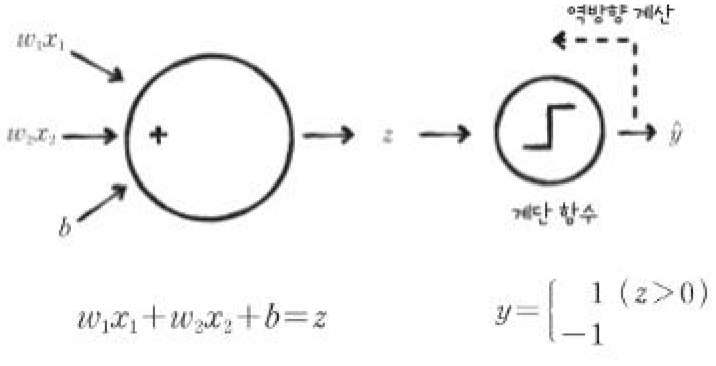
디지털 영상처리 연구실 연구보고서

정지우

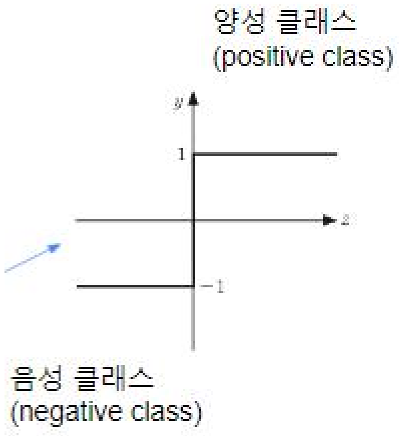
#이진 분류

퍼셉트론

우선 선형 회귀와 마찬가지로, 직선의 방정식을 활용한다.

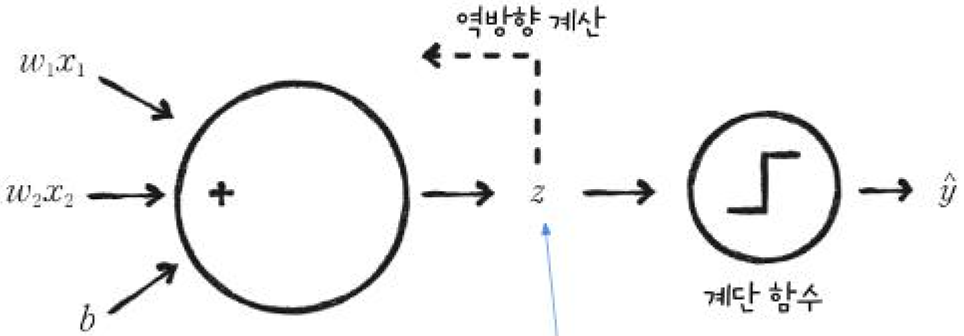


** 계단 함수 **



- 마지막 단계에서 샘플을 이진 분류 하기위해 계단 함수를 사용한다.
- 그리고 계단 함수를 통과한 값을 다시 가중치와 절편을 학습하는데 사용한다.

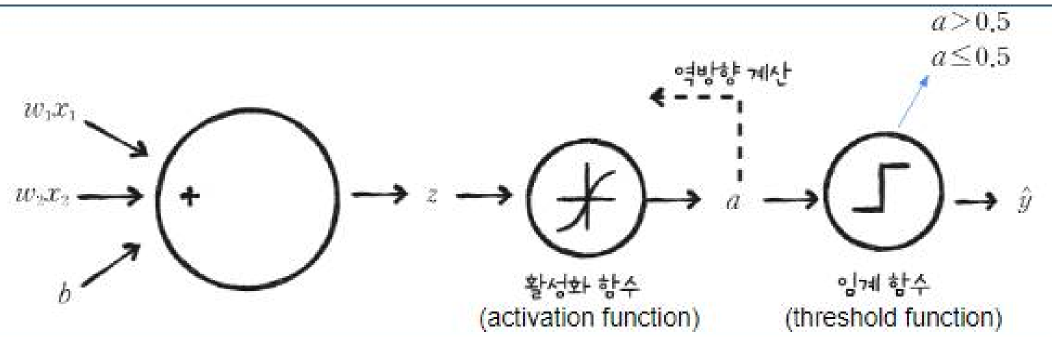
##아달린



선형 함수의 결과를 학습에 사용하는 특징이다.

