6. 참 거짓 판단 장치 : 로지스틱 회귀

서론

- 법정 드라마나 영화를 보면 검사가 피고인을 다그치는 장면이 종종 나옵니다.
- 검사의 예리한 질문에 피고인이 당황한 표정으로 변명을 늘어놓을 때 검사가 이렇게 소리칩니다.
- “예, 아니오로만 대답하세요!"
- 때로 할 말이 많아도 예 혹은 아니오로만 대답해야 할 때가 있습니다.
- 그런데 실은 이와 같은 상황이 딥러닝에서도 끊임없이 일어납니다.

서론

- 전달받은 정보를 놓고 참과 거짓중에 하나를 판단해 다음 단계로 넘기는 장치들이 딥러닝 내부에서 쉬지 않고 작동하는 것이지요.
- 딥러닝을 수행한다는 것은 겉으로 드러나지 않는 '미니 판단 장치'들을 이용해서 복잡한 연산을 해낸 끝에 최적의 예측 값을 내놓는 작업이라고 할 수 있습니다.

서론

- 이렇게 참과 거짓 중에 하나를 내놓는 과정은 로지스틱 회귀 (logistic regression) 의 원리를 거쳐 이루어집니다.
- 참인지 거짓인지를 구분하는 로지스틱 회귀의 원리를 이용해 '참, 거짓 미니 판단장치'를 만들어 주어진 입력 값의 특정을 추출합니다.
- 이를 저장해서 '모델 (model)'을 만듭니다.
- 그런 다음 누군가 비슷한 질문을 하면 지금까지 만들어 놓은 이 모델을 꺼내어 답을 합니다
- 이것이 바로 딥러닝의 동작원리입니다.
- 이제 딥러닝의 토대를 이루는 로지스틱 회귀에 대 하여 알아보겠습니다.

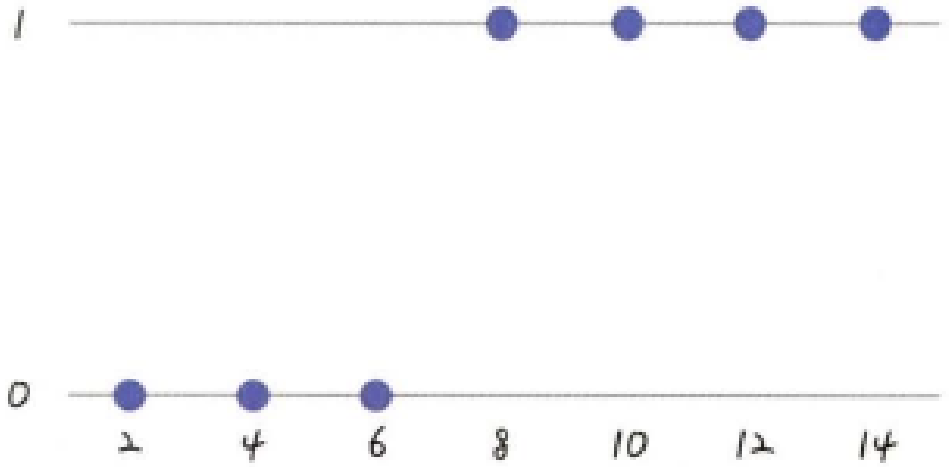
로지스틱 회귀의 정의

- 4장에서 공부한 시간과 성적 사이의 관계를 좌표에 나타냈을 때, 좌표의 형태가 직선으로 해결되는 선형 회귀를 사용하기에 적절했었음을 보았습니다.
- 그런데 직선으로 해결하기 에는 적절하지 않은 경우도 있습니다.
- 점수가 아니라 오직 합격과 불합격만 발표되는 시험이 있다고 합시다.
- 공부한 시간에 따른 합격 여부를 조사해 보니 표와 같았습니다.

