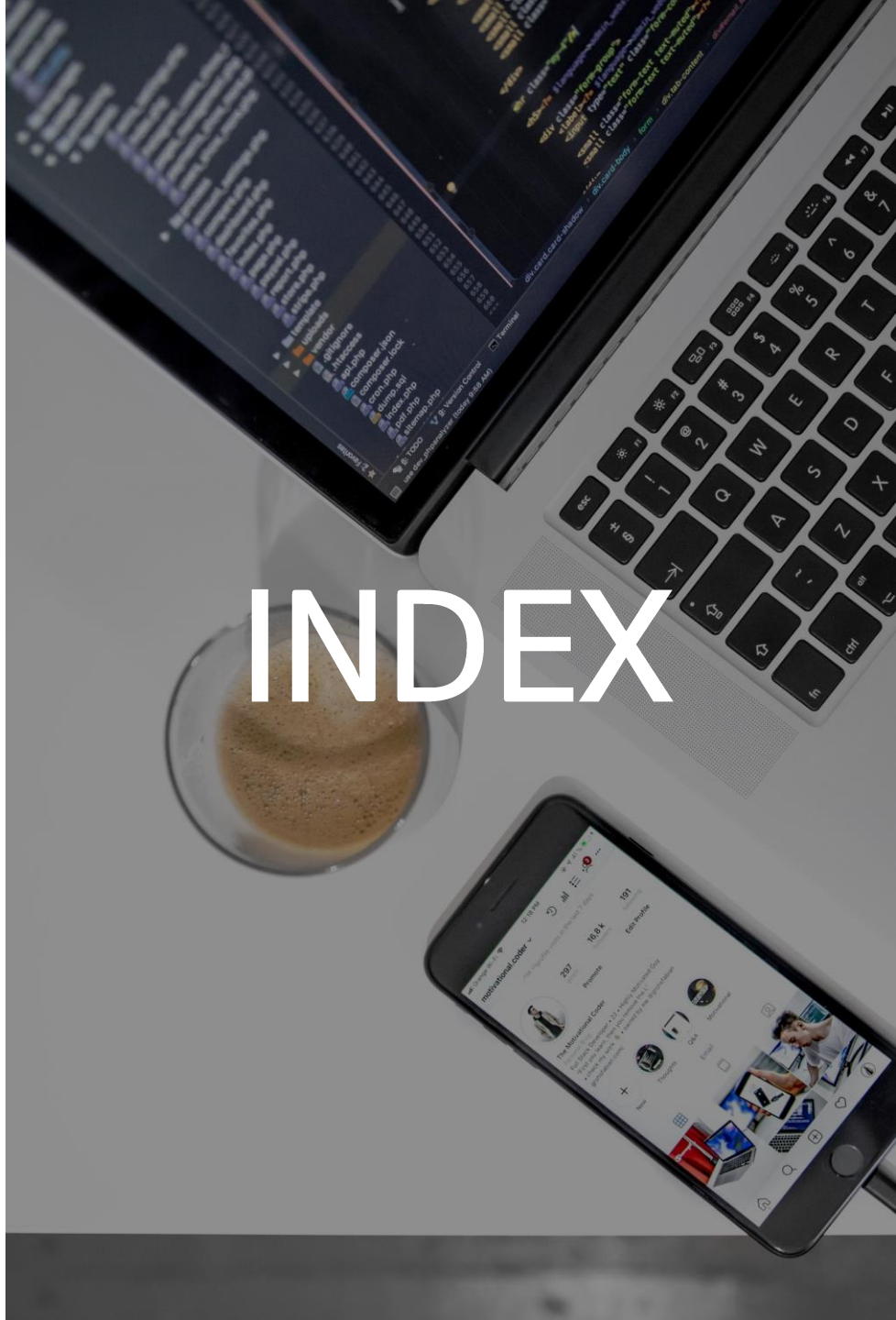
HAI Lecture

Lec 03. Linear Regression

with Pytorch Learning

INDEX



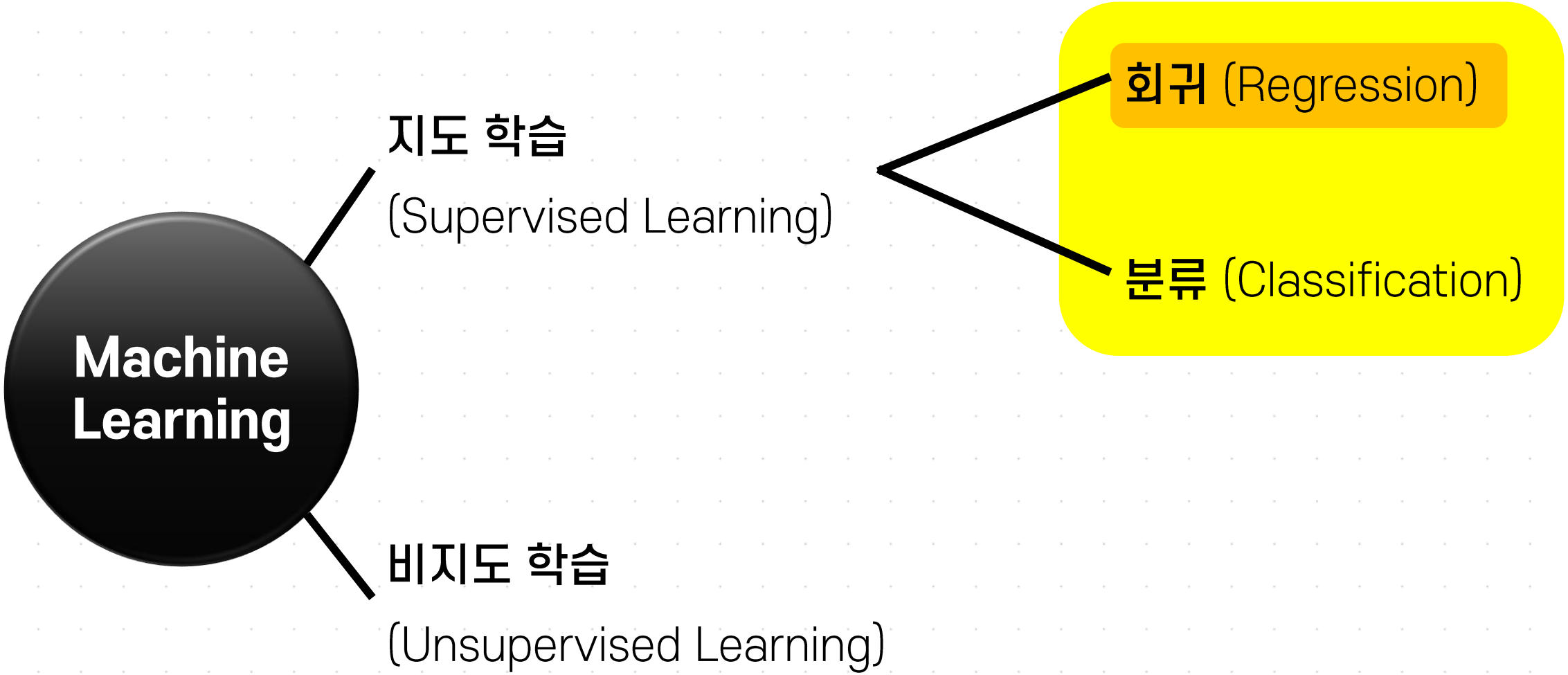
Model : **Model의 개념**

- 가정 = “데이터가 ~~한 패턴을 가지지 않을까?”

- *좋은 모델* 이란?

데이터의 패턴을 **정확**하게 학습한 모델