##로지스틱 회귀

- 아달린에서 발전된 형태



- 중간에 활성화 함수를 통과한 값을 사용한다는 점이 다르고, 임계 함수는 아달린이나 퍼셉트론의 계단 함수와 거의 비슷하지만 활성화 함수의 출력값을 사용한다는점이 다르다.

활성화 함수는 비선형 함수를 사용한다.

만일, 활성화 함수가 선형함수이면)

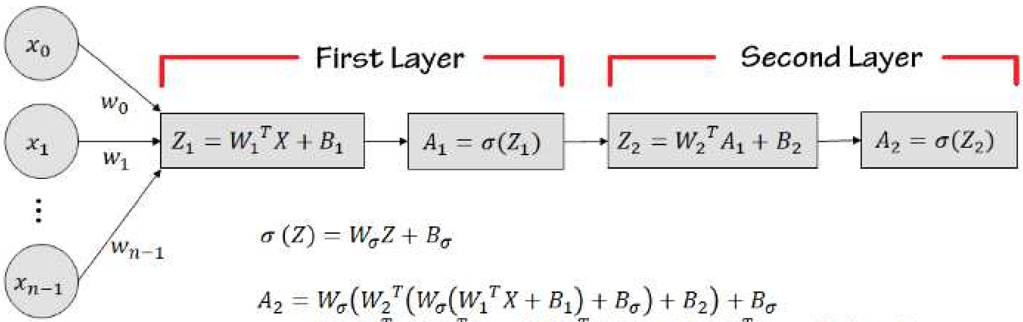
- 선형 함수 “$a = w_1x_1 + w_2x_2 + w_3x_3 + \ldots + w_nx_n$”
- 활성화 함수 “$y = ka$”

=> “ $y = k(w_1x_1 + w_2x_2 + w_3x_3 + \ldots + w_nx_n)$

즉, 다시 하나의 큰 선형 함수가 되기 때문에, 임계함수 앞에 여러개의 층을 쌓아도 결국 선형 함수라서 의미가 없다.

“선형 함수라서 의미가 없다”

=>



$$\sigma(Z) = W_\sigma Z + B_\sigma$$

$$\begin{aligned} A_2 &= W_\sigma(W_2{}^T(W_\sigma(W_1{}^T X + B_1) + B_\sigma) + B_2) + B_\sigma \\ &= \underbrace{W_\sigma W_2{}^T W_\sigma W_1{}^T}_{W} X + \underbrace{W_\sigma W_2{}^T W_\sigma B_1 + W_\sigma W_2{}^T B_\sigma + W_\sigma B_2 + B_\sigma}_{B} \\ &= WX + B \end{aligned}$$

2개의 Layer를 쌓아봤지만 X에 곱해지는 항들은 W로 치환가능하고,
입력과 무관한 상수들은 전체를 B로 치환 가능하기 때문에

WX+B라는 Single layer과 동일한 결과를 낸다.

다시말해 Deep 하게 쌓는 의미가 없어진다.

+) 그럼 비선형함수는 layer를 사용해서 쌓으면 의미가 있다는건데...

