

4. 선형회귀분석(Linner Regression)

1. 머신러닝 수식
2. 선형회귀 이해
3. 선형회귀 실습(python)
4. 다중선형회귀 실습(python)
5. Pytorch 선형회귀
6. nn.Model로 선형 회귀 구현
7. 클래스로 선형회귀 모델 구현하기
8. Tensorflow 활용 선형회귀 분석

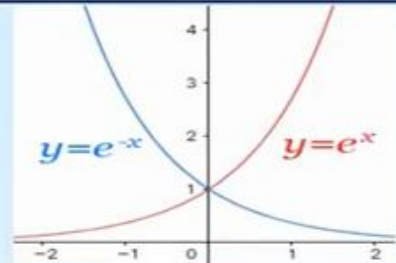
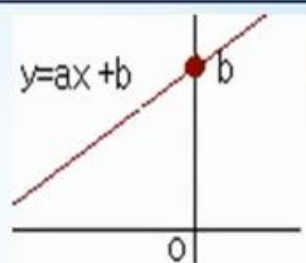
1. 머신러닝 수식

❖ 기본 수식

➤ 기본 수학 개념

$$y = ax + b$$

$$y = e^x$$



$$\sum_{k=1}^2 A_k = A_1 + A_2$$

$$\prod_{k=1}^2 A_k = A_1 \times A_2$$

➤ 행렬 (matrix) 연산

- 산술연산 (+, -, *, /)

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} + \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 2 & 2 \\ 3 & 5 \end{pmatrix}$$

$$(2 \times 2) + (2 \times 2) = (2 \times 2)$$

- 행렬 곱 (dot product)

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 0 \\ 3 & 4 & 0 \end{pmatrix}$$

$$(2 \times \boxed{2}) \cdot (\boxed{2}) \times 3 = (2 \times 3)$$

1. 머신러닝 수식

❖ 미분

미분 - derivative

미분을 왜 하는가? 미분으로 얻을 수 있는 인사이트?

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

$f(x)$ 를 미분하라

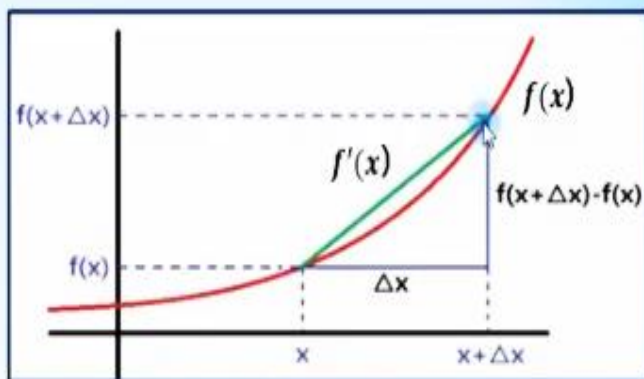
⇒ 입력변수 x 가 미세하게 변할때, 함수 f 가 얼마나 변하는지 알 수 있는 식을 구해라

⇒ 함수 $f(x)$ 는 입력 x 의 미세한 변화에 얼마나 민감하게 반응하는지 알 수 있는 식을 구해라

insight

⇒ 입력 x 를 현재 값에서 아주 조금 변화시키면, 함수 $f(x)$ 는 얼마나 변하는가?

⇒ 함수 $f(x)$ 는 입력 x 의 미세한 변화에 얼마나 민감하게 반응하는가?



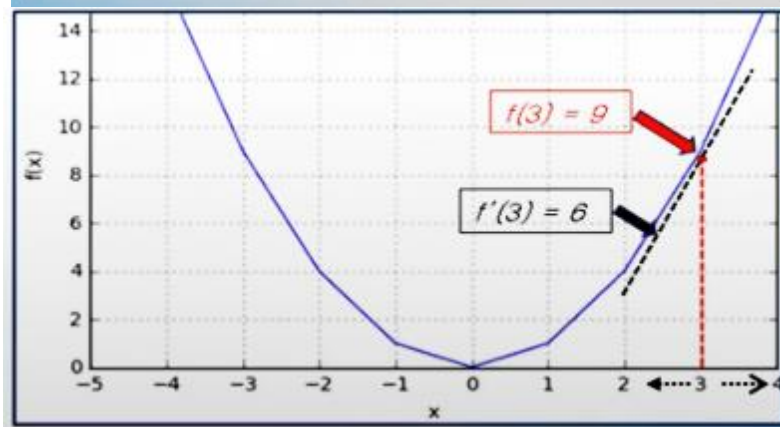
[예] 함수 $f(x) = x^2$ 일 경우, 미분 $f'(x) = 2x$

$f(3) = 9$ 해석

⇒ 입력 $x = 3$ 에서 출력은 9 임을 의미

$f'(3) = 6$ 해석

⇒ 입력 $x = 3$ 을 미세하게 변화시킬 때 함수는 현재 입력 값의 2배인 6 배 변화를 일으킴을 의미



1. 머신러닝 수식

❖ 머신러닝/딥러닝에 자주 사용되는 함수 미분

$$f(x) = \text{상수} \Rightarrow f'(x) = 0$$

$$f(x) = ax^n \Rightarrow f'(x) = nax^{n-1}$$

$$f(x) = e^x \Rightarrow f'(x) = e^x$$

$$f(x) = \ln x \Rightarrow f'(x) = \frac{1}{x}$$

$$f(x) = e^{-x} \Rightarrow f'(x) = -e^{-x}$$

$$[\text{예 1}] \quad f(x) = 3x^2 + e^x + 7 \Rightarrow f'(x) = 6x + e^x$$

$$[\text{예 2}] \quad f(x) = \ln x + \frac{1}{x} \Rightarrow f'(x) = \frac{1}{x} - \frac{1}{x^2}$$

참고

$$\frac{1}{x} = x^{-1}$$

1. 머신러닝 수식

❖ 편미분-partial derivative

- 입력 변수가 하나 이상인 다 변수 함수에서 미분하고자하는 변수 하나를 제외한 나머지 변수들을 상수로 취급하고 해당 변수를 미분하는 것
- 예를 들어 $f(x, y)$ 를 변수 x 에 대해 편미분 하는 경우 다음과 같이 나타냄

$$\frac{\partial f(x, y)}{\partial x}$$

[예1] $f(x, y) = 2x + 3xy + y^3$, 변수 x 에 대하여 편미분

$$\frac{\partial f(x, y)}{\partial x} = \frac{\partial(2x + 3xy + y^3)}{\partial x} = 2 + 3y$$

[예2] $f(x, y) = 2x + 3xy + y^3$, 변수 y 에 대하여 편미분

$$\frac{\partial f(x, y)}{\partial y} = \frac{\partial(2x + 3xy + y^3)}{\partial y} = 3x + 3y^2$$

[예3] 체중 함수가 '체중(야식, 운동)' 처럼 야식/운동에 영향을 받는 2변수 함수라고 가정할 경우, 편미분을 이용하면 각 변수 변화에 따른 체중 변화량을 구할 수 있음

현재 먹는 야식의 양에서
조금 변화를 줄 경우 체
중은 얼마나 변하는가 ?



$\frac{\partial \text{체중}}{\partial \text{야식}}$

현재 하고 있는 운동량에 조
금 변화를 줄 경우 체중은 얼
마나 변하는가 ?



$\frac{\partial \text{체중}}{\partial \text{운동}}$

1. 머신러닝 수식

❖ 연쇄법칙- chain rule

- 합성 함수란 여러 함수로 구성된 함수로서 합성 함수를 미분하려면 '합성함수를 구성하는 각 함수를 미분의 곱' 으로 나타내는 chain rule(연쇄법칙을 이용)

[합성함수 예1] $f(x) = e^{3x^2}$  함수 e^t , 함수 $t = 3x^2$ 조합

[합성함수 예2] $f(x) = e^{-x}$  함수 e^t , 함수 $t = -x$ 조합

- $f(x) = e^{3x^2}$ 을 chain rule로 미분하는 경우, $t = 3x^2$ 으로 놓으면 $f(x) = e^t$

chain rule 적용(약분개념)

$t = 3x^2$ 대입

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\cancel{\partial t}} \frac{\cancel{\partial t}}{\partial x} = \frac{\partial(e^t)}{\partial t} \frac{\partial(3x^2)}{\partial x} = (e^t)(6x) = (e^{3x^2})(6x) = 6xe^{3x^2}$$

- $f(x) = e^{-x}$ 을 chain rule로 미분하는 경우, $t = -x$ 으로 놓으면 $f(x) = e^t$

chain rule 적용(약분개념)

