Introduction to Deep Learning

2025 Summer DL Session 2주차

KOREA
UNIVERSITY

목차

01. 들어가기 전에,,

< 1주차 과제 우수자 >

22기 이세훈
22기 이은서

1주차 때 말씀드린것과 같이 세션 전반부는 과제 우수자의 발표가 있을 예정입니다.
매주 1~3명이 선정되오니, 세션리더와 함께 열심히 달려봅시다 :)

01. 들어가기 전에,,

< 1주차 과제 요약>

- **AI**를 활용해 어떤 문제를 해결하고 싶은지 생각해보기
- 원하는 출력을 얻기 위해 모델에 어떤 입력을 줘야 하는지 알아보기

Understand the difference between deep learning and machine learning,
think about the application possibilities of the 7 common patterns of AI

01. 들어가기 전에,,

학습 목표

1주차 과제의 의도

AI의 대표적인 7가지 패턴을 살펴봄으로써, 첫 시간에 배운 딥러닝의 개념
(머신러닝과의 차이, Loss, 최적화 등)이 얼마나 다양한 방식으로 활용되는지 이해하고,
자신이 왜 딥러닝을 공부해야 하는지에 대한 동기 부여를 제공하기 위함.

2주차 학습 목표

가장 간단하고 클래식한 모델인 linear regression과 softmax regression
(linear neural netwaork)을 통해 loss의 정의와 최소화 과정을 파악하자

02
Mathmetical Review

02. Mathmetical Review

01 행렬의 개념

- 행렬**(matrix)**은 벡터를 원소로 가지는 **2**차원 배열입니다**.**
- 행렬은 행**(row)**과 열**(column)**이라는 인덱스**(index)**를 가집니다**.**
- 파이썬에선 **numpy.array** 로 구현합니다**.**

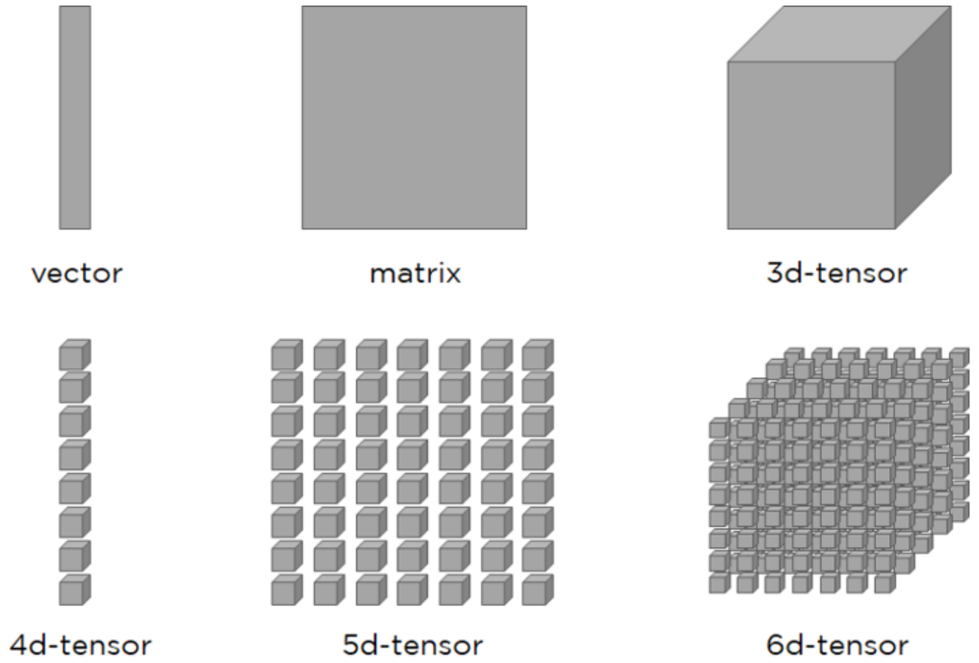
$$\mathbf{X} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

```
1 X = np.array([[1, 2, -1],
2                [-1, 0, 1],
3                [2, 1, 0]])
```

02. Mathmetical Review

01 텐서의 개념

- **N**차원 텐서**(tensor)**는 **N-1**차원 텐서를 원소로 가지는 배열입니다.
- **N**차원 텐서는 인덱스의 개수가 **N**개가 됩니다.
- 형태는 **numpy**의 **array**와 동일하지만, **GPU**나 다른 하드웨어 가속기에서 실행할 수 있다는 점에서 다릅니다.



02. Mathmetical Review

01 행렬/텐서곱

- 행렬곱의 직관적.시각적인 이해를 위해 아래 유튜브 영상을 추천드립니다.
  (제3장: 선형변환과 행렬 | 선형대수학의 본질, 3Blue1Brown)

- 행렬 곱셈(**matrix multiplication**)은 번째 행벡터와 번째 열벡터 사이의 내적을 성분으로 가지는 행렬을 계산합니다.

```
1 X = torch.tensor([[0, 1],
2                   [2, 3]])
```

```
1 Y = torch.tensor([[0, -1],
2                   [-2, -3]])
```

```
1 print(torch.matmul(X, Y)) # matrix
```

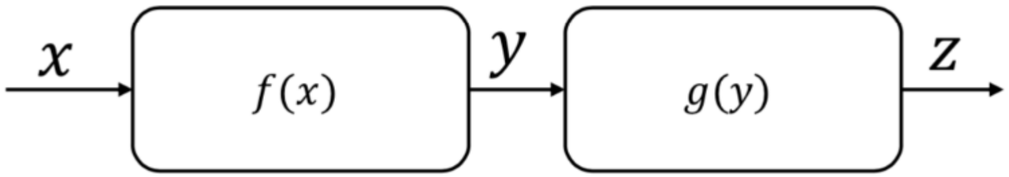
```
tensor([[ -2,  -3],
        [ -6, -11]])
```

```
1 print(X * Y) # element-wise
```

```
tensor([[ 0, -1],
        [-4, -9]])
```

02. Mathmetical Review

02 편미분, & 연쇄법칙



$$y = f(x) \qquad z = g(y)$$

$$\frac{dy}{dx} = \frac{df(x)}{dx} \qquad \frac{dz}{dy} = \frac{dg(y)}{dy}$$

연쇄 법칙 (Chain Rule)

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx} = \frac{dg(y)}{dy}\frac{df(x)}{dx}$$

합성 함수의 미분 (겉미분과 속미분)

$$\frac{dz}{dx} = \frac{dg\big(f(x)\big)}{dx} = g'\big(f(x)\big)f'(x)$$

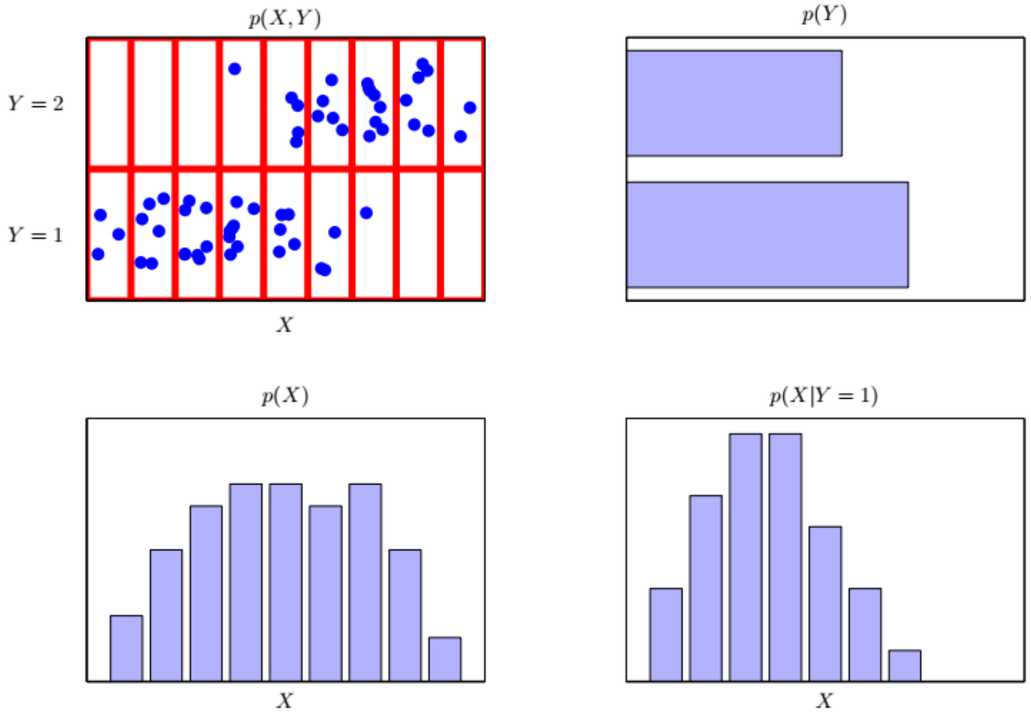
02. Mathmetical Review

03 확률론

- 기계학습에서 사용하는 손실함수는 데이터 분포와 모델 예측 분포의 차이, 즉 예측이 틀릴 위험**(risk)**를 최소화하는 방향으로 학습하도록 유도합니다.

- 조건부 확률: $P(y|x)$
입력변수 **x**로부터 추출된 패턴에 대한 **y**의 확률 분포로, 실제 분포와 가장 가까운 $P(y|\emptyset(x))$의 파라미터 θ를 찾는 것이 분류 문제의 목표입니다.

- 조건부 기댓값: $E_{y\sim p(y|x)}[y|x]$
주어진 **x**에 대한 **y**의 기댓값으로, **L2-norm**을 최소화하는$f_\theta(x)$와 동일한 회귀 문제의 목표입니다.



03
MLP & Loss

03. MLP

The structure of DNN (unformal)

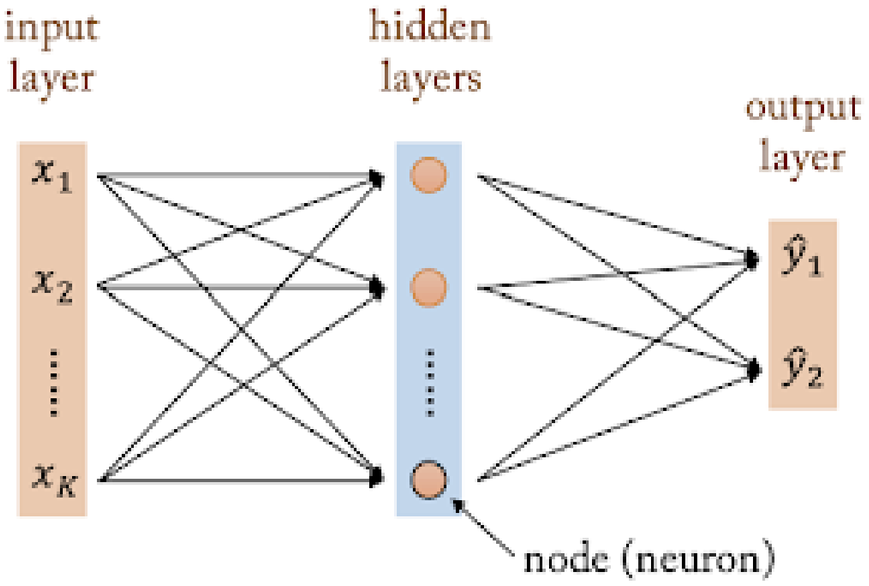
Given the sampled data, derive a predicted value,
And compare it with the ground truth value.

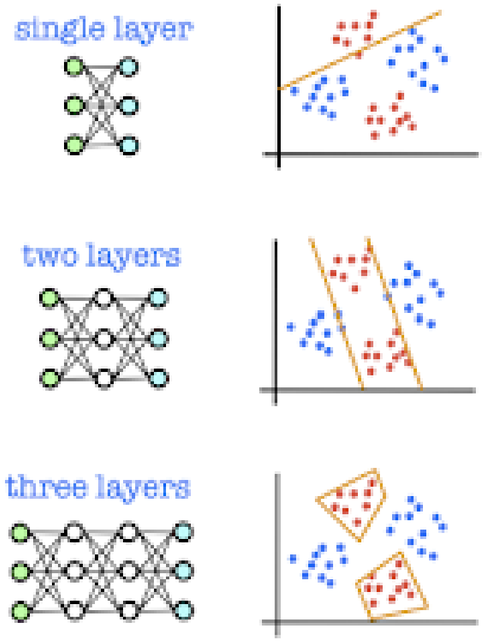
$$\begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_d \end{bmatrix} \rightarrow \mathbf{x}_i \rightarrow \text{Embedding} \rightarrow \text{Layers} \rightarrow \text{Output} \rightarrow \text{ReadOut} \rightarrow \hat{\mathbf{y}}_n \overset{compare}{\Leftrightarrow} \mathbf{y}_n$$