Linear Regression : **Machine Learning의 분류**



Linear Regression : **Machine Learning의 분류**

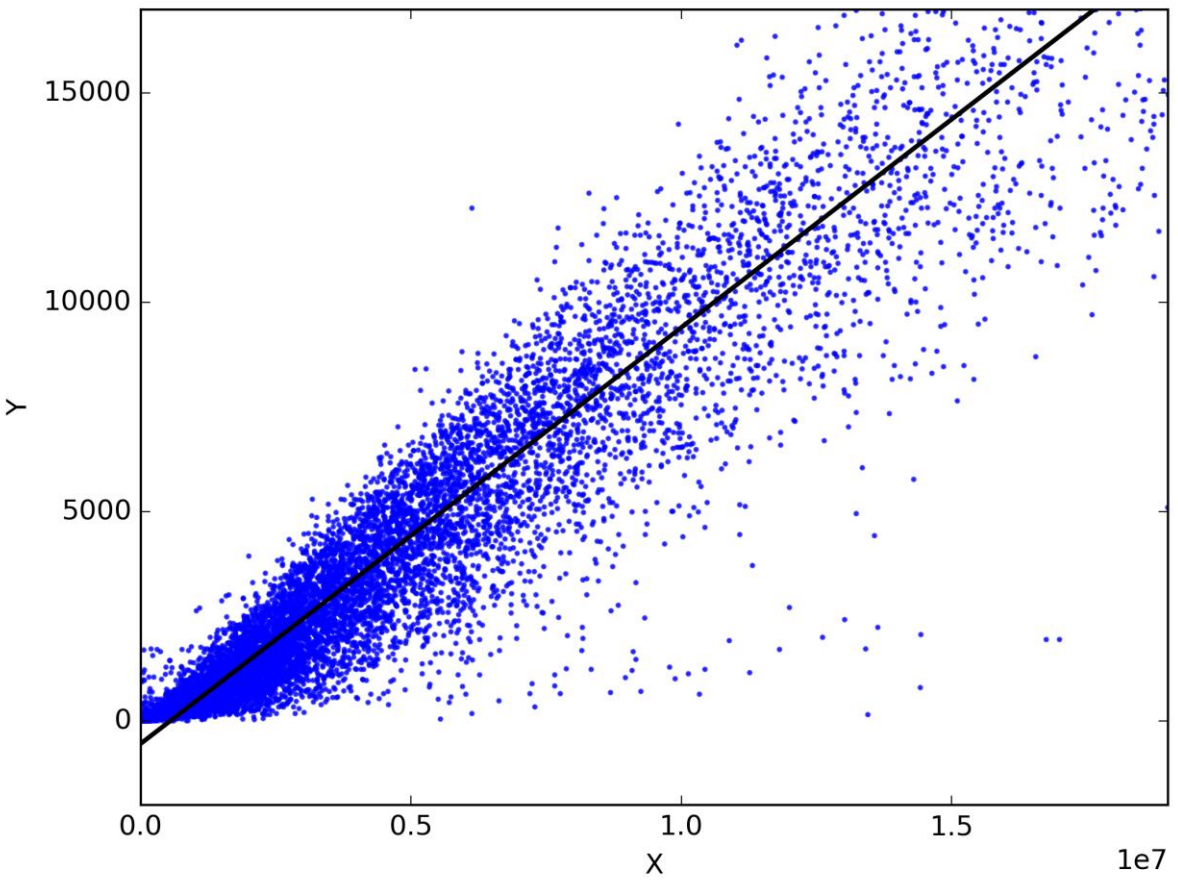
회귀 (Regression)

: 입력에 대해 연속적인 값을 대응시키는 문제

분류 (Classification)

: 입력에 대해 순서가 없는 클래스(라벨)를 대응시키는 문제

Linear Regression : **선형 회귀**



선형 회귀(Linear Regression)

