

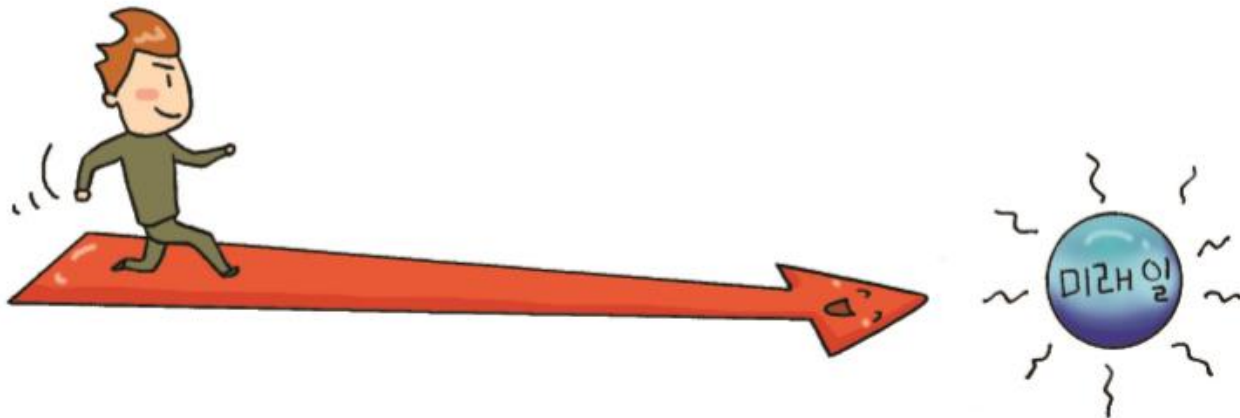
# 딥러닝의 동작 원리

# 가장 훌륭한 예측선 찾기: 선형 회귀

- 1 | 선형 회귀의 정의
- 2 | 가장 훌륭한 예측선이란?
- 3 | 최소 제곱법
- 4 | 코딩으로 확인하는 최소 제곱
- 5 | 평균 제곱근 오차
- 6 | 잘못 그은 선 바로잡기
- 7 | 코딩으로 확인하는 평균 제곱근 오차

- 딥러닝은 자그마한 통계의 결과들이 무수히 얹히고설켜 이루어지는 복잡한 연산의 결정체
- 딥러닝을 이해하려면 딥러닝의 가장 말단에서 이루어지는 가장 기본적인 두 가지 계산 원리, **선형 회귀**와 **로지스틱 회귀**를 알아야 함

- 가장 훌륭한 예측선 찾기란 통계학 용어인 **선형 회귀**(linear regression)를 쉽게 풀어 써 쓴 것
- 머신러닝은 제대로 된 선을 긋는 작업부터 시작됨
- 선의 방향을 잘 정하면 그 선을 따라가는 것만으로도 지금은 보이지 않는 미래의 것을 예측할 수 있기 때문



## 1 | 선형 회귀의 정의

“학생들의 중간고사 성적이 다 다르다.”

- 위 문장이 나타낼 수 있는 정보는 너무 제한적
- 학급의 학생마다 제각각 성적이 다르다는 당연한 사실 외에는 알 수 있는 게 없음

## 1 | 선형 회귀의 정의

"학생들의 중간고사 성적이 [     ]에 따라 다 다르다."

- 이 문장은 정보가 담길 여지를 열어 놓고 있음
- [     ] 부분에 시험 성적을 좌우할 만한 여러 가지 것이 들어간다면 좀 더 많은 사실을 전달할 수 있음
- 예를 들면 공부한 시간, 시험 당일의 컨디션, 사교육비 지출액 등이 들어갈 수 있음
- 무엇이 들어가든지 해당 성적의 이유를 나름대로 타당하게 설명할 수 있음
- 따라서 이 문장이 중간고사 성적의 차이와 이유를 나타낼 때 더욱 효과적

## 1 | 선형 회귀의 정의

- 여기서 [ ]에 들어갈 내용을 '정보'라고 함
- 머신러닝과 딥러닝은 이 정보가 필요함  
→ 많은 정보가 더 정확한 예측을 가능케하며, 이때의 '많은 정보'가 곧 '빅데이터'

## 1 | 선형 회귀의 정의

- 성적을 변하게 하는 '정보' 요소를  $x$ 라고 하고, 이  $x$  값에 의해 변하는 '성적'을  $y$ 라할때,
- $x$  값이 변함에 따라  $y$  값도 변한다
- 이때, 독립적으로 변할 수 있는 값  $x$ 를 독립 변수라고 함
- 이 독립 변수에 따라 종속적으로 변하는  $y$ 를 종속 변수라고 함

→ 선형 회귀란 독립 변수  $x$ 를 사용해 종속 변수  $y$ 의 움직임을 예측하고 설명하는 작업!



## 1 | 선형 회귀의 정의

- 독립 변수  $x$  하나만으로는 정확히 설명할 수 없을 때는  $x$  값을 여러 개( $x_1, x_2, x_3$  등) 준비한다.
- 하나의  $x$  값만으로도  $y$  값을 설명 할 수 있을 때 이를 **단순 선형 회귀**(simple linear regression)라고 함
- $x$  값이 여러 개 필요할 때는 **다중 선형 회귀**(multiple linear regression)라고 함

## 2 | 가장 훌륭한 예측선이란?

- 독립 변수가 하나뿐인 단순 선형 회귀의 예  
→ 성적을 결정하는 여러 요소 중에 '공부한 시간' 한 가지만 놓고 예측하는 경우
- 중간고사를 본 4명의 학생에게 각각 공부한 시간을 물어보고 이들의 중간고사 성적을 표와 같이 정리했을 때,

공부한 시간	2시간	4시간	6시간	8시간
성적	81점	93점	91점	97점

공부한 시간과 중간고사 성적 데이터

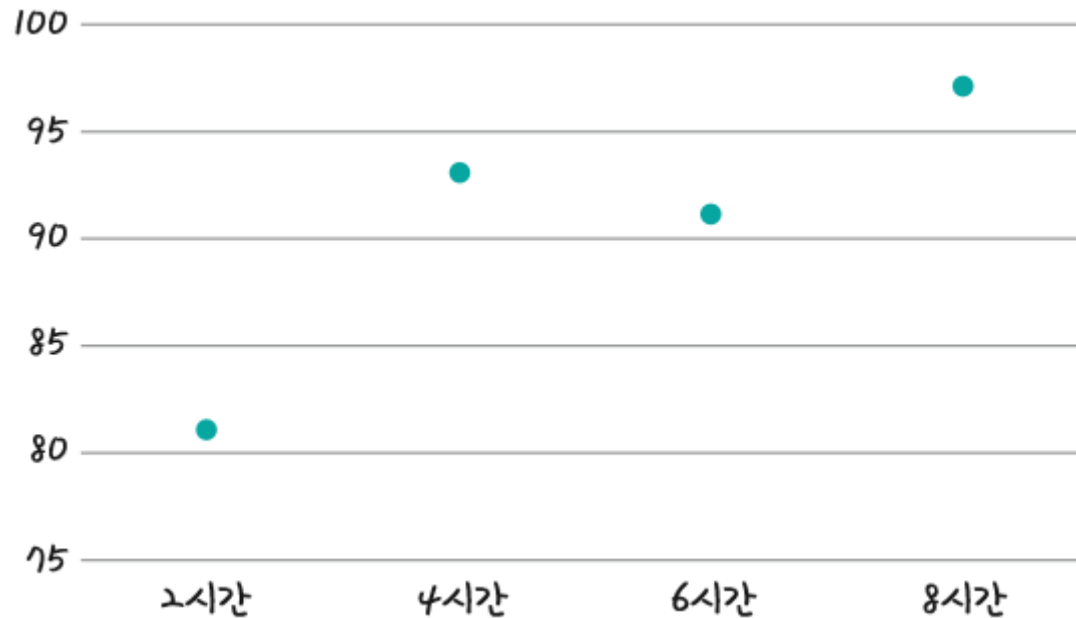
- 공부한 시간을  $x$ 라 하고 성적을  $y$ 라 할 때 집합  $x$ 와 집합  $y$ 를 다음과 같이 표현할 수 있음

$$x = \{2, 4, 6, 8\}$$

$$y = \{81, 93, 91, 97\}$$

## 2 | 가장 훌륭한 예측선이란?

- 이름 좌표 평면에 나타내면 그림과 같음



공부한 시간과 성적을 좌표로 표현

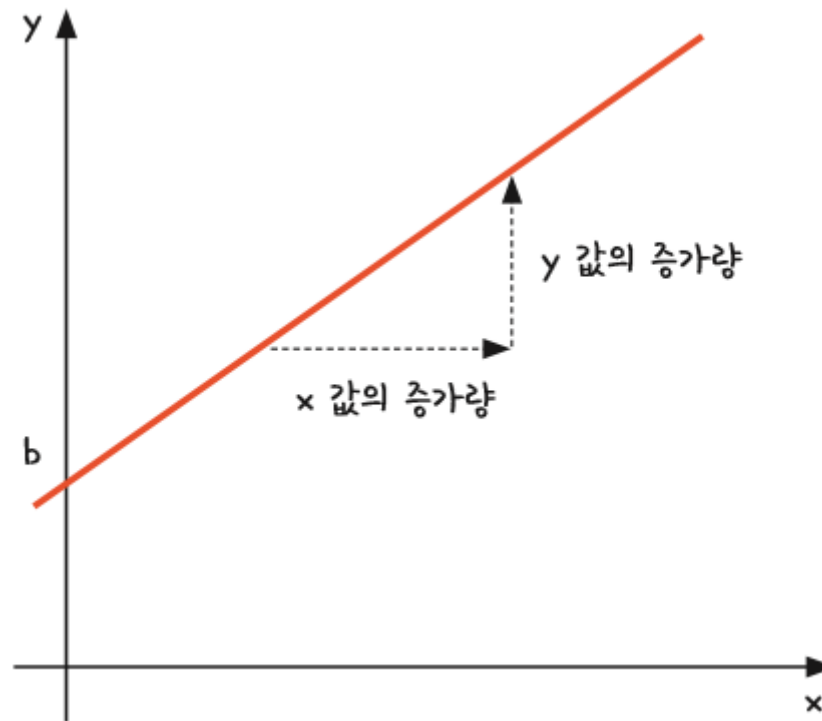
## 2 | 가장 훌륭한 예측선이란?

- 좌표 평면에 나타내 놓고 보니, 왼쪽이 아래로 향하고 오른쪽이 위를 향하는 일종의 '선형(직선으로 표시될 만한 형태)'을 보임
- 선형 회귀를 공부하는 과정은 이 점들의 특징을 가장 잘 나타내는 선을 그리는 과정과 일치
- 여기에서 선은 직선이므로 곧 일차 함수 그래프!  
다음과 같은 식으로 표현할 수 있음

$$y = ax + b$$

## 2 | 가장 훌륭한 예측선이란?

- $a$ 는 직선의 기울기, 즉  $\frac{y \text{ 값의 증가량}}{x \text{ 값의 증가량}}$  이고,  $b$ 는  $y$ 축을 지나는 값인 'y 절편'이 됨



일차 함수 그래프

## 2 | 가장 훌륭한 예측선이란?

- 여기서  $x$  값은 독립 변수이고  $y$  값은 종속 변수
- 즉,  $x$  값에 따라  $y$  값은 반드시 달라짐
- 다만, 정확하게 계산하려면 상수  $a$ 와  $b$ 의 값을 알아야 함
- 따라서 이 직선을 훌륭하게 그으려면
  1. 직선의 기울기  $a$  값과
  2.  $y$  절편  $b$  값을 정확히 예측해 내야 한다

## 2 | 가장 훌륭한 예측선이란?

- 선형 회귀는 곧 정확한 직선을 그려내는 과정
  - 최적의  $a$  값과  $b$  값을 찾아내는 작업!!
- 예측선을 그리는 이유
  - 잘 그어진 직선을 통해 우리는 표의 공부한 시간과 중간고사 성적 데이터에 들어 있지 않은 여러 가지 내용을 유추할 수 있다.
  - 예를 들어, 표에 나와 있지 않은 또 다른 학생의 성적을 예측하고 싶을때, 정확한 직선을 그어 놓았다면 이 학생이 몇 시간을 공부했는지만 물어보면 됨
  - 정확한  $a$ 와  $b$ 의 값을 따라 움직이는 직선에 학생이 공부한 시간인  $x$  값을 대입하면 예측 성적인  $y$  값을 구할 수 있는 것

## 2 | 가장 훌륭한 예측선이란?

- 딥러닝과 머신러닝의 '예측'이란?
  - 기존 데이터(정보)를 가지고 어떤 선이 그려질지를 예측한 뒤,
  - 아직 답이 나오지 않은 그 무언가를 그 선에 대입해 보는 것
- 선형 회귀의 개념을 이해하는 것은 딥러닝을 이해하는 중요한 첫걸음



### 3 | 최소 제곱법

- 정확한 기울기  $a$ 와 정확한  $y$  절편의 값  $b$ 를 알아내는 간단한 방법  
→ 최소 제곱법 (method of least squares)
- 최소 제곱법이란 회귀 분석에서 사용되는 표준 방식으로, 실험이나 관찰을 통해 얻은 데이터를 분석하여 미지의 상수를 구할 때 사용되는 공식
- 최소 제곱법 공식을 알고 적용한다면, 일차 함수의 기울기  $a$ 와  $y$  절편  $b$ 를 바로 구할 수 있다

### 3 | 최소 제곱법

- 지금 가진 정보가  $x$  값(입력 값, 여기서는 '공부한 시간')과  $y$  값(출력 값, 여기서는 '성적')일 때 최소 제곱법을 이용해 기울기  $a$ 를 구하는 방법
  - 각  $x$ 와  $y$ 의 편차를 곱해서 이를 합한 값을 구함
  - 그리고 이를  $x$  편차 제곱의 합으로 나눈다.

$$a = \frac{(x - x \text{ 평균})(y - y \text{ 평균}) \text{의 합}}{(x - x \text{ 평균})^2 \text{의 합}}$$

- 식으로 표현하면

$$a = \frac{\sum_{i=1}^n (x - \text{mean}(x)) (y - \text{mean}(y))}{\sum_{i=1}^n (x - \text{mean}(x))^2}$$

### 3 | 최소 제곱법

- 성적(y)과 공부한 시간(x)을 가지고 최소 제곱법을 이용해 기울기 a를 구하려면

1. x 값의 평균과 y 값의 평균을 각각 구한다.