- embedding : 사람이 쓰는 자연어 -> 기계가 이해할 수 있는 벡터로 바꾸는 과정 및 결과

03. MLP

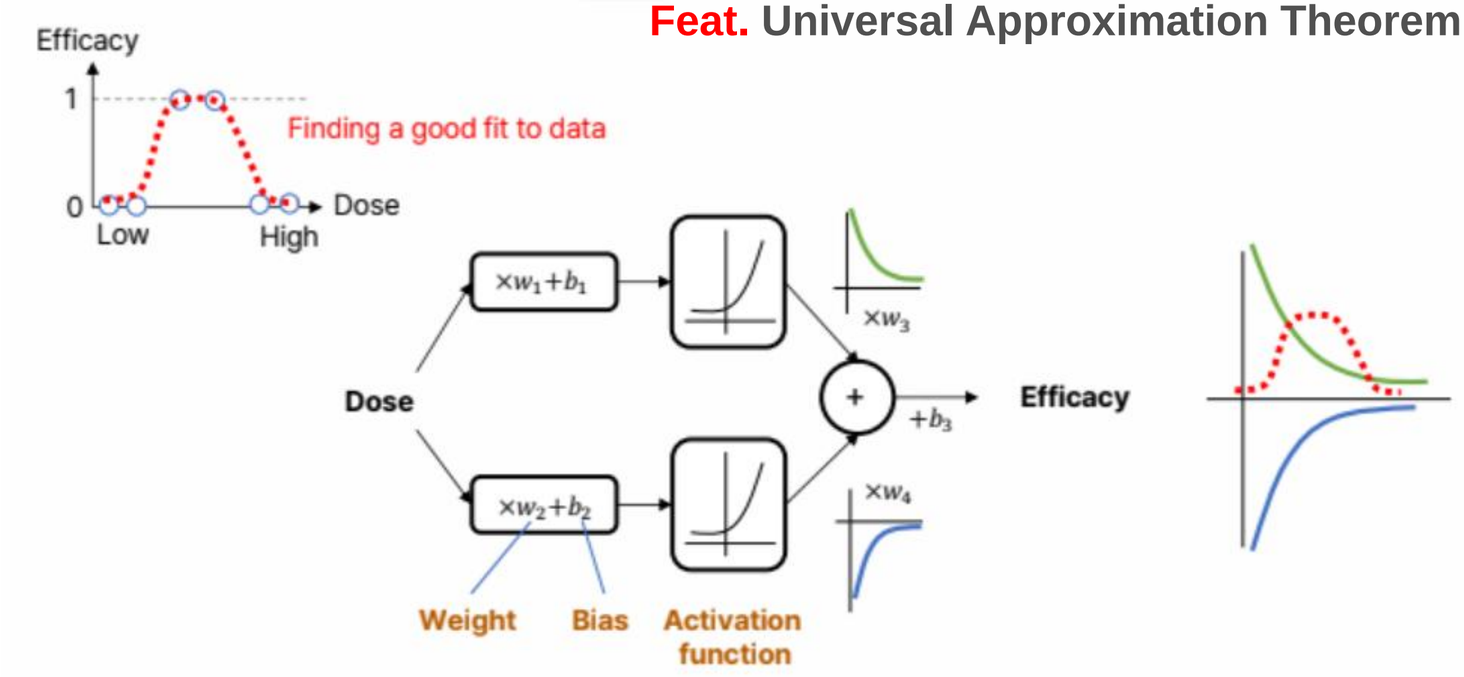
Multilayer Perceptron(MLP) : feedforward neural network의 class





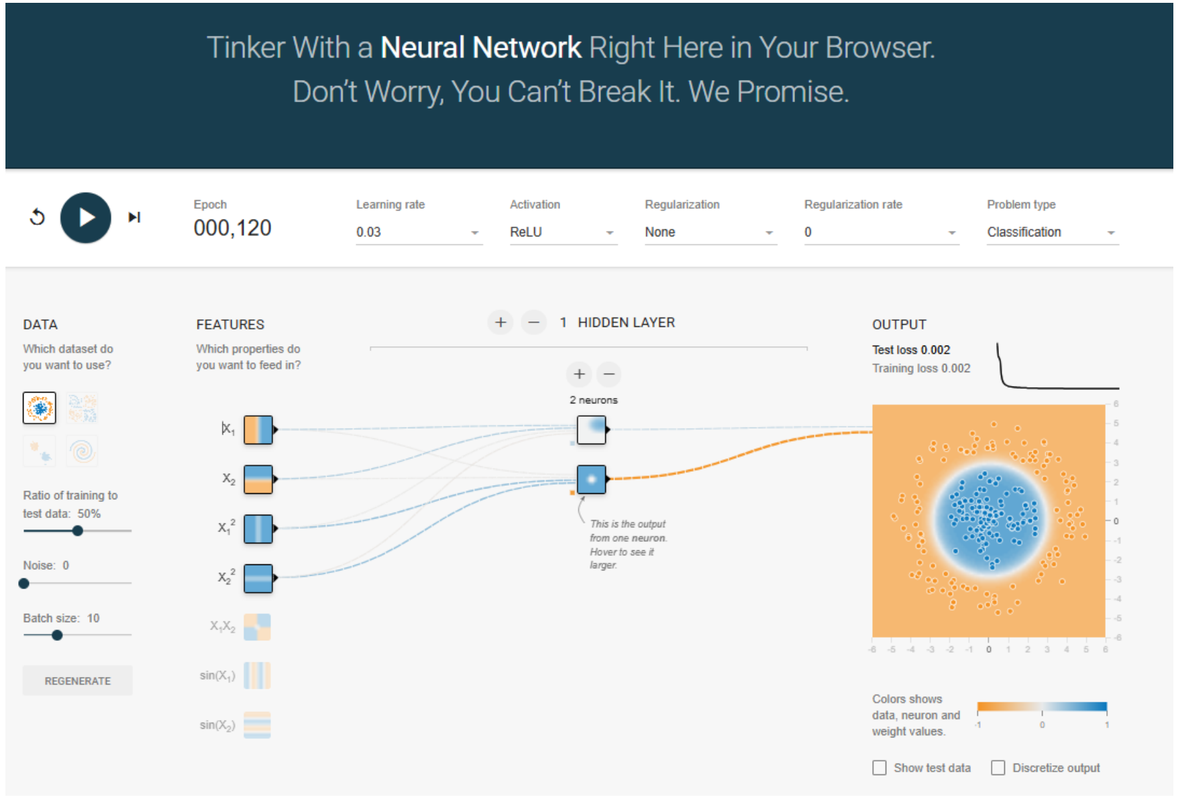
왜 여러 개의 레이어(Multi-Layer)가 필요할까 ?

Feat. Universal Approximation Theorem



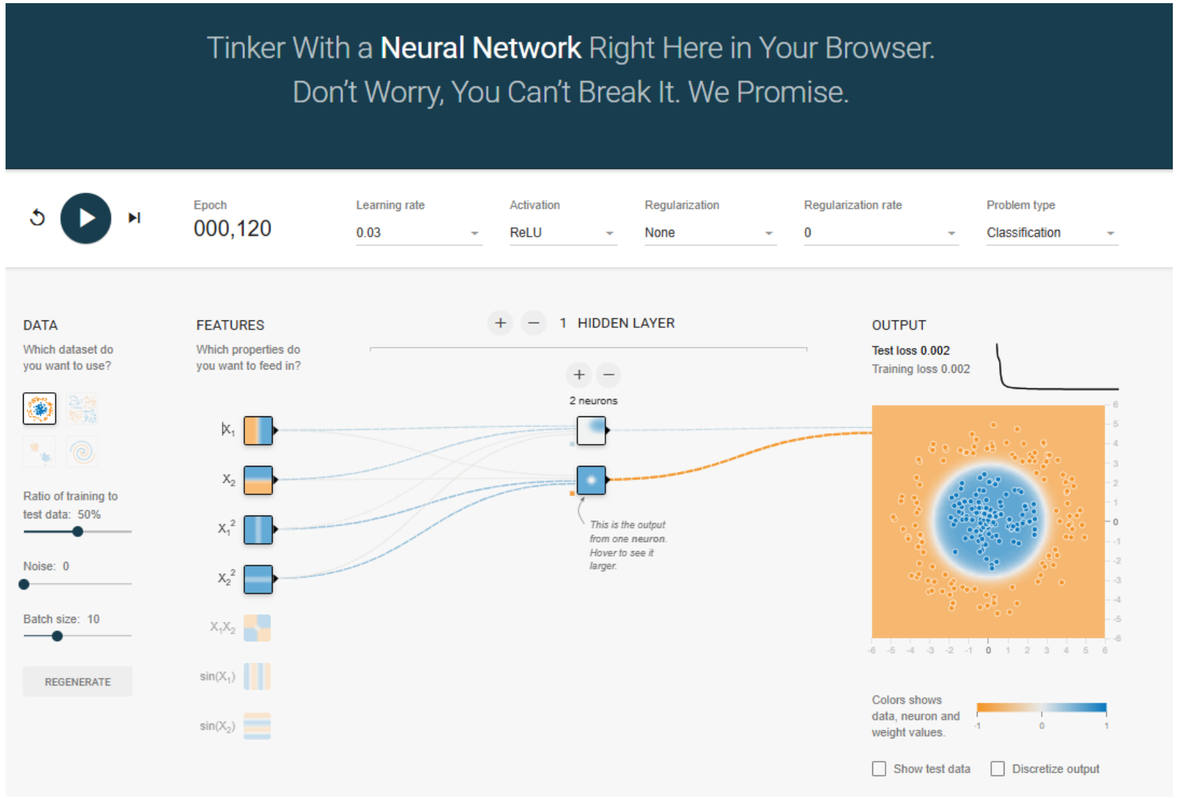
[The Essential Main Ideas of Neural Networks, StatQuest with Josh Starmer]

03. MLP



A neural network playground

03. MLP

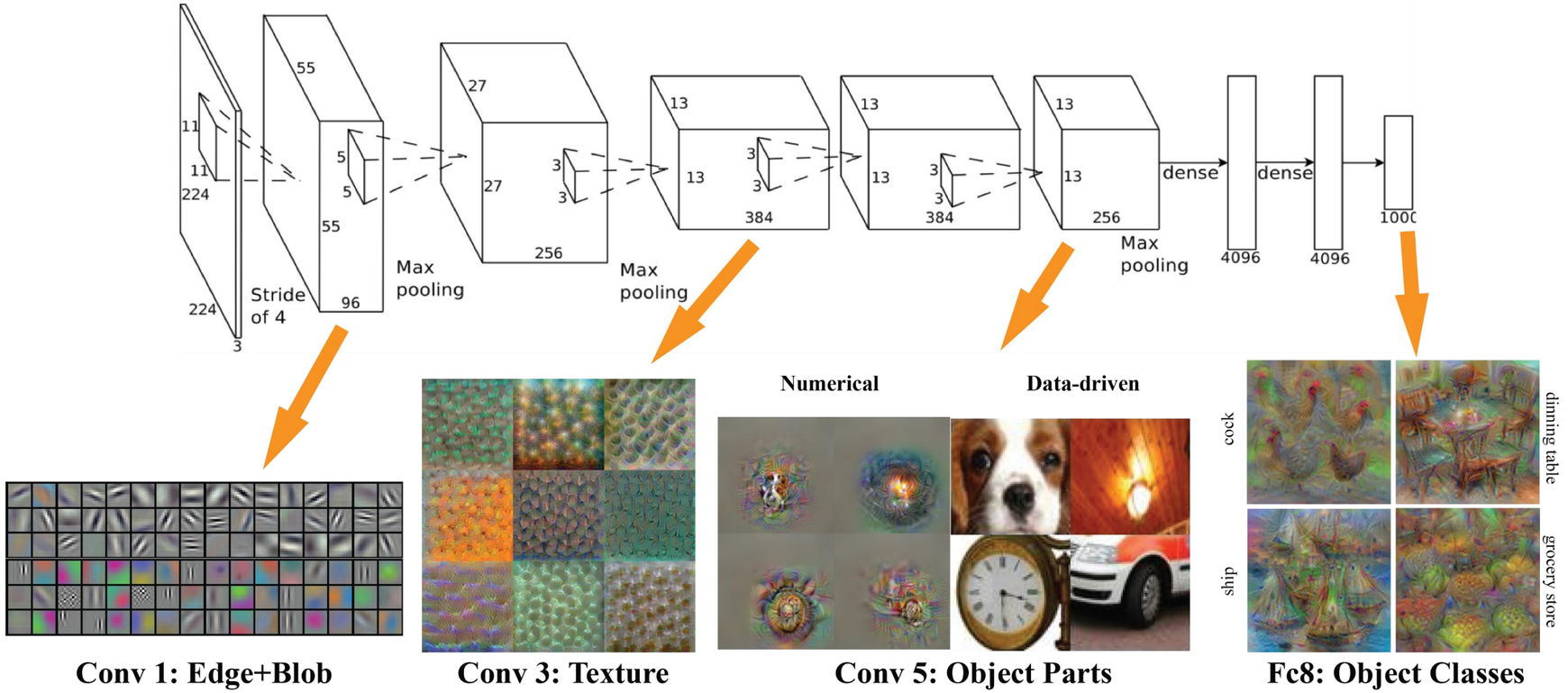


A neural network playground

첫번째 과제!

