

# 04-2

## 확률적 경사 하강법

핵심 키워드

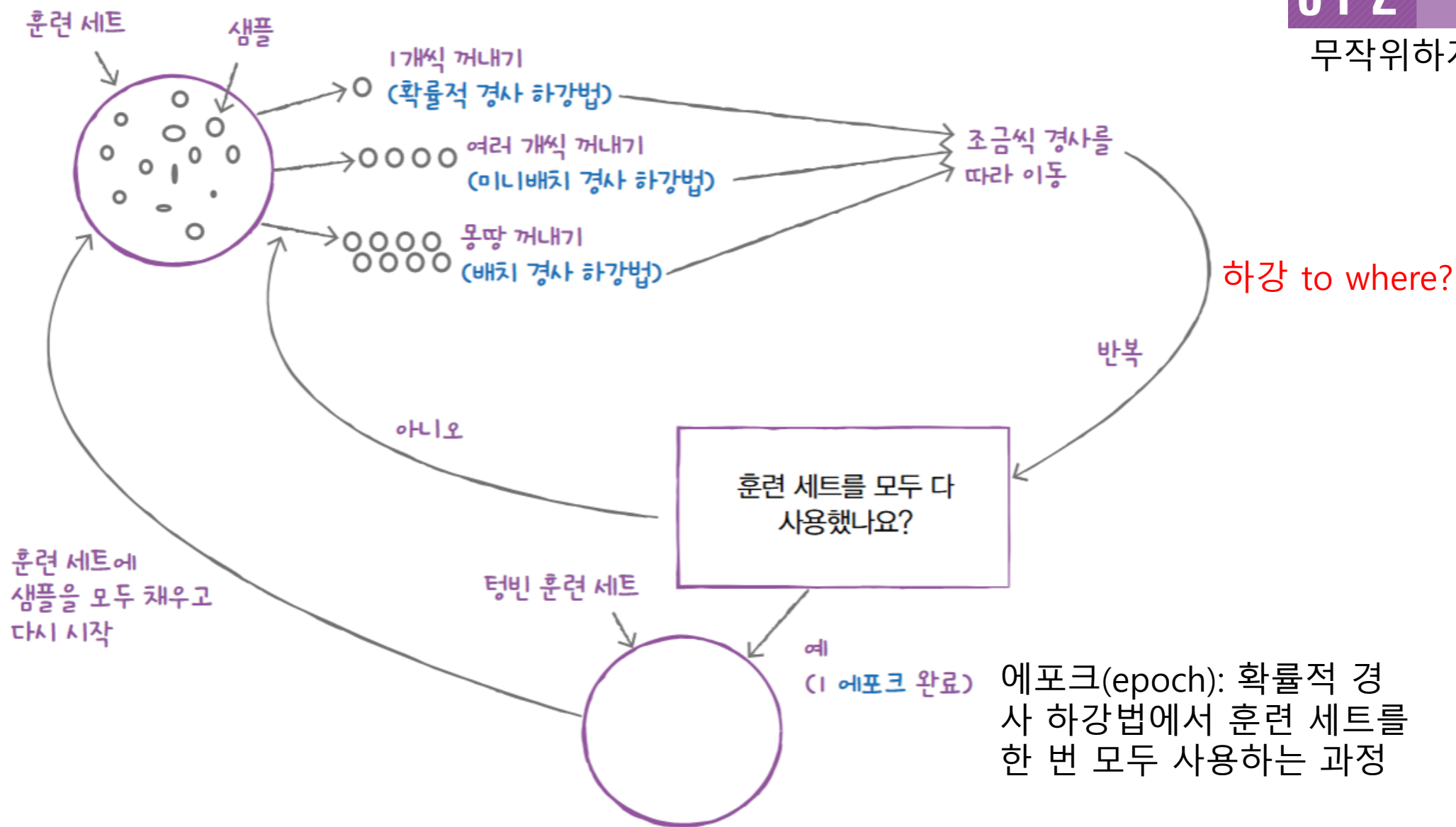
확률적 경사 하강법

손실 함수

에포크

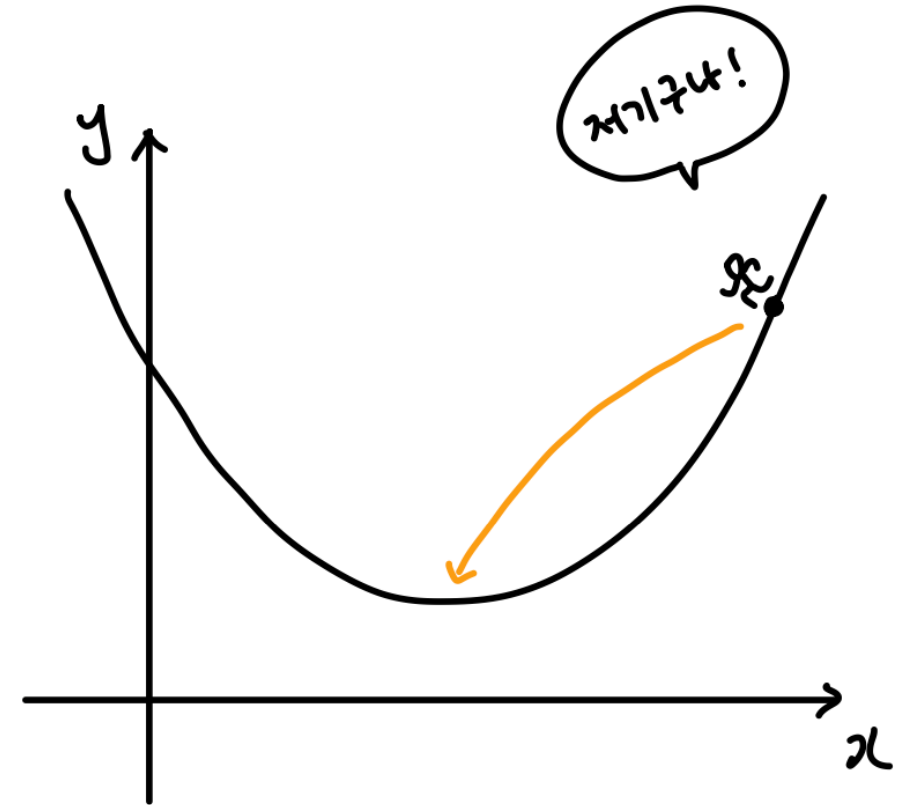
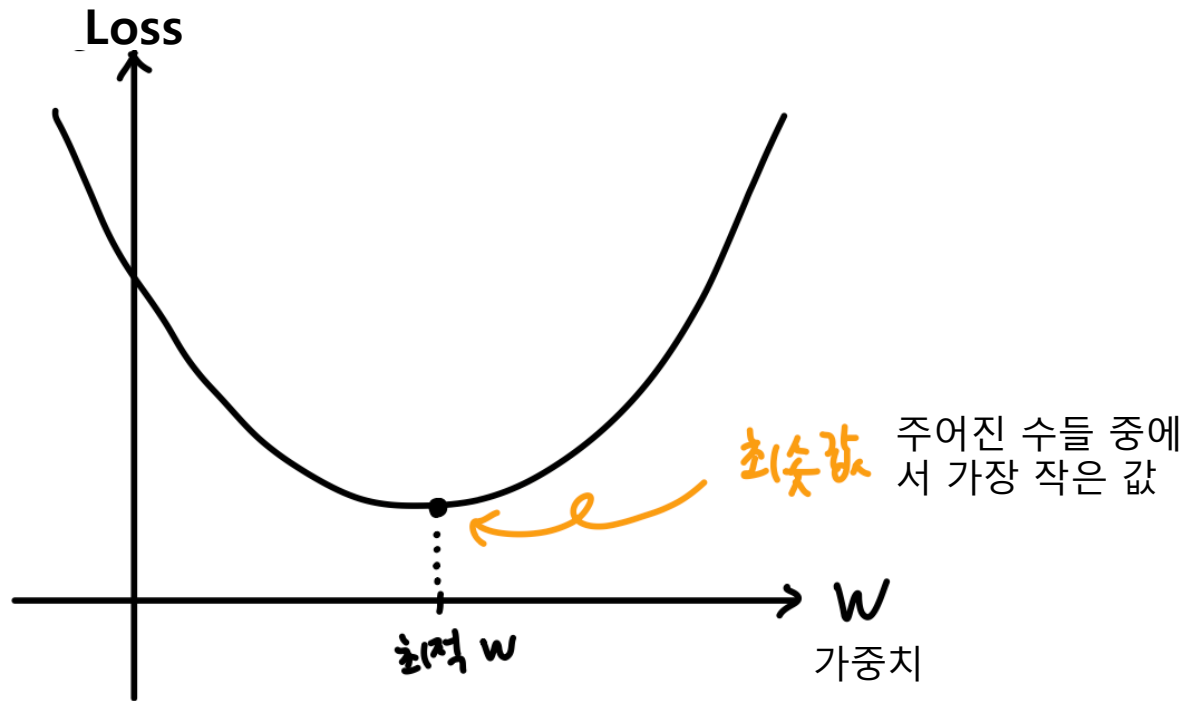
경사 하강법 알고리즘을 이해하고 대량의 데이터에서 분류 모델을 훈련하는 방법을 배웁니다.

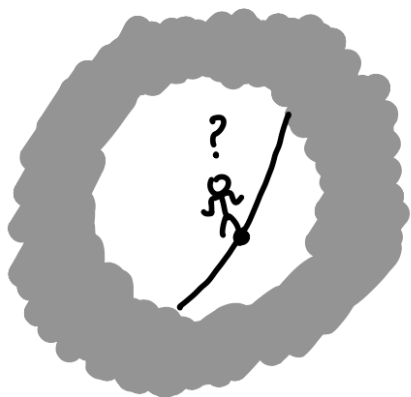
무작위하게 or 랜덤하게



**손실 함수 (loss function):** 머신러닝이나 딥러닝 모델이 예측한 값과 실제 값 사이의 차이를 측정하는 함수

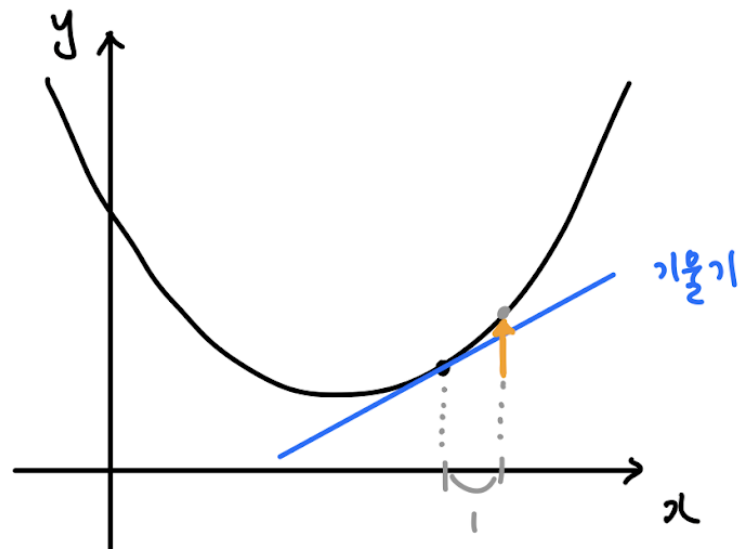
- 손실함수의 값을 최소화하기 위해 경사하강법(Gradient Descent)을 사용.
- 경사하강법은 손실함수의 기울기(gradient)를 계산하여, 기울기의 반대 방향으로 모델의 가중치를 업데이트하는 방식으로 동작.
- 이 때, 학습률(Learning Rate)은 각 업데이트 단계에서 가중치를 얼마나 조정할지 결정.
- Mean Squared Error, Binary Cross-Entropy, Categorical Cross-Entropy, Huber Loss 등





