

# (Artificial) Neural Networks in TensorFlow

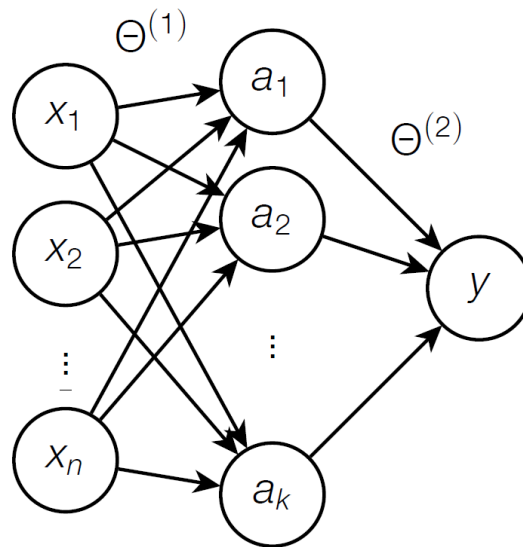
By Prof. Seungchul Lee  
iSystems Design Lab  
<http://isystems.unist.ac.kr/>  
UNIST

## Table of Contents

- I. 1. Artificial Neural Networks (ANN)
  - I. 1.1 Structure
  - II. 1.2. Training Neural Networks
- II. 2. Deep Learning Libraries
- III. 3. TensorFlow
  - I. 3.1. Computational Graph
  - II. 3.2. Example: Linear Regression using TensorFlow
- IV. 4. ANN with TensorFlow
  - I. 4.1. Import Library
  - II. 4.2. Load MNIST Data
  - III. 4.3. Build a Model
  - IV. 4.4. Define an ANN 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

# 1. Artificial Neural Networks (ANN)

## 1.1 Structure

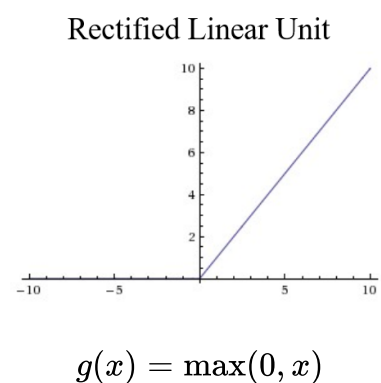
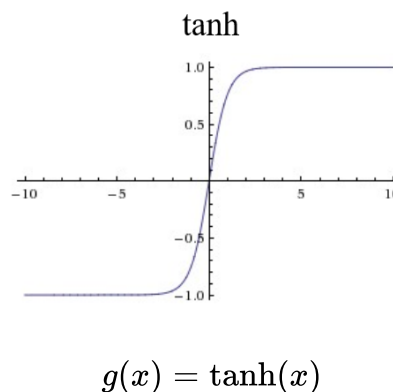
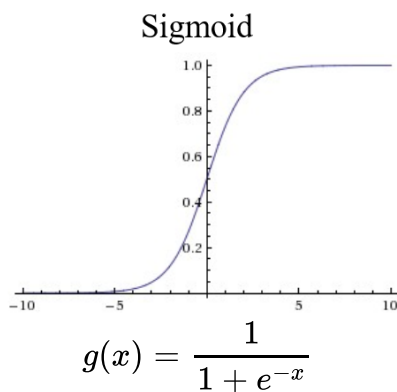


### Transformation

- Affine (or linear) transformation and nonlinear activation (layer)

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

- Nonlinear activation function



## 1.2. Training Neural Networks

### Loss Function

- Measures error between target values and predictions
- More or less the same as those for other parametric models, such as linear models

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

- Example

- Cross entropy:

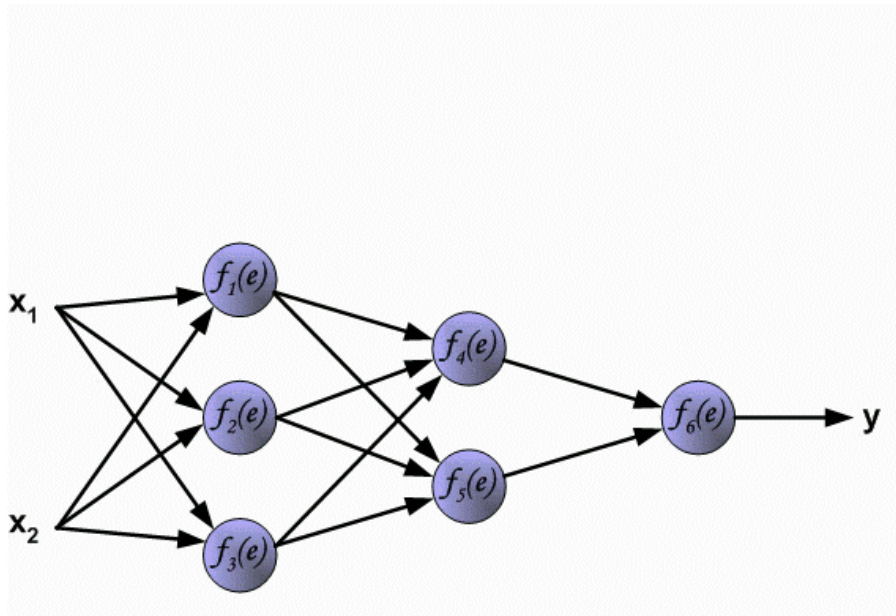
$$-\frac{1}{N} \sum_{i=1}^N y^{(i)} \log(h_{\theta}(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_{\theta}(x^{(i)}))$$

- Squared loss:

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

## Backpropagation

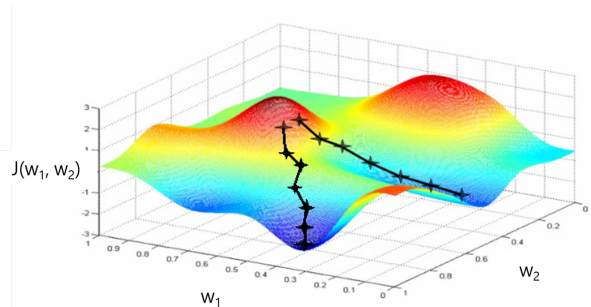
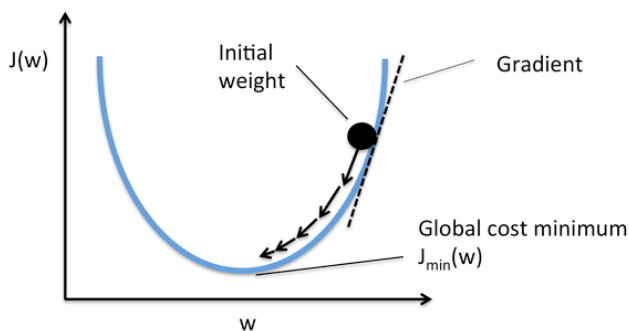
- Forward propagation
  - the initial information propagates up to the hidden units at each layer and finally produces output
- Backpropagation
  - allows the information from the cost to flow backwards through the network in order to compute the gradients



## (Stochastic) Gradient Descent

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

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



## 2. 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

## 3. TensorFlow

- tensorflow is an open-source software library for deep learning.

