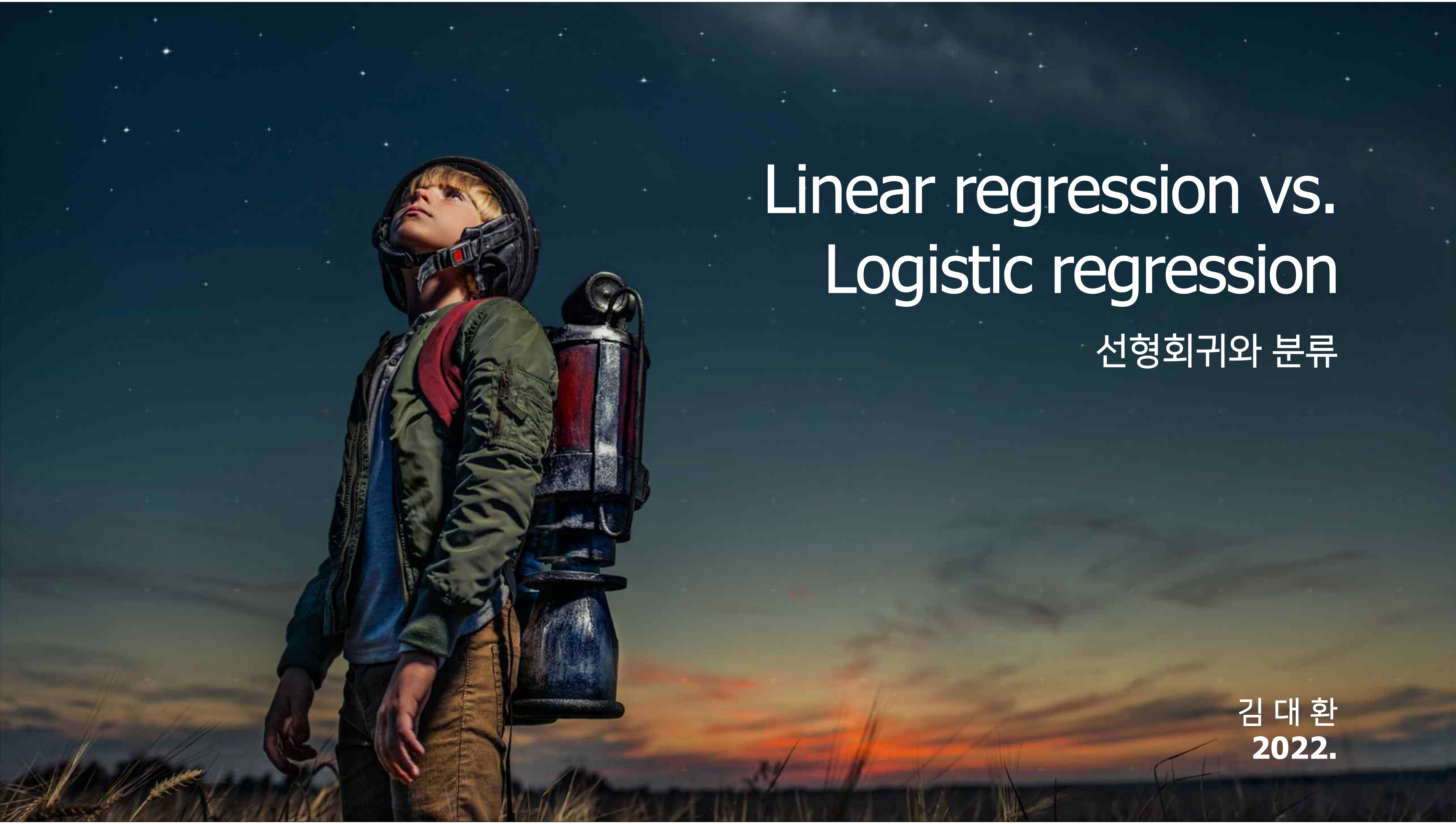


# Linear regression vs. Logistic regression

## 선형회귀와 분류

김 대 환

**2022.**

# 회귀(Regression)와 분류(Classification)

- 지도 학습을 통한 머신러닝
  - 크게 회귀(Regression)와 분류(Classification) 두 종류가 있다
  - 분류 작업과 회귀 작업을 구분하는 방법은 출력에 어떤 종류의 연속성이 있는지를 묻는 것

- 회귀
  - 연속되는 수 또는 부동 소수점 수(또는 수학 용어의 실수)를 예측하는 것

- 분류
  - 미리 정의된 가능성 목록에서 선택하는 클래스 레이블을 예측하는 것
  - 이진 분류(binary classification)와 멀티 클래스 분류(multiclass classification)로 나뉨