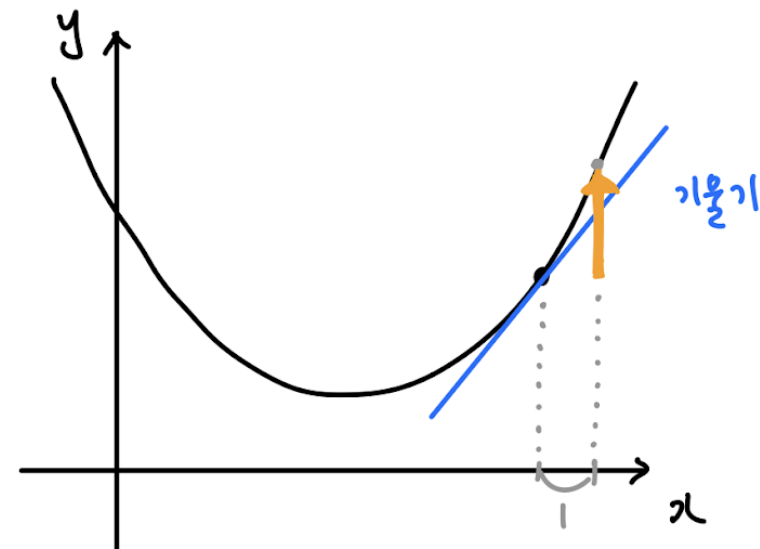
최솟값으로 가기 위해서는 어느 방향으로 몇 발자국 정도 내딛어볼 지 선택