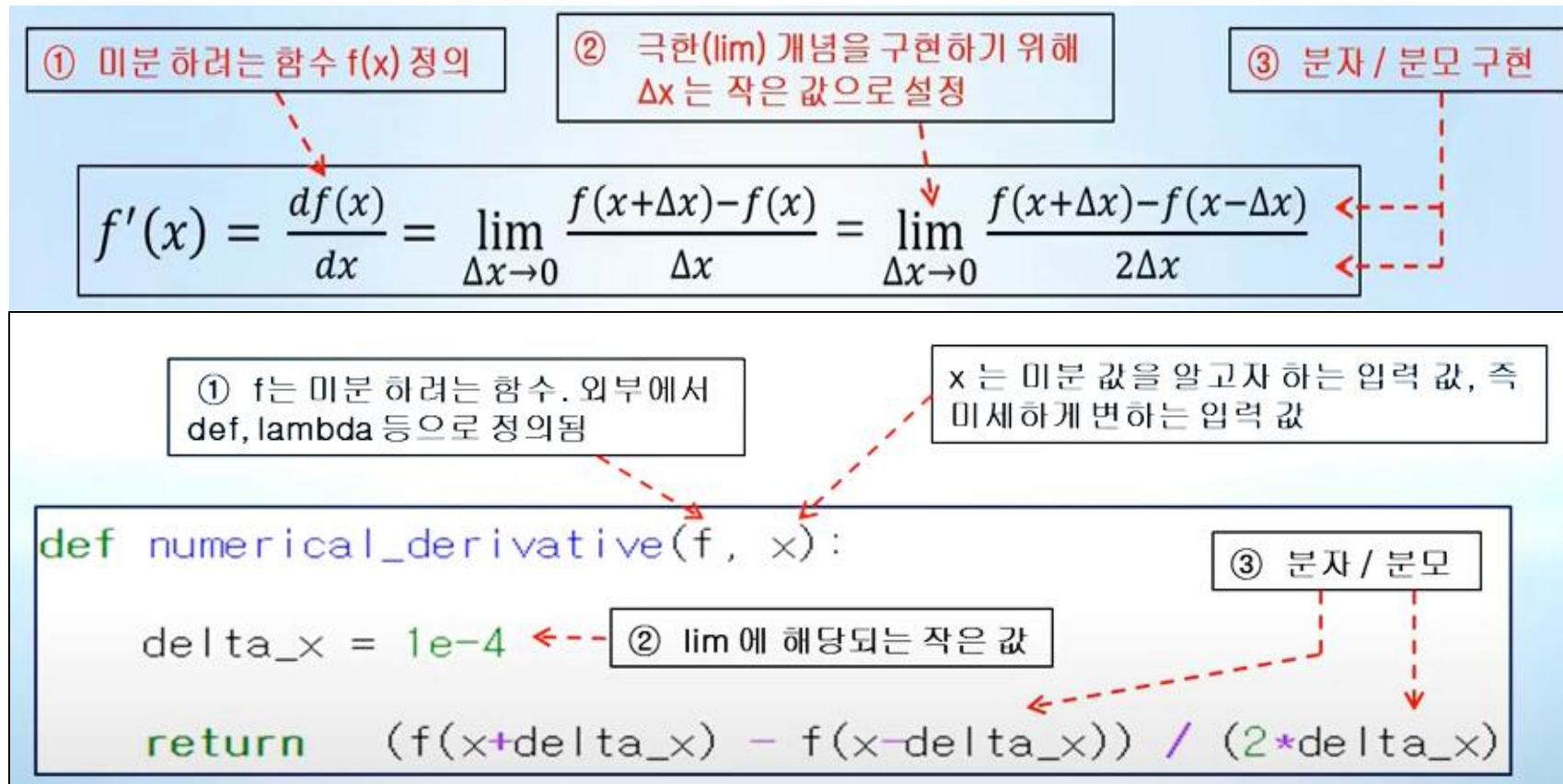
$t = -x$ 대입

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\cancel{\partial t}} \frac{\cancel{\partial t}}{\partial x} = \frac{\partial(e^t)}{\partial t} \frac{\partial(-x)}{\partial x} = (e^t)(-1) = (e^{-x})(-1) = -e^{-x}$$

1. 머신러닝 수식

❖ 수치미분 1차 버전-numerical derivative

- 수치미분은 주어진 입력 값이 미세하게 변할 때 함수 값 f 는 얼마나 변하는지를 계산하는 것



1. 머신러닝 수식

[예제 1] 함수 $f(x) = x^2$ 에서 미분계수 $f'(3)$ 을 구하기. 즉, $x=3$ 에서 값이 미세하게 변할 때, 함수 f 는 얼마나 변하는지 계산하라는 의미

$f'(3)$ 계산 과정 (참고: $f'(x) = 2x$)

$$f'(3) = \lim_{\Delta x \rightarrow 0} \frac{f(3+\Delta x) - f(3-\Delta x)}{2\Delta x}$$

↓ Δx 는 10^{-4} 대입

$$f'(3) = \frac{f(3+1e^{-4}) - f(3-1e^{-4})}{2 * 1e^{-4}}$$

↓ $f(x) = x^2$

$$f'(3) = \frac{(3+1e^{-4})^2 - (3-1e^{-4})^2}{2 * 1e^{-4}}$$

↓ result

$$f'(3) = 6.0$$

```
def my_func1(x):
```

```
    return x**2
```

```
def numerical_derivative(f, x):
```

```
    delta_x = 1e-4
```

```
    return (f(x+delta_x) - f(x-delta_x)) / (2*delta_x)
```

```
result = numerical_derivative(my_func1, 3)
```

```
print("result ==", result)
```

```
result == 6.000000000012662
```


1. 머신러닝 수식

❖ 수치미분-다변수

- 입력 변수가 하나 이상인 다 변수 함수의 경우 입력 변수는 서로 독립적이기 때문에 수치미분 또한 변수의 개수 만큼 개별적으로 계산하여야 함

[예] $f(x, y) = 2x + 3xy + y^3$ 라면, 입력 변수 x, y 두 개 이므로 $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}$ 각각 수치미분 수행.

$f'(1.0, 2.0)$ 값을 계산하기 위해서는,

$$f(x, 2) = 2x + 6x + 8$$

$\Rightarrow x = 1.0$ 에서의 미분계수는 변수 $y = 2.0$ 을 상수로 대입하여 $\frac{\partial f(x, 2)}{\partial x}$ 를 수행

$$f(1, y) = 2 + 3y + y^3$$

$\Rightarrow y = 2.0$ 에서의 미분계수 또한 변수 $x = 1.0$ 인 상수로 대입하여 $\frac{\partial f(1, y)}{\partial y}$ 를 수행

[insight] $f(x, y) = 2x + 3xy + y^3$,인 경우 $f'(1.0, 2.0) = (8.0, 15.0)$ 직관적 이해

$\Rightarrow x = 1.0$ 에서 미분 값을 구한다는 것은, y 값은 2.0 으로 고정한 상태에서, $x = 1.0$ 을 미세하게 변화시킬 때 $f(x, y)$ 는 얼마나 변화는지 알아보겠다는 의미. 즉, $y = 2.0$ 으로 고정된 상태에서 $x = 1.0$ 을 미세하게 변화시키면 $f(x, y)$ 는 8.0 만큼 변한다는 의미

$\Rightarrow y = 2.0$ 에서 미분 값을 구한다는 것은, x 값은 1.0 으로 고정한 상태에서, $y = 2.0$ 을 미세하게 변화시킬 때 $f(x, y)$ 는 얼마나 변화는지 알아보겠다는 의미. 즉, $x = 1.0$ 으로 고정된 상태에서 $y = 2.0$ 을 미세하게 변화시키면 $f(x, y)$ 는 15.0 만큼 변한다는 의미

1. 머신러닝 수식

❖ 수치미분 - 다변수

모든 입력변수에 대해 편미분하기 위해 iterator 획득

다변수 함수

모든변수를 포함하고 있는 numpy객체(배열, 행렬...)

```
def numerical_derivative(f, x): # 수치미분 debug version
    delta_x = 1e-4
    grad = np.zeros_like(x)

    it = np.nditer(x, flags=['multi_index'], op_flags=['readwrite'])

    while not it.finished:
        idx = it.multi_index

        tmp_val = x[idx]
        x[idx] = float(tmp_val) + delta_x
        fx1 = f(x) # f(x+delta_x)

        x[idx] = tmp_val - delta_x
        fx2 = f(x) # f(x-delta_x)
        grad[idx] = (fx1 - fx2) / (2*delta_x)

        x[idx] = tmp_val
        it.iternext()

    return grad
```