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

로지스틱 회귀의 정의

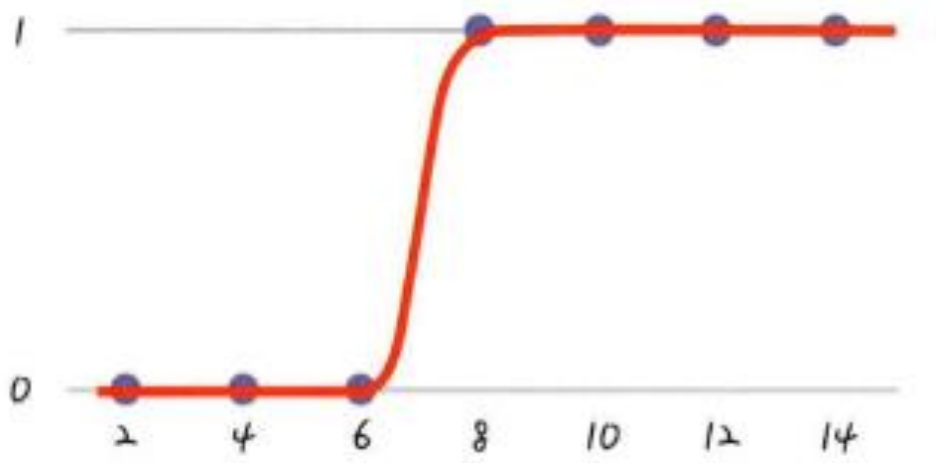
- 합격을 1, 불합격을 0이라 하고, 이를 좌표 평면에 표현하면 그림과 같습니다.



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

로지스틱 회귀의 정의

- 점들의 특성을 정확하게 담아내려면 직선이 아니라 다음과 같이 S자 형태여야 합니다.



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

시그모이드 함수

- 그런데 이러한 S자 형태로 그래프가 그려지는 함수가 있습니다.
- 바로 시그모이드 함수(sigmoid function) 입니다.
- 시그모이드 함수를 나타내는 방정식은 다음과 같습니다.

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

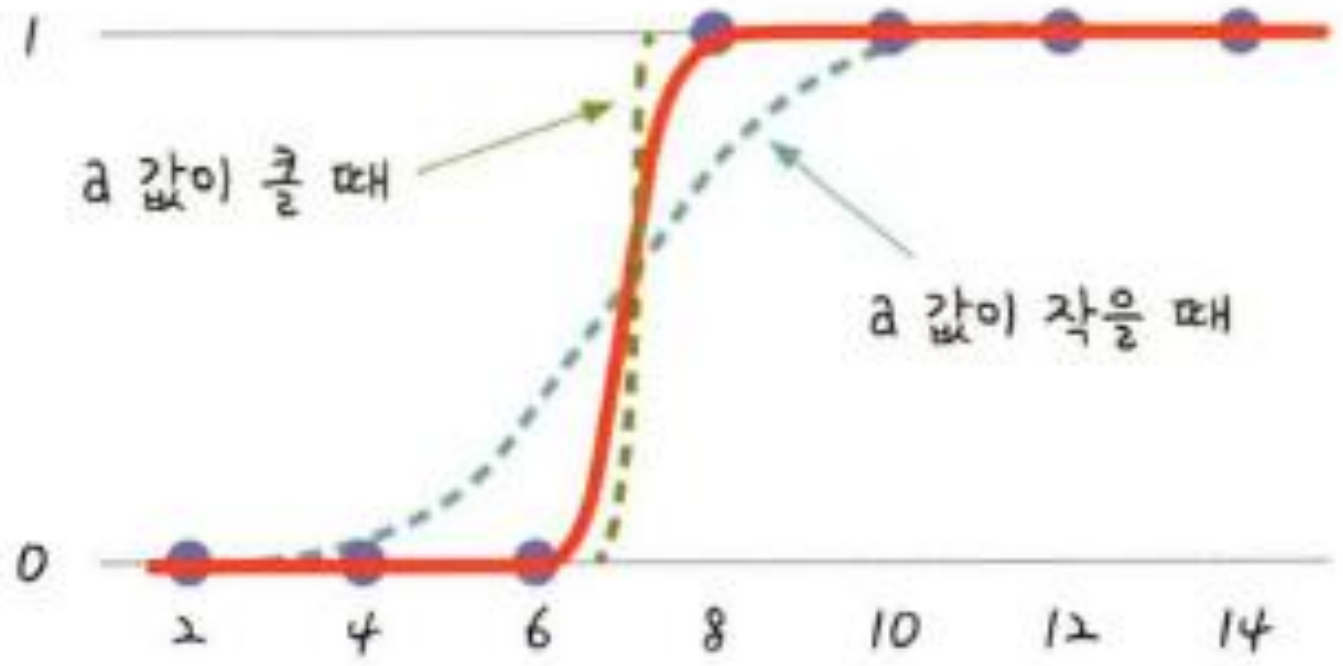
- 여기서 e는 자연 상수라고 불리는 무리수로 값은 2.71828 ... 입니다.
- 파이 (π)처럼 수학에서 중요하게 사용되는 상수로 고정된 값이므로 우리가 따로 구해야 하는 값은 아닙니다.

