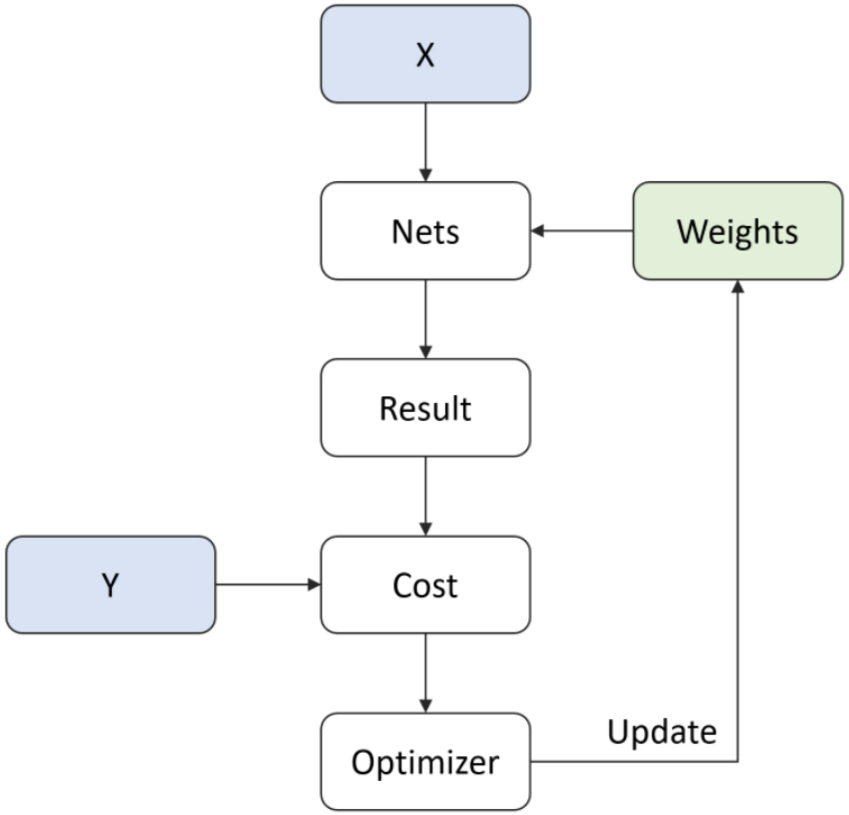


Iterative Optimization

- We will use
  - Mini-batch gradient descent
  - Adam optimizer

$$\begin{aligned} \min_{\theta} \quad & f(\theta) \\ \text{subject to} \quad & g_i(\theta) \leq 0 \end{aligned}$$

$$\theta := \theta - \alpha \nabla_\theta \left(h_\theta \left(x^{(i)} \right), y^{(i)} \right)$$



ANN with TensorFlow

- Import Library

```
# Import Library
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
```

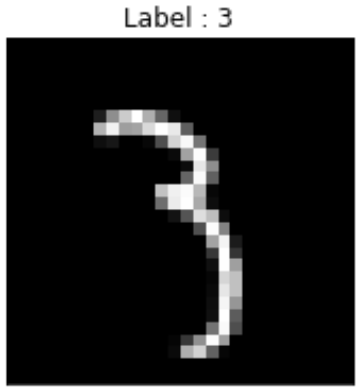
- Load MNIST Data
  - Download MNIST data from TensorFlow tutorial example

```
from tensorflow.examples.tutorials.mnist import input_data

mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
```

One Hot Encoding

- Batch maker

```
train_x, train_y = mnist.train.next_batch(1)
img = train_x[0,:].reshape(28,28)

plt.figure(figsize=(5,3))
plt.imshow(img,'gray')
plt.title("Label : {}".format(np.argmax(train_y[0,:])))
plt.xticks([])
plt.yticks([])
plt.show()
```



- One hot encoding

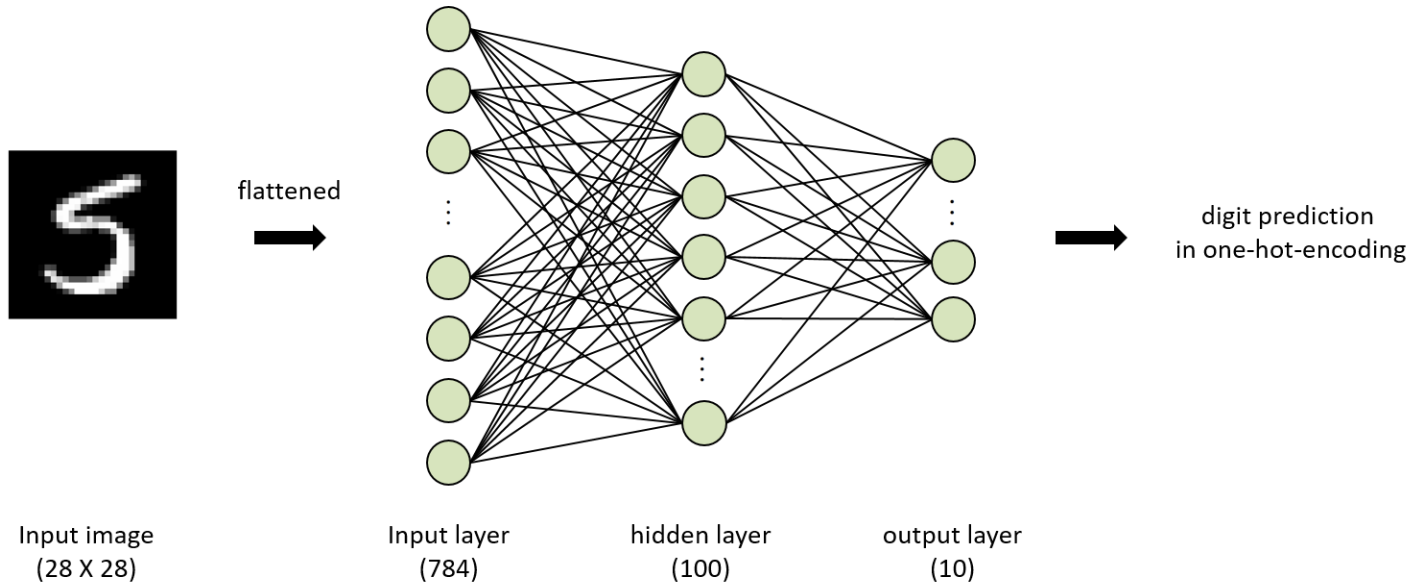
```
print ('Train labels : {}'.format(train_y[0, :]))
```

```
Train labels : [0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
```

ANN Structure

- Input size
- Hidden layer size
- The number of classes

```
n_input = 28*28
n_hidden = 100
n_output = 10
```



Weights & Biases and Placeholder

- Define parameters based on predefined layer size
- Initialize with normal distribution with $\mu = 0$ and $\sigma = 0.1$