종속 변수 Y와 한 개 이상의 독립 변수 X와의 선형 상관 관계를 모델링하는 선형 회귀 기법

$$e.g.\ y = a_1x_1 + a_2x_2 + a_3x_3 + \ldots$$

$$y = wx + b$$

Linear Regression : **선형 회귀**

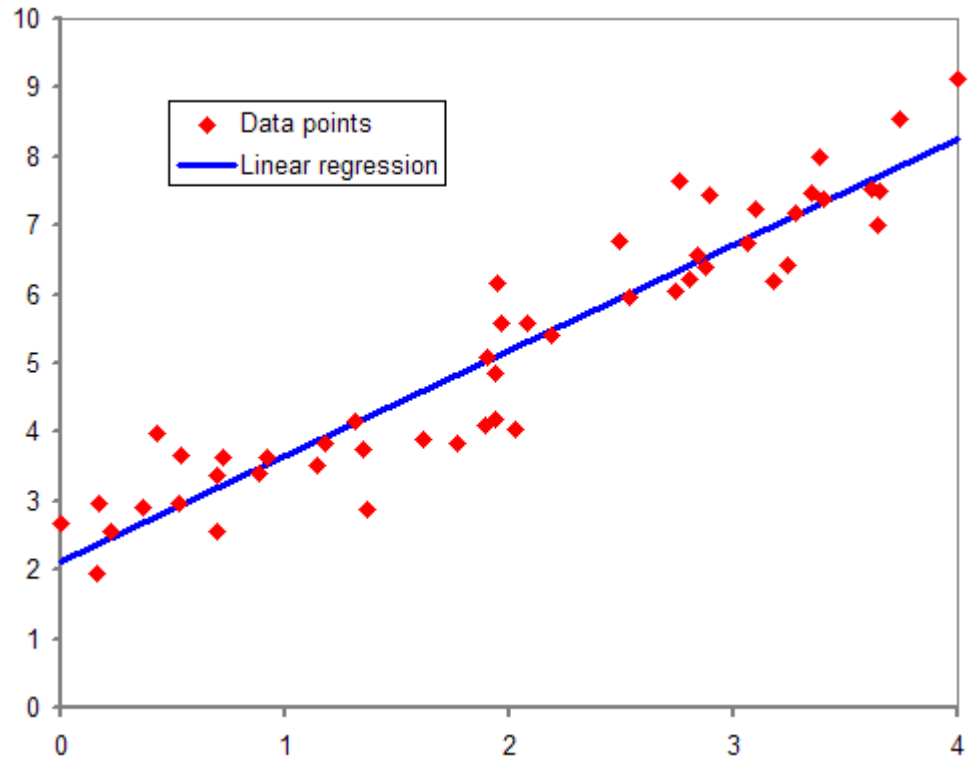
선형 회귀의 분류

1) 단순 선형 회귀

독립 변수 X가 1가지인 경우

2) 다중 선형 회귀

독립 변수 X가 2가지 이상인 경우



Linear Regression : **Hypothesis**

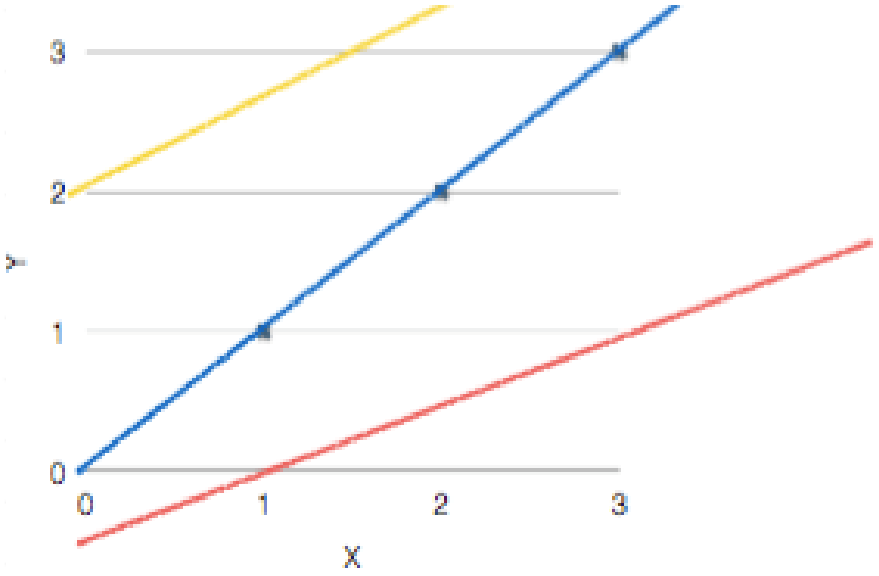
Linear Regression에서 사용하는 방정식을 가리키는 용어

$$y = a_1x_1 + a_2x_2 + a_3x_3 + \ldots$$

$$y = wx + b$$

$$H(x) = Wx + b$$

- W : weight (기울기)
- b : bias (절편)