시그모이드 함수

- 우리가 구해야 하는 값은 결국 ax + b 입니다.
- 이제 이 식이 익숙하지요?
- 선형 회귀에서 우리가 구해야 하는 것이 a와 b였듯이 여기서도 마찬가지입니다.
- 앞서 구한 직선의 방정식과는 다르게 여기에서 a와 b는 어떤 의미를 지니고 있을까요?
- 먼저 a는 그래프의 경사도를 결정합니다.

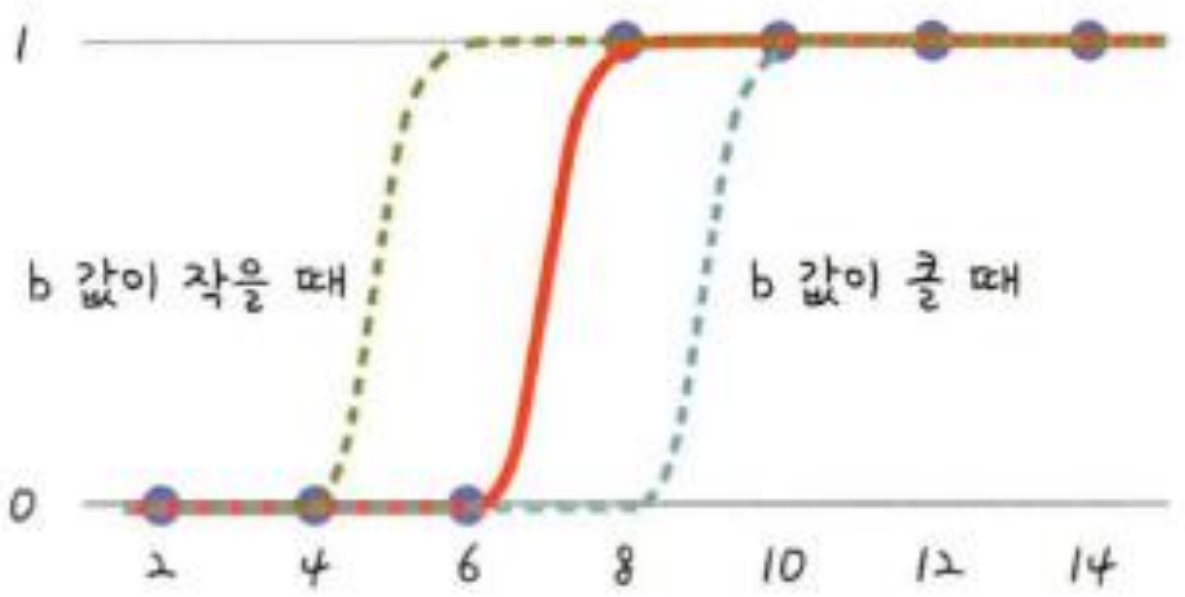
시그모이드 함수

- 그림과 같이 a 값이 커지면 경사가 커지고 a 값이 작아지면 경사가 작아집니다.