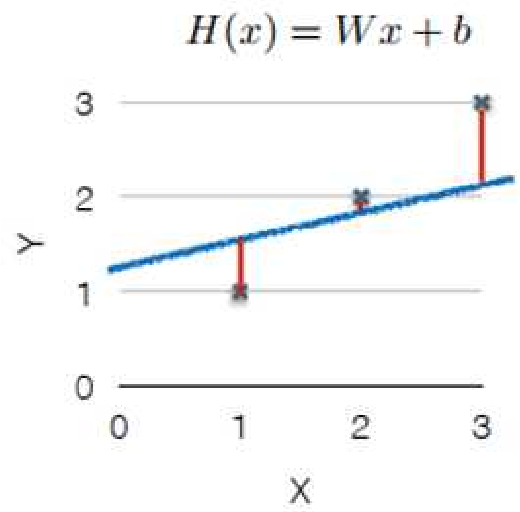
# 선형 회귀(Linear Regression)

## ● 선형 회귀 (Linear Regression)

- 입력에 대한 출력이 직선에 가까운 연속성을 가지는 것
- **독립변수(X)**로 **종속변수(Y)**를 예측하는 것

$$H(x) = WX + b$$

- H(x)는 입력 X에 대한 추론 값 **(Hypothesis),** W는 가중치**(weight)**, b는 편향**(bias)** 값을 의미한다
- 추론 값과 실제 값의 차이 (H(x) – Y) 발생
  - ✓ 처음 가정한 가중치와 편향 값은 실제와 많이 다를 수 있다
  - ✓ 머신 러닝은 학습을 통해 가중치와 편향 값을 실제와 가깝게 찾아가는 과정



# 선형 회귀(Linear Regression)

- 학습
  - 가정한 값과 실제 값의 차이가 클수록 손실(또는 비용)이 크다
  - 손실 (또는 비용)은 손실 (또는 비용) 함수로 표현하며, 실제 값과 가정한 값의 편차를 이용한다

$$\frac{(H(x)^{(1)} - y^{(1)})^2 + (H(x)^{(2)} - y^{(2)})^2 + (H(x)^{(3)} - y^{(3)})^2}{3}$$

  - $H(x)$ = Wx + b 이므로 손실 함수는 W와 b의 함수로 표현할 수 있고, 이를 일반화하면 아래와 같다

$$Cost(W, b) = \frac{1}{m} \sum_{i=1}^{m} (H(x)^{(i)} - y^{(i)})^2$$

# 선형 회귀(Linear Regression)

- 가중치(**weight**)와 손실(**cost**)과의 상관관계 표현

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

# tf Graph Input
X = [1., 2., 3.]
Y = [1., 2., 3.]
m = n_samples = len(X)

# Set model weights
W = tf.placeholder(tf.float32)

# Construct a linear model
hypothesis = tf.multiply(X, W)

# cost function
cost = tf.reduce_sum(tf.pow(hypothesis-Y, 2)) / (m)

# Initializing the variables
#init = tf.initialize_all_variables()
init = tf.global_variables_initializer()
```

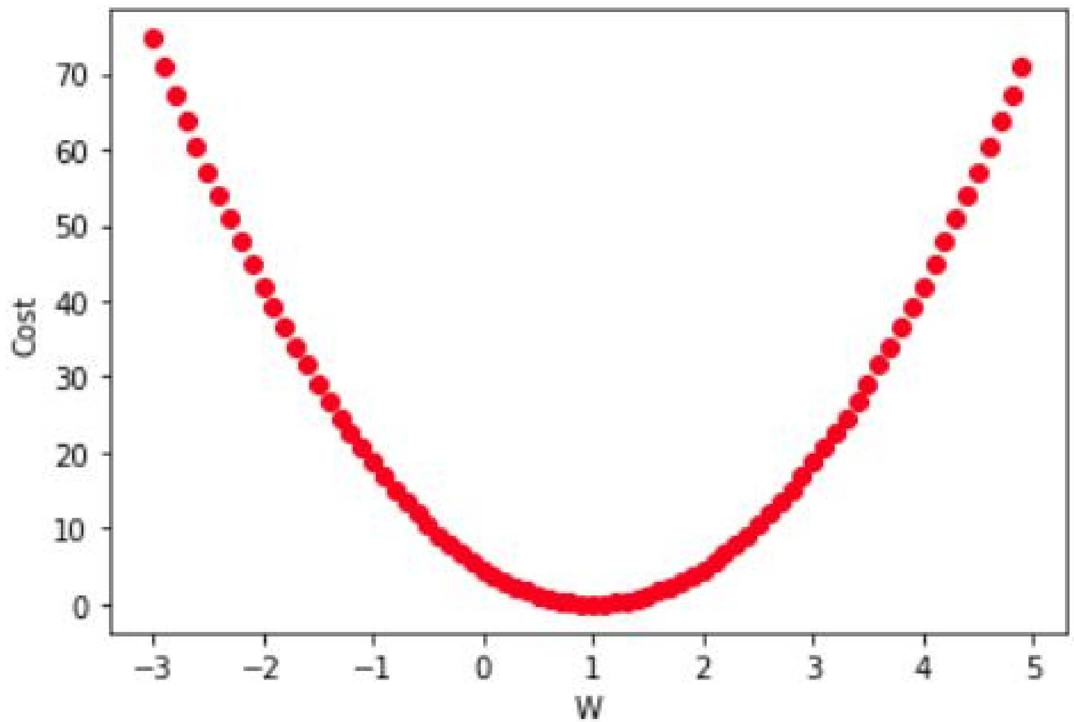
```
# For graphs
W_val = []
cost_val = []