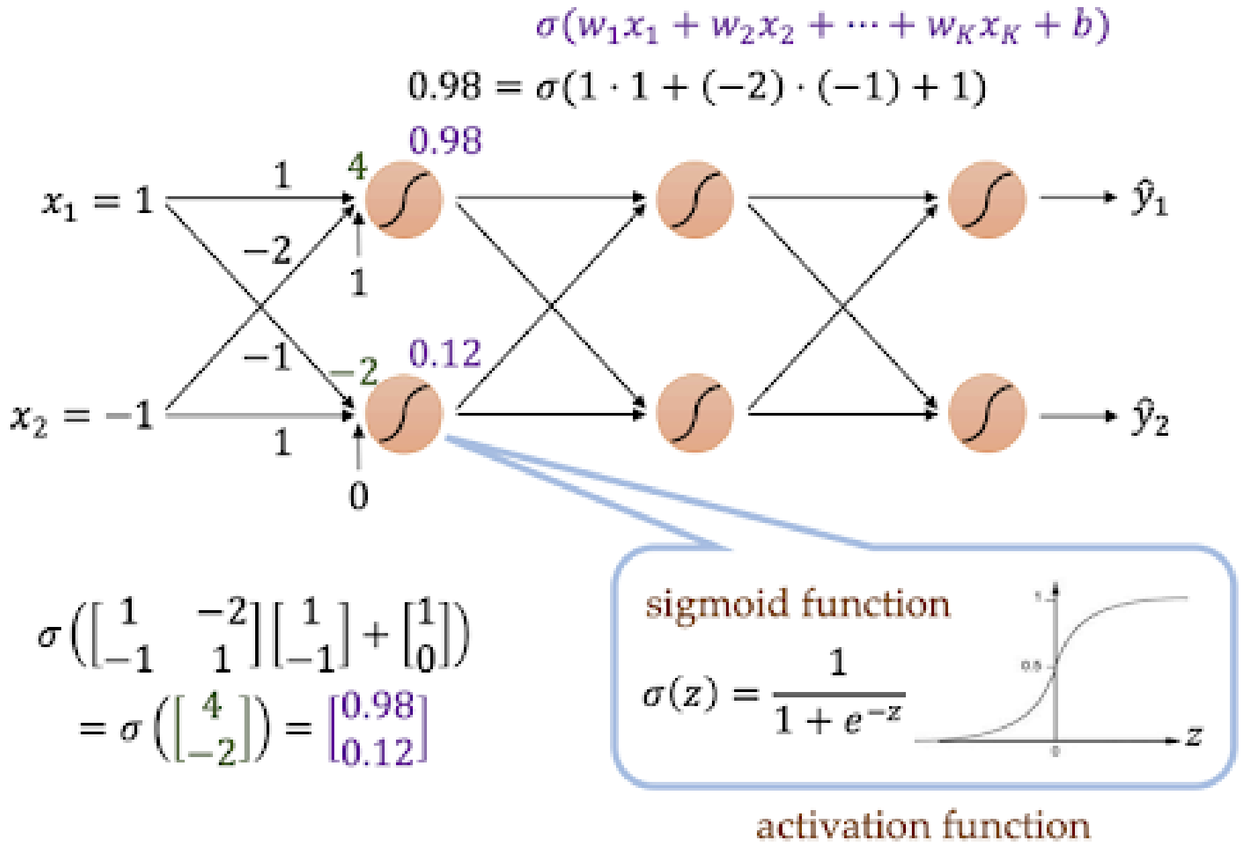
1. 나선 or 4분면 DATA 선택
2. 1000 epoch 이내로 test loss가 0.005 이하로 떨어지게끔 hidden Layer 및 뉴런 설계
   - Feature, activation, feature간의 가중합 고려
   - hyperparameter 바꿔보기
3. 스크린샷 후 두번째 과제와 함께 '2주차 과제_19기_박세훈.zip' 형식으로 깃허브 제출

How about CNN?



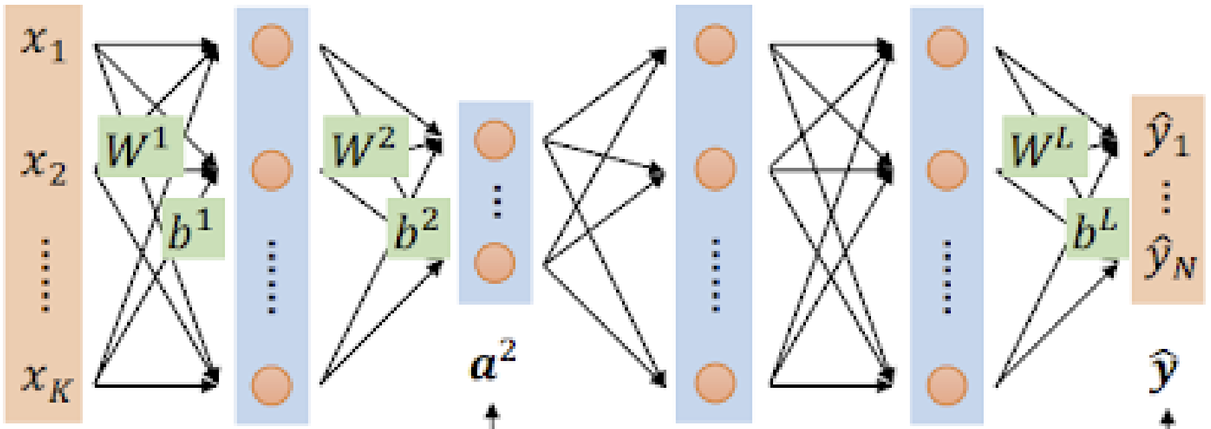
03. MLP

- [엑셀로 **MLP** 손수 계산하기]



input layer를 제외하면, 각 node는 non-linear activation function 사용
linear activation -> 단순 선형변환 연산일뿐!

03. MLP

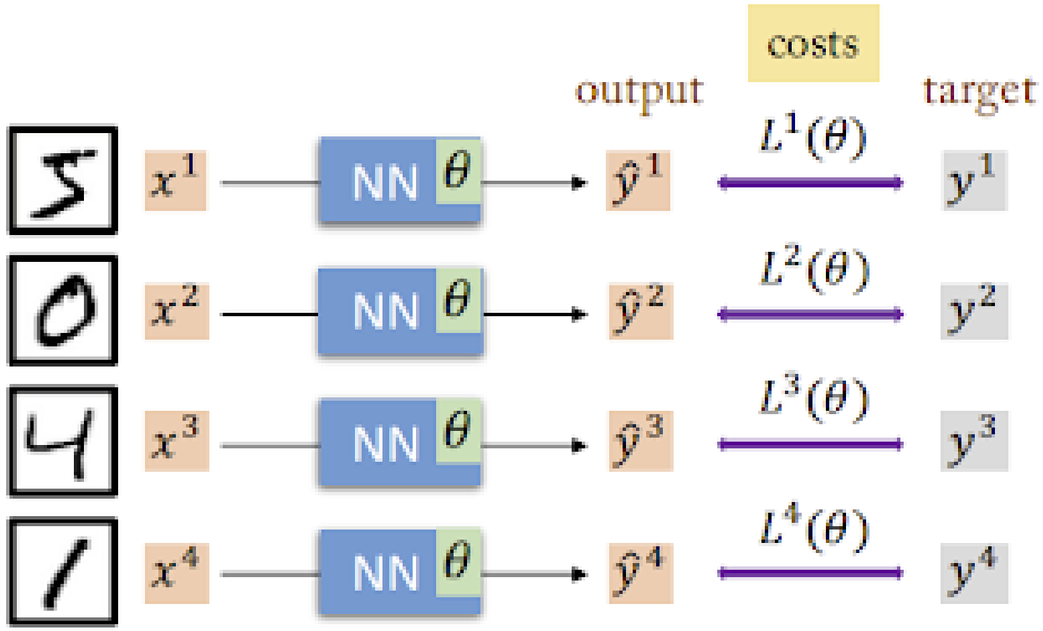


$$\hat{y} = f(x) = \sigma(W^L \cdots \sigma(W^2 \sigma(W^1 x + b^1) + b^2) \cdots + b^L)$$

input layer를 제외하면, 각 node는 non-linear activation function 사용
linear activation -> 단순 선형변환 연산일뿐!

03. Loss

Loss : 연산된 결과인 y_hat과 g.t y가 **얼마나 차이** 나는지에 대한 지표

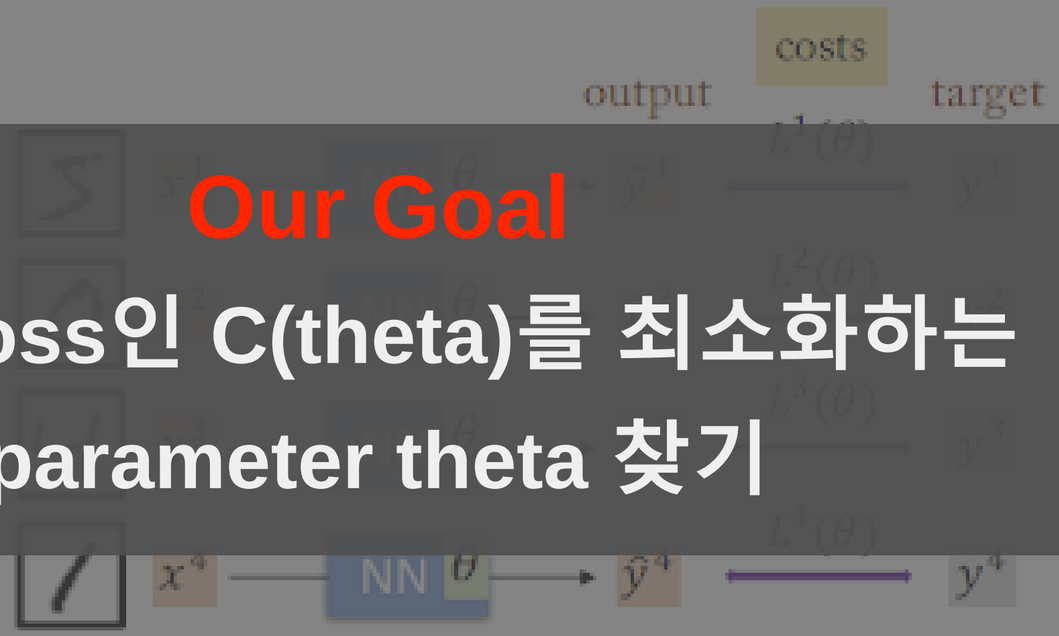


Total Cost

$$C(\theta) = \sum_{n=1}^{N} L^n(\theta)$$

given parameter theta, 각 data sample은 loss function에 의해 L(theta)가 주어짐
total loss = sum(train data에 대한 loss) : 현재 parameter theta가 얼마나 좋지 않은지에 대한 지표

03. Loss



Our Goal

전체 Loss인 C(theta)를 최소화하는 parameter theta 찾기

Total Cost

$$C(\theta) = \sum_{n=1}^{N} L^n(\theta)$$

03. Loss

01) Continuous (Linear Regression)

w와 b 를 추정(inference) or 학습(learning)하는 것이
통계학과 기계학습(ML)의 공통의 목표

$$\underset{\text{label}}{y} = \underset{\text{weight}}{w}\underset{\text{feature}}{x} + \underset{\text{bias}}{b} + \underset{\text{noise}}{\epsilon}$$

Assumptions

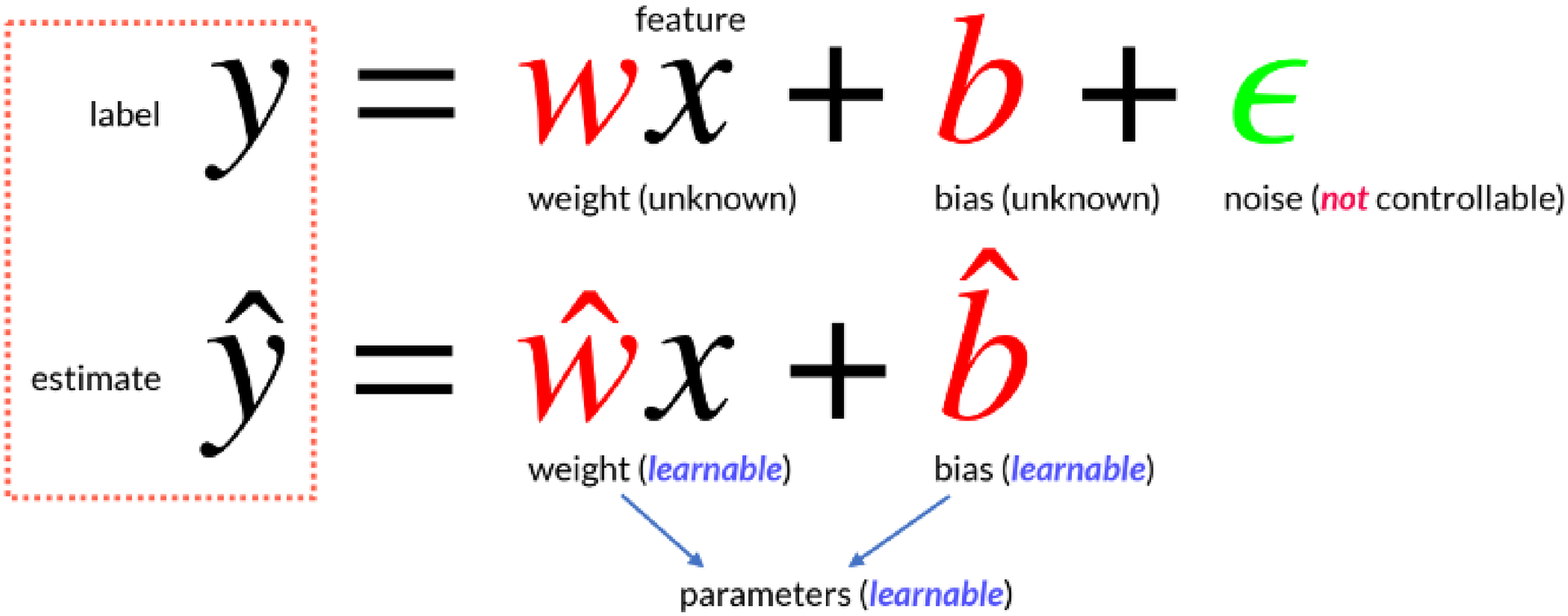
1. The relationship between the independent variables x and the dependent variable y is linear
2. We assume that any noise is well-behaved (following a Gaussian distribution)

03. Loss

01) Continuous (Linear Regression)

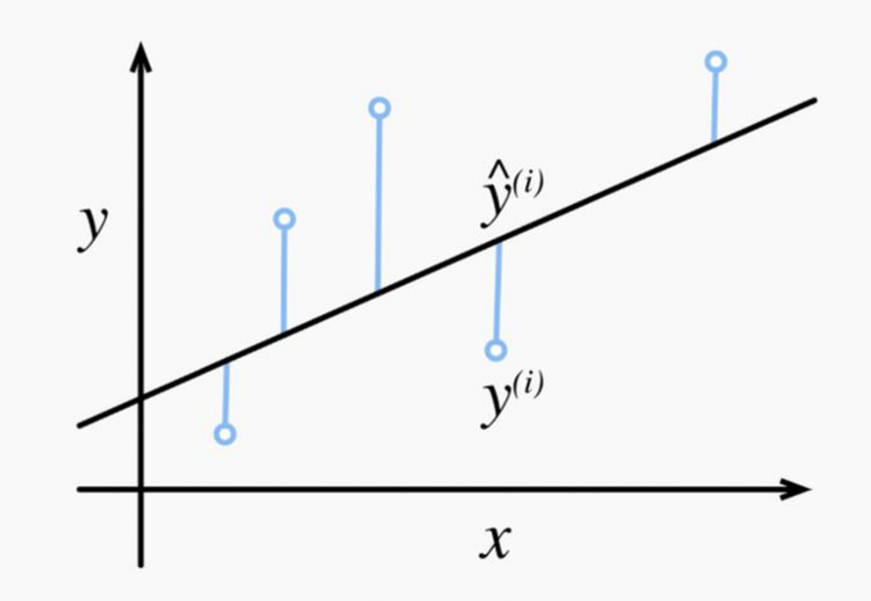
Ground Truth 를 모르는 상황에서
parameter 를 추정하는 것이 목표 !