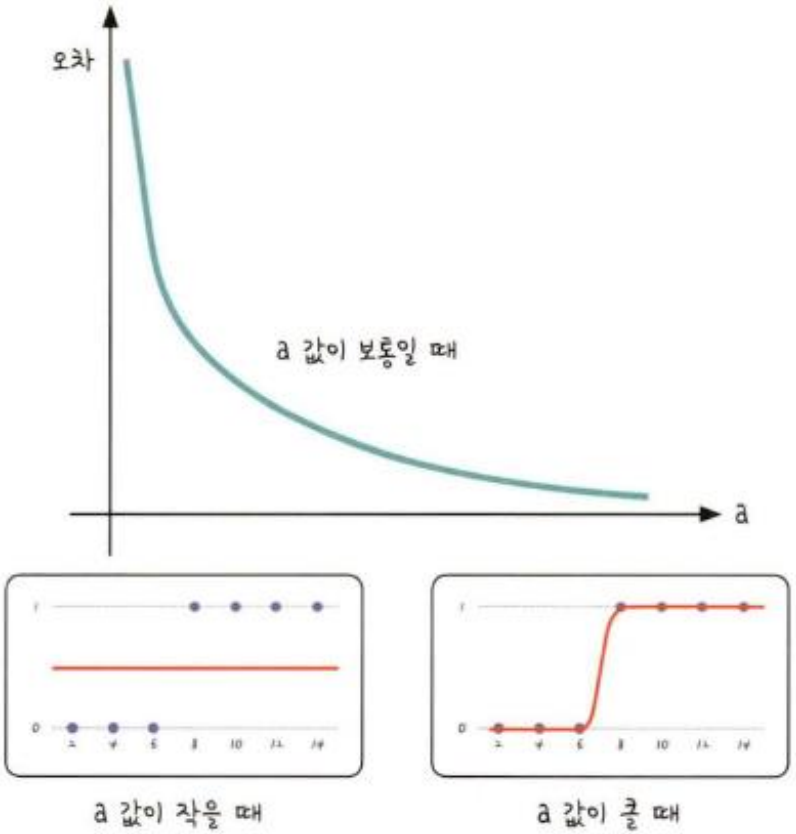
시그모이드 함수

- b는 그래프의 좌우 이동을 의미합니다.
- 그림과 같이 b 값이 크고 작아짐에 따라 그래프가 이동합니다.



