

# (Artificial) Neural Networks in TensorFlow

By Prof. Seungchul Lee  
Industrial AI Lab  
<http://isystems.unist.ac.kr/>  
POSTECH

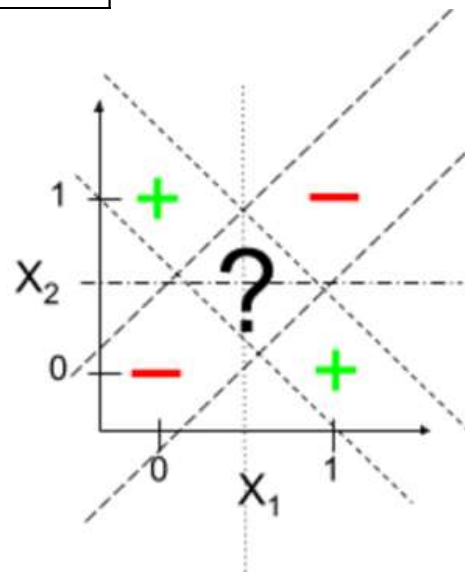
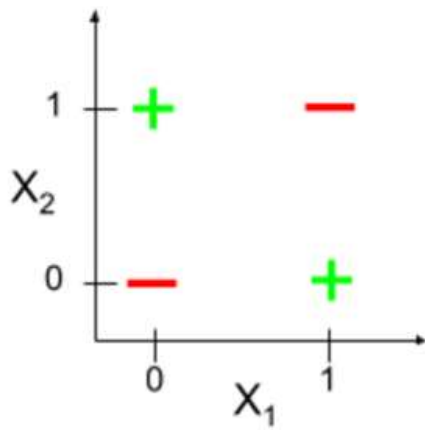
## Table of Contents

- I. 1. Artificial Neural Networks: Perceptron
  - I. 1.1. Perceptron for  $h(\theta)$  or  $h(\omega)$
  - II. 1.2. Artificial Neural Networks: MLP
- II. 2. Recall Supervised Learning Setup
- III. 3. Training Neural Networks
  - I. 3.1. Optimization
  - II. 3.2. Loss Function
  - III. 3.3. Learning
  - IV. 3.4. Deep Learning Libraries
- IV. 4. TensorFlow
  - I. 4.1. Import Library
  - II. 4.2. Load MNIST Data
  - III. 4.3. Build a Model
  - IV. 4.4. Define the ANN's Shape
  - V. 4.5. Define Weights, Biases and Network
  - VI. 4.6. Define Cost, Initializer and Optimizer
  - VII. 4.7. Summary of Model
  - VIII. 4.8. Define Configuration
  - IX. 4.9. Optimization
  - X. 4.10. Test

## XOR Problem

- Minsky-Papert Controversy on XOR
  - not linearly separable
  - limitation of perceptron

$x_1$	$x_2$	$x_1 \text{ XOR } x_2$
0	0	0
0	1	1
1	0	1
1	1	0

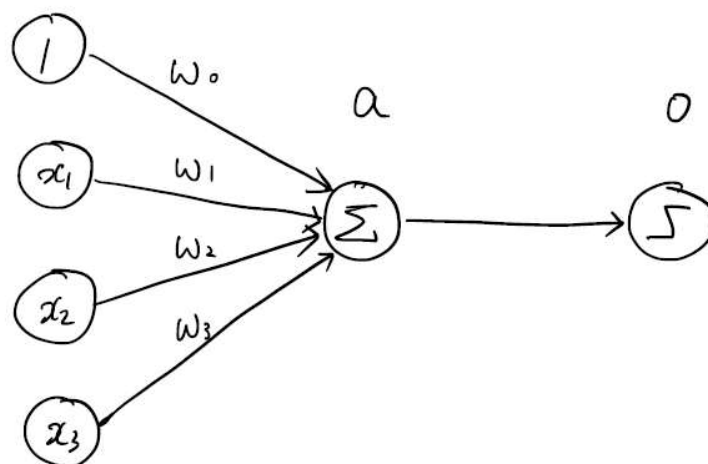


# 1. Artificial Neural Networks: Perceptron

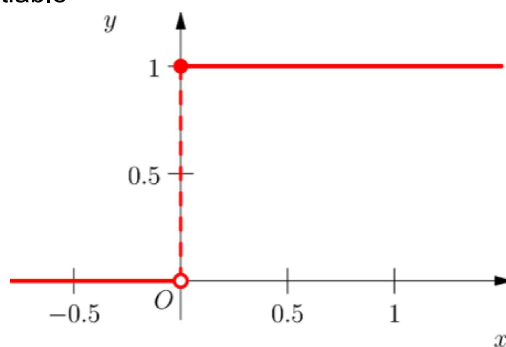
## 1.1. Perceptron for $h(\theta)$ or $h(\omega)$

- Neurons compute the weighted sum of their inputs
- A neuron is activated or fired when the sum  $a$  is positive

