둘째 마당

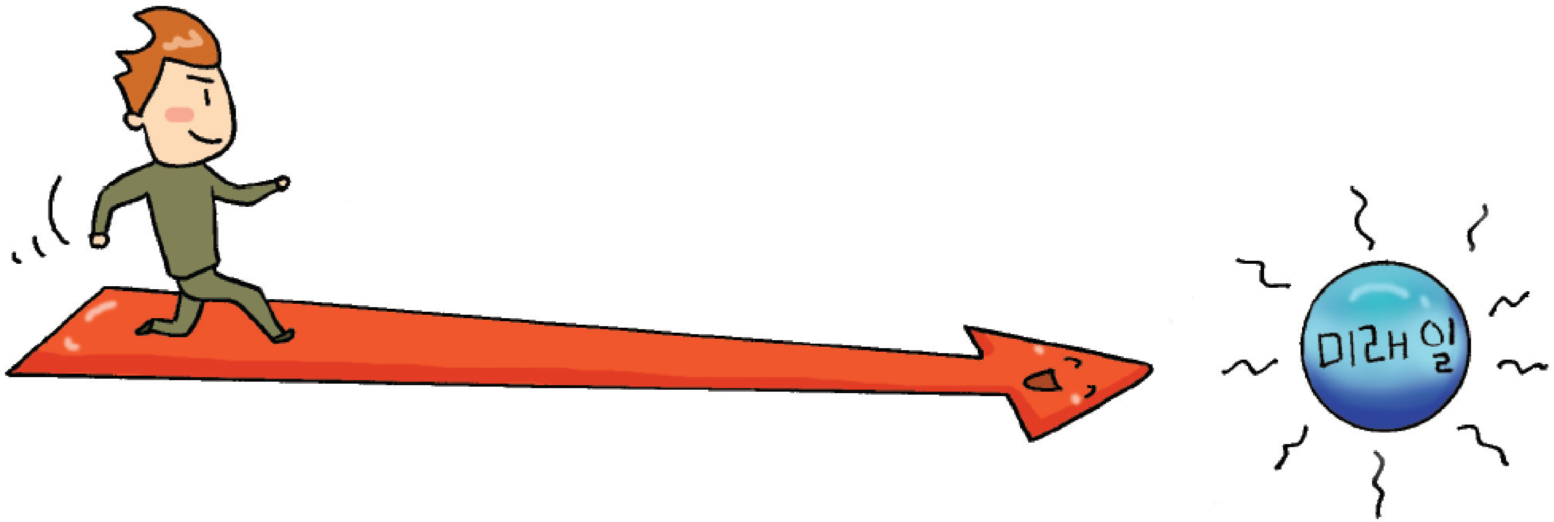
딥러닝의 동작 원리

3장 가장 훌륭한 예측선 긋기: 선형 회귀

3 가장 훌륭한 예측선 긋기: 선형 회귀

- 딥러닝을 이해하려면 딥러닝의 가장 말단에서 이루어지는 가장 기본적인 두 가지 계산 원리를 알아야 함

  → 바로 선형 회귀와 로지스틱 회귀임



1 | 선형 회귀의 정의

- 독립 변수 :

  'x값이 변함에 따라 y값도 변한다'는 이 정의 안에서, 독립적으로 변할 수 있는 x값

- 종속 변수 :

  독립 변수에 따라 종속적으로 변하는 값

- 선형 회귀 :

  독립 변수 x를 사용해 종속 변수 y의 움직임을 예측하고 설명하는 작업을 말함

1 | 선형 회귀의 정의

- 단순 선형 회귀(simple linear regression) :

  하나의 x값 만으로도 y값을 설명할 수 있을 때

- 다중 선형 회귀(multiple linear regression) :

  x값이 여러 개 필요할 때

2 | 가장 훌륭한 예측선이란?

- 우선 독립 변수가 하나뿐인 단순 선형 회귀의 예를 공부해 보자

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

표 3-1 공부한 시간과 중간고사 성적 데이터

- 여기서 공부한 시간을 x라 하고 성적을 y라 할 때 집합 X와 집합 Y를 다음과 같이 표현할 수 있음

$$X = \{2,\ 4,\ 6,\ 8\}$$
$$Y = \{81,\ 93,\ 91,\ 97\}$$

2 | 가장 훌륭한 예측선이란?

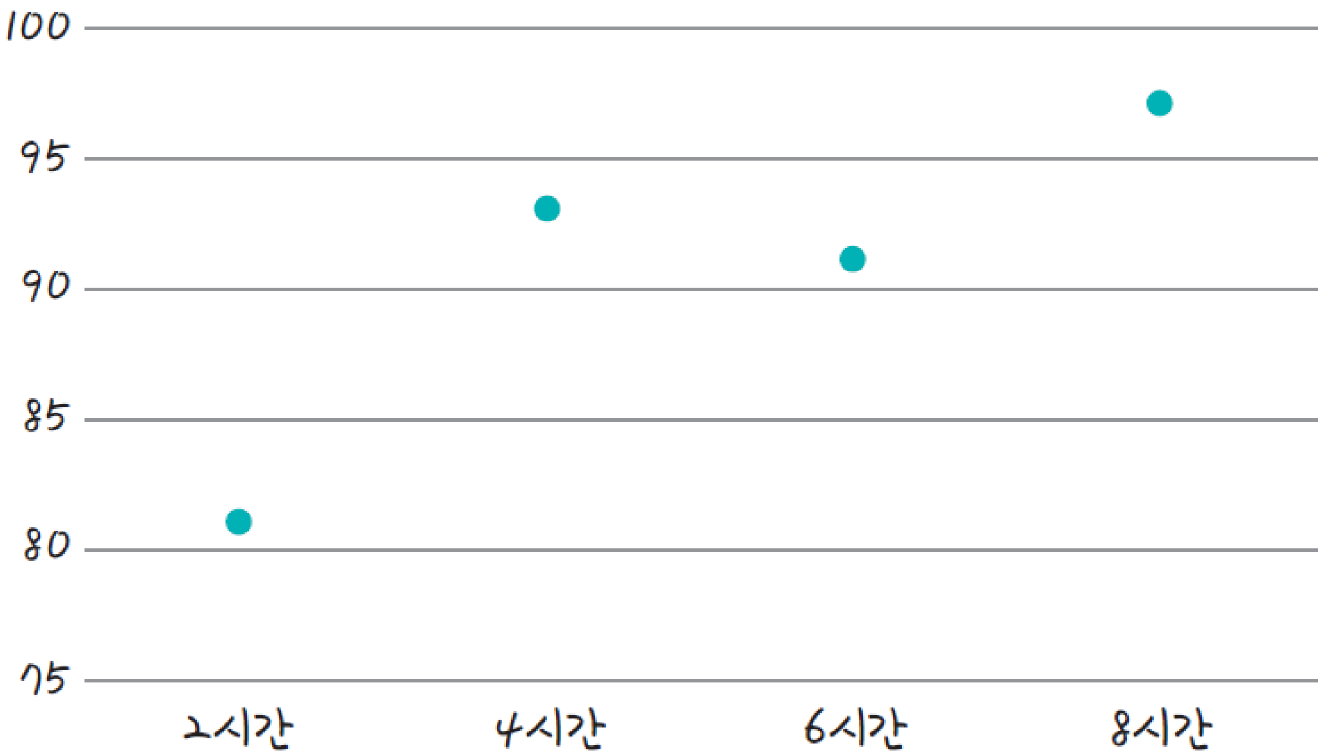


그림 3-1 공부한 시간과 성적을 좌표로 표현

2 | 가장 훌륭한 예측선이란?

- 선형 회귀를 공부하는 과정은 이 점들의 특징을 가장 잘 나타내는 선을 그리는 과정과 일치함
- 여기에서 선은 직선이므로 곧 일차 함수 그래프임

$$y = ax + b$$

- 여기서 x값은 독립 변수이고 y값은 종속 변수임
- 즉, x값에 따라 y값은 반드시 달라짐
- 다만, 정확하게 계산하려면 상수 a와 b의 값을 알아야 함
- 이 직선을 훌륭하게 그으려면 직선의 기울기 a값과 y절편 b값을 정확히 예측해 내야 함

3 | 최소 제곱법

- 최소 제곱법(method of least squares)이라는 공식을 알고 적용한다면, 이를 통해 일차 함수의 기울기 a와 절편 y를 바로 구할 수 있음
- 지금 가진 정보가 x값(입력 값, 여기서는 '공부한 시간')과 y값(출력 값, 여기서는 '성적')일 때 이를 이용해 기울기 a를 구하는 방법은 다음과 같음

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

→ 이것이 바로 최소 제곱법임

3 | 최소 제곱법

- x의 편차(각 값과 평균과의 차이)를 제곱해서 합한 값을 분모로 놓고, x와 y의 편차를 곱해서 합한 값을 분자로 놓으면 기울기가 나온다는 뜻임
  - 공부한 시간(x) 평균: $(2 + 4 + 6 + 8) \div 4 = 5$
  - 성적(y) 평균: $(81 + 93 + 91 + 97) \div 4 = 90.5$
- 이를 위 식에 대입하면 다음과 같음

$$a = \frac{(2-5)(81-90.5)+(4-5)(93-90.5)+(6-5)(91-90.5)+(8-5)(97-90.5)}{(2-5)^2+(4-5)^2+(6-5)^2+(8-5)^2}$$

$$= \frac{46}{20}$$

$$= 2.3$$

3 | 최소 제곱법

- 다음은 y절편인 b를 구하는 공식임

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

- 즉, y 의 평균에서 x의 평균과 기울기의 곱을 빼면 b 의 값이

$$b = 90.5 - (2.3 \times 5)$$
$$= 79$$

- $y = 2.3x + 79$ 예측 값을 구하기 위한 직선의 방정식이 완성됨

3 | 최소 제곱법

- 예측 값 :