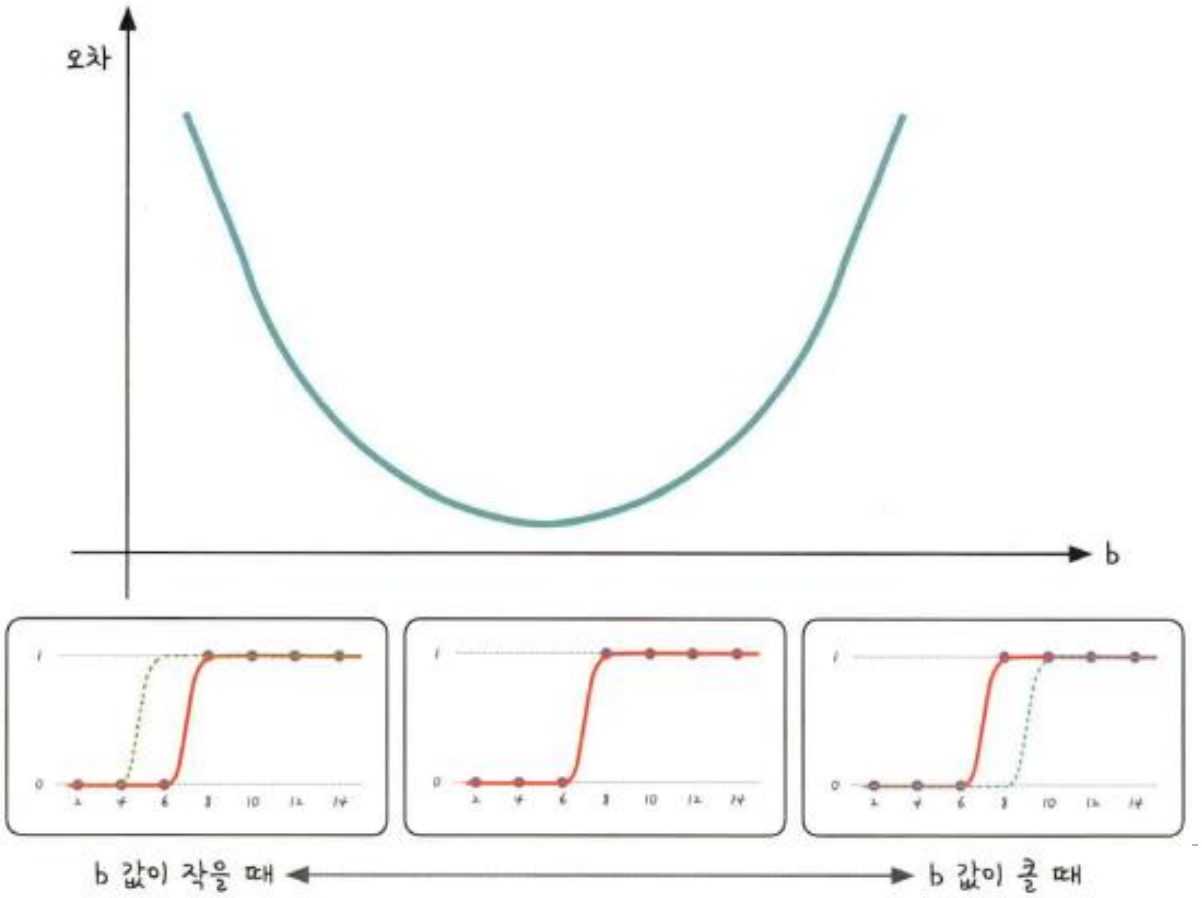
시그모이드 함수

- 따라서 a와 b의 값이 클수록 오차가 생깁니다.
- A 값이 크고 작아짐에 따라 오차는 다음과 같이 변합니다.
  - a 값이 작아지면 오차는 무한대로 커집니다.
  - 그런데 a 값이 커진다고 해서 오차가 무한대로 커지지는 않습니다.



시그모이드 함수

- 한편, b 값에 따른 오차의 그래프는 그림과 같습니다.
  - b 값은 너무 크거나 작을 경우 오차가 무한대로 커지므로 앞서와 마찬가지로 이차 함수 그래프로 표현할 수 있습니다.

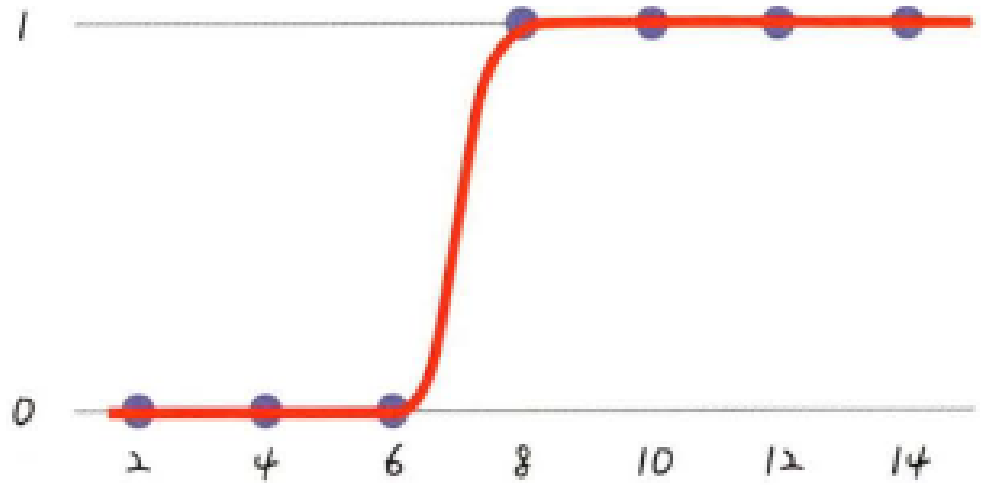


오차 공식

- 이제 우리에게 주어진 과제는 또다시 a와 b의 값을 구하는 것임을 알았습니다
- 시그모이드 함수에서 a와 b의 값을 어떻게 구해야 할까요? 답은 역시 경사 하강법입니다.
- 그런데 경사 하강법은 먼저 오차를 구한 다음 오차가 작은 쪽으로 이동시키는 방법이라고 했습니다.
- 그렇다면 이번에도 예측 값과 실제 값의 차이, 즉 오차를 구하는 공식이 필요합니다.

오차 공식

- 오차 공식을 도출하기 위해 시그모이드 함수 그래프의 특정을 다시 한번 살펴보겠습니다.



