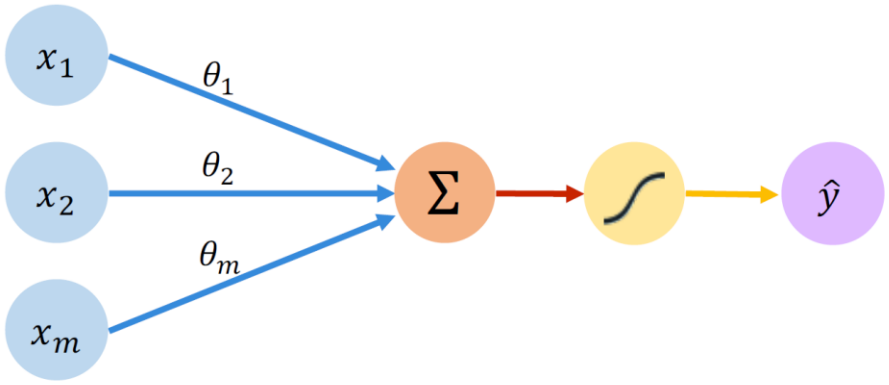
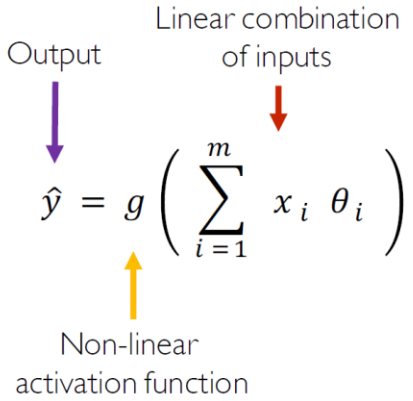
(Artificial) Neural Networks

Industrial AI Lab.

Prof. Seungchul Lee

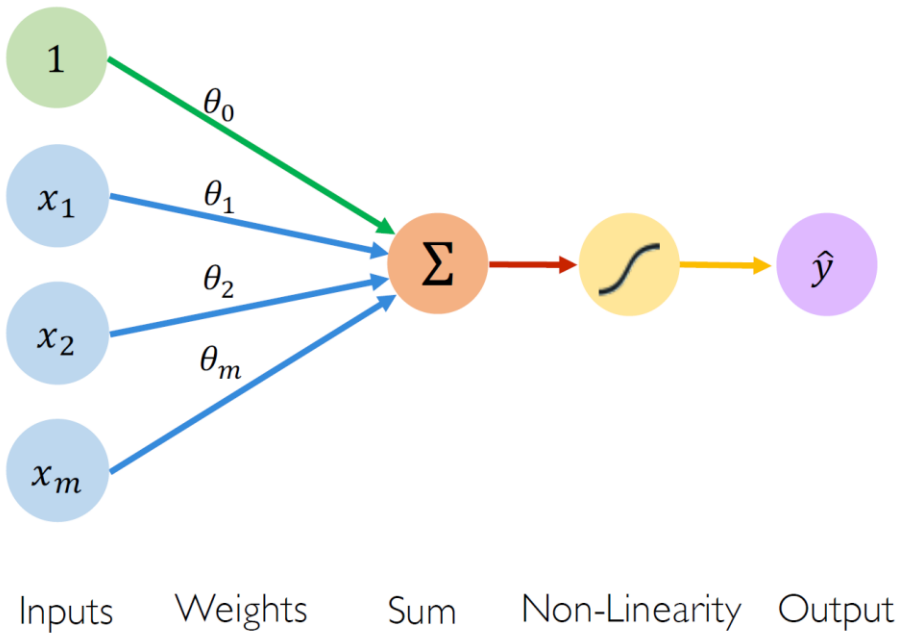
Perceptron: Forward Propagation

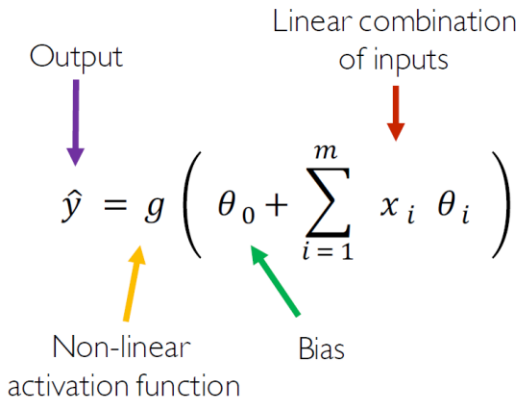




Source: 6.S191 Intro. to Deep Learning at MIT

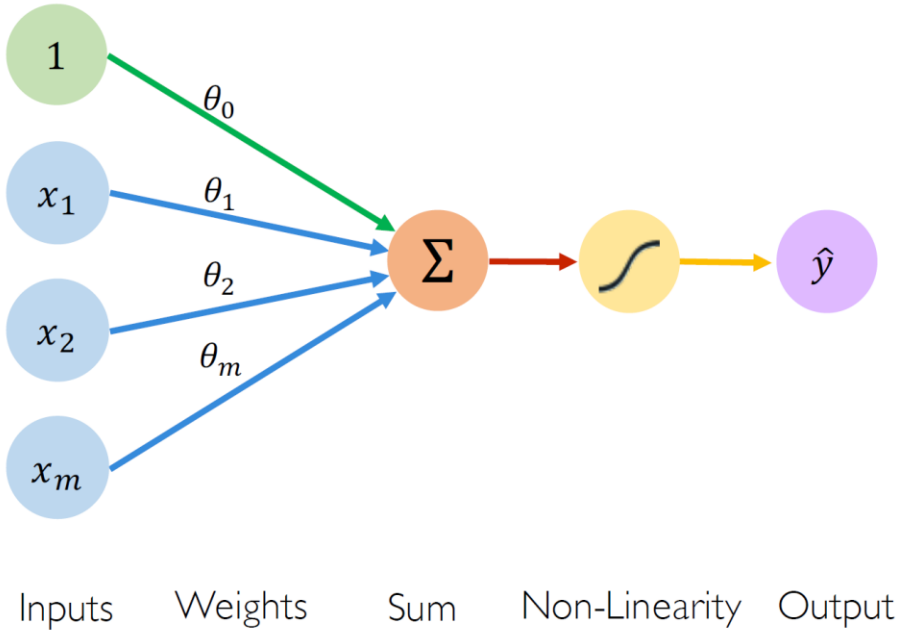
Perceptron: Forward Propagation





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

Perceptron: Forward Propagation



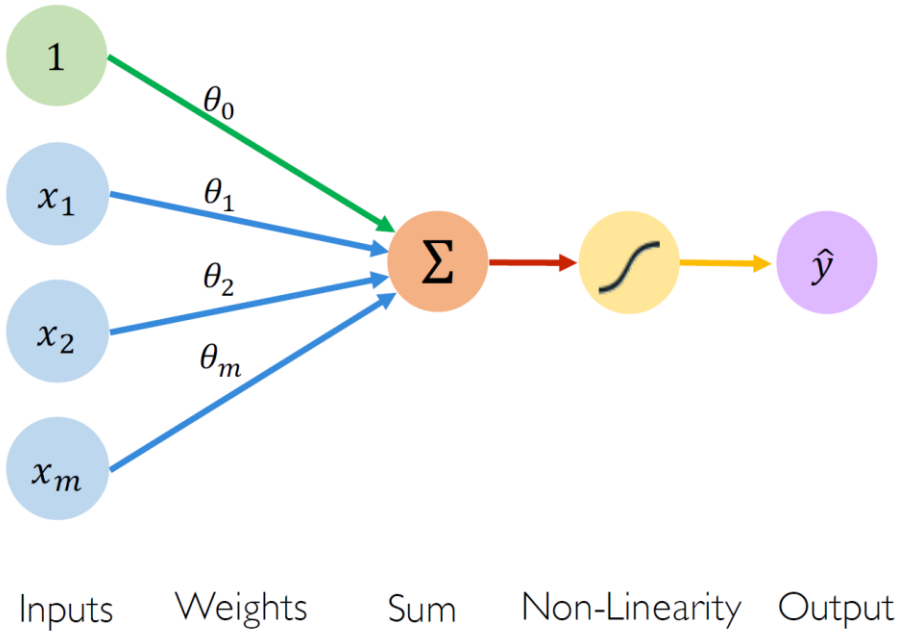
$$\hat{y} = g\left(\theta_0 + \sum_{i=1}^{m} x_i\, \theta_i\right)$$

$$\hat{y} = g\left(\theta_0 + \boldsymbol{X}^T \boldsymbol{\theta}\right)$$

where: $\boldsymbol{X} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$ and $\boldsymbol{\theta} = \begin{bmatrix} \theta_1 \\ \vdots \\ \theta_m \end{bmatrix}$

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

Perceptron: Forward Propagation

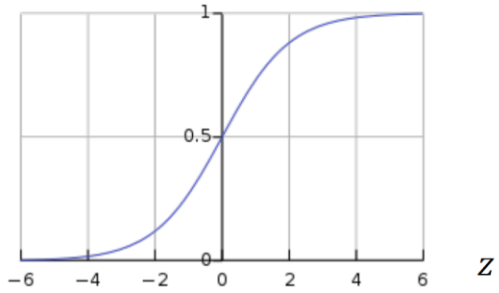


Activation Functions

$$\hat{y} = g(\theta_0 + \boldsymbol{X}^T \boldsymbol{\theta})$$

- Example: sigmoid function

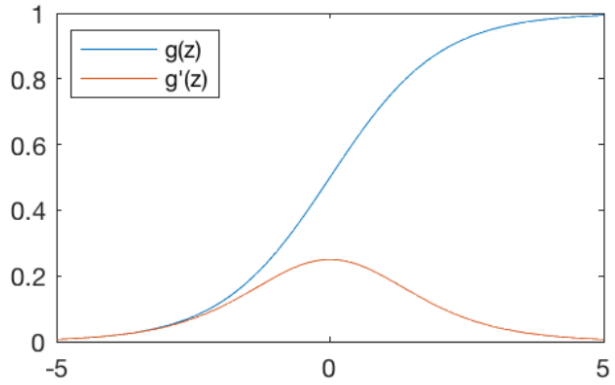
$$g(z) = \sigma(z) = \frac{1}{1 + e^{-z}}$$



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

Common Activation Functions

Sigmoid Function

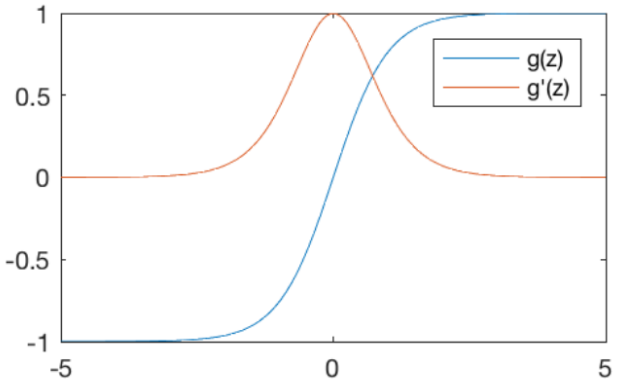


$$g(z) = \frac{1}{1 + e^{-z}}$$

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

`tf.nn.sigmoid(z)`

Hyperbolic Tangent

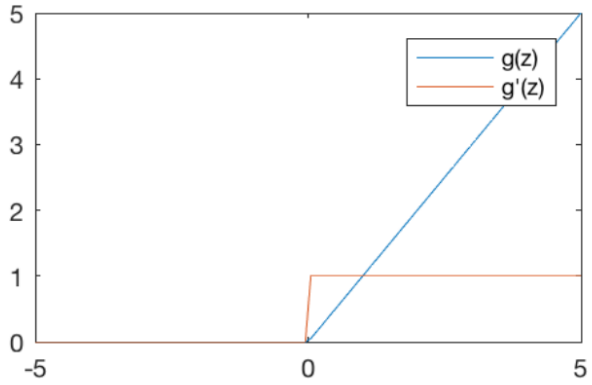


$$g(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

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

`tf.nn.tanh(z)`

Rectified Linear Unit (ReLU)



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

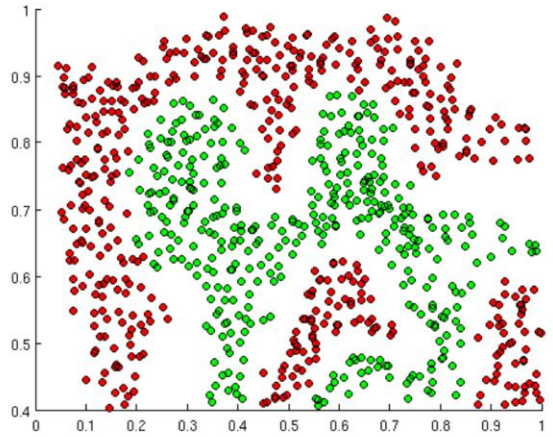
$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

`tf.nn.relu(z)`

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

Importance of Activation Functions

The purpose of activation functions is to ***introduce non-linearities*** *into the network*

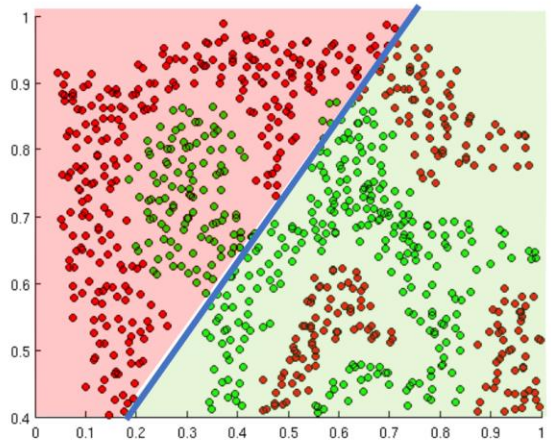


What if we wanted to build a Neural Network to distinguish green vs red points?

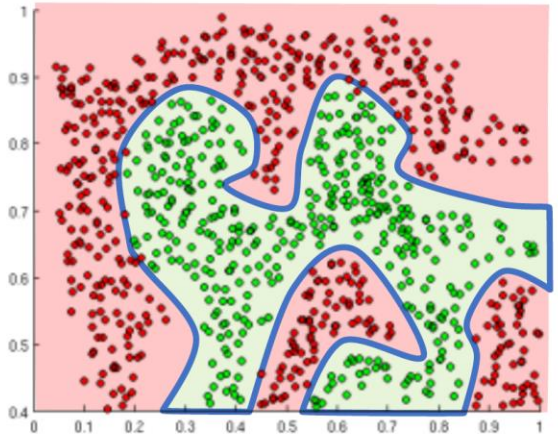
Source: 6.S191 Intro. to Deep Learning at MIT

Importance of Activation Functions

The purpose of activation functions is to ***introduce non-linearities*** *into the network*



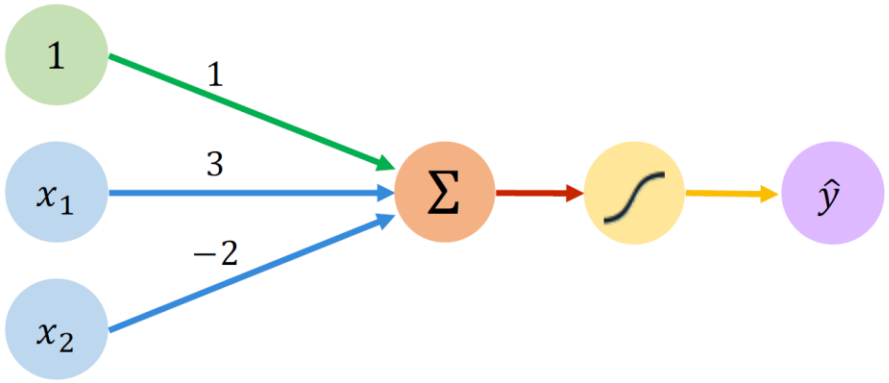
Linear Activation functions produce linear decisions no matter the network size



Non-linearities allow us to approximate arbitrarily complex functions

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

Perceptron: Example

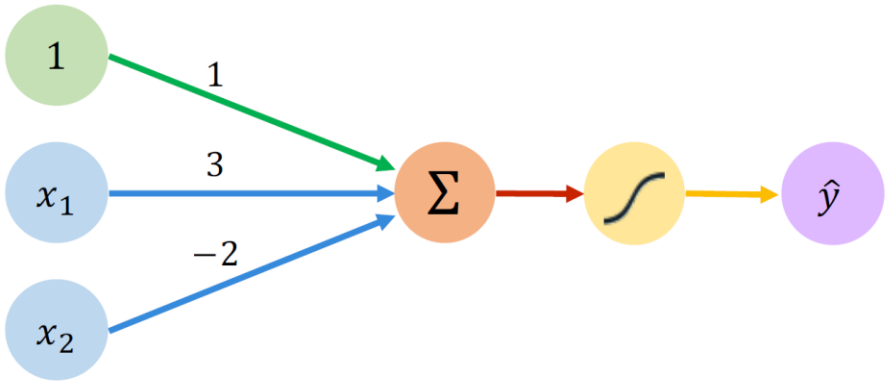


We have: $\theta_0 = 1$ and $\boldsymbol{\theta} = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$

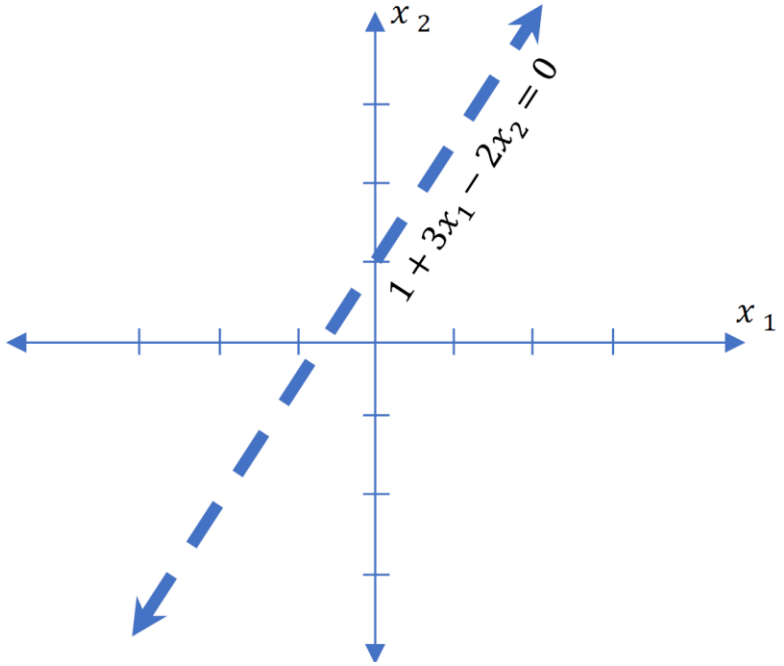
$$\hat{y} = g(\theta_0 + \boldsymbol{X}^T \boldsymbol{\theta})$$
$$= g\left(1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 3 \\ -2 \end{bmatrix}\right)$$
$$\hat{y} = g(\underbrace{1 + 3x_1 - 2x_2}_{\text{This is just a line in 2D!}})$$