- 공부한 시간(x) 평균:  $(2 + 4 + 6 + 8) \div 4 = 5$
- 성적(y) 평균:  $(81 + 93 + 91 + 97) \div 4 = 90.5$

2. 이를 식에 대입한다.

$$\begin{aligned} a &= \frac{(2-5)(81-90.5) + (4-5)(93-90.5) + (6-5)(91-90.5) + (8-5)(97-90.5)}{(2-5)^2 + (4-5)^2 + (6-5)^2 + (8-5)^2} \\ &= \frac{46}{20} \\ &= 2.3 \rightarrow \text{기울기는 2.3!} \end{aligned}$$

### 3 | 최소 제곱법

- 다음은  $y$  절편인  $b$ 를 구하는 공식

$$b = y \text{의 평균} - (x \text{의 평균} \times \text{기울기 } a)$$

- $y$ 의 평균에서  $x$ 의 평균과 기울기의 곱을 빼면  $b$ 의 값이 나온다는 의미
- 식으로 표현하면 다음과 같음

$$b = \text{mean}(y) - (\text{mean}(x) * a)$$

### 3 | 최소 제곱법

- 우리는 이미  $y$ 평균,  $x$ 평균, 그리고 조금 전 구한 기울기  $x$ 까지, 이 식을 풀기 위해 필요한 모든 변수를 알고 있음
- 이를 식에 대입해 보면,

$$\begin{aligned} b &= 90.5 - (2.3 \times 5) \\ &= 79 \quad \rightarrow y \text{ 절편 } b \text{는 } 79! \end{aligned}$$

- 이제 다음과 같이 예측 값을 구하기 위한 직선의 방정식이 완성됨

$$y = 2.3x + 79$$

### 3 | 최소 제곱법

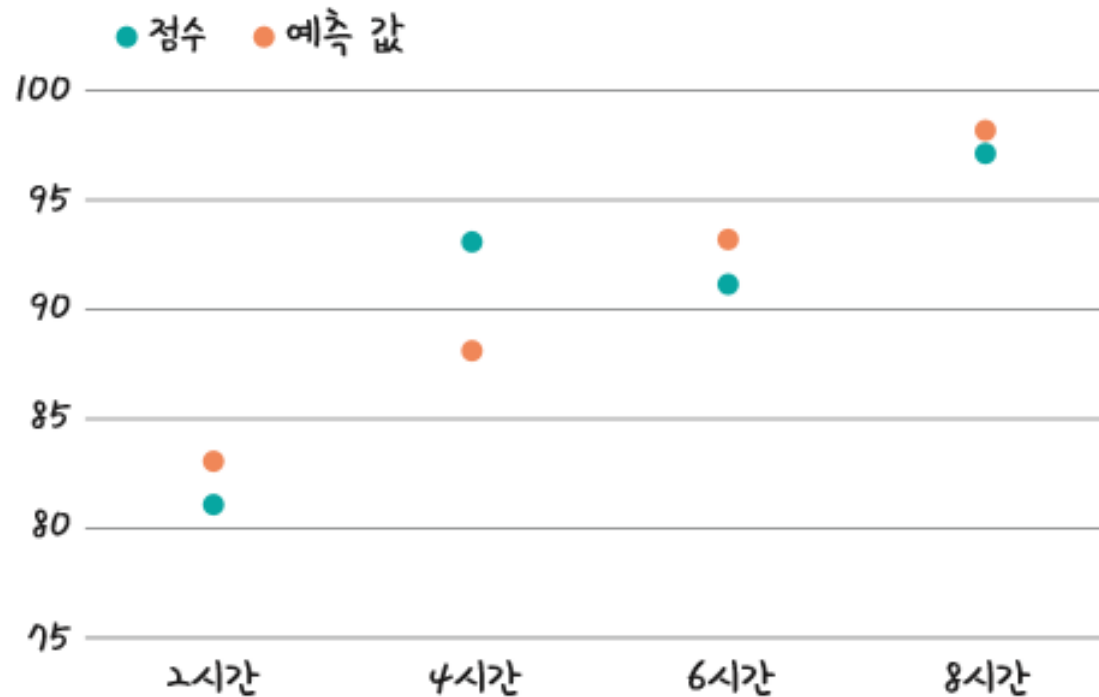
- 이 식에  $x$ 를 대입했을 때 나오는  $y$  값을 '예측 값'으로 정리하면,

공부한 시간	2	4	6	8
성적	81	93	91	97
예측 값	83.6	88.2	92.8	97.4

최소 제곱법 공식으로 구한 성적 예측 값

### 3 | 최소 제곱법

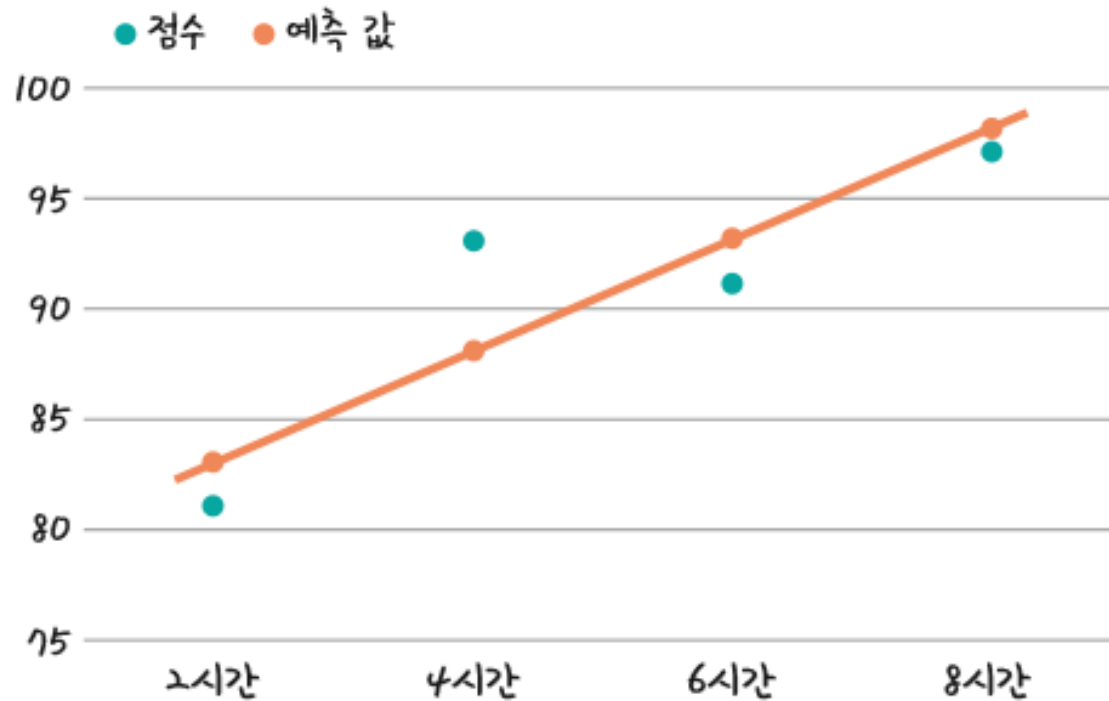
- 좌표 평면에 이 예측 값을 찍어 보자



공부한 시간, 성적, 예측 값을 좌표로 표현

### 3 | 최소 제곱법

- 예측한 점들을 연결해 직선을 그으면 그림 3-4와 같음



오차가 최저가 되는 직선의 완성



### 3 | 최소 제곱법

- 이것이 바로 오차가 가장 적은, 주어진 좌표의 특성을 가장 잘 나타내는 직선  
→ 우리가 원하는 예측 직선임!
- 이 직선에 우리는 다른  $x$  값(공부한 시간)을 집어넣어서 '공부량에 따른 성적을 예측'할 수 있음

## 4 | 코딩으로 확인하는 최소 제곱

- 지금까지 공부한 내용을 코딩으로 구현해 보자
- 먼저 넘파이 라이브러리를 불러와 간단히 np라는 이름으로 사용할 수 있게 설정
- 그리고 앞서 나온 데이터 값을 '리스트' 형식으로 다음과 같이 x와 y로 정의

```
import numpy as np
```

```
x = [2, 4, 6, 8]
```

```
y = [81, 93, 91, 97]
```

## 4 | 코딩으로 확인하는 최소 제곱

- 이제 최소 제곱근 공식에 의해 기울기  $a$ 와  $y$  절편  $b$ 의 값을 구해 보자
- $x$ 의 모든 원소의 평균을 구하는 넘파이 함수는 `mean()`
- `mx`라는 변수에  $x$  원소들의 평균값을, `my`에  $y$  원소들의 평균값을 입력

```
mx = np.mean(x)
my = np.mean(y)
```

## 4 | 코딩으로 확인하는 최소 제곱

- 앞서 살펴본 최소 제곱근 공식 중 분모의 값, 즉 'x의 평균값과 x의 각 원소들의 차를 제곱하라'는 파이썬 명령을 만들 차례
- 해당 식을 옮겨 보면 다음과 같음

$$\sum_{i=1}^n (x - \text{mean}(x))^2$$

- 다음과 같이 divisor라는 변수를 만들어 위 식을 파이썬으로 구현해 저장

```
divisor = sum([(mx - i)**2 for i in x])
```

## 4 | 코딩으로 확인하는 최소 제곱

- 이제 분자에 해당하는 부분을 구해 보자

$$\sum_{i=1}^n (x - \text{mean}(x))(y - \text{mean}(y))$$

- 다음과 같이 새로운 함수를 정의하여 dividend 변수에 분자의 값을 저장

```
def top(x, mx, y, my):  
    d = 0  
    for i in range(len(x)):  
        d += (x[i] - mx) * (y[i] - my)  
    return d  
dividend = top(x, mx, y, my)
```

→ 임의의 변수 d의 초기값을 0으로 설정한 뒤 x의 개수만큼 실행

→ d에 x의 각 원소와 평균의 차, y의 각 원소와 평균의 차를 곱해서 차례로 더하는  
최소 제곱법을 그대로 구현

## 4 | 코딩으로 확인하는 최소 제곱

- 앞에서 구한 분모와 분자를 계산하여 기울기  $a$ 를 구함

```
a = dividend / divisor
```

- $a$ 를 구하고 나면  $y$  절편을 구하는 공식을 이용해  $b$ 를 구할 수 있음

$$b = \text{mean}(y) - (\text{mean}(x) * a)$$

```
b = my - (mx*a)
```

## 4 | 코딩으로 확인하는 최소 제곱

### 선형 회귀 실습

- Linear\_Regression.py

```
import numpy as np

# x 값과 y 값
x=[2, 4, 6, 8]
y=[81, 93, 91, 97]

# x와 y의 평균값
mx = np.mean(x)
my = np.mean(y)
print("x의 평균값:", mx)
print("y의 평균값:", my)

# 기울기 공식의 분모
divisor = sum([(mx - i)**2 for i in x])
```



## 4 | 코딩으로 확인하는 최소 제곱



```
# 기울기 공식의 분자
def top(x, mx, y, my):
    d = 0
    for i in range(len(x)):
        d += (x[i] - mx) * (y[i] - my)
    return d
dividend = top(x, mx, y, my)

print("분모:", divisor)
print("분자:", dividend)

# 기울기와 y 절편 구하기
a = dividend / divisor
b = my - (mx*a)

# 출력으로 확인
print("기울기 a =", a)
print("y 절편 b =", b)
```



## 4 | 코딩으로 확인하는 최소 제곱

- 실행 결과

```
x의 평균값: 5.0  
y의 평균값: 90.5  
분모: 20.0  
분자: 46.0  
기울기 a = 2.3  
y 절편 b = 79.0
```

- 파이썬으로 최소 제곱법을 구현해 기울기 a의 값과 y 절편 b의 값이 각각 2.3과 79임을 구할 수 있었음

## 5 | 평균 제곱근 오차

- 최소 제곱법의 한계
  - '여러 개의 입력( $x$ )'값이 있는 경우 이 공식만으로 처리할수 없다.
- 딥러닝은 대부분 입력 값이 여러 개인 상황에서 이를 해결해야 함!