  x를 대입했을 때 나오는 y값

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

표 3-1 최소 제곱법 공식으로 구한 성적 예측 값

3 | 최소 제곱법

- 좌표 평면에 이 예측 값을 찍어 보자

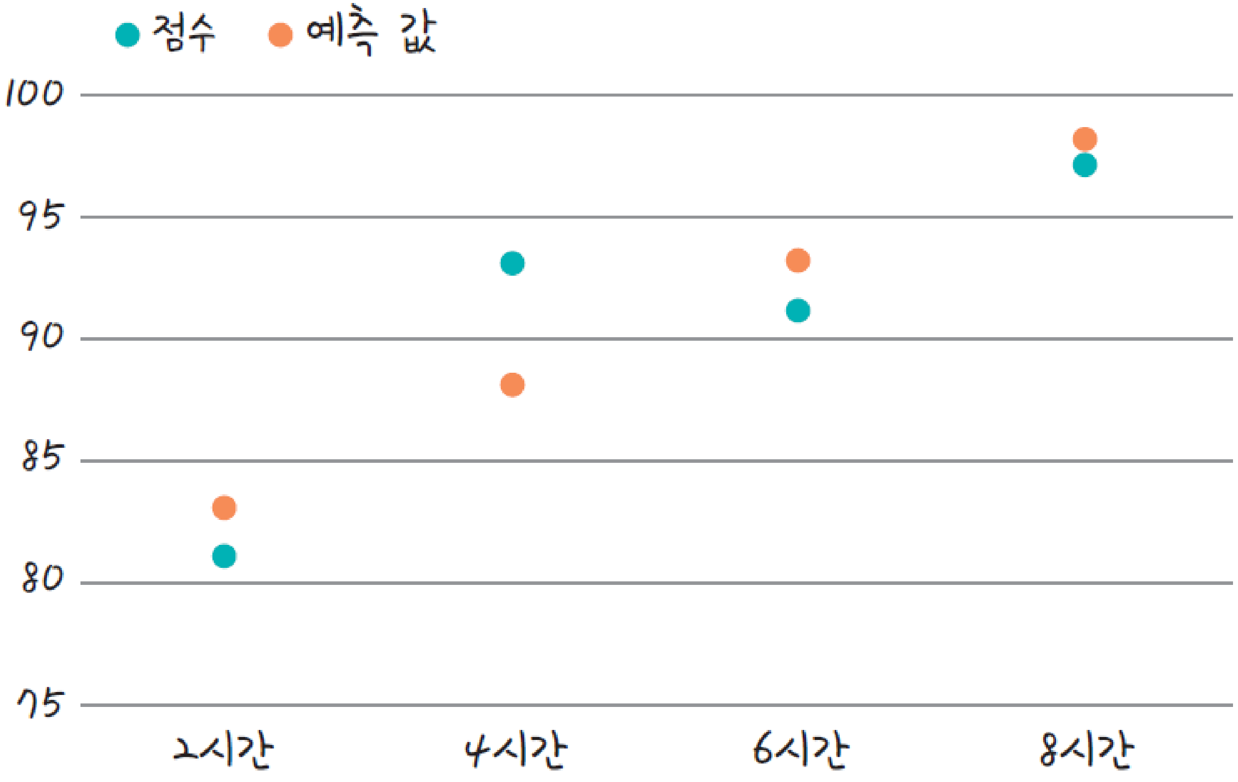


그림 3-2 공부한 시간, 성적, 예측 값을 좌표로 표현

3 | 최소 제곱법

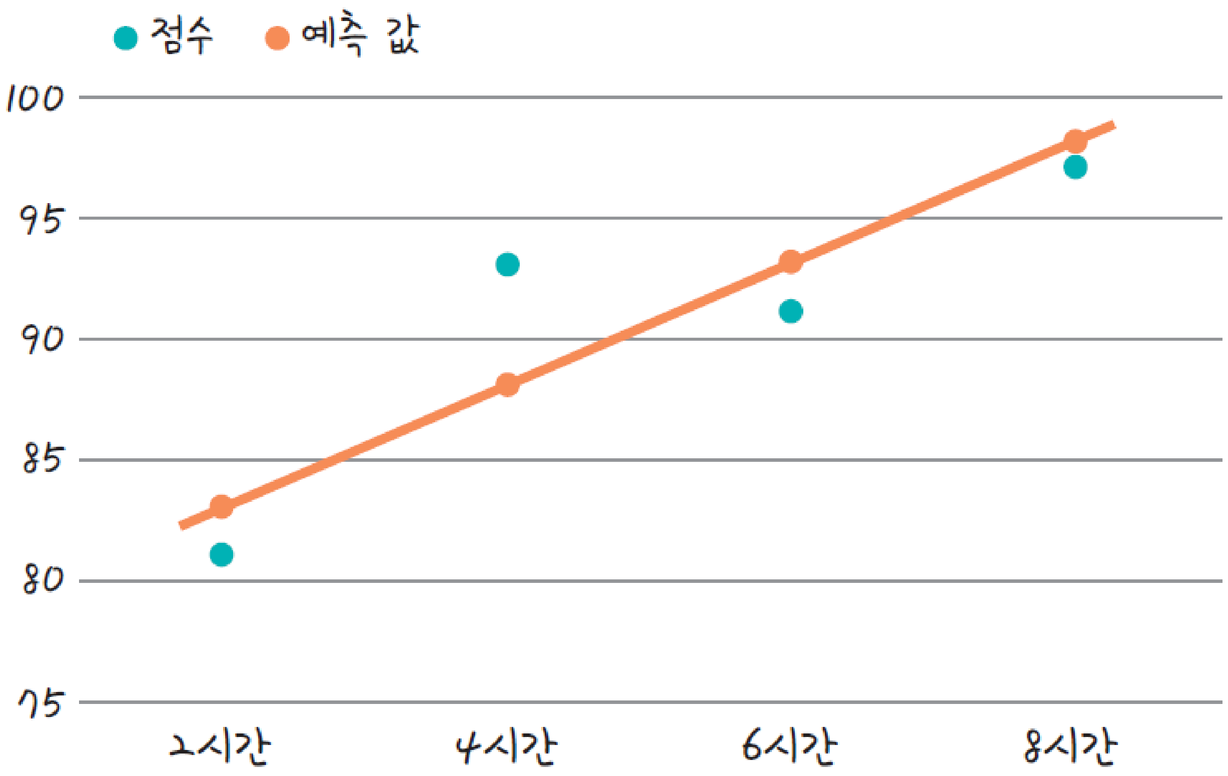


그림 3-3 오차가 최저가 되는 직선의 완성

3 | 최소 제곱법

- 이것이 바로 오차가 가장 적은, 주어진 좌표의 특성을 가장 잘 나타내는 직선임
- 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

- 넘파이 라이브러리를 불러옴
- 앞서 나온 데이터 값을 '리스트' 형식으로 다음과 같이 x와 y로 정의함

```
import numpy as np

x = [2, 4, 6, 8]
y = [81, 93, 91, 97]
```

TIP

파이썬 리스트를 만들려면 다음과 같이 리스트 이름을 정한 후 대괄호([ ])로 감싼 요소들을 쉼표(,)로 구분해 대입하면 됩니다.

리스트 이름 = [요소 1, 요소 2, 요소 3, ...]

4 | 코딩으로 확인하는 최소 제곱

- 이제 최소 제곱근 공식으로 기울기 y와 b절편 a의 값을 구해보자
- x 의 모든 원소의 평균을 구하는 넘파이 함수는 mean( )임
- mx 라는 변수에 x원소들의 평균값을, my에 y원소들의 평균값을 입력

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

4 | 코딩으로 확인하는 최소 제곱

- 최소 제곱근 공식 중 분모의 값, 즉 ' x의 각 원소와 x의 평균값들의 차를 제곱하라'는 파이썬 명령을 만들 차례임
- 다음과 같이 divisor라는 변수를 만들어 구현할 수 있음

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

TIP

- sum()은 Σ에 해당하는 함수입니다
- **2는 제곱을 구하라는 의미입니다.
- for i in x는 x의 각 원소를 한 번씩 i 자리에 대입하라는 의미입니다.

4 | 코딩으로 확인하는 최소 제곱

- 이제 분자에 해당하는 부분을 구함
- x와 y의 편차를 곱해서 합한 값을 구하면 됨
- 다음과 같이 새로운 함수를 정의하여 dividend 변수에 분자의 값을 저장함

```
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)
```

4 | 코딩으로 확인하는 최소 제곱

- 임의의 변수 d의 초깃값을 0으로 설정한 뒤 x의 개수만큼 실행함
- d에 x의 각 원소와 평균의 차, y의 각 원소와 평균의 차를 곱해서 차례로 더하는 최소 제곱법을 그대로 구현함

TIP

def는 함수를 만들 때 사용하는 예약어입니다. 여기서는 top()이라는 함수를 새롭게 만들었고, 그 안에 최소 제곱법의 분자식을 그대로 가져와 구현하였습니다.

4 | 코딩으로 확인하는 최소 제곱

- 이제 위에서 구한 분모와 분자를 계산하여 기울기 a를 구함

```
a = dividend / divisor
```

- a를 구하고 나면 y절편을 구하는 공식을 이용해 b를 구할 수 있음

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

4 | 코딩으로 확인하는 최소 제곱

코드 3-1 선형 회귀 실습

- 예제 소스: deeplearning_class/01_Linear_Square_Method.ipynb

```
import numpy as np


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

4 | 코딩으로 확인하는 최소 제곱

```
# 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)
```

4 | 코딩으로 확인하는 최소 제곱