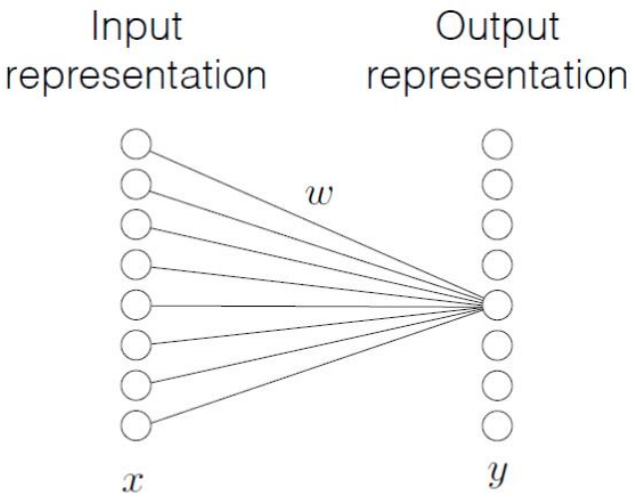
```
weights = {
    'hidden' : tf.Variable(tf.random_normal([n_input, n_hidden], stddev = 0.1)),
    'output' : tf.Variable(tf.random_normal([n_hidden, n_output], stddev = 0.1))
}

biases = {
    'hidden' : tf.Variable(tf.random_normal([n_hidden], stddev = 0.1)),
    'output' : tf.Variable(tf.random_normal([n_output], stddev = 0.1))
}
```

- Placeholder

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

Build a Model

- First, the layer performs several matrix multiplication to produce a set of linear activations



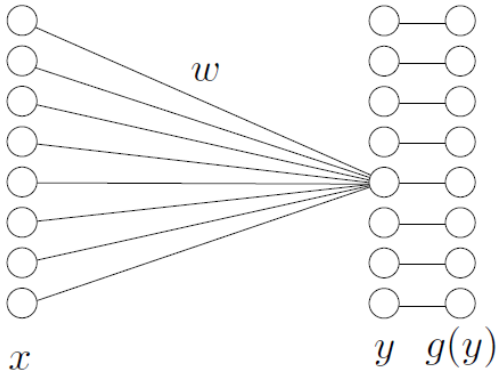
$$y_j = \left(\sum_i \omega_{ij} x_i \right) + b_j$$

$$y = \omega^T x + b$$

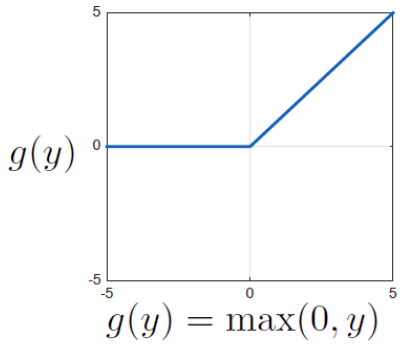
```python
# Define Network
def build_model(x, weights, biases):

    # first hidden layer
    hidden = tf.add(tf.matmul(x, weights['hidden']), biases['hidden'])
    # non-linear activate function
    hidden = tf.nn.relu(hidden)

    # Output layer
    output = tf.add(tf.matmul(hidden, weights['output']), biases['output'])

    return output
```

Build a Model

- Second, each linear activation is running through a nonlinear activation function



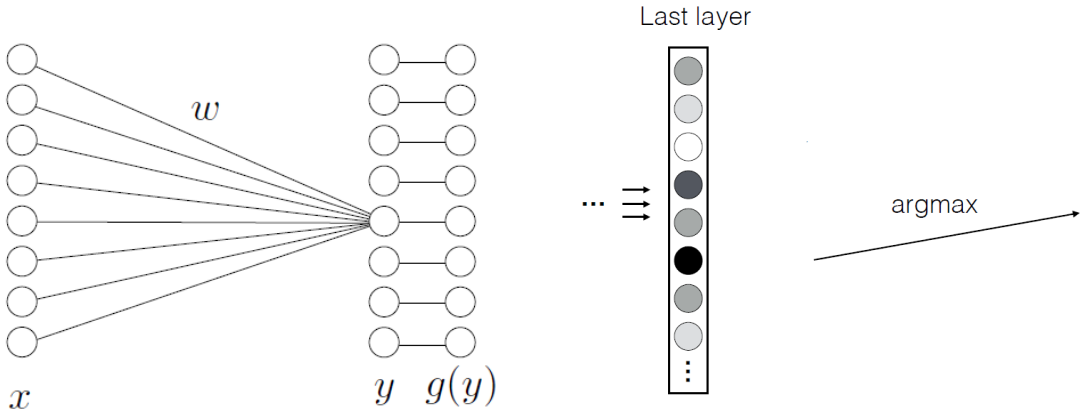
Rectified linear unit (ReLU)




```python
# Define Network
def build_model(x, weights, biases):

    # first hidden layer
    hidden = tf.add(tf.matmul(x, weights['hidden']), biases['hidden'])
    # non-linear activate function
    hidden = tf.nn.relu(hidden)

    # Output layer
    output = tf.add(tf.matmul(hidden, weights['output']), biases['output'])

    return output
```

Build a Model

- Third, predict values with an affine transformation




```python
# Define Network
def build_model(x, weights, biases):

    # first hidden layer
    hidden = tf.add(tf.matmul(x, weights['hidden']), biases['hidden'])
    # non-linear activate function
    hidden = tf.nn.relu(hidden)

    # Output layer
    output = tf.add(tf.matmul(hidden, weights['output']), biases['output'])

    return output
```

Loss and Optimizer

- Loss: softmax cross entropy

$$-\frac{1}{N}\sum_{i=1}^{N} y^{(i)} \log(h_\theta\left(x^{(i)}\right)) + (1-y^{(i)}) \log(1-h_\theta\left(x^{(i)}\right))$$

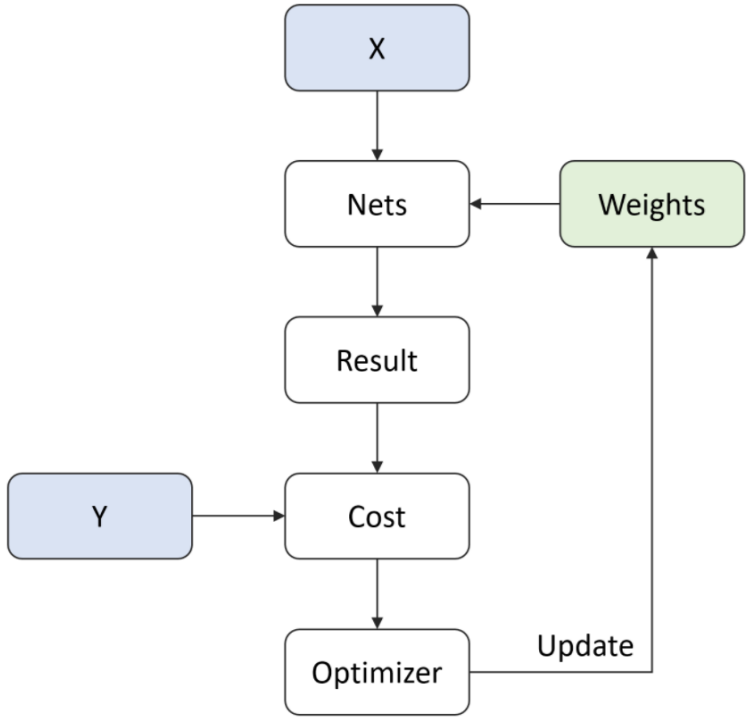
- Optimizer
  - AdamOptimizer: the most popular optimizer

```
# Define Loss
pred = build_model(x, weights, biases)
loss = tf.nn.softmax_cross_entropy_with_logits(logits = pred, labels = y)
loss = tf.reduce_mean(loss)

LR = 0.0001
optm = tf.train.AdamOptimizer(LR).minimize(loss)
```