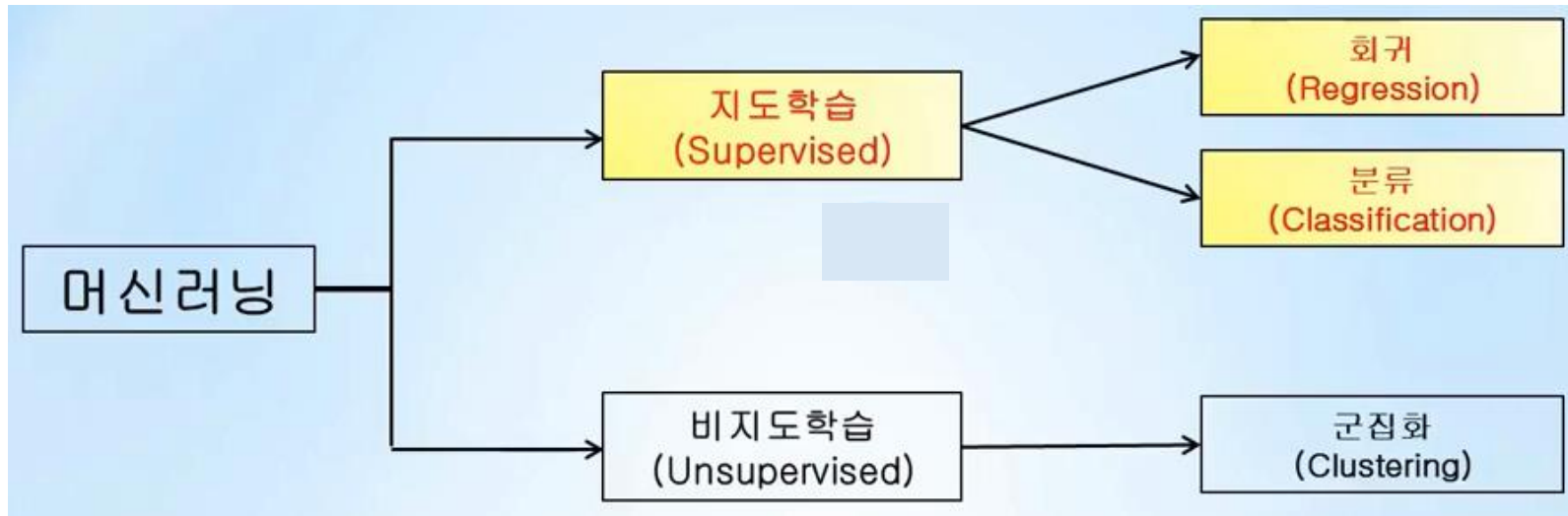
계산된 수치미분 값 저장 변수

Numpy 타입은 mutable이므로 원래 값 보관

하나의 변수에 대한 수치미분 계산

1. 선형회귀

❖ Machin Learning Type



1. 선형회귀

❖ 지도학습

- 입력 값(x)과 정답(t , label)을 포함하는 Training Data를 이용하여 학습을 하고, 그 학습된 결과를 바탕으로 미지의 데이터(Test Data)에 대해 미래 값을 예측(predict)하는 방법 -> 대부분의 머신러닝 문제는 지도학습에 해당됨
- 회귀(Regression)
 - Training Data를 이용하여 연속적인(숫자) 값을 예측하는 것,
 - 예: 공부 시간(입력)과 시험 성적 (정답)을 이용하여 임의의 공부시간에 대한 성적 예측
 - 예 : 집 평수(입력)와 가격 데이터(정답)을 이용하여 임의의 평수 가격 예측
- 분류(Classification)
 - Training Data를 이용하여 주어진 입력 값이 어떤 종류 값인지 구별하는 것
 - 예: 시험공부 시간(입력)과 Pass/Fail(정답)을 이용하여 당락 여부를 예측
 - 예: 집 평수(입력)와 가격 데이터(정답)을 이용하여 가격 분류(Low, Medium, High) 예측

Regression

공부시간 (x)	시험성적 (t)	집평수 (x)	가격 (t)
9	74	20	98
14	81	25	119
21	86	30	131
27	90	40	133
32	88	50	140
37	92	55	196

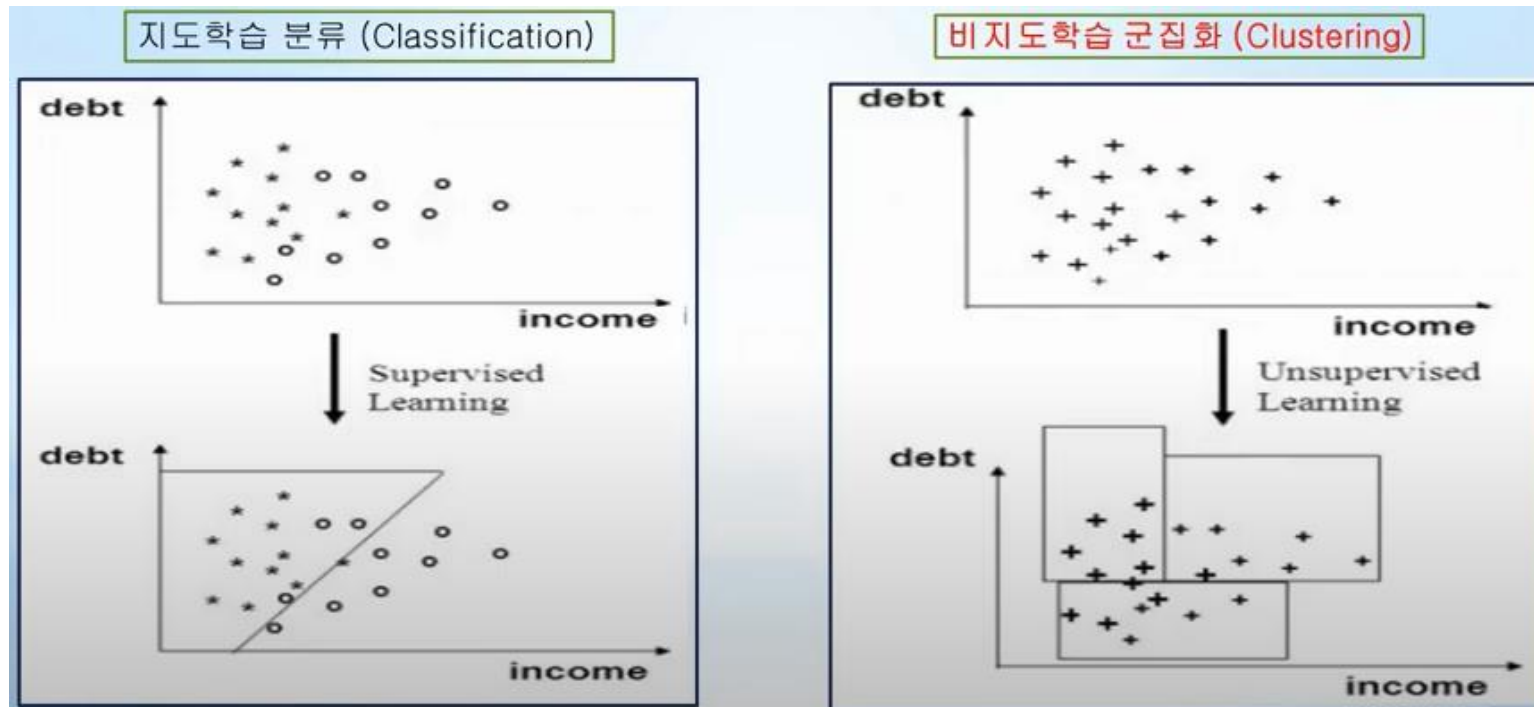
Classification

공부시간 (x)	시험성적 (t)	집평수 (x)	가격 (t)
9	Fail	20	Low
14	Fail	25	Low
21	Pass	30	Medium
27	Pass	40	Medium
32	Pass	50	Medium
37	Pass	55	High

1. 선형회귀

❖ 비지도 학습

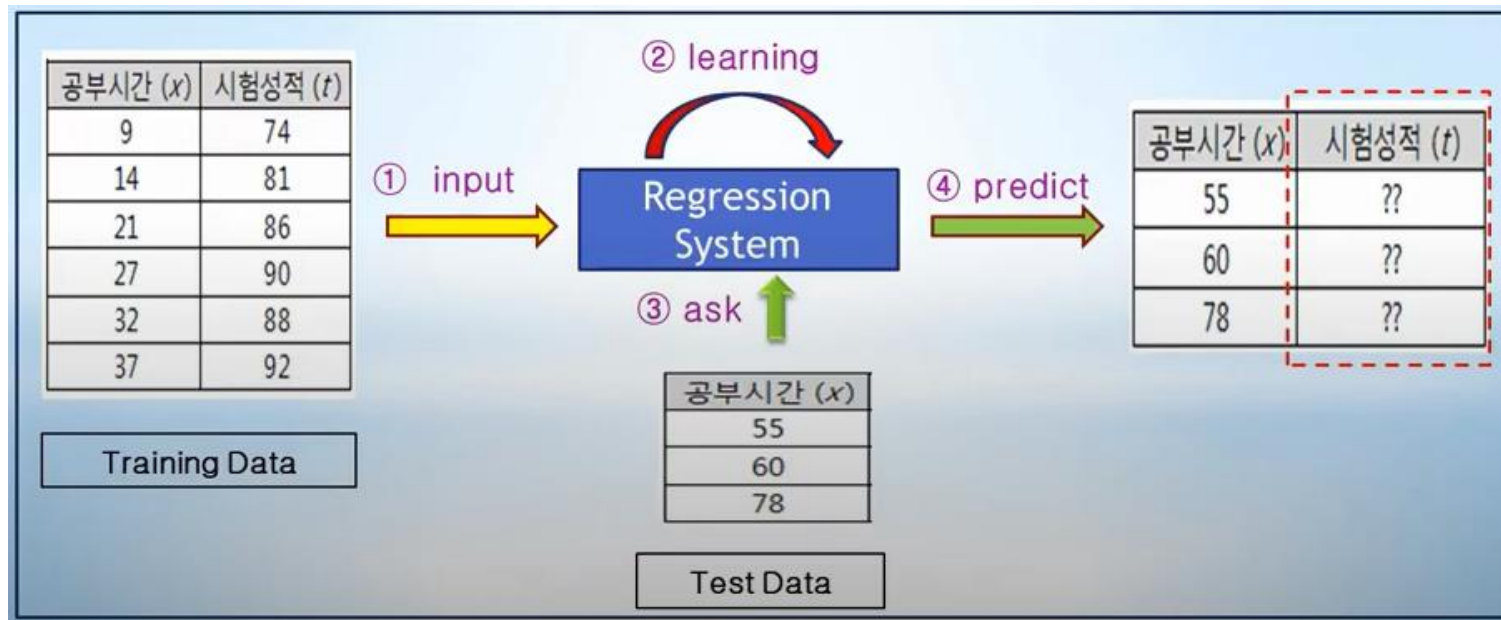
- 트레이닝 데이터에 정답은 없고 입력 데이터만 있기 때문에, 입력에 대한 정답을 찾는 것이 아닌 입력데이터의 패턴, 특성 등을 학습을 통해 발견하는 방법
- 예 : 군집화(Clustering) 알고리즘을 이용한 뉴스 그룹핑, 백화점의 상품 추천 시스템 등