```
# 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
```

5 | 평균 제곱 오차

- 여러 개의 입력 값을 계산할 때는 임의의 선을 그리고 난 후, 이 선이 얼마나 잘 그려졌는지를 평가하여 조금씩 수정해 가는 방법을 사용함
- 이를 위해 주어진 선의 오차를 평가하는 오차 평가 알고리즘이 필요함

6 | 잘못 그은 선 바로잡기

- 모든 딥러닝 프로젝트는 여러 개의 입력 변수를 다룸
- 가장 많이 사용하는 방법은 '일단 그리고 조금씩 수정해 나가기' 방식임
- 가설을 하나 세운 뒤 이 값이 주어진 요건을 충족하는지 판단하여 조금씩 변화를 줌
- 이 변화가 긍정적이면 오차가 최소가 될 때까지 이 과정을 계속 반복하는 방법
- 이는 딥러닝을 가능하게 해 주는 가장 중요한 원리 중 하나임

6 | 잘못 그은 선 바로잡기

- 선을 긋고 나서 수정하는 과정에서 빠지면 안 되는 것이 있음
- 나중에 그린 선이 먼저 그린 선보다 더 좋은지 나쁜지를 판단하는 방법임
- 즉, 각 선의 오차를 계산할 수 있어야 하고, 오차가 작은 쪽으로 바꾸는 알고리즘이 필요함

6 | 잘못 그은 선 바로잡기

- 지금부터 오차를 계산하는 방법을 알아보자

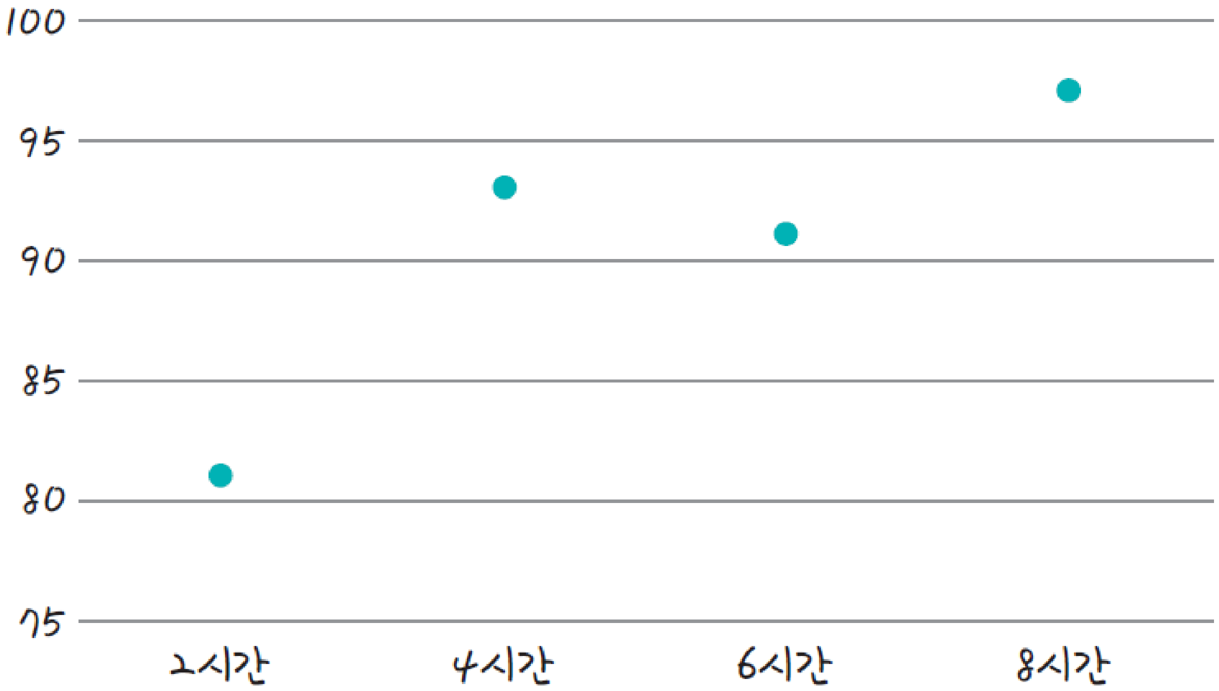


그림 3-4 공부한 시간과 성적의 관계도

6 | 잘못 그은 선 바로잡기

- 임의의 값을 대입한 뒤 오차를 구하고 이 오차를 최소화하는 방식을 사용해서 최종 a와 최종 b의 값을 구해 보자
- 대강 선을 그어보기 위해서 기울기 a와 절편 y를 임의의 수 3과 76이라고 가정해 보자

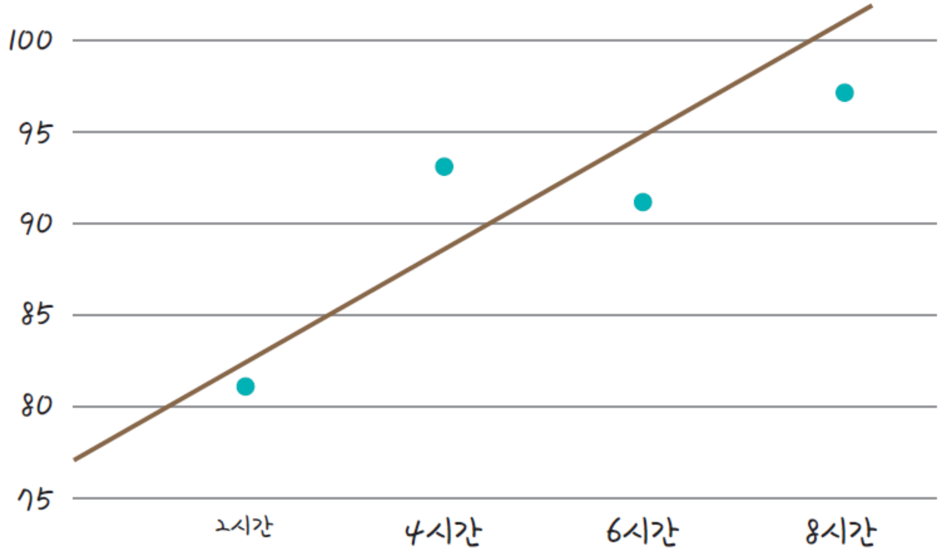


그림 3-5 임의의 직선 그려보기

6 | 잘못 그은 선 바로잡기

- 그림 3-5와 같은 임의의 직선이 어느 정도의 오차가 있는지를 확인하려면 각 점과 그래프 사이의 거리를 재면 됨

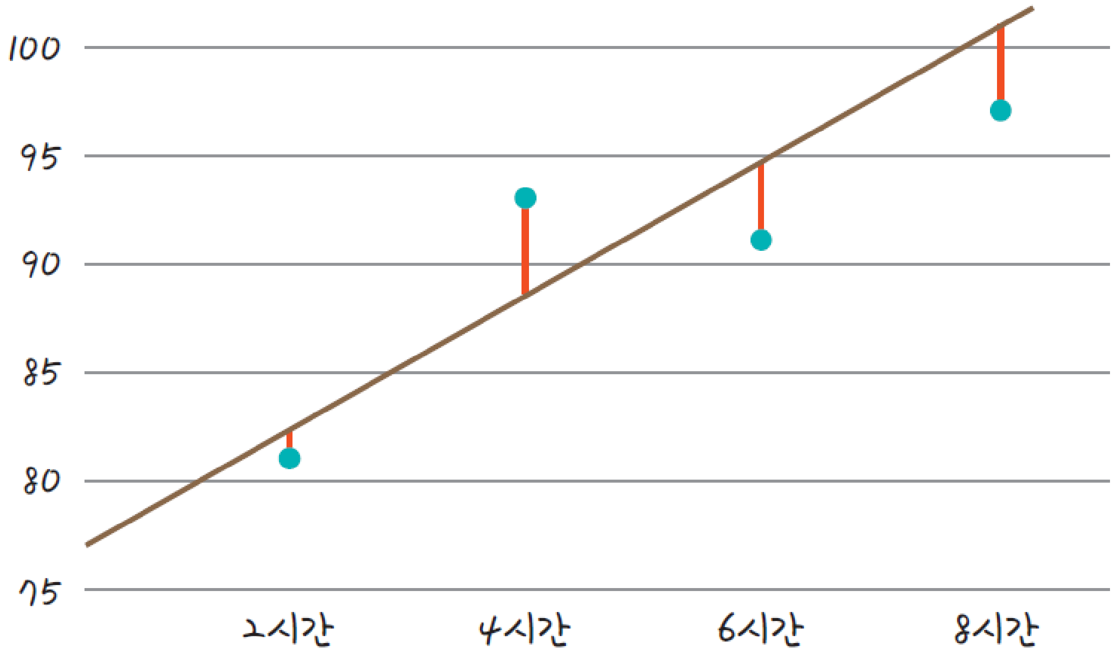


그림 3-6 임의의 직선과 실제 값 사이의 거리

6 | 잘못 그은 선 바로잡기

- 그림 3-6에서 볼 수 있는 빨간색 선은 직선이 잘 그어졌는지를 나타냄
- 이 직선들의 합이 작을수록 잘 그어진 직선이고, 이 직선들의 합이 클수록 잘못 그어진 직선이 됨

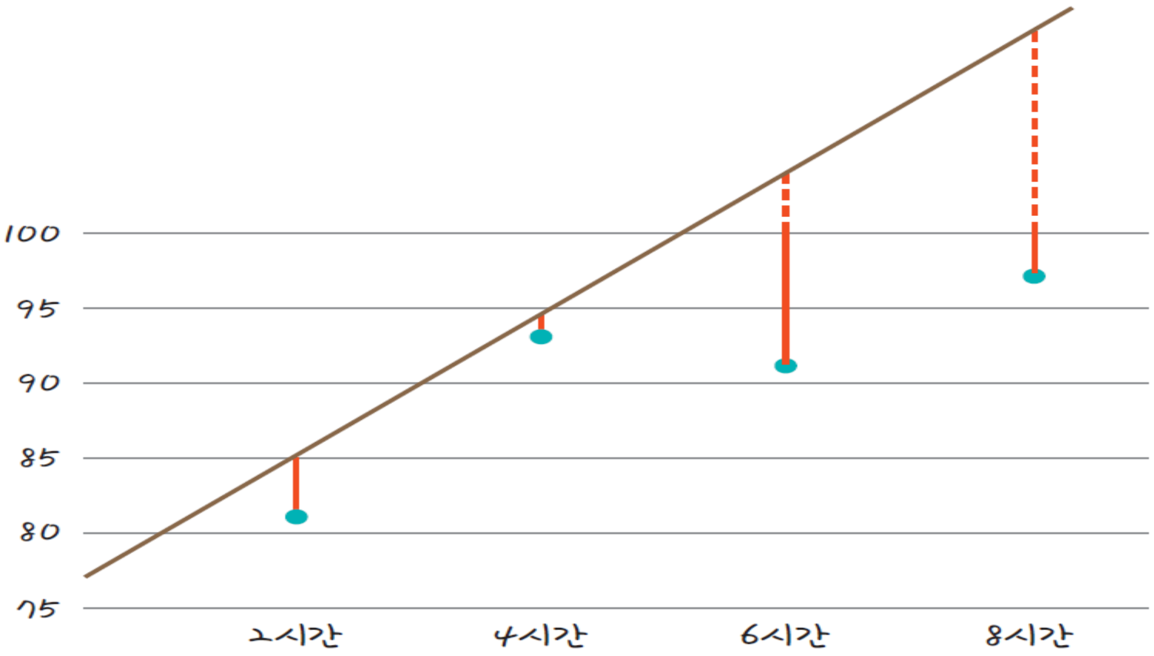


그림 3-7 기울기를 너무 크게 잡았을 때의 오차

6 | 잘못 그은 선 바로잡기

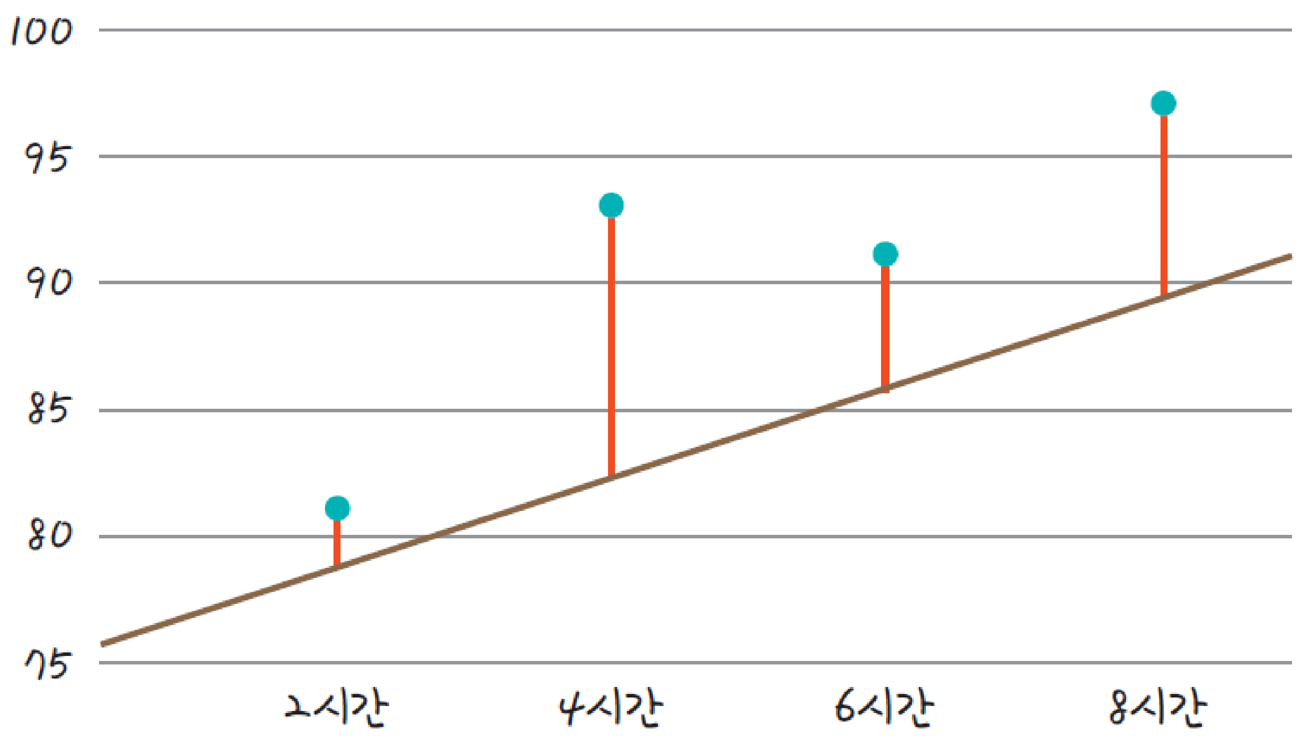


그림 3-8 기울기를 너무 작게 잡았을 때의 오차

6 | 잘못 그은 선 바로잡기

- 그래프의 기울기가 잘못 되었을수록 빨간색 선의 거리의 합, 즉 오차의 합도 커짐
- 만약 기울기가 무한대로 커지면 오차도 무한대로 커지는 상관관계가 있는 것을 알 수 있음
- 거리는 입력 데이터에 나와 있는 y의 '실제 값'과 x를 의 식에 대입해서 나오는 '예측 값'과의 차이를 통해 구할 수 있음

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

6 | 잘못 그은 선 바로잡기

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

표 3-3 주어진 데이터에서 오차 구하기

- 이렇게 해서 구한 오차를 모두 더하면 1 + (-5) + 3 + 3 = 2가 됨
- 이 값은 오차가 실제로 얼마나 큰지를 가늠하기에는 적합하지 않음
- 오차에 양수와 음수가 섞여 있어서 오차를 단순히 더해 버리면 합이 0이 될 수도 있기 때문임
- 부호를 없애야 정확한 오차를 구할 수 있음

6 | 잘못 그은 선 바로잡기

- 오차의 합을 구할 때는 각 오차의 값을 제곱해 줌

$$\text{오차의 합} = \sum_{i}^{n} (\hat{y}_i - y_i)^2$$

- 여기서 i 는 x가 나오는 순서를, n은 x 원소의 총 개수를 의미함
- $\hat{y}_i$는 x_i에 대응하는 '실제 값'이고 y_i는 x_i가 대입되었을 때 직선의 방정식(여기서는 $p = 3x + 76$)이 만드는 '예측 값'임
- 이 식으로 오차의 합을 다시 계산하면 1 + 25 + 9 + 9 = 44임

6 | 잘못 그은 선 바로잡기

- 평균 제곱 오차(Mean Squared Error, MSE) :

  오차의 합에 이어 각 x 값의 평균 오차를 이용함

  위에서 구한 값을 n으로 나누면 오차 합의 평균을 구할 수 있음

$$\text{평균 제곱 오차(MSE)} = \frac{1}{n}\sum(\hat{y}_i - y_i)^2$$

- 선형 회귀란 :

  임의의 직선을 그어 이에 대한 평균 제곱 오차를 구하고, 이 값을 가장 작게 만들어 주는 a와 b 값을 찾아가는 작업임

7 | 코딩으로 확인하는 평균 제곱 오차

- 이제 앞서 알아본 평균 제곱 오차를 파이썬으로 구현해 보자

```
fake_a_b = [3, 76]
```

7 | 코딩으로 확인하는 평균 제곱 오차

- 이번에는 data라는 리스트를 만들어 공부한 시간과 이에 따른 성적을 각각 짝을지어 저장함
- x 리스트와 y 리스트를 만들어 첫 번째 값을 x 리스트에 저장하고 두 번째 값을 y 리스트에 저장함

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

TIP

파이썬에서 i[0]은 i 값 중 첫 번째를, i[1]은 두 번째 값을 의미합니다.

7 | 코딩으로 확인하는 평균 제곱 오차

- 다음은 내부 함수를 만들 차례임
- predict( )라는 함수를 사용해 일차 방정식 y = ax + b를 구현함

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

7 | 코딩으로 확인하는 평균 제곱 오차

- 평균 제곱근 공식을 그대로 파이썬 함수로 옮기면 다음과 같음

$$\frac{1}{n}\sum(\hat{y}_i - y_i)^2$$

```
def mse(y_hat, y):
    return ((y_hat-y) ** 2).mean())
```

- 여기서 **2는 제곱을 구하라는 뜻이고, mean( )은 평균값을 구하라는 뜻
- 예측 값과 실제 값을 각각 mse( )라는 함수의 y_hat와 y 자리에 입력해서 평균 제곱을 구함

7 | 코딩으로 확인하는 평균 제곱 오차

- 이제 mse( ) 함수에 데이터를 대입하여 최종값을 구하는 함수

```
def mse_val(predict_result, y):
    return mse(np.array(predict_result), np.array(y))
```

- predict_result에는 앞서 만든 일차 방정식 함수 predict( )의 결괏값이 들어감
- 이 값과 y 값이 각각 예측 값과 실제 값으로 mse( ) 함수 안에 들어가게 됨

7 | 코딩으로 확인하는 평균 제곱 오차

- 이제 모든 x 값을 predict( ) 함수에 대입하여 예측 값을 구함
- 이 예측 값과 실제 값을 통해 최종값을 출력하는 코드를 다음과 같이 작성함

```
# 예측 값이 들어갈 빈 리스트
predict_result = []


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

7 | 코딩으로 확인하는 평균 제곱 오차

코드 3-2 선형 회귀 실습 2

- 예제 소스: deeplearning_class/02_Mean_Squared_Error.ipynb

```
import numpy as np


# 기울기 a와 y 절편 b
fake_a_b = [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]
```

7 | 코딩으로 확인하는 평균 제곱 오차

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

# MSE 함수
def mse(y_hat, y):
    return ((y_hat, y) ** 2).mean( ))

# MSE 함수를 각 y 값에 대입하여 최종 값을 구하는 함수
def mse_val(predict_result,y):
    return mse(np.array(predict_result), np.array(y))
```

7 | 코딩으로 확인하는 평균 제곱 오차

```
# 예측 값이 들어갈 빈 리스트
predict_result = []

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

# 최종 MSE 출력
print("mse 최종값: " + str(mse_val(predict_result,y)))
```

7 | 코딩으로 확인하는 평균 제곱 오차

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
mse 최종값: 11.0
```

4장 오차 수정하기: 경사 하강법

1 | 경사 하강법의 개요

2 | 학습률

3 | 코딩으로 확인하는 경사 하강법

4 | 다중 선형 회귀란

5 | 코딩으로 확인하는 다중 선형 회귀

4 오차 수정하기: 경사 하강법

- a를 무한대로 키우면 오차도 무한대로 커지고 a를 무한대로 작게 해도 역시 오차도 무한대로 커지는 이러한 관계는 이차 함수 그래프로 표현할 수 있음

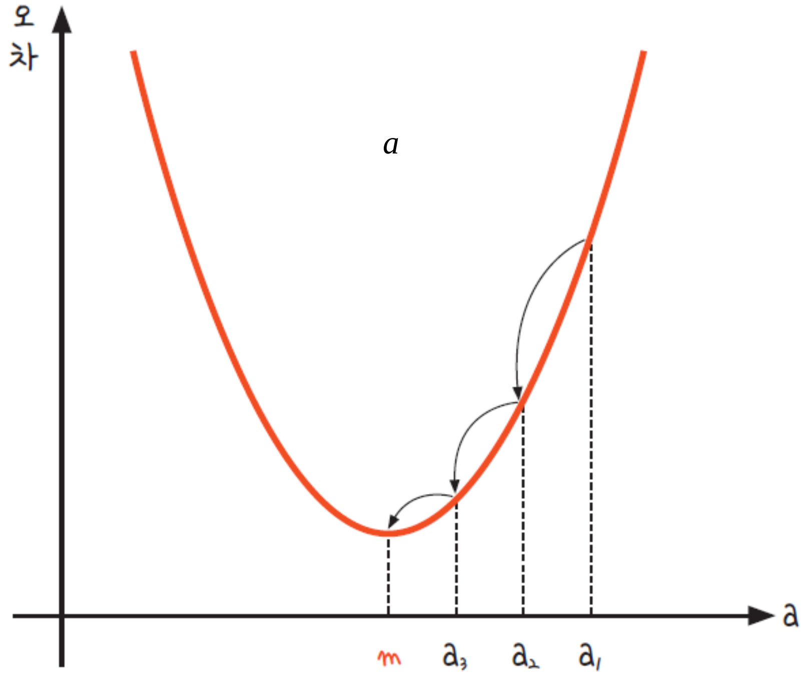


그림 4-1 기울기 a와 오차와의 관계: 적절한 기울기를 찾았을 때 오차가 최소화된다.