기울기: 순간 변화량, 모든 변수의 편미분을 벡터로 정리한 것



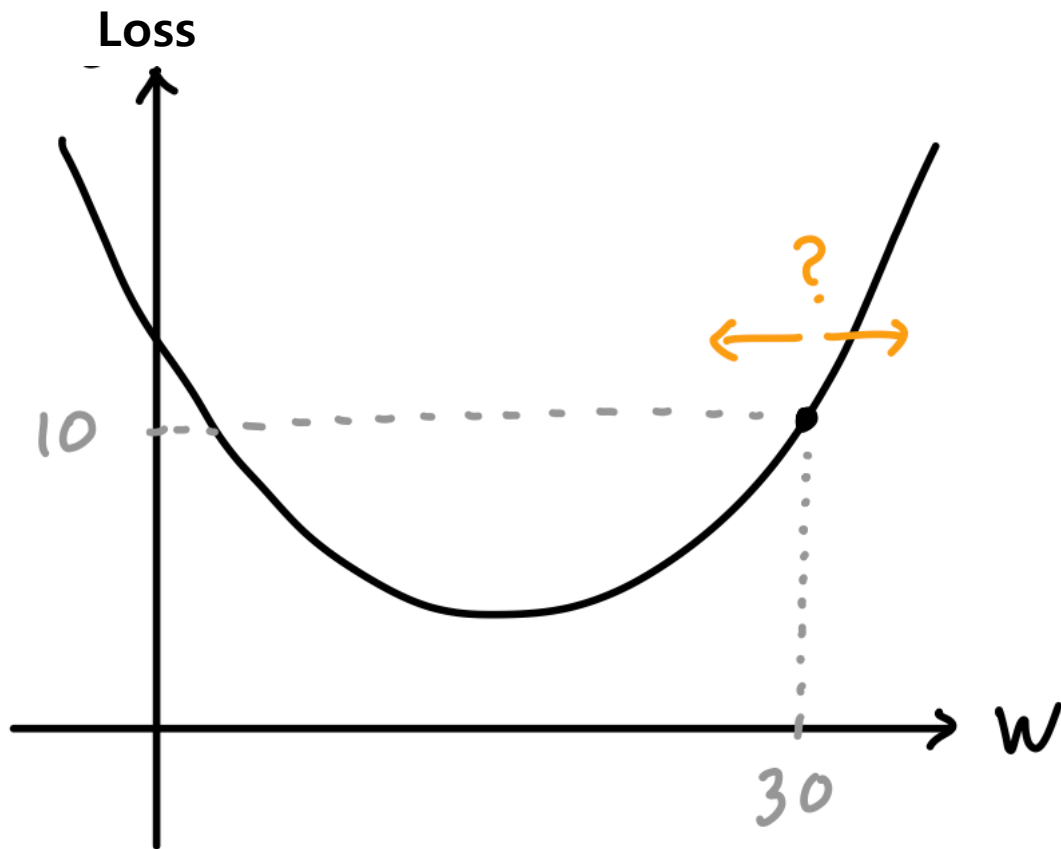
기울기가 작다

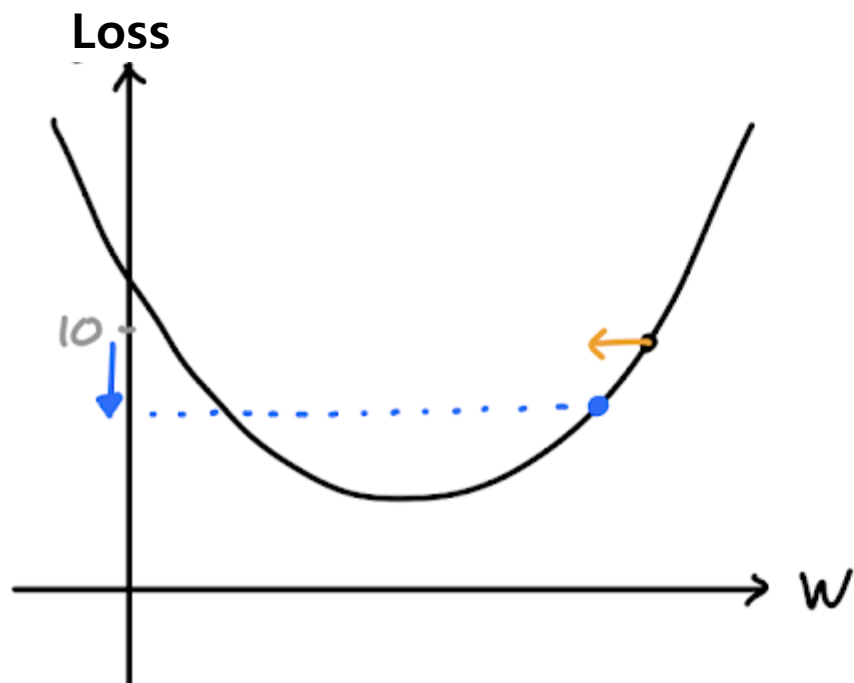
변화량이 적다



기울기가 크다

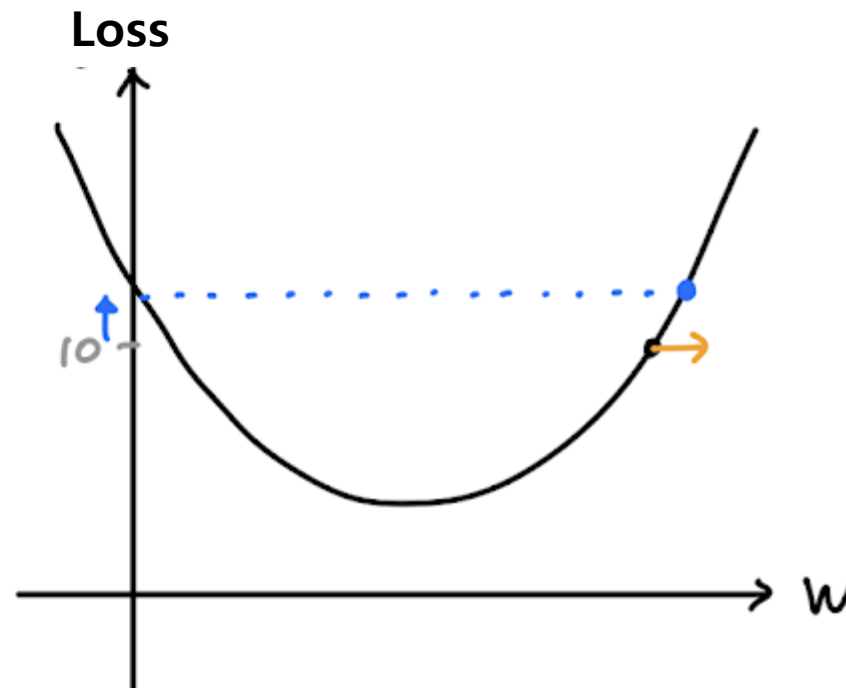