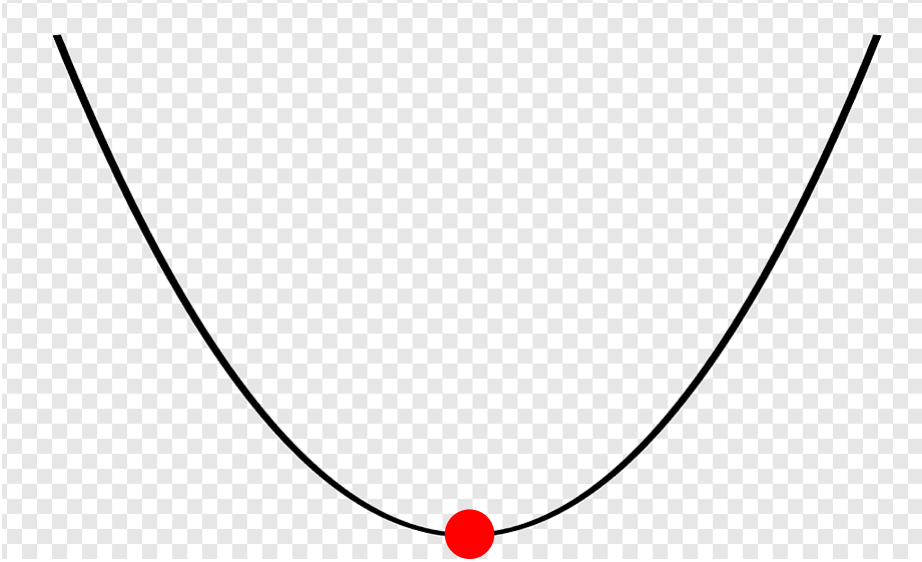


Linear Regression : **Loss Function**

Loss Function

모델을 통해 생성된 값과 실제값의 차이를 나타내는 함수

$$loss(w) = \frac{1}{N}\sum_{n=1}^{N}(\widehat{y_n} - y_n)^2$$



Optimization

seeks to **minimize** the loss function

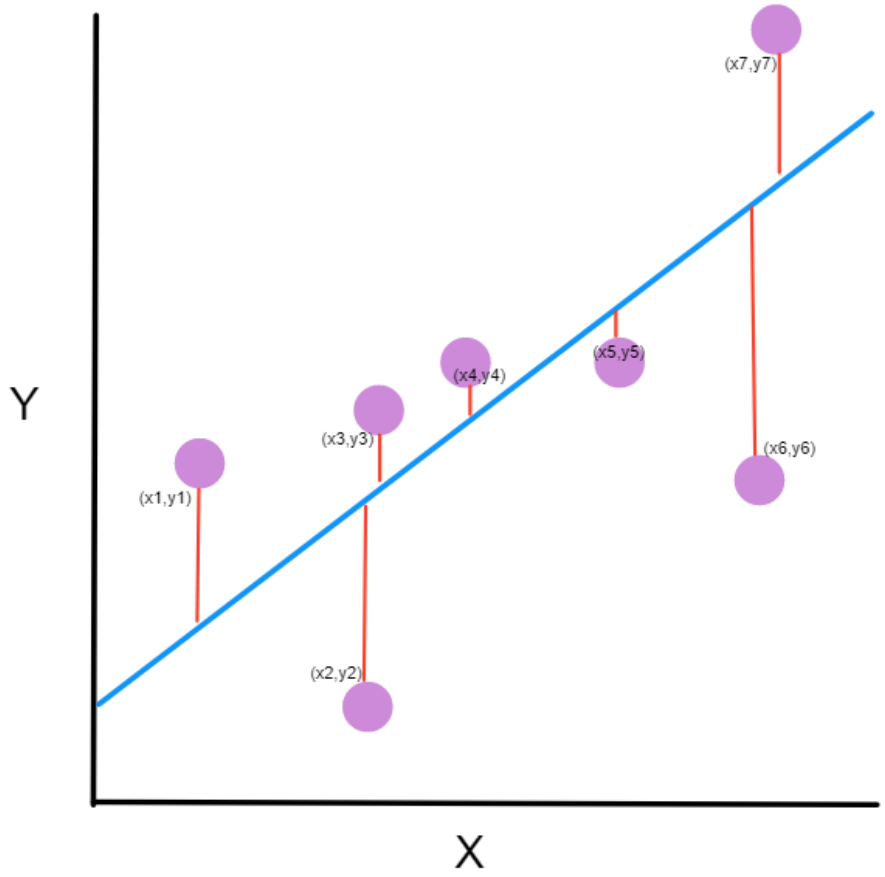
Linear Regression : **Loss Function**

평균 제곱 오차 (Mean Squared Error; MSE)

직선과 데이터 점의 차이의 제곱의 평균

$$loss(w) = \frac{1}{N}\sum_{n=1}^{N}(\widehat{y_n} - y_n)^2$$

```
def mse_line(x, t, w):
    y = w[0] * x + w[1] # y = Wx + b
    mse = np.mean((y - t) ** 2)
    return mse
```



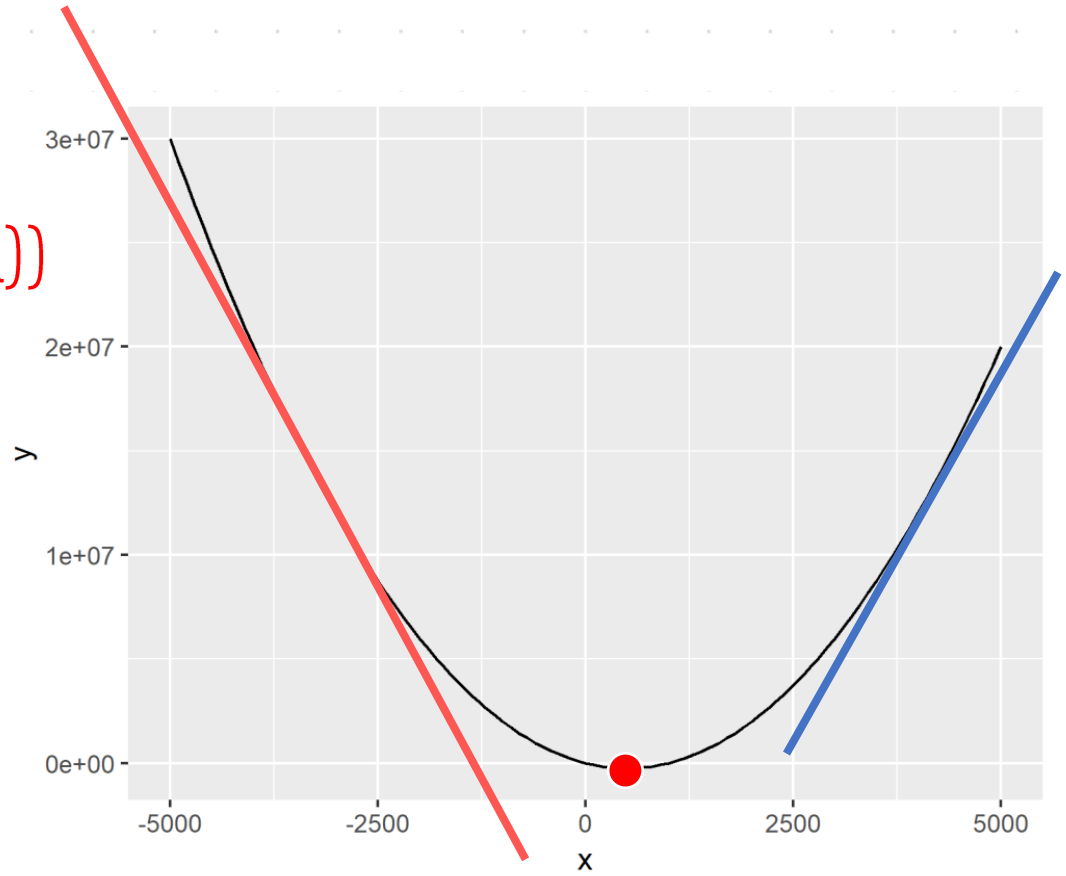
Gradient Descent : **What is our goal?**