Iteration Configuration

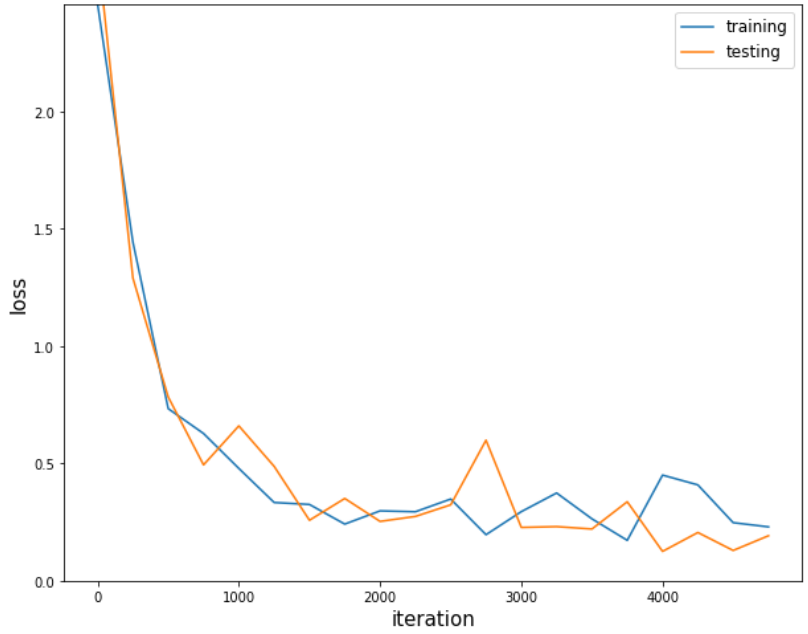
- Define parameters for training ANN
  - n_batch: batch size for mini-batch gradient descent
  - n_iter: the number of iteration steps
  - n_prt: check loss for every n_prt iteration

```
n_batch = 50      # Batch Size
n_iter = 5000     # Learning Iteration
n_prt = 250       # Print Cycle
```



Optimization

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

loss_record_training = []
loss_record_testing = []
for epoch in range(n_iter):
    train_x, train_y = mnist.train.next_batch(n_batch)
    sess.run(optm, feed_dict = {x: train_x, y: train_y})

    if epoch % n_prt == 0:
        test_x, test_y = mnist.test.next_batch(n_batch)
        c1 = sess.run(loss, feed_dict = {x: train_x, y: train_y})
        c2 = sess.run(loss, feed_dict = {x: test_x, y: test_y})
        loss_record_training.append(c1)
        loss_record_testing.append(c2)
        print ("Iter : {}".format(epoch))
        print ("Cost : {}".format(c1))

plt.figure(figsize=(10,8))
plt.plot(np.arange(len(loss_record_training))*n_prt,
         loss_record_training, label = 'training')
plt.plot(np.arange(len(loss_record_testing))*n_prt,
         loss_record_testing, label = 'testing')
plt.xlabel('iteration', fontsize = 15)
plt.ylabel('loss', fontsize = 15)
plt.legend(fontsize = 12)
plt.ylim([0,np.max(loss_record_training)])
plt.show()
```

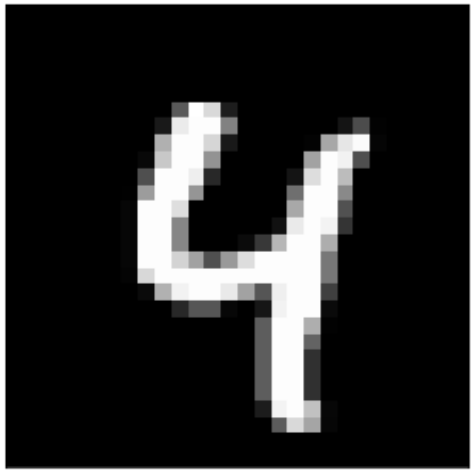


Test or Evaluation

```
test_x, test_y = mnist.test.next_batch(100)

my_pred = sess.run(pred, feed_dict = {x : test_x})
my_pred = np.argmax(my_pred, axis = 1)

labels = np.argmax(test_y, axis = 1)

accr = np.mean(np.equal(my_pred, labels))
print("Accuracy : {}%".format(accr*100))
```

```
Accuracy : 96.0%
```

```
test_x, test_y = mnist.test.next_batch(1)
logits = sess.run(tf.nn.softmax(pred), feed_dict = {x : test_x})
predict = np.argmax(logits)

plt.imshow(test_x.reshape(28,28), 'gray')
plt.xticks([])
plt.yticks([])
plt.show()

print('Prediction : {}'.format(predict))
np.set_printoptions(precision=2, suppress=True)
print('Probability : {}'.format(logits.ravel()))
```