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



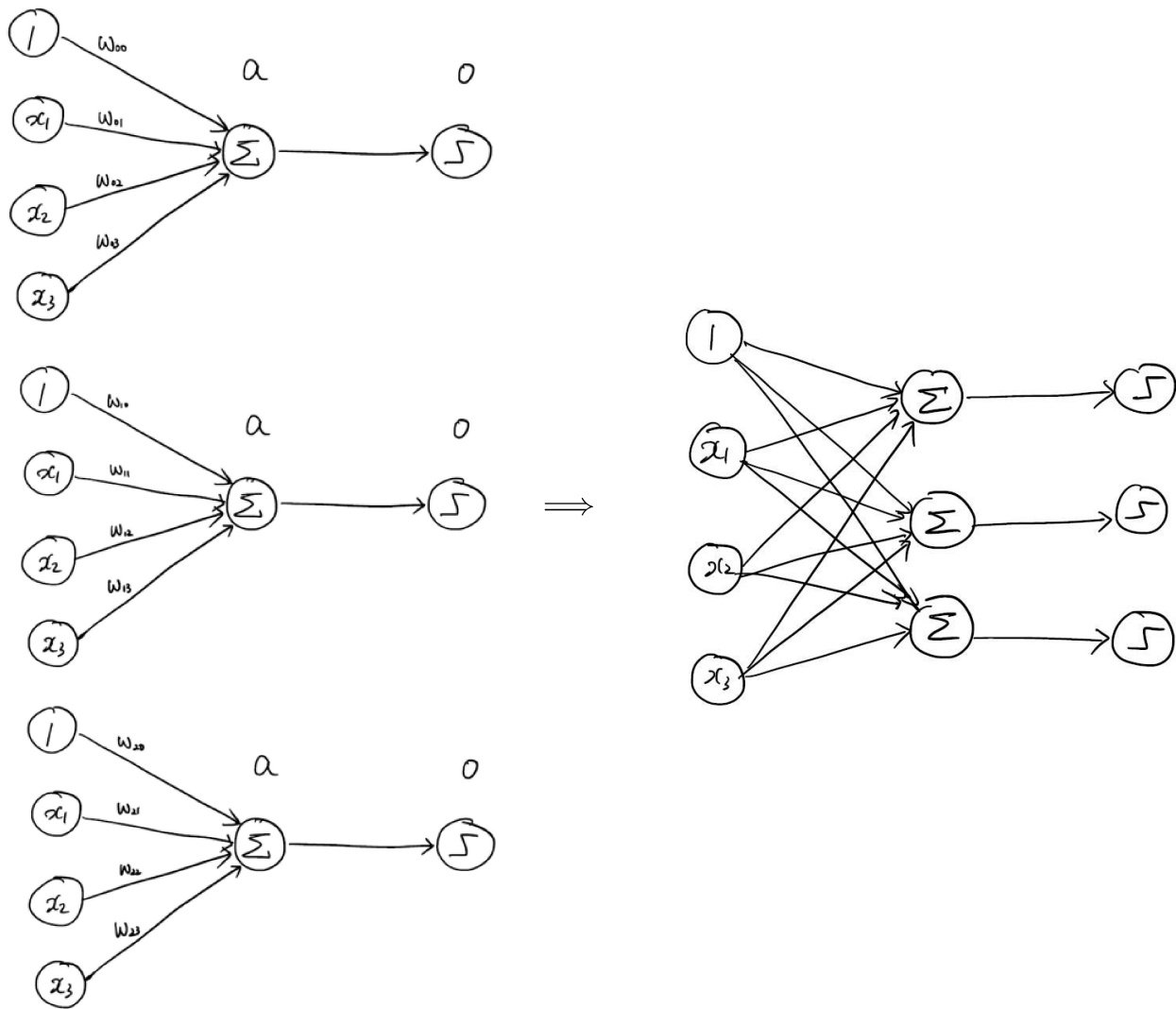
- A step function is not differentiable



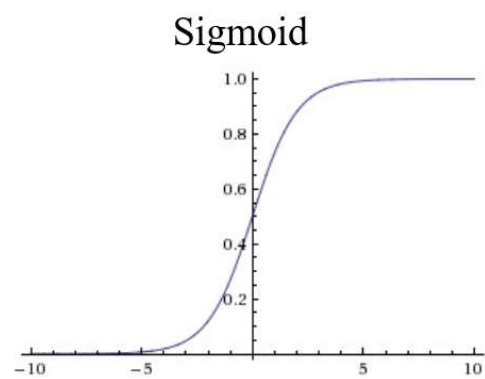
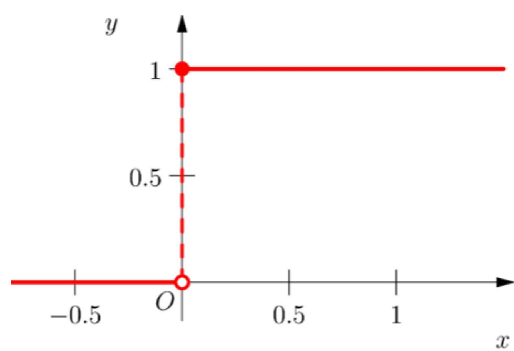
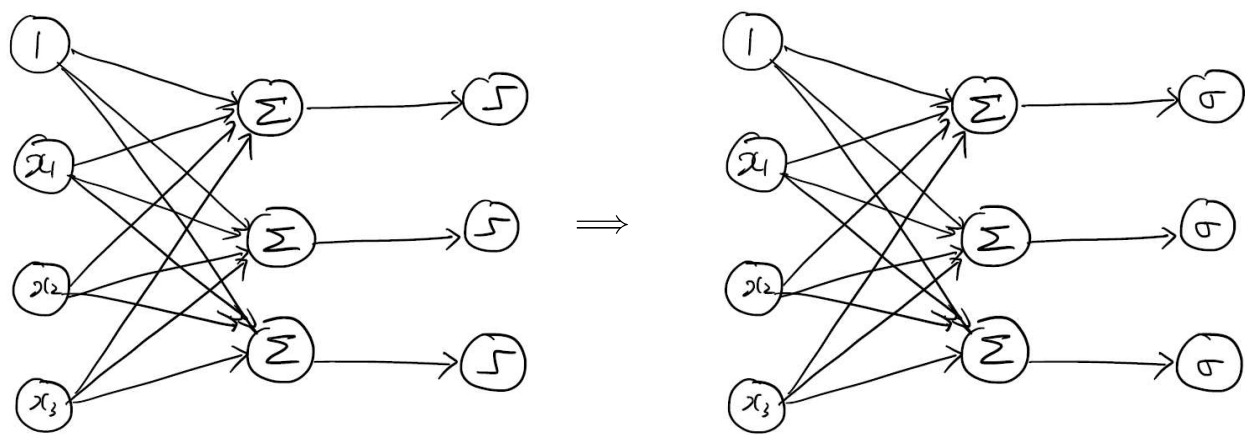
- One layer is often not enough

# 1.2. Artificial Neural Networks: MLP

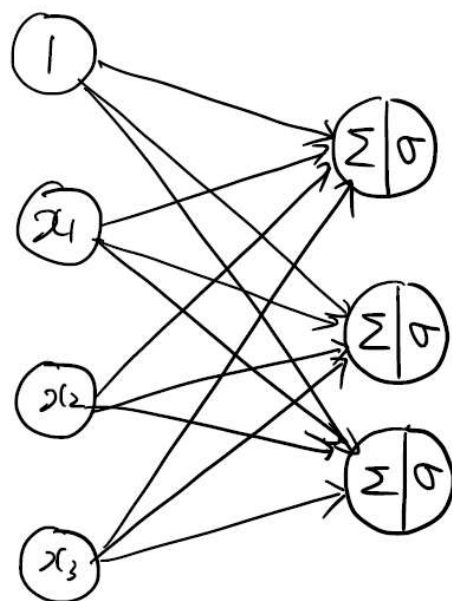
Multi-layer Perceptron (MLP) = Artificial Neural Networks (ANN) multi-neurons



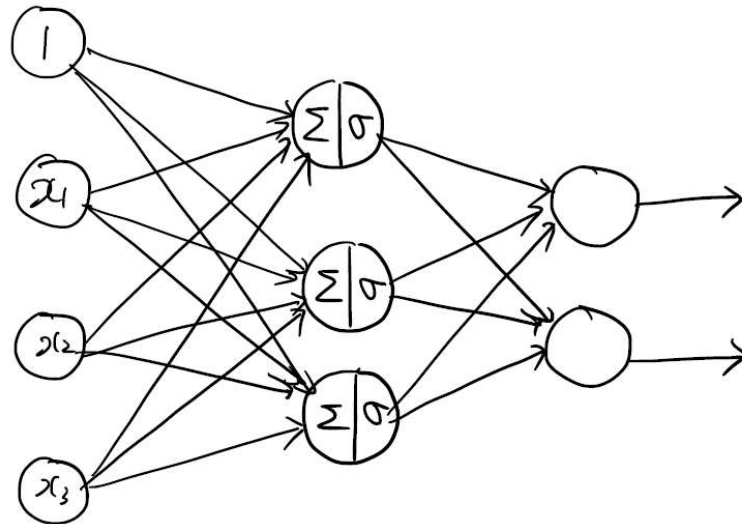
differentiable activation function



in a compact representation



## multi-layer perceptron

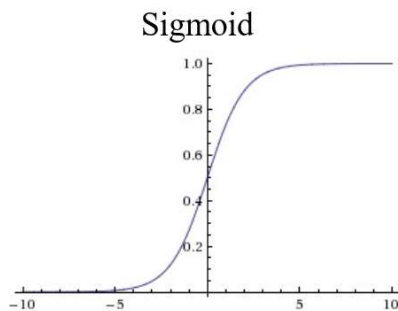


## Transformation

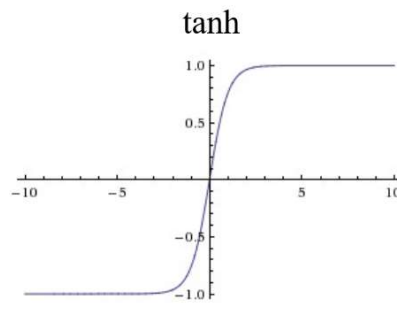
- Affine (or linear) transformation and nonlinear activation layer (notations are mixed:  
 $g = \sigma, \omega = \theta, \omega_0 = b$ )