GOAL: f(x^*) = min f(x)를 만족하는 x^* 찾기

f'(α) > 0 and x^* > α

→ $x^* = \alpha - \varepsilon \cdot \text{sign}(f'(\alpha))$

where ε > 0



f'(β) > 0 and x^* < β

→ $x^* = \beta - \varepsilon \cdot \text{sign}(f'(\beta))$

where ε > 0

Gradient Descent : **편미분 (Partial Differentiation)**

하나의 변수를 정한 후,

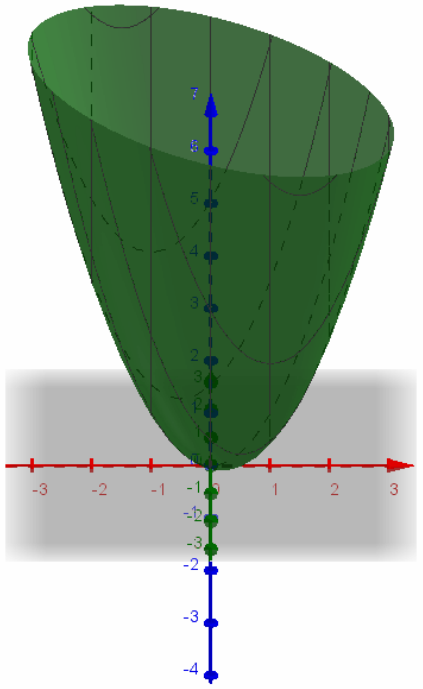
그 변수에만 주목하여 다른 변수는 모두 상수로 간주하여

미분하는 것

Example) $f(x,y) = x^2 + xy + y^2$

$$f'_x(x,y) = \frac{\partial f(x,y)}{\partial x} = 2x + y$$ x에 대한 편미분

$$f'_y(x,y) = \frac{\partial f(x,y)}{\partial y} = x + 2y$$ y에 대한 편미분



Gradient Descent : **Gradient Descent**

$$cost(W) = \frac{1}{m}\sum_{i=1}^{m}\left(W x^{(i)} - y^{(i)}\right)^2$$

$$\nabla W = \frac{\partial\, cost}{\partial W} = \frac{2}{m}\sum_{i=1}^{m}\left(W x^{(i)} - y^{(i)}\right)x^{(i)}$$

$$W := W - \alpha \nabla W$$

Learning rate

Gradient

Tensor with PyTorch : **Getting Started**

pytorch, numpy 라이브러리를 사용하기 위해
아래 부분처럼 해당 라이브러리를 import 해주세요.

```
import torch
import numpy as np
```

Tensor with Pytorch : **What is a Tensor?**

arrays와 matrices와 유사한 특성을 가진 특수화된 자료 구조

in **PyTorch**,

- 모델의 input/output 그리고 그것의 파라미터를 인코드에 사용된다.

- 계산을 가속화하기 위해 GPU 또는 기타 특수 하드웨어 장비 안에서 실행시킬 수 있다.

Tensor with Pytorch : **Tensor Initialization – 1**

i. 데이터로부터 직접 생성

Tensor는 데이터로부터 직접적으로 생성될 수 있다.

이때, 타입은 자동적으로 추론된다.

```
data = [[1, 2],[3, 4]]
x_data = torch.tensor(data)
```

Tensor with PyTorch : **Tensor Initialization – 2**

ii. Numpy 배열로부터 생성

Tensor는 Numpy의 ndarray 객체로부터 생성할 수 있다.

```
np_array = np.array(data)
x_np = torch.from_numpy(np_array)
```

Tensor with Pytorch : **Tensor Initialization – 3**

iii. 다른 Tensor로부터 생성

기존 Tensor를 바탕으로 새로운 Tensor를 만들 수 있다. 사용자로부터 새로운 값을 제공받지 않는 한, 새로운 Tensor는 인자로 들어온 Tensor의 특성들(shape, datatype)을 그대로 유지한다.

```
x_ones = torch.ones_like(x_data) # retains the properties of x_data
print(f"Ones Tensor: \n {x_ones} \n")

x_rand = torch.rand_like(x_data, dtype=torch.float) # overrides the datatype of
x_data
print(f"Random Tensor: \n {x_rand} \n")
```

Out:

```
Ones Tensor:
 tensor([[1, 1],
        [1, 1]])

Random Tensor:
 tensor([[0.6147, 0.7112],
        [0.5486, 0.8505]])
```