변화량이 많다





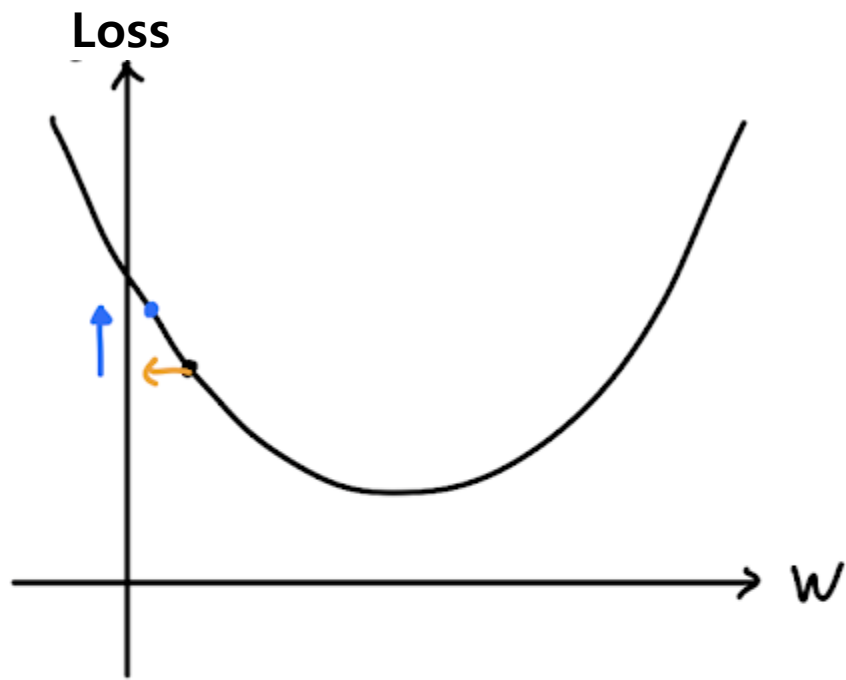
$W$ 를 왼쪽으로 이동

cost 감소 ✓

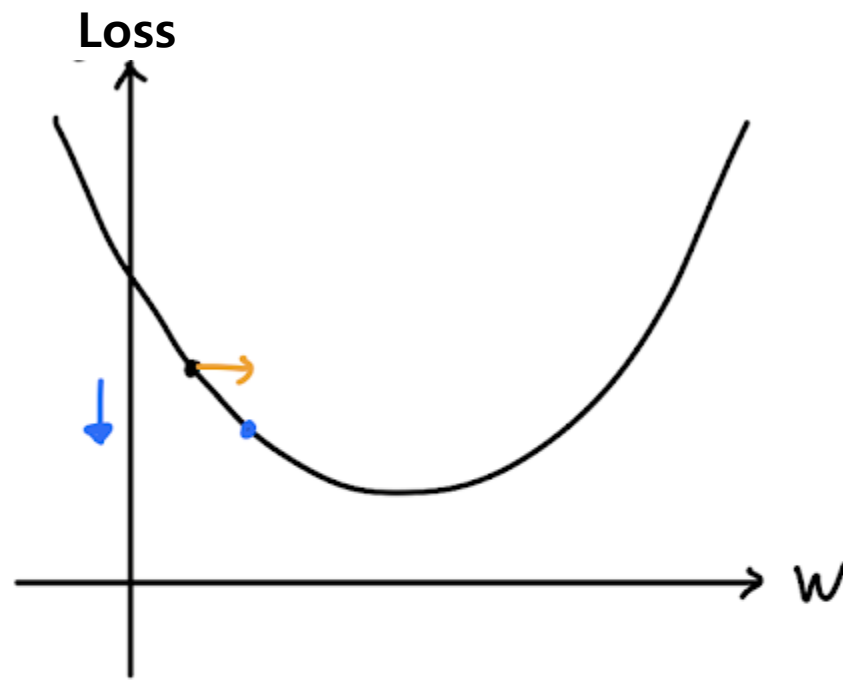


$W$ 를 오른쪽으로 이동

cost 증가

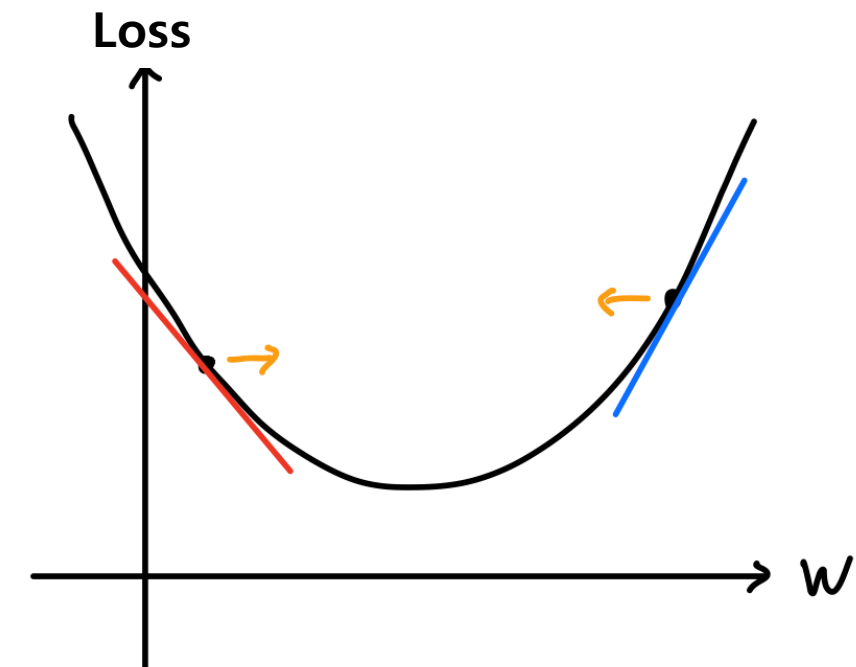