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

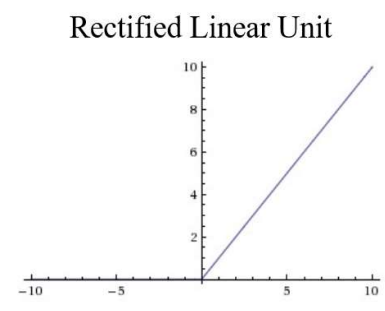
- Nonlinear activation functions ( $g = \sigma$ )



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



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



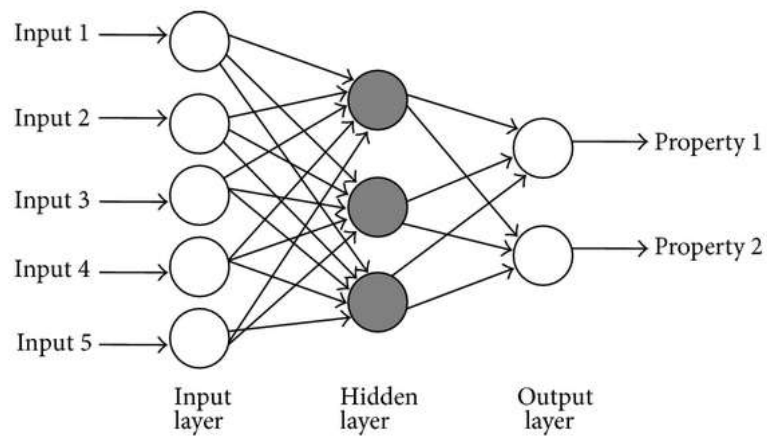
$$g(x) = \max(0, x)$$

## Structure

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

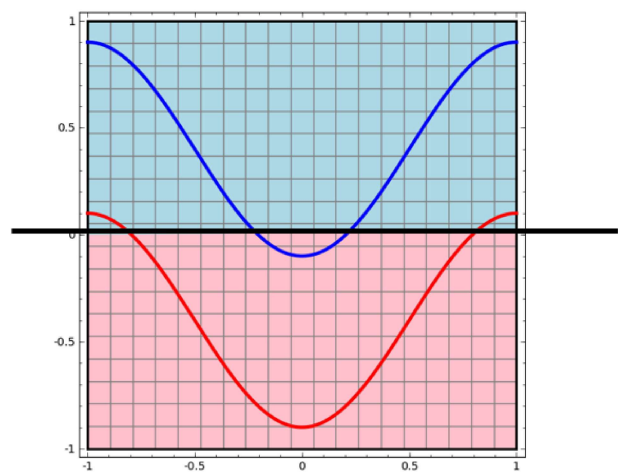
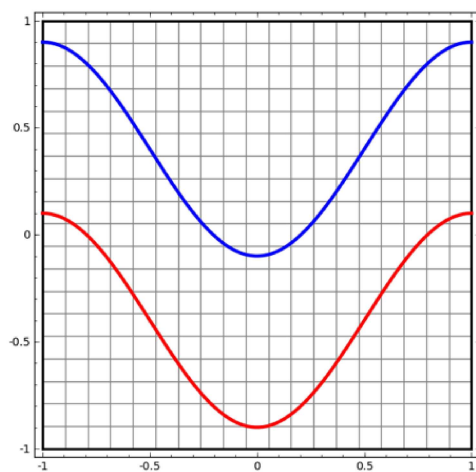
⇒ perceptrons with many layers and units

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



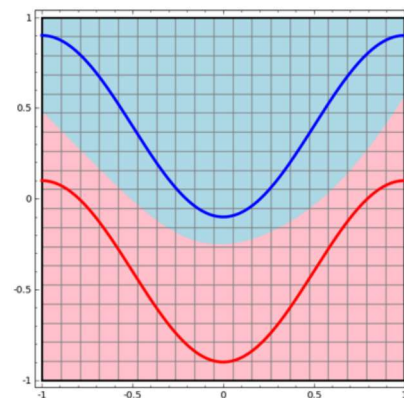
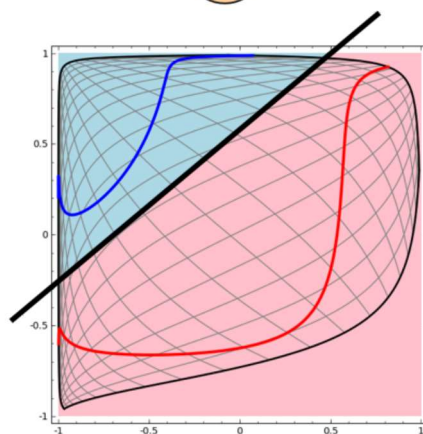
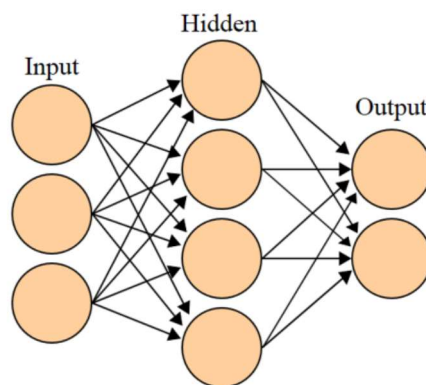
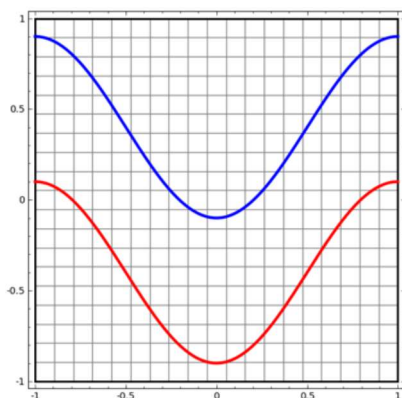
## Linear Classifier

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



## Neural Networks

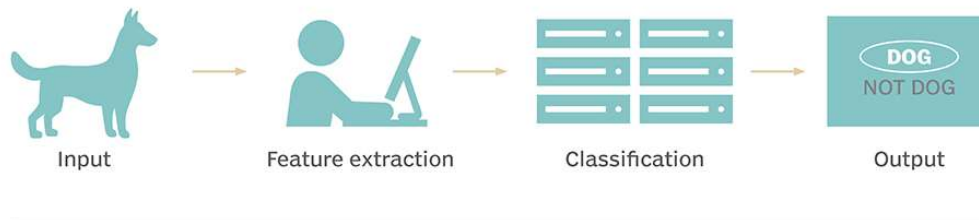
- The hidden layer learns a representation so that the data is linearly separable