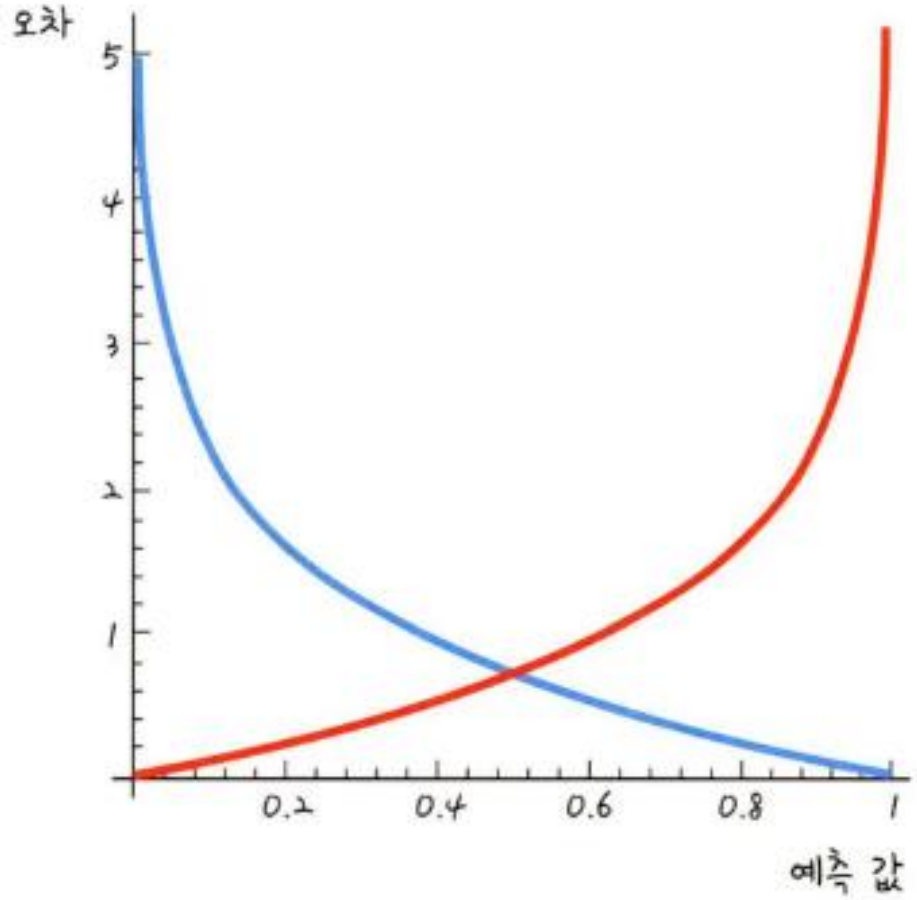
- 시그모이드 함수의 특징은 y 값이 0과 1 사이 라는 것입니다.
- 따라서 실제 값이 1일 때 예측 값이 0에 가까워지면 오차가 커져야 합니다.

오차 공식

- 반대로, 실제 값이 0 일 때 예측 값이 1 에 가까워지는 경우에도 오차는 커져야 합니다.
- 이를 공식으로 만들 수 있게 해 주는 함수가 바로 로그 함수입니다.

로그 함수

- 아래와 같이 y= 0.5 에 대칭하는 두 개의 로그 함수를 그려 보겠습니다.



로그 함수

- 파란색 선은 실제 값이 1일 때 사용할 수 있는 그래프입니다.
- 예측 값이 1 일 때 오차가 0 이고, 반대로 예측값이 0에 가까울수록 오차는 커집니다.
- 빨간색 선은 반대로 실제 값이 0 일 때 사용할 수 있는 함수입니다.
- 예측 값이 0 일 때 오차가 없고, 1에 가까워 질수록오차가 매우 커집니다.

로그 함수

- 앞의 파란색과 빨간색 그래프의 식은 각각 -log h와 -log(1 - h) 입니다.
- y의 실제 값이 1 일 때 - log h 그래프를 쓰고, 0 일 때 - log (1 - h) 그래프를 써야 합니다.
- 이는 다음과 같은 방법으로 해결할 수 있습니다.

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

- 실제 값 y가 1이면 B 부분이 없어집니다.
- 반대로 0 이면 A 부분이 없어집니다.
- 따라서 y 값에 따라 빨간색 그래프와 파란색 그래프를 각각 사용할 수 있게 됩니다.

코딩으로 확인하는 로지스틱 회귀

- 이를 그대로 텐서플로로 옮겨 보겠습니다.
- 먼저 텐서플로와 넘파이 라이브러리를 불러오고 x와 y값을 정해줍니다.

```
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[0] for y_row in data]
```

- a와 b의 값을 임의로 정합니다.
- 앞에서 배운 내용과 같습니다.

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

코딩으로 확인하는 로지스틱 회귀

- 이제 시그모이드 함수 방정식을 넘파이 라이브러리를 이용해 다음과 같이 작성합니다.

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

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

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

$$-\left(y \cdot \log h + (1-y)\log(1-h)\right)$$

- 오차를 구하고 그 평균값을 loss변수에 할당합니다.
- 평균을 구하기 위해 reduce_mean () 함수를 사용했습니다.

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

코딩으로 확인하는 로지스틱 회귀

- 이제 학습률을 지정하고 경사 하강법을 이용해 오차를 최소로 하는 값을 찾겠습니다.
- 앞서 배운 내용과 같습니다.