1. 선형회귀

❖ 선형회귀(Linear Regression)

- Training Data를 이용하여 데이터의 특성과 상관관계 등을 파악하고, 그 결과를 바탕으로 Training Data에 없는 미지의 데이터가 주어졌을 경우에, 그 결과를 연속적인 (숫자) 값으로 예측하는 것
- 예: 공부 시간과 시험성적 관계, 집 평수와 집 가격 관계 등



1. 선형회귀

❖ 선형회귀(Linear Regression) 이해

- 데이터에 대한 이해(Data Definition)
- 가설(Hypothesis) 수립
- 손실 계산하기(Compute loss)
- 경사 하강법(Gradient Descent)

2. 선형회귀

❖ 데이터의 이해(Data Definition)

- 공부한 시간과 점수에 대한 상관관계에 대한 선형회귀분석
- **훈련 데이터셋과 테스트 데이터셋**
 - **훈련 데이터 셋** : 예측을 위한 모델을 학습에 사용되는 데이터셋
 - **테스트 데이터 셋**: 학습이 끝난 후 모델의 성능을 테스트 하기 위한 데이터셋
 - 어떤 학생이 1시간 공부를 했더니 2점, 다른 학생이 2시간 공부를 했더니 4점, 또 다른 학생이 3시간을 공부했더니 6점을 맞았습니다. 그렇다면, 내가 4시간을 공부한다면 몇 점을 맞을 수 있을까요?

Hours (x)	Points (y)	
1	2	Training dataset
2	4	
3	6	
4	?	Test dataset



2. 선형회귀

❖ 데이터의 이해(Data Definition)

■ 훈련 데이터셋의 구성

- 모델을 학습시키기 위한 데이터는 파이토치의 텐서의 형태(torch.tensor)를 가지고 있어야 함
- 입력(문제)과 출력(정답)을 각기 다른 텐서에 저장, 보편적으로 입력은 x , 출력은 y 를 사용하여 표기
- x_{train} 은 공부한 시간, y_{train} 은 그에 맵핑되는 점수를 의미

```
x_train = torch.FloatTensor([[1], [2], [3]])  
y_train = torch.FloatTensor([[2], [4], [6]])
```

$$X_{\text{train}} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \quad Y_{\text{train}} = \begin{pmatrix} 2 \\ 4 \\ 6 \end{pmatrix}$$

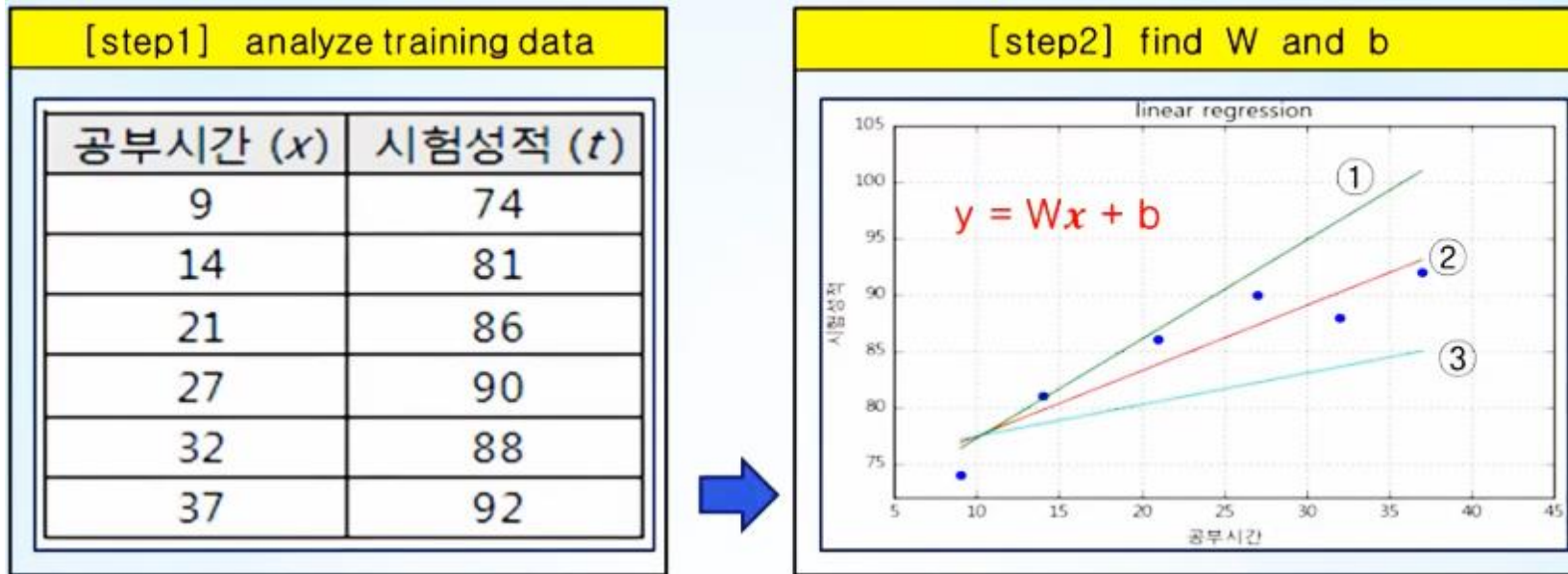
2. 선형회귀

❖ 가설(Hypothesis) 수립

- 머신러닝에서 입력(x)과 출력(y)로 $y=Wx+b$ 식을 세울 때 이 식을 가설(Hypothesis)라고 함
- 선형 회귀의 가설(직선의 방정식), y 대신 $H(x)$ 를 사용하기도 함.

$$y = Wx + b, \quad H(x) = Wx + b \quad W : \text{가중치(Weight)}, b : \text{편향(bias)}$$