## 2. Recall Supervised Learning Setup

### TRADITIONAL MACHINE LEARNING



### DEEP LEARNING



## 3. Training Neural Networks

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

### 3.1. Optimization

#### 3 key components

1. objective function  $f(\cdot)$
2. decision variable or unknown  $\theta$
3. constraints  $g(\cdot)$

#### In mathematical expression

$$\begin{aligned} & \min_{\theta} f(\theta) \\ & \text{subject to } g_i(\theta) \leq 0, \quad i = 1, \dots, m \end{aligned}$$

## 3.2. Loss Function

- Measures error between target values and predictions

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

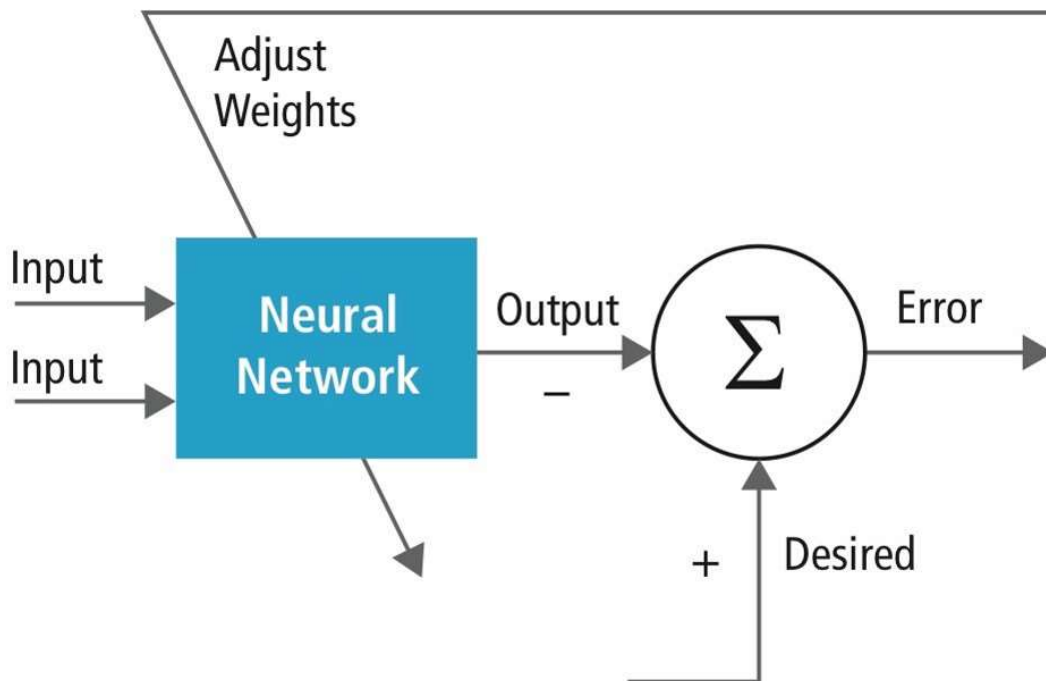
- Example

- Squared loss (for regression):

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

- Cross entropy (for classification):

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

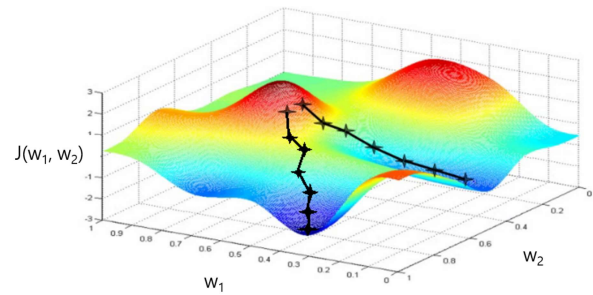
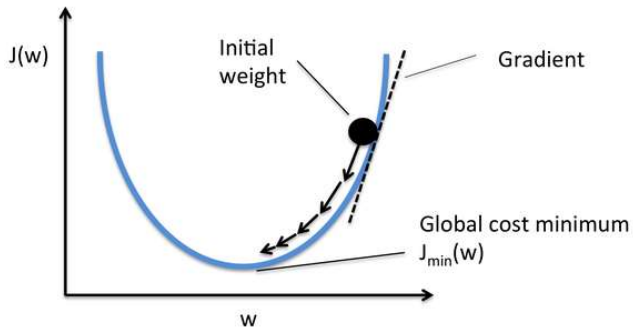


## 3.3. Learning

## (Stochastic) Gradient Descent

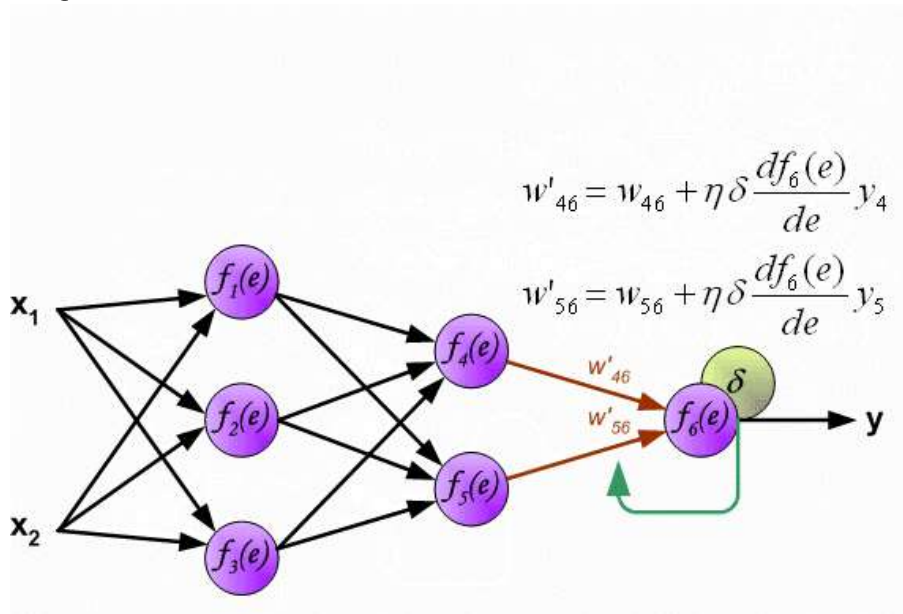
- Negative gradients points directly downhill of the cost function
- We can decrease the cost by moving in the direction of the negative gradient ( $\alpha$  is a learning rate)