$W$ 를 왼쪽으로 이동  
cost 증가



$W$ 를 오른쪽으로 이동  
cost 감소 ✓





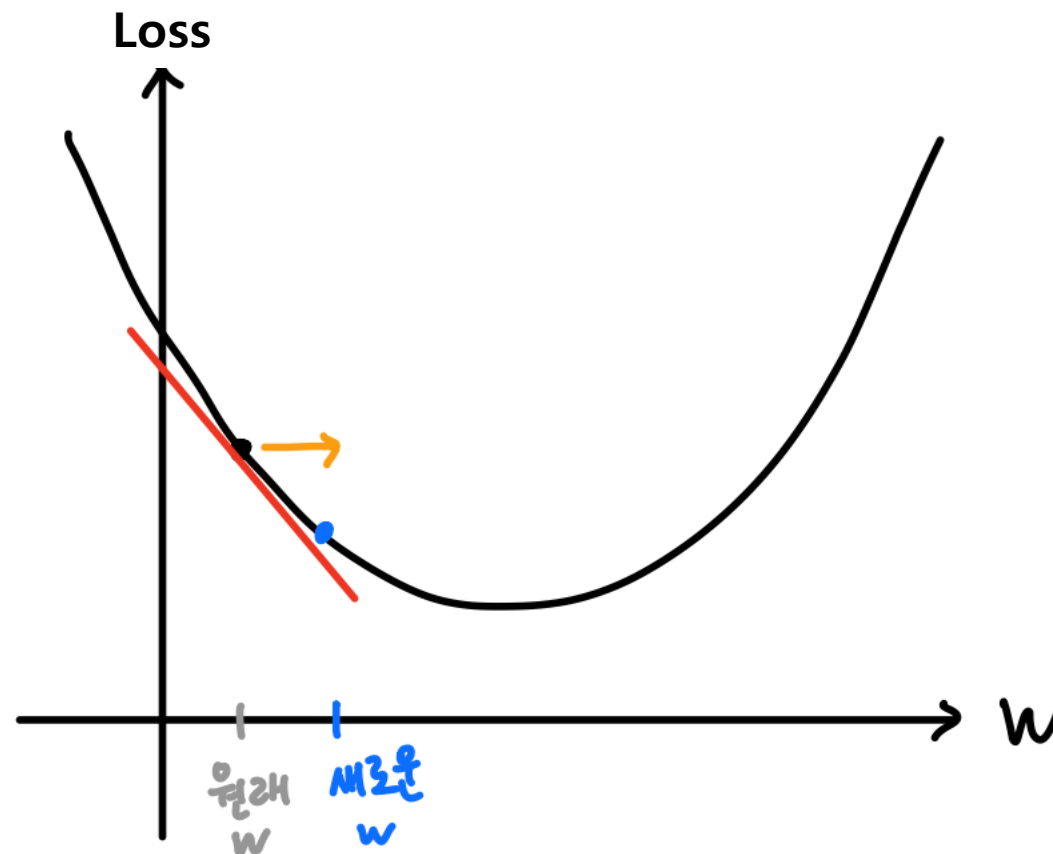
기울기가 음수

⇒ 오른쪽 (+)

기울기가 양수

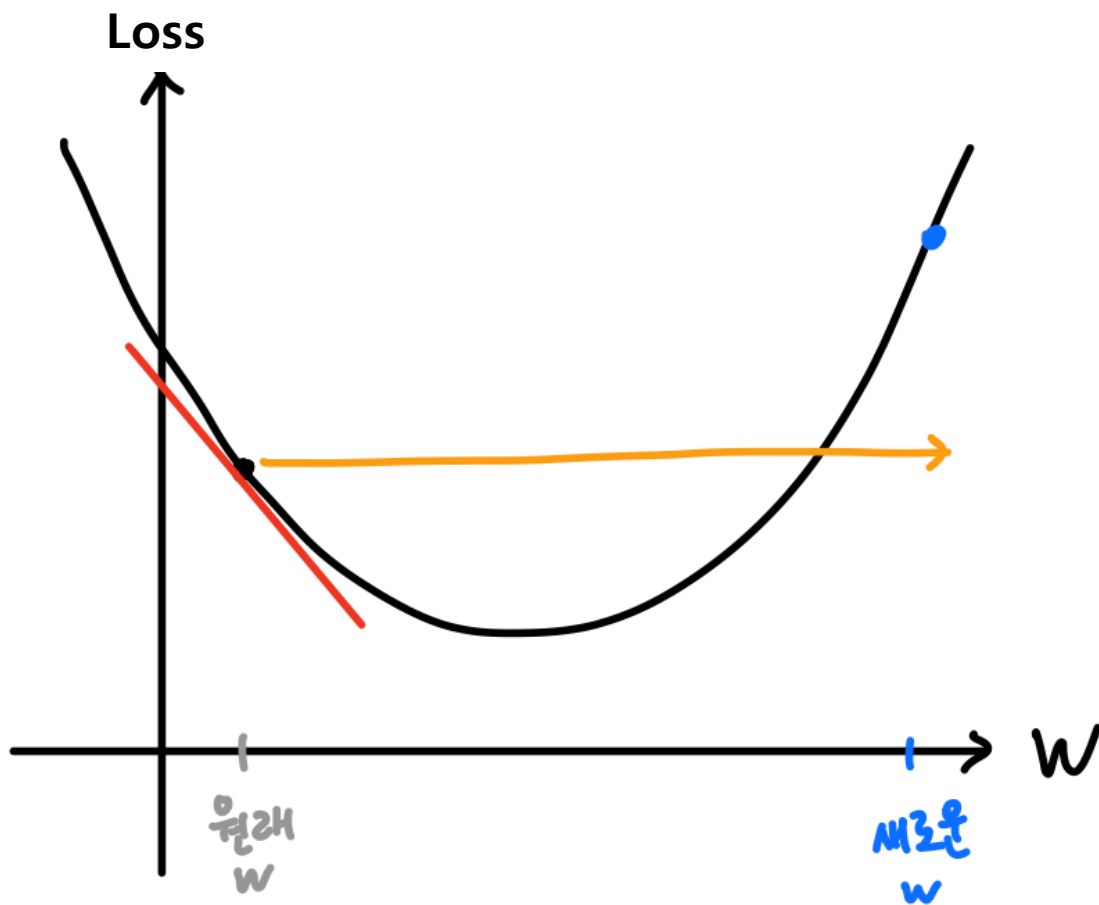
⇒ 왼쪽 (-)