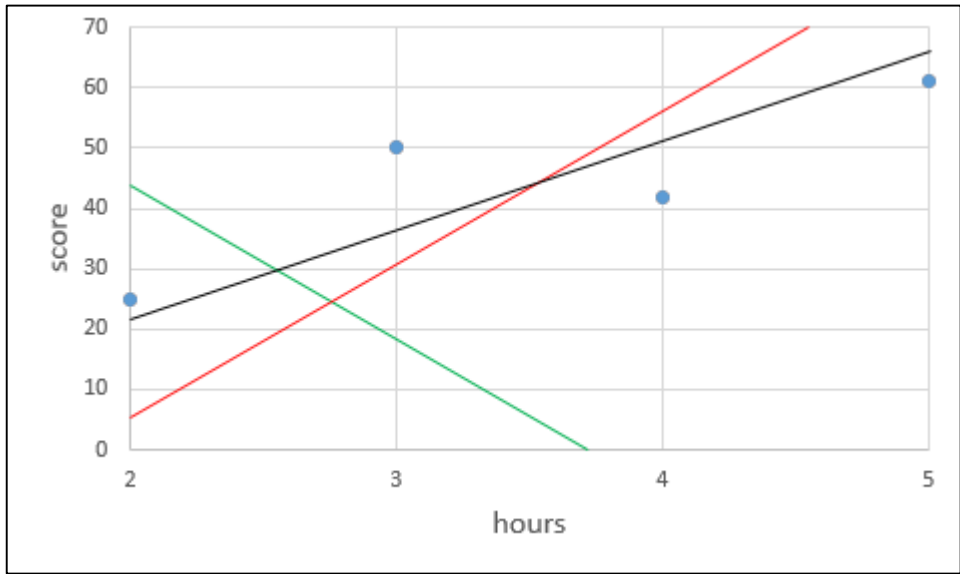
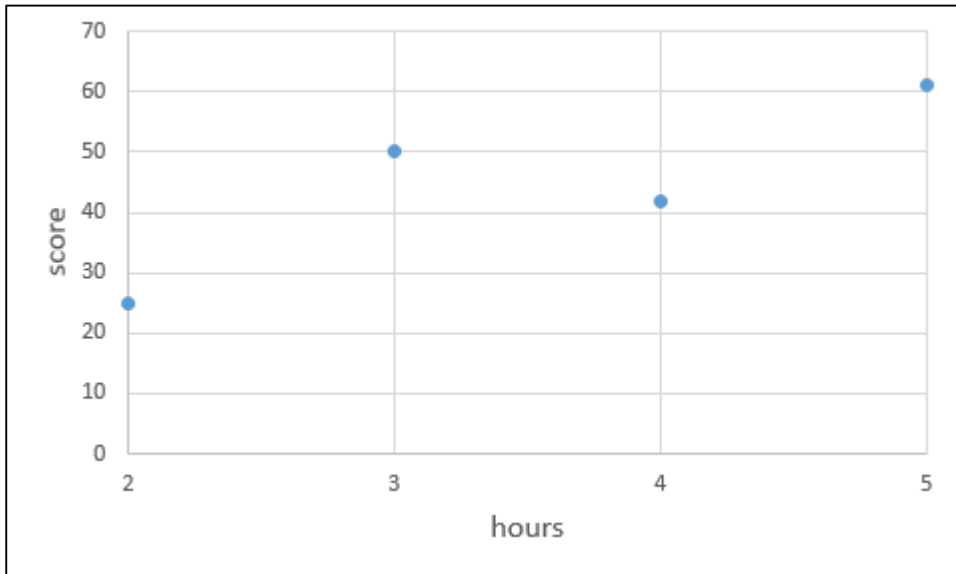
- Trainig data의 특징을 가장 잘 표현할 수 있는 가중치 W 와 바이어스 b 를 찾는 것이 선형회귀 학습의 목적임



2. 선형회귀

❖ 비용 함수(Cost function)

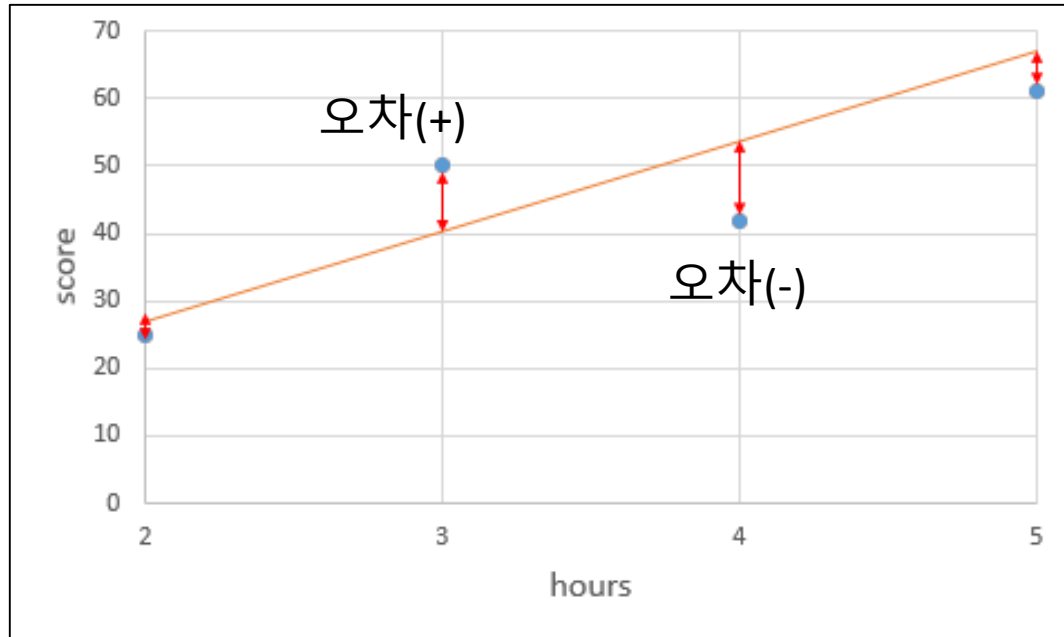
- 비용 함수(cost function) = 손실 함수(loss function) = 오차 함수(error function) = 목적 함수(objective function)
- 선형회귀 학습의 목적 도달 : 비용함수(cost function)이 최소 값(0에 가까운)인 $y(H : Hypothesis)$ 를 구함
- 어떤 4개의 훈련 데이터가 있고, 이를 2차원 그래프에 4개의 점으로 표현한 상태(왼쪽)
- 선형회귀분석의 목적은 4개의 점을 가장 잘 표현하는 직선 찾는 것



2. 선형회귀

❖ 비용 함수(Cost function)

- 오차(error) = 실제값(정답) - 예측된 값($H(x)$ 식에 의해 예측된 값)



$y=13x+1$ 에 대한 실제값, 예측 값, 오차

hours(x)	2	3	4	5
실제값	25	50	42	61
예측값	27	40	53	66
오차	-2	10	-9	-5

2. 선형회귀

❖ 비용 함수(Cost function)

- 총 오차(total error)

- 오차들의 합 = $(-2) + 10 + (-9) + (-5) = -6 \Rightarrow$ 문제점 : 오차 값 부호(+, -)에 의해 0에 가까운 값 구해 짐

- 오차 값을 제곱하여 더함
$$\sum_{i=1}^n [y^{(i)} - H(x^{(i)})]^2 = (-2)^2 + 10^2 + (-9)^2 + (-5)^2 = 210$$

- 평균 제곱 오차(Mean Squared Error, MSE)
$$\frac{1}{n} \sum_{i=1}^n [y^{(i)} - H(x^{(i)})]^2 = 210/4 = 52.5$$

- 비용 함수(Cost function)로 사용

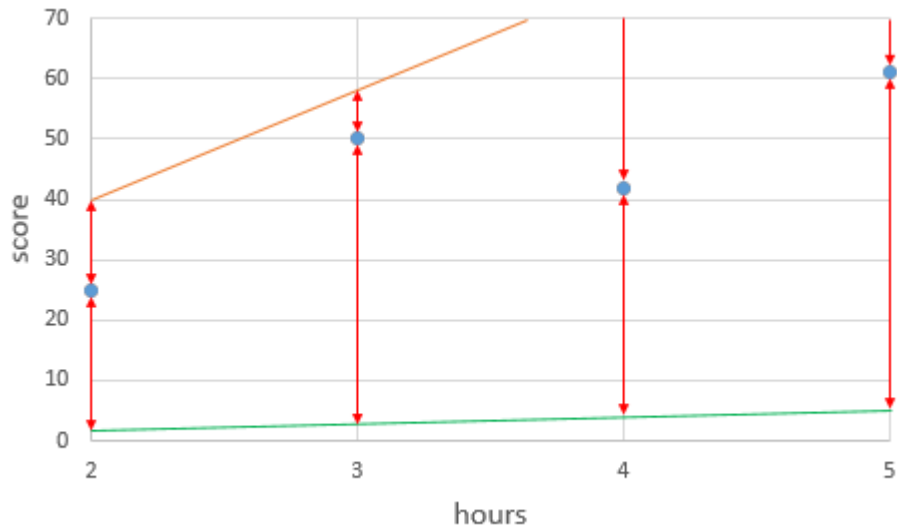
- 평균 제곱 오차를 W와 b에 의한 비용 함수(Cost function)로 재정의
$$cost(W, b) = \frac{1}{n} \sum_{i=1}^n [y^{(i)} - H(x^{(i)})]^2$$

- Cost(W,b)를 최소가 되게 W와 b를 구하면 훈련데이터를 잘 나타내는 직선을 구할 수 있음

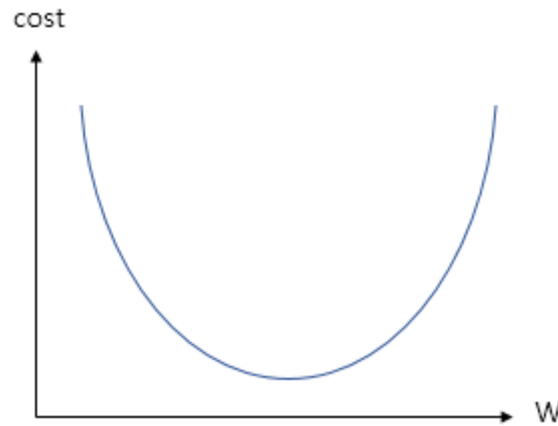
2. 선형회귀

❖ 옵티마이저 - 경사 하강법(Gradient Descent)