4 오차 수정하기: 경사 하강법

- 컴퓨터를 이용해 m 의 값을 구하려면 임의의 한 점(a_1)을 찍고 m이 점을 에 가까운 쪽으로 점점 이동($a_1 \to a_2 \to a_3$)시키는 과정이 필요함
- 경사 하강법(gradient descent) :
  그래프에서 오차를 비교하여 가장 작은 방향으로 이동시키는 방법이 있는데 바로 미분 기울기를 이용

1 | 경사 하강법의 개요

- $y = x^2$ 그래프에서 x 에 다음과 같이 a_1, m 그리고 a_2을 대입하여 그 자리에서 미분하면 그림 4-2처럼 각 점에서의 순간 기울기가 그려짐

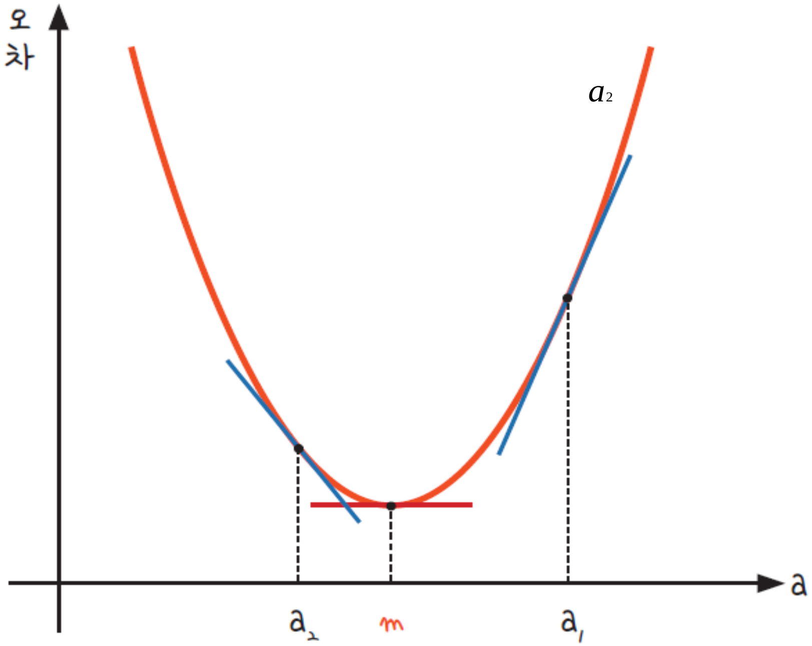


그림 4-2 순간 기울기가 0인 점이 곧 우리가 찾는 최솟값 m이다.

1 | 경사 하강법의 개요

- 여기서 눈여겨 봐야 할 것은 우리가 찾는 최솟값 m에서의 순간 기울기임
- 그래프가 이차 함수 포물선이므로 꼭짓점의 기울기는 x축과 평행한 선이 됨
- 즉, 기울기가 0임
- 우리가 할 일은 '미분 값이 0인 지점'을 찾는 것이 됨

1 | a_1에서 미분을 구함

2 | 구해진 기울기의 반대 방향(기울기가 +면 음의 방향, -면 양의 방향)으로 얼마간 이동시킨 a_2에서 미분을 구함(그림 4-3 참조).

3 | 위에서 구한 미분 값이 0이 아니면 위 과정을 반복함

1 | 경사 하강법의 개요

- 그림 4-3처럼 기울기가 0인 한 점(m)으로 수렴함

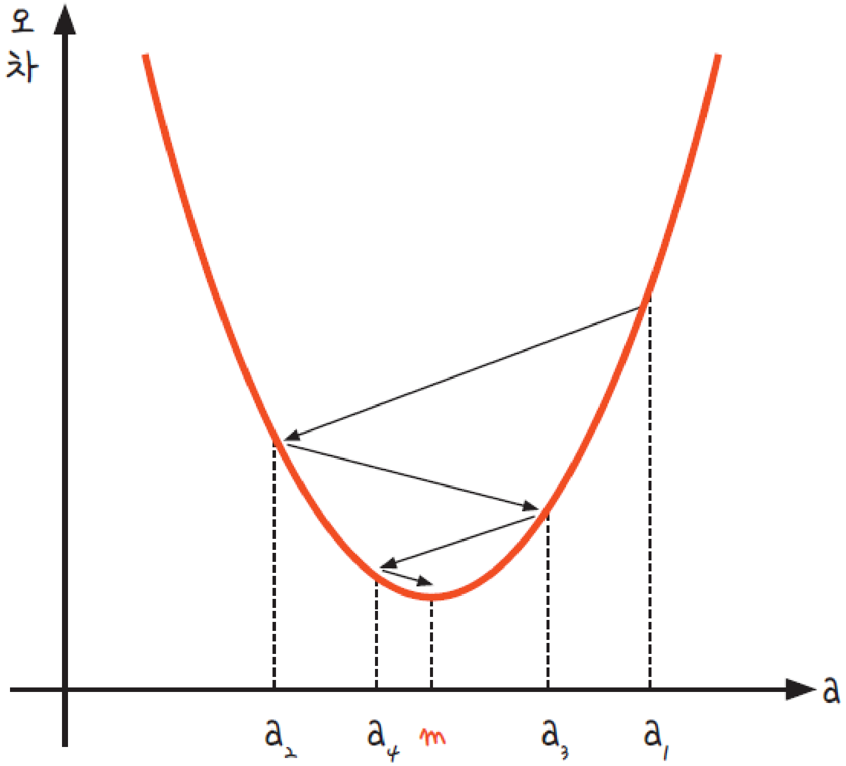


그림 4-3 최솟점 m을 찾아가는 과정

1 | 경사 하강법의 개요

- 경사 하강법 :

  이렇게 반복적으로 기울기 a를 변화시켜서 m의 값을 찾아내는 방법을 말함

2 | 학습률

- 기울기의 부호를 바꿔 이동시킬 때 적절한 거리를 찾지 못해 너무 멀리 이동시키면 a값이 한 점으로 모이지 않고 그림 4-4처럼 위로 치솟아 버림

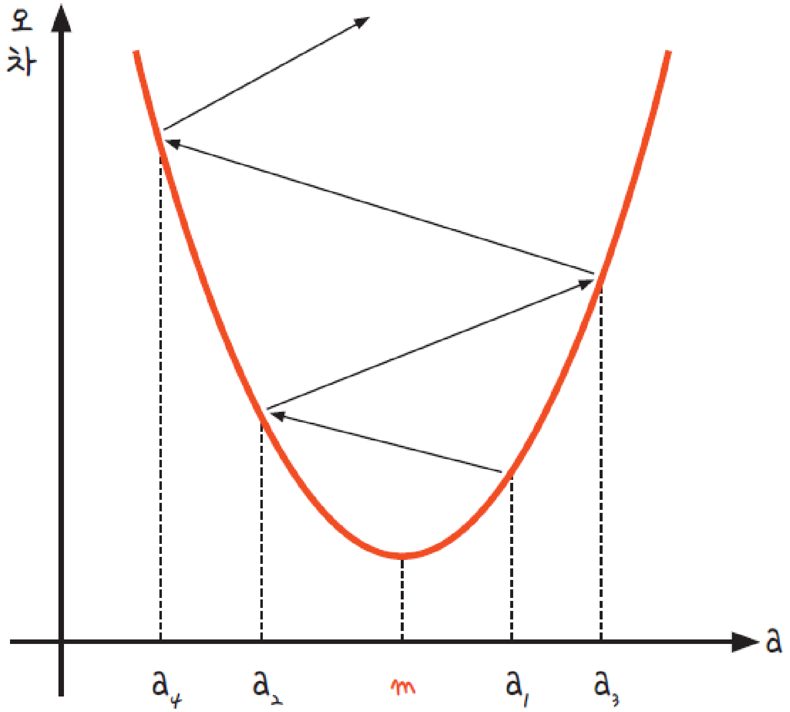


그림 4-4 학습률을 너무 크게 잡으면 한 점으로 수렴하지 않고 발산한다.

2 | 학습률

- 학습률 :

  어느 만큼 이동시킬지를 신중히 결정해야 하는데, 이때 이동 거리를 정해주는 것

- 딥러닝에서 학습률의 값을 적절히 바꾸면서 최적의 학습률을 찾는 것은 중요한 최적화 과정 중 하나임

2 | 학습률

- 경사 하강법
  - 오차의 변화에 따라 이차 함수 그래프를 만들고 적절한 학습률을 설정해 미분 값이 0인 지점을 구하는 것
  - y 절편 b 의 값도 이와 같은 성질을 가지고 있음
  - b 값이 너무 크면 오차도 함께 커지고, 너무 작아도 오차가 커짐
  - 최적의 b 값을 구할 때 역시 경사 하강법을 사용함

3 | 코딩으로 확인하는 경사 하강법

- 최솟값을 구하기 위해서는 이차 함수에서 미분을 해야 함
- 그 이차 함수는 평균 제곱 오차를 통해 나온다는 것임
- 평균 제곱 오차의 식을 다시 옮겨 보면 다음과 같음

$$\frac{1}{n}\sum(\hat{y}_i - y_i)^2$$

- 여기서 $\hat{y}_i$은 x_i를 집어 넣었을 때의 값이므로 $y_i = ax_i + b$를 대입하면 다음과 같이 바뀜

$$\frac{1}{n}\sum((ax_i + b) - y_i)^2$$

3 | 코딩으로 확인하는 경사 하강법

- 이 값을 미분할 때 우리가 궁금한 것은 a 와 b라는 것에 주의해야 함
- 식 전체를 미분하는 것이 아니라 필요한 값을 중심으로 미분해야 하기 때문임

$$a\text{로 편미분 한 결과} = \frac{2}{n}\sum(ax_i + b - y_i)x_i$$

$$b\text{로 편미분 한 결과} = \frac{2}{n}\sum(ax_i + b - y_i)$$

3 | 코딩으로 확인하는 경사 하강법

TIP

a로 편미분한 결과 유도 과정

$$
\begin{aligned}
\frac{a}{\partial a}MSE(a,b) &= \frac{1}{n}\sum[(ax_i+b-y_i)^2]' \\
&= \frac{2}{n}(ax_i+b-y_i)[(ax_i+b-y_i)]' \\
&= \frac{2}{n}\sum(ax_i+b-y_i)x_i
\end{aligned}
$$

b로 편미분한 결과 유도 과정

$$
\begin{aligned}
\frac{a}{\partial a}MSE(a,b) &= \frac{1}{n}\sum[(ax_i+b-y_i)^2]' \\
&= \frac{2}{n}(ax_i+b-y_i)[(ax_i+b-y_i)]' \\
&= \frac{2}{n}\sum(ax_i+b-y_i)
\end{aligned}
$$

3 | 코딩으로 확인하는 경사 하강법

- 이를 각각 파이썬 코드로 바꾸면 다음과 같음

```
y_pred = a * x_data + b   # 오차 함수인 y = ax + b를 정의한 부분
error = y_data - y_pred   # 실제값 – 예측값, 즉 오차를 구하는 식

# 평균 제곱 오차를 a로 미분한 결과
a_diff = -(2 / len(x_data)) * sum(x_data * (error))
# 평균 제곱 오차를 b로 미분한 결과
b_diff = -(2 / len(x_data)) * sum(y_data - y_pred)
```

3 | 코딩으로 확인하는 경사 하강법

- 여기에 학습률을 곱해 기존의 a값과 b값을 업데이트해 줌

```
a = a - lr * a_diff   # 미분 결과에 학습률을 곱한 후 기존의 a값을 업데이트
b = b - lr * b_diff   # 미분 결과에 학습률을 곱한 후 기존의 b값을 업데이트
```

3 | 코딩으로 확인하는 경사 하강법

- 중간 과정을 그래프로 표현하는 코드를 넣어 모두 정리하면 다음과 같이 코드가 완성됨

코드 4-1 경사 하강법 실습

- 예제 소스: deeplearning_class/03_Linear_Regression.ipynb

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# 공부 시간 X와 성적 Y의 리스트를 만들기
data = [[2, 81], [4, 93], [6, 91], [8, 97]]
x = [i[0] for i in data]
y = [i[1] for i in data]
```

3 | 코딩으로 확인하는 경사 하강법

```
# 그래프로 나타내기
plt.figure(figsize=(8,5))
plt.scatter(x, y)
plt.show()

# 리스트로 되어 있는 x와 y 값을 넘파이 배열로 바꾸기(인덱스를 주어 하나씩 불러와 계산이 가능하게 하기 위함)
x_data = np.array(x)
y_data = np.array(y)

# 기울기 a와 절편 b의 값 초기화
a = 0
b = 0
```

3 | 코딩으로 확인하는 경사 하강법

```
# 학습률 정하기
lr = 0.05

# 몇 번 반복될지 설정(0부터 세므로 원하는 반복 횟수에 +1)
epochs = 2001


# 경사 하강법 시작
for i in range(epochs):         # 에포크 수만큼 반복
    y_pred = a * x_data + b    # y를 구하는 식 세우기
    error = y_data - y_pred    # 오차를 구하는 식
    # 오차 함수를 a로 미분한 값
    a_diff = -(1/len(x_data)) * sum(x_data * (error))
    # 오차 함수를 b로 미분한 값
    b_diff = -(1/len(x_data)) * sum(y_data - y_pred)
```

3 | 코딩으로 확인하는 경사 하강법


```
    a = a - lr * a_diff   # 학습률을 곱해 기존의 a값 업데이트
    b = b - lr * b_diff   # 학습률을 곱해 기존의 b값 업데이트

    if i % 100 == 0:      # 100번 반복될 때마다 현재의 a값, b값 출력
        print("epoch=%.f, 기울기=%.04f, 절편=%.04f" % (i, a, b))