## 5 | 평균 제곱근 오차

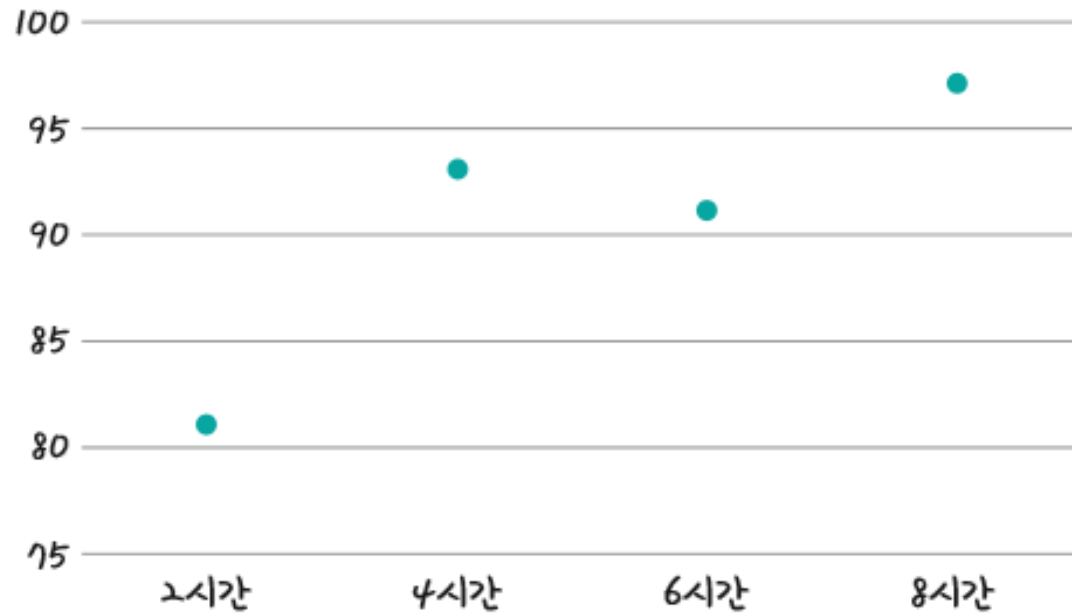
- 여러 개의 입력 값을 계산하는 방법
  - 임의의 선을 그리고 난 후
  - 이 선이 얼마나 잘 그려졌는지를 평가하여
  - 조금씩 수정해 가는 방법을 사용
- 이를 위해 주어진 선의 오차를 평가하는 오차 평가 알고리즘이 필요
  - 가장 많이 사용되는 방법: **평균 제곱근 오차**(root mean square error)

## 6 | 잘못 그은 선 바로잡기

- '일단 그리고 조금씩 수정해 나가기' 방식에 대하여
  - 가설을 하나 세운 뒤 이 값이 주어진 요건을 충족하는지를 판단하여 조금씩 변화를 주고, 이 변화가 긍정적이면 오차가 최소가 될 때까지 이 과정을 계속 반복하는 방법
- 나중에 그린 선이 먼저 그린 선보다 더 좋은지 나쁜지를 판단하기 위해 필요한 것은?
  - 각 선의 오차를 계산할 수 있어야 한다.
  - 이 오차가 작은 쪽으로 바꾸는 알고리즘이 필요하다.

## 6 | 잘못 그은 선 바로잡기

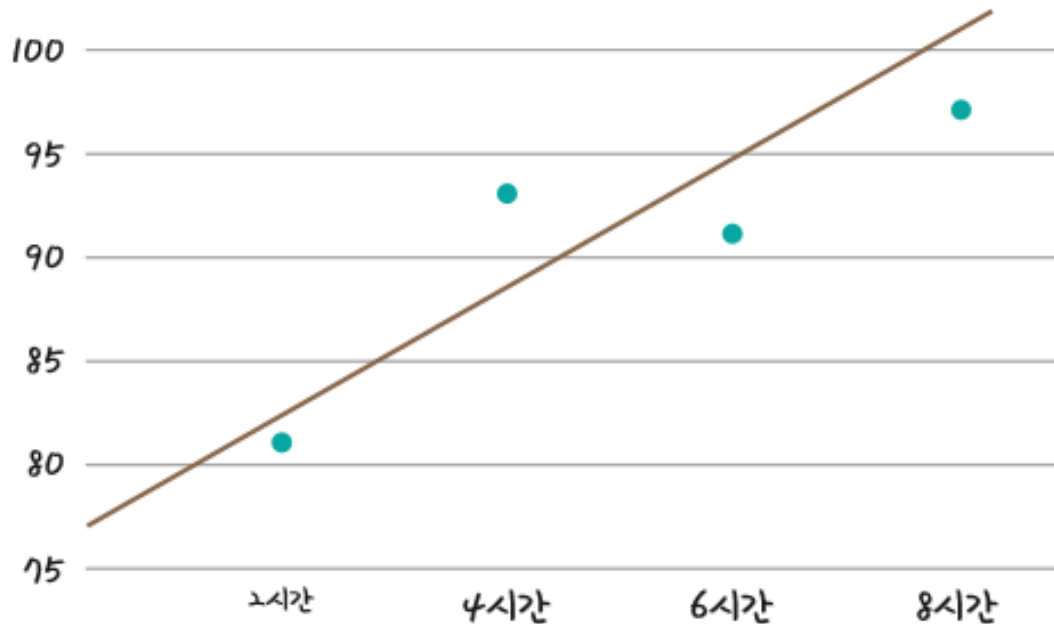
- 오차를 계산하는 방법 알아보자. 먼저 아래의 그래프에서,



공부한 시간과 성적의 관계도

## 6 | 잘못 그은 선 바로잡기

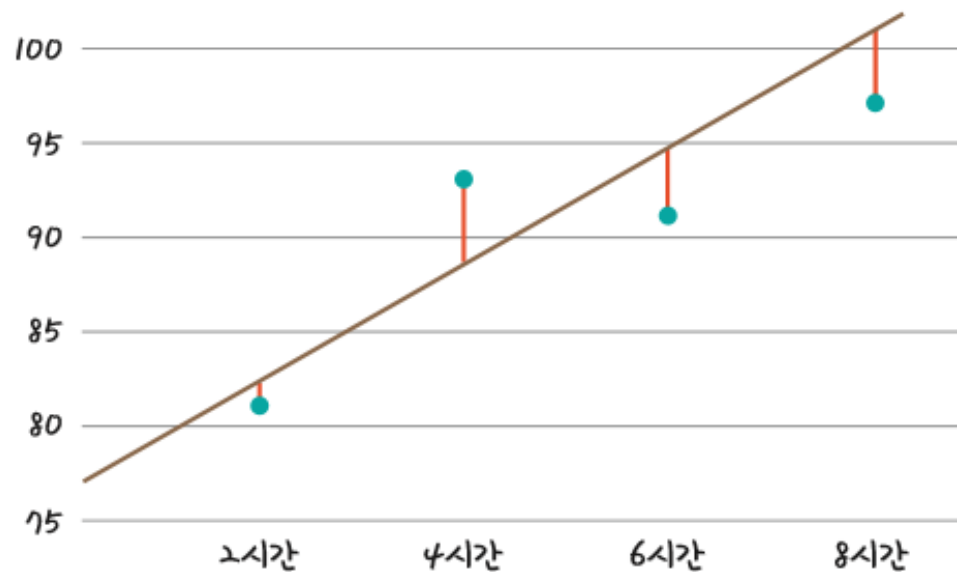
- 대강의 선을 긋기 위해 기울기  $a$ 와  $y$  절편  $b$ 를 임의의 수 3과 76이라고 가정해 본다면  $\rightarrow Y = 3x + 76$ 인 선을 그려야함



임의의 직선 그려보기

## 6 | 잘못 그은 선 바로잡기

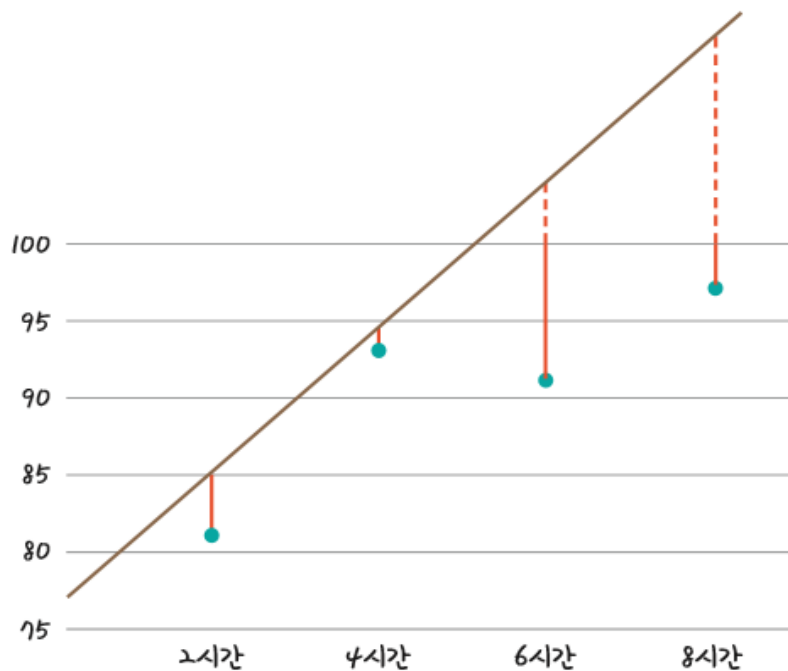
- 임의의 직선이 어느 정도의 오차가 있는지를 확인하려면 각 점과 그래프 사이의 거리를 재면 됨



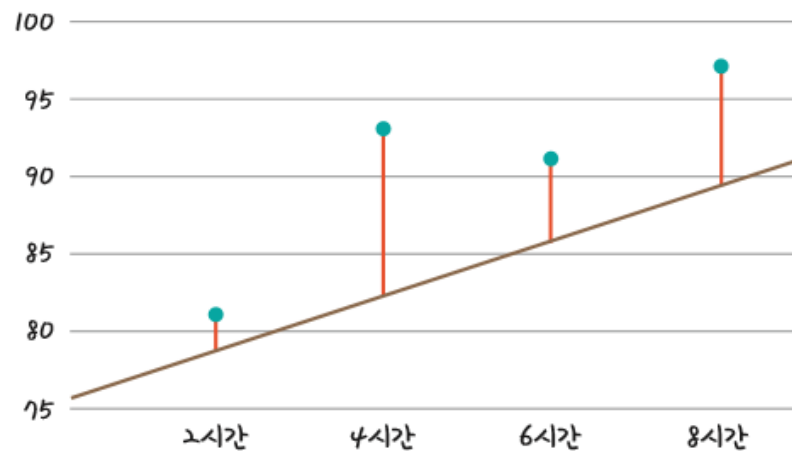
임의의 직선과 실제 값 사이의 거리

- 이 거리들의 합이 작을수록 잘 그어진 직선이고, 이 직선들의 합이 클수록 잘못 그어진 직선이 됨

## 6 | 잘못 그은 선 바로잡기



기울기를 너무 크게 잡았을 때의 오차



기울기를 너무 작게 잡았을 때의 오차

- 그래프의 기울기가 잘못 되었을 수록 빨간색 선의 거리의 합, 즉 오차의 합도 커짐  
→ 만약 기울기가 무한대로 커지면 오차도 무한대로 커지는 상관관계가 있다!



## 6 | 잘못 그은 선 바로잡기

- 빨간색 선의 거리의 합을 실제로 계산해 보자
- 거리는 입력 데이터에 나와 있는  $y$ 의 '실제 값'과  $x$ 를  $y = 3x + 76$ 의 식에 대입해서 나오는 '예측 값'과의 차이를 통해 구할 수 있음
- 예를 들어, 2시간 공부했을 때의 실제 나온 점수(81점)와 그래프  $y = 3x + 76$ 식에  $x = 2$ 를 대입했을 때(82점)의 차이가 곧 오차

→ 오차를 구하는 방정식

$$\text{오차} = \text{실제 값} - \text{예측 값}$$

## 6 | 잘못 그은 선 바로잡기

- 이 식에 주어진 데이터를 대입하여 얻을 수 있는 모든 오차의 값을 정리하면

공부한 시간( $x$ )	2	4	6	8
성적(실제 값, $y$ )	81	93	91	97
예측 값	82	88	94	100
오차	1	-5	3	3

주어진 데이터에서 오차 구하기

## 6 | 잘못 그은 선 바로잡기

- 부호를 없애야 정확한 오차를 구할 수 있음. 따라서 오차의 합을 구할 때는 각 오차의 값을 제곱해 준다

$$\text{오차의 합} = \sum_{i=1}^n (p_i - y_i)^2$$

- 여기서  $i$ 는  $x$ 가 나오는 순서를,  $n$ 은  $x$  원소의 총 개수를 의미
- $p_i$ 는  $x_i$ 에 대응하는 '실제 값'이고  $y_i$ 는  $x_i$ 가 대입되었을 때 직선의 방정식(여기서는  $y = 3x + 76$ )이 만드는 '예측 값'
- 이 식에 의해 오차의 합을 다시 계산하면  $1 + 25 + 9 + 9 = 44$

## 6 | 잘못 그은 선 바로잡기

- 오차의 합을  $n$ 으로 나누면 오차 합의 평균을 구할 수 있음  
→ 평균 제곱 오차(Mean Squared Error, MSE)

$$\text{평균 제곱 오차(MSE)} = \frac{1}{n} \sum_{i=1}^n (p_i - y_i)^2$$

- 이 식은 앞으로 머신러닝과 딥러닝을 공부할 때 자주 등장하는 중요한 식!
- 이 식에 따라 우리가 앞서 그은 임의의 직선은  $44/4 = 11$ 의 평균 제곱 오차를 갖는 직선이라고 말할 수 있음

## 6 | 잘못 그은 선 바로잡기

- 여기에 다시 제공근을 씌워 주면, **평균 제공근 오차**(Root Mean Squared Error, RMSE)라고 함

$$\text{평균 제공근 오차(RMSE)} = \sqrt{\frac{1}{n} \sum_{i=1}^n (p_i - y_i)^2}$$

- 앞서 그은 직선의 평균 제공근 오차는  $\sqrt{11} = 3.3166\cdots$ 이 된다.
- 평균 제공 오차 또는 평균 제공근 오차는 오차를 계산해서 앞선 추론이 잘 되었는지 평가하는 대표적인 공식

## 6 | 잘못 그은 선 바로잡기

- 잘못 그은 선 바로잡기는 곧 '평균 제곱근 오차'의 계산 결과가 가장 작은 선을 찾는 작업
- 선형 회귀란?
  - 임의의 직선을 그어 이에 대한 평균 제곱근 오차를 구하고
  - 이 값을 가장 작게 만들어 주는  $a$ 와  $b$  값을 찾아가는 작업!