기울기가 + (양수) 면 가중치는 - (왼쪽)  
기울기가 - (음수) 면 가중치는 + (오른쪽)



새로운  $W = \text{원래 } W + (\text{기울기의 반대 방향})$

새로운  $W = \text{원래 } W - \text{기울기}$



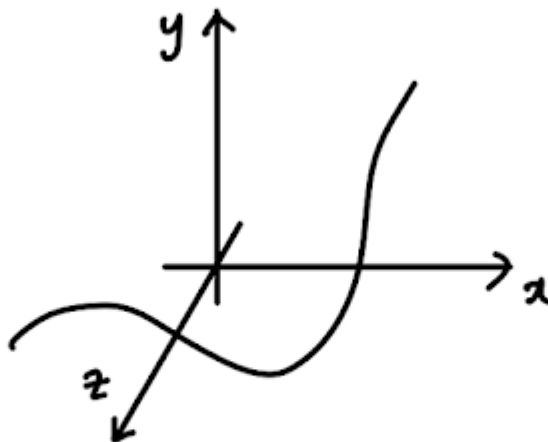
기울기가 너무 커지면 최솟값을 찾을 수 없음  
기울기에 작은 상수 (학습률)를 곱해서 사용함!

새로운  $W = \text{원래 } W - (\text{학습률}) \times \text{기울기}$

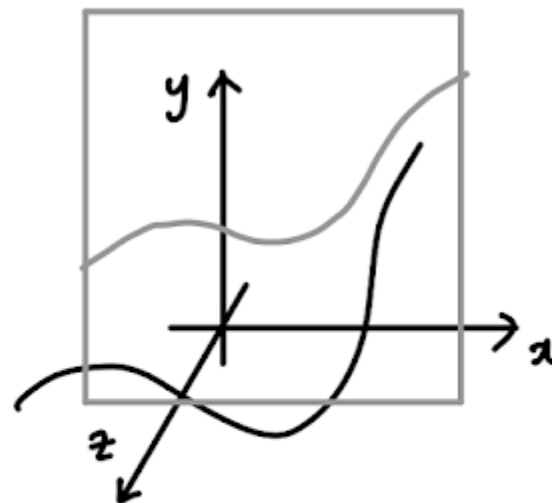
$$W := W - \underbrace{\eta}_{\text{학습률}} \underbrace{\frac{\partial C}{\partial W}}_{\text{기울기}}$$

편미분

$$\frac{\partial y}{\partial x}$$

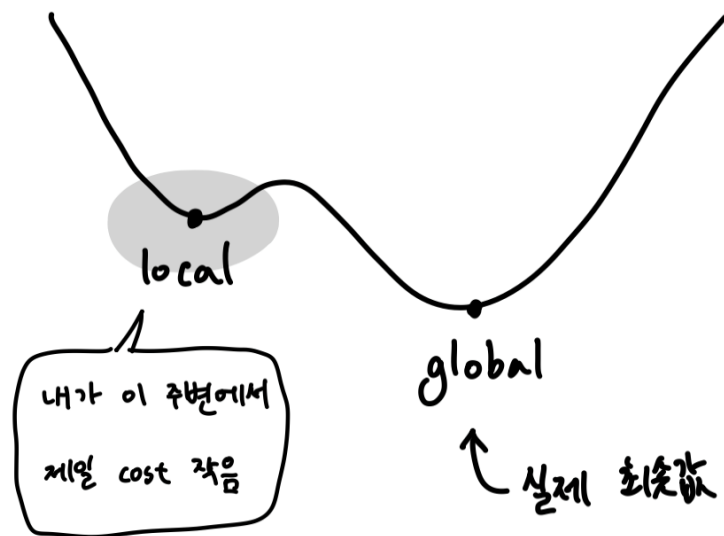


z 방향은 무시



x에 대한 y의 편미분 값은, z를 무시한 채 x, y 평면에서 기울기를 계산





Optimizer: 모멘텀, 아담

기울기만이 아니라 내려가는 가속도를 함께 반영하는 방법으로 그 문턱을 넘어갈 수 있음.

가속도라는 뜻인 모멘텀(momentum)이 그 방법. 더 다양한 상황에서 학습 실패가 적은 아담(adam) 등을 많이 사용.