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

Perceptron: Example

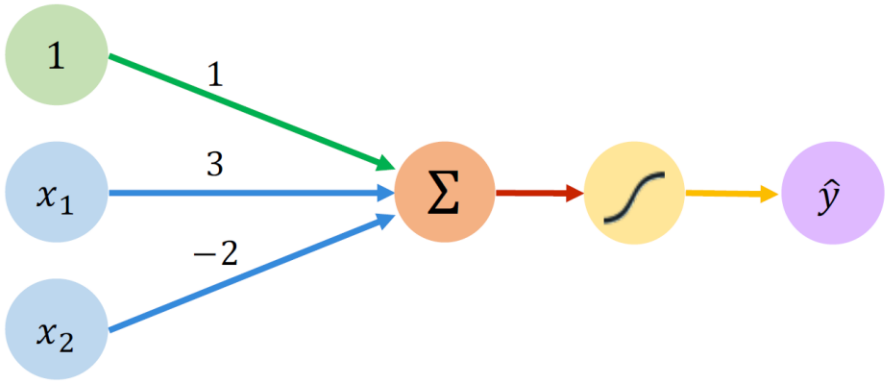


$$\hat{y} = g(1 + 3x_1 - 2x_2)$$



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

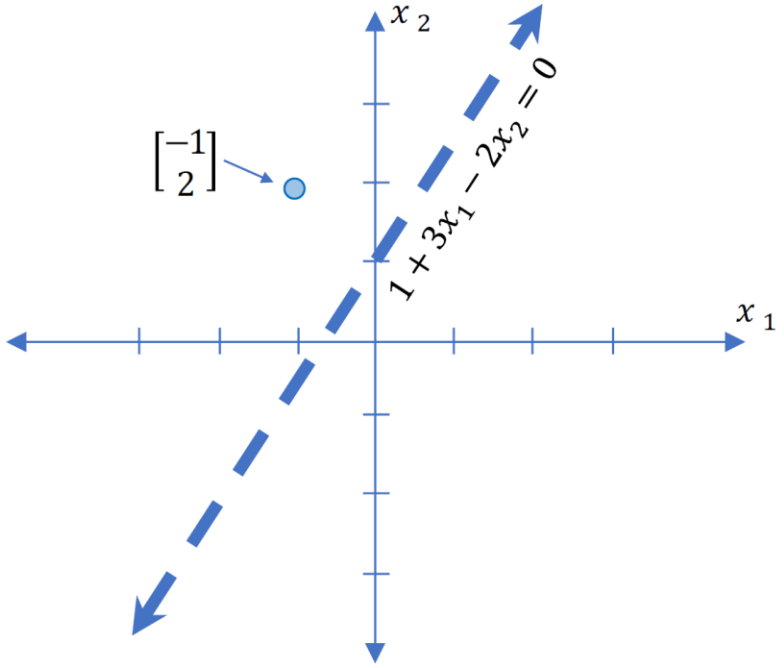
Perceptron: Example



Assume we have input: $\boldsymbol{X} = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$

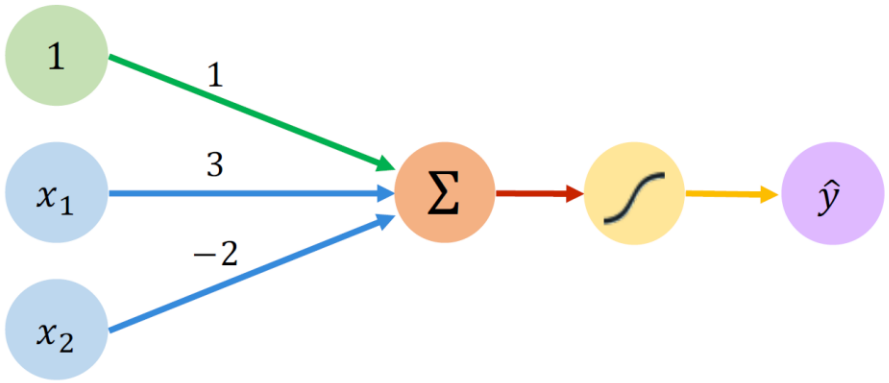
$$\hat{y} = g\left(1 + (3 * -1) - (2 * 2)\right)$$
$$= g(-6) \approx 0.002$$

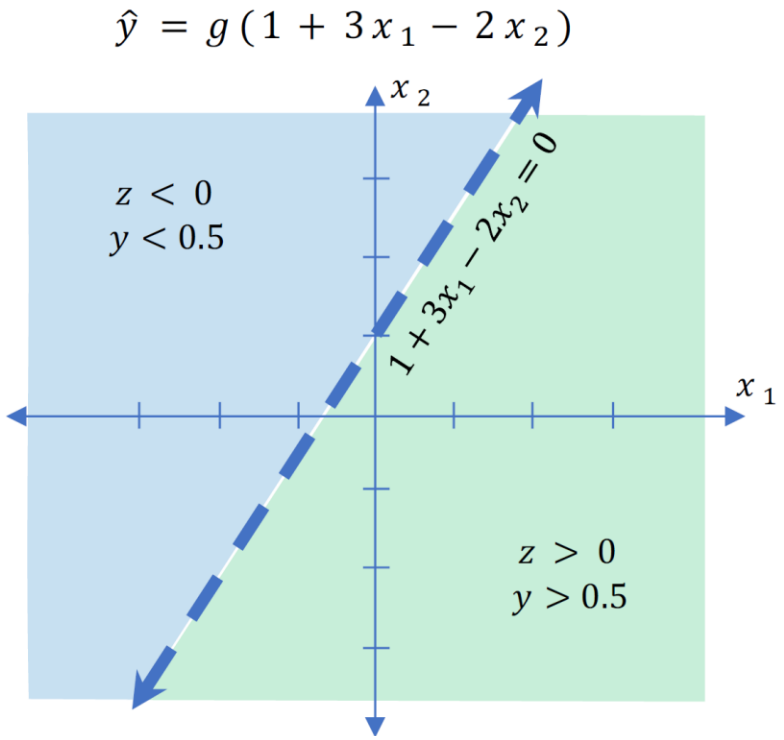
$$\hat{y} = g(1 + 3x_1 - 2x_2)$$



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

Perceptron: Example

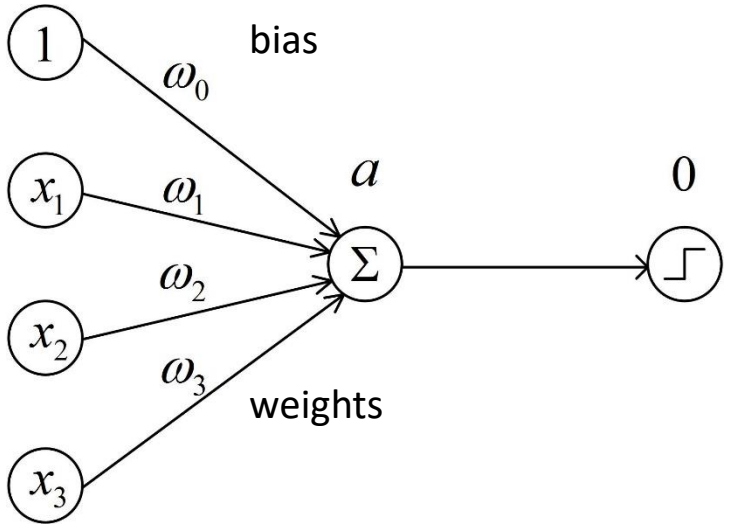




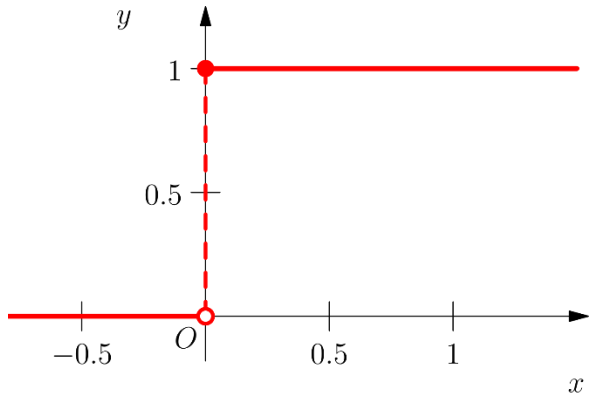
Source: 6.S191 Intro. to Deep Learning at MIT

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



$$a = \omega_0 + \omega_1 x_1 + \cdots$$
$$o = \sigma(\omega_0 + \omega_1 x_1 + \cdots)$$

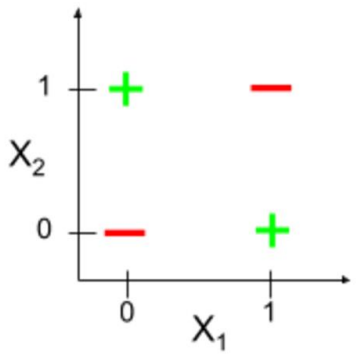


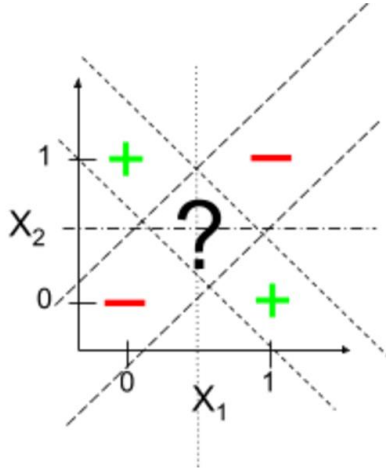
- A step function is not differentiable
- One layer is often not enough

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

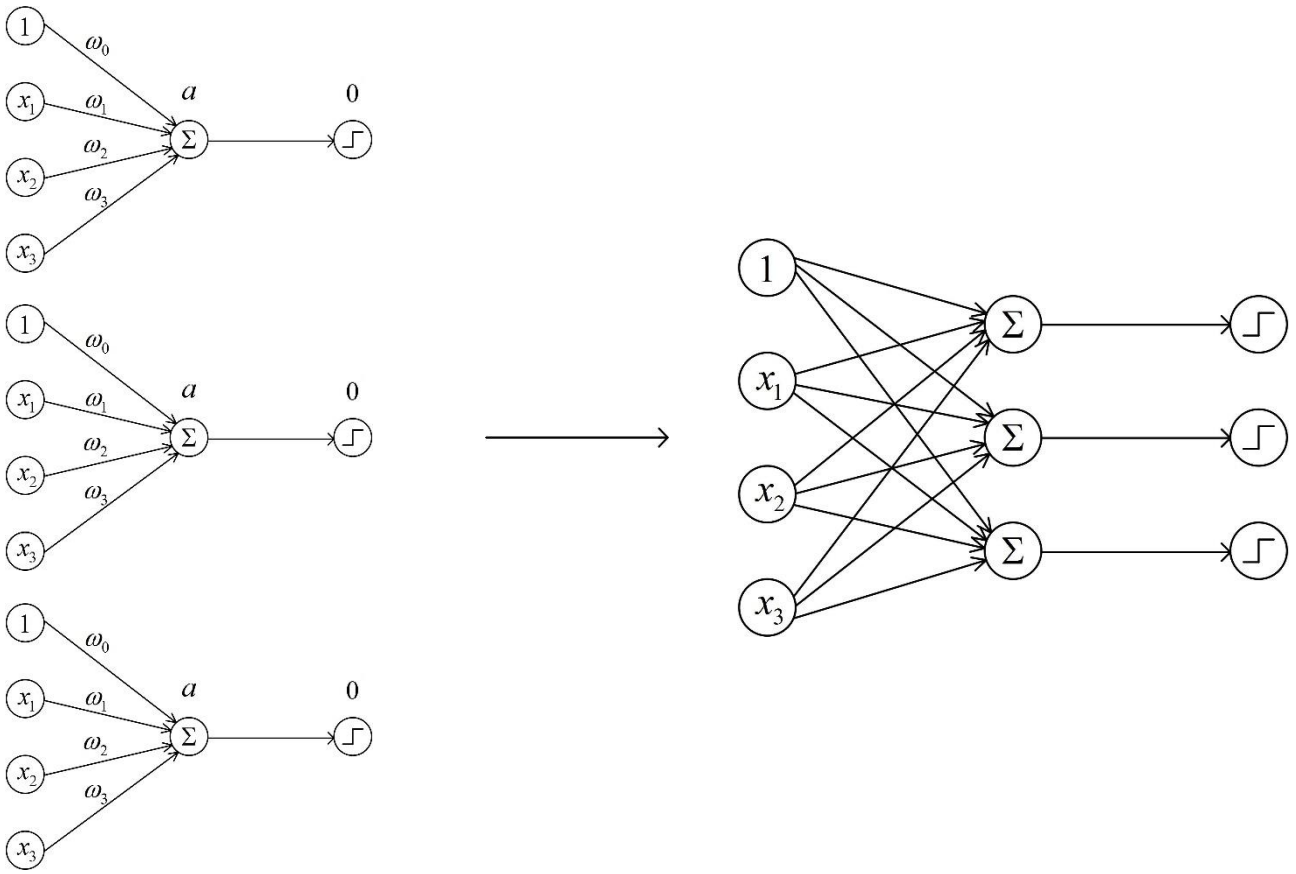




- Single neuron = one linear classification boundary

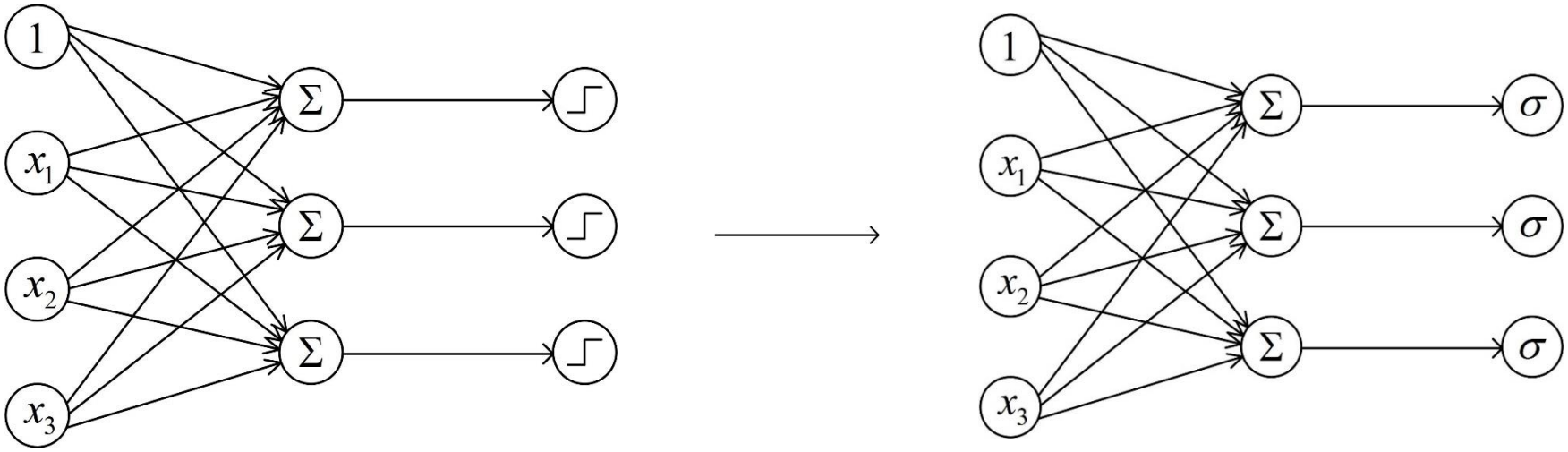
Artificial Neural Networks: MLP

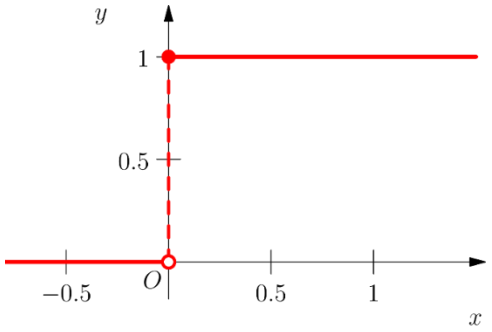
- Multi-layer Perceptron (MLP) = Artificial Neural Networks (ANN)
  - Multi neurons = multiple linear classification boundaries

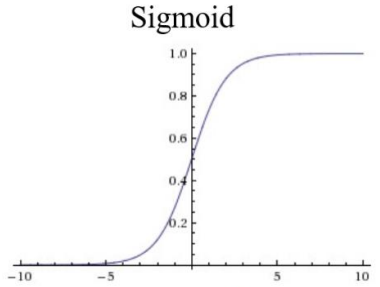


Artificial Neural Networks: Activation Func.