# 앞서 구한 기울기와 절편을 이용해 그래프를 다시 그리기
y_pred = a * x_data + b
plt.scatter(x, y)
plt.plot([min(x_data), max(x_data)], [min(y_pred), max(y_pred)])
plt.show()
```

3 | 코딩으로 확인하는 경사 하강법

실행 결과

```
epoch=0, 기울기=23.2000, 절편=4.5250
epoch=100, 기울기=7.9316, 절편=45.3932
epoch=200, 기울기=4.7953, 절편=64.109
epoch=300, 기울기=3.4056, 절편=72.4022
(중략)
epoch=1800, 기울기=2.3000, 절편=79.0000
epoch=1900, 기울기=2.3000, 절편=79.0000
epoch=2000, 기울기=2.3000, 절편=79.0000
```

3 | 코딩으로 확인하는 경사 하강법

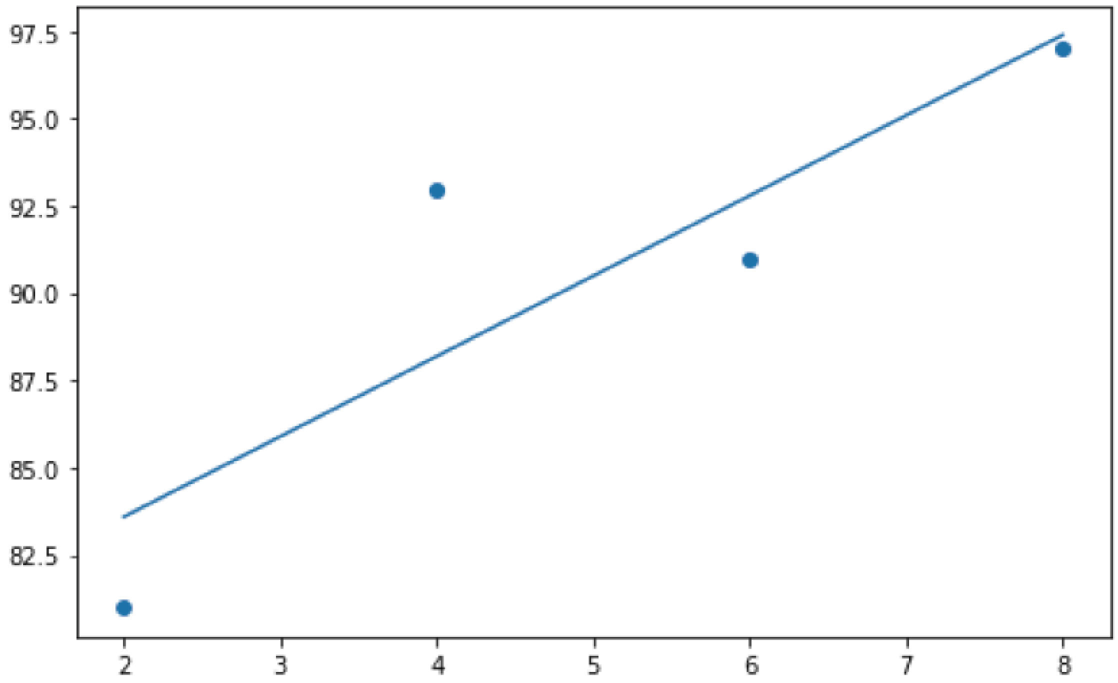


그림 4-5 그래프로 표현한 모

TIP 여기서 에포크(epoch)는 입력 값에 대해 몇 번이나 반복하여 실험했는지를 나타냅니다. 우리가 설정한 실험을 반복하고 100번마다 결과를 내놓습니다.

4 | 다중 선형 회귀란

- 더 정확한 예측을 하려면 추가 정보를 입력해야 하며, 정보를 추가해 새로운 예측값을 구하려면 변수의 개수를 늘려 다중 선형 회귀를

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

표 4-1 공부한 시간, 과외 수업 횟수에 따른 성적 데이터

4 | 다중 선형 회귀란

- 그럼 지금부터 두 개의 독립 변수 x_1 과 x_2 가 생긴 것임
- 이를 사용해 종속 변수 y를 만들 경우 기울기를 두 개 구해야 하므로 다음과 같은 식이 나옴

$$y = a_1x_1 + a_2x_2 + b$$

5 | 코딩으로 확인하는 다중 선형 회귀

- 이번에는 x 의 값이 두 개이므로 다음과 같이 data 리스트를 만들고 x_1 과 x_2 라는 두 개의 독립 변수 리스트를 만들어 줌

```
data = [[2, 0, 81], [4, 4, 93], [6, 2, 91], [8, 3, 97]]
x1 = [i[0] for i in data]
x2 = [i[1] for i in data]
y = [i[2] for i in data]
```

5 | 코딩으로 확인하는 다중 선형 회귀

- data가 그래프로 어떻게 보이는지를 확인해 보자
- 먼저 x, y 두 개의 축이던 이전과 달리 x_1, x_2, y 이렇게 세 개의 축이 필요함
- 3D 그래프를 그려주는 라이브러리를 아래와 같이 불러옴

```
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d   # 3D 그래프 그리는 라이브러리 가져오기

ax = plt.axes(projection='3d')     # 그래프 유형 정하기
ax.set_xlabel('study_hours')
ax.set_ylabel('private_class')
ax.set_zlabel('Score')
ax.scatter(x1, x2, y)
plt.show()
```

5 | 코딩으로 확인하는 다중 선형 회귀

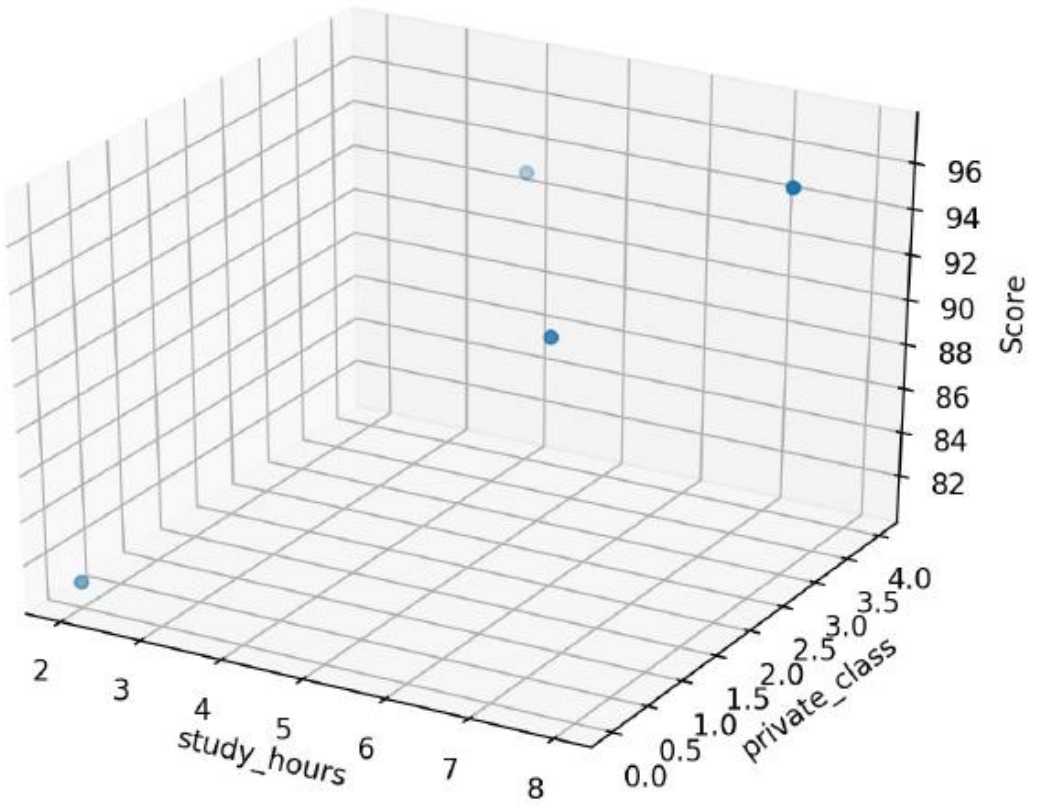


그림 4-6 축이 하나 더 늘어 3D로 배치된 모습

5 | 코딩으로 확인하는 다중 선형 회귀

- 이제 x 가 두 개가 되었으므로 x_1 과 x_2 두 가지의 변수를 지정함
- 각각의 값에 기울기 a 값이 다르므로 기울기도 a_1 과 a_2 이렇게 두 가지를 만듦
- 각각 앞서 했던 방법과 같은 방법으로 경사 하강법을 적용하고 학습률을 곱해 기존의 값을 업데이트 함

```
y_pred = a1 * x1_data + a2 * x2_data + b  # y를 구하는 식을 세우기
error = y_data - y_pred  # 오차를 구하는 식
a1_diff = -(1/len(x1_data)) * sum(x1_data * (error)) # 오차 함수를 a1로 미분한 값
a2_diff = -(1/len(x2_data)) * sum(x2_data * (error))  # 오차 함수를 a2로 미분한 값
```

5 | 코딩으로 확인하는 다중 선형 회귀

```
b_new = -(1/len(x1_data)) * sum(y_data - y_pred)  # 오차 함수를 b로 미분한 값
a1 = a1 - lr * a1_diff   # 학습률을 곱해 기존의 a1 값 업데이트
a2 = a2 - lr * a2_diff   # 학습률을 곱해 기존의 a2 값 업데이트
b = b - lr * b_diff      # 학습률을 곱해 기존의 b 값 업데이트
```

5 | 코딩으로 확인하는 다중 선형 회귀

코드 4-2 다중 선형 회귀 실습

- 예제 소스: deeplearning_class/04_Multi-Linear-Regression.ipynb

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d


# 공부 시간 X와 성적 Y의 리스트 만들기
data = [[2, 0, 81], [4, 4, 93], [6, 2, 91], [8, 3, 97]]
x1 = [i[0] for i in data]
x2 = [i[1] for i in data]
y = [i[2] for i in data]
```

5 | 코딩으로 확인하는 다중 선형 회귀

```
# 그래프로 확인
ax = plt.axes(projection='3d')
ax.set_xlabel('study_hours')
ax.set_ylabel('private_class')
ax.set_zlabel('Score')
ax.dist = 11
ax.scatter(x1, x2, y)
plt.show()

# 리스트로 되어 있는 x와 y 값을 넘파이 배열로 바꾸기(인덱스로 하나씩 불러와 계산할 수 있도록 하기 위함)
x1_data = np.array(x1)
x2_data = np.array(x2)
y_data = np.array(y)
```

5 | 코딩으로 확인하는 다중 선형 회귀


```
# 기울기 a와 절편 b의 값 초기화
a1 = 0
a2 = 0
b = 0

# 학습률
lr = 0.05

# 몇 번 반복할지 설정(0부터 세므로 원하는 반복 횟수에 +1)
epochs = 2001
```

5 | 코딩으로 확인하는 다중 선형 회귀


```
# 경사 하강법 시작
for i in range(epochs): # epoch 수 만큼 반복
    y_pred = a1 * x1_data + a2 * x2_data + b  # y를 구하는 식 세우기
    error = y_data - y_pred  # 오차를 구하는 식
    # 오차 함수를 a1로 미분한 값
    a1_diff = -(1/len(x1_data)) * sum(x1_data * (error))
    # 오차 함수를 a2로 미분한 값
    a2_diff = -(1/len(x2_data)) * sum(x2_data * (error))
    # 오차 함수를 b로 미분한 값
    b_new = -(1/len(x1_data)) * sum(y_data - y_pred)
    a1 = a1 - lr * a1_diff  # 학습률을 곱해 기존의 a1 값 업데이트
```

5 | 코딩으로 확인하는 다중 선형 회귀

```
    a2 = a2 - lr * a2_diff   # 학습률을 곱해 기존의 a2 값 업데이트
    b = b - lr * b_diff      # 학습률을 곱해 기존의 b값 업데이트

    if i % 100 == 0:         # 100번 반복될 때마다 현재의 a1, a2, b 값 출력
        print("epoch=%.f, 기울기1=%.04f, 기울기2=%.04f, 절편=%.04f"
% (i, a1, a2, b))
```

5 | 코딩으로 확인하는 다중 선형 회귀

실행 결과

```
epoch=0, 기울기 1=23.2000, 기울기 2=10.5625, 절편=4.5250
epoch=100, 기울기 1=6.4348, 기울기 2=3.9893, 절편=43.9757
epoch=200, 기울기 1=3.7255, 기울기 2=3.0541, 절편=62.5766
epoch=300, 기울기 1=2.5037, 기울기 2=2.6323, 절편=70.9656
epoch=400, 기울기 1=1.9527, 기울기 2=2.4420, 절편=74.7491
epoch=500, 기울기 1=1.7042, 기울기 2=2.3562, 절편=76.4554
(중략)
epoch=1500, 기울기 1=1.5001, 기울기 2=2.2857, 절편=77.8567
epoch=1600, 기울기 1=1.5000, 기울기 2=2.2857, 절편=77.8569
epoch=1700, 기울기 1=1.5000, 기울기 2=2.2857, 절편=77.8570
epoch=1800, 기울기 1=1.5000, 기울기 2=2.2857, 절편=77.8571
epoch=1900, 기울기 1=1.5000, 기울기 2=2.2857, 절편=77.8571
epoch=2000, 기울기 1=1.5000, 기울기 2=2.2857, 절편=77.8571
```