## 손실함수 최적화 알고리즘

### 1. Gradient Descent (경사 하강법)

- 경사 하강법은 손실 함수의 기울기(gradient)를 이용하여 모델의 파라미터를 업데이트
- 기울기는 손실 함수를 각 파라미터로 편미분하여 계산
- 파라미터를 기울기의 반대 방향으로 일정 크기(학습률)만큼 업데이트

$$\theta_{t+1} = \theta_t - \eta \cdot \nabla_{\theta} J(\theta)$$

모델 파라미터      학습률      손실 함수의 기울기

## 손실함수 최적화 알고리즘

### 2. Stochastic Gradient Descent (SGD, 확률적 경사 하강법)

- SGD는 경사 하강법의 변형으로, 매 업데이트마다 전체 데이터 대신 일부 데이터(미니배치)를 사용
- SGD는 전체 데이터를 사용하는 것보다 계산 효율이 높고, 국소 최적점에서 벗어날 수 있는 장점

$$\theta_{t+1} = \theta_t - \eta \cdot \nabla_{\theta} J(\theta; x^{(i:i+n)}, y^{(i:i+n)})$$

i번째부터 i+n번째까지의  
입력 데이터와 레이블

## 손실함수 최적화 알고리즘

### 3. Momentum (모멘텀)

- SGD에 관성의 개념을 도입한 방법입니다.
- 이전 업데이트의 방향을 일정 부분 유지하여, 업데이트의 안정성과 수렴 속도를 높입니다.


$$v_t = \gamma v_{t-1} + \eta \cdot \nabla_{\theta} J(\theta), \theta_{t+1} = \theta_t - v_t$$

시간 t에서의  
업데이트 속도

모멘텀 계수


## SGDClassifier

이번에도 fish\_csv\_data 파일에서 판다스 데이터프레임을 만들어 보겠습니다.

손코딩

```
import pandas as pd  
fish = pd.read_csv('https://bit.ly/fish_csv_data')
```

그다음 Species 열을 제외한 나머지 5개는 입력 데이터로 사용합니다. Species 열은 타겟 데이터입니다.

손코딩

```
fish_input = fish[['Weight', 'Length', 'Diagonal', 'Height', 'Width']].to_numpy()  
fish_target = fish['Species'].to_numpy()
```



사이킷런의 `train_test_split()` 함수를 사용해 이 데이터를 훈련 세트와 테스트 세트로 나눕니다.

 손코딩

```
from sklearn.model_selection import train_test_split
train_input, test_input, train_target, test_target = train_test_split(
    fish_input, fish_target, random_state=42)
```

이제 훈련 세트와 테스트 세트의 특성을 표준화 전처리합니다. 다시 한번 강조하지만 꼭 훈련 세트에서 학습한 통계 값으로 테스트 세트도 변환해야 합니다.

 손코딩

```
from sklearn.preprocessing import StandardScaler
ss = StandardScaler()
ss.fit(train_input)
train_scaled = ss.transform(train_input)
test_scaled = ss.transform(test_input)
```

네, 좋습니다. 특성값의 스케일을 맞추는 `train_scaled`와 `test_scaled` 두 넘파이 배열을 준비했습니다. 여기까지는 이전과 동일합니다. 사이킷런에서 확률적 경사 하강법을 제공하는 대표적인 분류용 클래스는 `SGDClassifier`입니다. `sklearn.linear_model` 패키지 아래에서 임포트해 보죠.

 손코딩

```
from sklearn.linear_model import SGDClassifier
```

`SGDClassifier`의 객체를 만들 때 2개의 매개변수를 지정합니다. `loss`는 손실 함수의 종류를 지정합니다. 여기에서는 `loss='log_loss'`로 지정하여 로지스틱 손실 함수를 지정했습니다. `max_iter`는 수행할 에포크 횟수를 지정합니다. 10으로 지정하여 전체 훈련 세트를 10회 반복하겠습니다. 그다음 훈련 세트와 테스트 세트에서 정확도 점수를 출력합니다.

**note** 다중 분류일 경우 `SGDClassifier`에 `loss='log_loss'`로 지정하면 클래스마다 이진 분류 모델을 만듭니다. 즉 도미는 양성 클래스로 두고 나머지를 모두 음성 클래스로 두는 방식입니다. 이런 방식을 OvR(One versus Rest)이라고 부릅니다.

손코딩

```
sc = SGDClassifier(loss='log_loss', max_iter=10, random_state=42)
sc.fit(train_scaled, train_target)
print(sc.score(train_scaled, train_target))
print(sc.score(test_scaled, test_target))
```



0.773109243697479

0.775

반복 횟수가 부족한 듯?

이 메서드는 `fit()` 메서드와 사용법이 같지만 호출할 때마다 1 에포크씩 이어서 훈련할 수 있습니다.

`partial_fit()` 메서드를 호출하고 다시 훈련 세트와 테스트 세트의 점수를 확인해 보겠습니다.

`partial_fit()` : 호출때마다 1 에포크씩 이어서 훈련

손코딩

```
sc.partial_fit(train_scaled, train_target)
print(sc.score(train_scaled, train_target))
print(sc.score(test_scaled, test_target))
```



0.8151260504201681

0.825

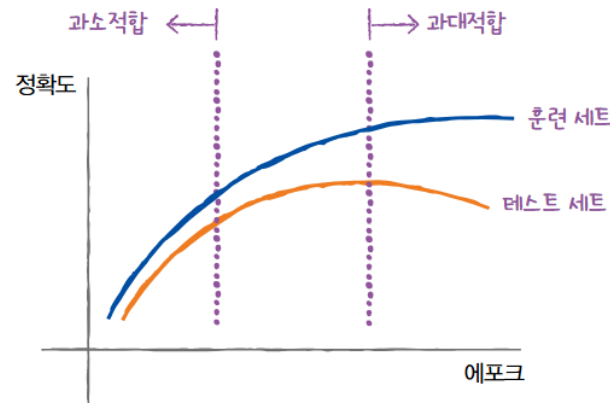
아직 점수가 낮지만 에포크를 한 번 더 실행하니 정확도가 향상되었습니다. 이 모델을 여러 에포크에서 더 훈련해 볼 필요가 있겠군요. 그런데 얼마나 더 훈련해야 할까요? 무작정 많이 반복할 수는 없고 어떤 기준이 필요하겠군요.

## 에포크와 과대/과소적합

3장에서 배웠던 과소적합과 과대적합을 기억하시나요? 확률적 경사 하강법을 사용한 모델은 에포크 횟수에 따라 과소적합이나 과대적합이 될 수 있습니다. 왜 이런 현상이 일어나는지 잠시 생각해 보죠.