## 7 | 코딩으로 확인하는 평균 제곱근 오차

- 평균 제곱근 오차를 파이썬으로 구현해 보자
- 먼저 임의로 정한 기울기  $a$ 와  $y$  절편  $b$ 의 값이 각각 3과 76이라고 할 때 리스트 'ab'를 만들어 여기에 이 값을 저장

```
ab = [3, 76]
```

## 7 | 코딩으로 확인하는 평균 제공근 오차

- 이번에는 'data'라는 리스트를 만들어 공부한 시간과 이에 따른 성적을 각각 짝을 지어 저장
- 그리고 x 리스트와 y 리스트를 만들어 첫 번째 값을 x 리스트에 저장하고 두 번째 값을 y 리스트에 저장

```
data = [[2, 81], [4, 93], [6, 91], [8, 97]]  
x = [i[0] for i in data]  
y = [i[1] for i in data]
```



## 7 | 코딩으로 확인하는 평균 제곱근 오차

- predict()라는 함수를 사용해 일차 방정식  $y = ax + b$ 를 구현

```
def predict(x):  
    return ab[0]*x + ab[1]
```

- 평균 제곱근 공식  $\sqrt{\frac{1}{n} \sum_{i=1}^n (p_i - y_i)^2}$  을 그대로 파이썬 함수로 옮기면 다음과 같음

```
def rmse(p, a):  
    return np.sqrt(((p - a) ** 2).mean())
```

- np.sqrt()은 제곱근을, \*\*2는 제곱을 구하라는 뜻
- mean()은 평균값을 구하라는 뜻
- 예측 값과 실제 값을 각각 rmse()라는 함수의 p와 a 자리에 입력해서 평균 제곱근을 구함

## 7 | 코딩으로 확인하는 평균 제곱근 오차

- 이제 `rmse()` 함수에 데이터를 대입하여 최종값을 구하는 함수 `rmse_val()`을 만들어 보자

```
def rmse_val(predict_result,y):  
    return rmse(np.array(predict_result), np.array(y))
```

- `predict_result`에는 앞서 만든 일차 방정식 함수 `predict()`의 결과값이 들어감
- 이 값과 `y` 값이 각각 예측 값과 실제 값으로 `rmse()` 함수 안에 들어가게 됨

## 7 | 코딩으로 확인하는 평균 제곱근 오차

- 이제 모든  $x$  값을 `predict()` 함수에 대입하여 예측 값을 구하고, 이 예측 값과 실제 값을 통해 최종값을 출력하는 코드를 다음과 같이 작성

```
# 예측 값이 들어갈 빈 리스트를 만든다.  
predict_result = []  
  
# 모든 x 값을 한 번씩 대입하여  
for i in range(len(x)):  
    # 그 결과 predict_result 리스트를 완성한다.  
    predict_result.append(predict(x[i]))  
    print("공부한 시간 = %.f, 실제 점수 = %.f, 예측 점수 = %.f" % (x[i], y[i],  
        predict(x[i])))
```

## 7 | 코딩으로 확인하는 평균 제곱근 오차

### 선형 회귀 실습2

- RMSE.py

```
import numpy as np


# 기울기 a와 y 절편 b
ab = [3, 76]

# x, y의 데이터 값
data = [[2, 81], [4, 93], [6, 91], [8, 97]]
x = [i[0] for i in data]
y = [i[1] for i in data]

#  $y = ax + b$ 에 a와 b 값을 대입하여 결과를 출력하는 함수
def predict(x):
    return ab[0]*x + ab[1]
```



## 7 | 코딩으로 확인하는 평균 제곱근 오차




```
# RMSE 함수
def rmse(p, a):
    return np.sqrt(((p - a) ** 2).mean())

# RMSE 함수를 각 y 값에 대입하여 최종 값을 구하는 함수
def rmse_val(predict_result, y):
    return rmse(np.array(predict_result), np.array(y))

# 예측 값이 들어갈 빈 리스트
predict_result = []

# 모든 x 값을 한 번씩 대입하여
for i in range(len(x)):
    # predict_result 리스트를 완성한다.
    predict_result.append(predict(x[i]))
    print("공부한 시간 = %.f, 실제 점수 = %.f, 예측 점수 = %.f" % (x[i], y[i],
    predict(x[i])))
```



## 7 | 코딩으로 확인하는 평균 제곱근 오차



```
# 최종 RMSE 출력  
print("rmse 최종값: " + str(rmse_val(predict_result,y)))
```

- 실행 결과

```
공부한 시간=2, 실제 점수=81, 예측 점수=82  
공부한 시간=4, 실제 점수=93, 예측 점수=88  
공부한 시간=6, 실제 점수=91, 예측 점수=94  
공부한 시간=8, 실제 점수=97, 예측 점수=100  
rmse 최종값: 3.31662479036
```

## 7 | 코딩으로 확인하는 평균 제곱근 오차

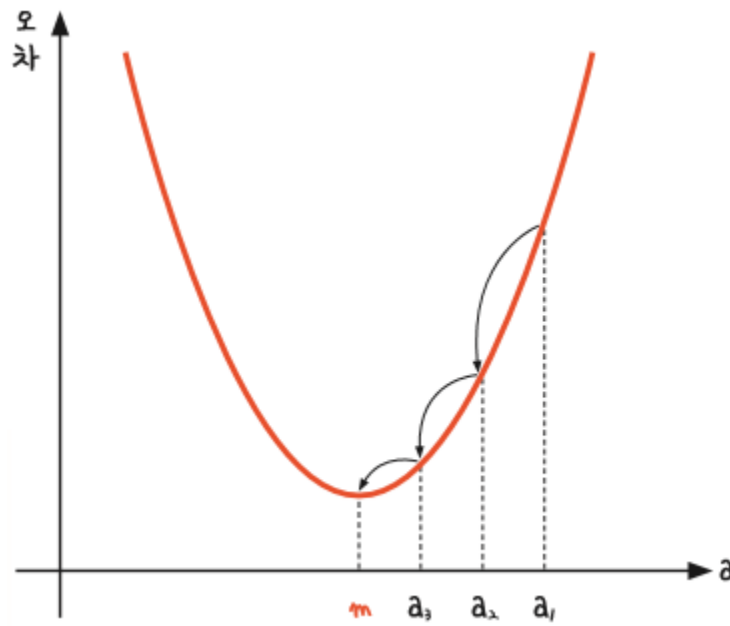
- 이를 통해 우리가 처음 가정한  $a = 3$ ,  $b = 76$ 은 오차가 약 3.3166이라는 것을 알게 됨
- 이제 남은 것은 이 오차를 줄이면서 새로운 선을 긋는 것
- 이를 위해서는  $a$ 와  $b$ 의 값을 적절히 조절하면서 오차의 변화를 살펴보고, 그 오차가 최소화되는  $a$ 와  $b$ 의 값을 구해야 함 (→ 다음장, '미분'의 필요성)

# 오차 수정하기: 경사 하강법

- 1 | 미분의 개념
- 2 | 경사 하강법의 개요
- 3 | 학습률
- 4 | 코딩으로 확인하는 경사 하강법
- 5 | 다중 선형 회귀란?
- 6 | 코딩으로 확인하는 다중 선형 회귀



- 기울기  $a$ 를 무한대로 키우면 오차도 무한대로 커지고  
 $a$ 를 무한대로 작게 해도 역시 오차도 무한대로 작아지는 이러한 관계는  
“이차 함수 그래프”로 표현할 수 있음



기울기  $a$ 와 오차와의 관계: 적절한 기울기를 찾았을 때 오차가 최소화된다.

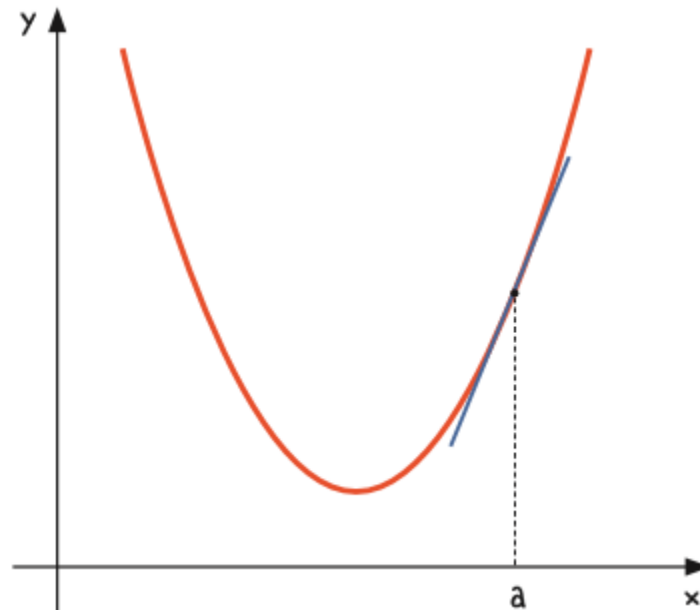
- 우리의 목표 : 오차가 가장 작은 점을 찾는것!!
- 미분 기울기를 이용해 오차를 비교하여 가장 작은 방향으로 이동시키는 방법  
→ 경사 하강법(gradient decent)

## 1 | 미분의 개념

- 순간 변화율의 의미
  - $a$ 가 변화량이 0에 가까울 만큼 아주 미세하게 변화했다면,
  - $y$  값의 변화 역시 아주 미세해서 0에 가까울 것
- 변화가 있긴 하지만, 그 움직임이 너무 미세하면?
  - 어느 쪽으로 '움직이려고 시도했다'는 정도의 느낌만 있을 뿐.
  - 이 느낌을 수학적으로 이름 붙인 것이 바로 '순간 변화율'

## 1 | 미분의 개념

- 순간 변화율은 '어느 쪽'이라는 방향성을 지니고 있으므로 이 방향에 맞추어 직선을 그릴 수가 있음
- 이 선이 바로 이 점에서의 '기울기'라고 불리는 접선



a에서의 순간 변화율은 곧 기울기다!

## 1 | 미분의 개념

- 미분이란?

- x 값이 아주 미세하게 움직일 때의 y 변화량을 구한 뒤,
- 이를 x의 변화량으로 나누는 과정
- 한 점에서의 순간 기울기

- “함수  $f(x)$ 를 미분하라”는  $\frac{d}{dx}f(x)$  라고 표기함.

$$\frac{d}{dx}f(x) = \lim_{\Delta x \rightarrow 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

① 함수  $f(x)$ 를  $x$ 로 미분하라는 것은

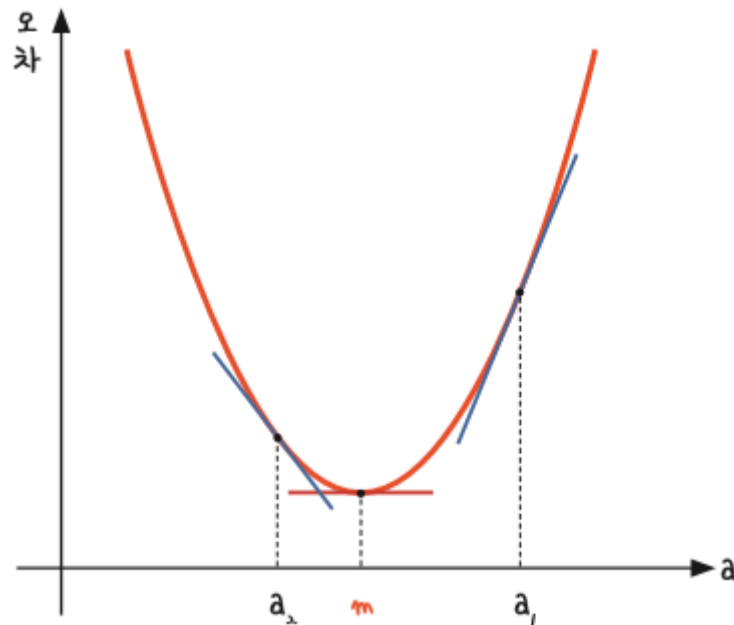
②  $x$ 의 변화량이 0에 가까울 만큼 작을 때

③  $y$  변화량의 차이를

④  $x$  변화량으로 나눈 값 (= 순간 변화율)을 구하라는 뜻

## 2 | 경사 하강법의 개요

- $y = x^2$  그래프에서  $x$ 에 다음 과 같이  $a_1, a_2$  그리고  $m$ 을 대입하여 그 자리에서 미분하면 그림 4-3처럼 각 점에서의 순간 기울기가 그려짐



순간 기울기가 0인 점이 곧 우리가 찾는 최솟값이다.

## 2 | 경사 하강법의 개요

- 여기서 눈여겨 봐야 할 것? → 꼭짓점의 기울기
- 꼭지점의 기울기는  $x$ 축과 평행한 선이 됨. (즉, 기울기가 0!)
- 따라서 우리가 할 일은 '미분 값이 0인 지점'을 찾는 것!