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

코딩으로 확인하는 로지스틱 회귀

- 마지막으로 텐서플로를 구동시켜 결과값을 출력하는 부분입니다.
- 앞서 배운 내용과 동일합니다.

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

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

코딩으로 확인하는 로지스틱 회귀

- 코드를 하나로 정리하면 다음과 같습니다.

```
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]

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

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

learning_rate = 0.5

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

코딩으로 확인하는 로지스틱 회귀

- 코드를 하나로 정리하면 다음과 같습니다.

```
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, y절편 b = %.4f" %(i, sess.run(loss), sess.run(a),
sess.run(b)))
```

코딩으로 확인하는 로지스틱 회귀

- 실행 결과는 다음과 같습니다.

```
Epoch: 0, loss = 4.0817, 기울기 a = 2.4706, y절편 b = -0.3620
Epoch: 6000, loss = 0.0152, 기울기 a = 2.9230, y절편 b = -20.3114
Epoch: 12000, loss = 0.0081, 기울기 a = 3.5648, y절편 b = -24.8081
Epoch: 18000, loss = 0.0055, 기울기 a = 3.9564, y절편 b = -27.5511
Epoch: 24000, loss = 0.0041, 기울기 a = 4.2385, y절편 b = -29.5268
Epoch: 30000, loss = 0.0033, 기울기 a = 4.4590, y절편 b = -31.0705
Epoch: 36000, loss = 0.0028, 기울기 a = 4.6399, y절편 b = -32.3371
Epoch: 42000, loss = 0.0024, 기울기 a = 4.7933, y절편 b = -33.4107
Epoch: 48000, loss = 0.0021, 기울기 a = 4.9263, y절편 b = -34.3424
Epoch: 54000, loss = 0.0019, 기울기 a = 5.0439, y절편 b = -35.1653
Epoch: 60000, loss = 0.0017, 기울기 a = 5.1491, y절편 b = -35.9020
```

- 오차(loss) 값이 점차 줄어들고 a와 b의 최적값을 찾아가는 것을 볼 수 있습니다.

여러 입력 값을 갖는 로지스틱 회귀

- 선형 회귀를 공부할 때와 마찬가지로 변수가 많아지면 더 정확하게 예측을 할 수 있습니다.
- 기본 개념은 설명했으므로 바로 코딩으로 표현해 보겠습니다.
- 먼저 변수를 하나 더 추가하여 x와 y 데이터를 만들어 줍니다.

```
x_data = [[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)
```

여러 입력 값을 갖는 로지스틱 회귀

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

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

- 플레이스 홀더는 입력값을 저장하는 일종의 그릇입니다.
- tf.placeholder('데이터형', '행렬의 차원', '이름')형태로 사용합니다.

여러 입력 값을 갖는 로지스틱 회귀

- 변수가 x에서 x_1, x_2로 추가되면 각각의 기울기 a_1, a_2도 계산해야 합니다.
- 즉, ax 부분이 $a_1x_1 + a_2x_2$ 로 바뀝니다.
- $a_1x_1 + a_2x_2$ 는 행렬곱을 이용해 $[a_1, a_2] * [x_1, x_2]$로도 표현할 수 있습니다.
- 텐서플로에서는 matmul() 함수를 이용해 행렬곱을 적용합니다.
- 또한, 이번에는 시그모이드를 계산하기 위해 텐서플로에 내장된 sigmoid () 함수를 사용해 보겠습니다.

여러 입력 값을 갖는 로지스틱 회귀

- 다음과 같이 구현합니다.

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

- 이전의 코드와 비교해 보기 바랍니다.

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

여러 입력 값을 갖는 로지스틱 회귀

- 코드를 하나로 정리하면 다음과 같습니다.

```
import tensorflow as tf
import numpy as np

seed = 0
np.random.seed(seed)
tf.set_random_seed(seed)

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)

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

a = tf.Variable(tf.random_uniform([2, 1], dtype = tf.float64))
b = tf.Variable(tf.random_uniform([1], dtype = tf.float64))

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

loss = -tf.reduce_mean(Y * tf.log(y) + (1 - Y) * tf.log(1- y))
```

여러 입력 값을 갖는 로지스틱 회귀

- 코드를 하나로 정리하면 다음과 같습니다.

```
learning_rate = 0.1

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

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

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_))
```

여러 입력 값을 갖는 로지스틱 회귀

- 결과를 보면, 오차(loss) 값이 점차 줄어들고 a_1, a_2와 b가 각각 최적값을 찾아가는 것이 보입니다.