5 | 코딩으로 확인하는 다중 선형 회귀

- 다중 선형 회귀 문제에서의 기울기 a_1, a_2와 절편 b의 값을 찾아 확인할 수 있음

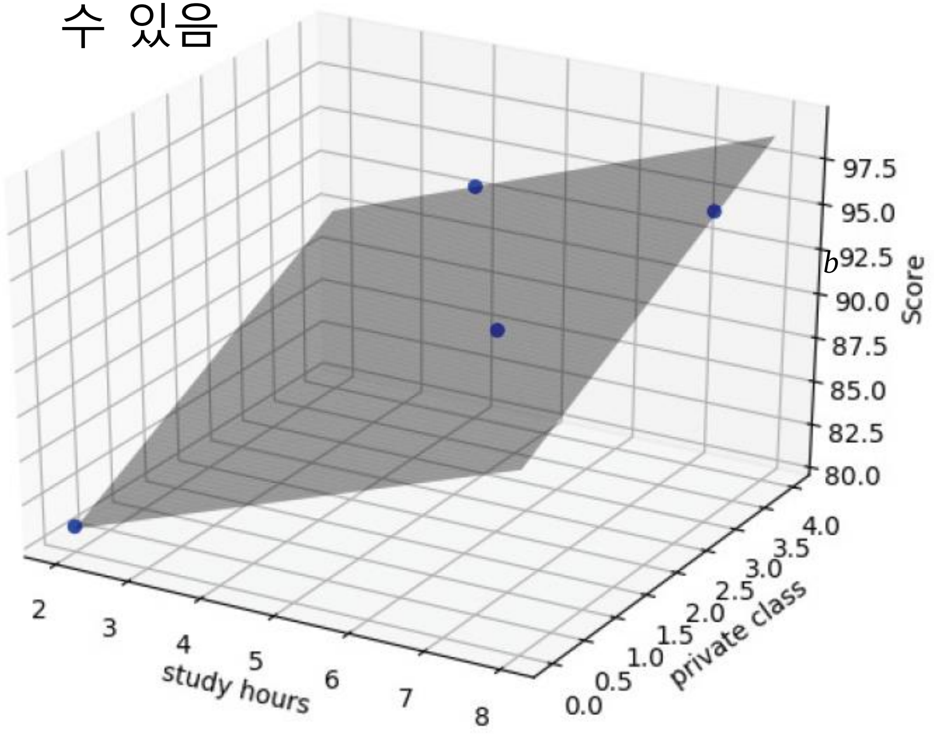


그림 4-7 다중 선형 회귀의 그래프: 차원이 하나 더 늘어난 모습

5 | 코딩으로 확인하는 다중 선형 회귀

- 1차원 예측 직선이 3차원 '예측 평면'으로 바뀜
- 과외 수업 횟수(privateclass)라는 새로운 변수가 추가됨
- 1차원 직선에서만 움직이던 예측 결과가 더 넓은 평면 범위 안에서 움직이게 됨
- 이로 인해 좀 더 정밀한 예측을 할 수 있게 된 것임

5장 참 거짓 판단 장치: 로지스틱 회귀

5 참 거짓 판단 장치: 로지스틱 회귀

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

5 참 거짓 판단 장치: 로지스틱 회귀

- 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
- 누군가 비슷한 질문을 하면 지금까지 만들어 놓은 이 모델을 꺼내어 답을 함

  → 이것이 바로 딥러닝의 동작 원리임

1 | 로지스틱 회귀의 정의

- 직선으로 해결하기에는 적절하지 않은 경우도 있음
- 점수가 아니라 오직 합격과 불합격만 발표되는 시험이 있다고 하자

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

표 5-1 공부한 시간에 따른 합격 여부

1 | 로지스틱 회귀의 정의

- 합격을 1, 불합격을 0이라 하고, 이를 좌표 평면에 표현하면 그림 5-1과 같음

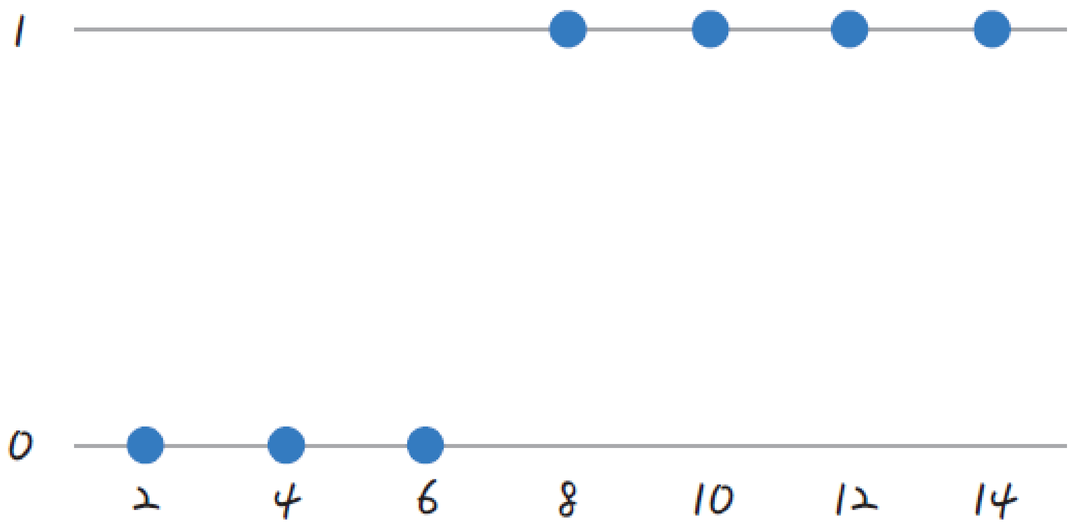


그림 5-1 합격과 불합격만 있을때의 좌표 표현

1 | 로지스틱 회귀의 정의

- 이 점들은 1과 0 사이의 값이 없으므로 직선으로 그리기가 어려움
- 점들의 특성을 정확하게 담아내려면 직선이 아니라 다음과 같이 S자

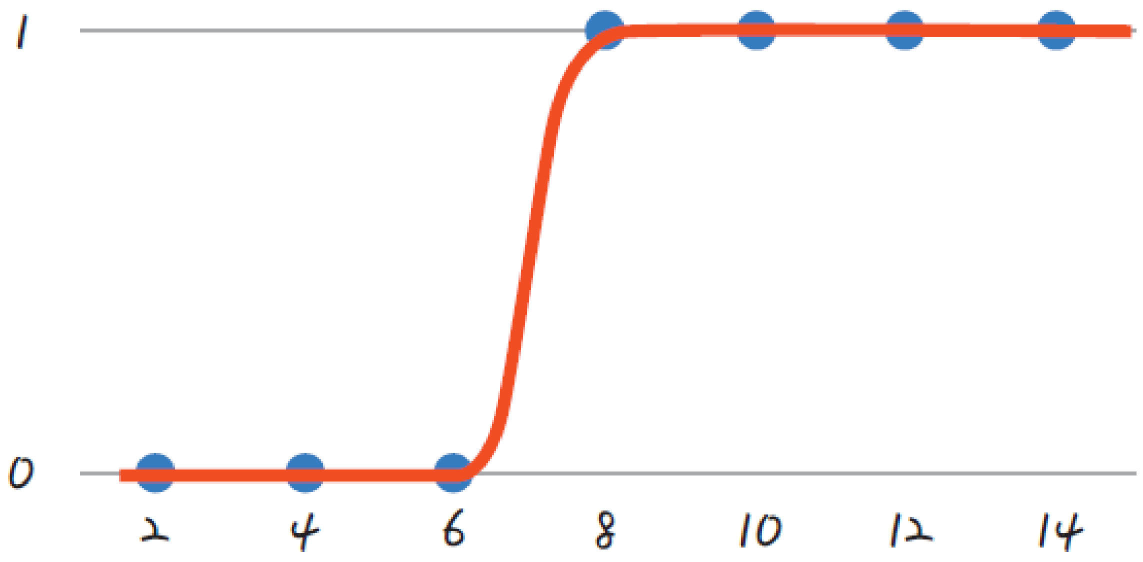


그림 5-2 각 점의 특성을 담은 선을 그었을 때

1 | 로지스틱 회귀의 정의

- 로지스틱 회귀 :

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

2 | 시그모이드 함수

- 시그모이드 함수(sigmoid function) :

  S자 형태로 그래프가 그려지는 함수

- 시그모이드 함수를 이용해 로지스틱 회귀를 풀어나가는 공식은 다음과

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

2 | 시그모이드 함수

- a 는 그래프의 경사도를 결정함

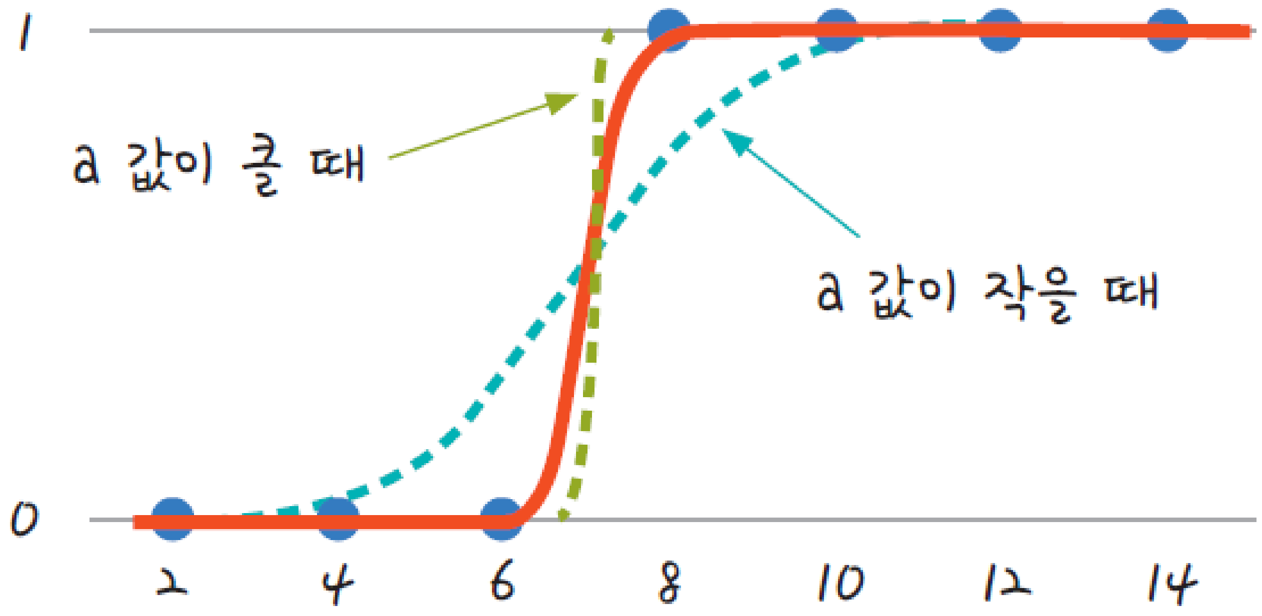


그림 5-3 a 값이 클 때와 작을 때의 그래프 변화

2 | 시그모이드 함수

- b 그래프의 좌우 이동을 의미함

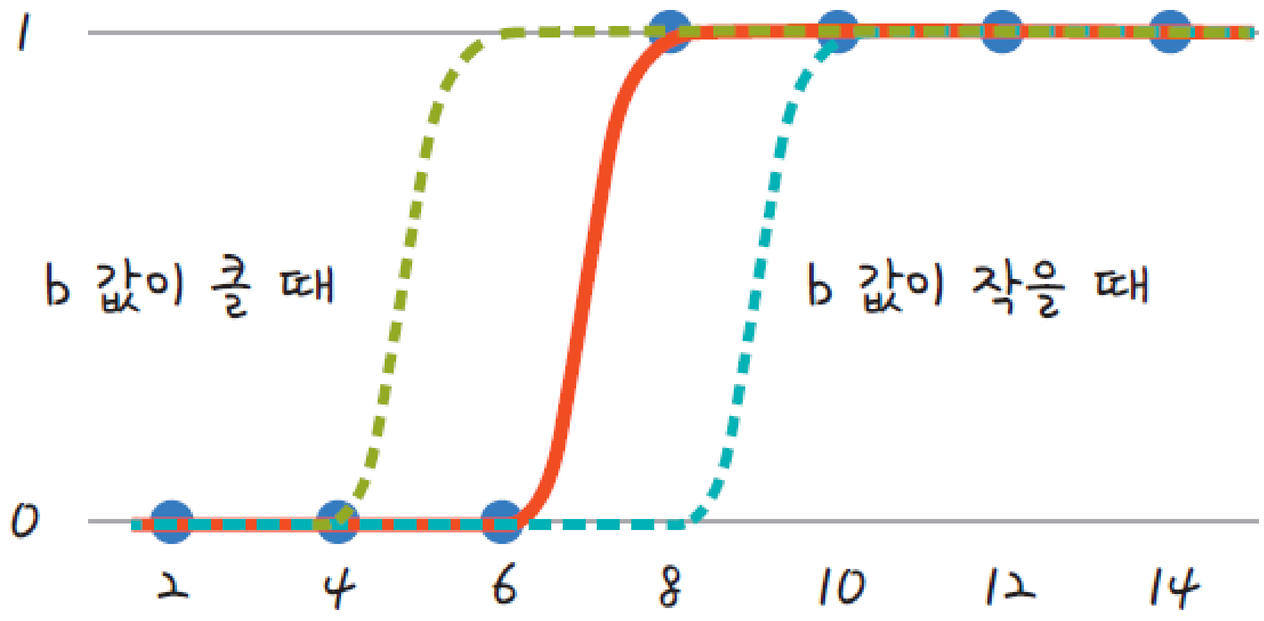


그림 5-4 b 값이 클 때와 작을 때의 그래프 변화

2 | 시그모이드 함수

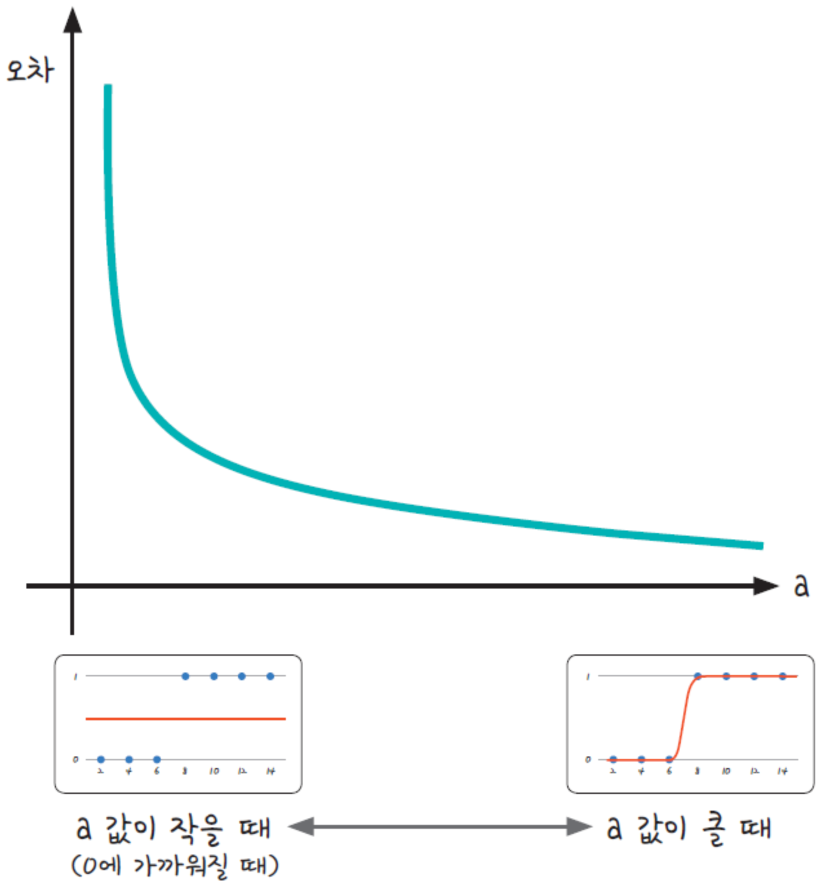


그림 5-5 a와 오차와의 관계:a가 작아질수록 오차는 무한대로 커지지만, a가 커진다고 해서 오차가 없어지지는 않는다.