# Launch the graph
sess = tf.Session()
sess.run(init)
for i in range(-30, 50):
    print (i*0.1, sess.run(cost, feed_dict={W: i*0.1}))
    W_val.append(i*0.1)
    cost_val.append(sess.run(cost, feed_dict={W: i*0.1
}))

# Graphic display
plt.plot(W_val, cost_val, 'ro')
plt.ylabel('Cost')
plt.xlabel('W')
plt.show()
```

# 선형 회귀(Linear Regression)

- 가중치**(weight)**와 손실**(cost)**과의 상관관계 표현
  - 가중치 W를 -3 부터 5까지 0.1 씩 값을 변경하면서 그 때의 Cost를 계산하여 그래프



# 경사 하강(Gradient Descent) 알고리즘

## 경사 하강법(Gradient Descent)

- Cost 함수를 W에 대해 편미분을 수행하여 접선의 기울기가 0에 가까워지는 지점을 찾는 알고리즘

$$Cost(W, b) = \frac{1}{m}\sum_{i=1}^{m}(H(x)^{(i)} - y^{(i)})^2$$

- 추론 (Hyposis) 함수 H(x) = Wx 를 대입한 후,  W에 대한 편미분 수식은 아래와 같다
  - ✓ 학습률(running rate)라고 하는 $\alpha$를 곱하고, 계수를 생략하기 위해 1/2을 곱하고 있다

$$\alpha\frac{\delta}{\delta W}Cost(W) = \alpha\frac{\delta}{\delta W}\frac{1}{2m}\sum_{i=1}^{m}(Wx^{(i)} - y^{(i)})^2$$

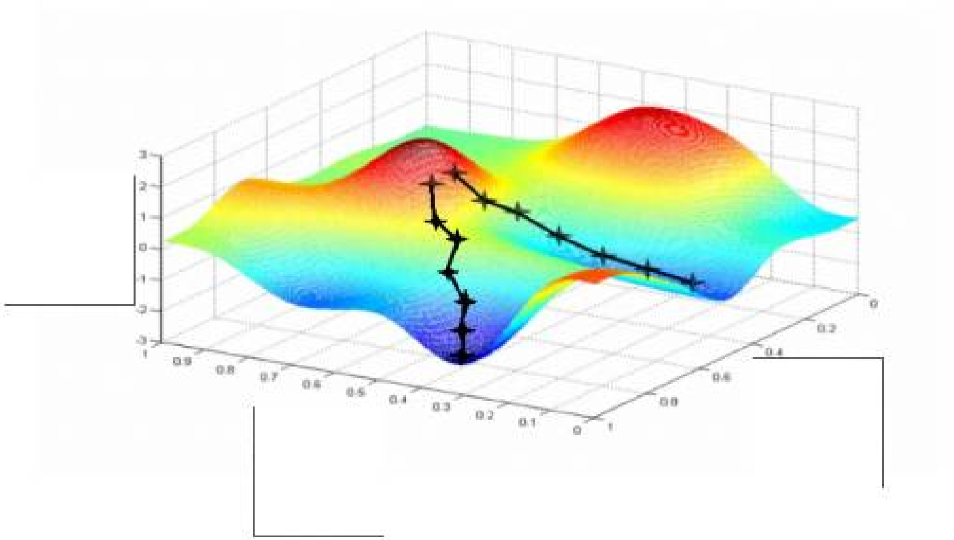
- 가중치 W 값은 학습이 진행되면서 직전 W 값에서 Cost를 줄이는 방향으로 조정되어야 한다

$$W := W - \alpha\frac{1}{m}\sum_{i=1}^{m}(Wx^{(i)} - y^{(i)})x^{(i)}$$

# Convex Function

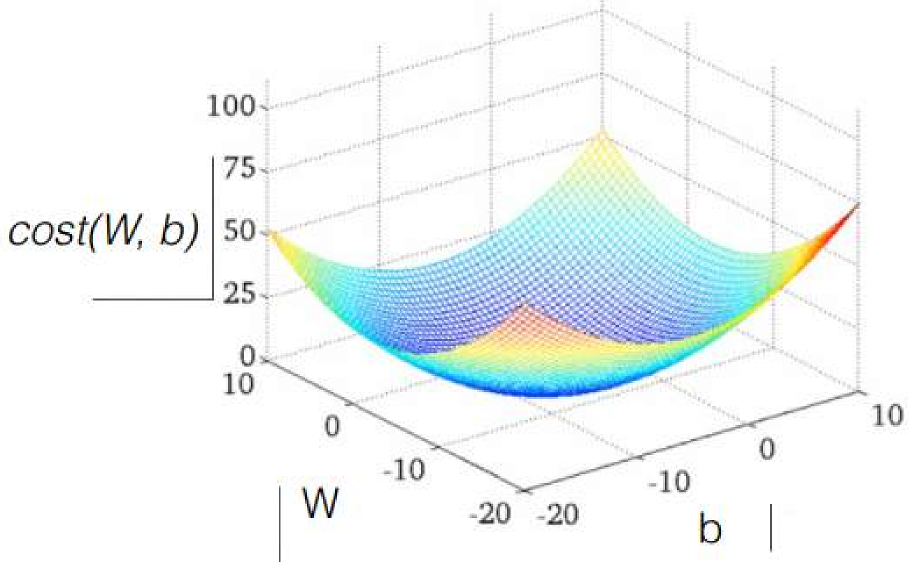
- **Cost 함수의 출력 그래프가 Convex Function 그래프와 닮은 꼴인지 확인한다**
  - 우리의 Cost 함수는 Convex function 이다

Non-convex function



Convex function

$$cost(W,b) = \frac{1}{m}\sum_{i=1}^{m}(H(x^{(i)}) - y^{(i)})^2$$



# 경사 하강(Gradient Descent) 알고리즘

- **Python으로 구현**