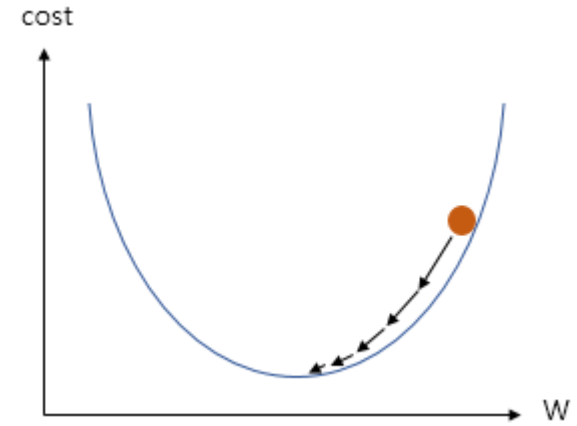
- 비용 함수(Cost Function)의 값을 최소로 하는 W 와 b 를 찾는 방법
- $H(x)=Wx$ ($b=0$)로 $\text{cost}(W)$ 계산하여 경사 하강법 적용



주황색 직선 : $y=20x$, 초록색 직선 : $y=x$



$H(x)=Wx$ ($b=0$)일 때 W 와 $\text{cost}(W)$ 사이 관계 그래프



cost 가 가장 최소값을 가지게 하는 w 를 찾기, w 가 무한대로 크거나 작으면 cost 값이 무한대로 커짐

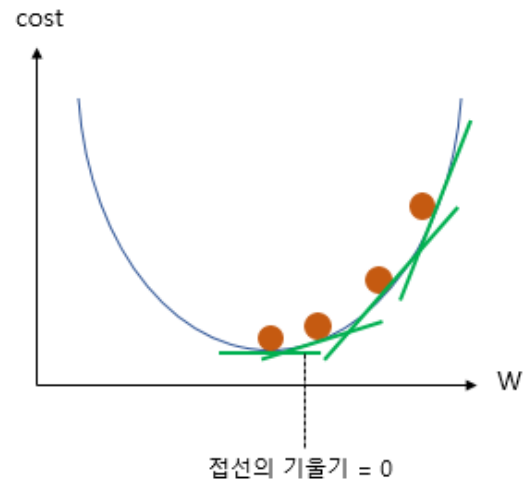
2. 선형회귀

❖ 옵티마이저 - 경사 하강법(Gradient Descent)

- 최소 cost를 찾는 경사 하강법(Gradient Descent)
 - 임의의 초기값 w 값을 정한 뒤에, 맨 아래의 볼록한 부분을 향해 점차 w 의 값을 수정
 - 임의의 w 에서 미분을 하여 기울기를 구하고 기울기를 따라 하강함
 - 기울기가 수평(0)이 되는 지점이 cost가 최소 값을 가짐
 - 현재 w 에 접선의 기울기를 구해 특정 숫자 α 를 곱한 값을 빼서 새로운 w 로 업데이트

$$\text{기울기} = \frac{\partial \text{cost}(W)}{\partial W}$$

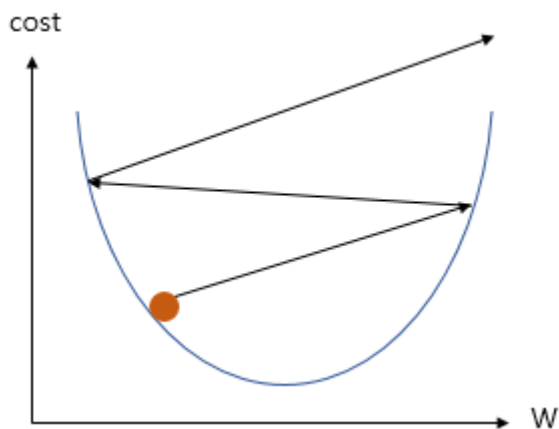
- 기울기가 음수일 때 : w 의 값이 증가: $W := W - \alpha \times (\text{음수기울기}) = W + \alpha \times (\text{양수기울기})$
- 기울기가 양수일 때: w 의 값의 감소: $W := W - \alpha \times (\text{양수기울기})$
- cost가 최소인 w 업데이트 식: $W := W - \alpha \frac{\partial}{\partial W} \text{cost}(W)$



2. 선형회귀

❖ 옵티마이저 - 경사 하강법(Gradient Descent)

- 경사 하강법에서 학습률(learning rate)
 - 학습률 α 는 w 의 값을 변경할 때, 얼마나 큰폭 변경할지를 결정
 - 학습률이 너무 크면 최소 값을 찾지 못하고 발산(오버슈팅)이 발생
 - 학습률이 너무 작으면 학습속도가 느려 짐, 적절한 학습률 선택 중요함.



3. 선형회귀 실습(Python)

❖ single data-1

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

훈련데이터

```
x_data1=np.array([1,2,3,4,5]).reshape(5,1)
t_data1=np.array([3,5,7,9,11]).reshape(5,1)
print(x_data1)
print(t_data1)
```

초기 가중치와 바이어스

```
w=np.random.rand(1,1)
b=np.random.rand(1)
print(w,b)
```

손실함수

```
def loss_func1(x, t):
    y=np.dot(x,w)+b #y=xw+b
    return (np.sum((t-y)**2))/(len(x))
```

미분함수

```
def numerical_derivative1(f,x):
    delta_x=1e-4
    grad=np.zeros_like(x)

    it=np.nditer(x, flags=['multi_index'],
                  op_flags=['readwrite'])

    while not it.finished:
        idx=it.multi_index
        tmp_val=x[idx]

        x[idx]=float(tmp_val)+delta_x
        fx1=f(x) # f(x+delta_x)

        x[idx]=float(tmp_val)-delta_x
        fx2=f(x) # f(x-delta_x)

        grad[idx]=(fx1-fx2)/(2*delta_x)

        x[idx]=tmp_val
        it.iternext()

    return grad
```

3. 선형회귀 실습(Python)

❖ single data-2

예측 함수

```
def predict1(x):  
    y=np.dot(x, w)+b  
    return y
```

학습

```
cost_list=[]  
learning_rate=1e-2  
  
f=lambda x: loss_func1(x_data1, t_data1)  
  
print("Initial error value=", loss_func1(x_data1, t_data1),  
      "Initial w=", w, "b=",b)  
  
for step in range(201):  
    w-=learning_rate*numerical_derivative1(f, w)  
    b-=learning_rate*numerical_derivative1(f, b)  
    cost_list.append([step, loss_func1(x_data1, t_data1)])  
  
    if step % 100==0:  
        print("step=", step, "error value=",  
              loss_func1(x_data1, t_data1), "w=",w, "b=",b)
```

4. 다중선형회귀 실습(python)

❖ multiple data-1

```
import numpy as np

x_data1 = [[73., 80., 75.],
           [93., 88., 93.],
           [89., 91., 90.],
           [96., 98., 100.],
           [73., 66., 70.]]

t_data1 = [[152.],
           [185.],
           [180.],
           [196.],
           [142.]]

x_data=np.array(x_data1)
t_data=np.array(t_data1)
print(x_data)
print(t_data)
```

```
# weight와 bias
w=np.random.rand(3,1)
b=np.random.rand(1)
print(w,b)
```

손실함수, 미분함수, 예측함수 생략
(single-data와 동일)

```
learning_rate=1e-6

f=lambda x: loss_func(x_data, t_data)

print("Initial error value=", loss_func(x_data, t_data),
      "Initial w=", w, "b=",b)

for step in range(20001):
    w-=learning_rate*numerical_derivative(f, w)
    b-=learning_rate*numerical_derivative(f, b)

    if step % 400==0:
        print("step=", step, "error value=",
              loss_func(x_data, t_data), "w=", w, "b=",b)
```

4. 다중선형회귀 실습(python)

❖ 실습예제:

- data/data-01-test-score.csv 파일을 사용하여 다중 선형회귀분석(Multivariable Linear regression)을 수행하라.

```
# csv 파일 불러오기
import pandas as pd
df=pd.read_csv('data/data-01-test-score.csv',header=None)
df
```

```
# 데이터프레임의 values 확인
df.values # numpy array 값
```

```
# df.values에서 입력값(data)과 정답(t_data) 값 추출
x_data=df.values[:, :-1]
print(data)
t_data=df.values[:, [-1]]
print(target)
```


5. Pytorch 선형회귀

❖ 파이토치로 선형회귀 구현

1. 기본세팅

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
```

현재 실행하고 있는 파이썬 코드를
재실행해도 다음에도 같은 결과가 나
오도록 랜덤 시드(random seed) 설정.

```
torch.manual_seed(1)
```

2. 변수 선언

```
x_train = torch.FloatTensor([[1], [2], [3]])
y_train = torch.FloatTensor([[2], [4], [6]])
```

```
print(x_train)
print(x_train.shape)
print(y_train)
print(y_train.shape)
```

3. 가중치와 편향의 초기화

가중치 W 와 편향 b 를 0으로 초기화하고
학습을 통해 값이 변경되는 변수임을 명
시함.

$$y = 0 \times x + 0$$

```
W = torch.zeros(1, requires_grad=True)
print(W)
```

```
b = torch.zeros(1, requires_grad=True)
print(b)
```

5. Pytorch 선형회귀

❖ 파이토치로 선형회귀 구현

4. 가설 세우기

직선의 방정식에 해당되는 가설을 선언.

$H(x) = Wx + b \Rightarrow H(x) = XW + b$ 로 변환

```
hypothesis = x_train * W + b  
print(hypothesis)
```

#5. 비용 함수 선언하기

선형 회귀의 비용 함수에 해당되는 평균 제곱 오차를 선언.