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



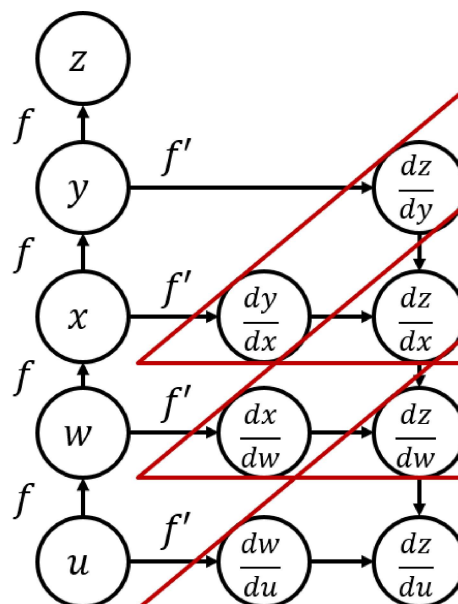
## Backpropagation

- 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



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

- Backpropagation
  - Update weights recursively



## Optimization procedure

Start at a random point

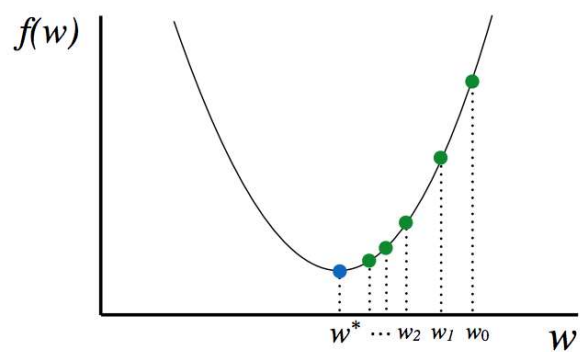
**Repeat**

Determine a descent direction

Choose a step size

Update

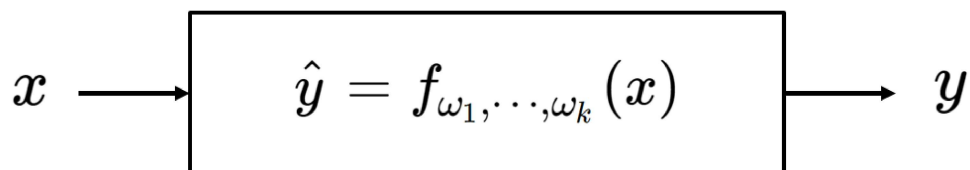
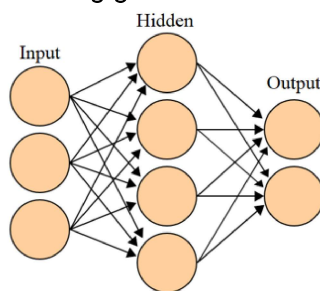
**Until** stopping criterion is satisfied



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

## Summary

- Learning weights and biases from data using gradient descent



## 3.4. Deep Learning Libraries

### Caffe

# Caffe

- Platform: Linux, Mac OS, Windows
- Written in: C++
- Interface: Python, MATLAB

### Theano

# theano

- Platform: Cross-platform
- Written in: Python
- Interface: Python

### Tensorflow



- Platform: Linux, Mac OS, Windows
- Written in: C++, Python
- Interface: Python, C/C++, Java, Go, R

## 4. TensorFlow

- TensorFlow (<https://www.tensorflow.org>) is an open-source software library for deep learning.

### Computational Graph

- `tf.constant`
- `tf.Variable`
- `tf.placeholder`

In [1]:

```
import tensorflow as tf

a = tf.constant([1, 2, 3])
b = tf.constant([4, 5, 6])

A = a + b
B = a * b
```

In [2]:

A

Out[2]:

<tf.Tensor 'add:0' shape=(3,) dtype=int32>

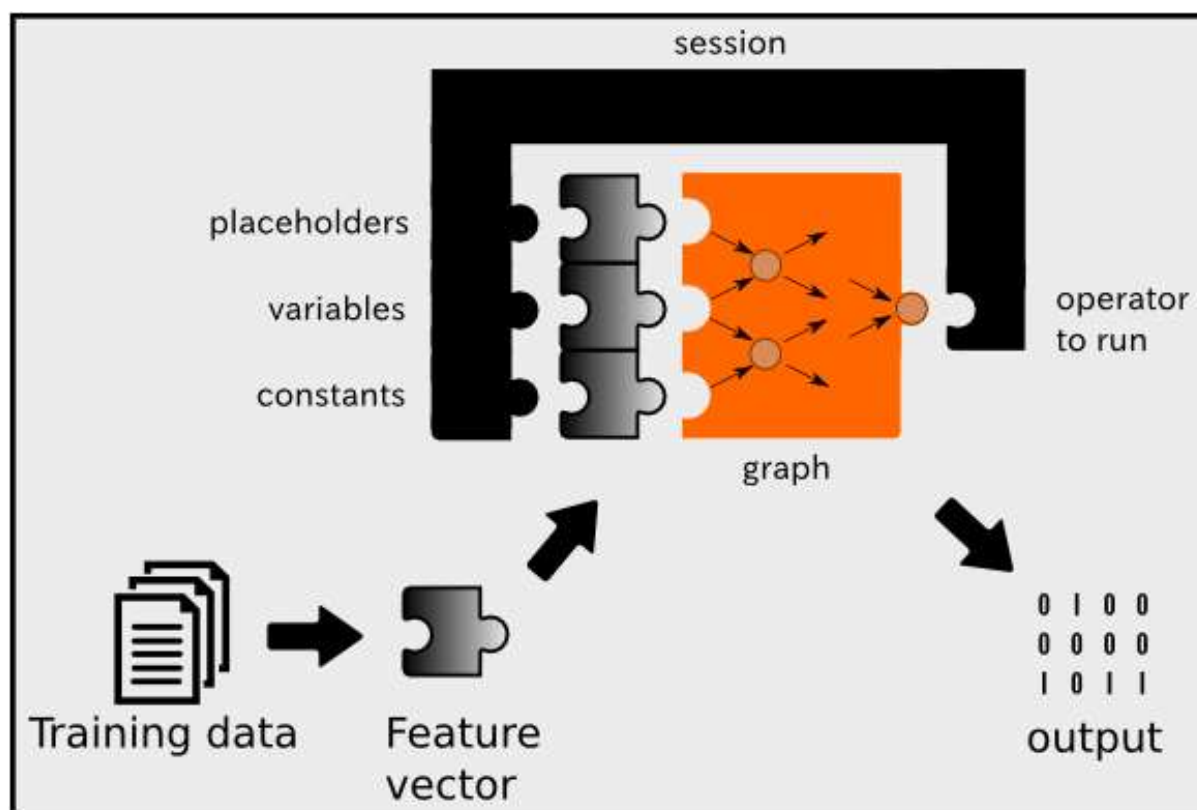
In [3]:

B

Out[3]:

<tf.Tensor 'mul:0' shape=(3,) dtype=int32>

To run any of the three defined operations, we need to create a session for that graph. The session will also allocate memory to store the current value of the variable.



In [4]:

```
sess = tf.Session()  
sess.run(A)
```

Out[4]:

```
array([5, 7, 9], dtype=int32)
```

In [5]:

```
sess.run(B)
```

Out[5]:

```
array([ 4, 10, 18], dtype=int32)
```

`tf.Variable` is regarded as the decision variable in optimization. We should initialize variables to use `tf.Variable`.

In [6]:

```
w = tf.Variable([1, 1])
```

In [7]:

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

In [8]:

```
sess.run(w)
```

Out[8]:

```
array([1, 1], dtype=int32)
```

The value of `tf.placeholder` must be fed using the `feed_dict` optional argument to `Session.run()`.