```
import tensorflow as tf

tf.set_random_seed(777)

x_data = [1, 2, 3]
y_data = [1, 2, 3]

# Find value for w to compute y_data = x_data * w
# We know that w should be 1,
# but let's use TensorFlow to figure it out
w = tf.Variable(tf.random_normal(([1]), name = "weight"))

X = tf.placeholder(tf.float32)
Y = tf.placeholder(tf.float32)

# Our hypothesis
hypothesis = X * w

#cost/loss function
cost = tf.reduce_mean(tf.square(hypothesis -Y))
```

```
learning_rate = 0.1
gradient = tf.reduce_mean((w*X -Y) * X)
descent = w - learning_rate * gradient
update = w.assign(descent)

# Launch the graph in a session
with tf.Session() as sess:
  # Initialiize globla variables in th graph
  sess.run(tf.global_variables_initializer())

  for step in range(20):
    _, cost_val, w_val  = sess.run([update, cost, w], feed
_dict={X: x_data, Y: y_data})
    print(step, cost_val, w_val)
```

# 경사 하강(Gradient Descent) 알고리즘

- **GradientDescentOptimizer** 라이브러리로 구현

```
import tensorflow as tf
tf.set_random_seed(777)  # for reproducibility

# X and Y data
x_train = [1, 2, 3]
y_train = [1, 2, 3]

# Try to find values for W and b to compute y_data =
 x_data * W + b
# We know that W should be 1 and b should be 0
# But let TensorFlow figure it out
W = tf.Variable(tf.random_normal([1]), name="weight")
b = tf.Variable(tf.random_normal([1]), name="bias")

# Our hypothesis XW+b
hypothesis = x_train * W + b

# cost/loss function
cost = tf.reduce_mean(tf.square(hypothesis -
 y_train))
```

```
# optimizer
train = tf.train.GradientDescentOptimizer(learning_rate=0.0
1).minimize(cost)

# Launch the graph in a session.
with tf.Session() as sess:
    # Initializes global variables in the graph.
    sess.run(tf.global_variables_initializer())