```
Prediction : 4
Probability : [0.   0.   0.   0.   0.9  0.   0.   0.01 0.   0.09]
```

Autoencoder

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

Autoencoders

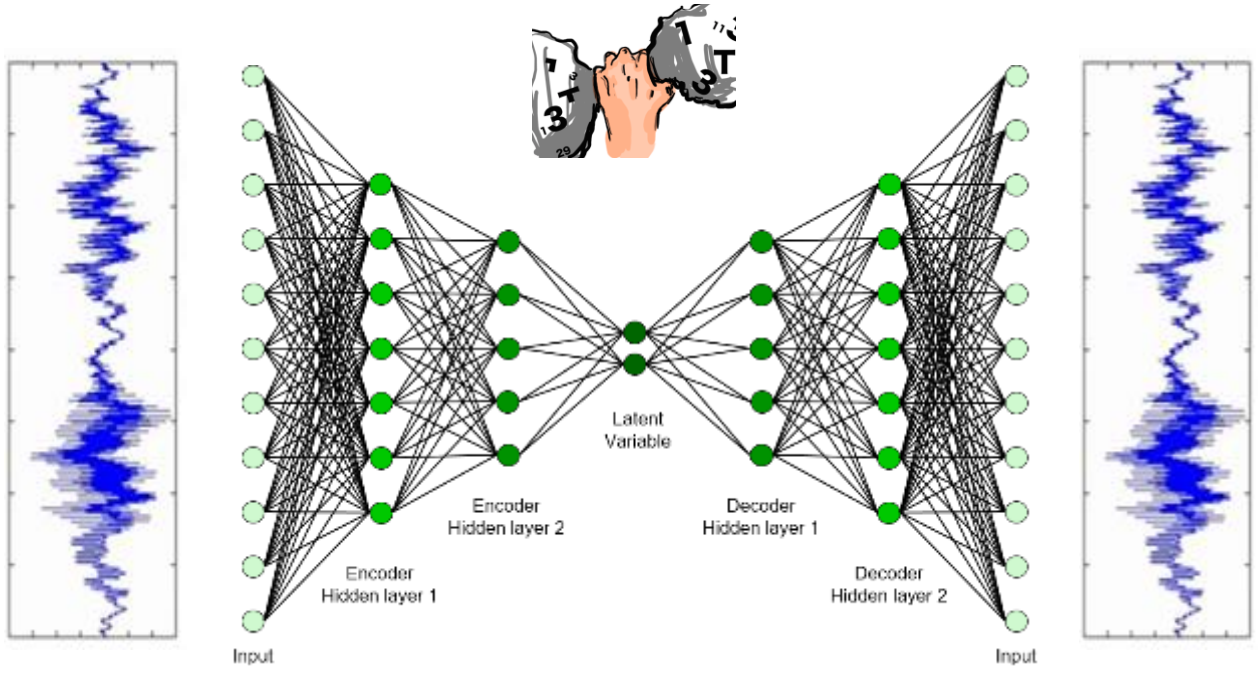
- It is like ‘deep learning version’ of unsupervised learning
- Definition
  - An autoencoder is a neural network that is trained to attempt to copy its input to its output
  - The network consists of two parts: an encoder and a decoder that produce a reconstruction
- Encoder and Decoder
  - Encoder 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

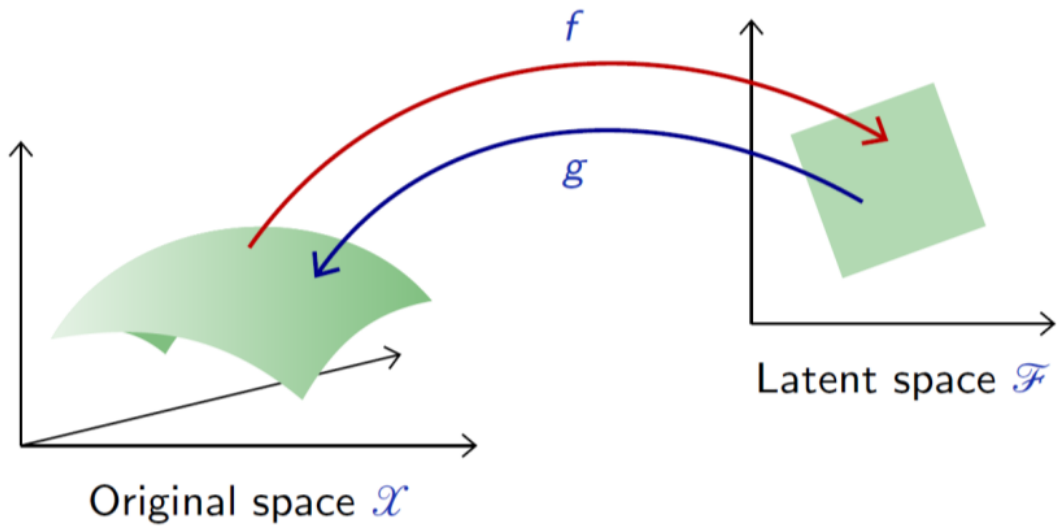
- Dimension reduction