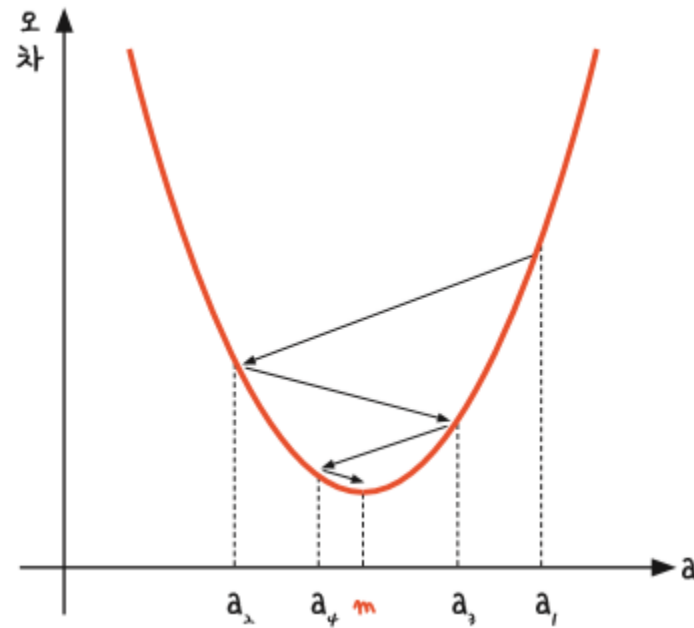
## 2 | 경사 하강법의 개요

- 이를 위해서 다음과 같은 과정을 거침
  - 1)  $a_1$ 에서 미분을 구한다.
  - 2) 구해진 기울기의 반대 방향 얼마간 이동시킨  $a_2$ 에서 미분을 구한다.
  - 3)  $a_3$ 에서 미분을 구한다.
  - 4) 3의 값이 0이 아니면  $a_2$ 에서 2~3번 과정을 반복한다.



## 2 | 경사 하강법의 개요

- 그러면 그림 4-4처럼 이동 결과가 한 점으로 수렴함

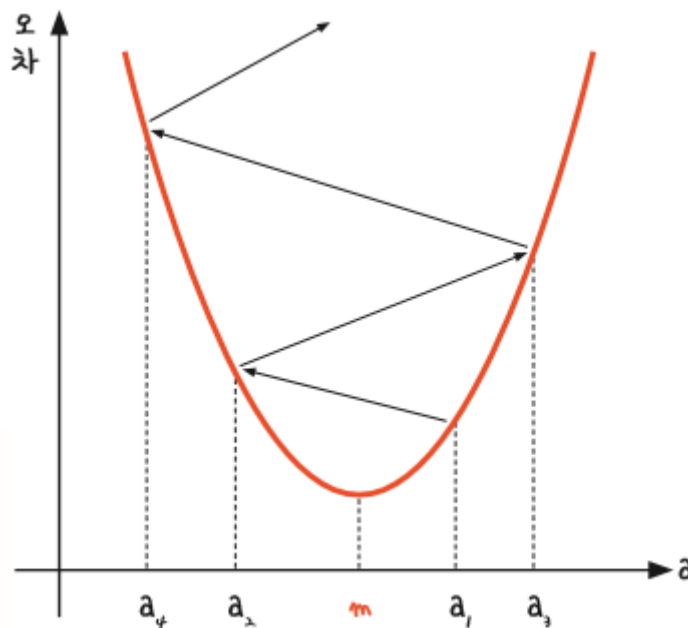


최솟점  $m$ 을 찾아가는 과정

- 경사 하강법은 이렇게 반복적으로 기울기  $a$ 를 변화시켜서  $m$ 의 값을 찾아내는 방법

### 3 | 학습률

- 기울기의 부호를 바꿔 이동시킬 때 적절한 거리를 찾지 못해 너무 멀리 이동시키면  $a$  값이 한 점으로 모이지 않고 위로 치솟아 버림
- 어느 만큼 이동시킬지를 정해주는 것 : **학습률**(learning rate)



학습률을 너무 크게 잡으면 한 점으로 수렴하지 않고 발산한다.

## 4 | 코딩으로 확인하는 경사 하강법

- 코딩을 통해 경사 하강법을 실제로 적용하여 a와 b의 값을 구해 보자

```
import tensorflow as tf
```

- 텐서플로 라이브러리를 불러옴
- 텐서플로는 구글이 오픈 소스 라이선스로 공개한 딥러닝 전문 라이브러리

## 4 | 코딩으로 확인하는 경사 하강법

- 데이터 입력과  $x$ ,  $y$ 를 지정 및 학습률이 추가

```
data = [[2, 81], [4, 93], [6, 91], [8, 97]]  
x_data = [x_row[0] for x_row in data]  
y_data = [y_row[1] for y_row in data]  
  
learning_rate = 0.1
```

## 4 | 코딩으로 확인하는 경사 하강법

- 이제 기울기  $a$ 와  $y$  절편  $b$ 의 값을 임의로 정한다.
  - 단, 기울기가 너무 커지거나 작아지면 실행 시간이 불필요하게 늘어나므로
  - 기울기는 0~10 사이에서,  $y$  절편은 0~100 사이에서 임의의 값을 얻게끔 함

```
a = tf.Variable(tf.random_uniform([1], 0, 10, dtype = tf.float64, seed = 0))  
b = tf.Variable(tf.random_uniform([1], 0, 100, dtype = tf.float64, seed = 0))
```

- Tensorflow 라이브러리를 `tf`라는 약어로 불러오고 변수의 값을 정할 때는 `Variable()` 함수를 이용
- `random_uniform()`은 임의의 수를 생성해 주는 함수로, 여기에 몇 개의 값을 뽑아낼지와 최솟값 및 최댓값을 적어 줌
- 예를 들어 `random_uniform([1], 0, 10,...)`은 0에서 10 사이에서 임의의 수 1개를 만들라는 뜻
- 데이터 형식은 실수형(`float64`)으로 지정하고, 실행 시 같은 값이 나올 수 있게 `seed` 값을 설정해 주었음

## 4 | 코딩으로 확인하는 경사 하강법

- 이제 일차 방정식  $ax + b$ 의 식을 구현해 보자

```
y = a * x_data + b
```

- 이어서 평균 제곱근 오차의 식을 구현해 보자
- 텐서플로를 이용해 평균 제곱근 오차를 다음과 같이 구현할 수 있음

```
rmse = tf.sqrt(tf.reduce_mean(tf.square( y - y_data )))
```

## 4 | 코딩으로 확인하는 경사 하강법

- 이제 경사 하강법을 실행할 순서!
- 텐서플로의 GradientDescentOptimizer() 함수를 이용하여
- 경사 하강법의 결과를 gradient\_decent에 할당시킨다.

```
gradient_decent = tf.train.GradientDescentOptimizer(learning_rate). minimize(rmse)
```

- 앞서 지정한 learning\_rate와 평균 제곱근 오차를 통해 구한 rmse를 사용

## 4 | 코딩으로 확인하는 경사 하강법

- 이제 텐서플로를 실행시키고, 결과값을 출력하는 부분

```
with tf.Session() as sess:
    # 변수 초기화
    sess.run(tf.global_variables_initializer())
    # 2001번 실행(0번째를 포함하므로)
    for step in range(2001):
        sess.run(gradient_decent)
        # 100번마다 결과 출력
        if step % 100 == 0:
            print("Epoch: %.f, RMSE = %.04f, 기울기 a = %.4f, y 절편 b = %.4f" %
                  (step, sess.run(rmse), sess.run(a), sess.run(b)))
```



## 4 | 코딩으로 확인하는 경사 하강법

- session 함수를 이용해 구동에 필요한 리소스를 컴퓨터에 할당하고 이를 실행시킬 준비를 함
- Session을 통해 구현될 함수를 텐서플로에서는 '그래프'라고 부르며, Session이 할당 되면 session.run('그래프명')의 형식으로 해당 함수를 구동시킴
- global\_variables\_initializer()는 변수를 초기화하는 함수.
- 앞서 만든 gradient\_descent를 총 필요한 수만큼 반복하여 실행
- 그리고 100번마다 RMSE, 기울기, y 절편을 출력하게 함

## 4 | 코딩으로 확인하는 경사 하강법

### 경사 하강법 실습

- Gradient-Descent.py

```
import tensorflow as tf

# x, y의 데이터 값
data = [[2, 81], [4, 93], [6, 91], [8, 97]]
x_data = [x_row[0] for x_row in data]
y_data = [y_row[1] for y_row in data]

# 기울기 a와 y 절편 b의 값을 임의로 정한다.
# 단, 기울기의 범위는 0 ~ 10 사이이며, y 절편은 0 ~ 100 사이에서 변하게 한다.
a = tf.Variable(tf.random_uniform([1], 0, 10, dtype = tf.float64, seed = 0))
b = tf.Variable(tf.random_uniform([1], 0, 100, dtype = tf.float64, seed = 0))

# y에 대한 일차 방정식 ax+b의 식을 세운다.
y = a * x_data + b
```



## 4 | 코딩으로 확인하는 경사 하강법



```
# 텐서플로 RMSE 함수
rmse = tf.sqrt(tf.reduce_mean(tf.square( y - y_data )))

# 학습률 값
learning_rate = 0.1

# RMSE 값을 최소로 하는 값 찾기
gradient_decent = tf.train.GradientDescentOptimizer(learning_rate).minimize(rmse)

# 텐서플로를 이용한 학습
with tf.Session() as sess:
    # 변수 초기화
    sess.run(tf.global_variables_initializer())
    # 2001번 실행(0번째를 포함하므로)
    for step in range(2001):
        sess.run(gradient_decent)
        # 100번마다 결과 출력
        if step % 100 == 0:
            print("Epoch: %.f, RMSE = %.04f, 기울기 a = %.4f, y 절편 b = %.4f" %
                  (step, sess.run(rmse), sess.run(a), sess.run(b)))
```

## 4 | 코딩으로 확인하는 경사 하강법

- 실행 결과

```
Epoch: 0, RMSE = 28.6853, 기울기 a = 7.2507, y 절편 b = 80.5525
Epoch: 100, RMSE = 2.8838, 기울기 a = 2.2473, y 절편 b = 79.3146
Epoch: 200, RMSE = 2.8815, 기울기 a = 2.2774, y 절편 b = 79.1348
Epoch: 300, RMSE = 2.8811, 기울기 a = 2.2903, y 절편 b = 79.0578
Epoch: 400, RMSE = 2.8810, 기울기 a = 2.2959, y 절편 b = 79.0247
Epoch: 500, RMSE = 2.8810, 기울기 a = 2.2982, y 절편 b = 79.0106
Epoch: 600, RMSE = 2.8810, 기울기 a = 2.2992, y 절편 b = 79.0045
Epoch: 700, RMSE = 2.8810, 기울기 a = 2.2997, y 절편 b = 79.0019
Epoch: 800, RMSE = 2.8810, 기울기 a = 2.2999, y 절편 b = 79.0008
Epoch: 900, RMSE = 2.8810, 기울기 a = 2.2999, y 절편 b = 79.0004
Epoch: 1000, RMSE = 2.8810, 기울기 a = 2.3000, y 절편 b = 79.0002
Epoch: 1100, RMSE = 2.8810, 기울기 a = 2.3000, y 절편 b = 79.0001
Epoch: 1200, RMSE = 2.8810, 기울기 a = 2.3000, y 절편 b = 79.0000
Epoch: 1300, RMSE = 2.8810, 기울기 a = 2.3000, y 절편 b = 79.0000
Epoch: 1400, RMSE = 2.8810, 기울기 a = 2.3000, y 절편 b = 79.0000
```



## 4 | 코딩으로 확인하는 경사 하강법



```
Epoch: 1500, RMSE = 2.8810, 기울기 a = 2.3000, y 절편 b = 79.0000
Epoch: 1600, RMSE = 2.8810, 기울기 a = 2.3000, y 절편 b = 79.0000
Epoch: 1700, RMSE = 2.8810, 기울기 a = 2.3000, y 절편 b = 79.0000
Epoch: 1800, RMSE = 2.8810, 기울기 a = 2.3000, y 절편 b = 79.0000
Epoch: 1900, RMSE = 2.8810, 기울기 a = 2.3000, y 절편 b = 79.0000
Epoch: 2000, RMSE = 2.8810, 기울기 a = 2.3000, y 절편 b = 79.0000
```

## 4 | 코딩으로 확인하는 경사 하강법

- 에포크(Epoch) : 입력 값에 대해 몇 번이나 반복하여 실험했는지를 나타냄
- 평균 제곱근 오차(RMSE)의 변화와 기울기  $a$ 가 2.3에 수렴하는 것 그리고  $y$  절편  $b$ 가 79에 수렴하는 과정을 볼 수 있음
- 기울기 2.3과  $y$  절편 79는 앞서 최소 제곱법을 통해 미리 확인한 정답 값과 같다!