Tensor with Pytorch : **Tensor Initialization – 3**

iii. 다른 Tensor로부터 생성

기존 Tensor를 바탕으로 새로운 Tensor를 만들 수 있다. 사용자로부터 새로운 값을 제공받지 않는 한, 새로운 Tensor는 인자로 들어온 Tensor의 특성들(shape, datatype)을 그대로 유지한다.

```
x_ones = torch.ones_like(x_data) # retains the properties of x_data
print(f"Ones Tensor: \n {x_ones} \n")

x_rand = torch.rand_like(x_data, dtype=torch.float) # overrides the datatype of
x_data
print(f"Random Tensor: \n {x_rand} \n")
```

Out:

```
Ones Tensor:
 tensor([[1, 1],
        [1, 1]])

Random Tensor:
 tensor([[0.6147, 0.7112],
        [0.5486, 0.8505]])
```

Tensor with Pytorch : **Attributes of a Tensor**

Tensor attributes

i) shape (object).shape

ii) datatype (object).dtype

iii) stored device (object).device

```
tensor = torch.rand(3,4)

print(f"Shape of tensor: {tensor.shape}")
print(f"Datatype of tensor: {tensor.dtype}")
print(f"Device tensor is stored on: {tensor.device}")
```

Out:

```
Shape of tensor: torch.Size([3, 4])
Datatype of tensor: torch.float32
Device tensor is stored on: cpu
```

Tensor with Pytorch : **Tensor Operations – 1**

다른 device에서 Tensor 사용하기

기본적으로 Tensor는 CPU에 생성됩니다.(by default)

GPU로 Tensor를 명시적으로 옮기기 위해서는,

.to method를 사용합니다.

```
# We move our tensor to the GPU if available
if torch.cuda.is_available():
  tensor = tensor.to('cuda')
```

Tensor with Pytorch : **Tensor Operations – 2**

Numpy 방식의 인덱싱과 슬라이싱

Numpy에서와 유사한 방식의 indexing과 slicing을 사용할 수 있다.

```
tensor = torch.ones(4, 4)
tensor[:,1] = 0
print(tensor)
```

```
tensor([[1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.]])
```

Tensor with Pytorch : **Tensor Operations – 4**

Element-wise Multiplication

※ 행렬 곱셈과 달리, 대응되는 원소끼리 곱하는 것에 주의한다.

i) tensor.mul(tensor)

ii) tensor * tensor

```
# This computes the element-wise product
print(f"tensor.mul(tensor) \n {tensor.mul(tensor)} \n")
# Alternative syntax:
print(f"tensor * tensor \n {tensor * tensor}")
```

Tensor with PyTorch : **Tensor Operations – 5**

Matrix Multiplication(행렬 곱셈)

두 개의 Tensor 사이의 행렬 곱셈을 수행한다.

i) tensor.matmul(tensor.T)

ii) tensor @ tensor.T

```
print(f"tensor.matmul(tensor.T) \n {tensor.matmul(tensor.T)} \n")
# Alternative syntax:
print(f"tensor @ tensor.T \n {tensor @ tensor.T}")
```

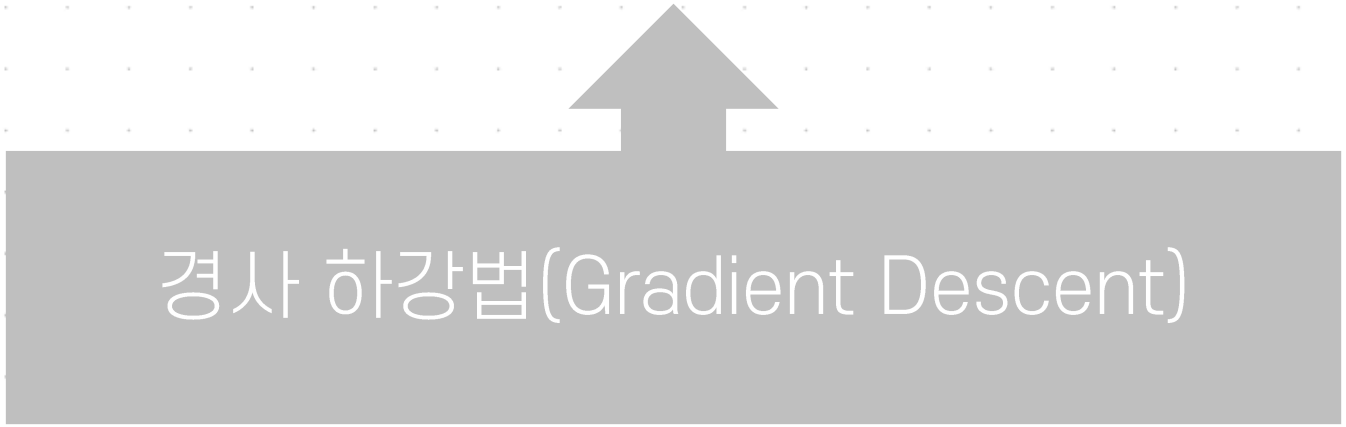
Optimizer : **최적화란 무엇인가?**

최적화(Optimization)