- Recover the input data
  - Learns an encoding of the inputs so as to recover the original input from the encodings as well as possible



Original space Latent space

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



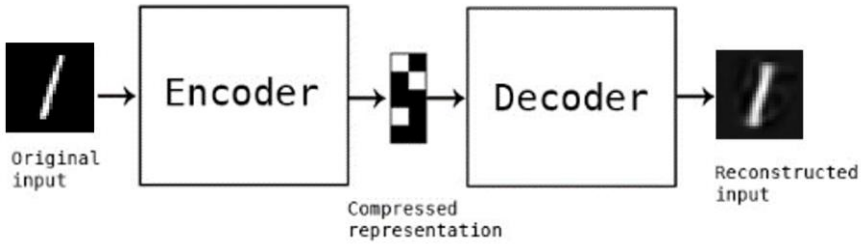
$$\mathbb{E}\left[\|X - g \circ f(X)\|^2\right] \approx 0$$

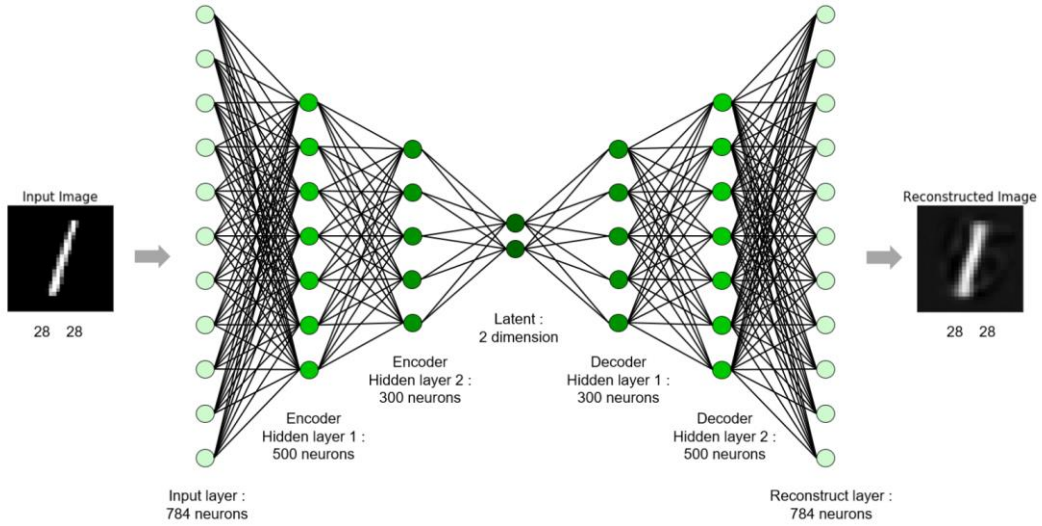
- A proper autoencoder has to capture a "good" parametrization of the signal, and in particular the statistical dependencies between the signal components.

Autoencoder with MNIST

Autoencoder with TensorFlow

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