```
step = 300, a1 = 0.7449, a2 = -0.4087, b = -2.6835, loss = 0.2515
step = 600, a1 = 0.7574, a2 = -0.1651, b = -4.0909, loss = 0.1824
step = 900, a1 = 0.6803, a2 = 0.1376, b = -5.1169, loss = 0.1438
step = 1200, a1 = 0.5873, a2 = 0.4245, b = -5.9331, loss = 0.1185
step = 1500, a1 = 0.4967, a2 = 0.6827, b = -6.6128, loss = 0.1005
step = 1800, a1 = 0.4137, a2 = 0.9120, b = -7.1959, loss = 0.0872
step = 2100, a1 = 0.3392, a2 = 1.1156, b = -7.7067, loss = 0.0769
step = 2400, a1 = 0.2729, a2 = 1.2972, b = -8.1613, loss = 0.0687
step = 2700, a1 = 0.2137, a2 = 1.4601, b = -8.5711, loss = 0.0621
step = 3000, a1 = 0.1608, a2 = 1.6072, b = -8.9440, loss = 0.0567
```

실제 값 적용하기

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

```
new_x = np.array([7, 6.]).reshape(1, 2)
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))
```

실제 값 적용하기

- 이를 실행하면 다음과 같이 출력됩니다.

```
공부한 시간: 7, 과외 수업 횟수: 6
합격 가능성:  95.38 %
```

- 7시간 공부하고 6 변의 과외를 받은 학생의 합격 가능성은 85.66%임을 알 수 있습니다.

로지스틱 회귀에서 퍼셉트론으로

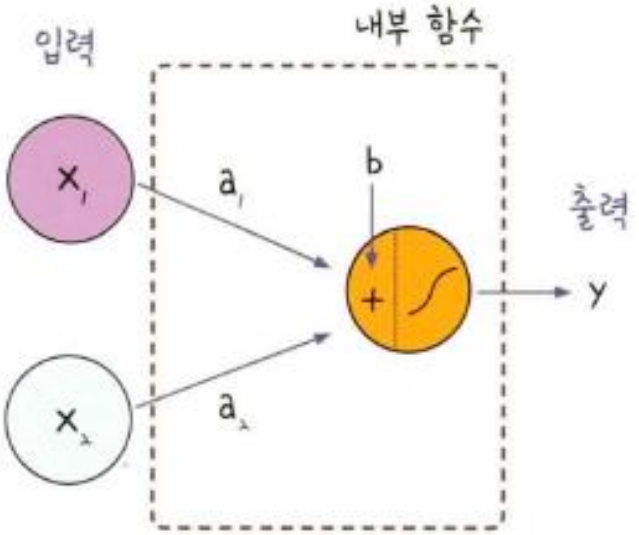
- 지금까지 배운 내용을 정리해 보겠습니다.
- 입력 값을 통해 출력 값을 구하는 함수 y는 다음과 같이 표현할 수 있습니다.

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

- 우리가 가진 값은 x_1과 x_2 입니다.
- 이를 입력 값이라고 합니다.
- 그리고 계산으로 얻는 값 y를 출력 값이라고 합니다.
- 즉, 출력 값을 구하려면 a 값과 b 값이 필요합니다.

로지스틱 회귀에서 퍼셉트론으로

- 이를 그림으로 나타내면 그림과 같습니다.



- x_1과 x_2가 입력되고, 각각 가중치 a_1, a_2를 만납니다.
- 여기에 b 값을 더한 후 시그모이드 함수를 거쳐 1 또는 0의 출력 값 y를 출력합니다.
- 우리가 지금까지 배운 내용을 그림은 한눈에 설명하고 있습니다.

로지스틱 회귀에서 퍼셉트론으로

- 그런데 이 그림을 아주 오래전에 그려서 발표한 사람이 있었습니다.
- 1957년, 코넬 항공 연구소의 프랑크 로젠블라트라는 사람은 이 개념을 고안해 발표하고 여기에 '퍼셉트론(perceptron)'이라는 이름을 붙였습니다.
- 이 퍼셉트론은 그 후 여러 학자들의 노력을 통해 인공 신경망, 오차 역전파 등의 발전을 거쳐 지금의 딥러닝으로 이어지게 됩니다.
- 다음 장에서는 퍼셉트론이 어떻게 딥러닝의 골격인 신경망을 구성하는지 설명 하도록 하겠습니다.

Q&A