## 5 | 다중 선형 회귀란?

- 4시간 공부한 친구는 88점을 예측했는데 이보다 좋은 93점을 받았고, 6시간 공부한 친구는 93점을 받을 것으로 예측했지만 91점을 받았음  
→ 예측과 실제값에 차이가 있다.
- 차이가 생기는 이유는 공부한 시간 이외의 다른 요소가 성적에 영향을 끼쳤기 때문
- 더 정확한 예측을 하려면 추가 정보를 입력해야 함
- 정보를 추가해 새로운 예측 값을 구하려면 변수의 개수를 늘려 '다중 선형 회귀'를 만들어 주어야 한다

## 5 | 다중 선형 회귀란?

- 예를 들어, 일주일 동안 받는 과외 수업 횟수를 조사해서 이를 기록해 보면,  
→ 두 개의 독립 변수  $x_1$ 과  $x_2$ 가 생긴 것

공부한 시간( $x_1$ )	2	4	6	8
과외 수업 횟수( $x_2$ )	0	4	2	3
성적( $y$ )	81	93	91	97

공부한 시간, 과외 수업 횟수에 따른 성적 데이터



## 5 | 다중 선형 회귀란?

- 이를 사용해 종속 변수  $y$ 를 만들 경우 기울기를 두 개 구해야 하므로 다음과 같은 식이 나옴

$$y = a_1x_1 + a_2x_2 + b$$

- 그러면 두 기울기  $a_1$ 과  $a_2$ 는 각각 어떻게 구할 수 있을까?
- 앞에서 배운 경사 하강법을 그대로 적용하면 됨

## 6 | 코딩으로 확인하는 다중 선형 회귀

- 지금까지 배운 내용을 토대로 다중 선형 회귀를 작성해 보자
- 텐서플로를 불러온 뒤  $x$ 와  $y$ 의 값을 지정하는 과정은 동일함
- 다만, 이번에는  $x_1$ 과  $x_2$ 라는 두 개의 독립 변수 리스트를 만들어 줌

```
import tensorflow as tf

# x1, x2, y의 데이터 값

data = [[2, 0, 81], [4, 4, 93], [6, 2, 91], [8, 3, 97]]
x1 = [x_row1[0] for x_row1 in data]
x2 = [x_row2[1] for x_row2 in data] # 새로 추가되는 값
y_data = [y_row[2] for y_row in data]
```

## 6 | 코딩으로 확인하는 다중 선형 회귀

- 이제 앞서 기울기의 값을 구하는 방식 그대로 또 하나의 기울기  $a_2$ 를 구함

```
a1 = tf.Variable(tf.random_uniform([1], 0, 10, dtype=tf.float64, seed=0))  
a2 = tf.Variable(tf.random_uniform([1], 0, 10, dtype=tf.float64, seed=0)) # 새로 추가  
    되는 값  
b = tf.Variable(tf.random_uniform([1], 0, 100, dtype=tf.float64, seed=0))
```

- 이제 새로운 방정식  $y = a_1x_1 + a_2x_2 + b$ 에 맞춰 다음과 같이 식을 세움

```
y = a1 * x1 + a2 * x2+ b
```

## 6 | 코딩으로 확인하는 다중 선형 회귀

- 나머지 라인은 앞서 배운 선형 회귀와 같음
- 결과를 출력하는 부분만 기울기가 두 개 나올 수 있게 수정

```
print("Epoch: %.f, RMSE = %.04f, 기울기 a1 = %.4f, 기울기 a2 = %.4f, y 절편 b  
= %.4f" % (step, sess.run(rmse), sess.run(a1), sess.run(a2), sess.run(b)))
```

## 6 | 코딩으로 확인하는 다중 선형 회귀

### 다중 선형 회귀 실습

- Multi-Linear-Regression.py

```
import tensorflow as tf

# x1, x2, y의 데이터 값

data = [[2, 0, 81], [4, 4, 93], [6, 2, 91], [8, 3, 97]]
x1 = [x_row1[0] for x_row1 in data]
x2 = [x_row2[1] for x_row2 in data] # 새로 추가되는 값
y_data = [y_row[2] for y_row in data]

# 기울기 a와 y 절편 b의 값을 임의로 정한다.
# 단, 기울기의 범위는 0 ~ 10 사이이며, y 절편은 0 ~ 100 사이에서 변하게 한다.
a1 = tf.Variable(tf.random_uniform([1], 0, 10, dtype=tf.float64, seed=0))
a2 = tf.Variable(tf.random_uniform([1], 0, 10, dtype=tf.float64, seed=0)) # 새로 추가
되는 값
b = tf.Variable(tf.random_uniform([1], 0, 100, dtype=tf.float64, seed=0))
```



## 6 | 코딩으로 확인하는 다중 선형 회귀



# 새로운 방정식

$y = a_1 * x_1 + a_2 * x_2 + b$

# 텐서플로 RMSE 함수

`rmse = tf.sqrt(tf.reduce_mean(tf.square( y - y_data )))`

# 학습률 값

`learning_rate = 0.1`

# RMSE 값을 최소로 하는 값 찾기

`gradient_decent = tf.train.GradientDescentOptimizer(learning_rate).minimize(rmse)`

# 학습이 진행되는 부분

`with tf.Session() as sess:`

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





## 6 | 코딩으로 확인하는 다중 선형 회귀

- 실행 결과

```
Epoch: 0, RMSE = 49.1842, 기울기 a1 = 7.5270, 기울기 a2 = 7.8160, y 절편 b = 80.5980
Epoch: 100, RMSE = 1.8368, 기울기 a1 = 1.1306, 기울기 a2 = 2.1316, y 절편 b = 78.5119
Epoch: 200, RMSE = 1.8370, 기울기 a1 = 1.1879, 기울기 a2 = 2.1487, y 절편 b = 78.1057
Epoch: 300, RMSE = 1.8370, 기울기 a1 = 1.2122, 기울기 a2 = 2.1571, y 절편 b = 77.9352
Epoch: 400, RMSE = 1.8370, 기울기 a1 = 1.2226, 기울기 a2 = 2.1607, y 절편 b = 77.8636
Epoch: 500, RMSE = 1.8370, 기울기 a1 = 1.2269, 기울기 a2 = 2.1622, y 절편 b = 77.8335
Epoch: 600, RMSE = 1.8370, 기울기 a1 = 1.2288, 기울기 a2 = 2.1628, y 절편 b = 77.8208
Epoch: 700, RMSE = 1.8370, 기울기 a1 = 1.2295, 기울기 a2 = 2.1631, y 절편 b = 77.8155
Epoch: 800, RMSE = 1.8370, 기울기 a1 = 1.2299, 기울기 a2 = 2.1632, y 절편 b = 77.8133
Epoch: 900, RMSE = 1.8370, 기울기 a1 = 1.2300, 기울기 a2 = 2.1632, y 절편 b = 77.8124
```

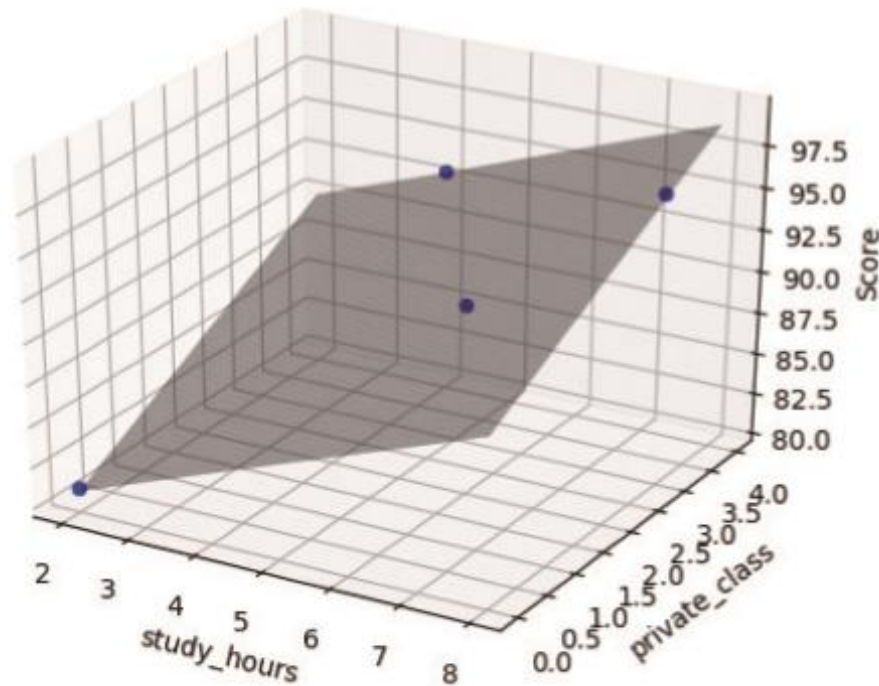
(중략)

```
Epoch: 2000, RMSE = 1.8370, 기울기 a1 = 1.2301, 기울기 a2 = 2.1633, y 절편 b = 77.8117
```



## 6 | 코딩으로 확인하는 다중 선형 회귀

- 다중 선형 회귀 문제에서의 기울기  $a$ 와  $y$  절편  $b$ 의 값을 구했음
- 이를 그래프로 표현하면 다음과 같음



다중 선형 회귀의 그래프: 차원이 하나 더 늘어난 모습

## 6 | 코딩으로 확인하는 다중 선형 회귀

- 1차원 예측 직선이 3차원 '예측 평면'으로 바뀌었음
- 과외 수업 횟수라는 새로운 변수가 추가되면서 1차원 직선에서만 움직이던 예측 결과가 더 넓은 평면 범위 안에서 움직이게 되었음 → 좀 더 정밀한 예측을 할 수 있게 됨

# 참 거짓 판단 장치: 로지스틱 회귀

- 1 | 로지스틱 회귀의 정의
- 2 | 시그모이드 함수
- 3 | 오차 공식
- 4 | 로그 함수
- 5 | 코딩으로 확인하는 로지스틱 회귀
- 6 | 여러 입력 값을 갖는 로지스틱 회귀
- 7 | 실제 값 적용하기
- 8 | 로지스틱 회귀에서 퍼셉트론으로

“예, 아니오로만 대답하세요!”

- 법정 드라마나 영화에서 검사가 피고인을 다그치는 장면의 흔한 대사
  - 때로 할 말이 많아도 예 혹은 아니오로만 대답해야 할 때가 있음
- 이와 같은 상황이 딥러닝에서도 끊임없이 일어남

- 전달받은 정보를 놓고 참과 거짓 중에 하나를 판단해 다음 단계로 넘기는 장치들이 딥러닝 내부에서 쉬지 않고 작동한다.
- 딥러닝을 수행한다는 것은 겉으로 드러나지 않는 '미니 판단 장치'들을 이용해서 복잡한 연산을 해낸 끝에 최적의 예측 값을 내놓는 작업!



- 이렇게 참과 거짓 중에 하나를 내놓는 과정은 **로지스틱 회귀(logistic regression)**의 원리를 거쳐 이루어짐
- 참인지 거짓인지를 구분하는 로지스틱 회귀의 원리를 이용해 '참, 거짓 미니 판단 장치'를 만들어 주어진 입력 값의 특징을 추출함
- 이를 저장해서 '모델(model)'을 만듦
- 그런 다음 누군가 비슷한 질문을 하면 지금까지 만들어 놓은 이 모델을 꺼내어 답을 함
- 이것이 바로 딥러닝의 동작 원리!

## 1 | 로지스틱 회귀의 정의

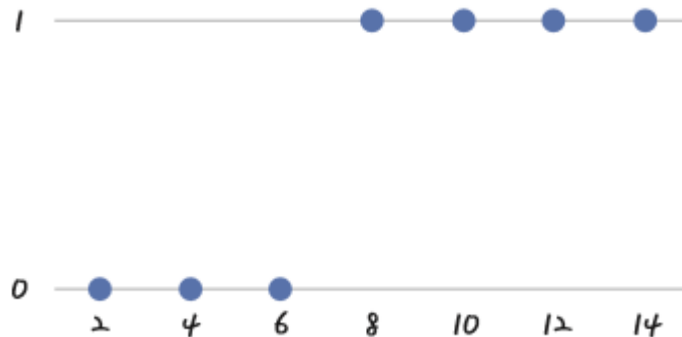
- 합격과 불합격만 발표되는 시험이 있다고 할때, 공부한 시간에 따른 합격 여부를 조사해 보면 다음과 같다.

공부한 시간	2	4	6	8	10	12	14
합격 여부	불합격	불합격	불합격	합격	합격	합격	합격

공부한 시간에 따른 합격 여부

## 1 | 로지스틱 회귀의 정의

- 합격을 1, 불합격을 0이라 하고, 이를 좌표 평면에 표현하면 다음과 같음



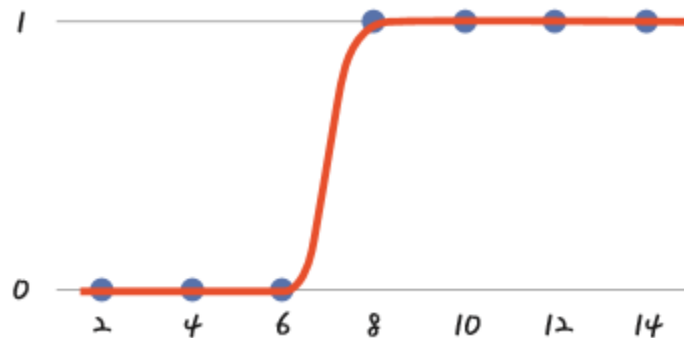
합격과 불합격만 있을 때의 좌표 표현

- 앞장에서 배운 대로 선을 그어 이 점의 특성을 잘 나타내는 일차 방정식을 만들 수 있을까?
- 이 점들은 1과 0 사이의 값이 없으므로 직선으로 그리기가 어려움



## 1 | 로지스틱 회귀의 정의

- 점들의 특성을 정확하게 담아내려면 직선이 아니라 다음과 같이 S자 형태여야 함



각 점의 특성을 담은 선을 그었을 때

- 로지스틱 회귀는 선형 회귀와 마찬가지로 적절한 선을 그려가는 과정
- 다만, 직선이 아니라, 참(1)과 거짓(0) 사이를 구분하는 S자 형태의 선을 그어 주는 작업