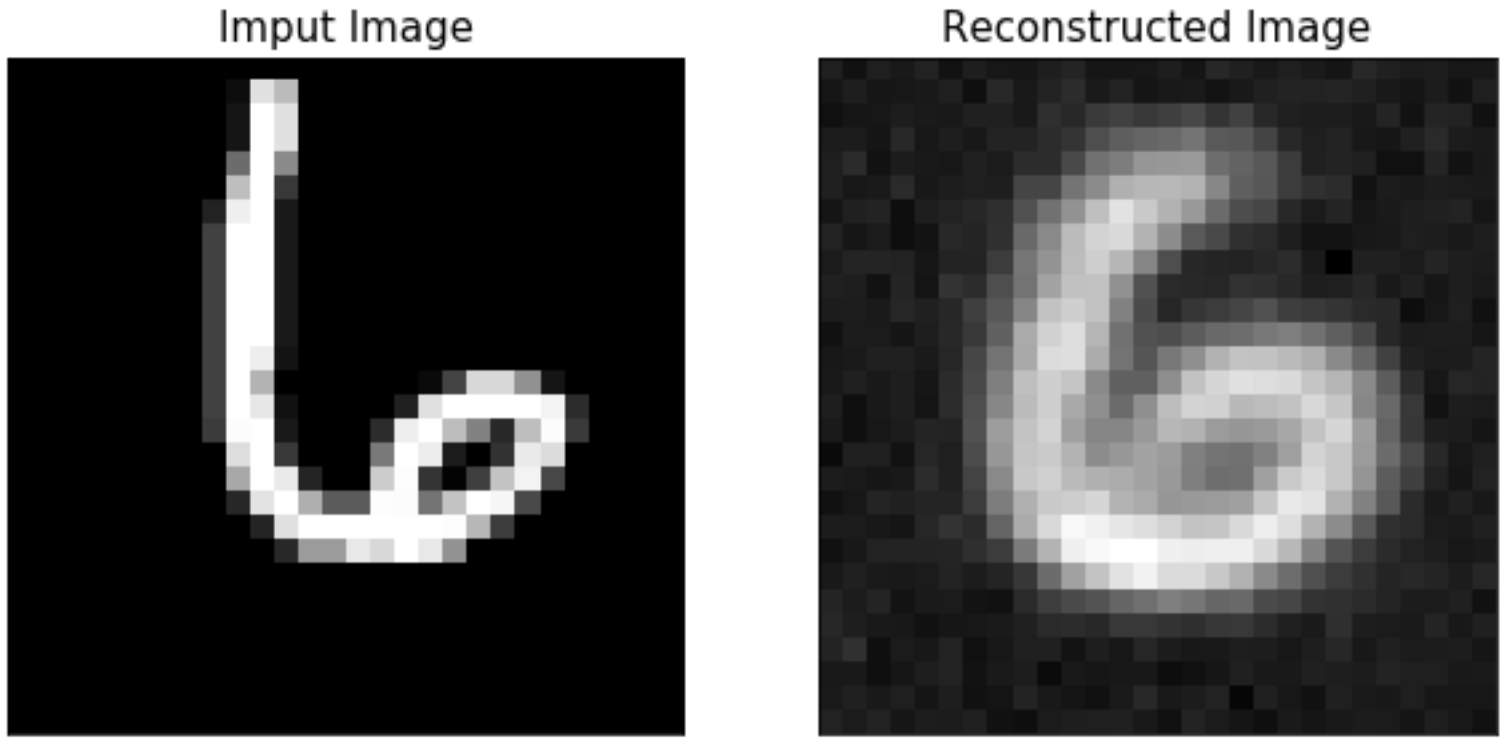




$$\frac{1}{m}\sum_{i=1}^{m}(t_i - y_i)^2$$

Test or Evaluation

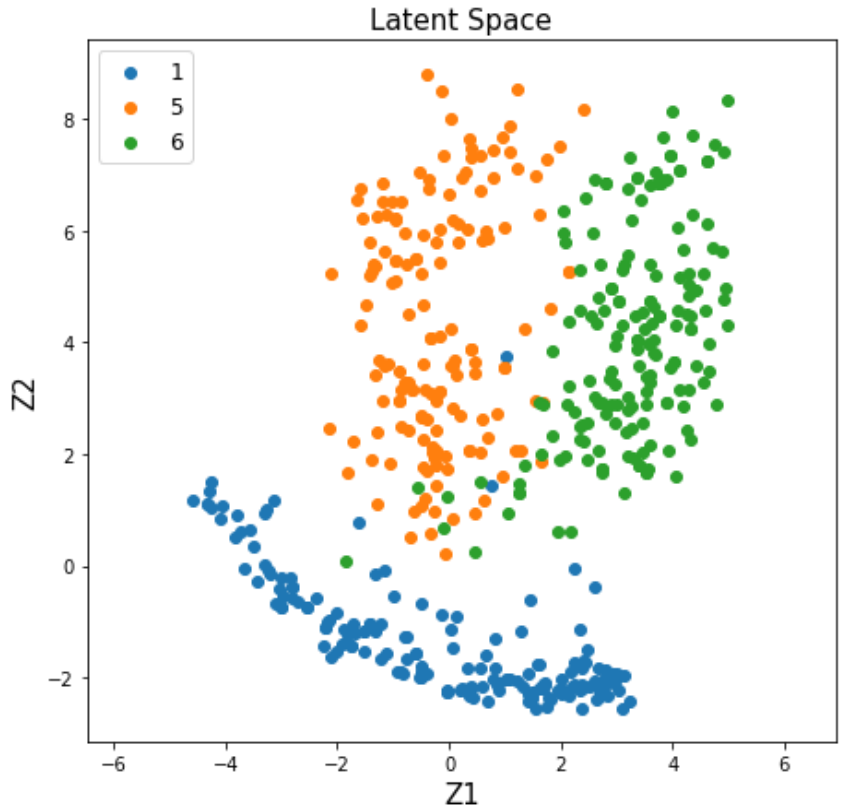
```
test_x, _ = test_batch_maker(1)
x_reconst = sess.run(reconst, feed_dict = {x: test_x})
```



Distribution in Latent Space

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

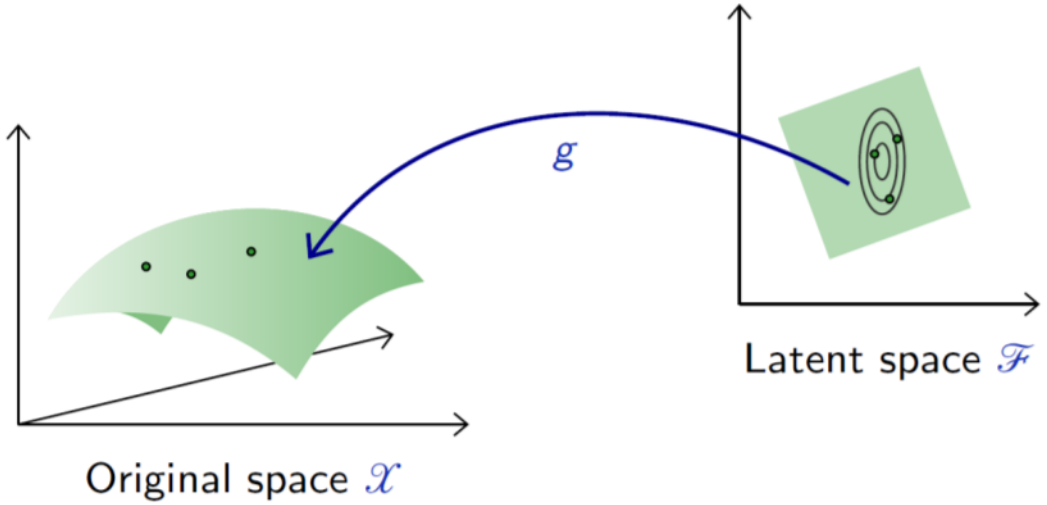
```
test_x, test_y = test_batch_maker(500)
test_y = np.argmax(test_y, axis = 1)
test_latent = sess.run(latent, feed_dict = {x: test_x})
```



Autoencoder as Generative Model

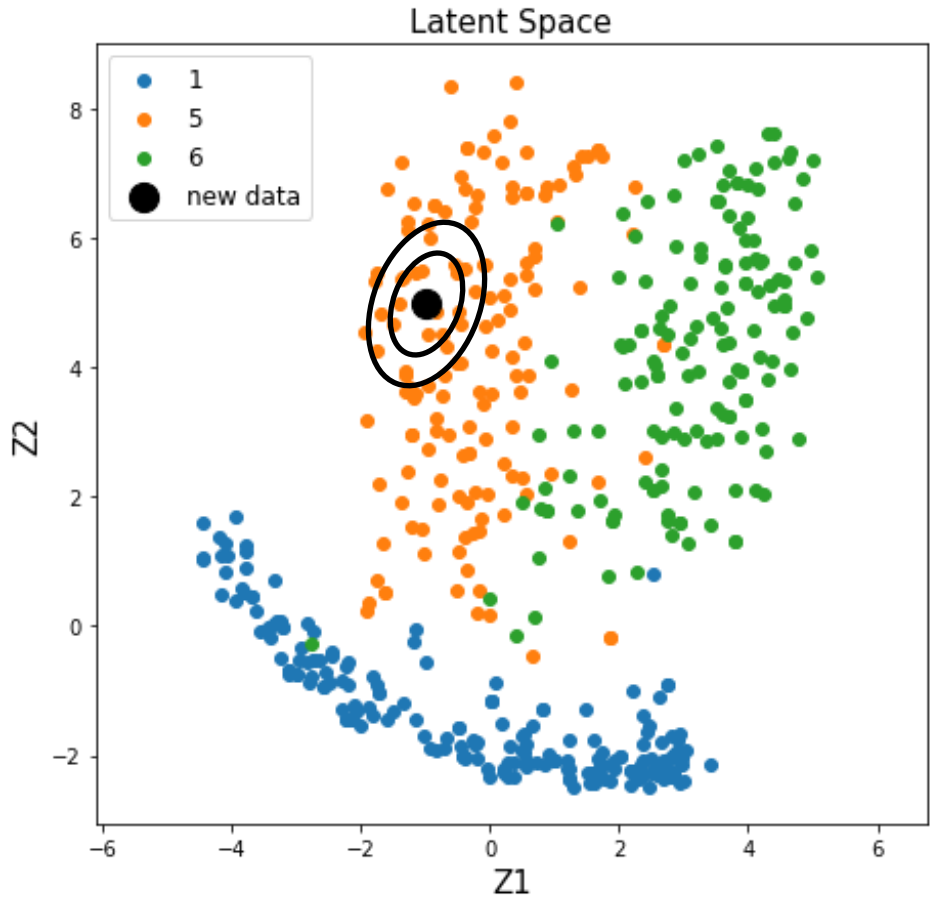