- Differentiable non-linear activation function



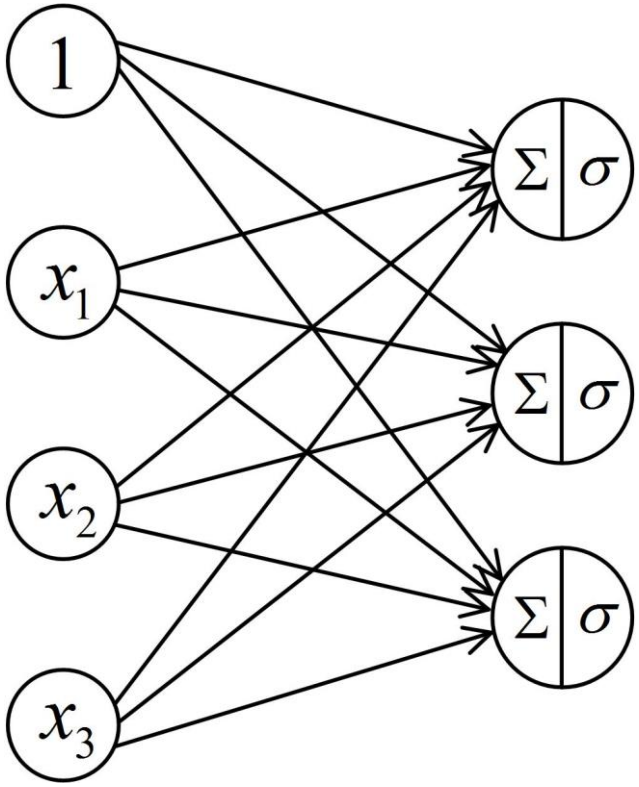




$$g(x) = \frac{1}{1 + e^{-x}}$$

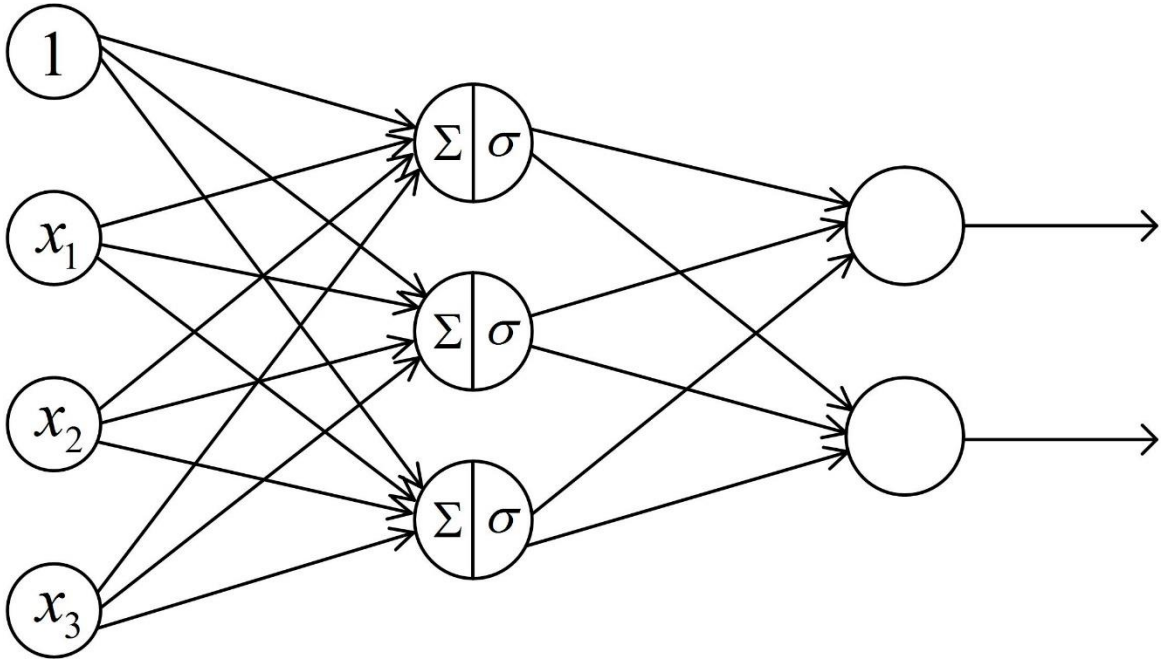
Artificial Neural Networks

- In a compact representation



Artificial Neural Networks

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



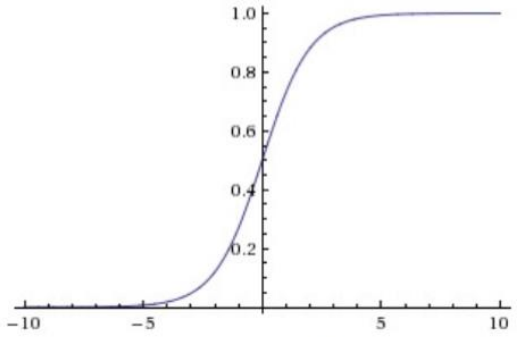
ANN: Transformation

- Affine (or linear) transformation and nonlinear activation layer (notations are mixed: $g = \sigma, \omega = \theta, \omega_0 = b$)

$$o(x) = g\left(\theta^T x + b\right)$$

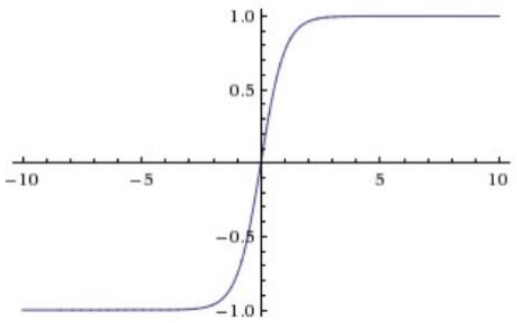
- Nonlinear activation functions ($g = \sigma$)

Sigmoid



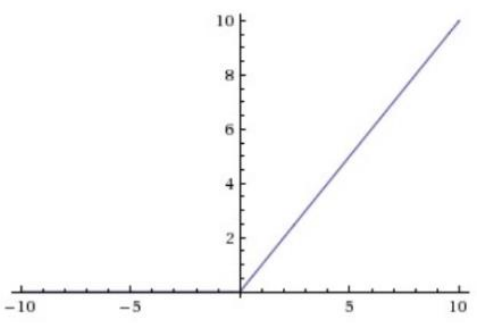
$$g(x) = \frac{1}{1 + e^{-x}}$$

tanh



$$g(x) = \tanh(x)$$

Rectified Linear Unit



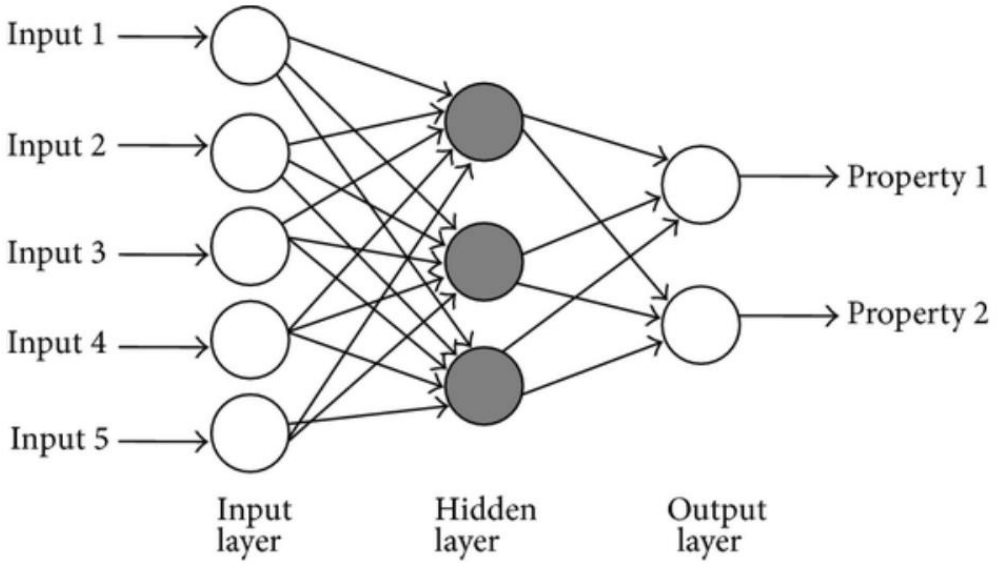
$$g(x) = \max(0, x)$$

ANN: Architecture

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

  $\Longrightarrow$ perceptron with many layers and units

$$o_2 = \sigma_2\left(\theta_2^T o_1 + b_2\right) = \sigma_2\left(\theta_2^T \sigma_1\left(\theta_1^T x + b_1\right) + b_2\right)$$



- The perceptron classification rule boils down to

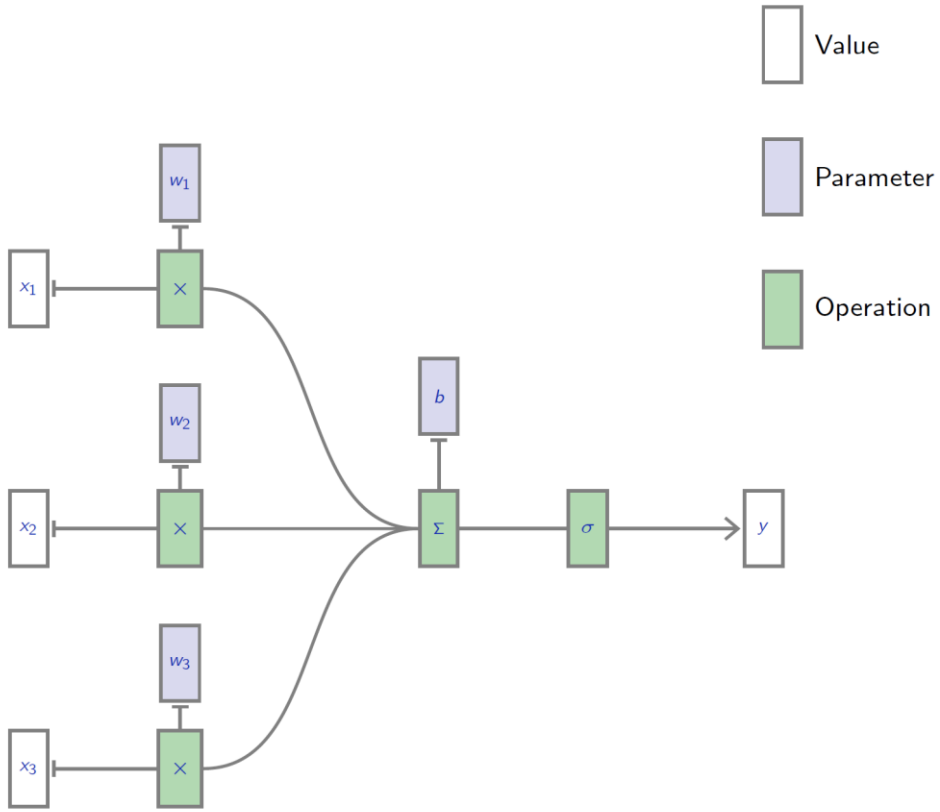
$$f(x) = \sigma(w \cdot x + b).$$

- For neural networks, the function σ that follows a linear operator is called the activation function.

Source: Dr. Francois Fleuret at EPFL

- We can also use tensor operations, as in

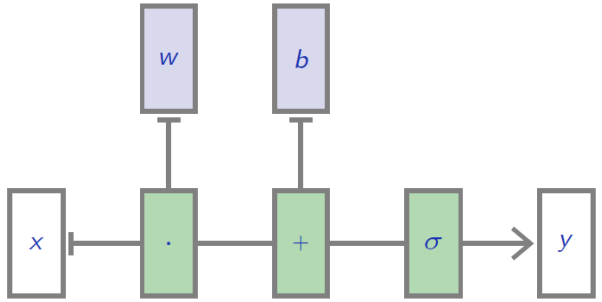
$$f(x) = \sigma(w \cdot x + b).$$



Source: Dr. Francois Fleuret at EPFL

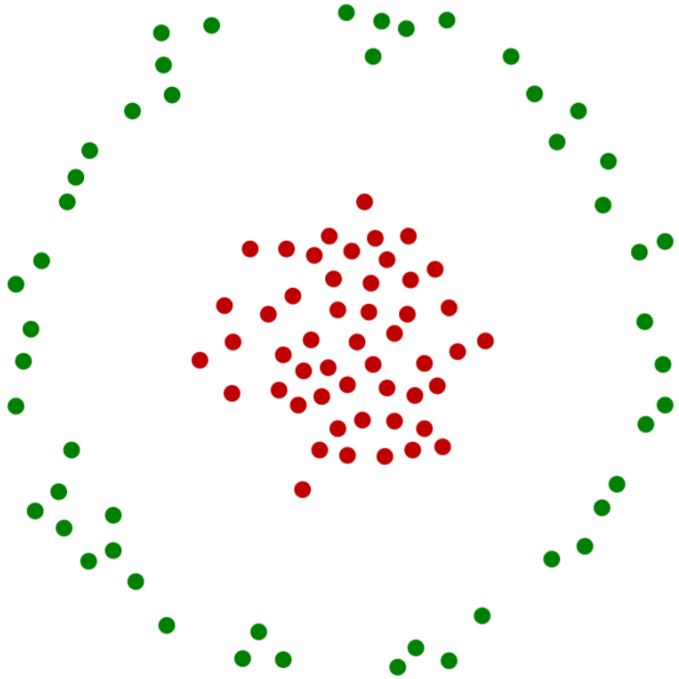
- We can represent this “neuron” as follows:

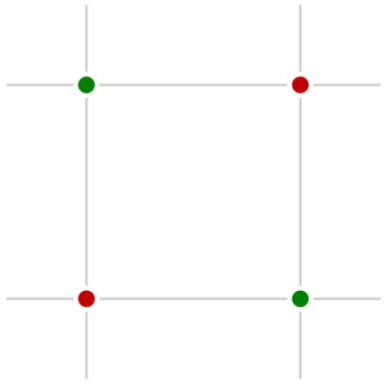
$$f(x) = \sigma(w \cdot x + b).$$



Source: Dr. Francois Fleuret at EPFL

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



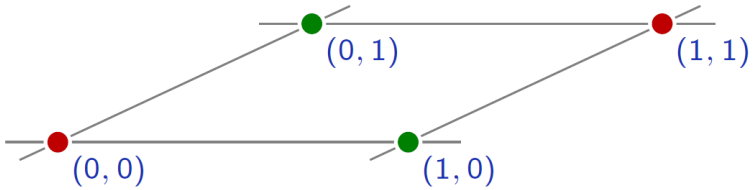


"xor"

Source: Dr. Francois Fleuret at EPFL

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

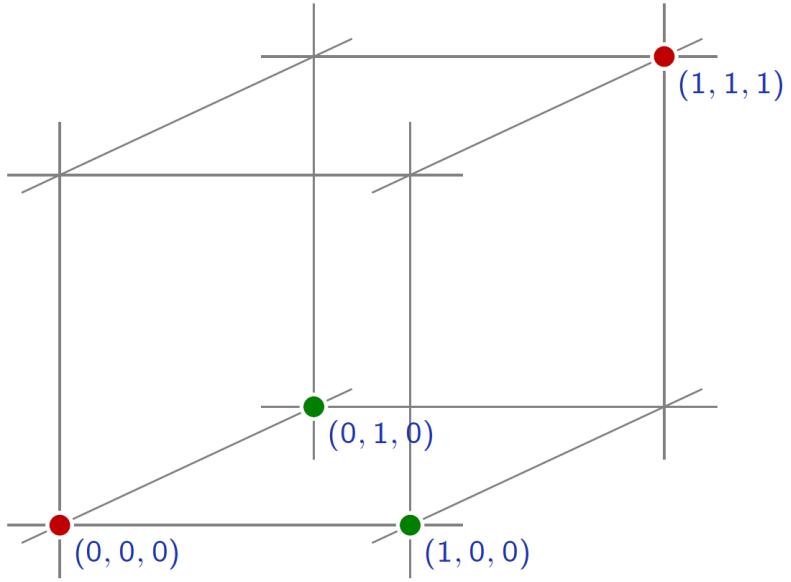
$$\Phi : (x_u, x_v) \mapsto (x_u, x_v, x_u x_v).$$



Source: Dr. Francois Fleuret at EPFL

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

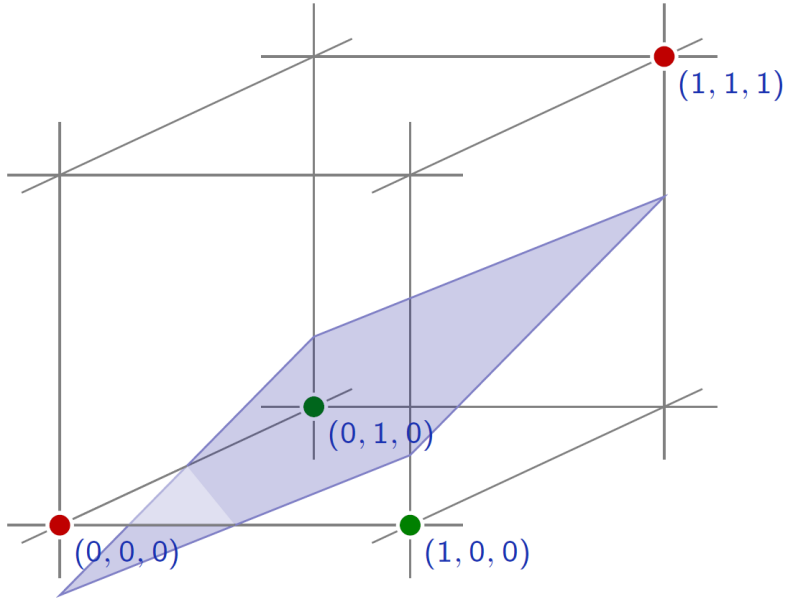
$$\Phi : (x_u, x_v) \mapsto (x_u, x_v, x_u x_v).$$



Source: Dr. Francois Fleuret at EPFL

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

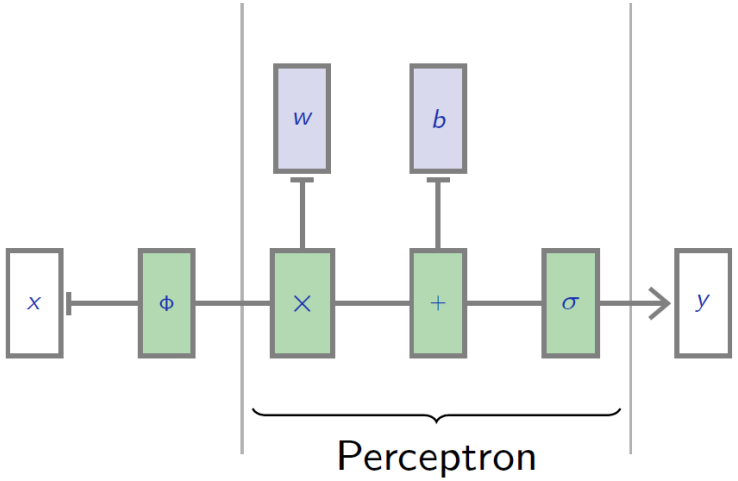
$$\Phi : (x_u, x_v) \mapsto (x_u, x_v, x_u x_v).$$



Source: Dr. Francois Fleuret at EPFL

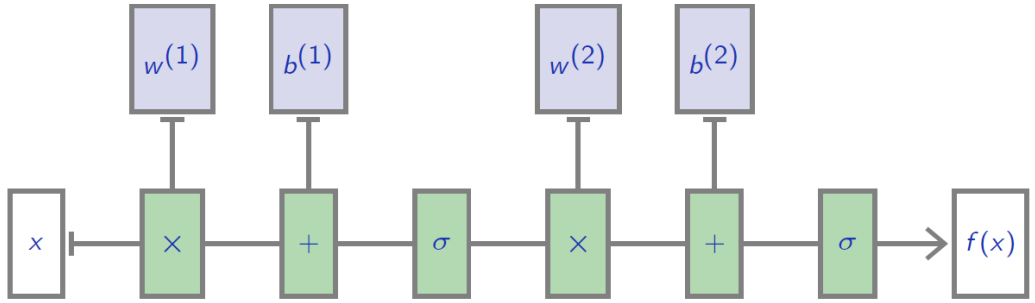
- Nonlinear mapping + neuron

$$\Phi : (x_u, x_v) \mapsto (x_u, x_v, x_u x_v).$$



Source: Dr. Francois Fleuret at EPFL

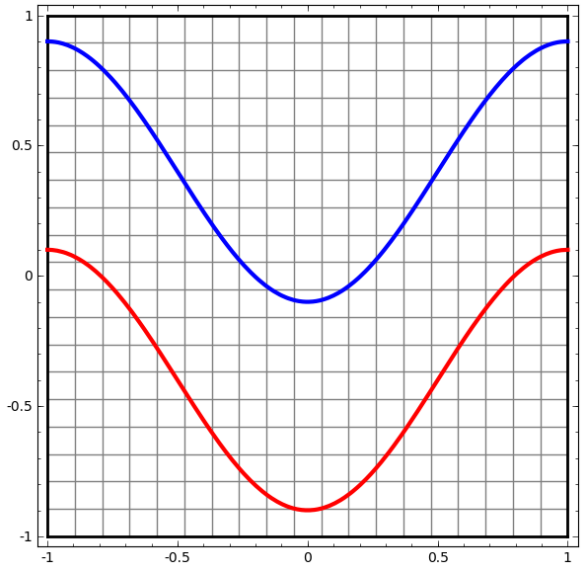
- Nonlinear mapping can be represented by another neurons
- We can generalize an MLP