Goal : minimize L2 distance



03. Loss

01) Continuous (Linear Regression)



Loss function (손실함수)

실제값과 예측값의 거리를 수치화 한 것
loss의 크기가 작을수록, 해당 모델은 실제값과 유사한 예측값을 출력한다. (= 모델의 성능이 좋다.)

=> Loss function을 최소화할 수 있는 최적의 w와 b를 찾자!

03. Loss

01) Continuous (Linear Regression)

Training data: $\{(\boldsymbol{x}^1, y^1), \ldots, (\boldsymbol{x}^N, y^N)\}$ where $\boldsymbol{x}^n = (x_1^n, \ldots, x_K^n)^T$

Hypothesis: $h_\theta(\boldsymbol{x}) = w_1 x_1 + \cdots + w_K x_K + b \cdot 1 = \boldsymbol{w}^T \boldsymbol{x}$

considered as $\theta = \boldsymbol{w} = (w_1, \ldots, w_K, b)^T$ and $\boldsymbol{x} = (x_1, \ldots, x_K, 1)^T$.

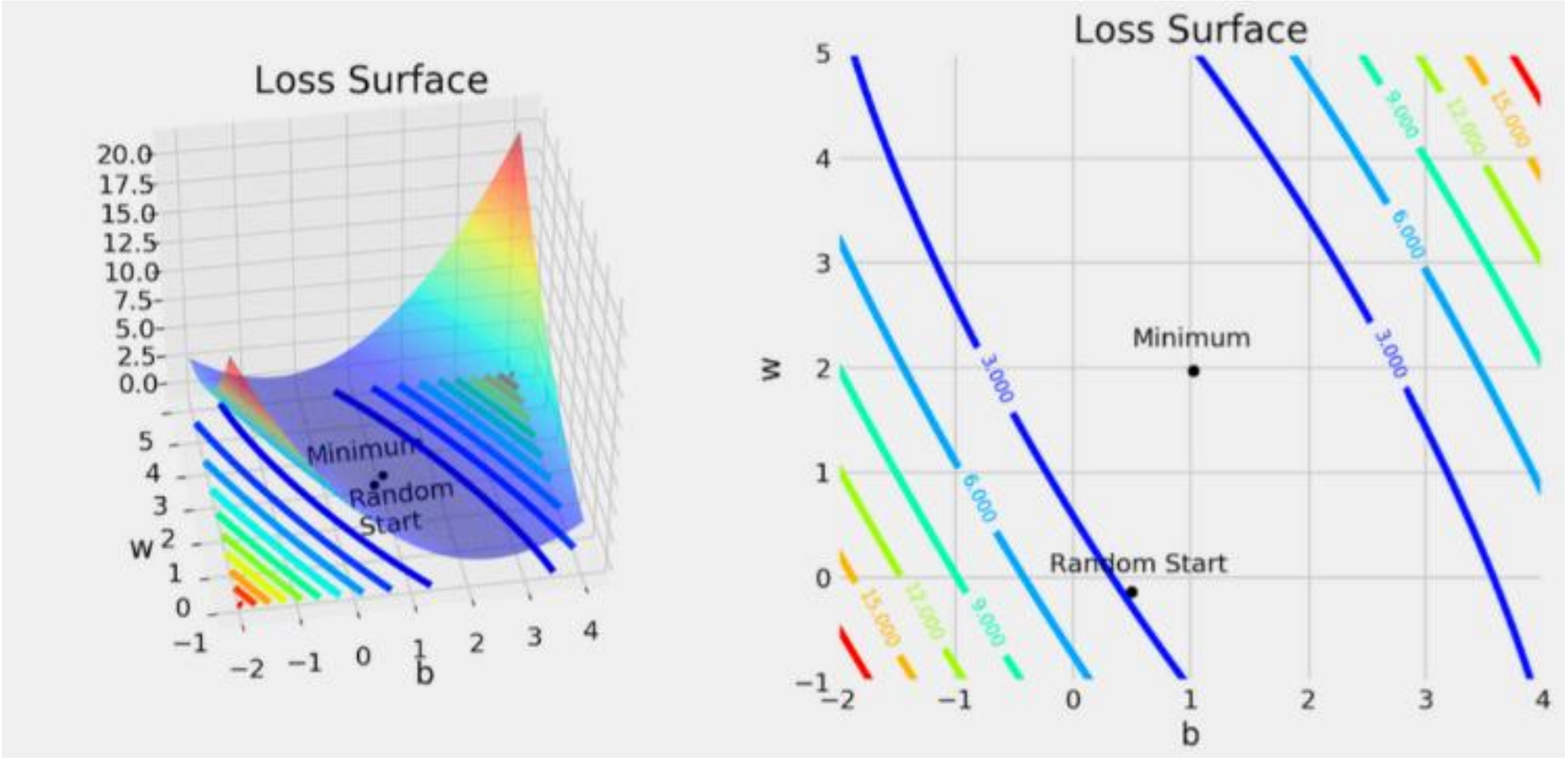
Cost: $C(\theta) = \frac{1}{2N} \sum_{n=1}^{N} (\boldsymbol{w}^T \boldsymbol{x}^n - y^n)^2$ $\Leftarrow$ MSE (mean square error)

회귀 분석에서는 loss 함수로
mean-squared error를 종종 사용
이외, MAE, RMSE, MAPE도 존재함.

$$\mathrm{MSE}(\hat{w}, \hat{b}) = \|\mathbf{y} - \hat{\mathbf{y}}\|_2 = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

03. Loss

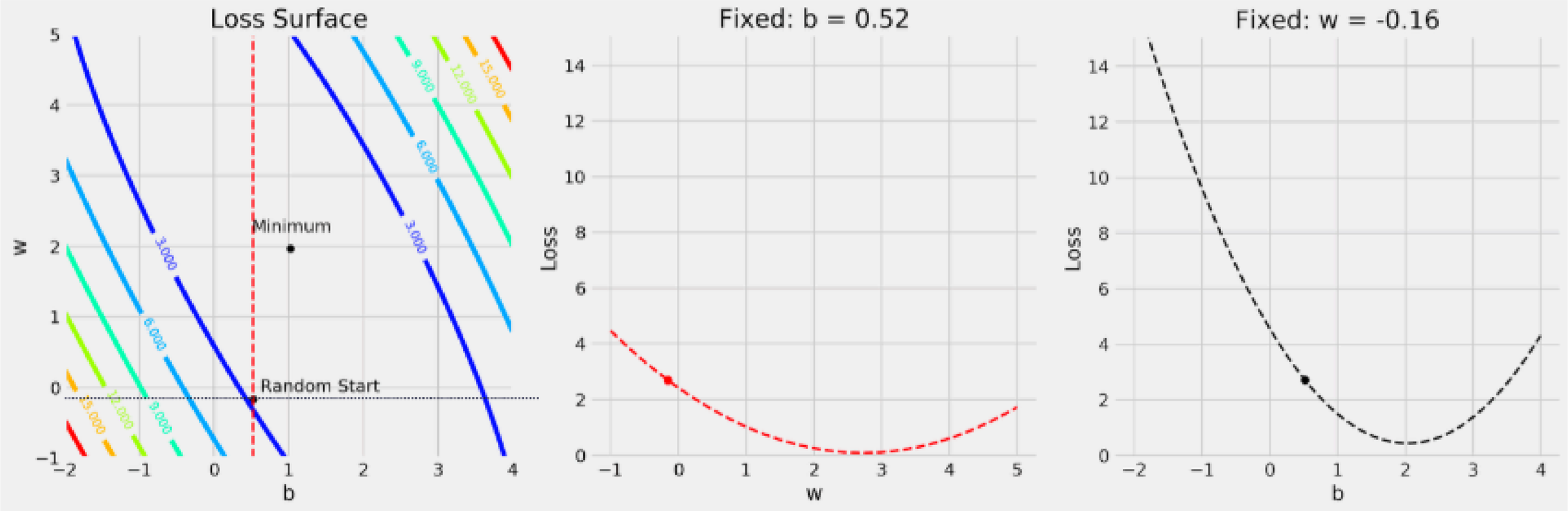
01) Continuous (Linear Regression)



03. Loss

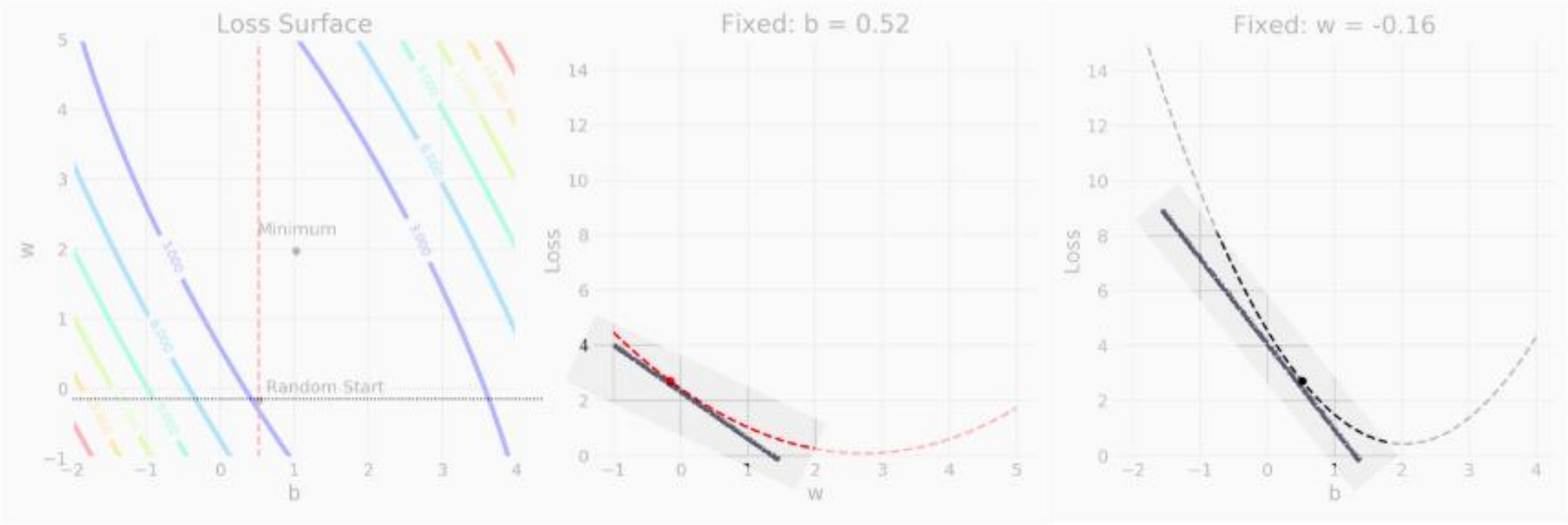
01) Continuous (Linear Regression)

Let us cut the loss surface!



03. Loss

01) Continuous (Linear Regression)



03. Loss

01) Continuous (Linear Regression)

Gradient descent

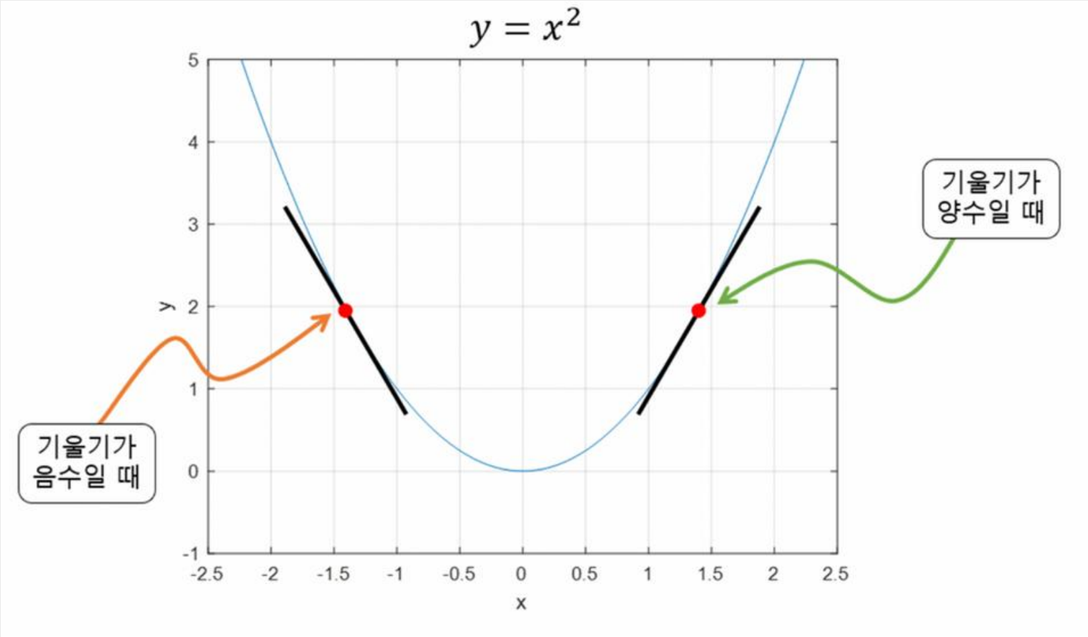
손실 함수 값이 낮아지는 방향으로 파라미터 값을 변형시켜가면서 최종적으로는
최솟값을 갖도록 하는 파라미터를 찾는 방법
3주차에 추가적으로 다룰 예정

iteratively reducing the error by updating the parameters in the direction that incrementally lowers the loss function.