2 | 시그모이드 함수

- b 값에 따른 오차의 그래프는 그림 5-6과 같음

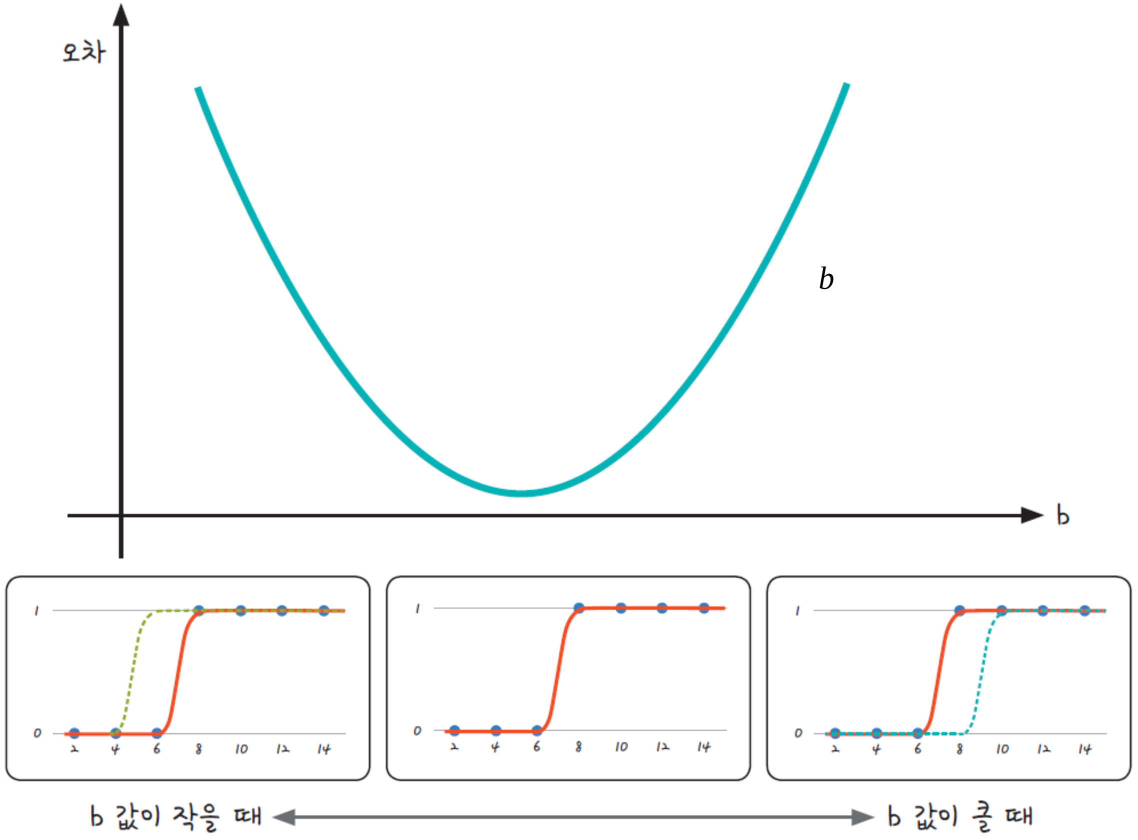


그림 5-6 b와 오차와의 관계: b 값이 너무 작아지거나 커지면 오차도 이에 따라 커진다.

3 | 오차 공식

- 경사 하강법은 먼저 오차를 구한 다음 오차가 작은 쪽으로 이동시키는 방법

- 함

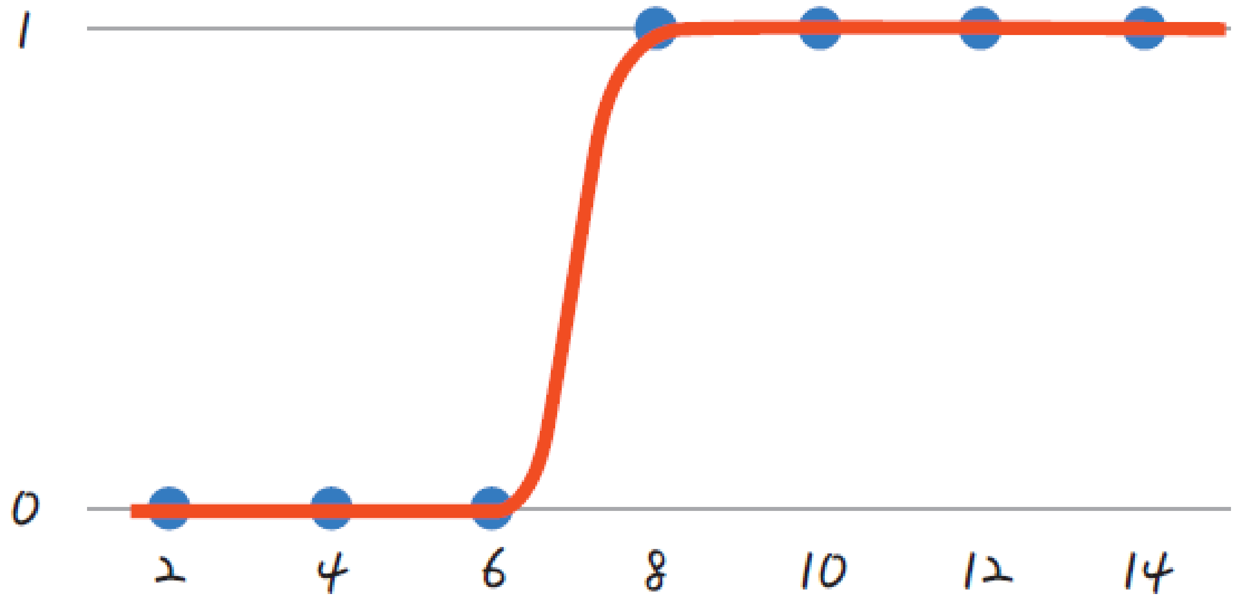


그림 5-7 시그모이드 함수 그래프

3 | 오차 공식

- 시그모이드 함수의 특징은 y 값이 0과 1 사이라는 것임
- 실제 값이 1일 때 예측 값이 0에 가까워지면 오차가 커짐
- 반대로, 실제 값이 0일 때 예측 값이 1에 가까워지는 경우에도 오차는 커짐
- 이를 공식으로 만들 수 있게 해 주는 함수가 바로 로그 함수임

4 | 로그 함수

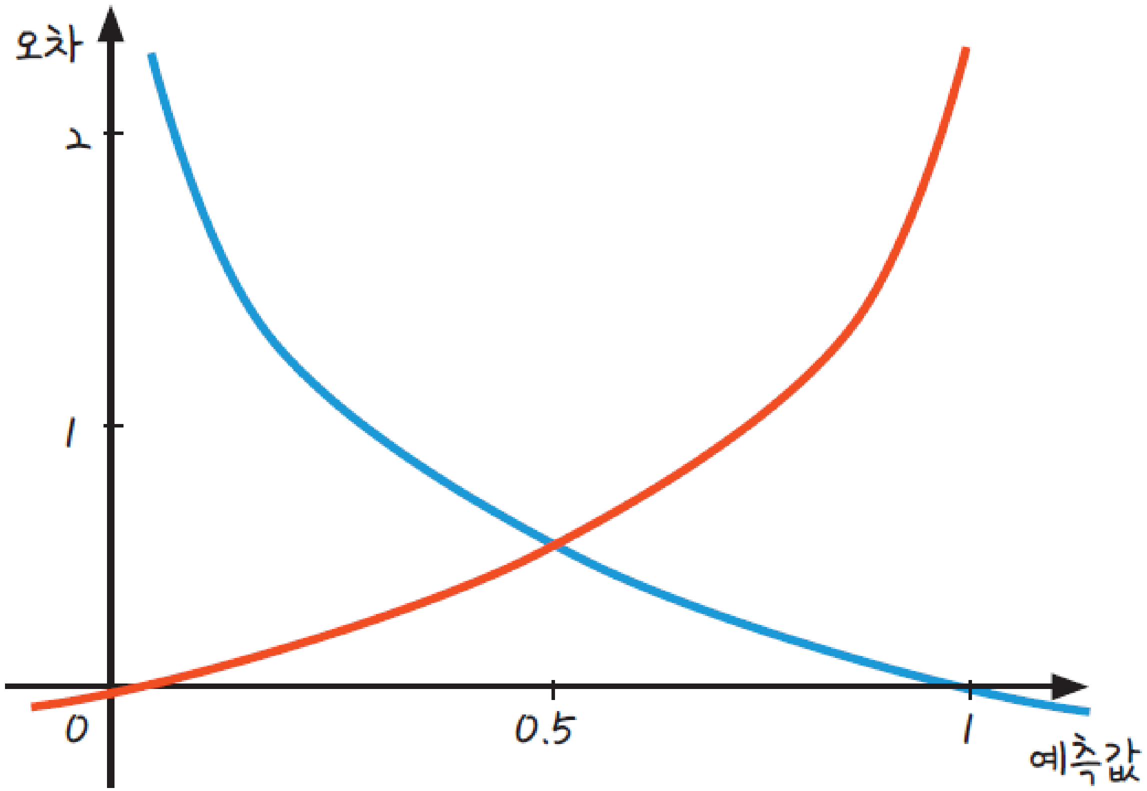


그림 5-8 실제 값이 1일 때(파란색)와 0일 때(빨간색) 로그 함수 그래프

4 | 로그 함수

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

4 | 로그 함수

- 파란색과 빨간색 그래프의 식은 각각 $-\log h$ 와 $-\log(1-h)$ 임
- 실제 값이 1일 때는 $-\log h$ 그래프를 쓰고, 0일 때는 $-\log(1-h)$ 그래프를 써야 함

$$-\{\underbrace{y\_data \log h}_{\mathrm{A}} + \underbrace{(1-y\_data)\log(1-h)}_{\mathrm{B}}\}$$

- 실제 값을 y_data라 할 때, 이 값이 1이면 B 부분이 없어짐
- 반대로 0이면 A부분이 없어짐
- 실제 값에 따라 빨간색 그래프와 파란색 그래프를 각각 사용할 수 있게 됨

5 | 코딩으로 확인하는 로지스틱 회귀

- 로지스틱 회귀를 위해서는
  - 시그모이드 함수를 사용한다는 것
  - 0부터 1사이의 값을 가지는 특성 때문에 로그 함수를 함께 써야 한다는 것
- 이제 경사 하강법을 이용해 a와 b의 최적 값을 찾는 과정을 해 볼 차례임

```
data = [[2, 0], [4, 0], [6, 0], [8, 1], [10, 1], [12, 1], [14, 1]]

x_data = [i[0] for i in data]    # 공부한 시간 데이터
y_data = [i[1] for i in data]    # 합격 여부
```

5 | 코딩으로 확인하는 로지스틱 회귀

```
import matplotlib.pyplot as plt


plt.scatter(x_data, y_data)
plt.xlim(0, 15)
plt.ylim(-.1, 1.1)
```