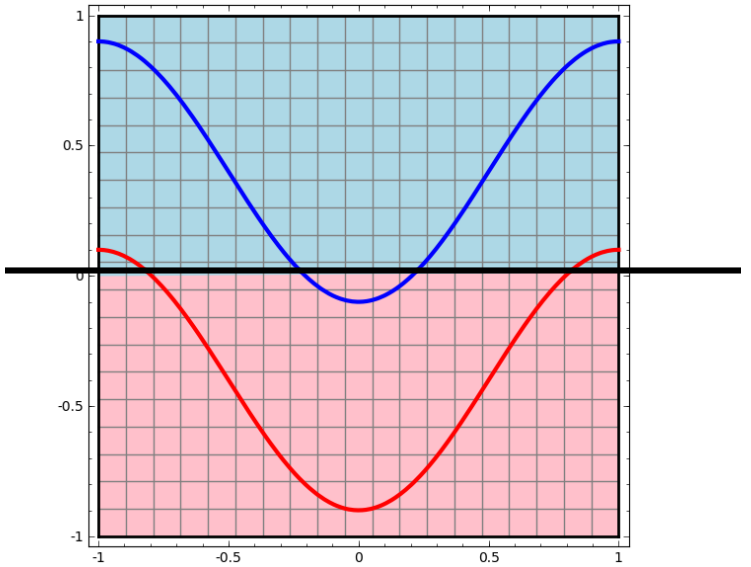


Source: Dr. Francois Fleuret at EPFL

Linear Classifier

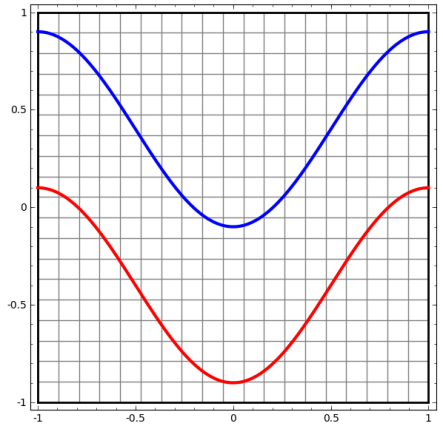
- Perceptron tries to separate the two classes of data by dividing them with a line

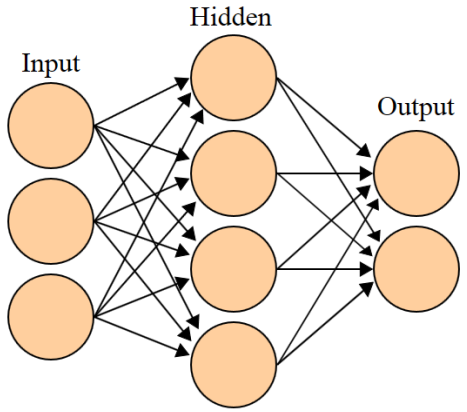


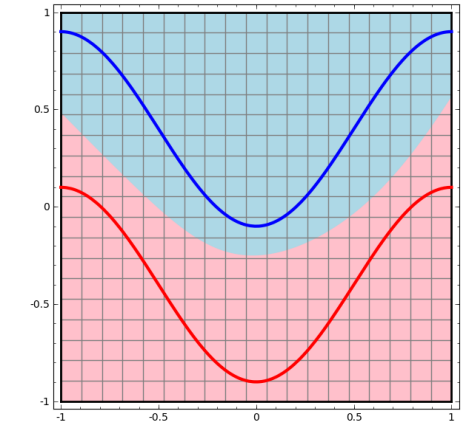


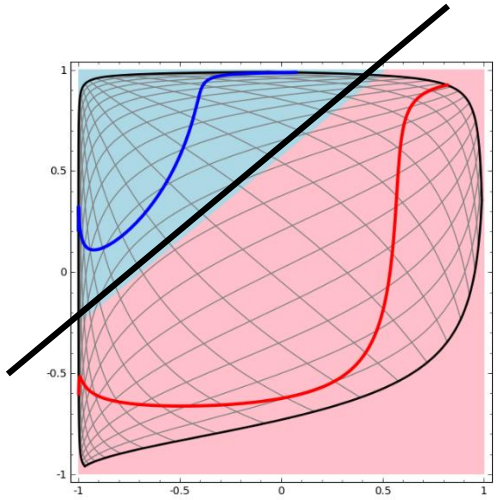
Neural Networks

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





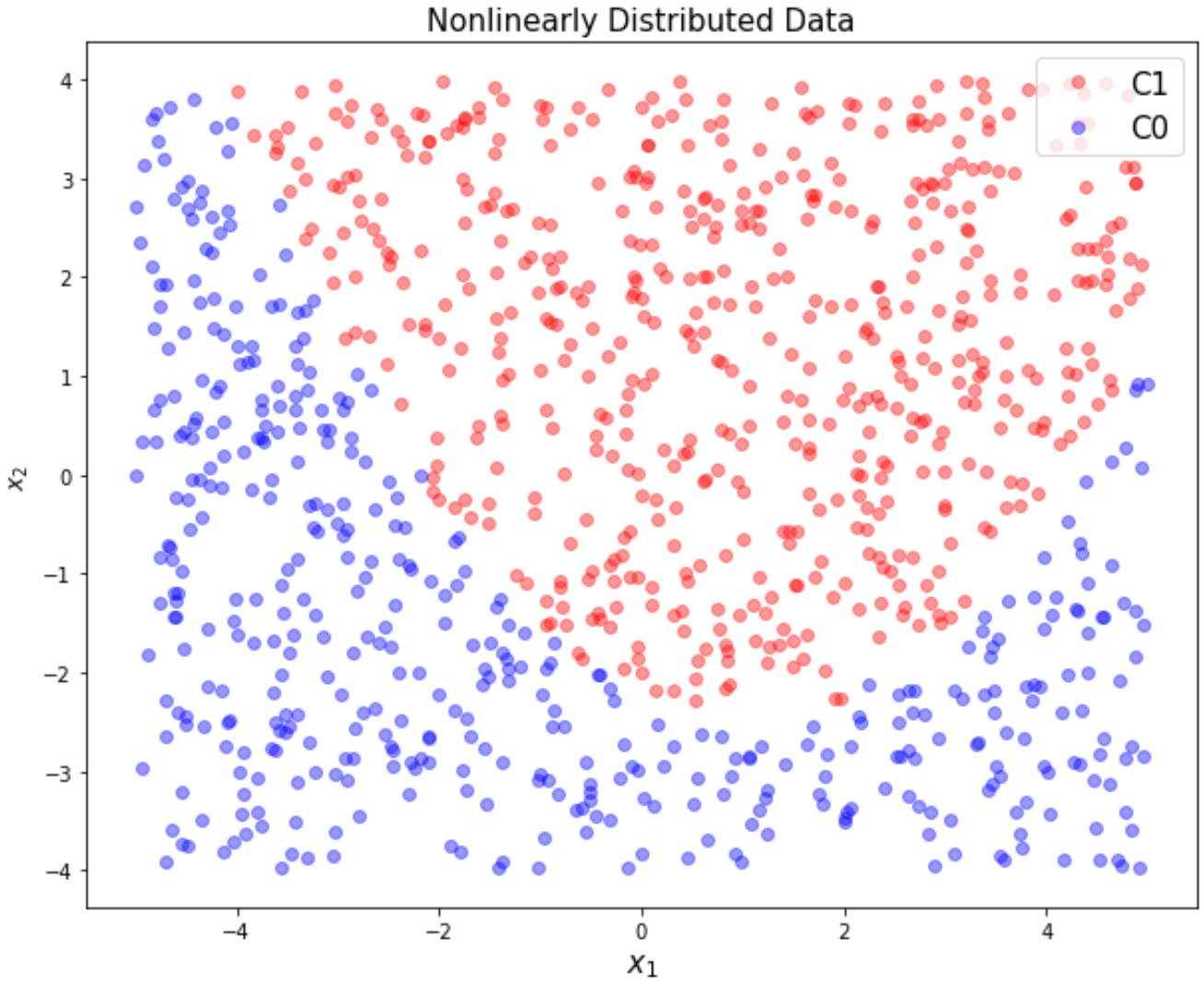




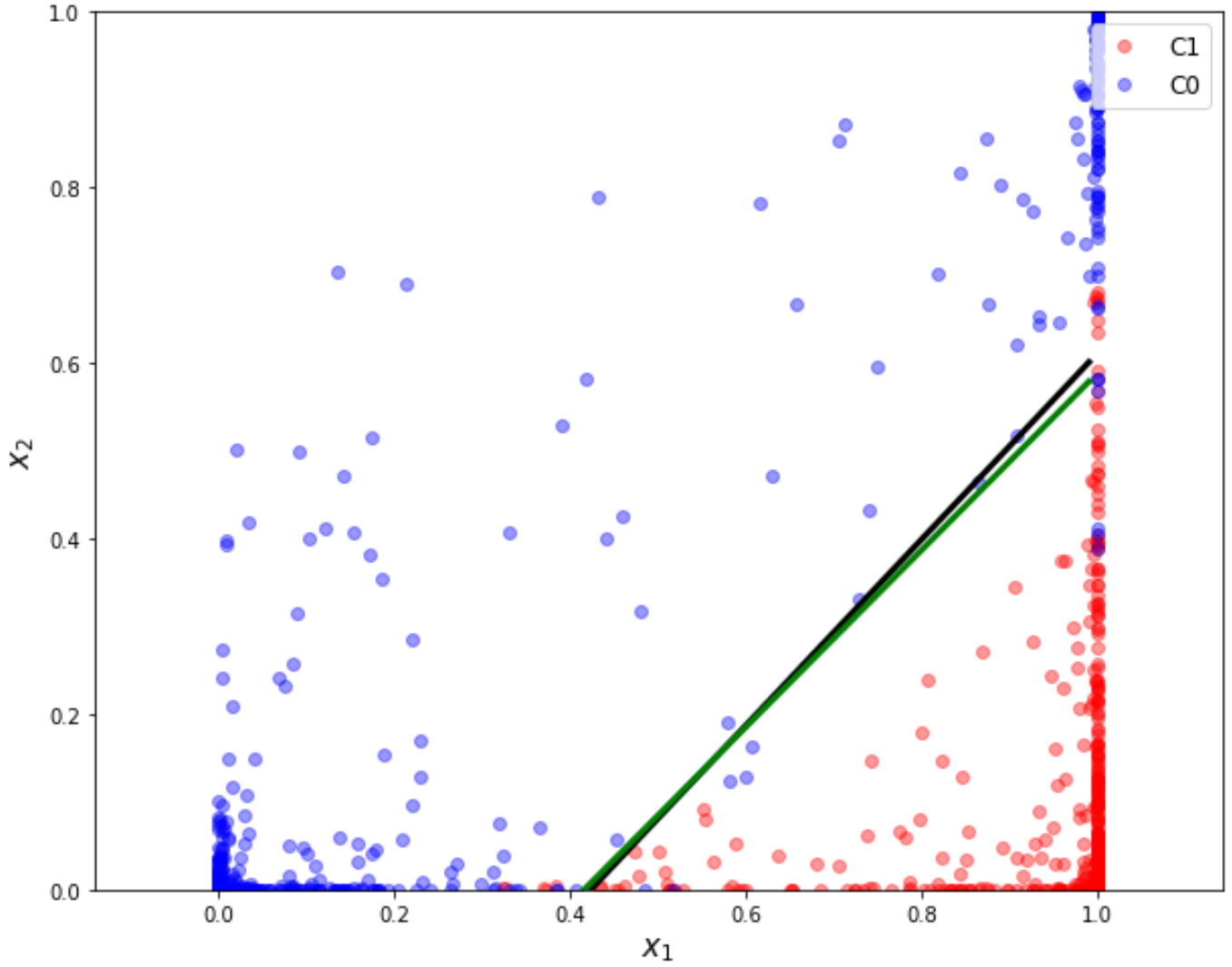
Understanding a Network's Behavior

- Understanding what is happening in a deep architectures after training is complex and the tools we have at our disposal are limited.
- We can look at
  - the network's parameters, filters as images,
  - internal activations as images,
  - distributions of activations on a population of samples,
  - derivatives of the response(s) wrt the input,
  - maximum-response synthetic samples,
  - adversarial samples.