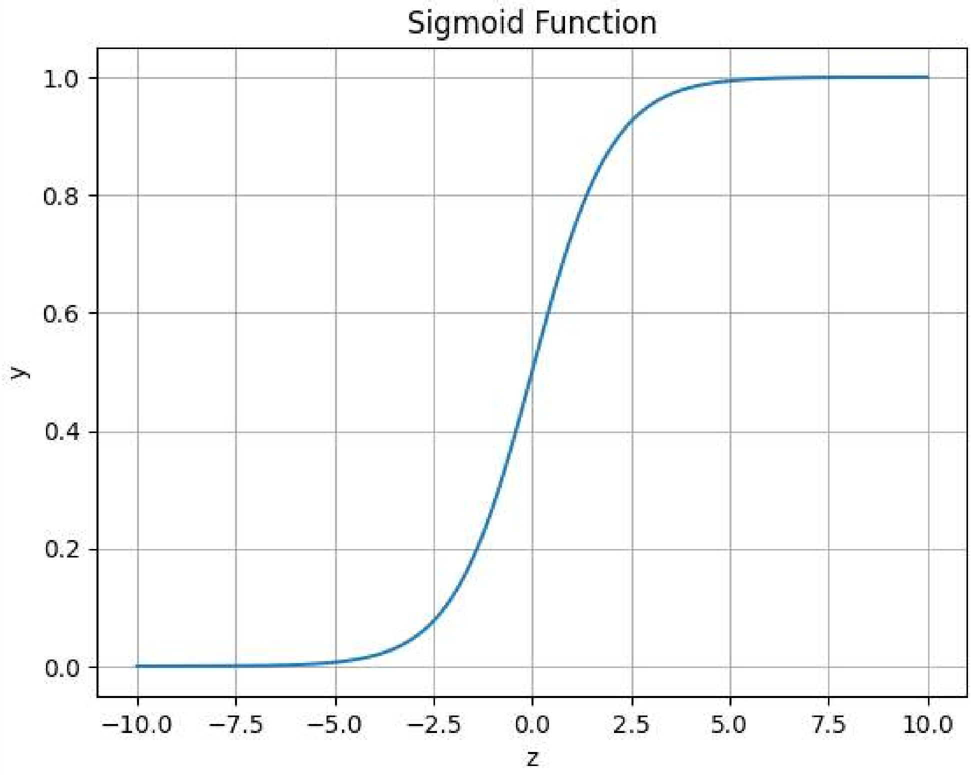
직관적으로, 수학적으로 모르겠음

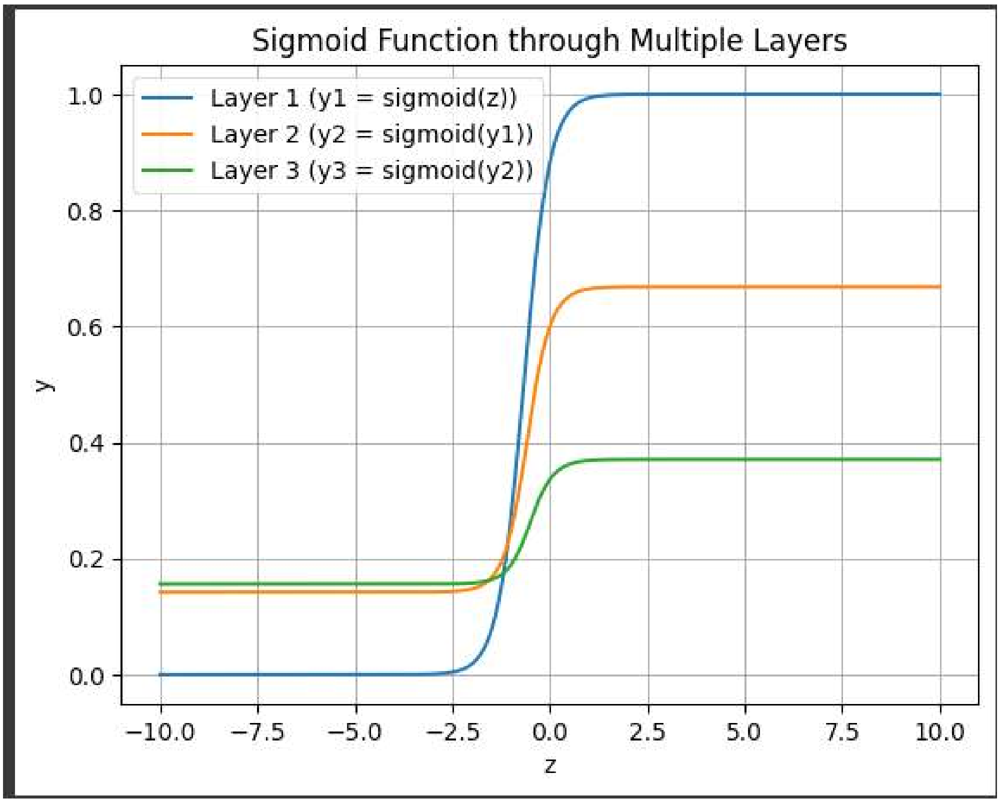
그래서 그냥 그래프를 그렸다.