### 3.1. 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>
```

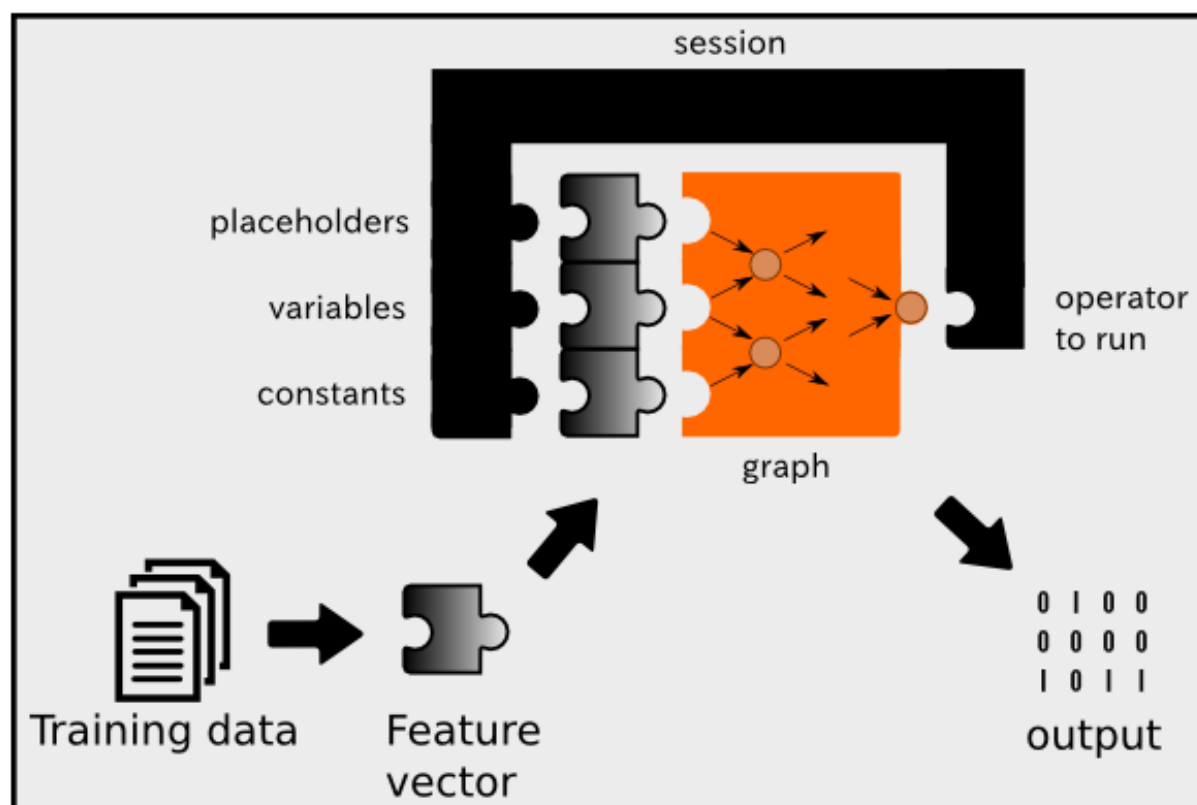
In [3]:

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)
```

## 3.2. Example: Linear Regression using TensorFlow

Given  $\begin{cases} x_i : \text{inputs} \\ y_i : \text{outputs} \end{cases}$ , Find  $\omega_1$  and  $\omega_2$

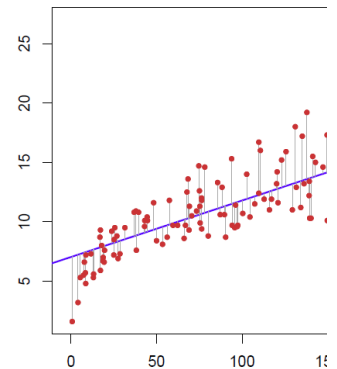
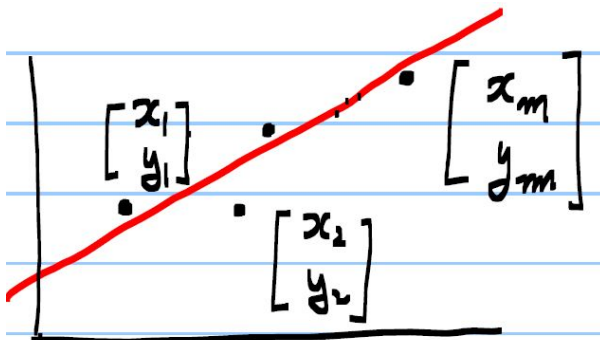
$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}, \quad y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} \approx \hat{y}_i = \omega_1 x_i + \omega_2$$