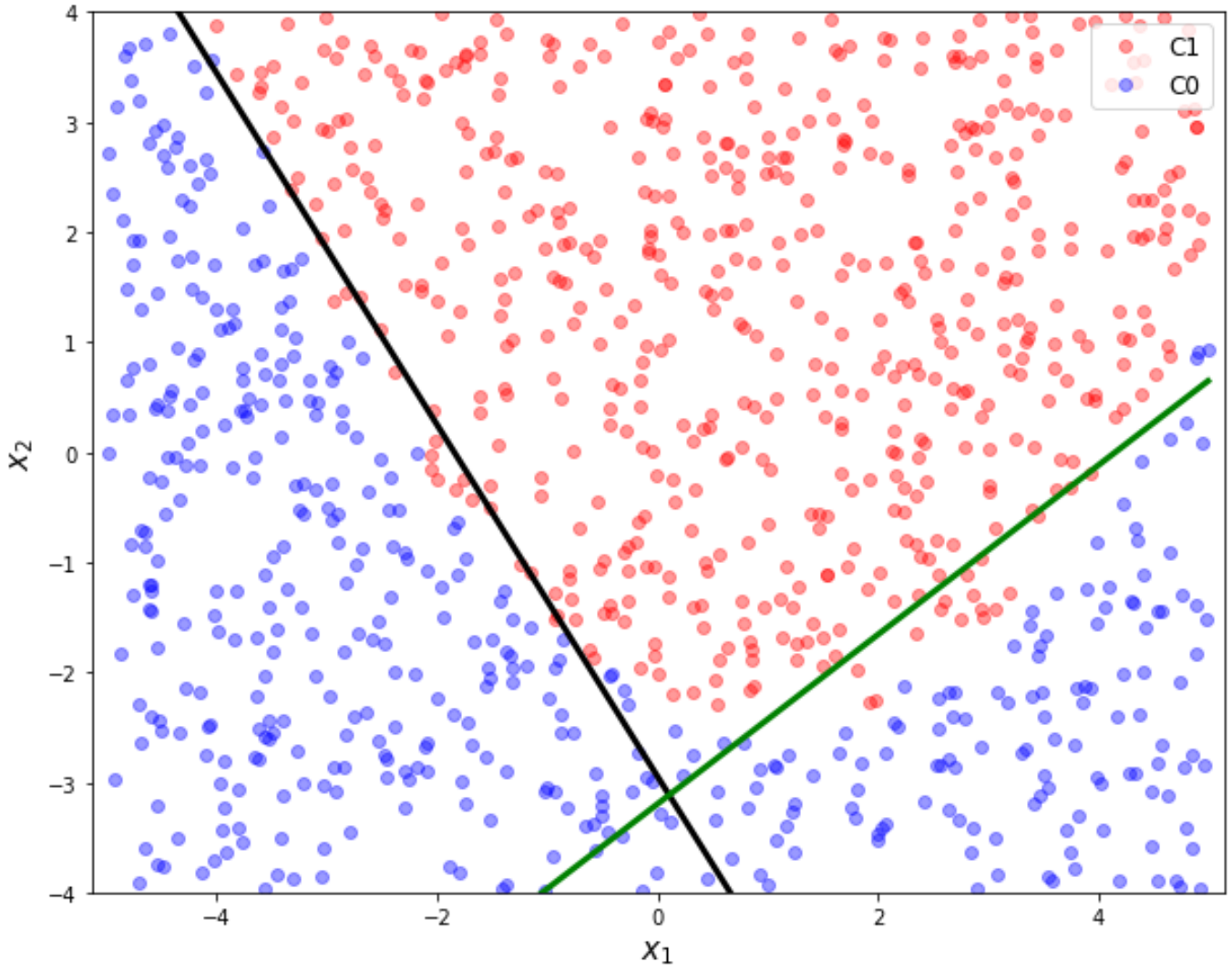
Nonlinearly Distributed Data



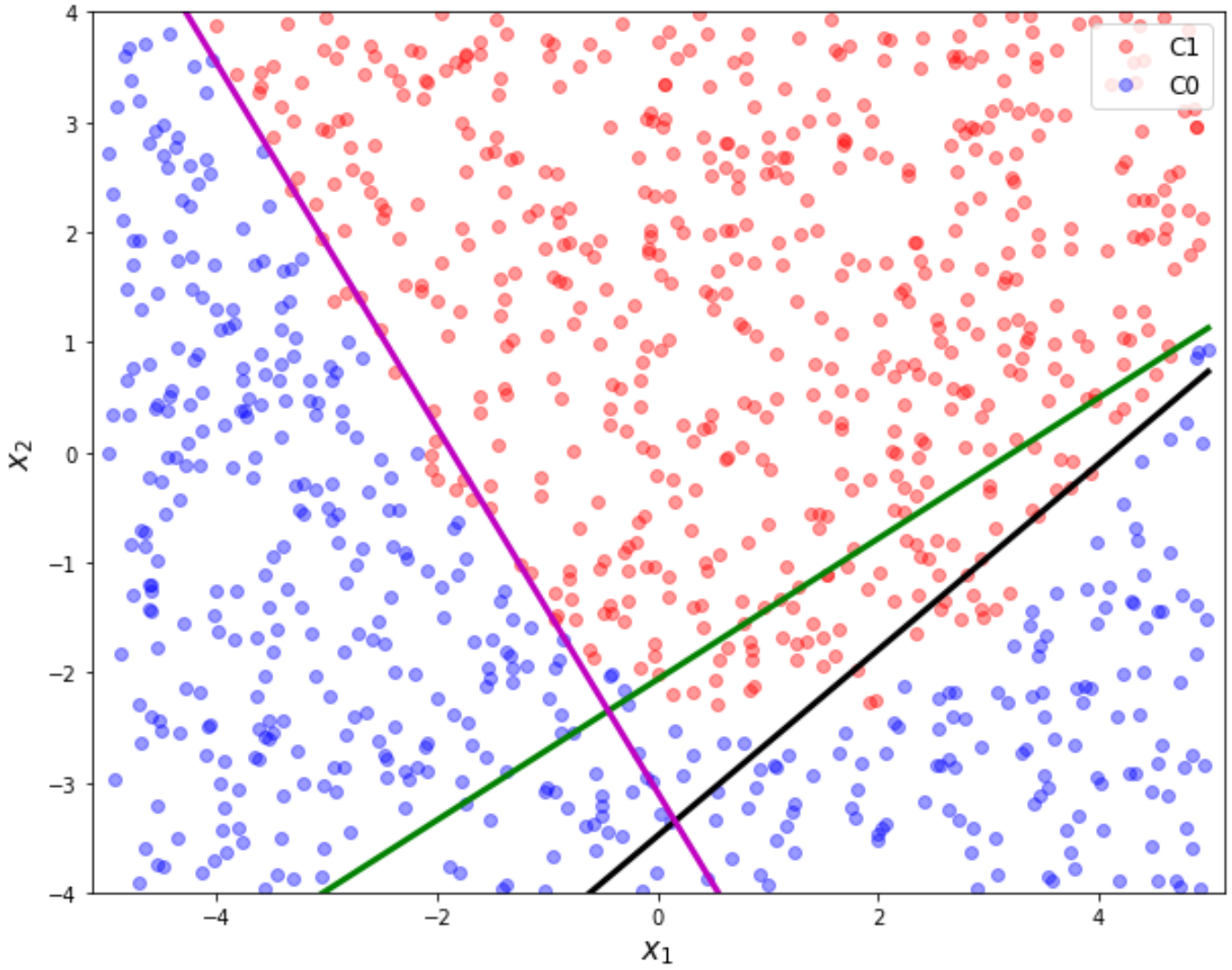
Multi Layers

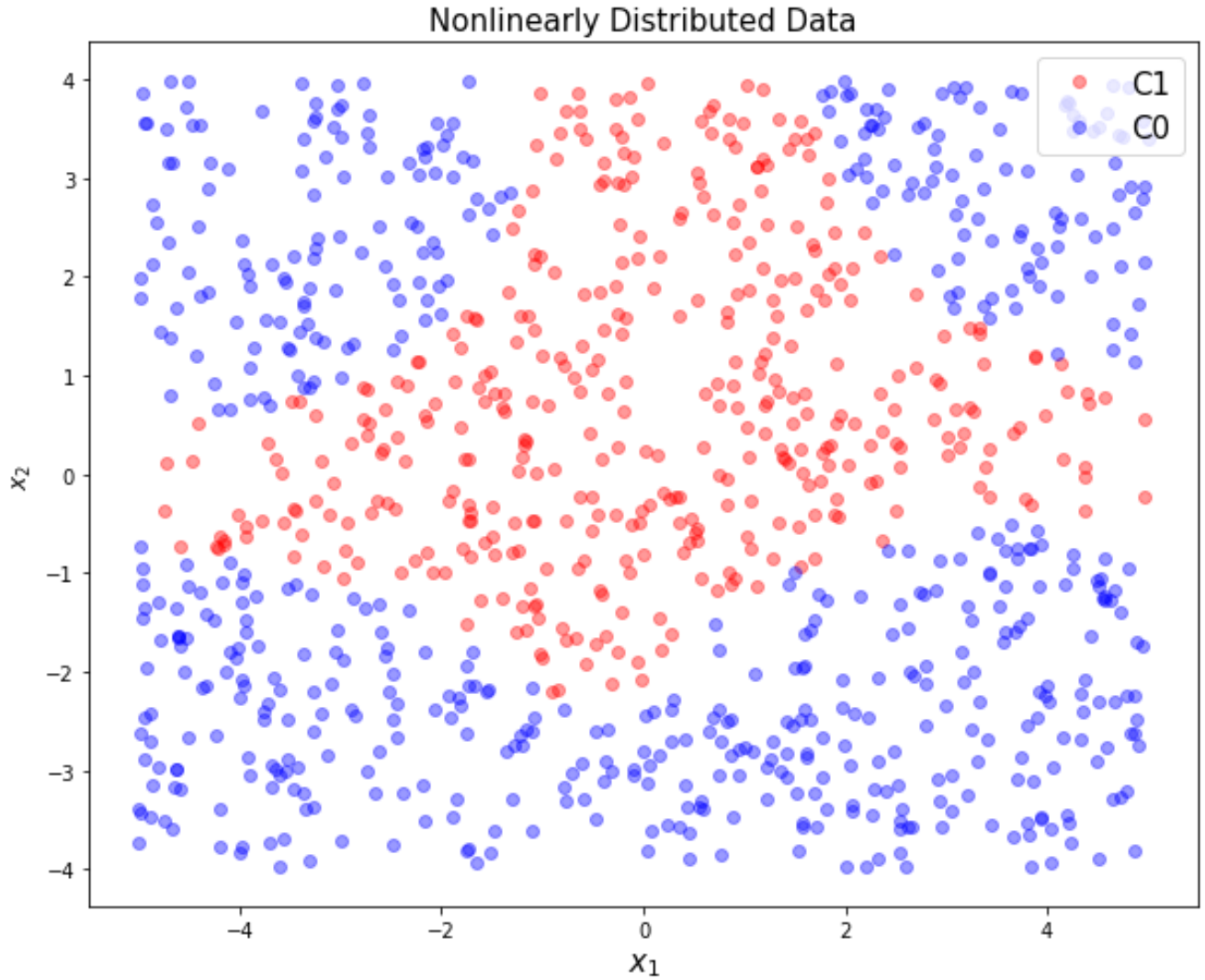


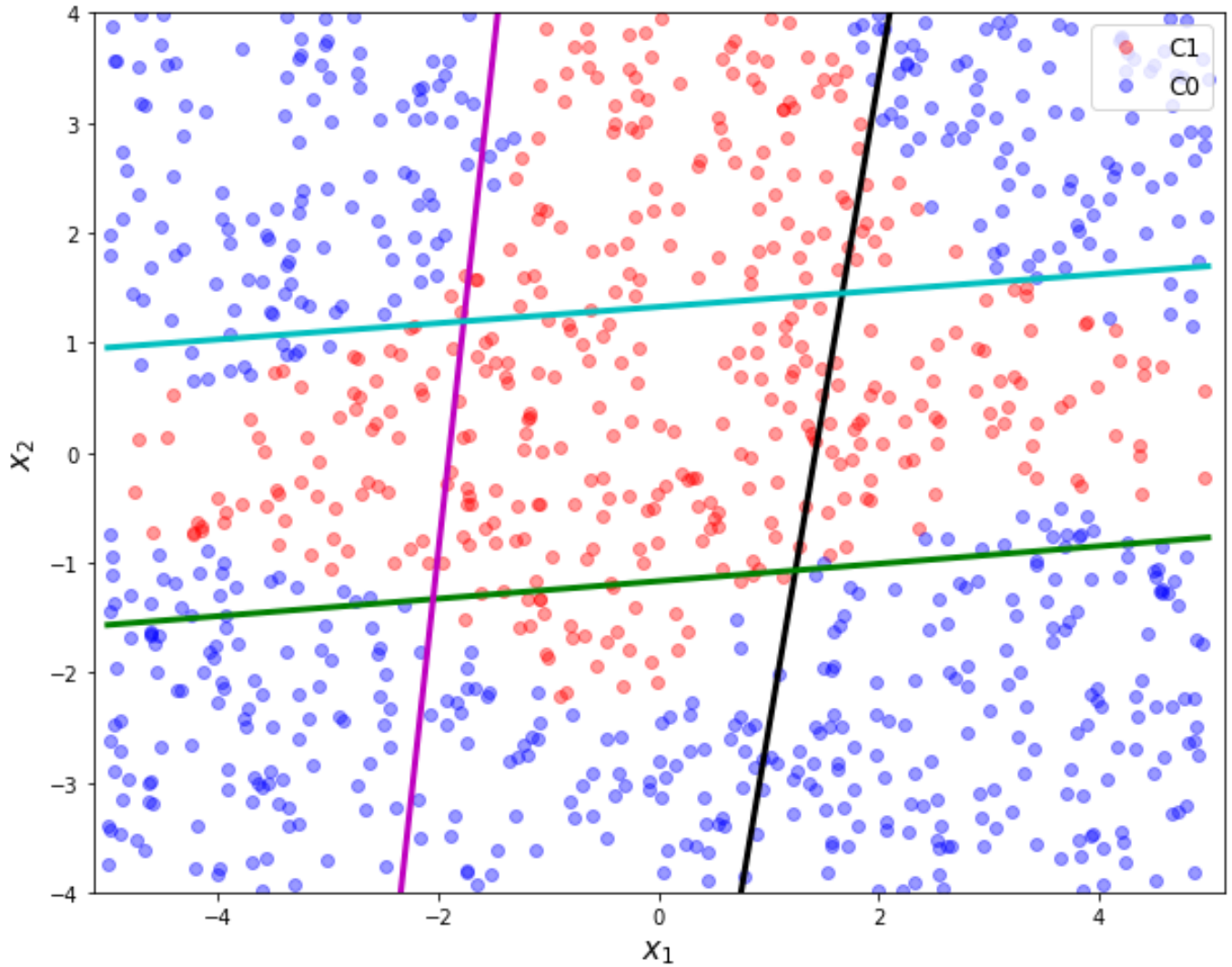
Multi Layers

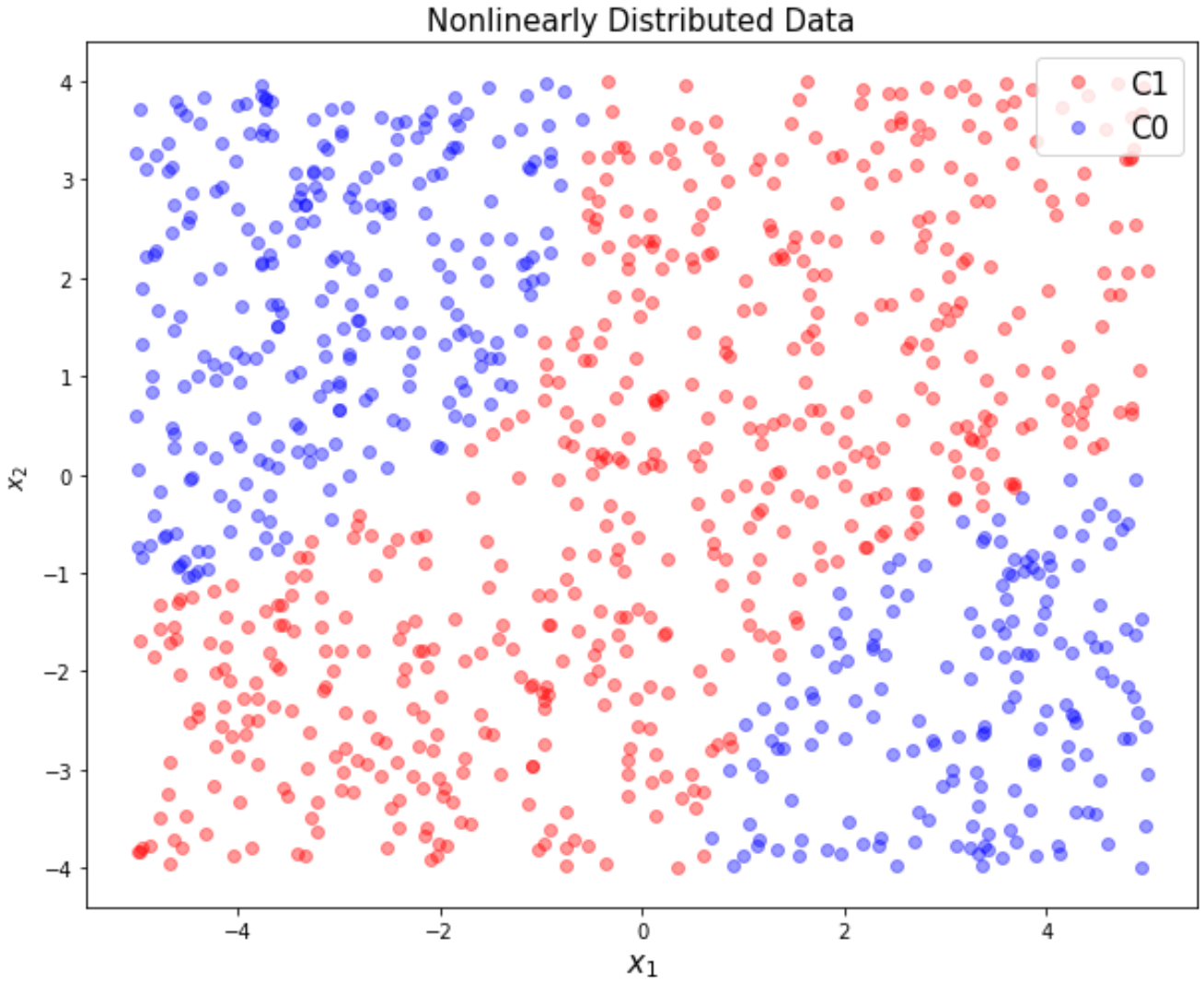


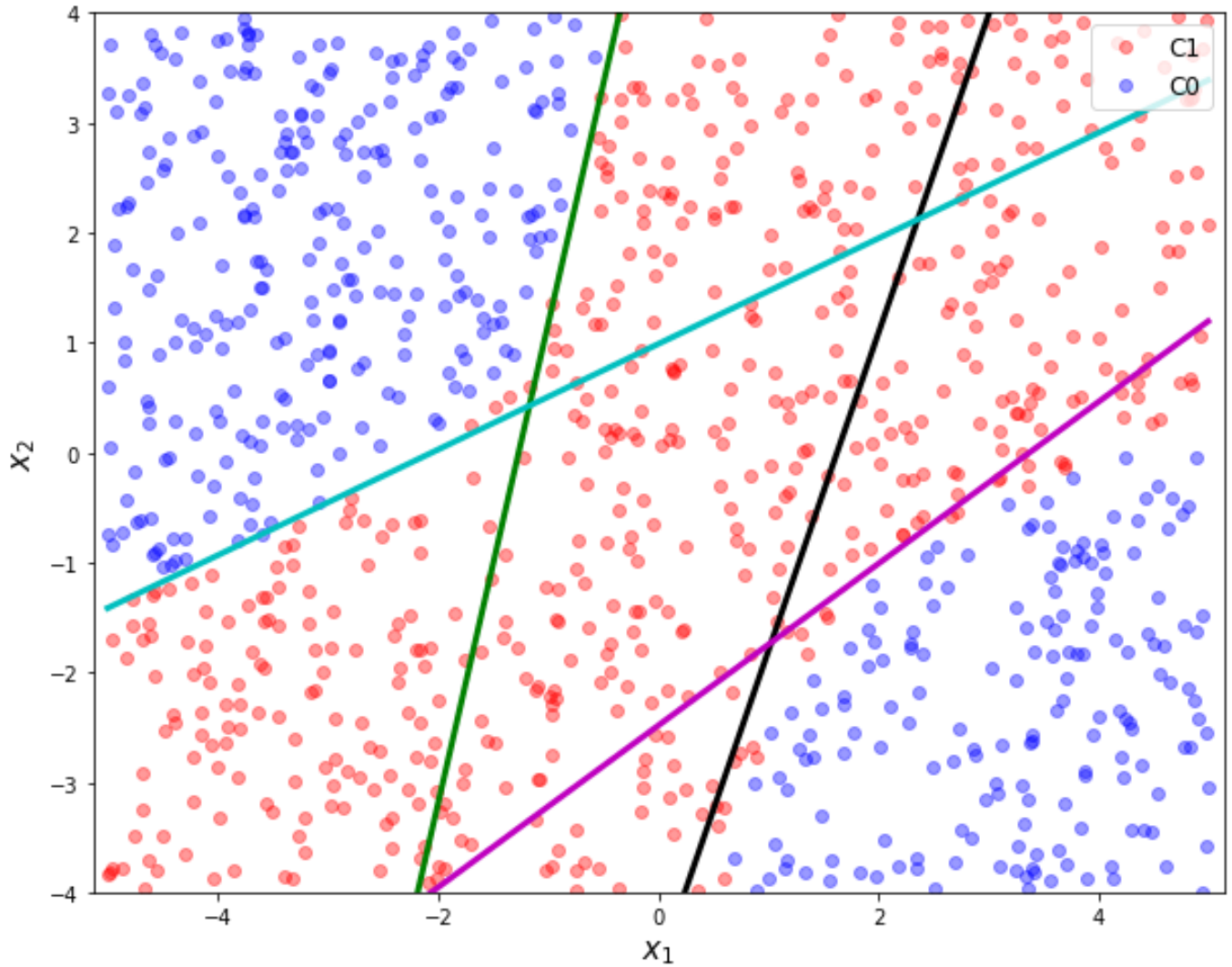
Multi Neurons




Nonlinearly Distributed Data
C1
C0
x_1
x_2
4
3
2
1
0
−1
−2
−3
−4
−4
−2
0
2
4


C1
C0
x_1
x_2
4
3
2
1
0
−1
−2
−3
−4
−4
−2
0
2
4


Nonlinearly Distributed Data
C1
C0
x_1
x_2
4
3
2
1
0
−1
−2
−3
−4
−4
−2
0
2
4


C1
C0
x_1
x_2
4
3
2
1
0
−1
−2
−3
−4
−4
−2
0
2
4

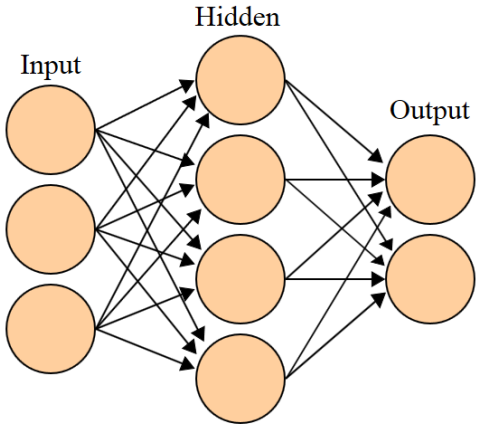
ANN Training

Industrial AI

Prof. Seungchul Lee

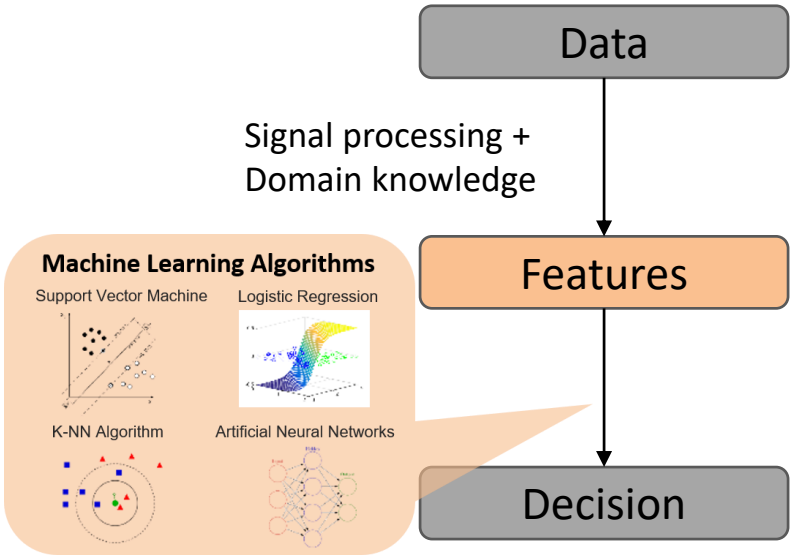
Summary

- Learning weights and biases from data using gradient descent

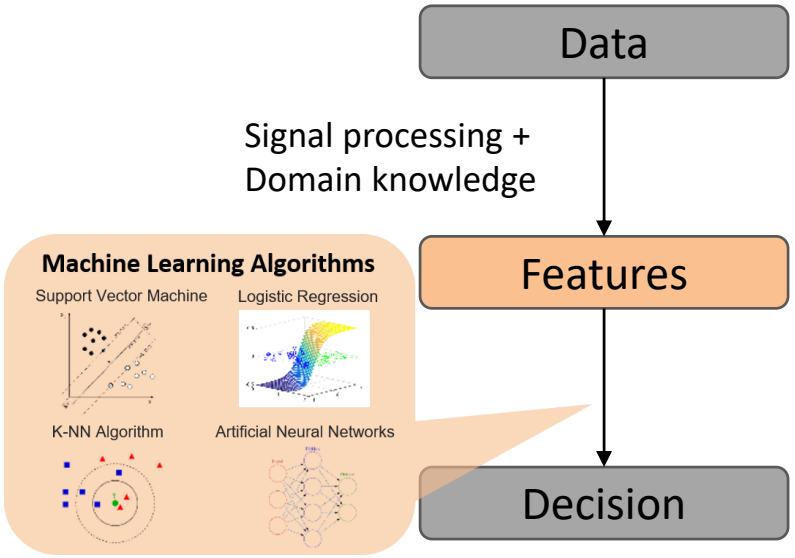


$$x \rightarrow \boxed{\hat{y} = f_{\omega_1,\cdots,\omega_k}(x)} \rightarrow y$$