5 | 코딩으로 확인하는 로지스틱 회귀

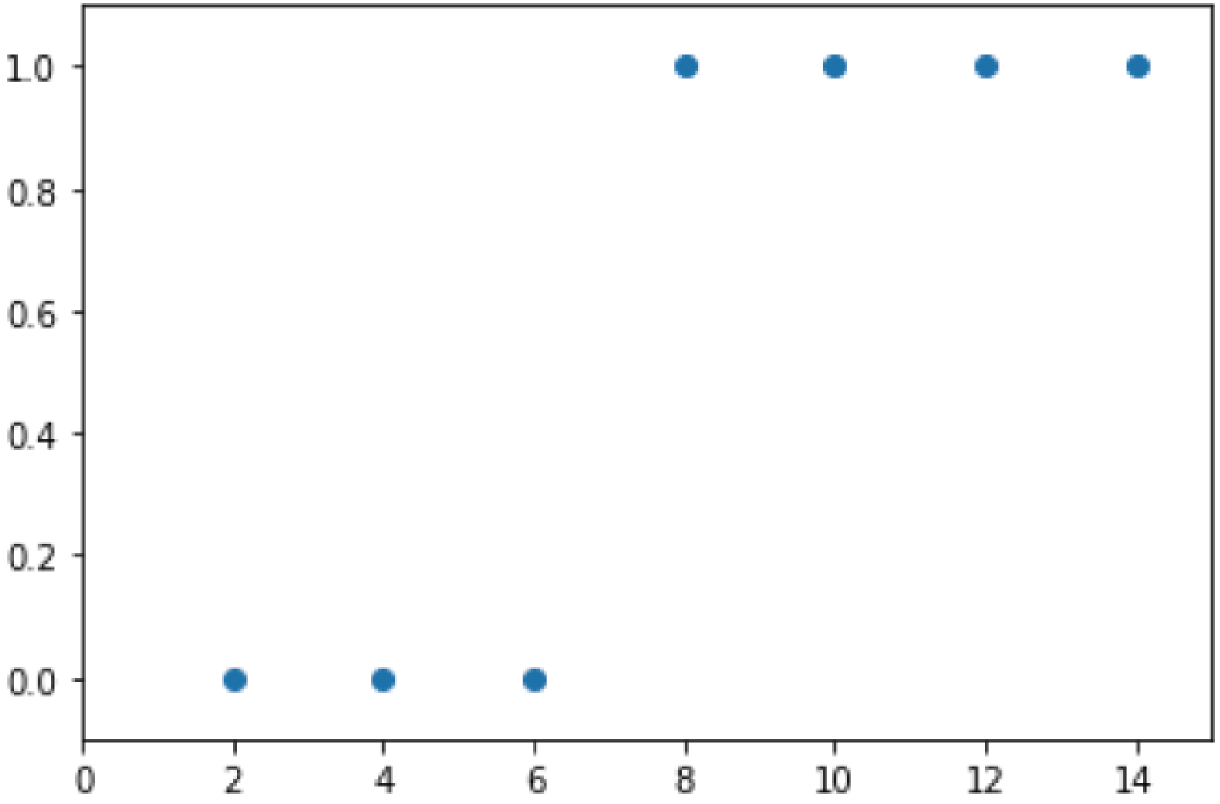


그림 5-9 공부 시간에 따른 합격 여부를 그래프로 나타낸 모습

5 | 코딩으로 확인하는 로지스틱 회귀

- 기울기와 a와 절편 b의 값을 초기화하고 학습률을 정함
- 학습률은 임의로 0.05로 정함

```
a = 0
b = 0

lr = 0.05   # 학습률
```

```
def sigmoid(x):                        # sigmoid라는 이름의 함수 정의
    return 1 / (1 + np.e ** (-x))  # 시그모이드 식의 형태 그대로 파이썬으로 옮김
```

5 | 코딩으로 확인하는 로지스틱 회귀

- 경사 하강법을 시행할 순서임
- a와 b로 편미분한 값(a_diff, b_diff)에 학습률(lr)을 곱하여 각각 업데이트 하는 방법은 앞서 배운 선형 회귀와 같음
- 다만 오차를 구하는 함수가 다르므로 a_diff과 b_diff를 결정하는 식에 변화가 생김
- 로지스틱 회귀의 오차 함수는 조금 전에 배운 것을 이용해 다음과 같은 식으로 표시할 수 있음

$$-\frac{1}{n}\sum y\log h+(1-y)\log(1-h))$$

5 | 코딩으로 확인하는 로지스틱 회귀

- 이 식을 편미분한 후 파이썬 코드로 옮겨 놓은 결과가 다음과 같다는 것은 기억하기 바람

```
for i in range(2001):
    for x_data, y_data in data:
        # a에 관한 편미분. 앞서 정의한 sigmoid 함수 사용
        a_diff = x_data*(sigmoid(a*x_data + b) - y_data)
        # b에 관한 편미분
        b_diff = sigmoid(a*x_data + b) - y_data
        # a를 업데이트 하기 위해 a – diff에 학습률 lr을 곱한 값을 a에서 뺌
        a = a - lr * a_diff
        # b를 업데이트 하기 위해 b – diff에 학습률 lr을 곱한 값을 b에서 뺌
        b = b - lr * b_diff
```

5 | 코딩으로 확인하는 로지스틱 회귀

코드 5-1 코딩으로 확인하는 로지스틱 회귀

- 예제 소스: deeplearnig_class/05_Logistic_regression.ipynb

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# 공부 시간 X와 합격 여부 Y의 리스트 만들기
data = [[2, 0], [4, 0], [6, 0], [8, 1], [10, 1], [12, 1], [14, 1]]

x_data = [i[0] for i in data]
y_data = [i[1] for i in data]
```

5 | 코딩으로 확인하는 로지스틱 회귀

```
# 그래프로 나타내기
plt.scatter(x_data, y_data)
plt.xlim(0, 15)
plt.ylim(-.1, 1.1)

# 기울기 a와 절편 b의 값 초기화
a = 0
b = 0

#학습률
lr = 0.05
```

5 | 코딩으로 확인하는 로지스틱 회귀

```
# 시그모이드 함수 정의
def sigmoid(x):
    return 1 / (1 + np.e ** (-x))

# 경사 하강법 실행
# 1,000번 반복될 때마다 각 x_data 값에 대한 현재의 a 값, b 값 출력
for i in range(2001):
    for x_data, y_data in data:
        a_diff = x_data*(sigmoid(a * x_data + b) - y_data)
        b_diff = sigmoid(a * x_data + b) - y_data
        a = a - lr * a_diff
        b = b - lr * b_diff
        if i % 1000 == 0:
            print("epoch=%.f, 기울기=%.04f, 절편=%.04f" % (i, a, b))
```

5 | 코딩으로 확인하는 로지스틱 회귀

```
# 앞서 구한 기울기와 절편을 이용해 그래프 그리기
plt.scatter(x, y)
plt.xlim(0, 15)
plt.ylim(-.1, 1.1)
x_range = (np.arange(0, 15, 0.1))  # 그래프로 나타낼 x 값의 범위 정하기
plt.plot(np.arange(0, 15, 0.1), np.array([sigmoid(a * x + b)
for x in x_range]))
plt.show( )
```

5 | 코딩으로 확인하는 로지스틱 회귀

실행 결과

```
epoch=0, 기울기=-0.0500, 절편=-0.0250
epoch=0, 기울기=-0.1388, 절편=-0.0472
epoch=0, 기울기=-0.2268, 절편=-0.0619
epoch=0, 기울기=0.1201, 절편=-0.0185
epoch=0, 기울기=0.2374, 절편=-0.0068
epoch=0, 기울기=0.2705, 절편=-0.0040
epoch=0, 기울기=0.2860, 절편=-0.0029
epoch=1000, 기울기=1.4978, 절편=-9.9401
epoch=1000, 기울기=1.4940, 절편=-9.9411
epoch=1000, 기울기=1.4120, 절편=-9.9547
epoch=1000, 기울기=1.4949, 절편=-9.9444
epoch=1000, 기울기=1.4982, 절편=-9.9440
```

5 | 코딩으로 확인하는 로지스틱 회귀

```
epoch=1000, 기울기=1.4984, 절편=-9.9440
epoch=1000, 기울기=1.4985, 절편=-9.9440
epoch=2000, 기울기=1.9065, 절편=-12.9489
epoch=2000, 기울기=1.9055, 절편=-12.9491
epoch=2000, 기울기=1.8515, 절편=-12.9581
epoch=2000, 기울기=1.9057, 절편=-12.9514
epoch=2000, 기울기=1.9068, 절편=-12.9513
epoch=2000, 기울기=1.9068, 절편=-12.9513
epoch=2000, 기울기=1.9068, 절편=-12.9513
```

5 | 코딩으로 확인하는 로지스틱 회귀

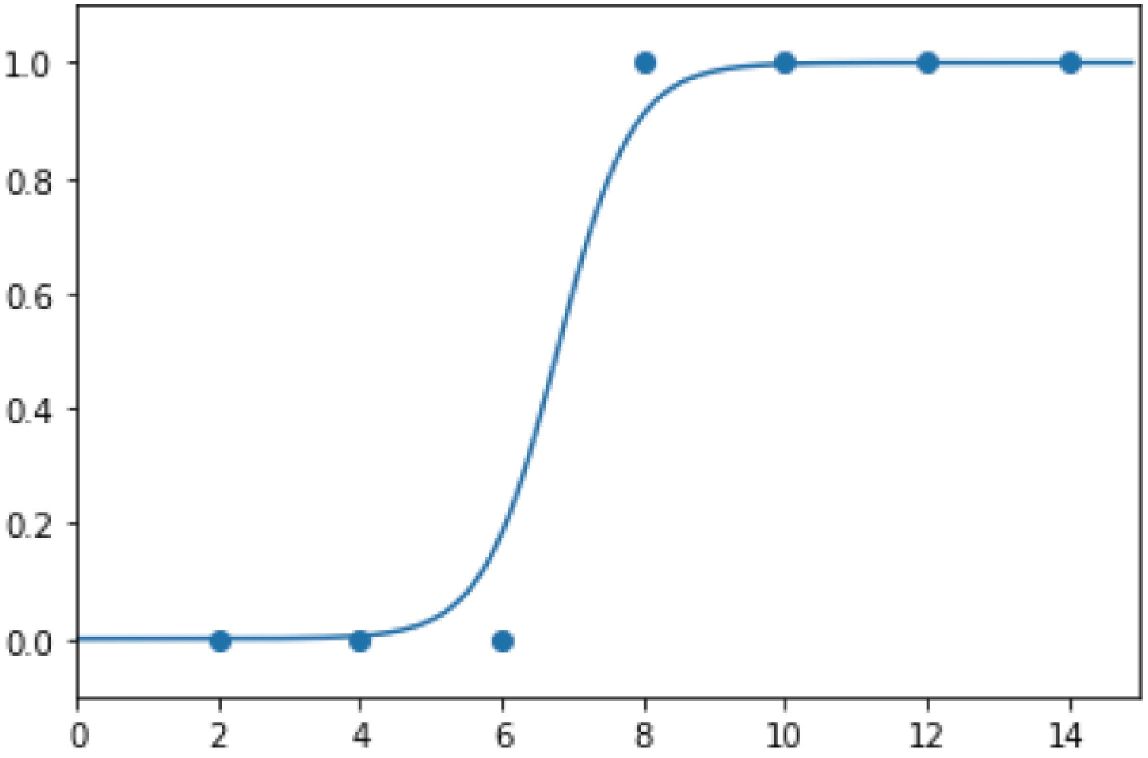


그림 5-10 시그모이드 형태로 출력된 그래프

5 | 코딩으로 확인하는 로지스틱 회귀

- 시그모이드 형태의 함수가 잘 만들어지도록 a와 b의 값이 수렴된 것을 알 수 있음
- 만약 여기에 입력 값이 추가되어 세 개 이상의 입력 값을 다룬다면 시그모이드 함수가 아니라 소프트맥스(softmax)라는 함수를 써야 함

6 | 로지스틱 회귀에서 퍼셉트론으로

- 입력 값을 통해 출력 값을 구하는 함수 y는 다음과 같이 표현할 수 있음

$$y = a_1x_1 + a_2x_2 + b$$

- 입력 값 :

  우리가 가진 값은 x_1과 x_2

- 출력 값 :

  계산으로 얻는 값 y

- 즉, 출력 값을 구하려면 a_1값, a_2값 그리고 b값이 필요함

6 | 로지스틱 회귀에서 퍼셉트론으로

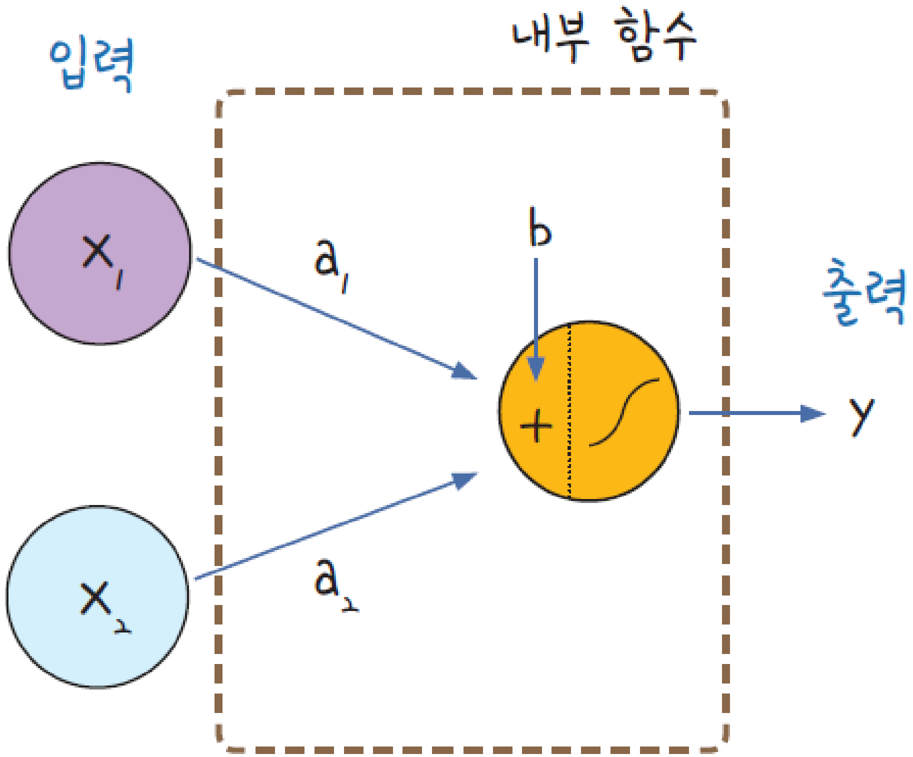


그림 5-11 로지스틱 회귀를 퍼셉트론 방식으로 표현한 예