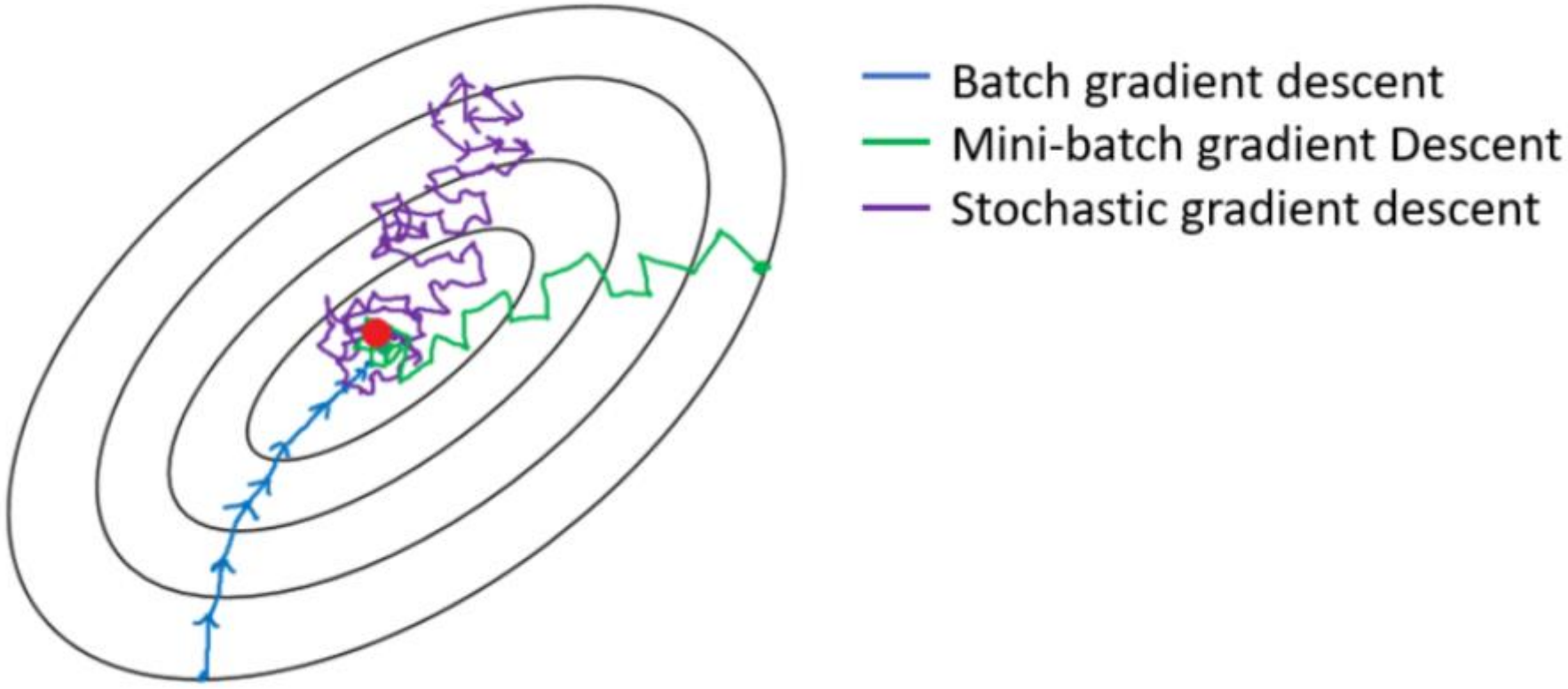
학습의 목적: Loss function의 값을 최대한 낮추는 매개변수를 찾는 것

매개변수 : 가중치(weight) + 편향(bias)

→ 매개변수의 최적값을 찾는 문제로 귀결됨(최적화)



Optimizer : **Gradient Descent Algorithm**



Optimizer : Gradient Descent Algorithm

Batch Gradient Descent(BGD)

전체 training set을 사용하는 것

→ 한 step 당 전체 데이터를 계산하므로 계산량이 많아짐

Stochastic Gradient Descent(SGD)

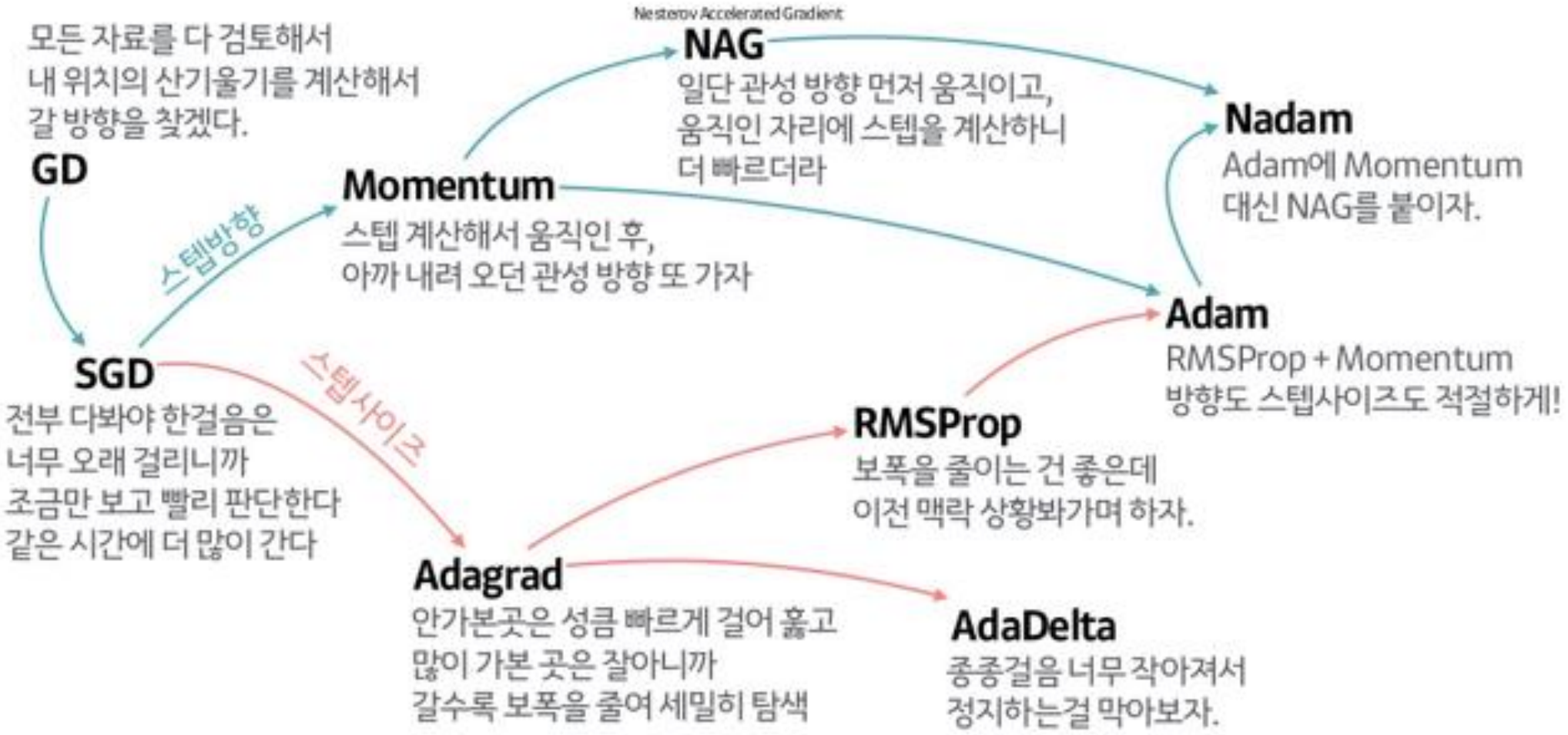
전체 대신 일부의 데이터 모음(mini-batch)에 대해서만 loss 계산