Generative Capabilities

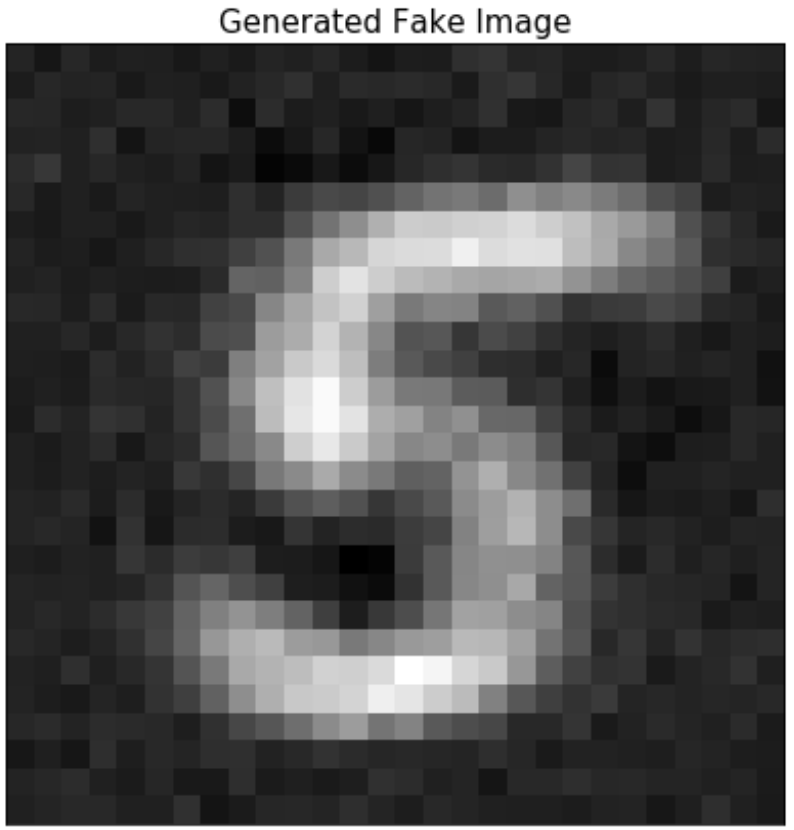
- We can assess the generative capabilities of the decoder g by introducing a [simple] density model q^z over the latent space $\mathcal{F}$, sample there, and map the samples into the image space $\mathcal{X}$ with g.



Source: Dr. Francois Fleuret at EPEL

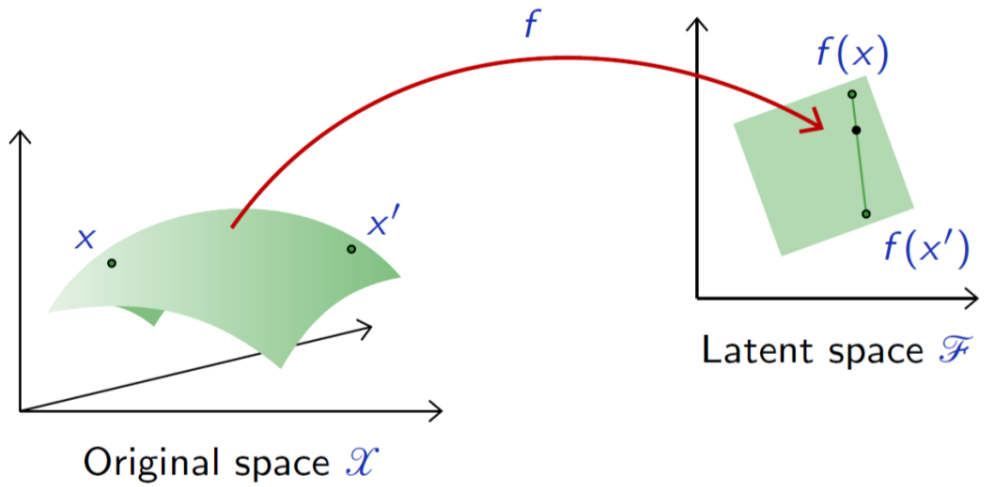
MNIST Example





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

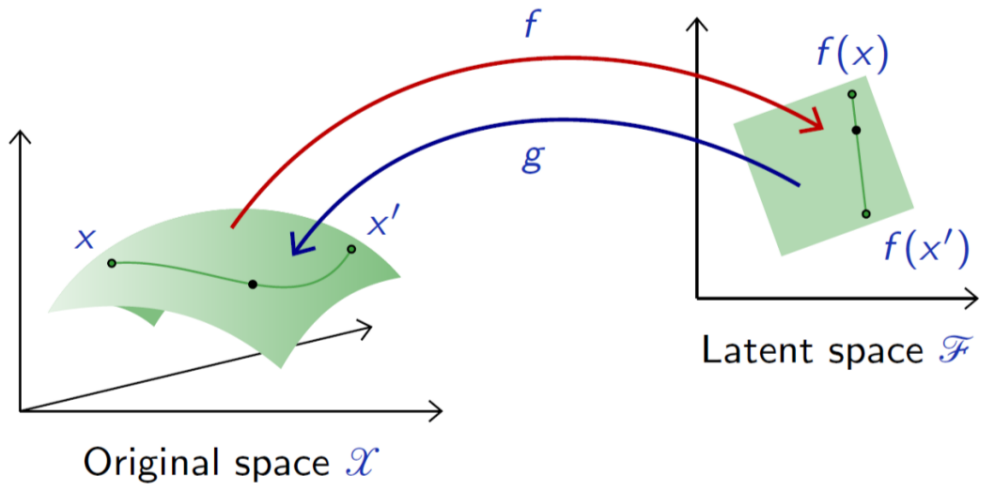


Source: Dr. Francois Fleuret at EPEL

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

$$g\left((1-\alpha)f(x)+\alpha f(x')\right)$$