- $\hat{y}_i$  : predicted output
- $\omega = \begin{bmatrix} \omega_1 \\ \omega_2 \end{bmatrix}$  : Model parameters

$$\hat{y}_i = f(x_i, \omega) \text{ in general}$$

- in many cases, a linear model to predict  $y_i$  used

$$\hat{y}_i = \omega_1 x_i + \omega_2 \text{ such that } \min_{\omega_1, \omega_2} \sum_{i=1}^m (\hat{y}_i - y_i)^2$$



### Data Generation

In [11]:

```
import numpy as np
print(np.random.rand(10))
print(np.random.randint(0,10,size=10))
```

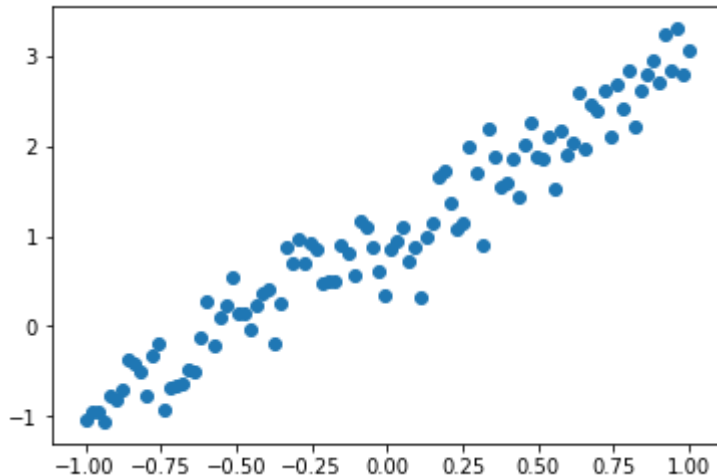
```
[ 0.867954    0.87020979  0.84237255  0.15221631  0.60090298  0.67016313
  0.48355536  0.89991     0.37963787  0.49485645]
[0 0 8 9 1 8 9 4 2 5]
```

In [12]:

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

data_x = np.linspace(-1, 1, 100)
data_y = 2 * data_x + 1 + np.random.randn(*data_x.shape) * 0.3

plt.scatter(data_x, data_y)
plt.show()
```



### Parameter Learning (or Estimation) by using TensorFlow

In [13]:

```
# Define decision variables in tf

weights = {
    'w' : tf.Variable(tf.random_normal([1], stddev=0.1))
}
biases = {
    'b' : tf.Variable(tf.random_normal([1], stddev=0.1))
}
```

In [14]:

```
x = tf.placeholder(tf.float32, [10])
y = tf.placeholder(tf.float32, [10])
```

$$\hat{y}_i = \omega x_i + b$$

In [15]:

```
# define model

def model(x, weights, biases):
    output = tf.add(tf.multiply(x, weights['w']), biases['b'])
    return output
```



$$\min_{\omega, b} \frac{1}{m} \sum_{i=1}^m (y_i - \hat{y}_i)^2$$

In [16]:

```
# define loss

pred = model(x, weights, biases)
loss = tf.square(tf.subtract(y, pred))
loss = tf.reduce_mean(loss)
```

In [17]:

```
# define optimizer

LR = 0.04
# optm = tf.train.AdamOptimizer(LR).minimize(loss)
optm = tf.train.GradientDescentOptimizer(LR).minimize(loss)
```

In [18]:

```
# tf.Variable initializer

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

In [19]:

```
# optimizing

n_iter = 200
n_prt = 20

for epoch in range(n_iter):
    idx = np.random.randint(0, 100, 10)
    train_x, train_y = data_x[idx], data_y[idx]
    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))

w_hat = sess.run(weights['w'])
b_hat = sess.run(biases['b'])

sess.close()
```