→ BGD에 비해 부정확하나 소요시간이 빠름

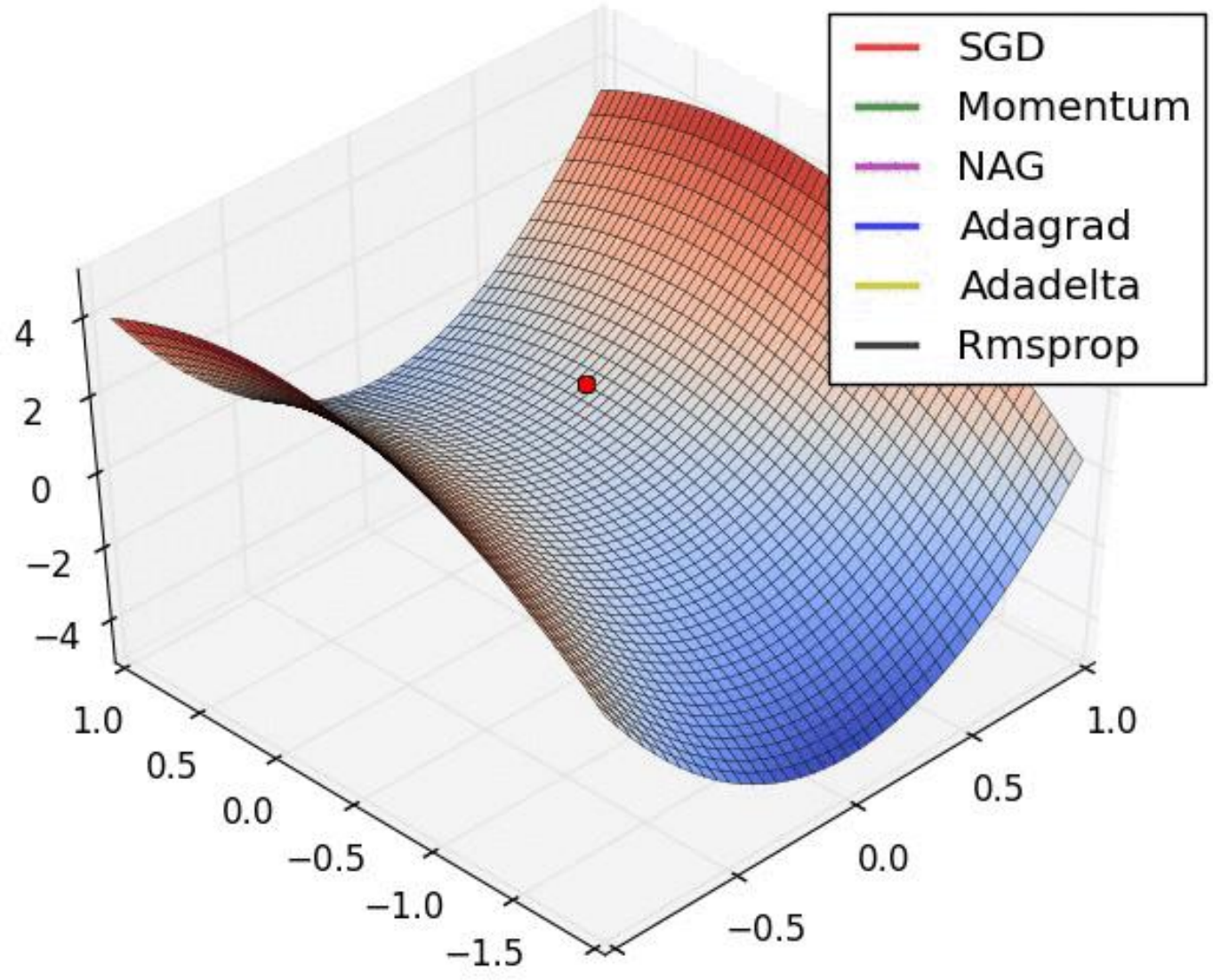
BGD에서는 local minimum에 빠지는 문제가 존재하나,

SGD에서는 더 좋은 방향으로 수렴할 가능성 존재

Optimizer : **Gradient Descent Algorithm**



Optimizer : **Gradient Descent Algorithm**



Optimizer : G.D. with PyTorch

in PyTorch: torch.optim

- 시작할 때 Optimizer 정의
- optimizer.zero_grad( ) : gradient 초기화(0으로 초기화)
- cost.backward( ) : gradient 계산
- optimizer.step( ) : gradient descent 수행

```
# optimizer 설정
optimizer = optim.SGD([W], lr=0.15)

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

오늘의 실습

: PyTorch를 이용하여

Linear Regression을 구현해봅시다


Thank You