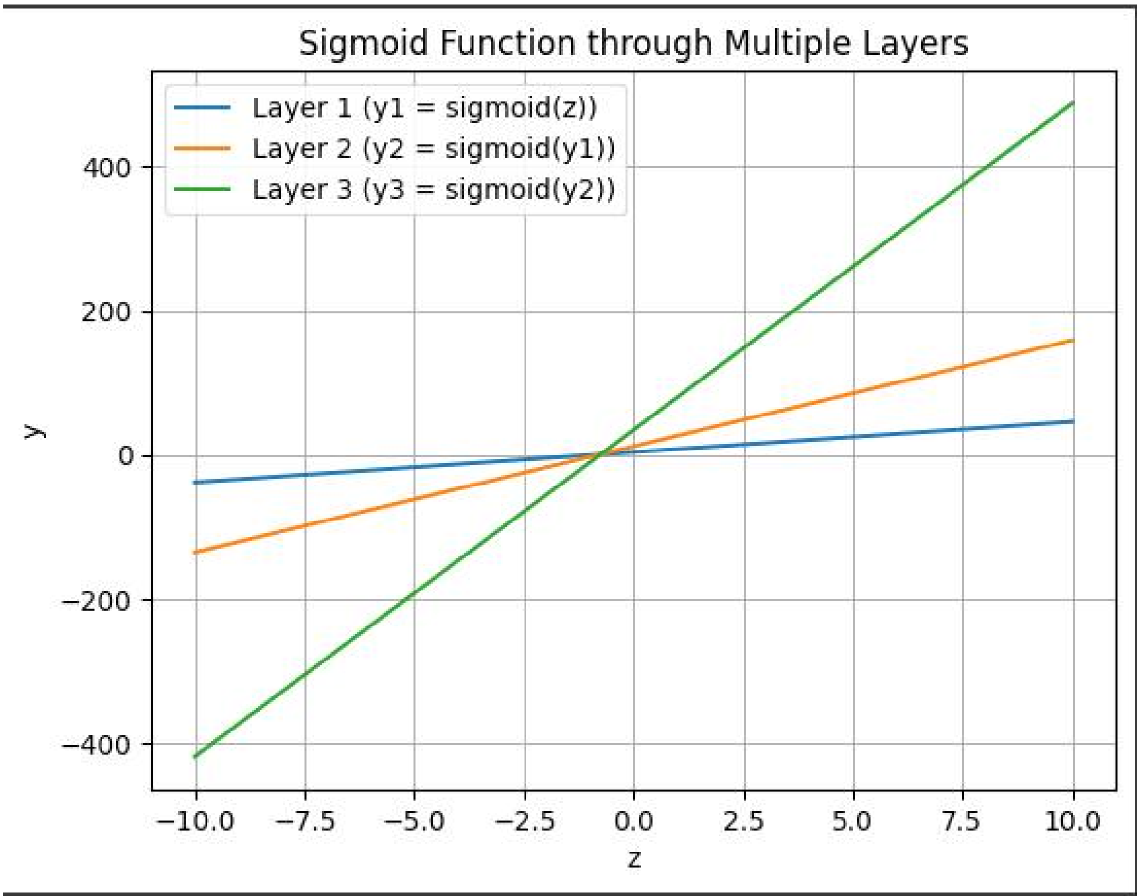
- 비선형 함수의 대표적인 시그모이드 함수를 사용함 -

```python
import numpy as np
import matplotlib.pyplot as plt

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

# z 값의 범위를 설정합니다.
z = np.linspace(-10, 10, 1000)

a = 3*z+2

# 첫 번째 레이어 출력
y1 = sigmoid(a)

s = 2.5*y1-1.8

# 두 번째 레이어 출력
y2 = sigmoid(s)

d = 2.2*y2-2

# 세 번째 레이어 출력
y3 = sigmoid(d)

# 그래프를 그립니다.
plt.plot(z, y1, label='Layer 1 (y1 = sigmoid(z))')
plt.plot(z, y2, label='Layer 2 (y2 = sigmoid(y1))')
plt.plot(z, y3, label='Layer 3 (y3 = sigmoid(y2))')
plt.title('Sigmoid Function through Multiple Layers')
plt.xlabel('z')
plt.ylabel('y')
plt.grid(True)
plt.legend()
plt.show()
```