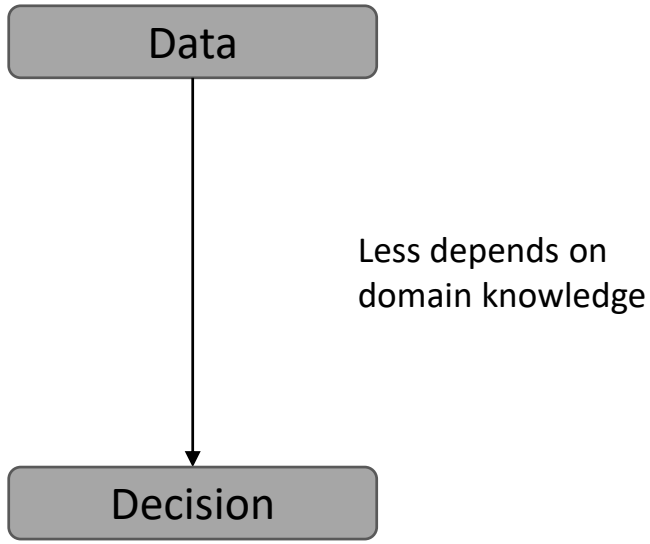
Machine Learning



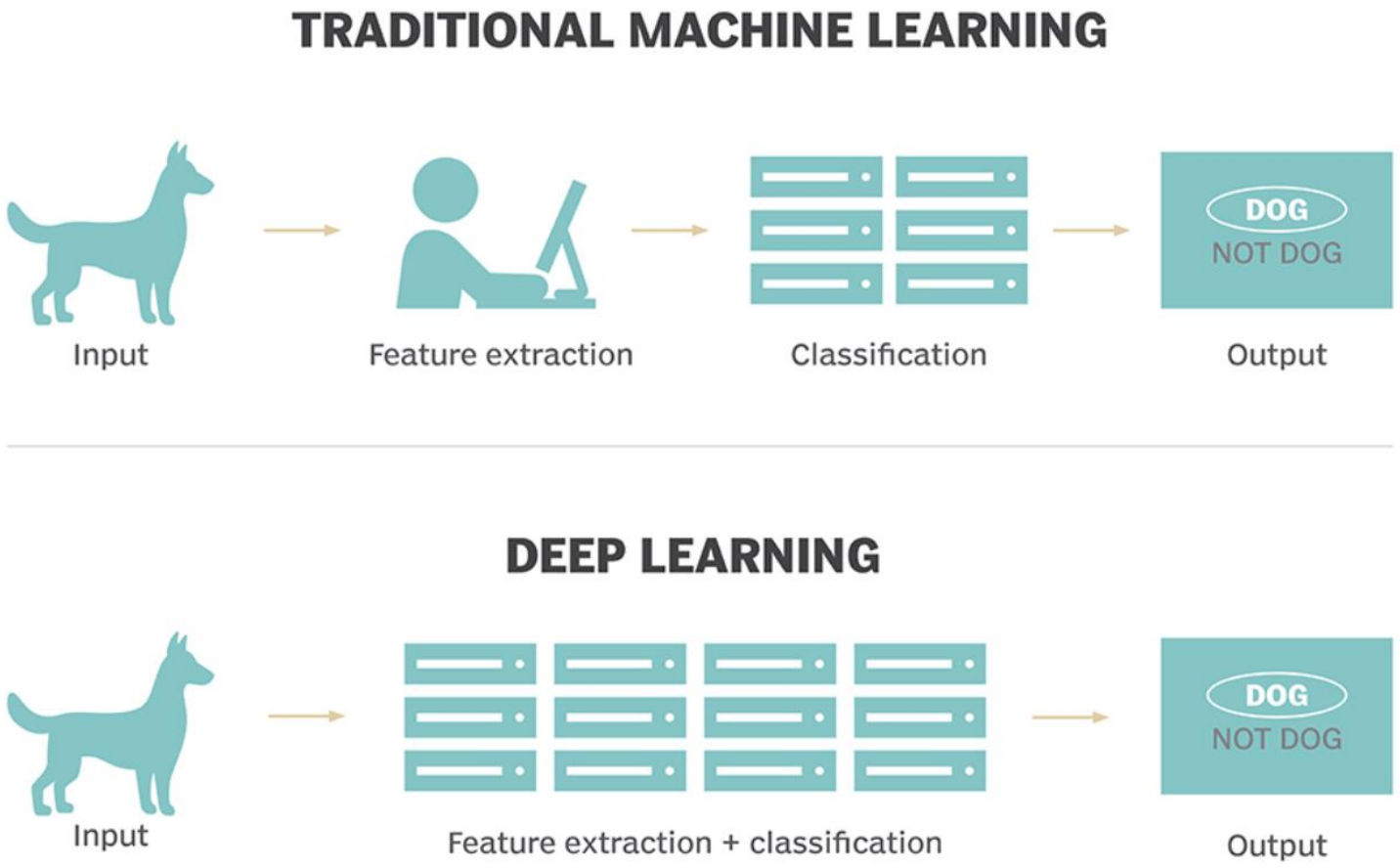
Machine Learning



Deep Learning

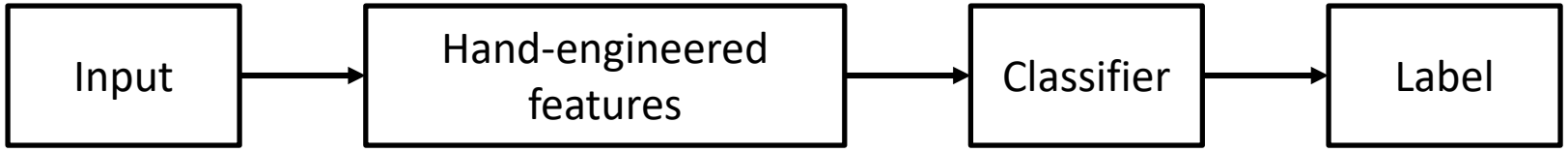


Recall Supervised Learning Setup

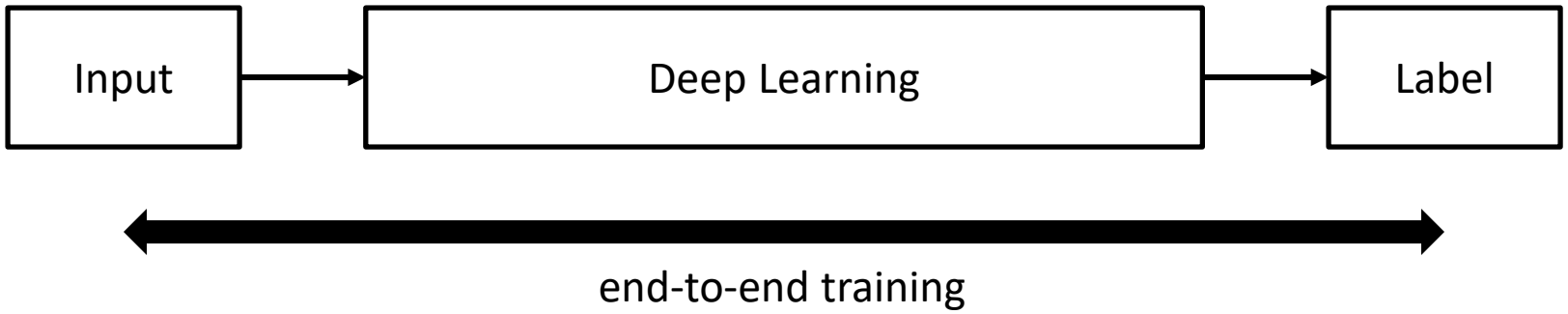


Machine Learning and Deep Learning

- Machine Learning



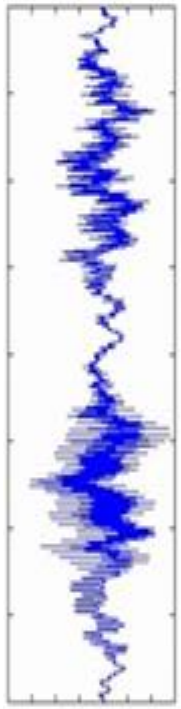
- Deep supervised learning

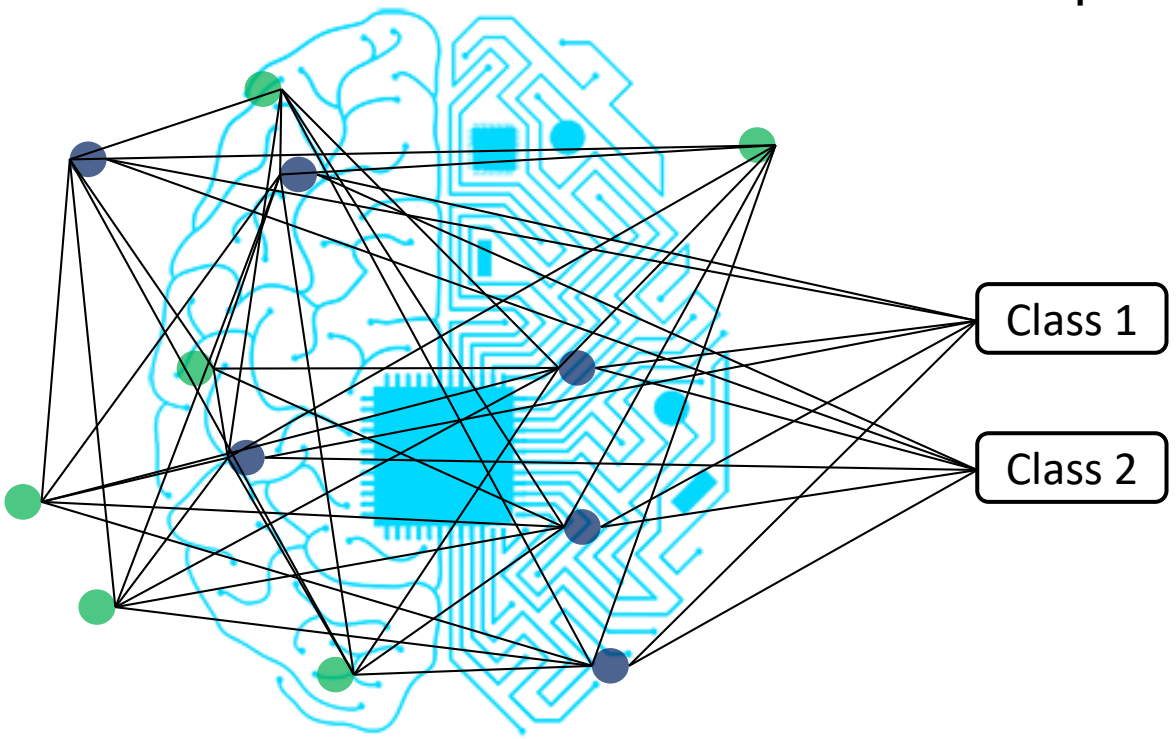


Artificial Neural Networks

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

Input

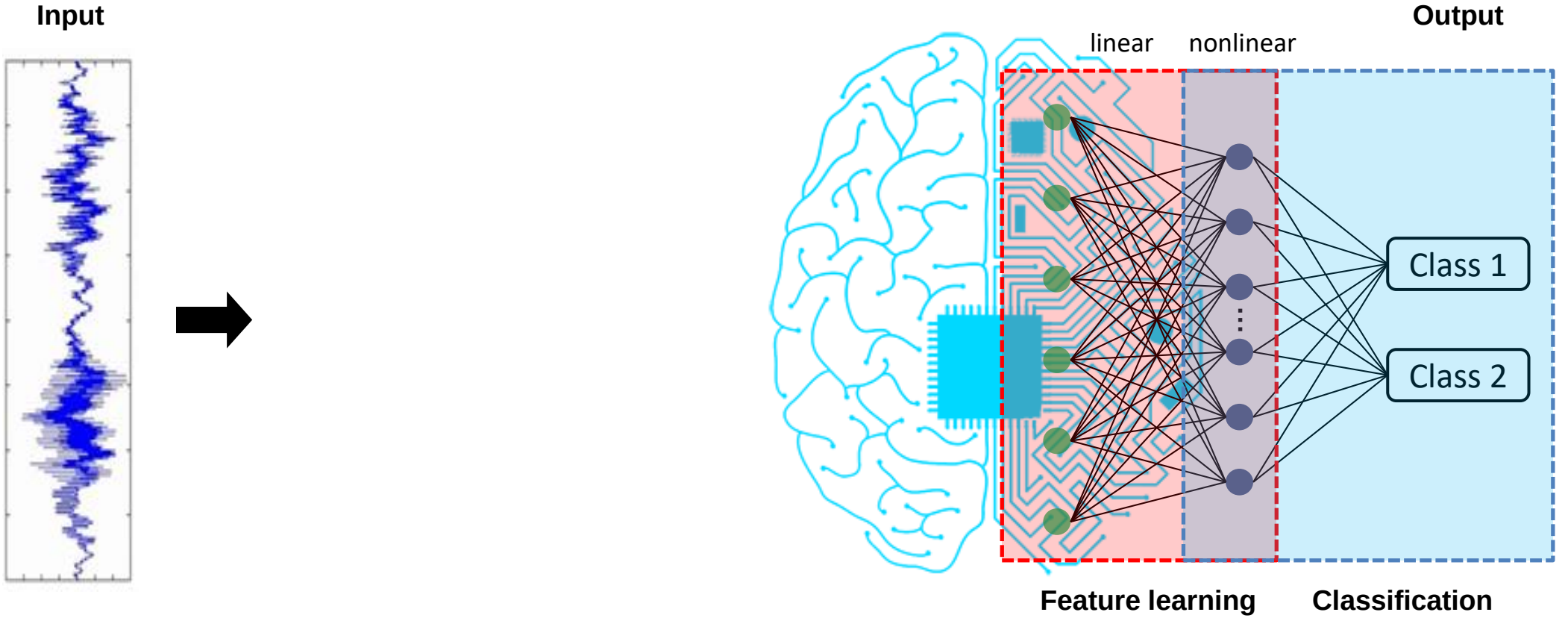




Output

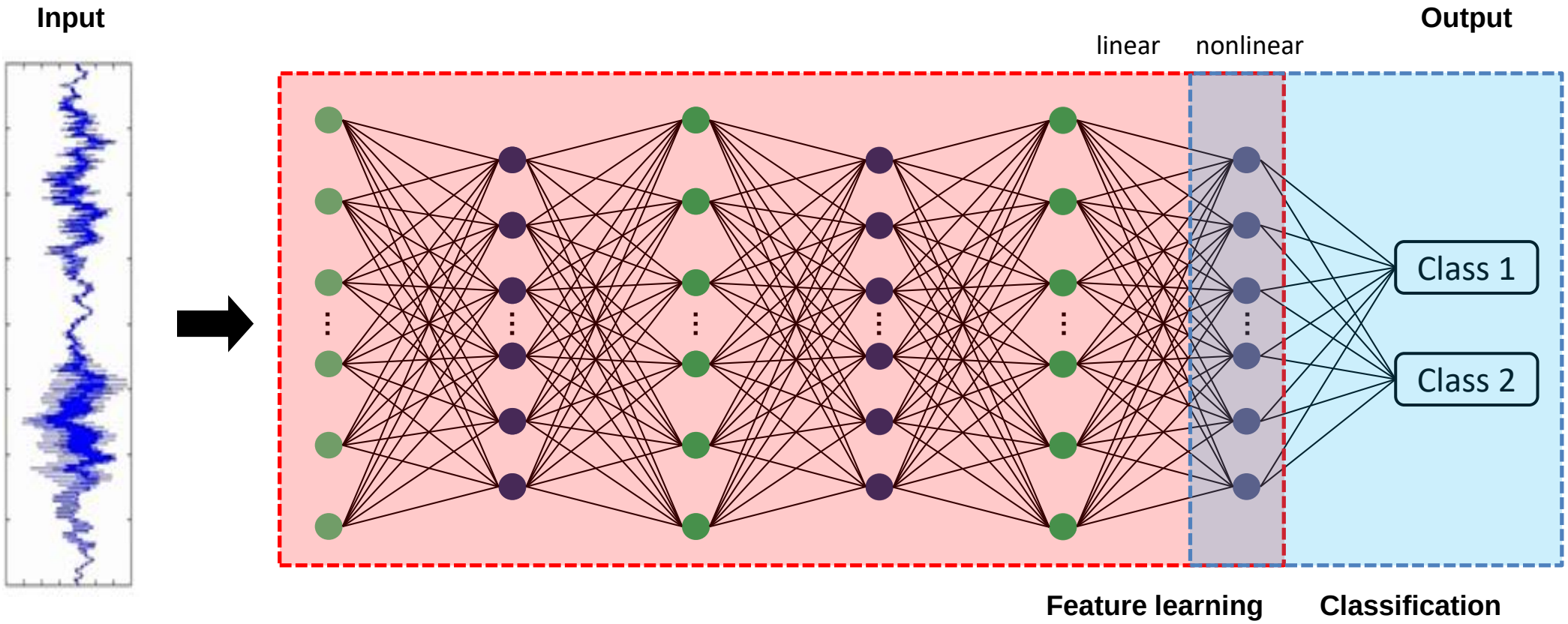
Artificial Neural Networks

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



Deep Artificial Neural Networks

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



Training: Backpropagation

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):

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

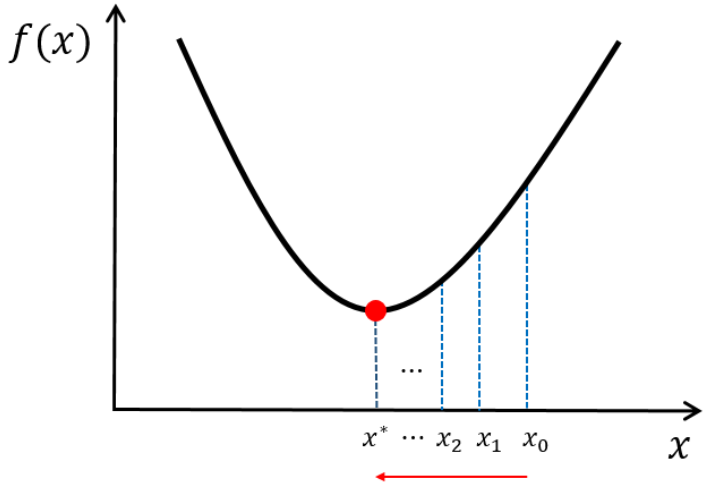
  - Cross entropy (for classification):