    # Fit the line
    for step in range(2001):
        _, cost_val, W_val, b_val = sess.run([train, cost,
W, b])

        if step % 20 == 0:
            print(step, cost_val, W_val, b_val)
```

# Multi-Variable Linear Regression

- 입력 변수가 여러 개인 경우
  - 예를 들어, 입력 변수 세 개에 대한 예측함수

$$H(x_1, x_2, x_3) = x_1w_1 + x_2w_2 + x_3w_3 + b$$

  - 입력 변수와 가중치에 대한 곱은 행렬 곱의 형태로 표현

$$\begin{pmatrix} x_1 & x_2 & x_3 \end{pmatrix} * \begin{pmatrix} w_1 \\ w_2 \\ w_3 \end{pmatrix}$$

# Multi-Variable Linear Regression

- 입력 변수가 여러 개인 경우
  - 수식을 행렬의 형태로 표현하면 많은 데이터도 매우 간단히 기술할 수 있다

| x1 | x2 | x3 | y |
|---|---|---|---|
| 73 | 80 | 75 | 152 |
| 93 | 88 | 93 | 185 |
| 89 | 91 | 90 | 180 |
| 96 | 98 | 100 | 196 |
| 73 | 66 | 70 | 142 |

$$\begin{pmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{pmatrix} * \begin{pmatrix} w_1 \\ w_2 \\ w_3 \end{pmatrix} = \begin{pmatrix} x_{11}w_1 + x_{12}w_2 + x_{13}w_3 \\ x_{21}w_1 + x_{22}w_2 + x_{23}w_3 \\ x_{31}w_1 + x_{32}w_2 + x_{33}w_3 \end{pmatrix}$$

# Multi-Variable Linear Regression

## Multi-Variable Linear Regression

- 여러 개의 입력 변수를 행렬식으로 표현

```python
import tensorflow as tf
tf.set_random_seed(777)  # for reproducibility

x_data = [[73., 80., 75.],
          [93., 88., 93.],
          [89., 91., 90.],
          [96., 98., 100.],
          [73., 66., 70.]]
y_data = [[152.],
          [185.],
          [180.],
          [196.],
          [142.]]

# placeholders for a tensor that will be always fed.
X = tf.placeholder(tf.float32, shape=[None, 3])
Y = tf.placeholder(tf.float32, shape=[None, 1])

W = tf.Variable(tf.random_normal([3, 1]), name='weight')
b = tf.Variable(tf.random_normal([1]), name='bias')
```

```python
# Hypothesis
hypothesis = tf.matmul(X, W) + b

# Simplified cost/loss function
cost = tf.reduce_mean(tf.square(hypothesis - Y))

# Minimize
optimizer = tf.train.GradientDescentOptimizer(learning_rate=1e-5)
train = optimizer.minimize(cost)

# Launch the graph in a session.
sess = tf.Session()
# Initializes global variables in the graph.
sess.run(tf.global_variables_initializer())