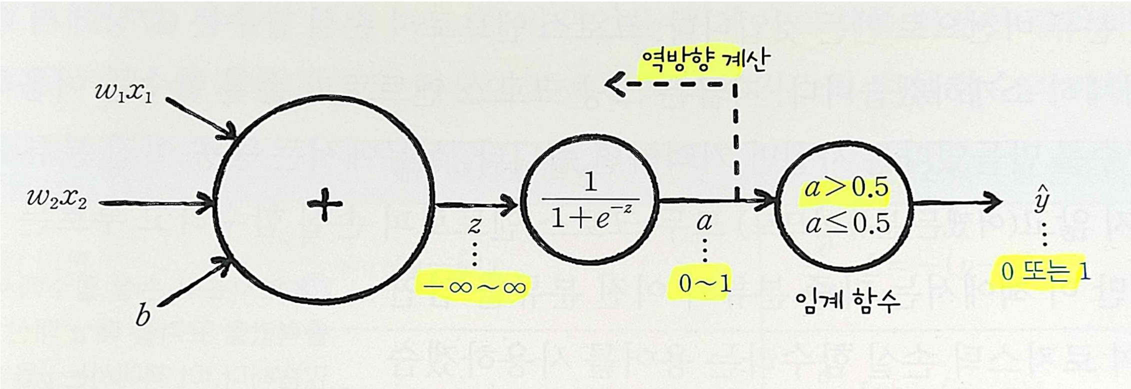
```python
import numpy as np
import matplotlib.pyplot as plt

# 시그모이드 함수 정의
def linefunction(x):
    return 1.4 * x+1

# z 값의 범위를 설정합니다.
z = np.linspace(-10, 10, 1000)

a = 3*z+2

# 첫 번째 레이어 출력
y1 = linefunction(a)

s = 2.5*y1-1.8

# 두 번째 레이어 출력
y2 = linefunction(s)

d = 2.2*y2-2

# 세 번째 레이어 출력
y3 = linefunction(d)

# 그래프를 그립니다.
plt.plot(z, y1, label='Layer 1 (y1 = sigmoid(z))')
plt.plot(z, y2, label='Layer 2 (y2 = sigmoid(y1))')
plt.plot(z, y3, label='Layer 3 (y3 = sigmoid(y2))')
plt.title('Sigmoid Function through Multiple Layers')
plt.xlabel('z')
plt.ylabel('y')
plt.grid(True)
plt.legend()
plt.show()
```


Sigmoid Function
1.0
0.8
0.6
0.4
0.2
0.0
y
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
z


Sigmoid Function through Multiple Layers
Layer 1 (y1 = sigmoid(z))
Layer 2 (y2 = sigmoid(y1))
Layer 3 (y3 = sigmoid(y2))
1.0
0.8
0.6
0.4
0.2
0.0
y
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
z


Sigmoid Function through Multiple Layers
Layer 1 (y1 = sigmoid(z))
Layer 2 (y2 = sigmoid(y1))
Layer 3 (y3 = sigmoid(y2))
400
200
0
−200
−400
y
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
z

로지스틱 회귀 중간정리



로지스틱 회귀는 이진분류가 목표이므로,

우선 -∞ ~ ∞의 범위를 가지는 z값을 조절해해서 활성화 함수(시그모이드)를 사용한다.

=> 확률처럼 해석하기 위해서이다.

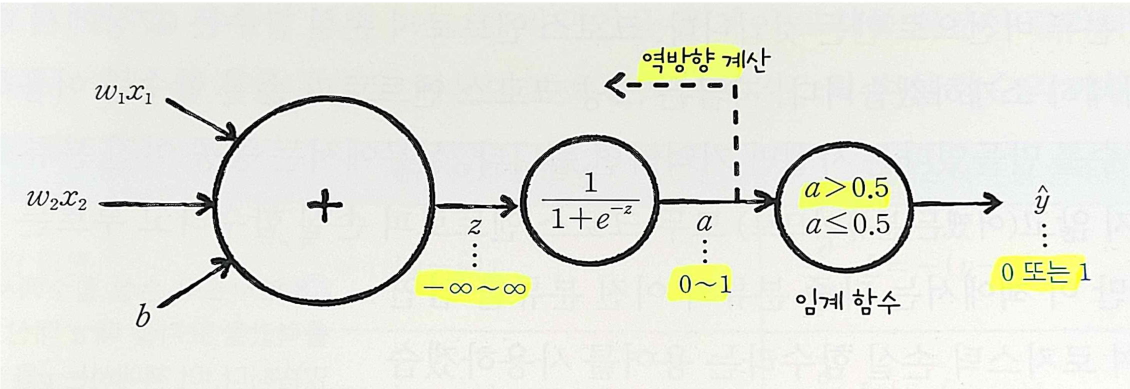
이후 임계함수를 통해 1 or 0 으로 나눈다.