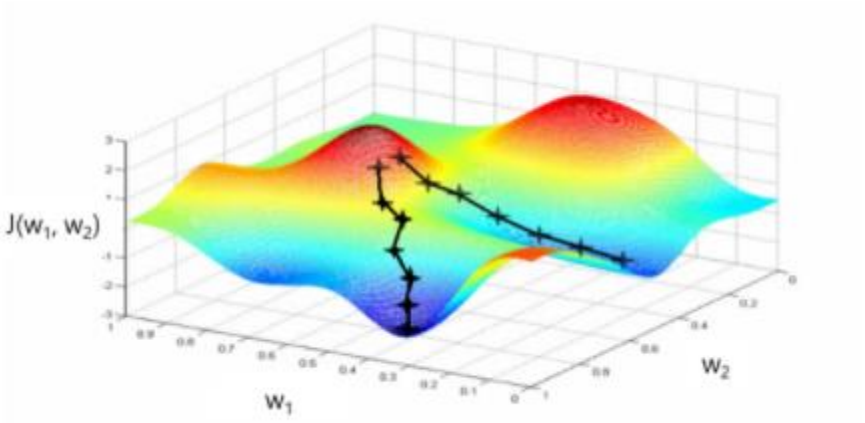
$$-\frac{1}{m} \sum_{i=1}^{m} y^{(i)} \log\left(h_{\omega}\left(x^{(i)}\right)\right) + \left(1 - y^{(i)}\right) \log\left(1 - h_{\omega}\left(x^{(i)}\right)\right)$$

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)

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

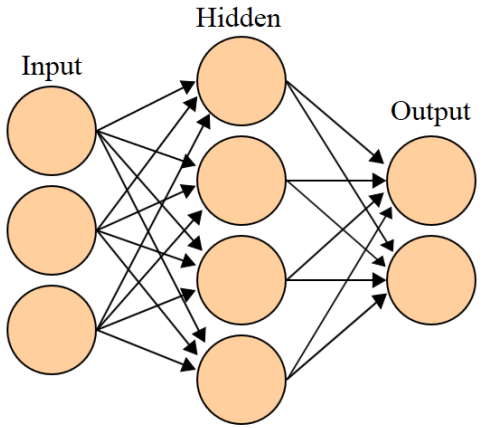


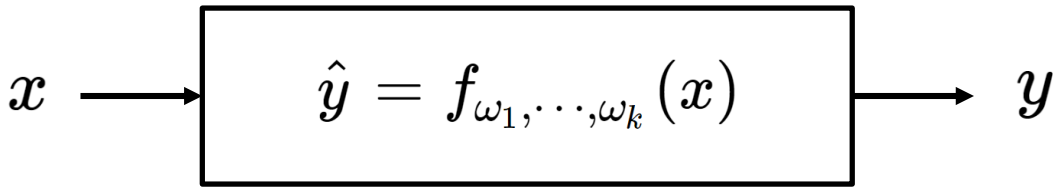


Gradients in ANN

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

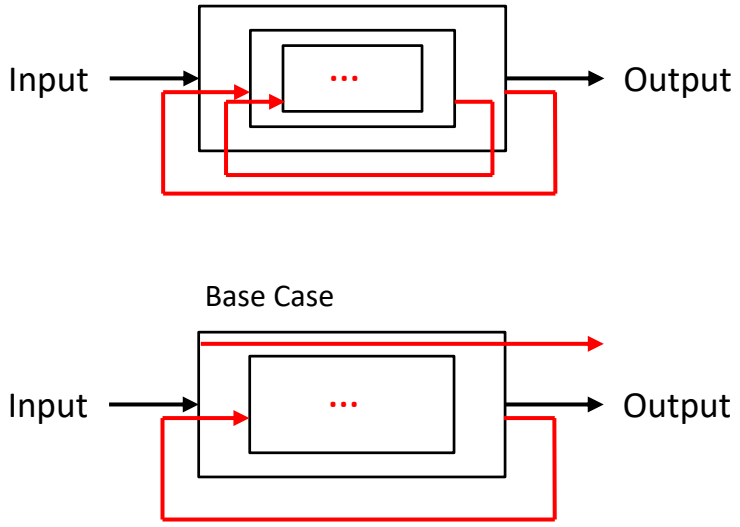
- Structural constraint of NN:
  - Composition of functions
  - Chain rule
  - Dynamic programming





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

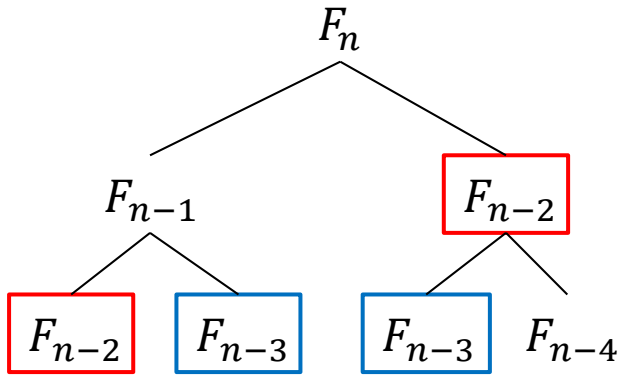
- Fibonacci numbers:

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

Naïve Recursive Algorithm

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

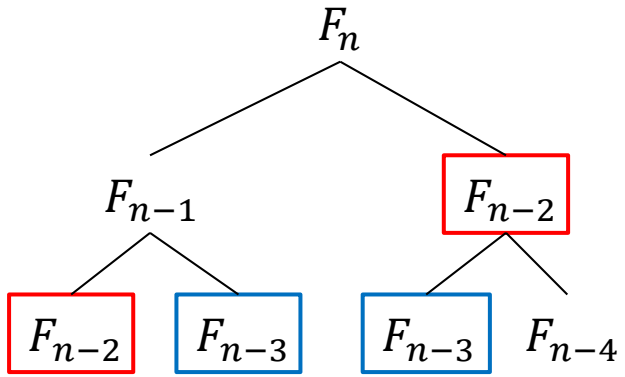
- It works. Is it good?



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



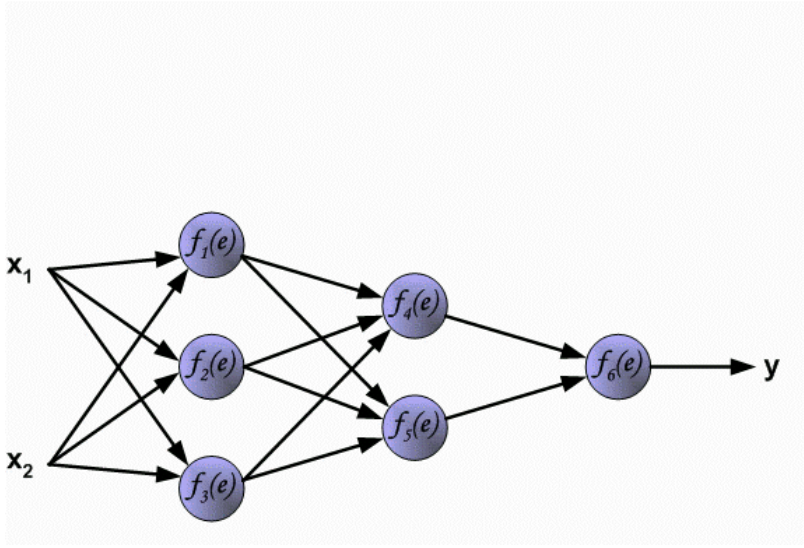
Dynamic Programming Algorithm

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

- DP ≈ recursion + memorization

Training Neural Networks: Backpropagation Learning

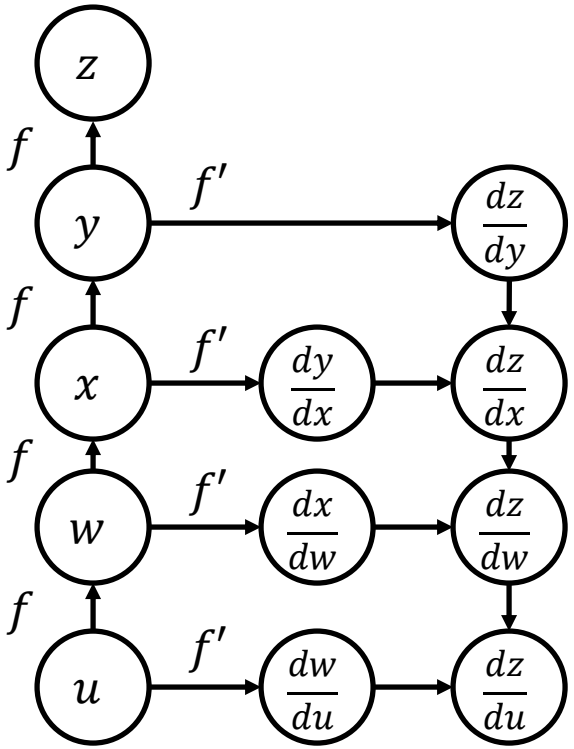
- Forward propagation
  - the initial information propagates up to the hidden units at each layer and finally produces output
- Backpropagation
  - allows the information from the cost to flow backwards through the network in order to compute the gradients



Backpropagation

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

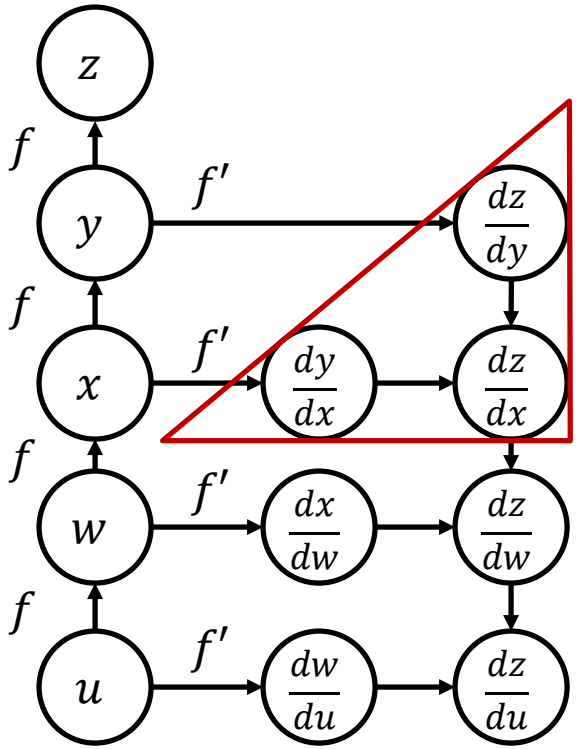
- Backpropagation
  - Update weights recursively



Backpropagation

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

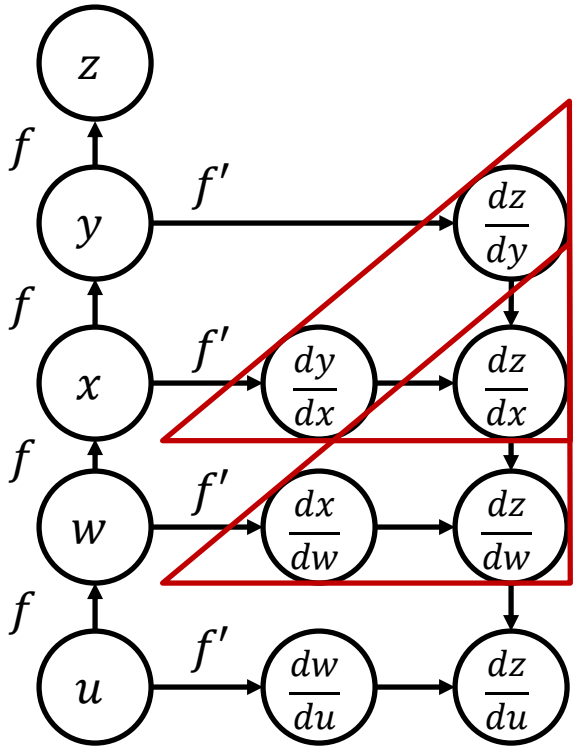
- Backpropagation
  - Update weights recursively



Backpropagation

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

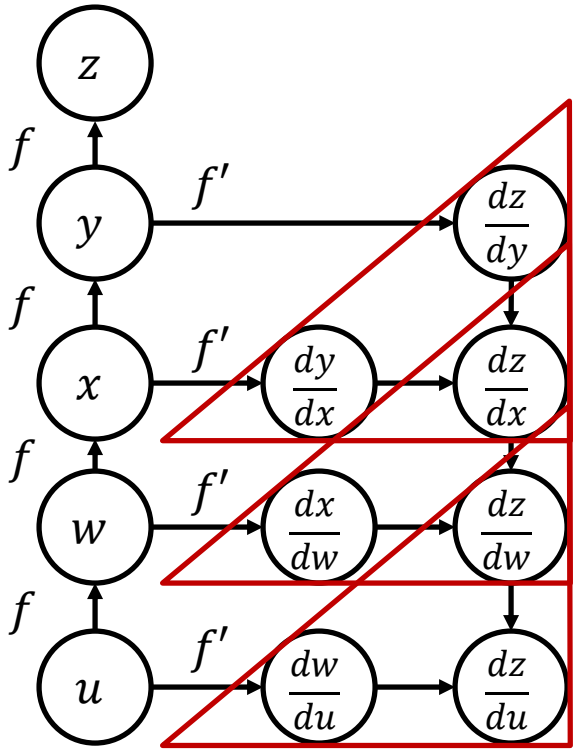
- Backpropagation
  - Update weights recursively



Backpropagation

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

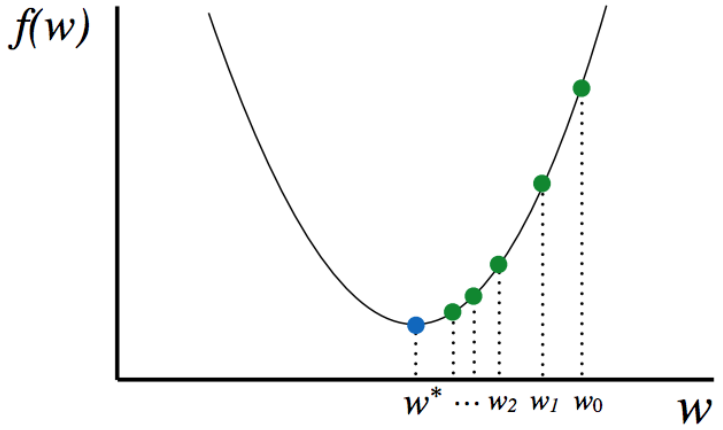
- Backpropagation
  - Update weights recursively with memory



Training Neural Networks

- Optimization procedure

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

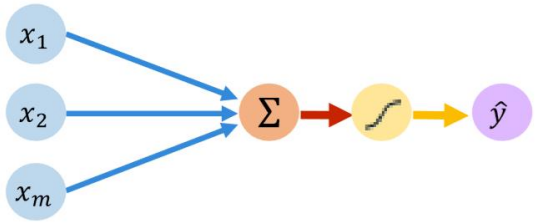


- It is not easy to numerically compute gradients in network in general.
  - The good news: people have already done all the "hard work" of developing numerical solvers (or libraries)
  - There are a wide range of tools → We will use the TensorFlow

Core Foundation Review

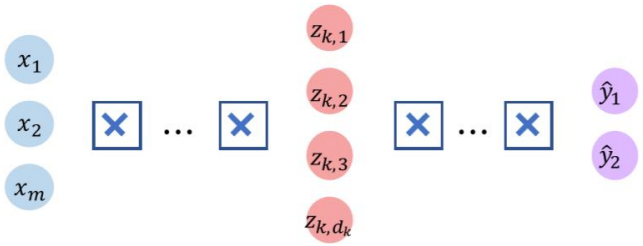
The Perceptron

- Structural building blocks
- Nonlinear activation functions



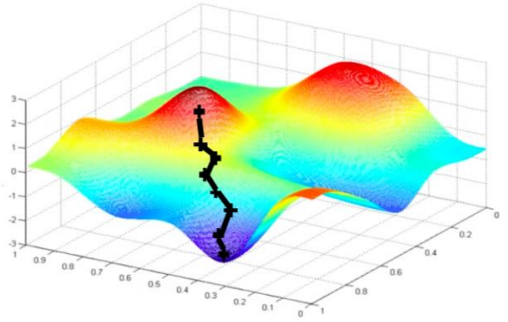
Neural Networks

- 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

(Artificial) Neural Networks with TensorFlow

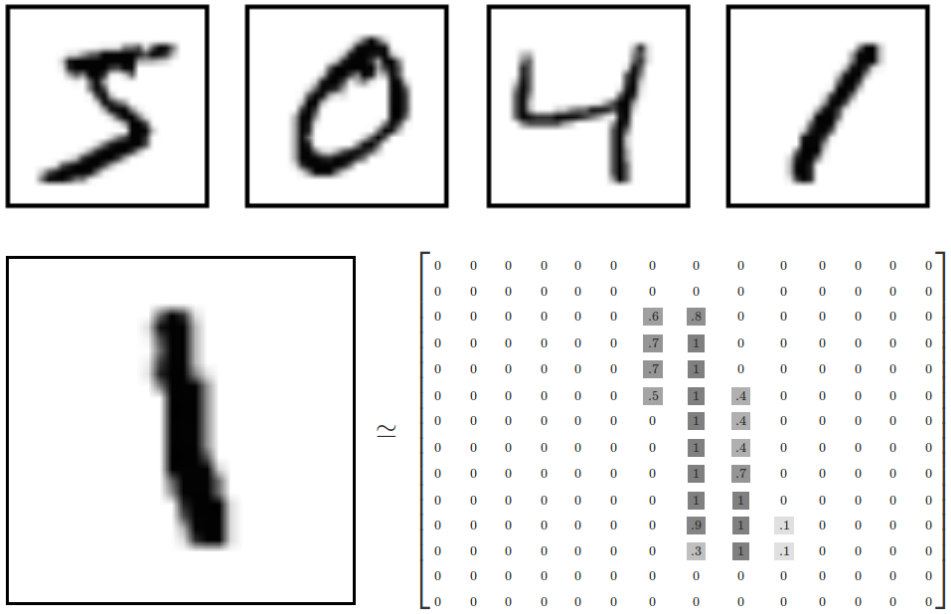
Industrial AI Lab.