for step in range(2001):
    cost_val, hy_val, _ = sess.run(
        [cost, hypothesis, train], feed_dict={X: x_data, Y: y_dat
a})
    if step % 10 == 0:
        print(step, "Cost: ", cost_val, "\nPrediction:\n", hy_val)
```

# 로지스틱 회귀 (logistic regression)

- 종속 변수로 범주형 데이터를 입력 데이터로 주었을 때 결과가 특정 분류**(classification)** 나뉨
  - 영국의 통계학자인 D. R. Cox에 의해 1958년에 제안

- 종속 변수가 이항형 문제**(**유효한 범주의 갯수가 두개인 경우**)**를 지칭할 때 사용

- 두 개 이상의 범주를 가지는 경우에는 다항 로지스틱 회귀**(multinominal logistic regression)** 또는 서수 로지스틱 회귀**(ordinal logistic regression)**라 한다

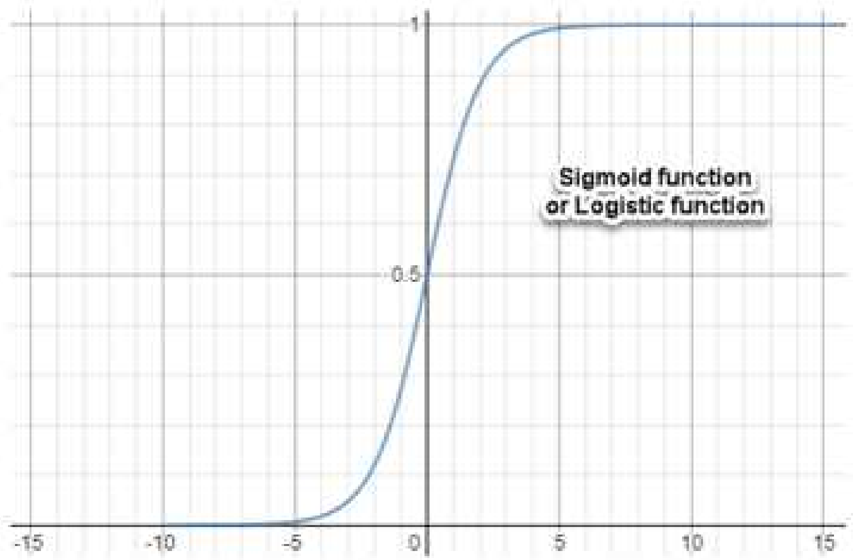
- 선형 회귀의 문제점
  - 추론 함수(H(x) = Wx + b)의 결과 값이 0보다 아주 작거나 1보다 매우 큰 값을 가질 수 있기 때문에, True나 False 혹은 Pass 또는 Fail 의 경계를 합리적으로 나누기 어렵다

# 로지스틱 회귀 (logistic regression)

- 선형 함수와 로지스틱 회귀의 추론 함수

$$h(x) = Wx + b = z \qquad g(z) = \frac{1}{(1+e^{-z})}$$

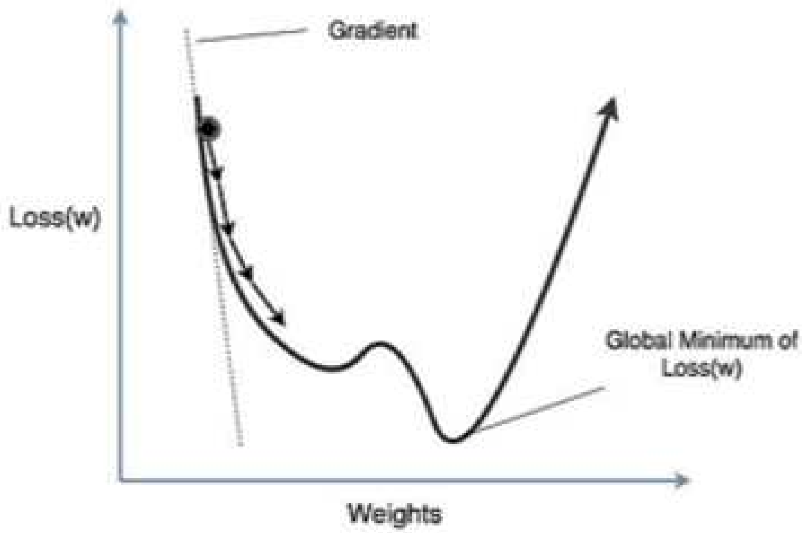
- 시그모이드 함수 그래프



# 로지스틱 회귀 (logistic regression)

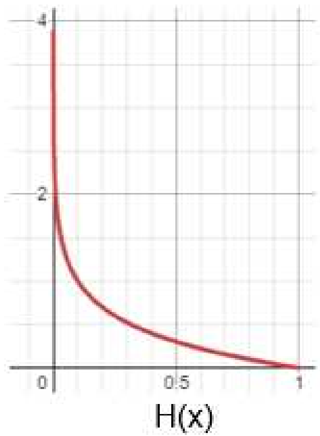
- **손실/비용(Cost) 함수**
  - 추론 함수의 변경으로, 손실 함수도 변경 되어야 함
  - 변경없이 사용했을 경우

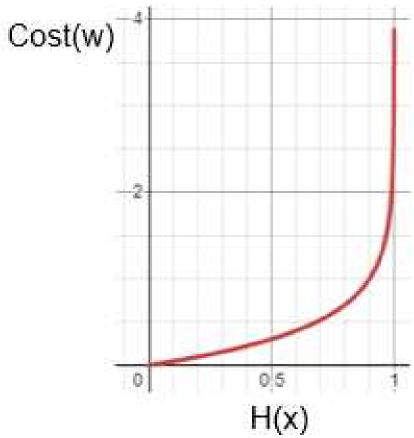
$$Cost(W,b) = \frac{1}{m}\sum_{i=1}^{m}(H(x)^{(i)} - y^{(i)})^2 \quad when \quad H(x) = \frac{1}{(1+e^{-(wx+b)})}$$



# 로지스틱 회귀 (logistic regression)

- 수정된 손실/비용(**Cost**) 함수
  - 실제 값 1일 때, 0으로 예측하면 손실이 무한대, 그 반대도 마찬가지
  - 실제 값 1일 때, 1을 예측하면 손실이 0, 그 반대도 마찬가지




$$cost(W) = \begin{cases} -\log(h(x)) & : y = 1 \\ -\log(1 - h(x)) & : y = 0 \end{cases}$$

- 이를 하나의 수식으로 표현하면,

$$c(W) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}\log(h(x)^{(i)}) + (1 - y^{(i)})\log(1 - h(x)^{(i)})]$$

- 조정되는 가중치는

$$W := W - \alpha\frac{\delta}{\delta W}cost(W)$$

# 로지스틱 회귀 (logistic regression)

- 로지스틱 회귀에 대한 파이썬 예제

```
import tensorflow as tf

# for reproducibility
tf.set_random_seed(777)

x_data = [[1, 2],
          [2, 3],
          [3, 1],
          [4, 3],
          [5, 3],
          [6, 2]]
y_data = [[0],
          [0],
          [0],
          [1],
          [1],
          [1]]
```