In [9]:

```
x = tf.placeholder(tf.float32, [2, 2])
```

In [10]:

```
sess.run(x, feed_dict={x : [[1,2],[3,4]]})
```

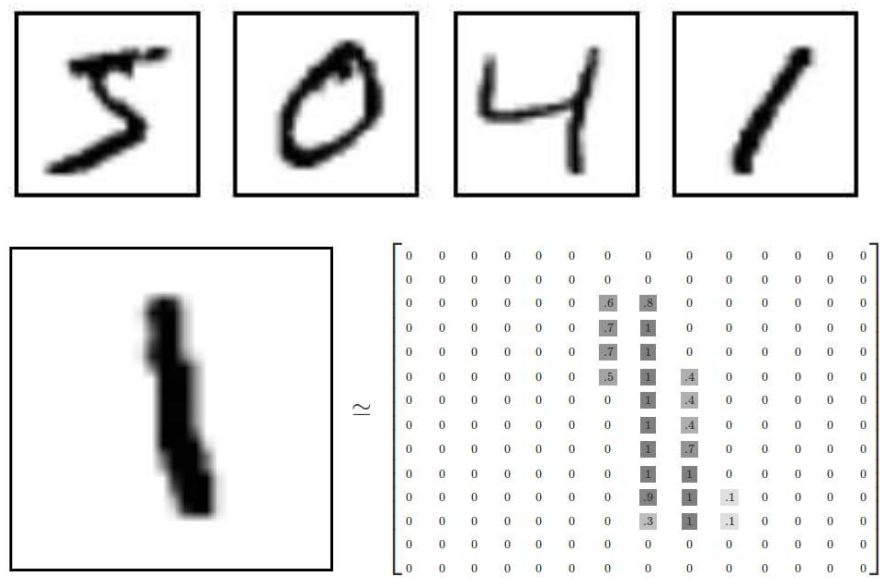
Out[10]:

```
array([[ 1.,  2.],  
       [ 3.,  4.]], dtype=float32)
```

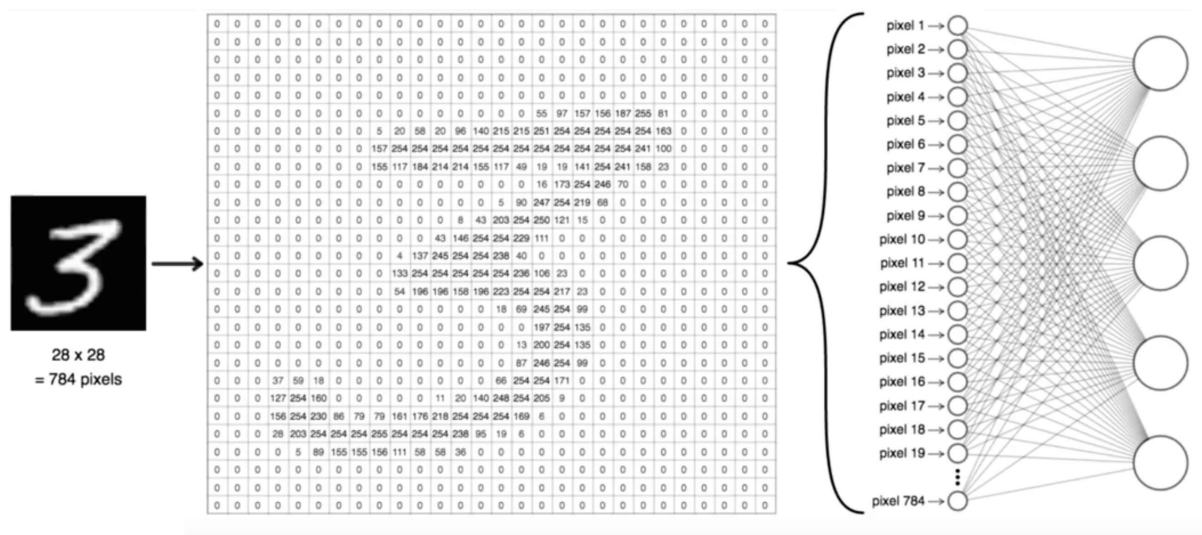


ANN with TensorFlow

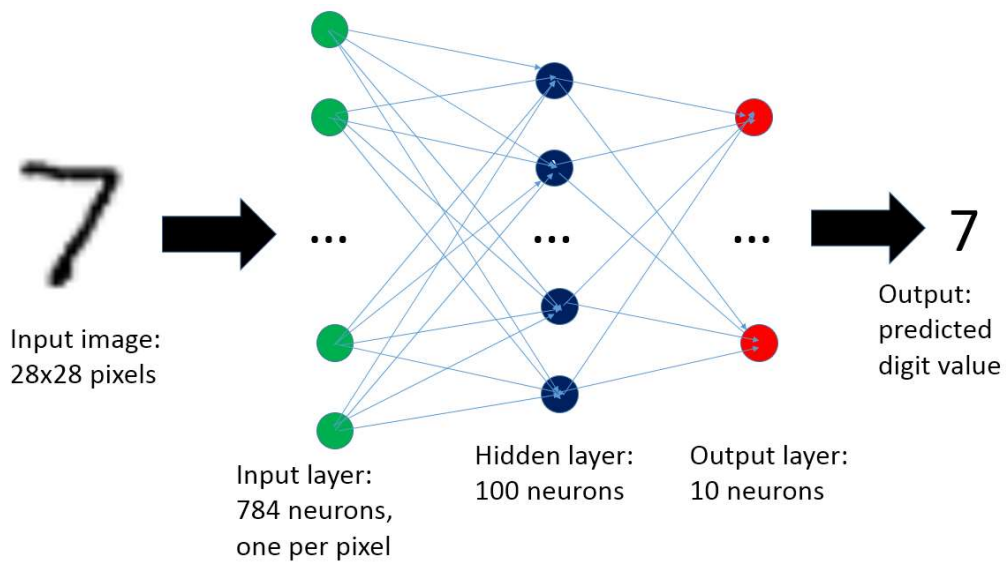
- MNIST (Mixed National Institute of Standards and Technology database) database
  - Handwritten digit database
  - $28 \times 28$  gray scaled image
  - Flattened matrix into a vector of  $28 \times 28 = 784$



- feed a gray image to ANN



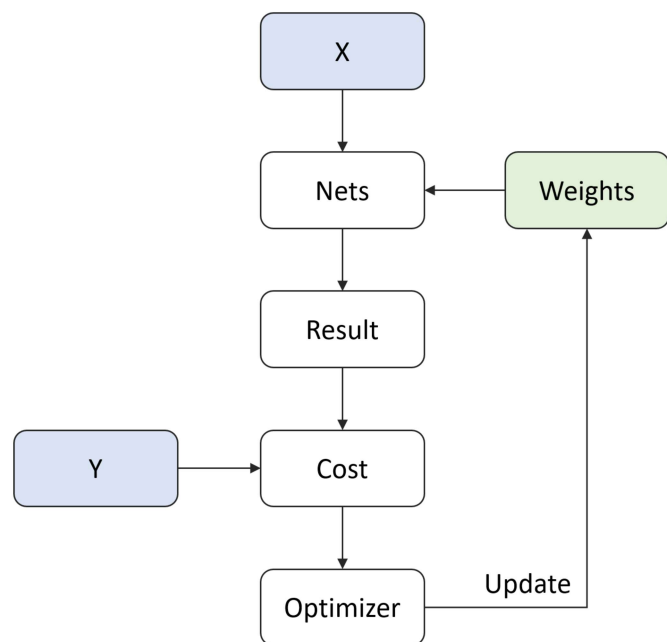
- our network model



$$\min_{\theta} f(\theta)$$

subject to  $g_i(\theta) \leq 0$

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



## 4.1. Import Library

In [11]:

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

## 4.2. Load MNIST Data

- Download MNIST data from tensorflow tutorial example

In [12]:

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

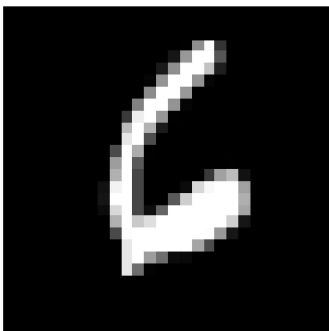
```
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
```

In [13]:

```
train_x, train_y = mnist.train.next_batch(10)
img = train_x[3,:].reshape(28,28)

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

Label : 6



One hot encoding

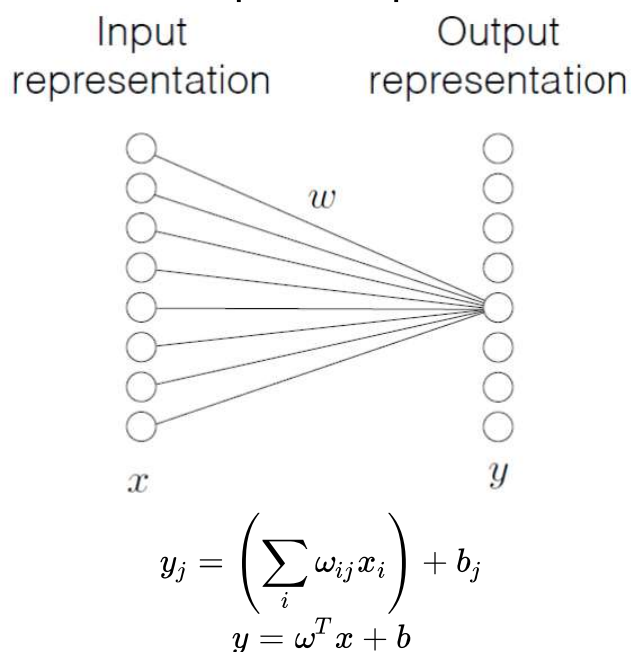
In [14]:

```
print ('Train labels : {}'.format(train_y[3, :]))
```

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

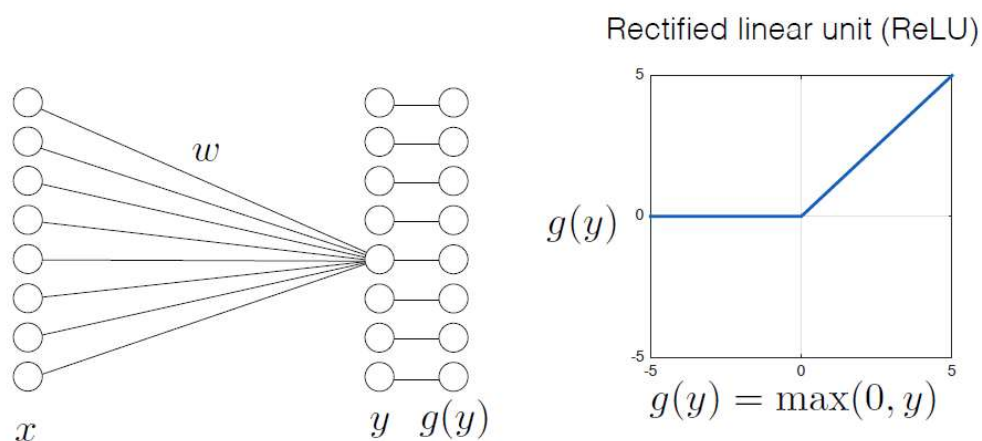
## 4.3. Build a Model

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



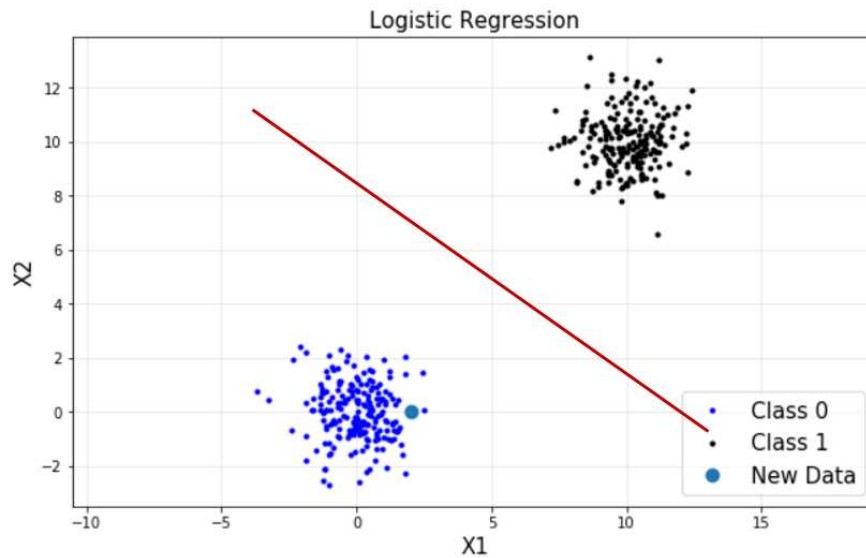
```
# hidden1 = tf.matmul(x, weights['hidden1']) + biases['hidden1']  
hidden1 = tf.add(tf.matmul(x, weights['hidden1']), biases['hidden1'])
```

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



```
hidden1 = tf.nn.relu(hidden1)
```

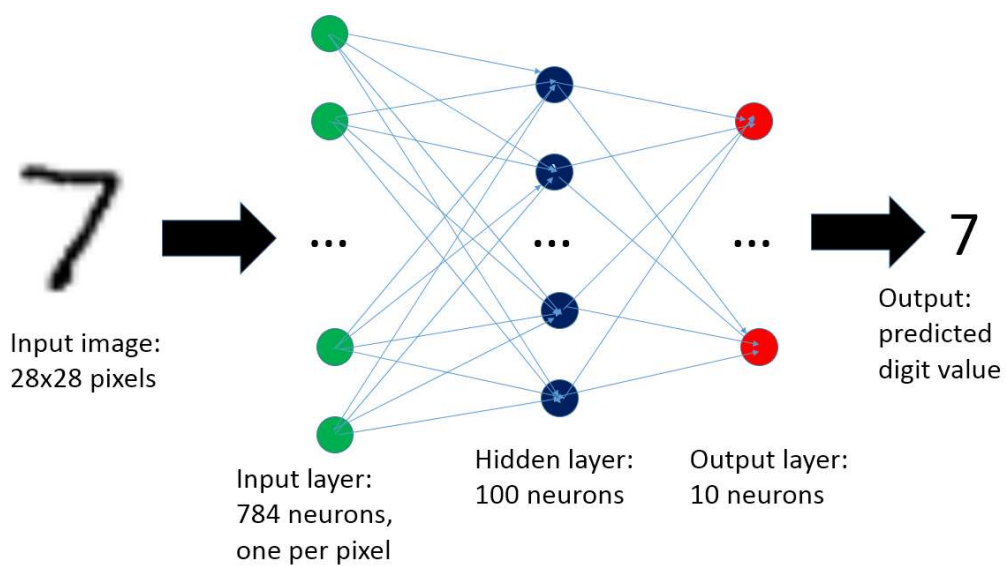
Third, predict values with an affine transformation