## 2 | 시그모이드 함수

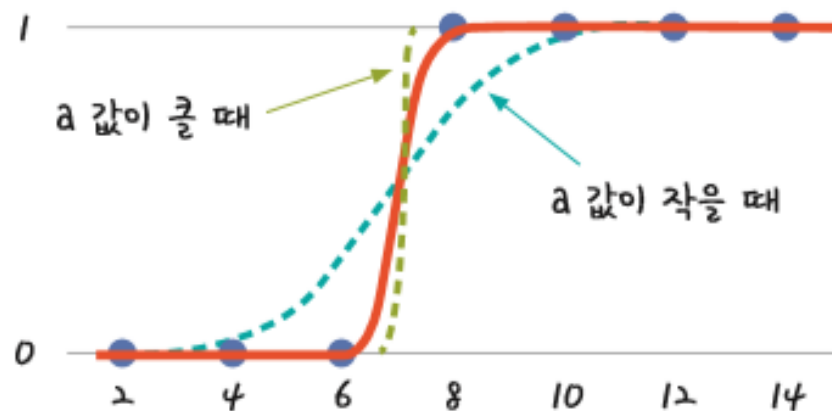
- S자 형태로 그래프가 그려지는 함수가 있다! → 시그모이드 함수(sigmoid function)
- 시그모이드 함수를 나타내는 방정식

$$y = \frac{1}{1 + e^{(ax+b)}}$$

- 여기서 e는 자연 상수라고 불리는 무리수로 값은 2.71828...
- 파이( $\pi$ )처럼 수학에서 중요하게 사용되는 상수로 고정된 값이므로 우리가 따로 구해야 하는 값은 아님
- 우리가 구해야 하는 값은 결국  $ax + b$

## 2 | 시그모이드 함수

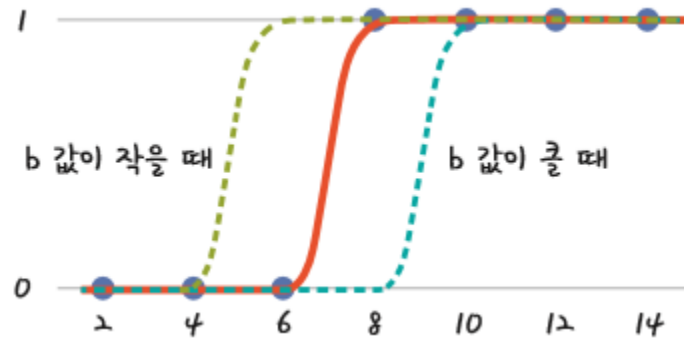
- 선형 회귀에서 우리가 구해야 하는 것이  $a$ 와  $b$ 였듯이 여기서도 마찬가지
- 앞서 구한 직선의 방정식과는 다르게 여기에서  $a$ 와  $b$ 는 어떤 의미를 지니고 있을까?
- 먼저  $a$ 는 그래프의 경사도를 결정
- 그림과 같이  $a$  값이 커지면 경사가 커지고  $a$  값이 작아지면 경사가 작아짐



$a$  값이 클 때와 작을 때의 그래프 변화

## 2 | 시그모이드 함수

- $b$ 는 그래프의 좌우 이동을 의미
- 그림과 같이  $b$  값이 크고 작아짐에 따라 그래프가 이동함

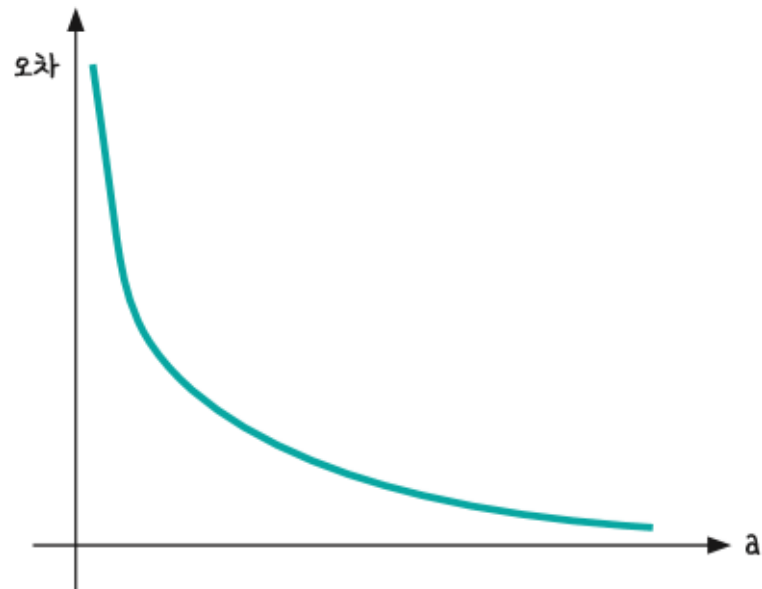


$b$  값이 클 때와 작을 때의 그래프 변화

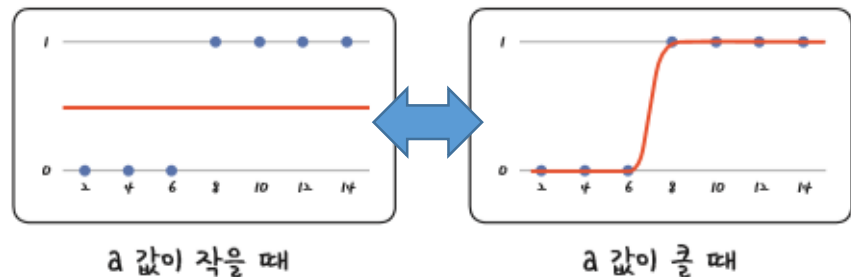
- 따라서  $a$ 와  $b$ 의 값이 클수록 오차가 생김

## 2 | 시그모이드 함수

- $a$  값이 크고 작아짐에 따라 오차는 다음과 같이 변함
- $a$  값이 작아지면 오차는 무한대로 커지지만,  $a$  값이 커진다고 해서 오차가 무한대로 커지지는 않음

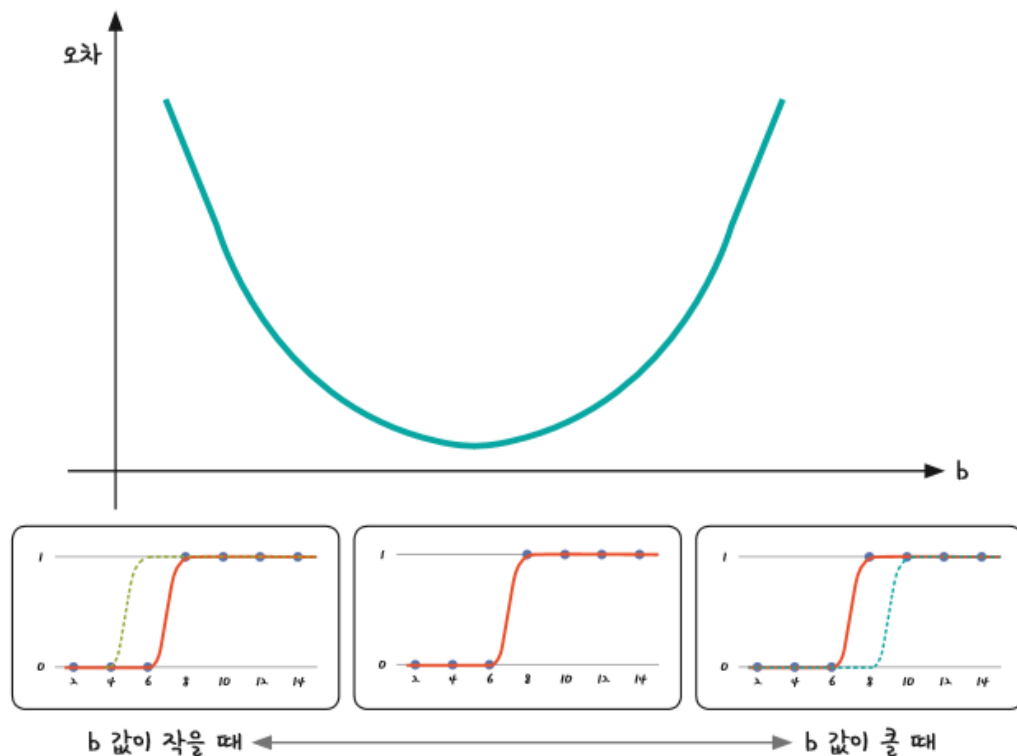


$a$ 와 오차와의 관계:  $a$ 가 작아질수록 오차는 무한대로 커지지만,  $a$ 가 커진다고 해서 오차가 무한대로 커지지는 않는다.



## 2 | 시그모이드 함수

- $b$  값에 따른 오차의 그래프는 그림과 같음
- $b$  값은 너무 크거나 작을 경우 오차가 무한대로 커지므로 앞서와 마찬가지로 이차 함수 그래프로 표현할 수 있음

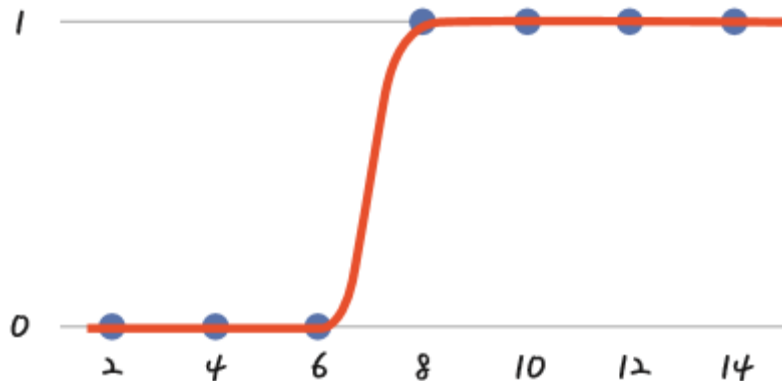


$b$ 와 오차와의 관계:  $a$  값이 너무 작아지거나 커지면 오차도 무한대로 커진다.

### 3 | 오차 공식

- 이제 우리에게 주어진 과제는 또다시  $a$ 와  $b$ 의 값을 구하는 것임
  - 시그모이드 함수에서  $a$ 와  $b$ 의 값을 어떻게 구해야 할까?
  - 답은 역시 **경사 하강법**
- 
- 그런데 경사 하강법은 먼저 오차를 구한 다음 오차가 작은 쪽으로 이동시키는 방법
  - 그렇다면 이번에도 예측 값과 실제 값의 차이, 즉 **오차**를 구하는 공식이 필요함

### 3 | 오차 공식



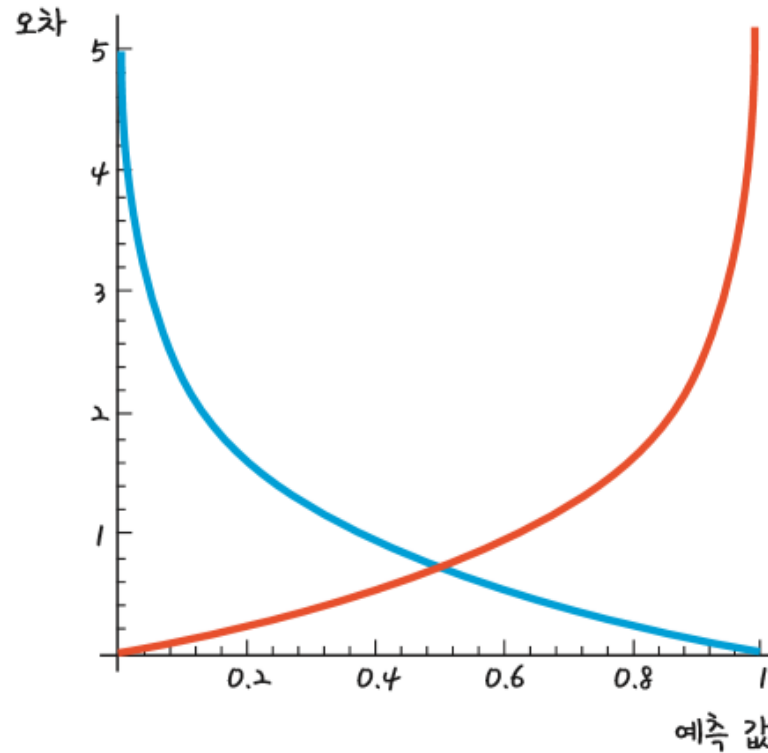
시그모이드 함수 그래프

- 시그모이드 함수의 특징은  $y$  값이 0과 1 사이라는 것
- 따라서 실제 값이 1일 때 예측 값이 0에 가까워지면 오차가 커져야 함
- 반대로, 실제 값이 0일 때 예측 값이 1에 가까워지는 경우에도 오차는 커져야 함
- 이를 공식으로 만들 수 있게 해 주는 함수: **로그 함수**



## 4 | 로그 함수

- 아래와 같이  $y = 0.5$ 에 대칭하는 두 개의 로그 함수를 그려보자



실제 값이 1일 때(파란색)와 0일 때(빨간색) 로그 함수 그래프

## 4 | 로그 함수

- 파란색 선은 실제 값이 1일 때 사용할 수 있는 그래프
- 예측 값이 1일 때 오차가 0이고, 반대로 예측 값이 0에 가까울수록 오차는 커짐
- 빨간색 선은 반대로 실제 값이 0일 때 사용할 수 있는 함수
- 예측 값이 0일 때 오차가 없고, 1에 가까워질수록 오차가 매우 커짐

## 4 | 로그 함수

- 앞의 파란색과 빨간색 그래프의 식은 각각  $-\log h$ 와  $-\log(1 - h)$
- $y$ 의 실제 값이 1일 때  $-\log h$  그래프를 쓰고, 0일 때  $-\log(1 - h)$  그래프를 써야 함
- 이는 다음과 같은 방법으로 해결할 수 있음

$$-\underbrace{\{y \log h\}}_A + \underbrace{\{(1 - y) \log(1 - h)\}}_B$$

- 실제 값  $y$ 가 1이면 B 부분이 없어짐
- 반대로 0이면 A 부분이 없어짐
- 따라서  $y$  값에 따라 빨간색 그래프와 파란색 그래프를 각각 사용할 수 있게 됨