```
# placeholders for a tensor that will be always fed.
X = tf.placeholder(tf.float32, shape=[None, 2])
Y = tf.placeholder(tf.float32, shape=[None, 1])

W = tf.Variable(tf.random_normal([2, 1]), name='weight')
b = tf.Variable(tf.random_normal([1]), name='bias')

# Hypothesis using sigmoid: tf.div(1., 1. + tf.exp(tf.matmul(X, W)))
hypothesis = tf.sigmoid(tf.matmul(X, W) + b)

# cost/loss function
cost = -tf.reduce_mean(Y * tf.log(hypothesis) + (1 - Y) *
                       tf.log(1 - hypothesis))

train = tf.train.GradientDescentOptimizer(learning_rate=0.01).minimize
(cost)

# Accuracy computation
# True if hypothesis>0.5 else False
predicted = tf.cast(hypothesis > 0.5, dtype=tf.float32)
accuracy = tf.reduce_mean(tf.cast(tf.equal(predicted, Y), dtype=tf.flo
at32))
```

# 로지스틱 회귀 (logistic regression)

- 로지스틱 회귀에 대한 파이썬 예제

```
# Launch graph
with tf.Session() as sess:
    # Initialize TensorFlow variables
    sess.run(tf.global_variables_initializer())

    for step in range(10001):
        cost_val, _ = sess.run([cost, train], feed_dict={X: x_data, Y: y_data})
        if step % 200 == 0:
            print(step, cost_val)

    # Accuracy report
    h, c, a = sess.run([hypothesis, predicted, accuracy],
                       feed_dict={X: x_data, Y: y_data})
    print("\nHypothesis: ", h, "\nCorrect (Y): ", c, "\nAccuracy: ", a)
```

출력 결과:

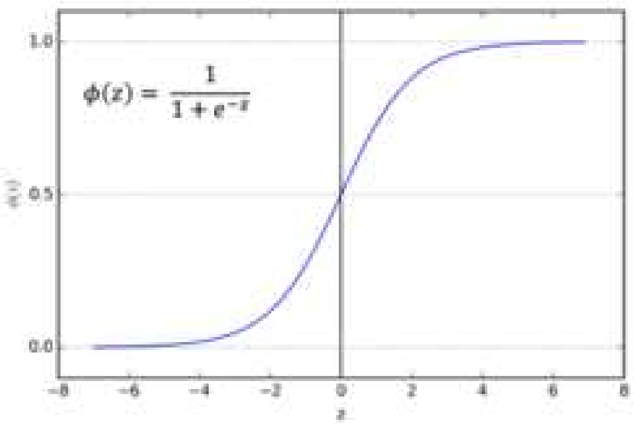
```
Hypothesis:

[[0.03074074]
 [0.15884674]
 [0.3048678 ]
 [0.78138095]
 [0.93957484]
 [0.98016894]]
Correct (Y):