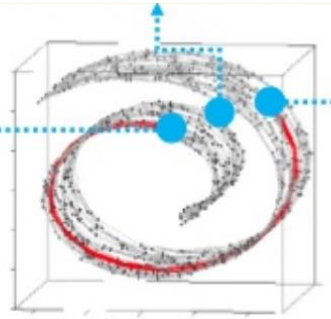


Source: Dr. Francois Fleuret at EPEL

Interpolation in High Dimension

Reasonable distance metric

Interpolation in high dimension

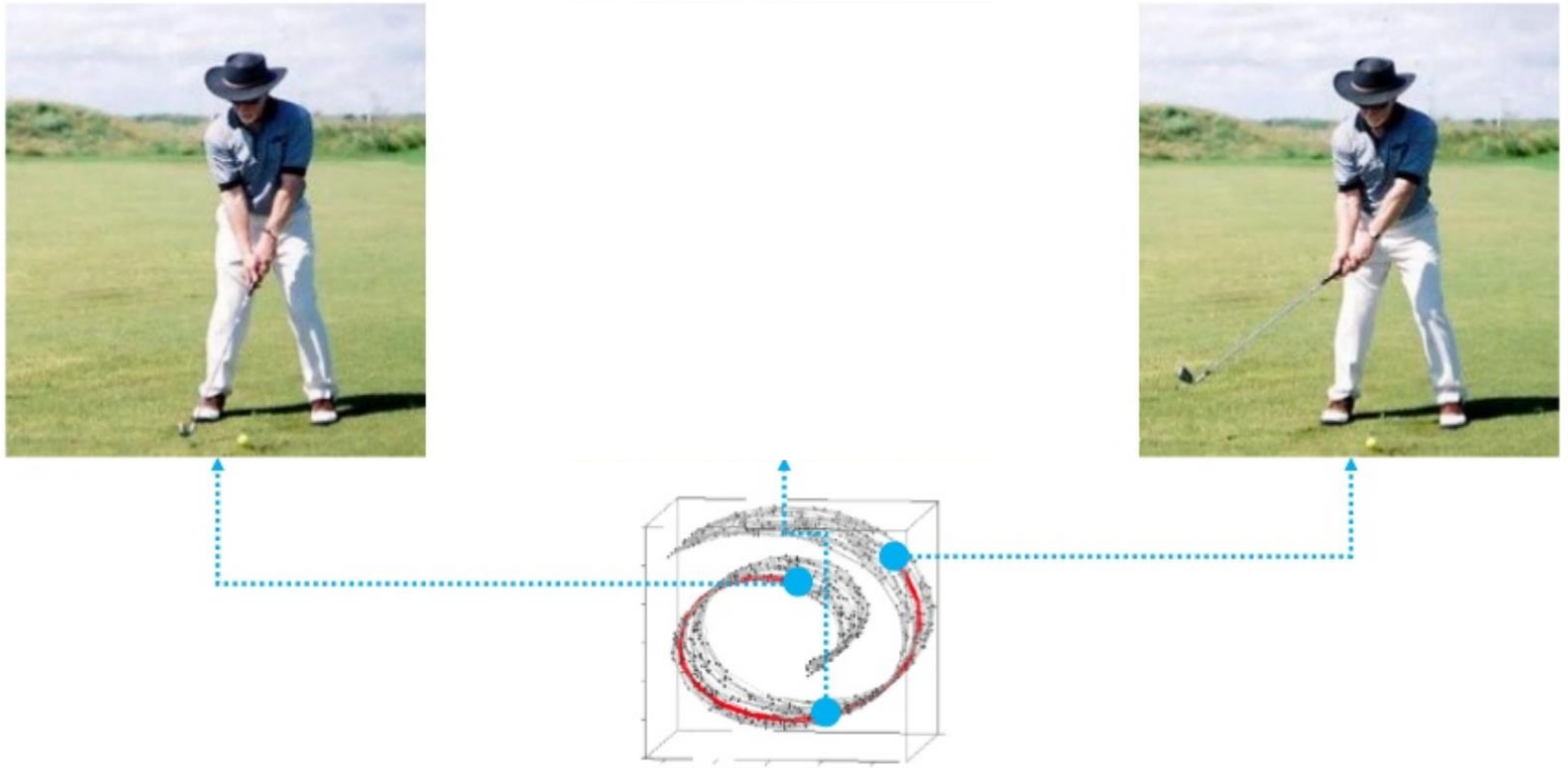


https://www.cs.cmu.edu/~efros/courses/AP06/presentations/ThompsonDimensionalityReduction.pdf

Interpolation in Manifold

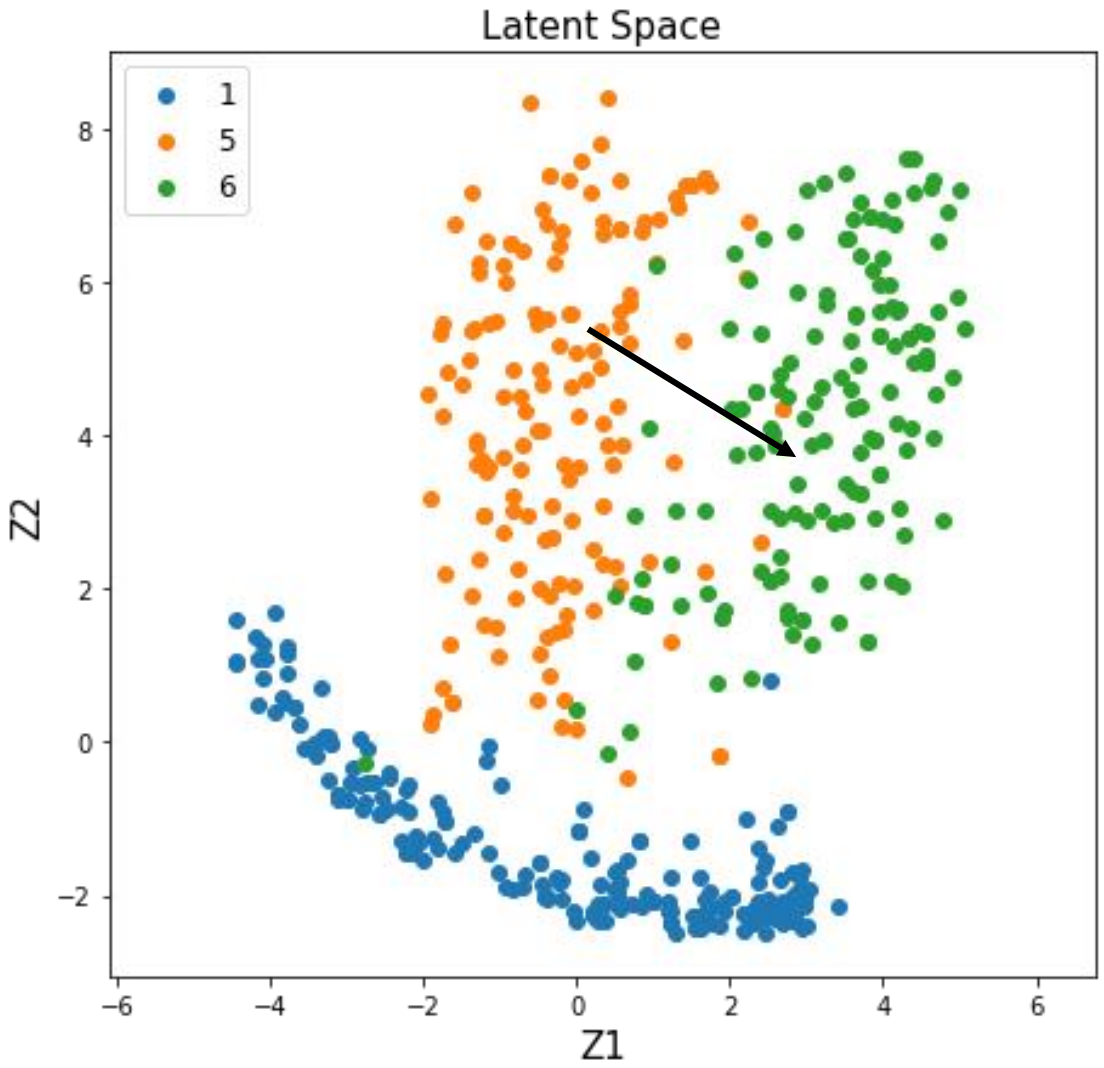
Reasonable distance metric

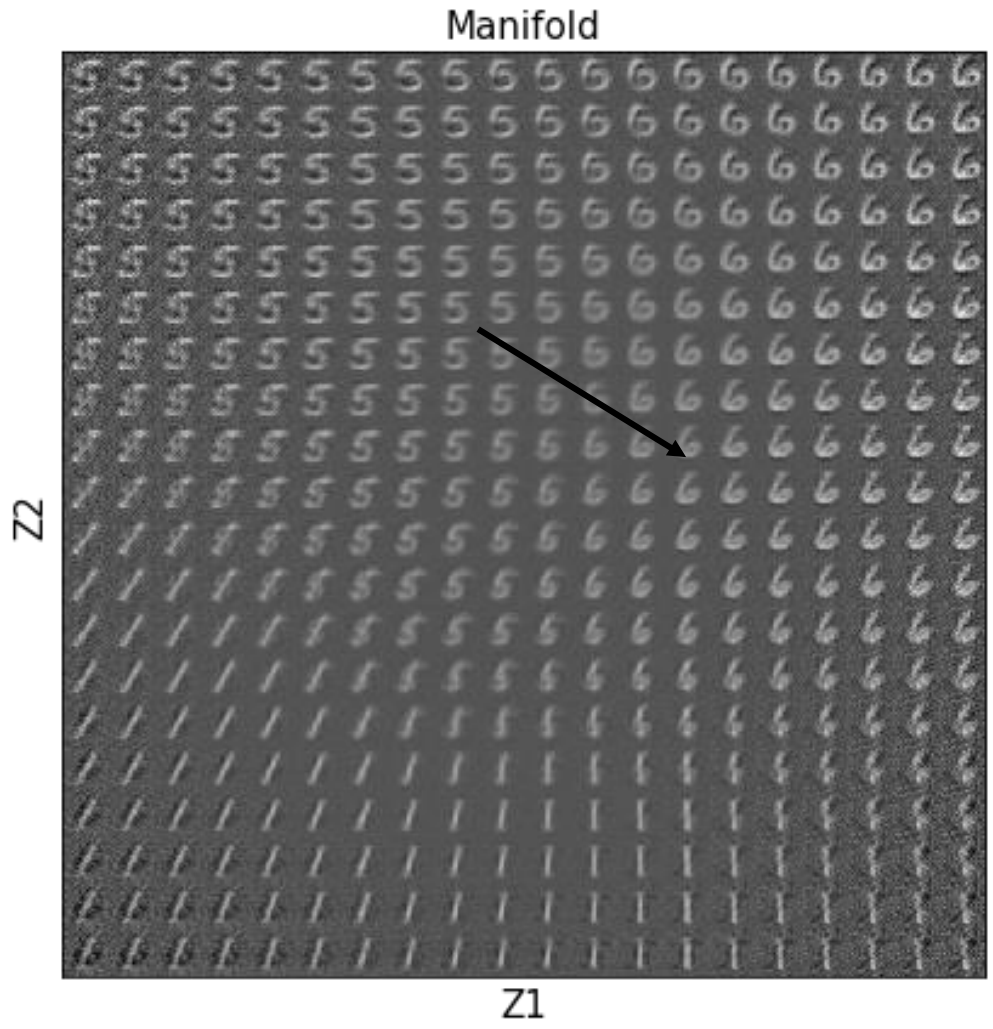
Interpolation in manifold



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

MNIST Example: Walk in the Latent Space





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.