에포크 횟수가 적으면 모델이 훈련 세트를 덜 학습합니다. 마치 산을 다 내려오지 못 하고 훈련을 마치는 셈이죠. 에포크 횟수가 충분히 많으면 훈련 세트를 완전히 학습할 것입니다. 훈련 세트에 아주 잘 맞는 모델이 만들어집니다.

바꾸어 말하면 적은 에포크 횟수 동안에 훈련한 모델은 훈련 세트와 테스트 세트에 잘 맞지 않는 과소적합된 모델일 가능성이 높습니다. 반대로 많은 에포크 횟수 동안에 훈련한 모델은 훈련 세트에 너무 잘 맞아 테스트 세트에는 오히려 점수가 나쁜 과대적합된 모델일 가능성이 높습니다.



조기 종료 (early stopping):  
과대적합이 시작하기 전에 훈련을 멈춤

손코딩

```
import numpy as np
sc = SGDClassifier(loss='log_loss', random_state=42)
train_score = []
test_score = []
classes = np.unique(train_target)
```

300번의 에포크 동안 훈련을 반복하여 진행해 보겠습니다. 반복마다 훈련 세트와 테스트 세트의 점수를 계산하여 train\_score, test\_score 리스트에 추가합니다.

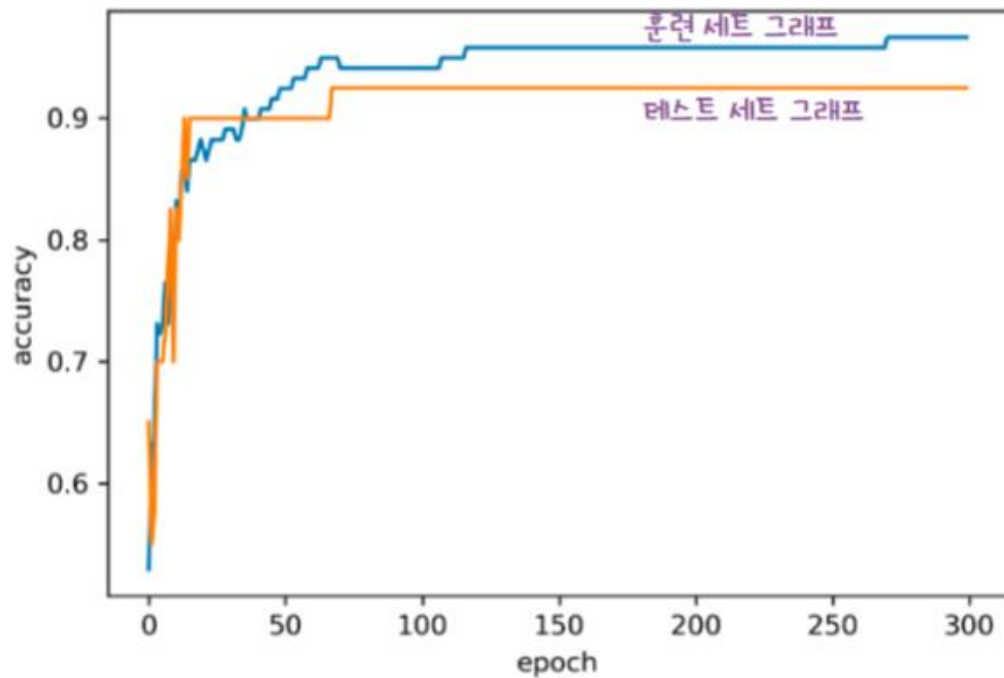
손코딩

```
for _ in range(0, 300):
    sc.partial_fit(train_scaled, train_target, classes=classes)
    train_score.append(sc.score(train_scaled, train_target))
    test_score.append(sc.score(test_scaled, test_target))
```

**note** 파이썬의 `_`는 특별한 변수입니다. 나중에 사용하지 않고 그냥 버리는 값을 넣어두는 용도로 사용하죠. 여기서는 0에서 299까지 반복 횟수를 임시 저장하기 위한 용도로 사용했습니다.

손코딩

```
import matplotlib.pyplot as plt  
plt.plot(train_score)  
plt.plot(test_score)  
plt.xlabel('epoch')  
plt.ylabel('accuracy')  
plt.show()
```



~100 epoch에서 달라지기 시작하는 듯?

그럼 SGDClassifier의 반복 횟수를 100에 맞추고 모델을 다시 훈련해 보겠습니다. 그리고 최종적으로 훈련 세트와 테스트 세트에서 점수를 출력합니다.

 손코딩

```
sc = SGDClassifier(loss='log_loss', max_iter=100, tol=None, random_state=42)
sc.fit(train_scaled, train_target)
```

```
print(sc.score(train_scaled, train_target))
print(sc.score(test_scaled, test_target))
```

```
0.957983193277311
0.925
```

SGDClassifier는 일정 에포크 동안 성능이 향상되지 않으면 더 훈련하지 않고 자동으로 멈춥니다. tol 매개변수에서 향상될 최소값을 지정합니다. 앞의 코드에서는 tol 매개변수를 None으로 지정하여 자동으로 멈추지 않고 max\_iter=100 만큼 무조건 반복하도록 하였습니다.



이 섹션을 마무리하기 전에 SGDClassifier의 loss 매개변수를 잠시 알아보겠습니다. 사실 loss 매개변수의 기본값은 'hinge'입니다. **힌지 손실**<sup>hinge loss</sup>은 **서포트 벡터 머신**<sup>support vector machine</sup>이라 불리는 또 다른 머신러닝 알고리즘을 위한 손실 함수입니다. 여기에서는 힙지 손실과 서포트 벡터 머신에 대해 더 자세히 다루지 않습니다. 하지만 서포트 벡터 머신이 널리 사용하는 머신러닝 알고리즘 중 하나라는 점과 SGDClassifier가 여러 종류의 손실 함수를 loss 매개변수에 지정하여 다양한 머신러닝 알고리즘을 지원한다는 것만 기억해 주세요.

간단한 예로 힙지 손실을 사용해 같은 반복 횟수 동안 모델을 훈련해 보겠습니다.

손코딩

```
sc = SGDClassifier(loss='hinge', max_iter=100, tol=None, random_state=42)
sc.fit(train_scaled, train_target)
print(sc.score(train_scaled, train_target))
print(sc.score(test_scaled, test_target))
```

```
0.9495798319327731
0.925
```