=> 이제 가중치와 절편을 적절히 업데이트 해야한다.

로지스틱 손실함수

$$L = -(ylog(a) + (1-y)\log(1-a))$$

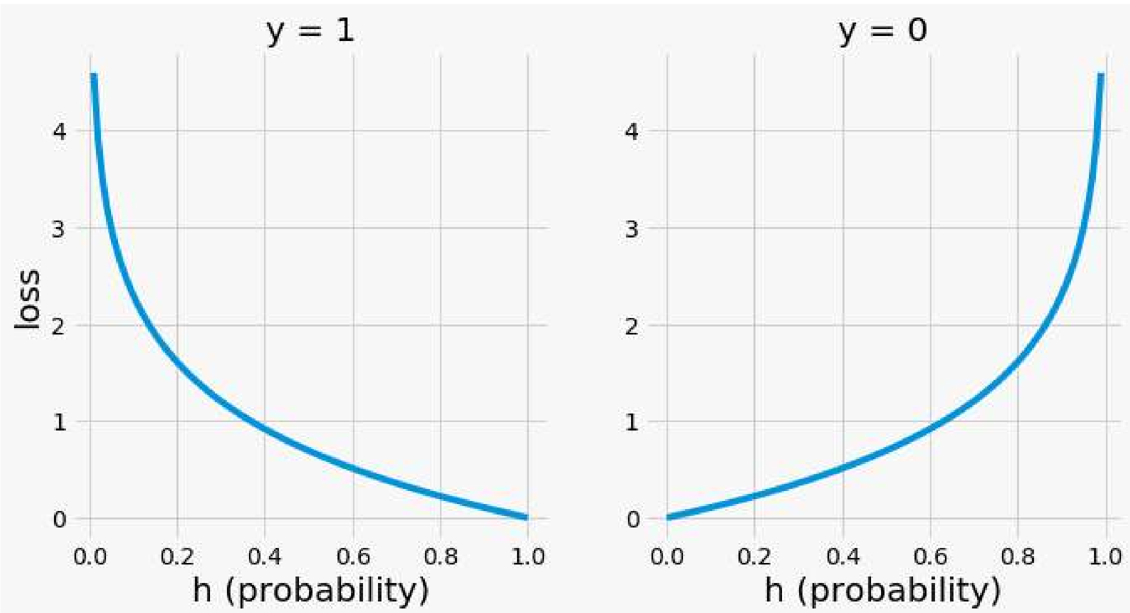
- y는 타깃, 즉 실제 정답값이다. (이진분류 이기 때문에 0 or 1)
- a는 활성화 함수가 출력한 값이다.



그렇기 때문에 L은,

- y가 1인 경우(양성 클래스) : -log(a)

- y가 0인 경우(음성 클래스) : -log(1-a)



=> 즉, 예측이 잘못되면 손실이 점진적으로 증가하고, 올바른 예측을 하면 손실이 작아지는 모델로써 경사하강법과 마찬가지로 최소화 하는 계수를 찾을 거다.

$$L = -(ylog(a) + (1-y)\log(1-a))$$

$$\frac{\partial}{\partial w_i} L = -(y-a)x_i$$

$$\frac{\partial}{\partial b} L = -(y-a)1$$

	제곱 오차의 미분	로지스틱 손실 함수의 미분
가중치에 대한 미분	$\frac{\partial SE}{\partial w} = -(y-\hat{y})x$	$\frac{\partial}{\partial w_i} L = -(y-a)x_i$
절편에 대한 미분	$\frac{\partial SE}{\partial b} = -(y-\hat{y})1$	$\frac{\partial}{\partial b} L = -(y-a)1$

=> 제곱오차와 별반 다르지 않다는걸 알 수 있다. 단지 y_hat에서 비선형함수인 활성화 함수를 통과한 값을 사용한다는 점이다.