예시) 산 정상에서 눈을 감고 내려올 때, 모든 방향으로 산을 더듬어가며 산의 높이가 낮아지는 방향으로 한발씩 내딛어가는 과정

03. Loss

01) Continuous (Linear Regression)

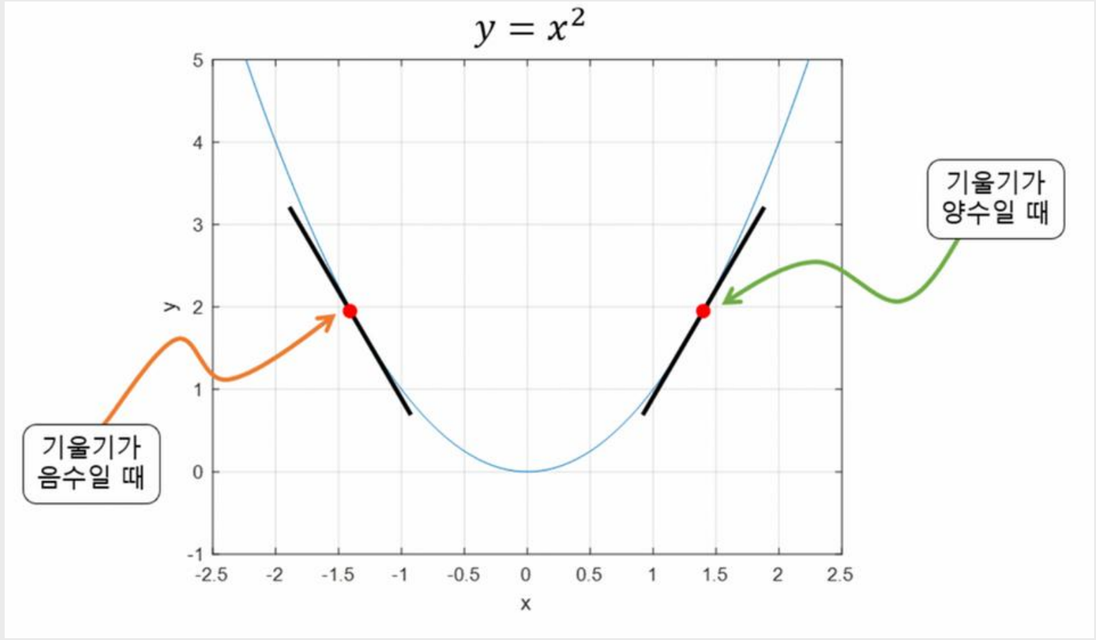


Gradient descent

손실 함수 값이 낮아지는 방향으로 파라미터 값을 변형시켜가면서 최종적으로는 최솟값을 갖도록 하는 파라미터를 찾는 방법

03. Loss

01) Continuous (Linear Regression)



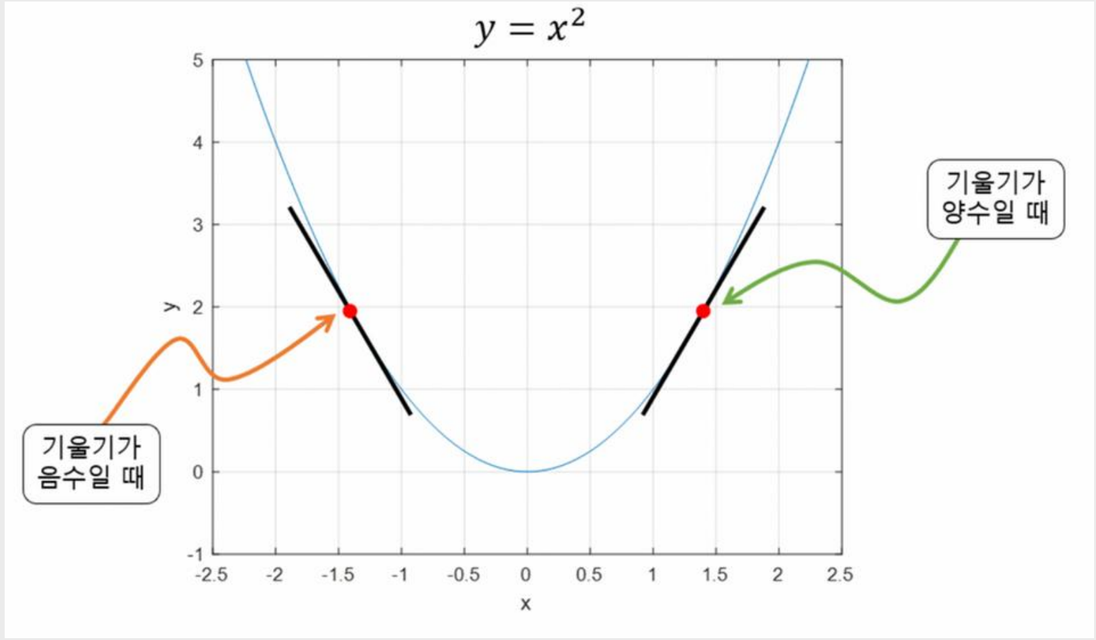
Gradient descent

함수의 기울기(즉, gradient)를 이용해
x의 값을 어디로 옮겼을 때 함수가 최소값을
찾는지 알아보는 방법

$$x_{i+1} = x_i - \text{이동 거리} \times \text{기울기의 부호}$$

03. Loss

01) Continuous (Linear Regression)



Gradient descent

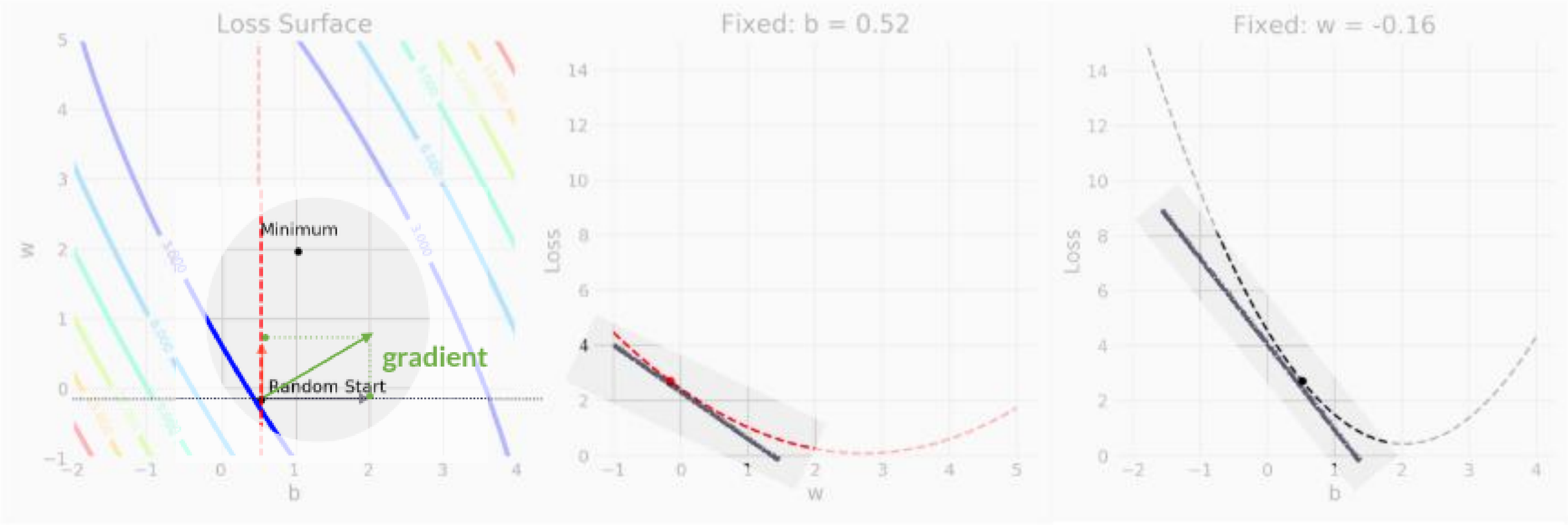
함수의 기울기(즉, gradient)를 이용해
x의 값을 어디로 옮겼을 때 함수가 최소값을
찾는지 알아보는 방법

$x_{i+1} = x_i -$ 이동 거리 $\times$ 기울기의 부호

$$x_{i+1} = x_i - \alpha \frac{df}{dx}(x_i)$$

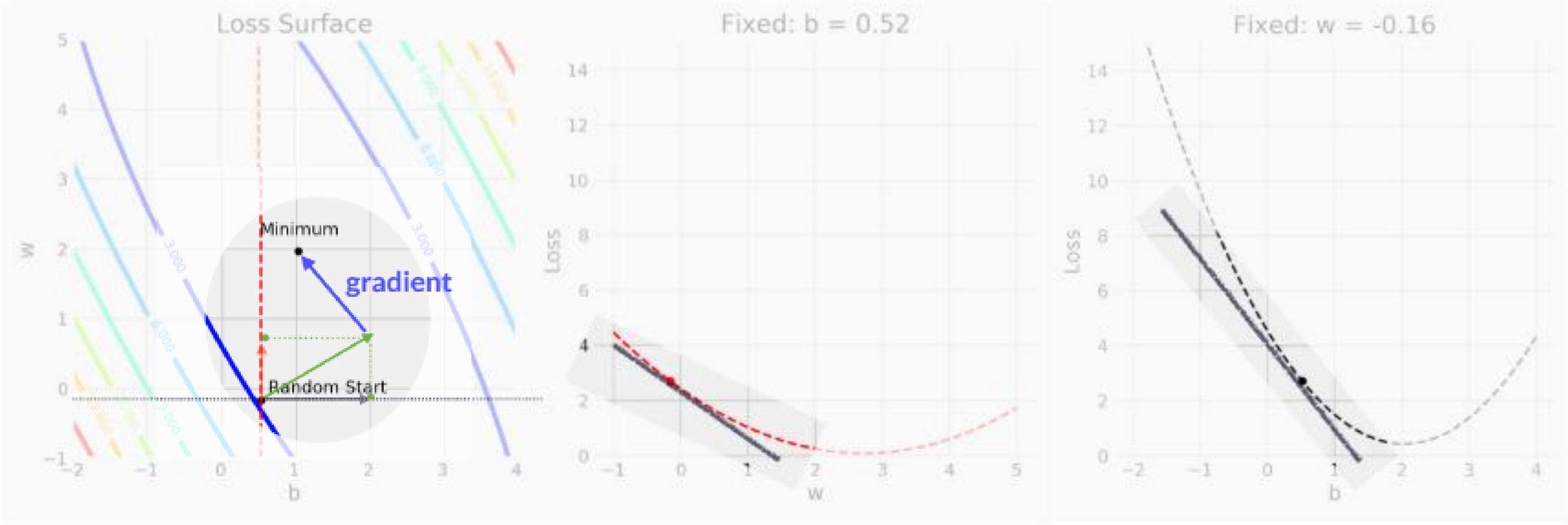
03. Loss

01) Continuous (Linear Regression)



03. Loss

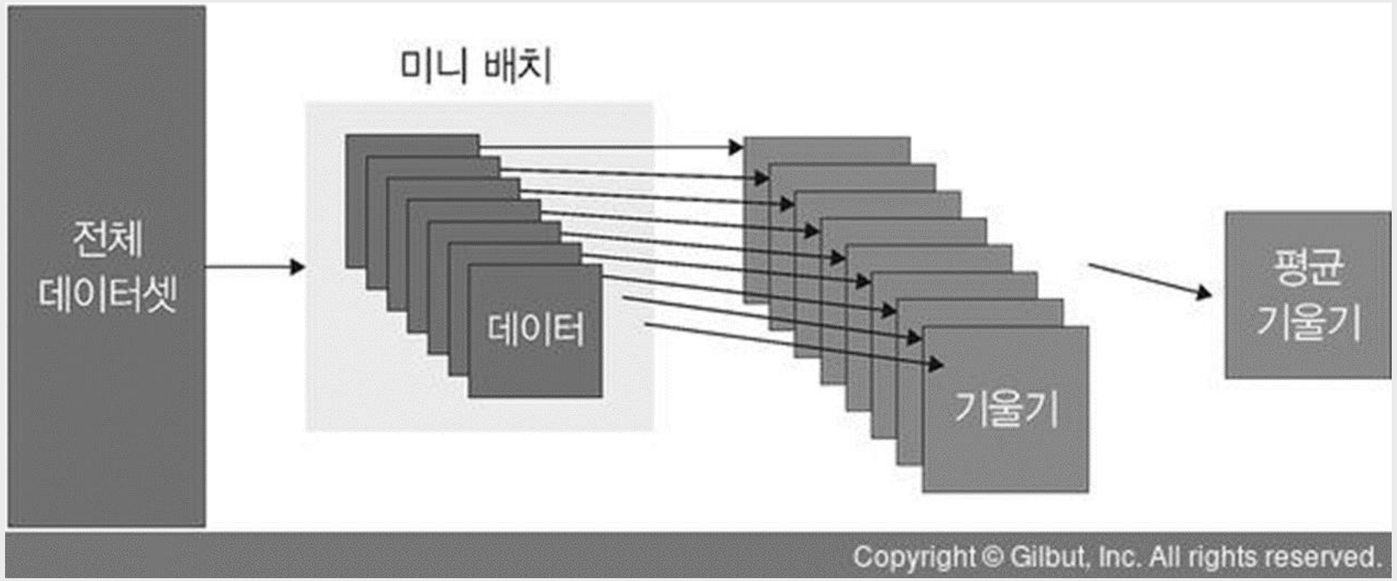
01) Continuous (Linear Regression)



03. Loss

01) Continuous (Linear Regression)

mini-batch Gradient Descent



전체 데이터셋에 대해 기울기를 한번만 계산하여 모델의 파라미터를 업데이트 하는 BGD (Batch Gradient Descent) 보다 전체 데이터셋을 미니 배치(mini-batch) 여러 개로 나누고, 미니 배치 한 개마다 기울기를 구한 후 그것의 평균 기울기를 이용하여 모델을 업데이트해서 학습하는 방법

03. Loss

01) Continuous (Linear Regression)

mini-batch Gradient Descent

Q : 왜 굳이 데이터셋을 미니배치로 쪼개 경사하강법을 진행하나요?