[[0.]
 [0.]
 [0.]
 [1.]
 [1.]
 [1.]]
Accuracy:  1.0
```

# 소프트맥스 (Softmax)

- 두 가지 이상의 클래스를 분류해야 할 경우
  - 시그모이드 함수는 하나의 입력에 대해 0과 1 사이의 값을 출력한다.
  - 입력이 늘어나면 이에 대한 시그모이드 함수는 각각에 대해 0과 1 사이의 확률 값을 출력한다





$$\text{logits} \quad \begin{bmatrix} z_0 \\ z_1 \\ \vdots \\ z_n \end{bmatrix} \xrightarrow{\text{Sigmoid}} \begin{bmatrix} \frac{1}{1+\exp(-z_0)} \\ \frac{1}{1+\exp(-z_1)} \\ \vdots \\ \frac{1}{1+\exp(-z_n)} \end{bmatrix} \quad \text{independent probabilites}$$

# 소프트맥스 (Softmax)

- 두 가지 이상의 클래스를 분류해야 할 경우
  - 소프트맥스는 다중 입력에 대한 확률을 구하고 그 합이 1이 되도록 만든다


  - 전체 클래스 수가 세 개라면 소프트 맥스 함수는 다음과 같은 결과가 나온다

$$S(z) = [\frac{e^{z_1}}{e^{z_1}+e^{z_2}+e^{z_3}}, \frac{e^{z_2}}{e^{z_1}+e^{z_2}+e^{z_3}}, \frac{e^{z_3}}{e^{z_1}+e^{z_2}+e^{z_3}}] = [p_1, p_2, p_3]$$

# 소프트맥스 (Softmax)

- 두 가지 이상의 클래스를 분류해야 할 경우
  - 소프트맥스 함수의 생김새는 `k번째 확률/전체 확률`로써 일반화 시켜 보면 아래와 같다.

$$S(z)_k = \frac{e^{zk}}{\sum_{i=1}^{n} e^{zi}}$$

  - 시그모이드 함수를 S라 하면 소프트맥스 함수는 다음을 통해 유도된다.

$$S = \frac{1}{e^{-t}+1} \text{ 이므로, } \frac{S}{1-S} = e^{t}$$

  - 지수 함수는 단조증가 함수(계속 증가하는 함수) 이므로 인자들의 대소 관계가 변하지 않고 작은 차이도 구별하기 쉽도록 커지게 된다.
  - $e^x$에 대한 미분도 원래 값과 동일하기 때문에 미분하기 좋다

# 소프트맥스 (Softmax)

- 손실/비용 함수는 Cross-Entropy Loss 함수를 많이 사용한다

$$CE = -\sum_{i}^{C} y_i log(s_i)$$

- 정답 $y = \begin{bmatrix}1\\0\end{bmatrix}$ 에 대해 예측 $s = \begin{bmatrix}1\\0\end{bmatrix}$ 했다면 $\begin{bmatrix}1\\0\end{bmatrix} \odot \begin{bmatrix}0\\\infty\end{bmatrix} = \begin{bmatrix}0\\0\end{bmatrix}$로 손실이 없다
- 반면, $s = \begin{bmatrix}0\\1\end{bmatrix}$ 으로 예측 했다면 $\begin{bmatrix}1\\0\end{bmatrix} \odot \begin{bmatrix}\infty\\0\end{bmatrix} = \begin{bmatrix}\infty\\0\end{bmatrix}$ 되어 손실이 무한대가 된다
- 로지스틱 회귀를 이용한 이진 분류에 했을 경우 아래와 같으며 결과가 동일한 것을 알 수 있다

$$CE = -\sum_{i}^{C=2} y_i log(s_i) = -y_1 \log(s_1) - (1-y_1)\log(1-s_1)$$

- 소프트맥스 함수를 적용한 후, 출력 값에 'one-hot Encoding'을 적용하여 컴퓨터가 처리하기 쉽게 만든다
  - ✓ **'one-hot Encoding'**은 소프트맥스 출력에서 가장 높은 확률 값을 1로 하고 나머지는 0으로 만드는 기법이다

# 소프트맥스 (Softmax)

- 소프트맥스 함수 적용의 예

```python
import tensorflow as tf

# for reproducibility
tf.set_random_seed(777)
x_data = [[1, 2, 1, 1],
          [2, 1, 3, 2],
          [3, 1, 3, 4],
          [4, 1, 5, 5],
          [1, 7, 5, 5],
          [1, 2, 5, 6],
          [1, 6, 6, 6],
          [1, 7, 7, 7]]
y_data = [[0, 0, 1],
          [0, 0, 1],
          [0, 0, 1],
          [0, 1, 0],
          [0, 1, 0],
          [0, 1, 0],
          [1, 0, 0],
          [1, 0, 0]]
```

```python
X = tf.placeholder("float", [None, 4])
Y = tf.placeholder("float", [None, 3])
nb_classes = 3

W = tf.Variable(tf.random_normal([4, nb_classes]), name='weight')
b = tf.Variable(tf.random_normal([nb_classes]), name='bias')

# tf.nn.softmax computes softmax activations
# softmax = exp(logits) / reduce_sum(exp(logits), dim)
hypothesis = tf.nn.softmax(tf.matmul(X, W) + b)

# Cross entropy cost/loss
cost = tf.reduce_mean(-tf.reduce_sum(Y * tf.log(hypothesis), axis=1))

optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.1).minimi
ze(cost)
```

# 소프트맥스 (Softmax)

- 소프트맥스 함수 적용의 예

```
# Launch graph
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer(
))

    for step in range(2001):
            _, cost_val = sess.run([optimizer
, cost], feed_dict={X: x_data, Y: y_data})

            if step % 200 == 0:
                print(step, cost_val)
```

```
    print('--------------')
    # Testing & One-hot encoding
    a = sess.run(hypothesis, feed_dict={X: [[1, 11, 7, 9]]})
    print(a, sess.run(tf.argmax(a, 1)))

    print('--------------')
    b = sess.run(hypothesis, feed_dict={X: [[1, 3, 4, 3]]})
    print(b, sess.run(tf.argmax(b, 1)))

    print('--------------')
    c = sess.run(hypothesis, feed_dict={X: [[1, 1, 0, 1]]})
    print(c, sess.run(tf.argmax(c, 1)))

    print('--------------')
    all = sess.run(hypothesis, feed_dict={X: [[1, 11, 7, 9]
, [1, 3, 4, 3], [1, 1, 0, 1]]})
    print(all, sess.run(tf.argmax(all, 1)))
```

934v00