## 5 | 코딩으로 확인하는 로지스틱 회귀

- 이를 그대로 텐서플로로 옮겨 보자
- 먼저 텐서플로와 넘파이 라이브러리를 불러오고 x와 y 값을 정해 줌

```
import tensorflow as tf  
import numpy as np
```

```
data = [[2, 0], [4, 0], [6, 0], [8, 1], [10, 1], [12, 1], [14, 1]]  
x_data = [x_row[0] for x_row in data]  
y_data = [y_row[1] for y_row in data]
```

## 5 | 코딩으로 확인하는 로지스틱 회귀

- a와 b의 값을 임의로 정함
- 앞에서 배운 내용과 같음

```
a = tf.Variable(tf.random_normal([1], dtype=tf.float64, seed=0))  
b = tf.Variable(tf.random_normal([1], dtype=tf.float64, seed=0))
```

- 이제 시그모이드 함수 방정식을 넘파이 라이브러리를 이용해 다음과 같이 작성

$$y = \frac{1}{1 + e^{(ax+b)}}$$

```
y = 1/(1 + np.e**(a * x_data + b))
```

## 5 | 코딩으로 확인하는 로지스틱 회귀

- 오차를 구하는 함수 역시 넘파이와 텐서플로를 이용해 다음과 같이 작성할 수 있음

$$\text{오차} = -\text{평균}(y \cdot \log h + (1 - y) \log(1 - h))$$

```
loss = -tf.reduce_mean(np.array(y_data) * tf.log(y) + (1 - np.array(y_data)) *  
tf.log(1 - y))
```

→ 오차를 구하고 그 평균값을 loss 변수에 할당

→ 평균을 구하기 위해 reduce\_mean() 함수를 사용

## 5 | 코딩으로 확인하는 로지스틱 회귀

- 이제 학습률을 지정하고 경사 하강법을 이용해 오차를 최소로 하는 값을 찾아 보자

```
learning_rate = 0.5  
gradient_decent = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)
```

- 텐서플로를 구동시켜 결과값을 출력하는 부분

```
with tf.Session() as sess:  
    sess.run(tf.global_variables_initializer())  
  
    for i in range(60001):  
        sess.run(gradient_decent)  
        if i % 6000 == 0:  
            print("Epoch: %.f, loss = %.4f, 기울기 a = %.4f, y 절편 = %.4f" %  
                  (i, sess.run(loss), sess.run(a), sess.run(b)))
```

## 5 | 코딩으로 확인하는 로지스틱 회귀

### 코딩으로 확인하는 로지스틱 회귀

- Logistic\_Regression.py

```
import tensorflow as tf
import numpy as np

# x, y의 데이터 값
data = [[2, 0], [4, 0], [6, 0], [8, 1], [10, 1], [12, 1], [14, 1]]
x_data = [x_row[0] for x_row in data]
y_data = [y_row[1] for y_row in data]

# a와 b의 값을 임의로 정한다.
a = tf.Variable(tf.random_normal([1], dtype=tf.float64, seed=0))
b = tf.Variable(tf.random_normal([1], dtype=tf.float64, seed=0))

# y 시그모이드 함수의 방정식을 세운다.
y = 1/(1 + np.e**(a * x_data + b))
```





## 5 | 코딩으로 확인하는 로지스틱 회귀



```
# loss를 구하는 함수
loss = -tf.reduce_mean(np.array(y_data) * tf.log(y) + (1 - np.array(y_data)) *
                        tf.log(1 - y))

# 학습률 값
learning_rate = 0.5

# loss를 최소로 하는 값 찾기
gradient_decent = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)

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



## 5 | 코딩으로 확인하는 로지스틱 회귀



```
for i in range(60001):  
    sess.run(gradient_decent)  
    if i % 6000 == 0:  
        print("Epoch: %.f, loss = %.4f, 기울기 a = %.4f, y 절편 = %.4f" %  
              (i, sess.run(loss), sess.run(a), sess.run(b)))
```

## 5 | 코딩으로 확인하는 로지스틱 회귀

- 실행 결과

```
Epoch: 0, loss = 1.2676, 기울기 a = 0.1849, y 절편 = -0.4334
Epoch: 6000, loss = 0.0152, 기울기 a = -2.9211, y 절편 = 20.2982
Epoch: 12000, loss = 0.0081, 기울기 a = -3.5637, y 절편 = 24.8010
Epoch: 18000, loss = 0.0055, 기울기 a = -3.9557, y 절편 = 27.5463
Epoch: 24000, loss = 0.0041, 기울기 a = -4.2380, y 절편 = 29.5231
Epoch: 30000, loss = 0.0033, 기울기 a = -4.4586, y 절편 = 31.0675
Epoch: 36000, loss = 0.0028, 기울기 a = -4.6396, y 절편 = 32.3346
Epoch: 42000, loss = 0.0024, 기울기 a = -4.7930, y 절편 = 33.4086
Epoch: 48000, loss = 0.0021, 기울기 a = -4.9261, y 절편 = 34.3406
Epoch: 54000, loss = 0.0019, 기울기 a = -5.0436, y 절편 = 35.1636
Epoch: 60000, loss = 0.0017, 기울기 a = -5.1489, y 절편 = 35.9005
```

- 오차(loss) 값이 점차 줄어듦과 a와 b의 최적값을 찾아가는 것을 볼 수 있음

## 6 | 여러 입력 값을 갖는 로지스틱 회귀

- 선형 회귀를 공부할 때와 마찬가지로 변수가 많아지면 더 정확하게 예측을 할 수 있음
- 변수를 하나 더 추가하여 x와 y 데이터를 만들어 줌

```
x_data = np.array([[2, 3],[4, 3],[6, 4],[8, 6],[10, 7],[12, 8],[14, 9]])  
y_data = np.array([0, 0, 0, 1, 1, 1,1]).reshape(7, 1)
```

## 6 | 여러 입력 값을 갖는 로지스틱 회귀

- 이제 텐서플로에서 데이터를 담는 플레이스 홀더(placeholder)를 정해 줌

```
X = tf.placeholder(tf.float64, shape=[None, 2])  
Y = tf.placeholder(tf.float64, shape=[None, 1])
```

## 6 | 여러 입력 값을 갖는 로지스틱 회귀

- 변수가  $x$ 에서  $x_1, x_2$ 로 추가되면, 각각의 기울기  $a_1, a_2$ 도 계산해야 함
- 즉,  $ax$  부분이  $a_1x_1 + a_2x_2$ 로 바뀜
- $a_1x_1 + a_2x_2$ 는 행렬곱을 이용해  $[a_1, a_2] * [x_1, x_2]$ 로도 표현할 수 있음
- 텐서플로에서는 `matmul()` 함수를 이용해 행렬곱을 적용
- 시그모이드를 계산하기 위해 텐서플로에 내장된 `sigmoid()` 함수를 사용

```
y = tf.sigmoid(tf.matmul(X, a) + b)
```

## 6 | 여러 입력 값을 갖는 로지스틱 회귀

### 코딩으로 확인하는 다중 로지스틱 회귀


- Multi\_Logistic\_Regression.py

```
import tensorflow as tf
import numpy as np

# 실행할 때마다 같은 결과를 출력하기 위한 seed 값 설정
seed = 0
np.random.seed(seed)
tf.set_random_seed(seed)

# x, y의 데이터 값
x_data = np.array([[2, 3],[4, 3],[6, 4],[8, 6],[10, 7],[12, 8],[14, 9]])
y_data = np.array([0, 0, 0, 1, 1, 1,1]).reshape(7, 1)

# 입력 값을 플레이스 홀더에 저장
X = tf.placeholder(tf.float64, shape=[None, 2])
Y = tf.placeholder(tf.float64, shape=[None, 1])
```



## 6 | 여러 입력 값을 갖는 로지스틱 회귀



```
# 기울기 a와 바이어스 b의 값을 임의로 정함
a = tf.Variable(tf.random_uniform([2,1], dtype=tf.float64))
# [2,1] 의미: 들어오는 값은 2개, 나가는 값은 1개
b = tf.Variable(tf.random_uniform([1], dtype=tf.float64))

# y 시그모이드 함수의 방정식을 세움
y = tf.sigmoid(tf.matmul(X, a) + b)

# 오차를 구하는 함수
loss = -tf.reduce_mean(Y * tf.log(y) + (1 - Y) * tf.log(1 - y))

# 학습률 값
learning_rate=0.1

# 오차를 최소로 하는 값 찾기
gradient_descent = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)
```





## 6 | 여러 입력 값을 갖는 로지스틱 회귀



```
predicted = tf.cast(y > 0.5, dtype=tf.float64)
accuracy = tf.reduce_mean(tf.cast(tf.equal(predicted, Y), dtype=tf.float64))

# 학습
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())

    for i in range(3001):
        a_, b_, loss_, _ = sess.run([a, b, loss, gradient_decent], feed_dict={X:
x_data, Y: y_data})
        if (i + 1) % 300 == 0:
            print("step=%d, a1=%.4f, a2=%.4f, b=%.4f, loss=%.4f" % (i + 1, a_[0],
a_[1], b_, loss_))
```

## 6 | 여러 입력 값을 갖는 로지스틱 회귀

- 실행 결과

```
step=300, a1=0.8426, a2=-0.5997, b=-2.3907, loss=0.2694
step=600, a1=0.8348, a2=-0.3166, b=-3.8630, loss=0.1932
step=900, a1=0.7423, a2=0.0153, b=-4.9311, loss=0.1510
step=1200, a1=0.6372, a2=0.3245, b=-5.7765, loss=0.1235
step=1500, a1=0.5373, a2=0.5996, b=-6.4775, loss=0.1042
step=1800, a1=0.4471, a2=0.8421, b=-7.0768, loss=0.0900
step=2100, a1=0.3670, a2=1.0561, b=-7.6003, loss=0.0791
step=2400, a1=0.2962, a2=1.2458, b=-8.0652, loss=0.0705
step=2700, a1=0.2336, a2=1.4152, b=-8.4834, loss=0.0636
step=3000, a1=0.1779, a2=1.5675, b=-8.8635, loss=0.0579
```

- 오차(loss) 값이 점차 줄어듦고  $a_1$ ,  $a_2$ 와  $b$ 가 각각 최적값을 찾아가는 것이 보임

## 7 | 실제 값 적용하기

- 이제 조금 전에 만든 다중 로지스틱 회귀 스크립트를 실제로 사용해 예측 값을 구해 보자
- 다음 코드를 추가하면 공부한 시간과 과외 수업 횟수를 직접 입력해 볼 수 있음
- 예를 들어 7시간 공부하고 과외를 6번 받은 학생의 합격 가능성을 계산해 보자

# 어떻게 활용하는가

```
new_x = np.array([7, 6.]).reshape(1, 2) # [7, 6]은 각각 공부한 시간과  
과외 수업 횟수  
new_y = sess.run(y, feed_dict={X: new_x})  
  
print("공부한 시간: %d, 과외 수업 횟수: %d" % (new_x[:,0], new_x[:,1]))  
print("합격 가능성: %6.2f %" % (new_y*100))
```

## 7 | 실제 값 적용하기

- 이를 실행하면 다음과 같이 출력됨

공부한 시간: 7, 과외 수업 횟수: 6

합격 가능성: 85.66 %

- 7시간 공부하고 6번의 과외를 받은 학생의 합격 가능성은 85.66%임을 알 수 있음

## 8 | 로지스틱 회귀에서 퍼셉트론으로

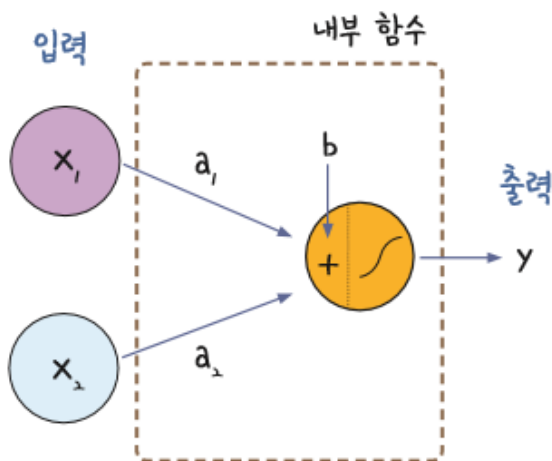
- [정리] 입력 값을 통해 출력 값을 구하는 함수  $y$ 는 다음과 같이 표현할 수 있음

$$y = a_1x_1 + a_2x_2 + b$$

- 우리가 가진 값은  $x_1$ 과  $x_2$ . 이를 입력 값이라고 하며,
- 계산으로 얻는 값  $y$ 를 출력 값이라고 함

## 8 | 로지스틱 회귀에서 퍼셉트론으로

- 이름 그림으로 나타내면 그림과 같음



로지스틱 회귀를 퍼셉트론 방식으로 표현한 예

- $x_1$ 과  $x_2$ 가 입력되고, 각각 가중치  $a_1$ ,  $a_2$ 를 만남
- 여기에  $b$  값을 더한 후 시그모이드 함수를 거쳐 1 또는 0의 출력 값  $y$ 를 출력

## 8 | 로지스틱 회귀에서 퍼셉트론으로

- 이 그림의 개념이 1957년, 코넬 항공 연구소의 프랑크 로젠블라트라는 사람이 발표한, '퍼셉트론(perceptron)'
- 이 퍼셉트론은 그 후 여러 학자들의 노력을 통해 인공 신경망, 오차 역전파 등의 발전을 거쳐 지금의 딥러닝으로 이어지게 됨