A :
1. 메모리 효율성
2. 업데이트 빈도 및 수렴 속도 향상
3. 일반화 능력 향상
4. 병렬 처리와 성능 최적화

03. Loss

02) Discrete (Softmax Regression)

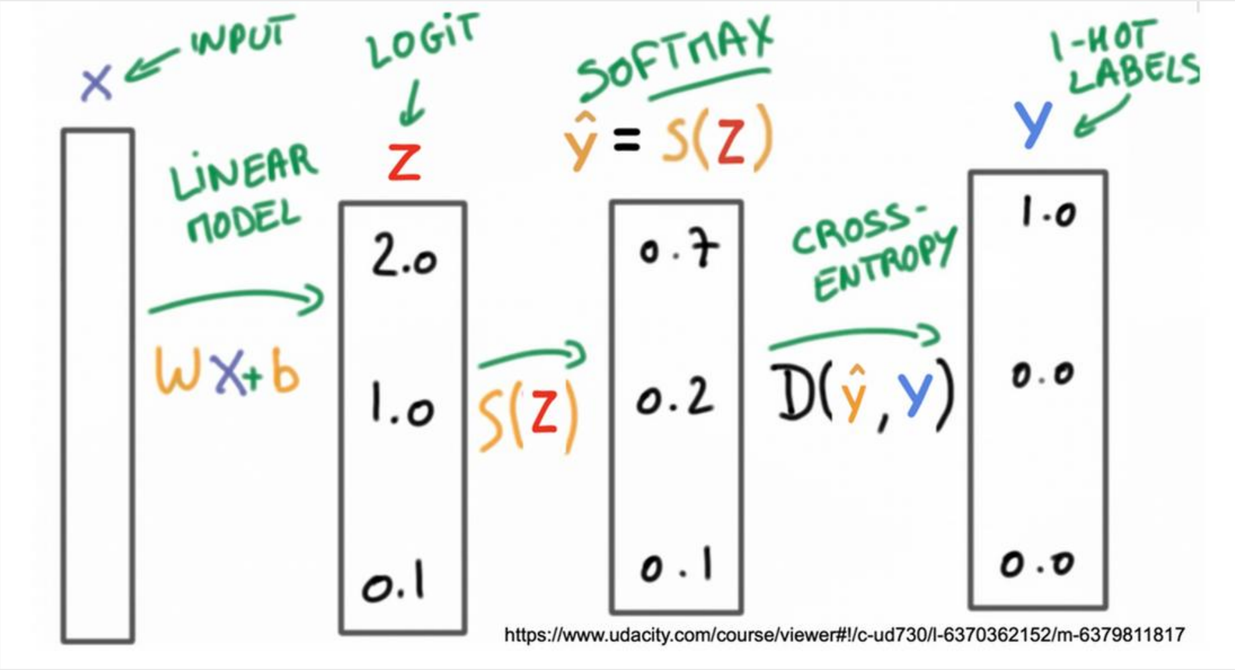
Softmax regression

클래스가 두 개 이상일 때 각 클래스에 속할 확률을 추정하고, 그 확률들을 기반으로 분류 결정을 내리는 다중 클래스 분류 문제를 해결하기 위해 사용되는 강력한 알고리즘

03. Loss

02) Discrete (Softmax Regression)

Softmax regression



step1) Linear model을 통한 Logit 출력
step2) logit의 probability 변환

$$\text{logit}(P) = \log\left(\frac{P}{1-P}\right)$$

logit : 각 클래스에 대한 출력이 아직 확률로 변환되기 전의 원시 점수(raw scores)를 의미

03. Loss

02) Discrete (Softmax Regression)

Softmax regression

$$\hat{\mathbf{y}} = \mathrm{softmax}(\mathbf{o}) \quad \text{where} \quad \hat{y}_j = \frac{\exp(o_j)}{\sum_k \exp(o_k)}.$$

1 : ensuring sum to 1
2 : ensuring non-negativity (0 ≤ y^j ≤ 1)

step1) Linear model을 통한 Logit 출력 (affine function)
step2) logit의 probability 변환 (softmax function)

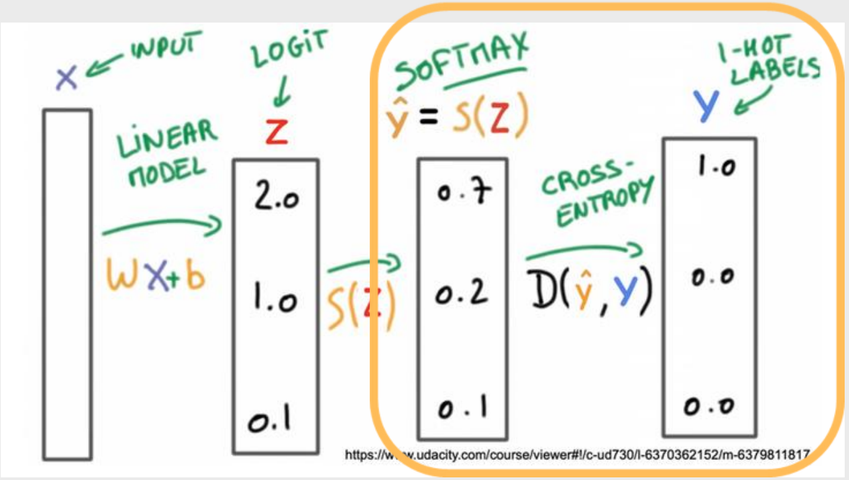
$$\underset{j}{\mathrm{argmax}}\, \hat{y}_j = \underset{j}{\mathrm{argmax}}\, o_j.$$

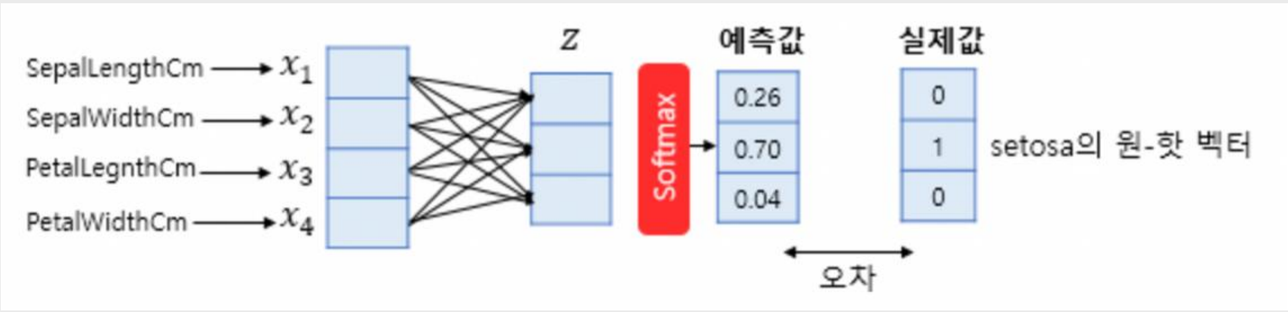
Linear? Nonlinear?

03. Loss

02) Discrete (Softmax Regression)

Softmax regression





step3) 실제값과 예측값의 오차를 계산하여 파라미터 업데이트 (역전파)

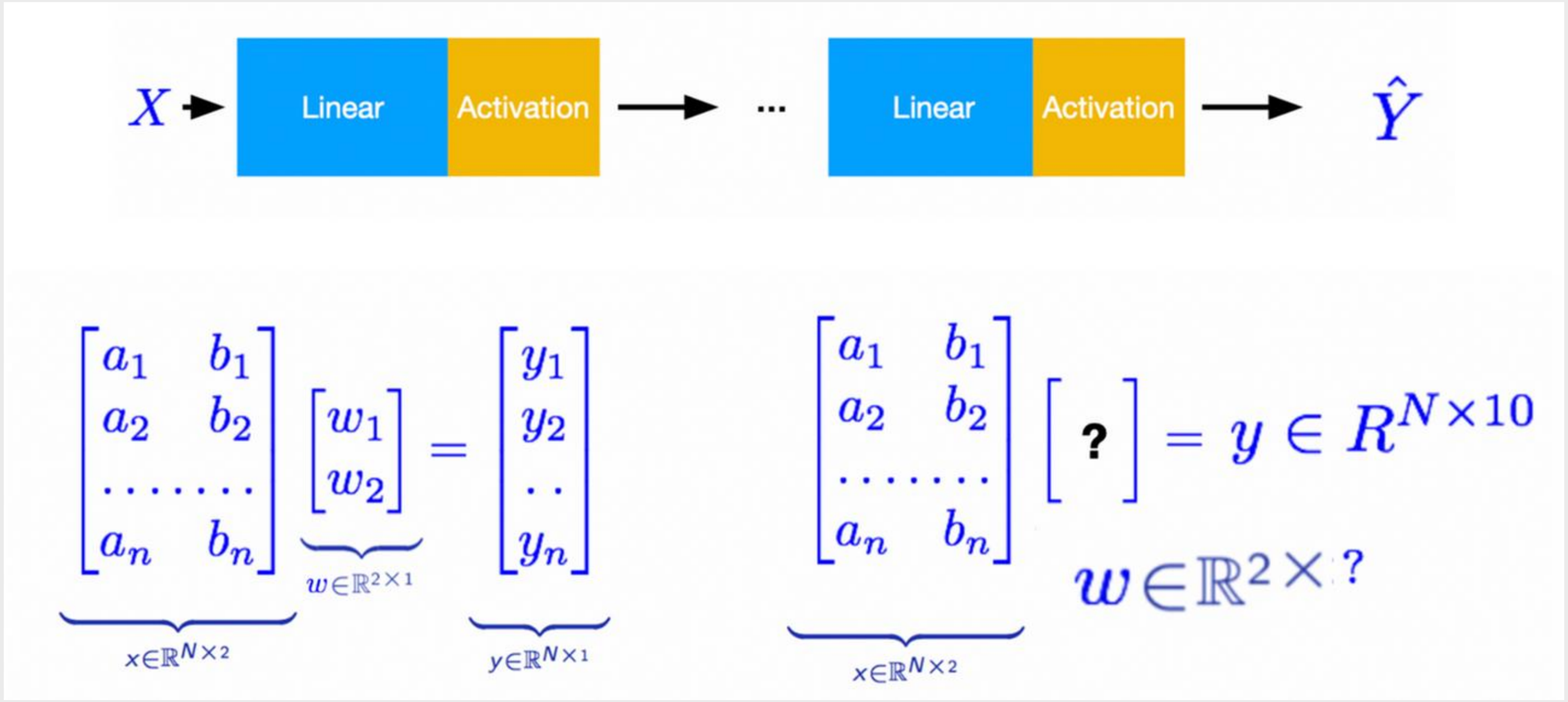
cross-entropy loss

$$L = -\sum_i y_i \log(\hat{y}_i)$$

03. Loss

02) Discrete (Softmax Regression)

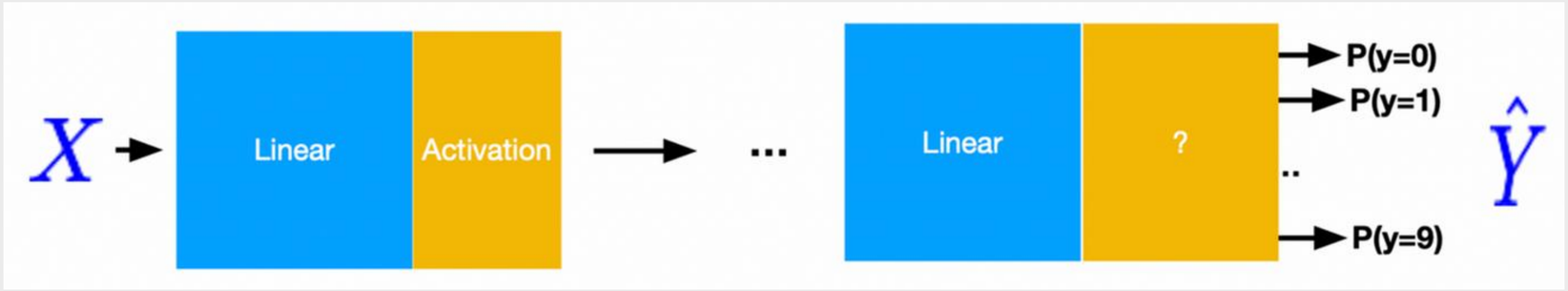
Softmax regression



03. Loss

02) Discrete (Softmax Regression)

Softmax regression



Dimensionality (number of inputs) : d

Batch size : n

Categories : q

Quiz

Dimension of Minibatch features X :