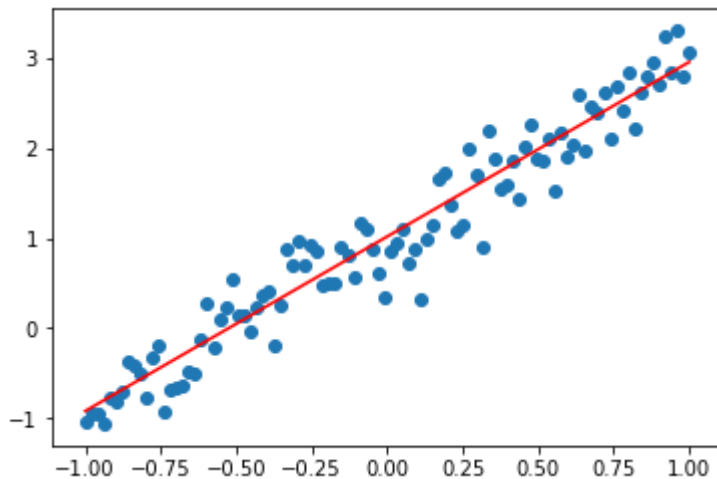
```
Iter : 0
Cost : 2.357508897781372
Iter : 20
Cost : 0.5955565571784973
Iter : 40
Cost : 0.3767774701118469
Iter : 60
Cost : 0.11836300045251846
Iter : 80
Cost : 0.21461530029773712
Iter : 100
Cost : 0.07787320017814636
Iter : 120
Cost : 0.06905560195446014
Iter : 140
Cost : 0.03441782668232918
Iter : 160
Cost : 0.037106819450855255
Iter : 180
Cost : 0.05782034248113632
```

In [20]:

```
print ("w_hat : {}".format(w_hat))
print ("b_hat : {}".format(b_hat))
```

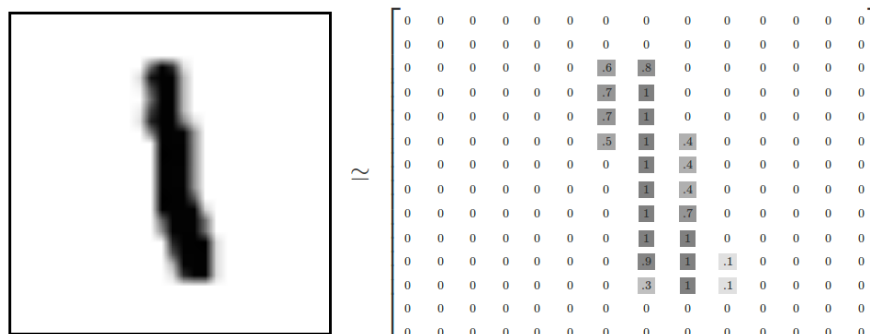
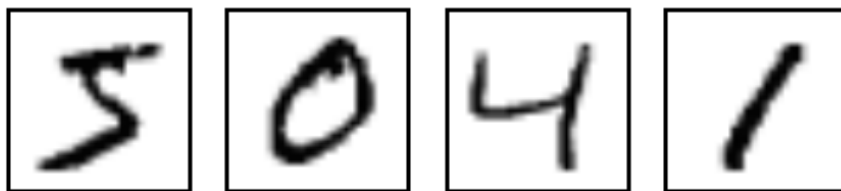
```
plt.scatter(data_x, data_y)
learned_y = data_x*w_hat + b_hat
plt.plot(data_x, learned_y, 'r')
plt.show()
```