가중치(w)와 절편(b)를 업데이트 하는법

선형회귀와 마찬가지로 가중치에서 손실함수를 가중치에 편미분한 값을 빼면된다

$$w_i = w_i - \frac{\partial L}{\partial w_i} = w_i + (y-a)x_i$$

$$b = b - \frac{\partial L}{\partial b} = b + (y-a)1$$

예시

```python
class LogisticNeuron:

    def __init__(self):
        self.w = None
        self.b = None

    def forpass(self, x):
        z = np.sum(x * self.w) + self.b  # 직선 방정식을 계산합니다
        return z

    def backprop(self, x, err):
        w_grad = x * err    # 가중치에 대한 그래디언트를 계산합니다
        b_grad = 1 * err    # 절편에 대한 그래디언트를 계산합니다
        return w_grad, b_grad

    def activation(self, z):
        z = np.clip(z, -100, None) # 안전한 np.exp() 계산을 위해
        a = 1 / (1 + np.exp(-z))  # 시그모이드 계산
        return a

    def fit(self, x, y, epochs=100):
        self.w = np.ones(x.shape[1])      # 가중치를 초기화합니다.
        self.b = 0                        # 절편을 초기화합니다.
        for i in range(epochs):           # epochs만큼 반복합니다
            for x_i, y_i in zip(x, y):    # 모든 샘플에 대해 반복합니다
                z = self.forpass(x_i)     # 정방향 계산
                a = self.activation(z)    # 활성화 함수 적용
                err = -(y_i - a)          # 오차 계산
                w_grad, b_grad = self.backprop(x_i, err) # 역방향 계산
                self.w -= w_grad          # 가중치 업데이트
                self.b -= b_grad          # 절편 업데이트

    def predict(self, x):
        z = [self.forpass(x_i) for x_i in x]     # 정방향 계산
        a = self.activation(np.array(z))         # 활성화 함수 적용
        return a > 0.5
```

```python
neuron = LogisticNeuron()
neuron.fit(x_train, y_train)
```

```python
np.mean(neuron.predict(x_test) == y_test)
```

```
0.8245614035087719
```

##매 에포크마다 훈련세트의 순서를 섞어서 손실함수를 더 줄이자

ex)

첫 번째 에포크)

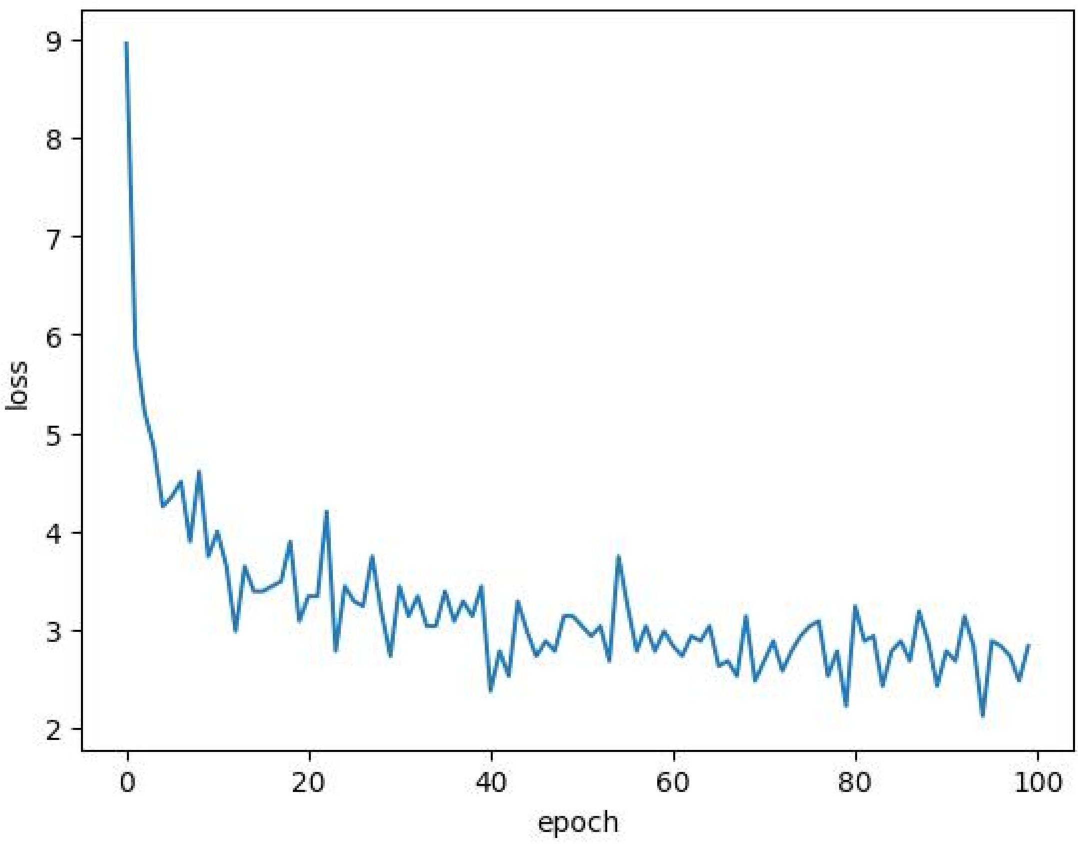
1st. test_x -> 2nd. test_x -> 3rd. test_x . . . .