```
cost = torch.mean((hypothesis - y_train) ** 2)  
print(cost)
```

6. 경사 하강법 구현하기

'SGD'는 경사 하강법의 종류, lr은 학습률(learning rate)를 의미
학습 대상인 w와 b가 SGD의 입력됨.

```
optimizer = optim.SGD([W, b], lr=0.01)
```

```
# gradient를 0으로 초기화  
optimizer.zero_grad()
```

```
# 비용 함수를 미분하여 gradient 계산  
cost.backward()
```

```
# w와 b를 업데이트  
optimizer.step()
```

5. Pytorch 선형회귀

```
# 데이터
x_train = torch.FloatTensor([[1], [2], [3]])
y_train = torch.FloatTensor([[2], [4], [6]])

# 모델 초기화
W = torch.zeros(1, requires_grad=True)
b = torch.zeros(1, requires_grad=True)

# optimizer 설정
optimizer = optim.SGD([W, b], lr=0.01)

nb_epochs = 1999 # 원하는만큼 경사 하강법을 반복
```

```
for epoch in range(nb_epochs + 1):

    # H(x) 계산
    hypothesis = x_train * W + b

    # cost 계산
    cost = torch.mean((hypothesis - y_train) ** 2)

    # cost로 H(x) 개선
    optimizer.zero_grad()
    cost.backward()
    optimizer.step()

    # 100번마다 로그 출력
    if epoch % 100 == 0:
        print('Epoch {:4d}/{:} W: {:.3f}, b: {:.3f} Cost: {:.6f}'.format(
            epoch, nb_epochs, W.item(), b.item(), cost.item()
        ))
```

5. Pytorch 선형회귀

❖ optimizer.zero_grad()가 필요한 이유

- 파이토치는 미분을 통해 얻은 기울기를 이전에 계산된 기울기 값에 누적시키는 특징이 있음

```
import torch
w = torch.tensor(2.0, requires_grad=True)
print(w)

nb_epochs = 20
for epoch in range(nb_epochs + 1):

    z = 2*w

    z.backward()
    print('수식을 w로 미분한 값 : {}'.format(w.grad))
```

```
tensor(2., requires_grad=True)
수식을 w로 미분한 값 : 2.0
수식을 w로 미분한 값 : 4.0
수식을 w로 미분한 값 : 6.0
수식을 w로 미분한 값 : 8.0
수식을 w로 미분한 값 : 10.0
수식을 w로 미분한 값 : 12.0
수식을 w로 미분한 값 : 14.0
수식을 w로 미분한 값 : 16.0
수식을 w로 미분한 값 : 18.0
수식을 w로 미분한 값 : 20.0
수식을 w로 미분한 값 : 22.0
수식을 w로 미분한 값 : 24.0
수식을 w로 미분한 값 : 26.0
수식을 w로 미분한 값 : 28.0
수식을 w로 미분한 값 : 30.0
수식을 w로 미분한 값 : 32.0
수식을 w로 미분한 값 : 34.0
수식을 w로 미분한 값 : 36.0
수식을 w로 미분한 값 : 38.0
수식을 w로 미분한 값 : 40.0
수식을 w로 미분한 값 : 42.0
```

5. Pytorch 선형회귀

❖ torch.manual_seed()를 하는 이유

- torch.manual_seed()를 사용한 프로그램의 결과는 다른 컴퓨터에서 실행시켜도 동일한 결과를 얻을 수 있음

```
torch.manual_seed(3)
print('랜덤 시드가 3일 때')
for i in range(1,3):
    | print(torch.rand(1))
```

✓ 0.6s

랜덤 시드가 3일 때
tensor([0.0043])
tensor([0.1056])

```
torch.manual_seed(5)
print('랜덤 시드가 5일 때')
for i in range(1,3):
    | print(torch.rand(1))
```

✓ 0.3s

랜덤 시드가 5일 때
tensor([0.8303])
tensor([0.1261])

```
torch.manual_seed(3)
print('랜덤 시드가 다시 3일 때')
for i in range(1,3):
    | print(torch.rand(1))
```

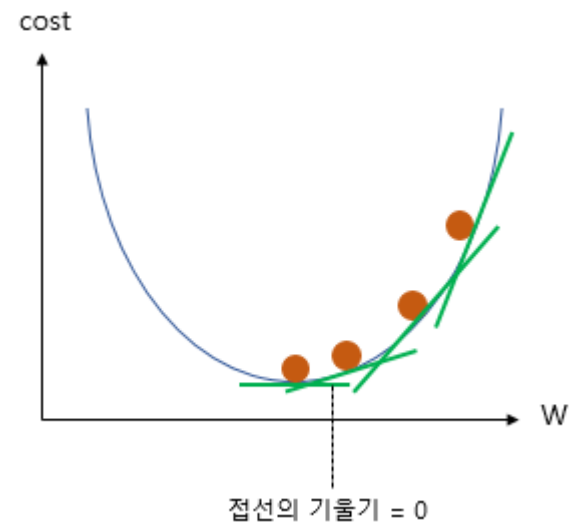
✓ 0.3s

랜덤 시드가 다시 3일 때
tensor([0.0043])
tensor([0.1056])

5. Pytorch 선형회귀

❖ 자동 미분(Autograd)- 경사 하강법 리뷰

- 경사 하강법은 비용 함수를 미분하여 이 함수의 기울기(gradient)를 구해서 비용이 최소화 되는 방향을 찾아내는 알고리즘
- 비용 함수(=손실 함수, 오차 함수)는 비용이 최소화 되는 방향이라는 표현 대신 손실이 최소화 되는 방향 또는 오차를 최소화 되는 방향임
- 모델이 복잡해질수록 경사 하강법을 넘파이 등으로 직접 코딩하는 것은 까다로움
- 파이토치에서는 이런 수고를 하지 않도록 자동 미분(Autograd)을 지원



5. Pytorch 선형회귀

❖ 자동 미분(Autograd) 실습

$2w^2 + 5$ 자동미분하기

```
#자동 미분(Autograd) 실습하기

w = torch.tensor(2.0, requires_grad=True)
y = w**2
z = 2*y + 5

z.backward() # 기울기 계산

# w.grad를 출력하면 w가 속한 수식을 w로 미분한 값이 저장
print('수식을 w로 미분한 값 : {}'.format(w.grad)) #
```

✓ 0.3s

수식을 w로 미분한 값 : 8.0

5. Pytorch 선형회귀 분석

❖ 다중선형회귀(Multivariable Linear regression)

- 다중선형회귀 데이터 : 독립변수가 다수(예: 3)인 것

Quiz 1 (x1)	Quiz 2 (x2)	Quiz 3 (x3)	Final (y)
73	80	75	152
93	88	93	185
89	91	80	180
96	98	100	196
73	66	70	142

가설: $H(x) = w_1x_1 + w_2x_2 + w_3x_3 + b$

훈련 데이터

```
x1_train = torch.FloatTensor([[73], [93], [89], [96], [73]])
x2_train = torch.FloatTensor([[80], [88], [91], [98], [66]])
x3_train = torch.FloatTensor([[75], [93], [90], [100], [70]])
y_train = torch.FloatTensor([[152], [185], [180], [196], [142]])
```

가중치 w와 편향 b 초기화

```
w1 = torch.zeros(1, requires_grad=True)
w2 = torch.zeros(1, requires_grad=True)
w3 = torch.zeros(1, requires_grad=True)
b = torch.zeros(1, requires_grad=True)
```

optimizer 설정