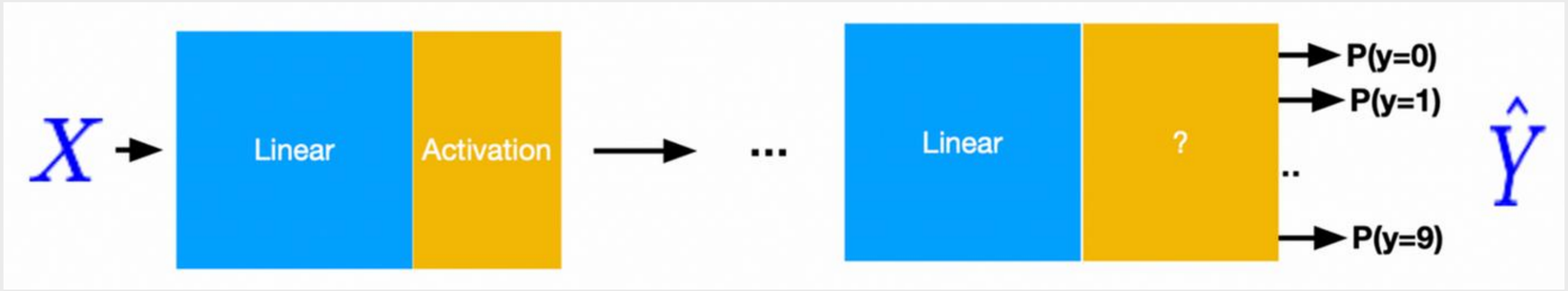
Dimension of Weights :

Dimension of Bias :

03. Loss

02) Discrete (Softmax Regression)

Softmax regression



Dimensionality (number of inputs) : d

Batch size : n

Categories : q

Answer

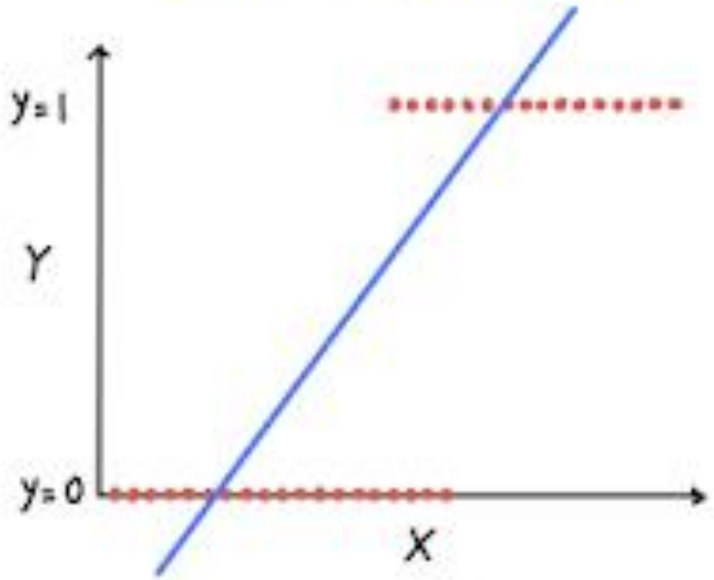
Dimension of Minibatch features X : n x d

Dimension of Weights : d x q

Dimension of Bias : 1 x q

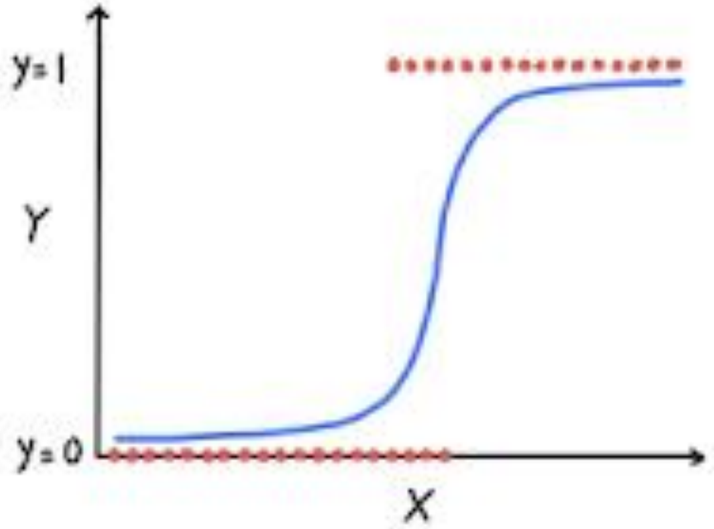
03. Loss

Linear Regression



$$y^n = \boldsymbol{w}^T \boldsymbol{x}^n$$

Logistic Regression



$$P(y^n = 1 \mid \boldsymbol{x}^n; \theta) = \sigma(\boldsymbol{w}^T \boldsymbol{x}^n)$$

04
Code Exploration

04. Code Exploration

Introduction

```python
class SGD(d2l.HyperParameters):
    """Minibatch stochastic gradient descent."""
    def __init__(self, params, lr):
        self.save_hyperparameters()

    def step(self):
        for param in self.params:
            param -= self.lr * param.grad

    def zero_grad(self):
        for param in self.params:
            if param.grad is not None:
```

DEF가 아닌 CLASS로 지정하는 이유?

1.모듈의 재사용성 -> 모델의 구성요소를 모듈화, 재사용 가능
2.상태 관리 : nn.Module을 상속받은 class는 가중치 초기화, 저장, 로딩 쉬움
3.코드 가독성과 구조화 : 복잡한 모델에서 가독성, 유지보수성 크게 향상
4.내장 함수와의 통합 : 내장 함수 원활한 사용 및 모델 학습 및 평가 모드 전환 등의 작업 간소화
5.연산 그래프와 자동미분 : forward method활용가능
6.확장성, 커스터마이징 : 사용자 정의 layer, function 쉽게 추가 가능

04. Code Exploration

Introduction

Class 정의

내부에서 데이터 운용 가능

원하는 과정을 한꺼번에 지정 가능

ex) 함수는 각 과정에 대한 함수 지정 -> 각 함수 실행

class는 class 내부에서 해결 가능(간결한 명령)

object가 가지는 attribute별로 관리 가능

(class 내부에서 어떤 option으로 실행되었는지)

참고 : https://leedakyeong.tistory.com/entry/Class-%EC%9D%B4%ED%95%B4%ED%95%98%EA%B8%B0-Class%EB%A5%BC-%EC%93%B0%EB%8A%94-%EC%9D%B4%EC%9C%A0-Class-vs-function

04. Code Exploration

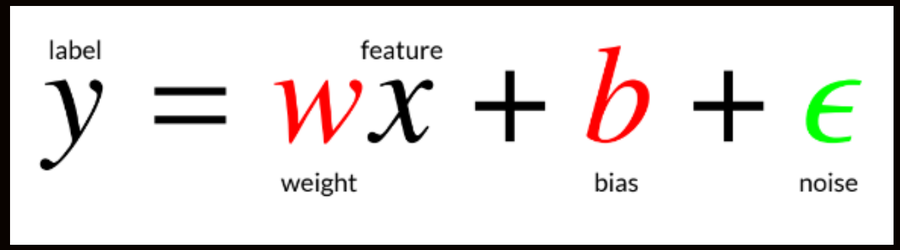
Data Generation

```
import numpy as np

true_b = 1
true_w = 2

N = 100

np.random.seed(42)
x = np.random.rand(N,1)  # Uniform
epsilon = (0.1 * np.random.randn(N,1))  # standard normal dist  # noise
y = true_b + true_w * x + epsilon  # data genneration
```



04. Code Exploration

Additionals about Data

```
from torch.utils.data import Dataset
class CustomDataset(Dataset):
  def __init__(self,x_train,y_train): #추후 customerdataset할 때 type error나타나서 수정함.
    self.x_data = x_train
    self.y_data = [[y] for y in y_train] #target에 해당하는 값을 2D list로 반환해줌 (왜 2D list?) => longtensor들어가려면 list구조여야함.
#  데이터셋의 전처리를 해주는 부분

  def __len__(self):
    return len(self.x_data)
#  데이터셋의 길이. 즉, 총 샘플의 수를 적어주는 부분

  def __getitem__(self, idx):
    x = torch.FloatTensor(self.x_data[idx]).to(device)
    y = torch.LongTensor(self.y_data[idx]).to(device)
#  데이터셋에서 특정 1개의 샘플을 가져오는 함수
```

DL에서 데이터셋 모두 가져올 때 메모리 터지는 일 발생 -> batch를 나눠 모델에게 입력
데이터를 한 번에 부르지 않고 필요할 때 불러 써야함 (getitem part)
이 때, 어떻게 가져오고, 얼마나 가져올지 customizing필요 -> CustomDataset Class로 data를 불러오는 것

04. Code Exploration

Additionals about Data

```
batch_size = 8

dataset = CustomDataset(x_train,y_train)
dataloader = DataLoader(dataset, batch_size=batch_size)
```

dataloader를 사용하는 이유
-> mini batch를 만들어주는 역할, 해당 batch 크기로 바꿔줌
-> option 중 num_worker 존재 : load속도 조절 가능(default = 0) -> CPU와 GPU 작업간 밸런스
->shuffle여부도 결정할 수 있음

참고 : https://jybaek.tistory.com/799 (num_worker 결정 기준에 대한 설명)

04. Code Exploration

Train-Validation Split

```
# Shuffles the indices
idx = np.arange(N)
split_index = int(N * 0.8) # train-validation split

train_idx = idx[:split_index]
val_idx = idx[split_index:]

# Generates train and validation sets
x_train, y_train = x[train_idx], y[train_idx]
x_val, y_val = x[val_idx], y[val_idx]
```

04. Code Exploration

Train-Validation Split

학습-검증 데이터를 4:1로 나누는 것은 관행적인 세팅입니다.
(p-value 0.05와 같은!)

```
# Shuffles the indices
idx = np.arange(N)
split_index = int(N * 0.8) # train-validation split

train_idx = idx[:split_index]
val_idx = idx[split_index:]

# Generates train and validation sets
x_train, y_train = x[train_idx], y[train_idx]
x_val, y_val = x[val_idx], y[val_idx]
```

04. Code Exploration

Train-Validation Split

```
# Shuffles the indices
idx = np.arange(N)
split_index = int(N * 0.8) # train-validation split

train_idx = idx[:split_index]
val_idx = idx[split_index:]

# Generates train and validation sets
x_train, y_train = x[train_idx], y[train_idx]
x_val, y_val = x[val_idx], y[val_idx]
```

이 부분은 scikit-learn 에서
train_test_split 으로 대체 가능!

04. Code Exploration

Gradient Descent for Linear Regression

```python
def train_model_numpy(lr = 0.1, epochs = 1000):
  # Initialize parameters
  b = np.random.randn(1)
  w = np.random.randn(1)

  for epoch in range(epochs):
    # Loss computation
    y_hat = b + w * x_train
    error = (y_hat - y_train)
    mse_loss = np.mean(error ** 2)

    # Gradient computation
    b_grad = 2 * np.mean(error)
    w_grad = 2 * np.mean(x_train * error)
    b = b - lr * b_grad
    w = w - lr * w_grad
  return b, w
```

lr(learning rate), epochs 를 hyperparameter 라 부릅니다.

04. Code Exploration

Gradient Descent for Linear Regression

```python
def train_model_numpy(lr = 0.1, epochs = 1000):
  # Initialize parameters
  b = np.random.randn(1)
  w = np.random.randn(1)

  for epoch in range(epochs):
    # Loss computation
    y_hat = b + w * x_train
    error = (y_hat - y_train)
    mse_loss = np.mean(error ** 2)

    # Gradient computation
    b_grad = 2 * np.mean(error)
    w_grad = 2 * np.mean(x_train * error)
    b = b - lr * b_grad
    w = w - lr * w_grad
  return b, w
```

Initial parameter 는 **random** 하게 선택합니다.
딥러닝 성능의 variation은 여기서 많이 옵니다.

04. Code Exploration

Gradient Descent for Linear Regression

```
def train_model_numpy(lr = 0.1, epochs = 1000):
  # Initialize parameters
  b = np.random.randn(1)
  w = np.random.randn(1)

  for epoch in range(epochs):
    # Loss computation
    y_hat = b + w * x_train
    error = (y_hat - y_train)
    mse_loss = np.mean(error ** 2)

    # Gradient computation
    b_grad = 2 * np.mean(error)
    w_grad = 2 * np.mean(x_train * error)
    b = b - lr * b_grad
    w = w - lr * w_grad
  return b, w
```

학습중인 parameter 를 기반으로 loss를 계산합니다.

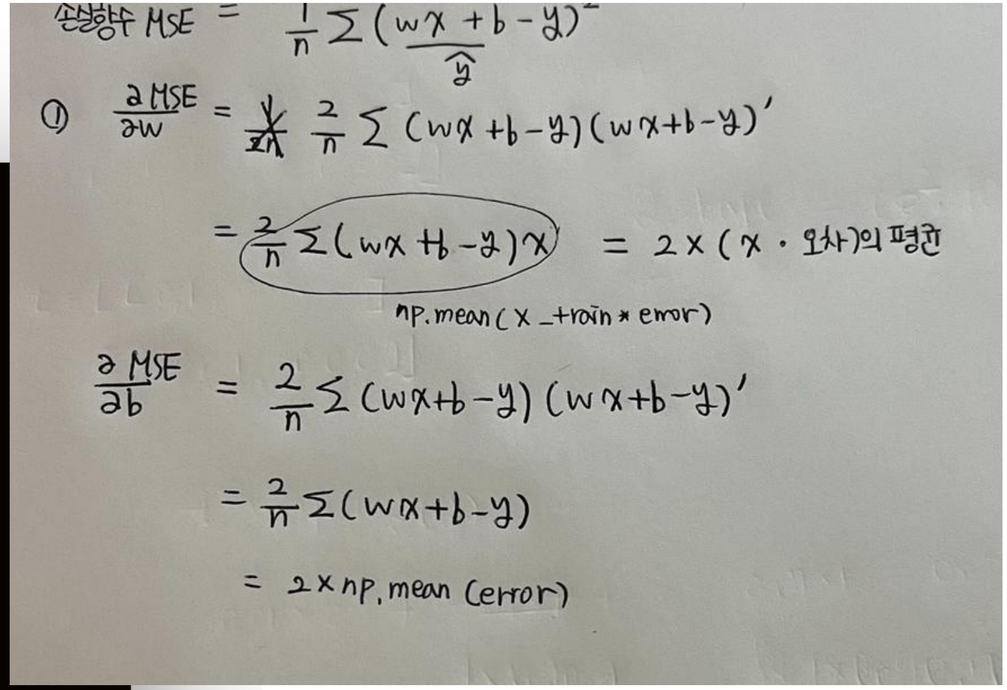
04. Code Exploration

Gradient Descent for Linear Regression

```python
def train_model_numpy(lr = 0.1, epochs = 1000):
  # Initialize parameters
  b = np.random.randn(1)
  w = np.random.randn(1)

  for epoch in range(epochs):
    # Loss computation
    y_hat = b + w * x_train
    error = (y_hat - y_train)
    mse_loss = np.mean(error ** 2)

    # Gradient computation
    b_grad = 2 * np.mean(error)
    w_grad = 2 * np.mean(x_train * error)
    b = b - lr * b_grad
    w = w - lr * w_grad
  return b, w
```



위에서 정의한 **mse loss** 의 미분을 계산해서 코딩하는 부분입니다.