Prof. Seungchul Lee

ANN in TensorFlow: MNIST

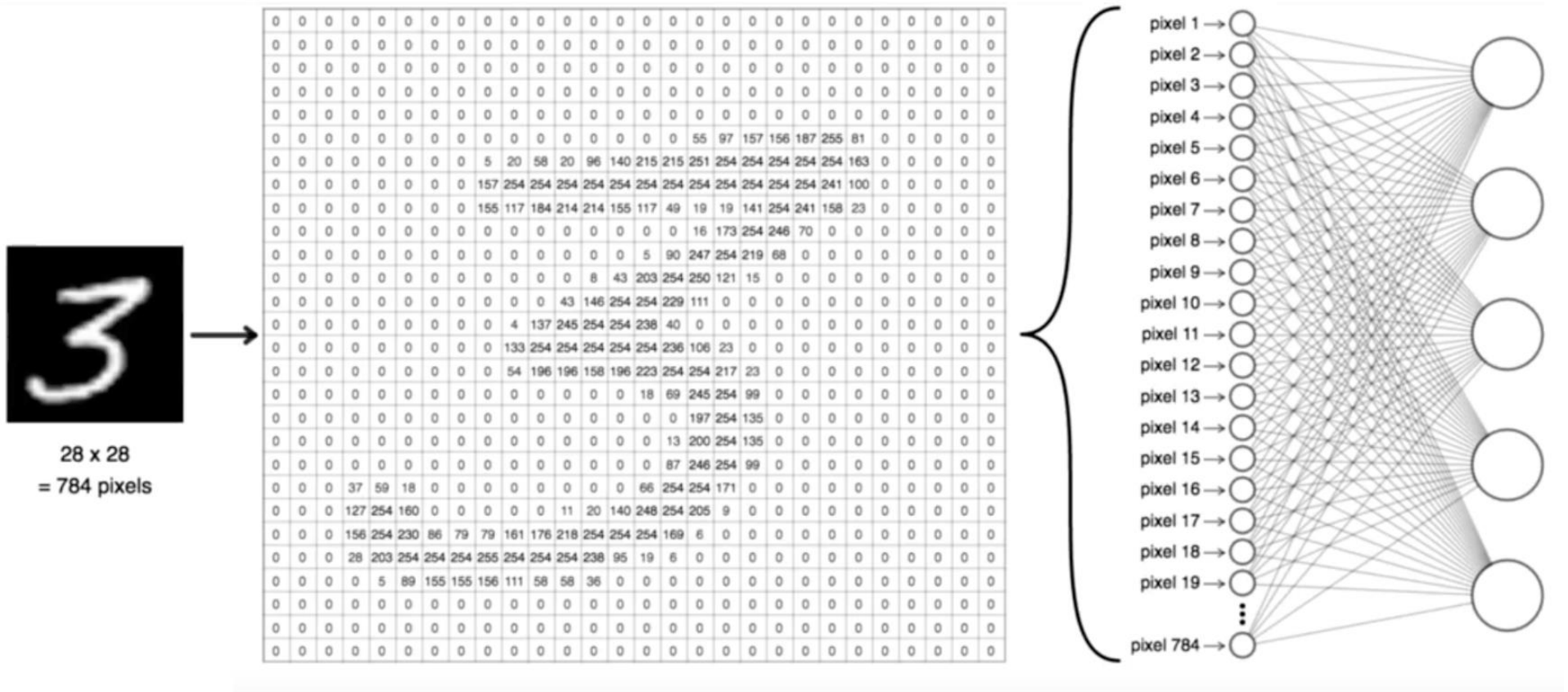
ANN with MNIST

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

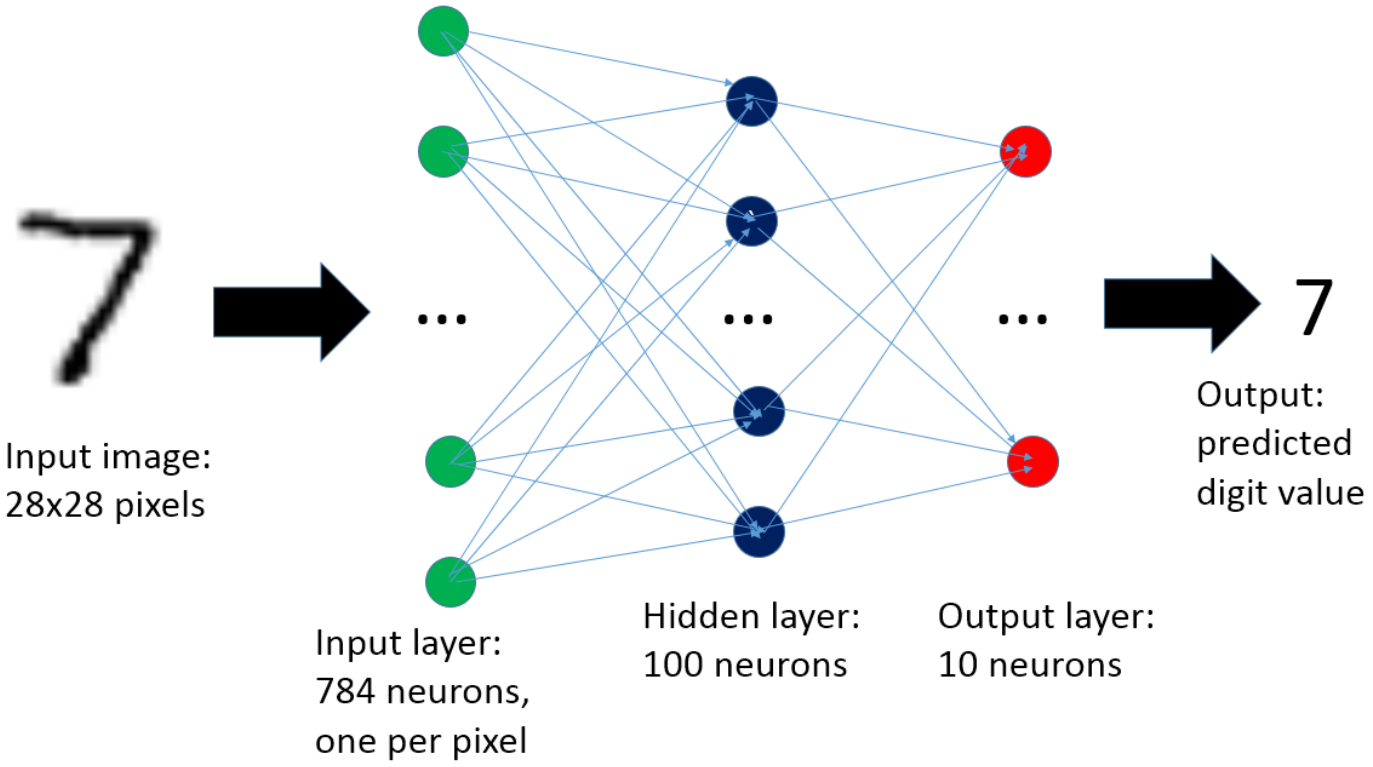


ANN with TensorFlow

- Feed a gray image to ANN



Our Network Model

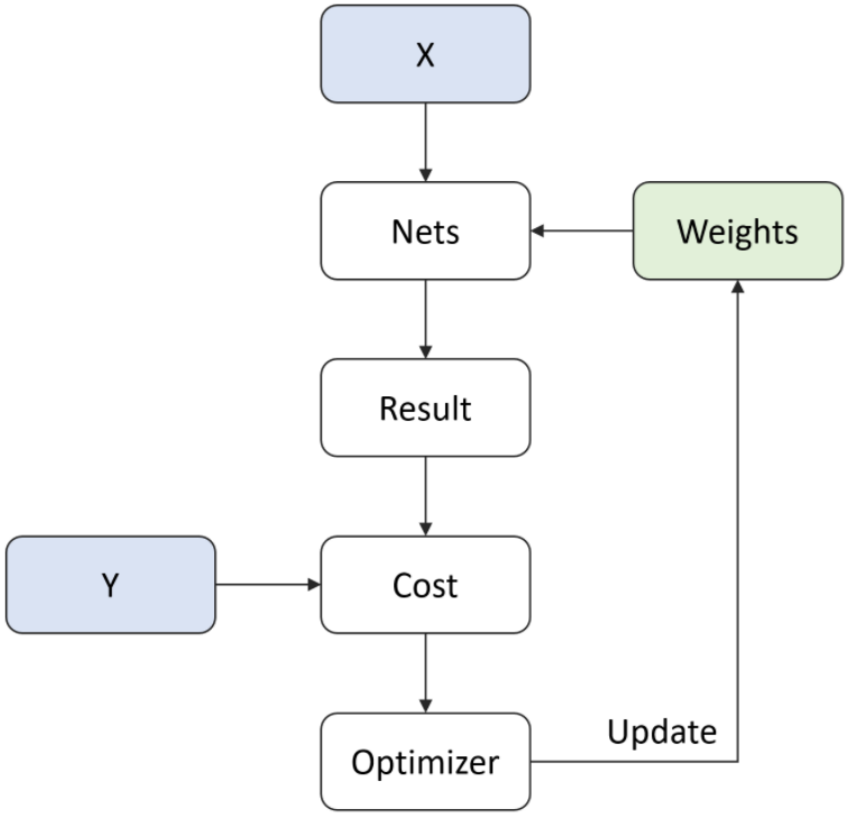


Iterative Optimization

- 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
```

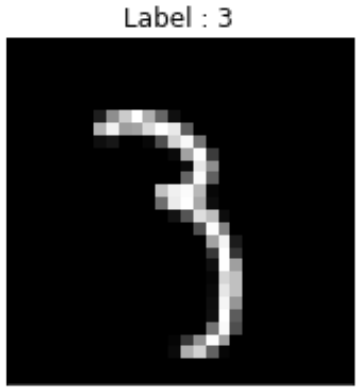
- Load MNIST Data
  - Download MNIST data from TensorFlow tutorial example

```
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

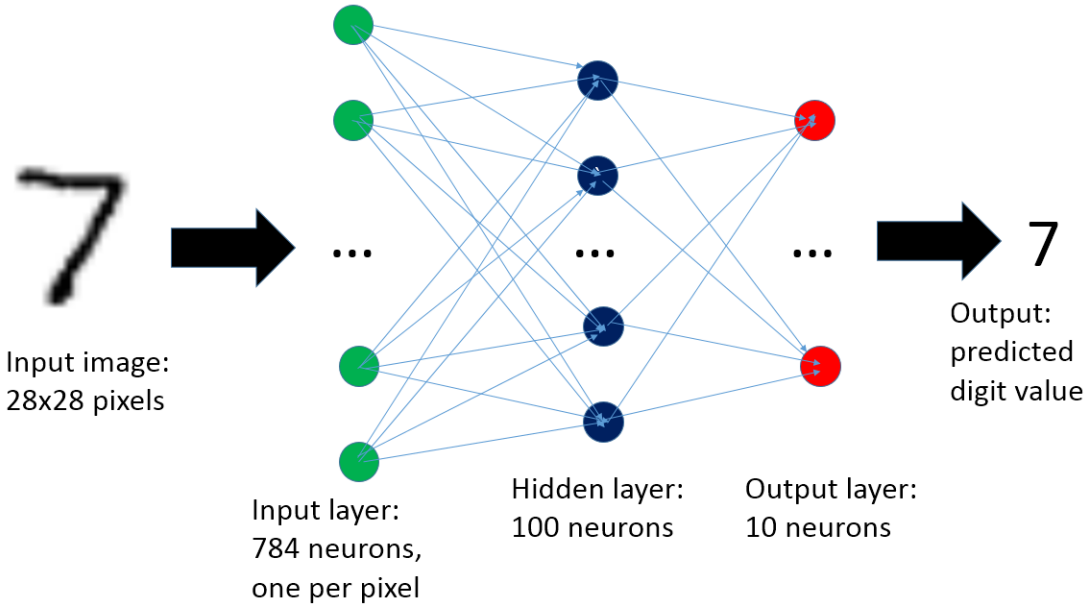
```
print ('Train labels : {}'.format(train_y[0, :]))
```

```
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 $\sigma = 0.1$

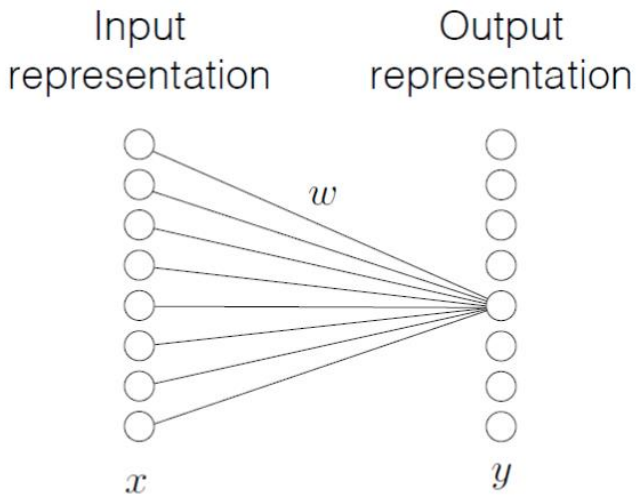
```
weights = {
    'hidden' : tf.Variable(tf.random_normal([n_input, n_hidden], stddev = 0.1)),
    'output' : tf.Variable(tf.random_normal([n_hidden, n_output], stddev = 0.1))
}

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



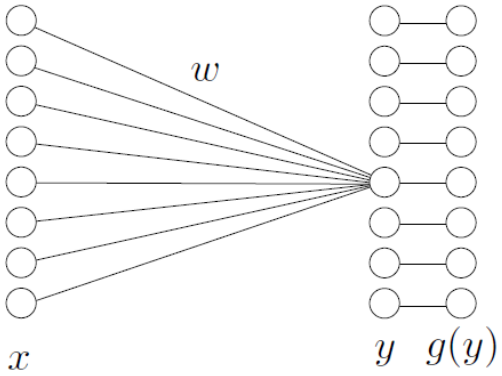
$$y_j = \left(\sum_i \omega_{ij} x_i \right) + b_j$$

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

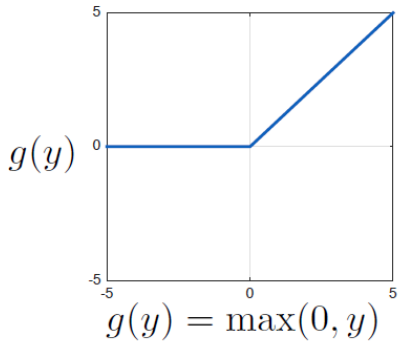
```python
# Define Network
def build_model(x, weights, biases):

    # 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



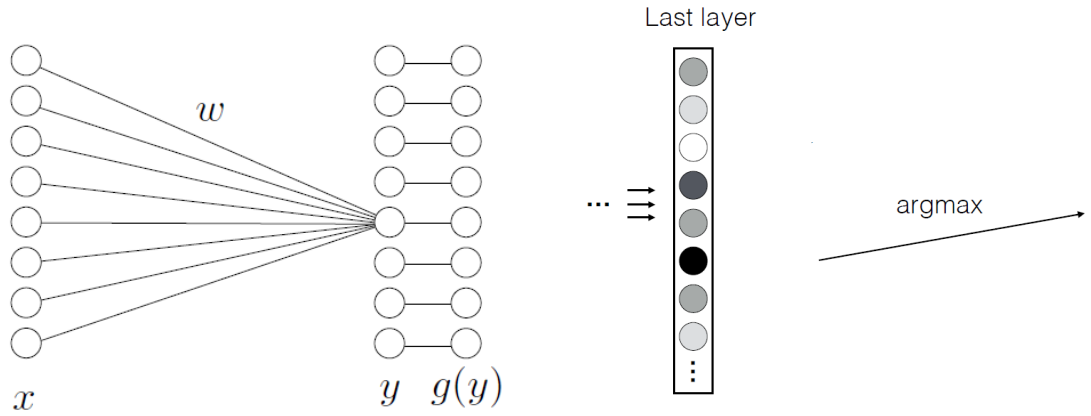
Rectified linear unit (ReLU)




```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

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

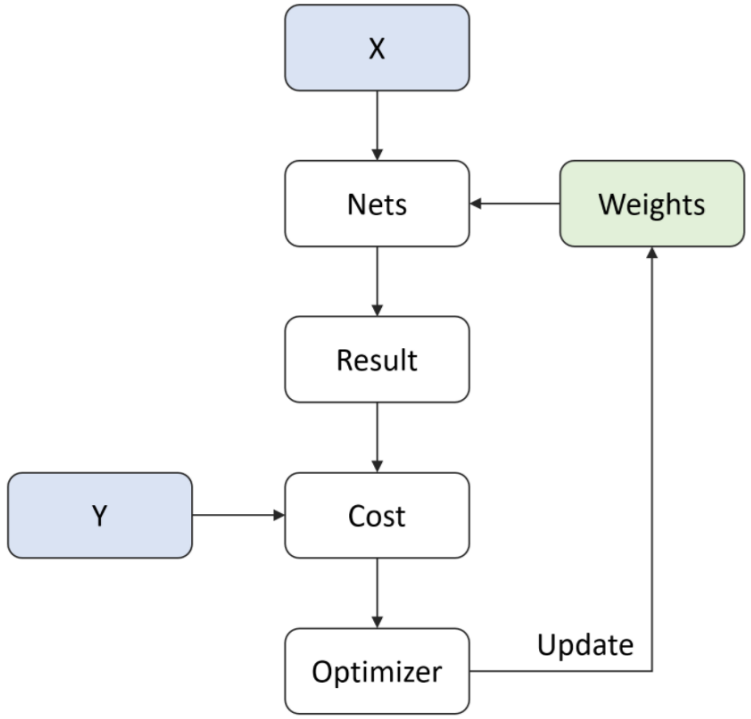
- Optimizer
  - AdamOptimizer: the most popular optimizer

```
# 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

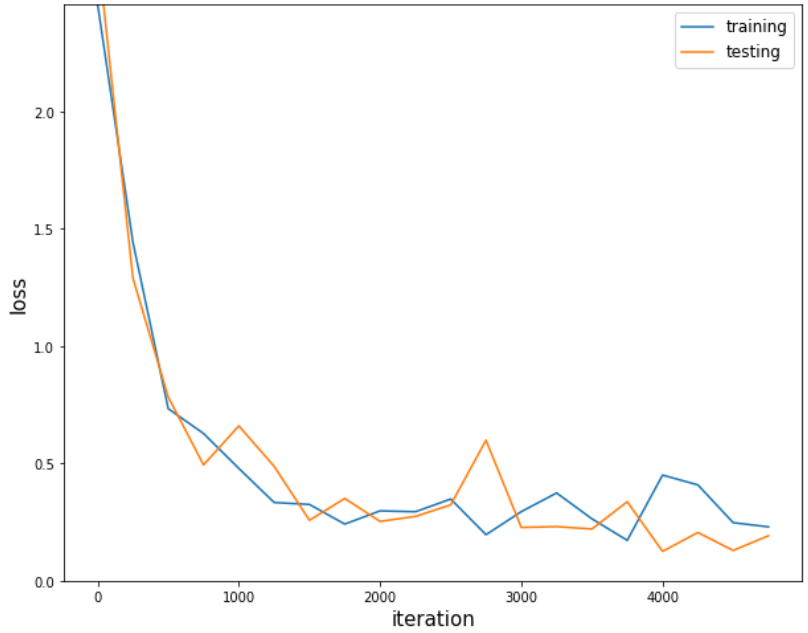
- Define parameters for training ANN
  - n_batch: batch size for mini-batch gradient descent
  - n_iter: the number of iteration steps
  - n_prt: check loss for every n_prt iteration

```
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()
```

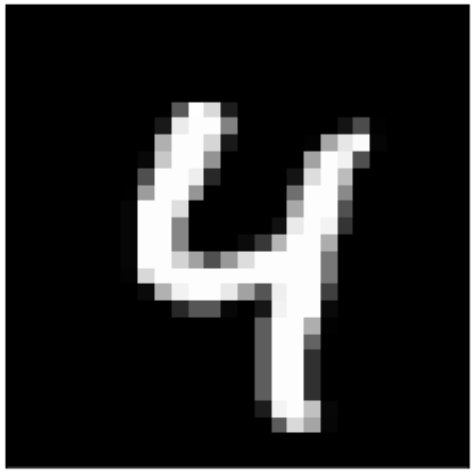


Test or Evaluation

```
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]
```