```
optimizer = optim.SGD([w1, w2, w3, b], lr=1e-5)
```

H(x) 계산

```
hypothesis = x1_train * w1 + x2_train * w2 + x3_train * w3 + b
```


5. Pytorch 선형회귀 분석4.

❖ 다중선형회귀(Multivariable Linear regression)

- 행렬의 곱벡터와 행렬 연산으로 바꾸기

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} = \begin{bmatrix} 58 \\ \end{bmatrix}$$

벡터 연산으로 이해하기

$$H(X) = w_1x_1 + w_2x_2 + w_3x_3 \quad \text{식을}$$

$$\begin{pmatrix} x_1 & x_2 & x_3 \end{pmatrix} \cdot \begin{pmatrix} w_1 \\ w_2 \\ w_3 \end{pmatrix} = (x_1w_1 + x_2w_2 + x_3w_3)$$

$$H(X) = XW \quad \text{변환}$$

5. Pytorch 선형회귀

❖ 다중선형회귀(Multivariable Linear regression)

- 행렬의 연산으로 변경

Quiz 1 (x1)	Quiz 2 (x2)	Quiz 3 (x3)	Final (y)
73	80	75	152
93	88	93	185
89	91	80	180
96	98	100	196
73	66	70	142

$$X = \begin{pmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \\ x_{41} & x_{42} & x_{43} \\ x_{51} & x_{52} & x_{53} \end{pmatrix}$$

$$H(X) = XW + B$$

$$H(X) = XW$$
$$\begin{pmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \\ x_{41} & x_{42} & x_{43} \\ x_{51} & x_{52} & x_{53} \end{pmatrix} \begin{pmatrix} w_1 \\ w_2 \\ w_3 \end{pmatrix} = \begin{pmatrix} x_{11}w_1 + x_{12}w_2 + x_{13}w_3 \\ x_{21}w_1 + x_{22}w_2 + x_{23}w_3 \\ x_{31}w_1 + x_{32}w_2 + x_{33}w_3 \\ x_{41}w_1 + x_{42}w_2 + x_{43}w_3 \\ x_{51}w_1 + x_{52}w_2 + x_{53}w_3 \end{pmatrix}$$

$$\begin{pmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \\ x_{41} & x_{42} & x_{43} \\ x_{51} & x_{52} & x_{53} \end{pmatrix} \begin{pmatrix} w_1 \\ w_2 \\ w_3 \end{pmatrix} + \begin{pmatrix} b \\ b \\ b \\ b \\ b \end{pmatrix} = \begin{pmatrix} x_{11}w_1 + x_{12}w_2 + x_{13}w_3 + b \\ x_{21}w_1 + x_{22}w_2 + x_{23}w_3 + b \\ x_{31}w_1 + x_{32}w_2 + x_{33}w_3 + b \\ x_{41}w_1 + x_{42}w_2 + x_{43}w_3 + b \\ x_{51}w_1 + x_{52}w_2 + x_{53}w_3 + b \end{pmatrix}$$

5. Pytorch 선형회귀

```
x_train = torch.FloatTensor([[73, 80, 75],
                              [93, 88, 93],
                              [89, 91, 80],
                              [96, 98, 100],
                              [73, 66, 70]])
y_train = torch.FloatTensor([[152], [185], [180], [196], [142]])

# 모델 초기화
W = torch.zeros((3, 1), requires_grad=True)
b = torch.zeros(1, requires_grad=True)
# optimizer 설정
optimizer = optim.SGD([W, b], lr=1e-5)

nb_epochs = 20
for epoch in range(nb_epochs + 1):
```

```
    for epoch in range(nb_epochs + 1):

        # H(x) 계산
        # 편향 b는 브로드 캐스팅되어 각 샘플에 더해짐.
        hypothesis = x_train.matmul(W) + b

        # cost 계산
        cost = torch.mean((hypothesis - y_train) ** 2)

        # cost로 H(x) 개선
        optimizer.zero_grad()
        cost.backward()
        optimizer.step()

        print('Epoch {:4d}/{:4d} hypothesis: {} Cost: {:.6f}'.format(
            epoch, nb_epochs, hypothesis.squeeze().detach(), cost.item()
        ))
```

6. nn.Module로 구현하는 선형 회귀

❖ nn.Module

- 파이토치에서 이미 구현되어 제공되고 있는 함수들을 불러올 수 있음
- `nn.Linear()` : 선형 회귀 모델을 구현 함수
- `nn.functional.mse_loss()` : 평균 제곱오차 구현 함수
- 실습 : 단순선형회귀 분석, 다중 선형회귀 분석

6. nn.Module로 구현하는 선형 회귀- single data 실습

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import numpy as np
```

```
# 훈련데이터
torch.manual_seed(1)
x_data=torch.FloatTensor([[1],[2],[3],[4],[5]])
t_data=torch.FloatTensor([[3],[5],[7],[9],[11]])
```

```
# 모델 생성
model=nn.Linear(1,1) # input_dim=1, output_dim=1
print(list(model.parameters()))
```

```
# 최적화
optimizer=optim.SGD(model.parameters(), lr=0.01)
```

```
# 학습
nb_epochs=20000

for epoch in range(nb_epochs):
    prediction=model(x_data)
    cost=F.mse_loss(prediction, t_data)

    optimizer.zero_grad()
    cost.backward()
    optimizer.step()

    if epoch%400==0:
        print('Epoch:', epoch, 'Cost:',cost.item())
```

```
# 예측
new_data=torch.FloatTensor([[10]])
y_pred=model(new_data)
print(y_pred)
print(list(model.parameters()))
```

6. nn.Module로 구현하는 선형 회귀- multi data 실습

❖ 실습 예제 1 :

```
# 데이터
x_train = torch.FloatTensor([[73, 80, 75],
                             [93, 88, 93],
                             [89, 91, 90],
                             [96, 98, 100],
                             [73, 66, 70]])
y_train = torch.FloatTensor([[152], [185], [180], [196], [142]])
```

❖ 실습 예제 2 :

- data/data-01-test-socre.csv 파일을 사용하여 다중 선형회귀분석(Multivariable Linear regression)을 수행하라.

7. 클래스로 파이토치 모델 구현하기

❖ 단순선형회귀 구현

```
torch.manual_seed(1)
x_data=torch.FloatTensor([[1],[2],[3],[4],[5]])
t_data=torch.FloatTensor([[3],[5],[7],[9],[11]])
```

```
class LinearRegressionModel(nn.Module):
    def __init__(self): # 모델 생성 관련 함수
        super().__init__()
        self.linear=nn.Linear(1,1)

    def forward(self, x): # 학습관련 함수
        return self.linear(x)
```

```
# 모델 생성 및 최적화
model=LinearRegressionModel()
print(list(model.parameters()))
optimizer=optim.SGD(model.parameters(), lr=0.01)
```

```
# 학습
nb_epochs=20000
for epoch in range(nb_epochs):
    prediction=model(x_data)
    cost=F.mse_loss(prediction, t_data)
    optimizer.zero_grad()
    cost.backward()
    optimizer.step()

    if epoch%400==0:
        print('Epoch', epoch, 'Cost:', cost.item())
```

```
# 예측
new_data=torch.FloatTensor([[10]])
pred_y=model(new_data)
print(pred_y)
print(list(model.parameters()))
```

7. 클래스로 파이토치 모델 구현하기

❖ 다중선형회귀 구현

데이터

```
x_train = torch.FloatTensor([[73, 80, 75],
                              [93, 88, 93],
                              [89, 91, 90],
                              [96, 98, 100],
                              [73, 66, 70]])
y_train = torch.FloatTensor([[152], [185],
                              [180], [196], [142]])
```

모델 생성과 학습을 위한 클래스

```
class MultiLinearRegressionModel(nn.Module):
    def __init__(self):
        super().__init__()
        # 다중선형회귀, input_dim=3, output_dim=1
        self.linear=nn.Linear(3,1)

    def forward(self, x):
        return self.linear(x)
```

모델 생성

```
model=MultiLinearRegressionModel()
optimizer=optim.SGD(model.parameters(), lr=1e-6)
```

학습

```
nb_epochs=20000
for epoch in range(nb_epochs):
    prediction=model(x_train)
    cost=F.mse_loss(prediction, y_train)
    optimizer.zero_grad()
    cost.backward()
    optimizer.step()

    if epoch%400==0:
        print('Epoch', epoch, 'Cost:', cost.item())
```

예측

```
new_data=torch.FloatTensor([[80,90,75]])
pred_y=model(new_data)
print(pred_y)
print(list(model.parameters()))
```


8. Tensorflow 활용 선형회귀 분석

```
import numpy as np
import tensorflow as tf

# 데이터 정의
x_data = np.array([[1,2],[2,3],[3,4],[4,5],[5,6]], dtype=np.float32)
t_data = np.array([[3],[5],[7],[9],[11]], dtype=np.float32)
```

```
# 모델 정의:  $y = wx + b$ 
class LinearRegressionModel(tf.Module):
    def __init__(self):
        # w는 (2,1), b는 (1,) 형태
        self.W = tf.Variable(tf.random.normal([2,1]), name='weight')
        self.b = tf.Variable(tf.random.normal([1]), name='bias')

    def __call__(self, x):
        return tf.matmul(x, self.W) + self.b
```

8. Tensorflow 활용 선형회귀 분석

```
# 손실 함수: MSE
```

```
def loss_fn(model, x, t):  
    y = model(x)  
    return tf.reduce_mean(tf.square(y - t))
```

```
# 학습
```

```
model = LinearRegressionModel()  
optimizer = tf.optimizers.SGD(learning_rate=0.01)  
  
for step in range(1000):  
    with tf.GradientTape() as tape:  
        loss = loss_fn(model, x_data, t_data)  
        gradients = tape.gradient(loss, [model.W, model.b])  
        optimizer.apply_gradients(zip(gradients, [model.W, model.b]))  
  
    if step % 100 == 0:  
        print(f"Step:{step}, Loss:{loss.numpy()}, W: {model.W.numpy().flatten()}, b: {model.b.numpy()}")
```

```
# 예측
```

```
print("\n예측 결과:")  
print(model(x_data).numpy())
```

7. 클래스로 파이토치 모델 구현하기

❖ 실습

- data/data-01-test-score.csv file을 활용하여 Pytorch클래스 모델 실습하기.
- data/data-01-test-score.csv file을 활용하여 Tensorflow 클래스 모델 실습하기.