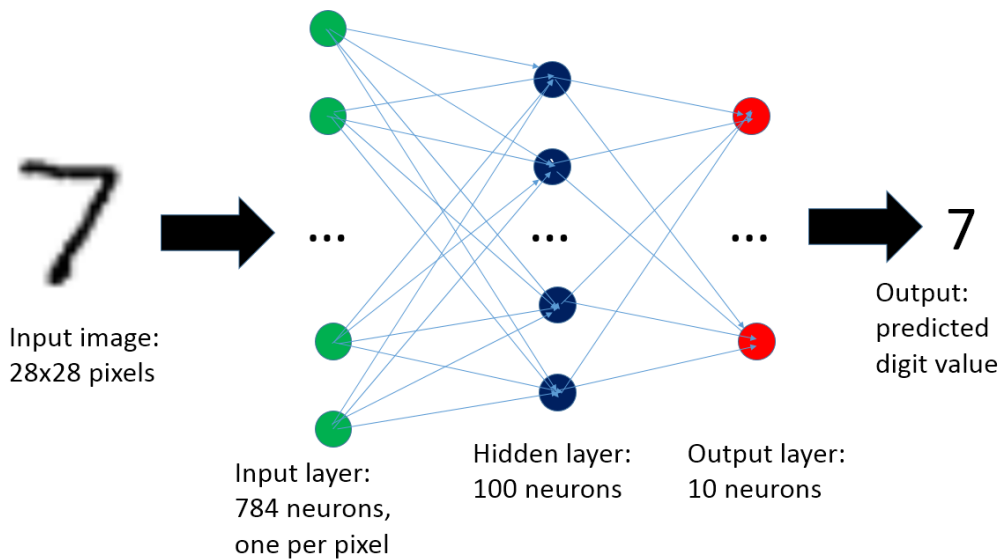
```
w_hat : [ 1.93427074]
b_hat : [ 1.01676476]
```



## 4. ANN with TensorFlow

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





In [21]:

```
%%html
<center><iframe src="https://www.youtube.com/embed/z0bynQjEpII?start=2088&end=3137"
width="560" height="315" frameborder="0" allowfullscreen></iframe></center>
```

ml4a @ itp-nyu :: 01 introduction, neural networks



## 4.1. Import Library

In [22]:

```
# 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 [23]:

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

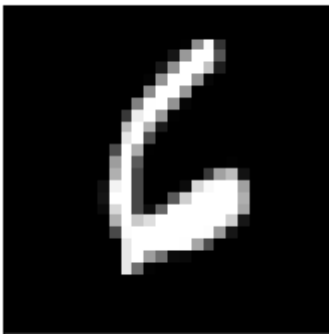
```
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
```

In [24]:

```
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

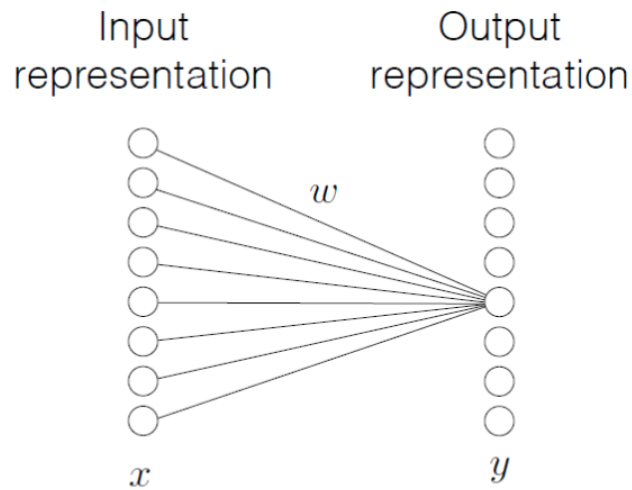
In [25]:

```
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

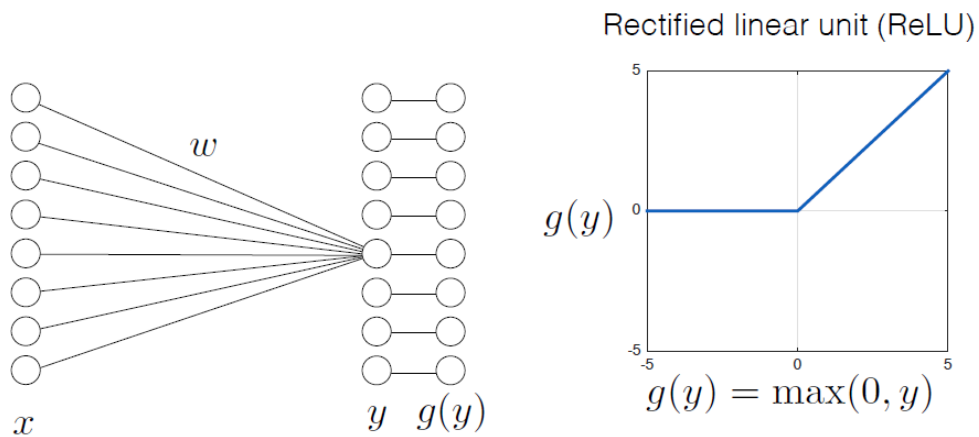


$$y_j = \left( \sum_i \omega_{ij} x_i \right) + b_j$$

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

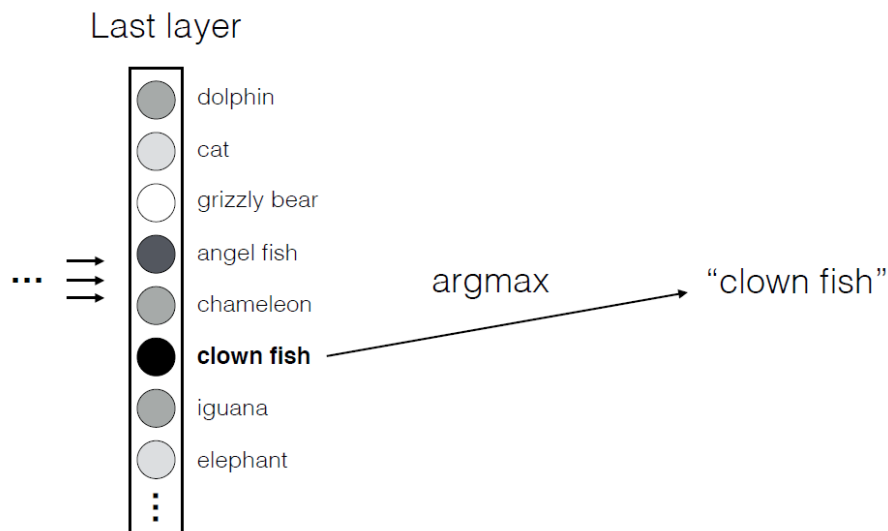
```
# hidden1 = tf.matmul(x, weights['hidden1']) + biases['hidden1']
hidden1 = tf.add(tf.matmul(x, weights['hidden1']), biases['hidden1'])
```

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



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

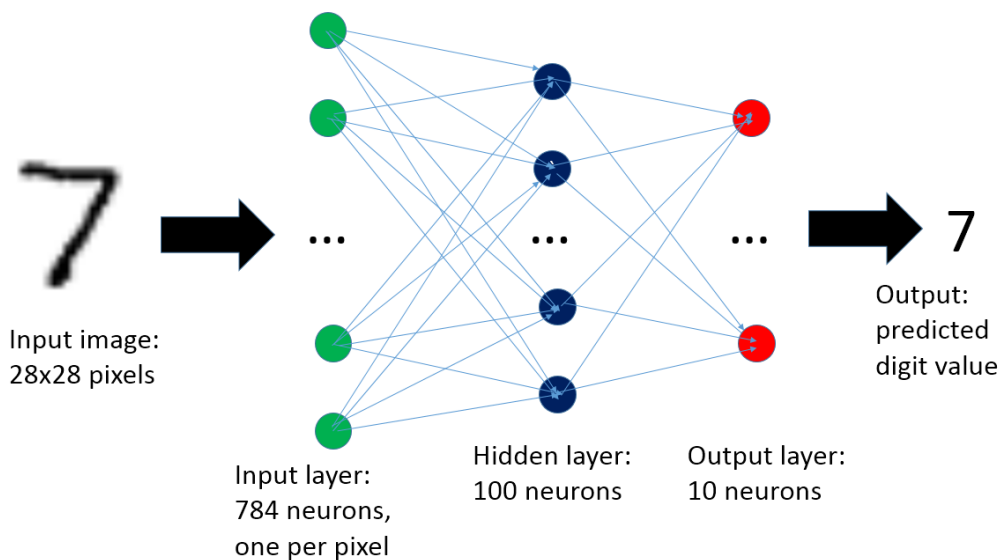
### Third, predict values with affine transformation



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

## 4.4. Define an ANN Shape

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



In [26]:

```
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 [27]:

```
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 [28]:

```
# 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 [29]:

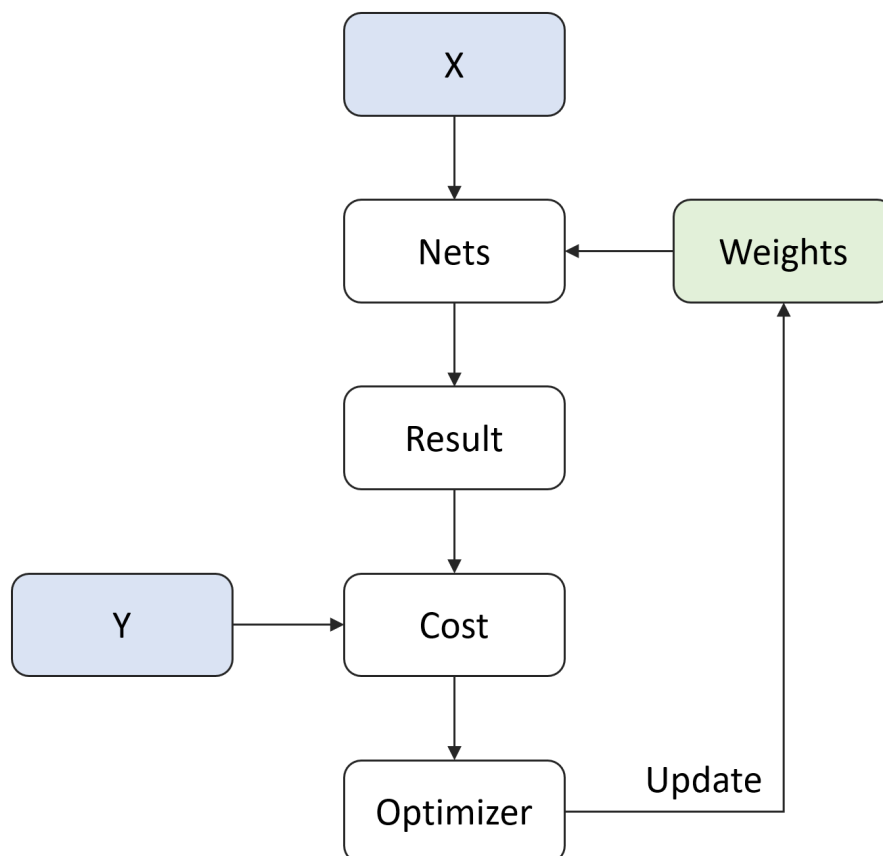
```
# Define Cost
LR = 0.0001

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)
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 [30]:

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

## 4.9. Optimization

In [31]:

```
# 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 : 2.465341567993164
Iter : 250
Cost : 1.483945369720459
Iter : 500
Cost : 0.7660821676254272
Iter : 750
Cost : 0.5961087942123413
Iter : 1000
Cost : 0.43914544582366943
Iter : 1250
Cost : 0.5471336245536804
Iter : 1500
Cost : 0.3527510464191437
Iter : 1750
Cost : 0.40383994579315186
Iter : 2000
Cost : 0.31749841570854187
Iter : 2250
Cost : 0.26046645641326904
```

## 4.10. Test

In [32]:

```
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))
```

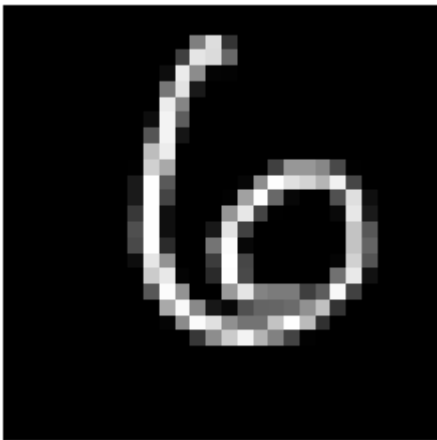
Accuracy : 95.0%

In [33]:

```
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.    0.    0.08 0.01 0.01 0.    0.89 0.    0.01 0.
]
```

In [34]:

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