```
# output = tf.matmul(hidden1, weights['output']) + biases['output']  
output = tf.add(tf.matmul(hidden1, weights['output']), biases['output'])
```

## 4.4. Define the ANN's Shape

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



In [15]:

```
n_input = 28*28
n_hidden1 = 100
n_output = 10
```

## 4.5. Define Weights, Biases and Network

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

In [16]:

```
weights = {
    'hidden1' : tf.Variable(tf.random_normal([n_input, n_hidden1], stddev = 0.1)),
    'output' : tf.Variable(tf.random_normal([n_hidden1, n_output], stddev = 0.1)),
}

biases = {
    'hidden1' : tf.Variable(tf.random_normal([n_hidden1], stddev = 0.1)),
    'output' : tf.Variable(tf.random_normal([n_output], stddev = 0.1)),
}

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

In [17]:

```
# Define Network
def build_model(x, weights, biases):
    # first hidden layer
    hidden1 = tf.add(tf.matmul(x, weights['hidden1']), biases['hidden1'])
    # non linear activate function
    hidden1 = tf.nn.relu(hidden1)

    # Output layer with linear activation
    output = tf.add(tf.matmul(hidden1, weights['output']), biases['output'])
    return output
```

## 4.6. Define Cost, Initializer and Optimizer

### Loss

- Classification: Cross entropy
  - Equivalent to apply logistic regression

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

### Initializer

- Initialize all the empty variables

### Optimizer

- AdamOptimizer: the most popular optimizer

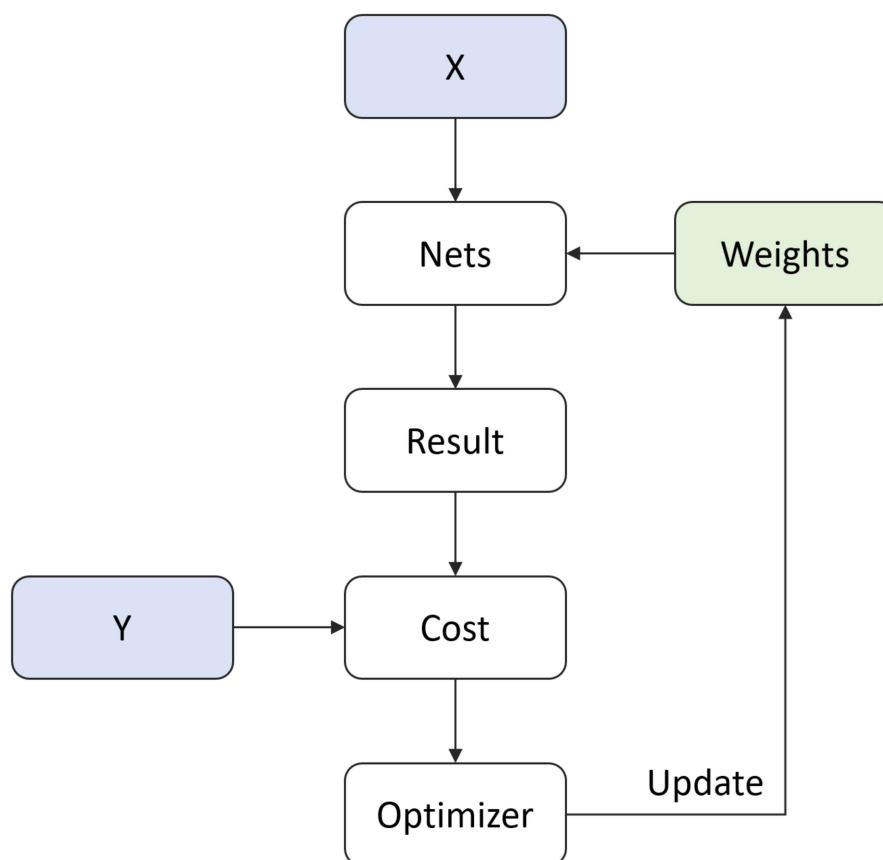
In [18]:

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

# optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)
LR = 0.0001
optm = tf.train.AdamOptimizer(LR).minimize(loss)

init = tf.global_variables_initializer()
```

## 4.7. Summary of Model



## 4.8. Define Configuration

- Define parameters for training ANN
  - **n\_batch**: batch size for stochastic gradient descent
  - **n\_iter**: the number of learning steps
  - **n\_prt**: check loss for every **n\_prt** iteration

In [19]:

```
n_batch = 50      # Batch Size
n_iter = 2500     # Learning Iteration
n_prt = 250      # Print Cycle
```

## 4.9. Optimization

In [20]:

```
# Run initialize
# config = tf.ConfigProto(allow_soft_placement=True) # GPU Allocating policy
# sess = tf.Session(config=config)
sess = tf.Session()
sess.run(init)

# Training cycle
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:
        c = sess.run(loss, feed_dict={x: train_x, y: train_y})
        print ("Iter : {}".format(epoch))
        print ("Cost : {}".format(c))
```

```
Iter : 0
Cost : 2.49772047996521
Iter : 250
Cost : 1.4581751823425293
Iter : 500
Cost : 0.7719646692276001
Iter : 750
Cost : 0.5604625344276428
Iter : 1000
Cost : 0.4211314022541046
Iter : 1250
Cost : 0.5066109299659729
Iter : 1500
Cost : 0.32460322976112366
Iter : 1750
Cost : 0.412553608417511
Iter : 2000
Cost : 0.26275649666786194
Iter : 2250
Cost : 0.481355220079422
```

## 4.10. Test



In [21]:

```
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))
```

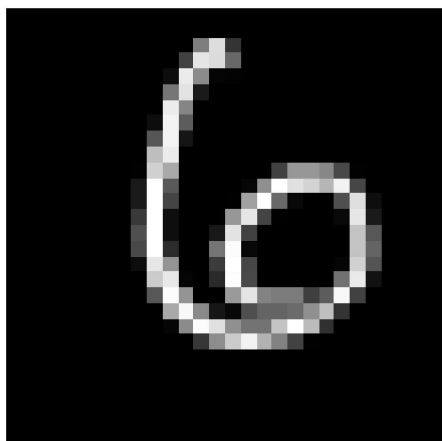
Accuracy : 96.0%

In [22]:

```
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 : 6
Probability : [ 0.01  0.01  0.07  0.    0.01  0.01  0.89  0.    0.01  0.
]
```

In [23]:

```
%%javascript
$.getScript('https://kmahelona.github.io/ipython_notebook_goodies/ipython_notebook_toc.
js')
```