두 번째 에포크)

4th. test_x -> 1st. test_x -> 3rd. test_x . . . .

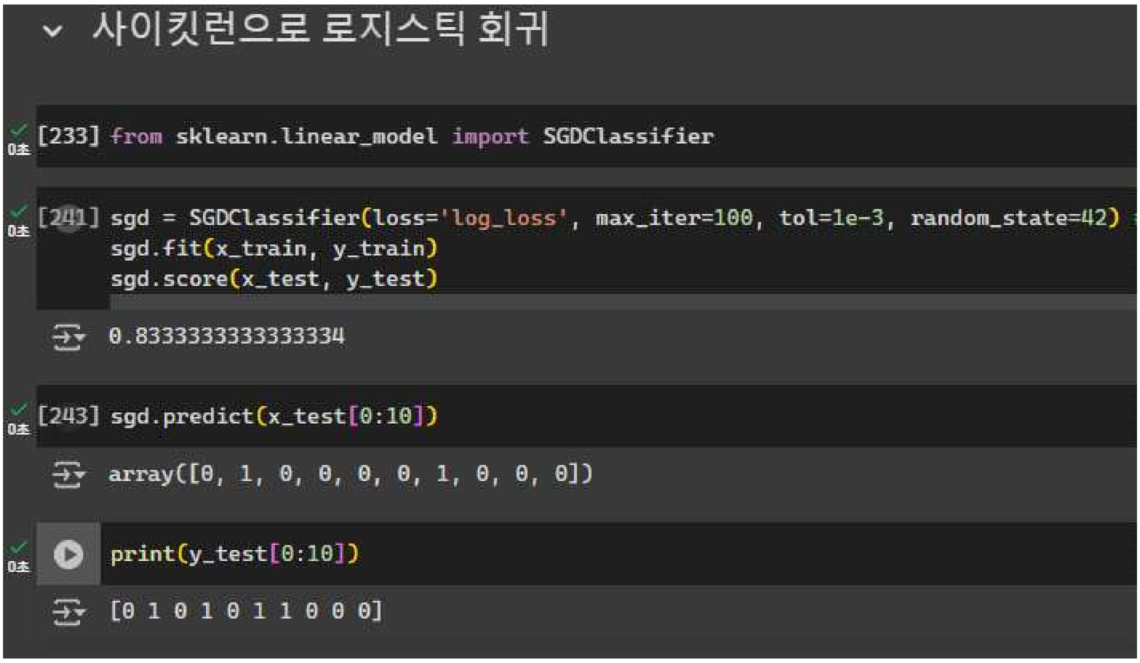
이렇게 랜덤하게 각각의 세트를 설정해서 진행하면, 동일한 것을 계속해서 반복하는것보다 변화를 줄수 있다.

```
layer = SingleLayer()
layer.fit(x_train, y_train)
layer.score(x_test, y_test)

0.9385964912280702
```



=> 에포크를 진행할때마다 손실함수는 크게봤을 때 낮아진다.

사이킷런으로 로지스틱 회귀




```
[233] from sklearn.linear_model import SGDClassifier

[241] sgd = SGDClassifier(loss='log_loss', max_iter=100, tol=1e-3, random_state=42)
      sgd.fit(x_train, y_train)
      sgd.score(x_test, y_test)

0.8333333333333334

[243] sgd.predict(x_test[0:10])

array([0, 1, 0, 0, 0, 0, 1, 0, 0, 0])

print(y_test[0:10])

[0 1 0 1 0 1 1 0 0 0]
```

=> log_loss는 로지스틱회귀, 100은 반복에포크, tol만큼 손실함수값이 감소하지 않으면 더이상 의미없다고 판단하고 중단