04. Code Exploration

Gradient Descent for Linear Regression

```python
def train_model_numpy(lr = 0.1, epochs = 1000):
  # Initialize parameters
  b = np.random.randn(1)
  w = np.random.randn(1)

  for epoch in range(epochs):
    # Loss computation
    y_hat = b + w * x_train
    error = (y_hat - y_train)
    mse_loss = np.mean(error ** 2)

    # Gradient computation
    b_grad = 2 * np.mean(error)
    w_grad = 2 * np.mean(x_train * error)
    b = b - lr * b_grad
    w = w - lr * w_grad
  return b, w
```

이 부분이 **gradient descent** 를 이용한 parameter 최적화 코드입니다.

04. Code Exploration

Gradient Descent for Linear Regression

```python
def train_model_numpy(lr = 0.1, epochs = 1000):
  # Initialize parameters
  b = np.random.randn(1)
  w = np.random.randn(1)

  for epoch in range(epochs):
    # Loss computation
    y_hat = b + w * x_train
    error = (y_hat - y_train)
    mse_loss = np.mean(error ** 2)

    # Gradient computation
    b_grad = 2 * np.mean(error)
    w_grad = 2 * np.mean(x_train * error)
    b = b - lr * b_grad
    w = w - lr * w_grad
  return b, w
```

이 과정을 지정된 **epochs** 만큼 반복하도록 **for 문**을 사용합니다.

04. Code Exploration

Data Generation with PyTorch

```
import torch

# create tensor at CPU
x_train_tensor = torch.as_tensor(x_train)
y_train_tensor = torch.as_tensor(y_train)

# create tensor ar GPU
device = 'cuda' if torch.cuda.is_available() else 'cpu'
x_train_tensor = torch.as_tensor(x_train).to(device)
y_train_tensor = torch.as_tensor(y_train).to(device)
```

NumPy 로 이미 만들어둔 데이터를 재활용하도록 하겠습니다.

GPU가 가능하면 GPU에 업로드하고 아니면 CPU에 업로드하는 태그입니다.

04. Code Exploration

Data Generation with PyTorch

```python
import torch

# create tensor at CPU
x_train_tensor = torch.as_tensor(x_train)
y_train_tensor = torch.as_tensor(y_train)

# create tensor ar GPU
device = 'cuda' if torch.cuda.is_available() else 'cpu'
x_train_tensor = torch.as_tensor(x_train).to(device)
y_train_tensor = torch.as_tensor(y_train).to(device)
```

GPU텐서는 CPU텐서와 통신 없이는
서로 계산이 불가능하므로 주의!

04. Code Exploration

Gradient Descent by PyTorch

```python
def train_model_torch(lr = 0.1, epochs=1000):
  # Initialize parameters
  b = torch.randn(1, requires_grad = True, dtype = torch.float, device = device)
  w = torch.randn(1, requires_grad = True, dtype = torch.float, device = device)

  for epoch in range(epochs):
    # Loss computation
    y_hat = b + w * x_train_tensor
    error = (y_hat - y_train_tensor)
    mse_loss = torch.mean(error ** 2)
    # Gradient computation and descent
    mse_loss.backward()
    with torch.no_grad():
      b -= lr * b.grad  # in-place operation
      w -= lr * w.grad
    b.grad.zero_()
    w.grad.zero_()
  return b, w, [])
```

- requires_grad=True
  그래디언트(기울기)를 계산해야 함을 의미
- dtype=torch.float
  텐서의 데이터 타입을 부동소수점으로 설정
- device =device
  정해진 디바이스에 텐서를 태우는 역할

04. Code Exploration

Additional for Gradient Descent

1. torch.no_grad()

- **사용 상황: torch.no_grad()**는 그래디언트 계산이 필요 없는 경우에 사용됩니다. 이는 주로 모델을 평가하거나 추론(inference)할 때 사용됩니다.
- **역할:** 이 블록 내에서 수행되는 모든 연산은 그래디언트 추적이나 저장을 하지 않습니다.
- **예시:** 모델의 가중치를 업데이트하지 않고 출력만을 계산할 때 사용합니다.

```
with torch.no_grad():
    for input, target in dataset:
        output = model(input)
```

04. Code Exploration

Additional for Gradient Descent

3. detach()

- **사용 상황: detach()**는 기존 계산 그래프로부터 텐서를 분리하여 그래디언트 계산에서 제외할 때 사용됩니다.
- **역할:** 분리된 텐서는 동일한 데이터를 공유하지만, 그래디언트 계산에는 참여하지 않습니다.
- **예시:** 기존 텐서의 값은 유지하되, 그래디언트 계산에는 참여시키고 싶지 않을 때 사용합니다.

```
x = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)
y = x * 2
z = y.detach()  # z는 y와 동일한 값을 가지지만, 그래디언트 계산에는 참여하지 않습니다.
```

04. Code Exploration

Gradient Descent by PyTorch

```python
def train_model_torch(lr = 0.1, epochs=1000):
  # Initialize parameters
  b = torch.randn(1, requires_grad = True, dtype = torch.float, device = device)
  w = torch.randn(1, requires_grad = True, dtype = torch.float, device = device)

  for epoch in range(epochs):
    # Loss computation
    y_hat = b + w * x_train_tensor
    error = (y_hat - y_train_tensor)
    mse_loss = torch.mean(error ** 2)
    # Gradient computation and descent
    mse_loss.backward()
    with torch.no_grad():
      b -= lr * b.grad  # in-place operation
      w -= lr * w.grad
    b.grad.zero_()
    w.grad.zero_()
  return b, w, [])
```

모델 연산 부분은 정확히 일치!

04. Code Exploration

Gradient Descent by PyTorch

```python
def train_model_torch(lr = 0.1, epochs=1000):
  # Initialize parameters
  b = torch.randn(1, requires_grad = True, dtype = torch.float, device = device)
  w = torch.randn(1, requires_grad = True, dtype = torch.float, device = device)

  for epoch in range(epochs):
    # Loss computation
    y_hat = b + w * x_train_tensor
    error = (y_hat - y_train_tensor)
    mse_loss = torch.mean(error ** 2)
    # Gradient computation and descent
    mse_loss.backward()
    with torch.no_grad():
      b -= lr * b.grad  # in-place operation
      w -= lr * w.grad
    b.grad.zero_()
    w.grad.zero_()
  return b, w, [])
```

Pytorch 는 **backward** 를 통해 미분 계산이 가능!

04. Code Exploration

Gradient Descent by PyTorch

```python
def train_model_torch(lr = 0.1, epochs=1000):
  # Initialize parameters
  b = torch.randn(1, requires_grad = True, dtype = torch.float, device = device)
  w = torch.randn(1, requires_grad = True, dtype = torch.float, device = device)

  for epoch in range(epochs):
    # Loss computation
    y_hat = b + w * x_train_tensor
    error = (y_hat - y_train_tensor)
    mse_loss = torch.mean(error ** 2)
    # Gradient computation and descent
    mse_loss.backward()
    with torch.no_grad():
      b -= lr * b.grad  # in-place operation
      w -= lr * w.grad
    b.grad.zero_()
    w.grad.zero_()
  return b, w, [])
```

in-place operation 을 사용해 with torch.no_grad를 통해 autograd를 멈춥니다.

04. Code Exploration

Gradient Descent by PyTorch

```python
def train_model_torch(lr = 0.1, epochs=1000):
  # Initialize parameters
  b = torch.randn(1, requires_grad = True, dtype = torch.float, device = device)
  w = torch.randn(1, requires_grad = True, dtype = torch.float, device = device)

  for epoch in range(epochs):
    # Loss computation
    y_hat = b + w * x_train_tensor
    error = (y_hat - y_train_tensor)
    mse_loss = torch.mean(error ** 2)
    # Gradient computation and descent
    mse_loss.backward()
    with torch.no_grad():
      b -= lr * b.grad  # in-place operation
      w -= lr * w.grad
    b.grad.zero_()
    w.grad.zero_()
  return b, w, [])
```

in-place operation ?

새로운 메모리 공간을 할당하지 않고, 기존의 데이터를 직접 수정하는 연산을 의미합니다. 이는 메모리 사용을 줄이고, 때로는 연산 속도를 향상시킬 수 있습니다.

04. Code Exploration

Additional for Gradient Descent

2. requires_grad

- **사용 상황: requires_grad**는 텐서에 대한 그래디언트 계산을 활성화하거나 비활성화할 때 사용됩니다.
- **역할: requires_grad=True**로 설정된 텐서는 연산을 추적하며, 그래디언트 계산에 포함됩니다.
- **예시:** 특정 텐서에 대해 그래디언트를 계산하고자 할 때 사용합니다.

```
x = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)
y = x * 2
z = y.mean()
z.backward()  # 여기서 x에 대한 그래디언트가 계산됩니다.
```

04. Code Exploration

Gradient Descent by PyTorch

```python
def train_model_torch(lr = 0.1, epochs=1000):
  # Initialize parameters
  b = torch.randn(1, requires_grad = True, dtype = torch.float, device = device)
  w = torch.randn(1, requires_grad = True, dtype = torch.float, device = device)

  for epoch in range(epochs):
    # Loss computation
    y_hat = b + w * x_train_tensor
    error = (y_hat - y_train_tensor)
    mse_loss = torch.mean(error ** 2)
    # Gradient computation and descent
    mse_loss.backward()
    with torch.no_grad():
      b -= lr * b.grad  # in-place operation
      w -= lr * w.grad
    b.grad.zero_()
    w.grad.zero_()
  return b, w, [])
```

torch.no_grad()?

모델을 학습할 때 기울기 계산을 비활성화하여 메모리 사용량을 줄이고 계산 속도를 향상시킵니다. computation graph를 그리지 않도록 하는 장치입니다. 자동 미분(autograd) 기능이 불필요한 계산을 수행하는 것을 방지합니다.

04. Code Exploration

Gradient Descent by PyTorch

```python
def train_model_torch(lr = 0.1, epochs=1000):
  # Initialize parameters
  b = torch.randn(1, requires_grad = True, dtype = torch.float, device = device)
  w = torch.randn(1, requires_grad = True, dtype = torch.float, device = device)

  for epoch in range(epochs):
    # Loss computation
    y_hat = b + w * x_train_tensor
    error = (y_hat - y_train_tensor)
    mse_loss = torch.mean(error ** 2)
    # Gradient computation and descent
    mse_loss.backward()
    with torch.no_grad():
      b -= lr * b.grad  # in-place operation
      w -= lr * w.grad
    b.grad.zero_()
    w.grad.zero_()
  return b, w, [])
```

torch.no_grad() 를 쓰는 이유

in-place operation을 쓰면 같은 메모리 저장소에 계속 업데이트가 되므로 torch.no_grad() 를 써야 오류가 나지 않습니다.
만약 in-place 연산이 그래디언트 추적 상태에서 수행된다면, 원래 텐서의 값을 변경함으로써 이전에 그래프에 기록된 연산의 흐름이 손상될 수 있다. 이는 그래디언트 계산에 필요한 원본 데이터를 잃게 만들고, 결과적으로 오류를 발생시킵니다.
파라미터 업데이트 과정에서는 그래디언트 계산이 필요 없으므로 computation graph도 그릴 필요가 없습니다. 즉, 메모리 절약을 위해 in-place operation을 쓰든 안쓰든 파라미터 업데이트 과정에서는 torch.no_grad() 를 써 주는 게 좋습니다.

04. Code Exploration

Gradient Descent by PyTorch

```python
def train_model_torch(lr = 0.1, epochs=1000):
  # Initialize parameters
  b = torch.randn(1, requires_grad = True, dtype = torch.float, device = device)
  w = torch.randn(1, requires_grad = True, dtype = torch.float, device = device)

  for epoch in range(epochs):
    # Loss computation
    y_hat = b + w * x_train_tensor
    error = (y_hat - y_train_tensor)
    mse_loss = torch.mean(error ** 2)
    # Gradient computation and descent
    mse_loss.backward()
    with torch.no_grad():
      b -= lr * b.grad  # in-place operation
      w -= lr * w.grad
    b.grad.zero_()
    w.grad.zero_()
  return b, w, [])
```

gradient를 초기화 시켜줍니다.
초기화 시켜주지 않으면 누적합으로 계산하기 때문!

05

다음 주차 예고

05. 다음주차 예고

3주차 세션의 Key Word

MLP
Multi Layer Perceptrons

Propagation

Optimization

05. 다음주차 예고

2주차 과제

공부 과제

d2l
5장 Multilayer Perceptron

오승상 딥러닝
3, 4강

코드 과제

슬랙 공지로
추후 전달 예정

Thank You

2025 Summer DL Session 2주차 끝

!

KOREA
UNIVERSITY