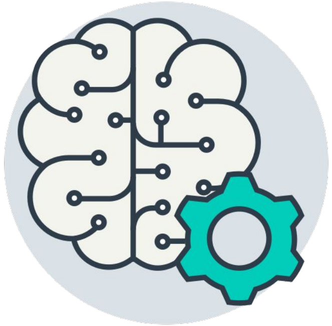
Multivariate Linear Regression

- Simple Linear Regression 복습
- Multivariate Linear Regression 이론
- Naive Data Representation
- Matrix Data Representation
- Multivariate Linear Regression
- `nn.Module` 소개
- `F.mse_loss` 소개

Simple Linear Regression

1



2

# of hours studied

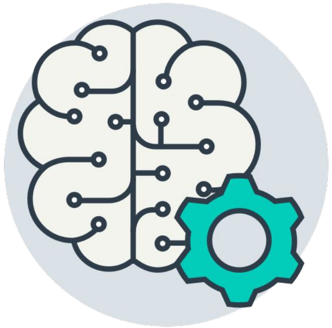
Test score

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

Multivariate Linear Regression

73
80
75

Quiz Scores



152

Final Score Prediction

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

Data

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_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]])
```

Hypothesis Function

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

x 라는 vector 와 W 라는 matrix의 곱

$$H(x) = w_1 x_1 + w_2 x_2 + w_3 x_3 + b$$

입력변수가 3개라면 weight 도 3개!

Hypothesis Function: Naive

- 단순한 hypothesis 정의!
- 하지만 x 가 길이 1000의 vector라면...?

```
# H(x) 계산
hypothesis = x1_train * w1 + x2_train * w2 + x3_train * w3 + b
```

$$H(x) = w_1 x_1 + w_2 x_2 + w_3 x_3 + b$$

Hypothesis Function: Matrix

```
# H(x) 계산
hypothesis = x_train.matmul(W) + b # or .mm or @
```

- `matmul()`로 한번에 계산
  a. 더 간결하고,
  b. x 의 길이가 바뀌어도 코드를 바꿀 필요가 없고
  c. 속도도 더 빠르다!

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

Cost function: MSE

- 기존 Simple Linear Regression 과 동일한 공식!

$$cost(W) = \frac{1}{m} \underbrace{\sum_{i=1}^{m}}_{\text{Mean}} \left(\underbrace{H(x^{(i)})}_{\text{Prediction}} - \underbrace{y^{(i)}}_{\text{Target}} \right)^2$$

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

Gradient Descent with `torch.optim`

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

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

```python
# optimizer 설정
optimizer = optim.SGD([W, b], lr=1e-5)

# optimizer 사용법
optimizer.zero_grad()
cost.backward()
optimizer.step()
```

Full Code with `torch.optim` (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]])

# 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)
```

1. 데이터 정의

2. 모델 정의

3. `optimizer` 정의

Full Code with `torch.optim` (2)

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

    # H(x) 계산
    hypothesis = x_train.matmul(W) + b # or .mm or @

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

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

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

4. Hypothesis 계산

5. Cost 계산 (MSE)

6. Gradient descent

Results

```
Epoch    0/20 hypothesis: tensor([0., 0., 0., 0., 0.]) Cost: 29661.800781
Epoch    1/20 hypothesis: tensor([67.2578, 80.8397, 79.6523, 86.7394, 61.6605]) Cost: 9298.520508
Epoch    2/20 hypothesis: tensor([104.9128, 126.0990, 124.2466, 135.3015,  96.1821]) Cost: 2915.713135
Epoch    3/20 hypothesis: tensor([125.9942, 151.4381, 149.2133, 162.4896, 115.5097]) Cost: 915.040527
Epoch    4/20 hypothesis: tensor([137.7968, 165.6247, 163.1911, 177.7112, 126.3307]) Cost: 287.936005
Epoch    5/20 hypothesis: tensor([144.4044, 173.5674, 171.0168, 186.2332, 132.3891]) Cost: 91.371017
Epoch    6/20 hypothesis: tensor([148.1035, 178.0144, 175.3980, 191.0042, 135.7812]) Cost: 29.758139
Epoch    7/20 hypothesis: tensor([150.1744, 180.5042, 177.8508, 193.6753, 137.6805]) Cost: 10.445305
Epoch    8/20 hypothesis: tensor([151.3336, 181.8983, 179.2240, 195.1707, 138.7440]) Cost: 4.391228
Epoch    9/20 hypothesis: tensor([151.9824, 182.6789, 179.9928, 196.0079, 139.3396]) Cost: 2.493135
Epoch   10/20 hypothesis: tensor([152.3454, 183.1161, 180.4231, 196.4765, 139.6732]) Cost: 1.897688
Epoch   11/20 hypothesis: tensor([152.5485, 183.3610, 180.6640, 196.7389, 139.8602]) Cost: 1.710541
Epoch   12/20 hypothesis: tensor([152.6620, 183.4982, 180.7988, 196.8857, 139.9651]) Cost: 1.651413
Epoch   13/20 hypothesis: tensor([152.7253, 183.5752, 180.8742, 196.9678, 140.0240]) Cost: 1.632387
Epoch   14/20 hypothesis: tensor([152.7606, 183.6184, 180.9164, 197.0138, 140.0571]) Cost: 1.625923
Epoch   15/20 hypothesis: tensor([152.7802, 183.6427, 180.9399, 197.0395, 140.0759]) Cost: 1.623412
Epoch   16/20 hypothesis: tensor([152.7909, 183.6565, 180.9530, 197.0538, 140.0865]) Cost: 1.622141
Epoch   17/20 hypothesis: tensor([152.7968, 183.6643, 180.9603, 197.0618, 140.0927]) Cost: 1.621253
Epoch   18/20 hypothesis: tensor([152.7999, 183.6688, 180.9644, 197.0662, 140.0963]) Cost: 1.620500
Epoch   19/20 hypothesis: tensor([152.8014, 183.6715, 180.9666, 197.0686, 140.0985]) Cost: 1.619770
Epoch   20/20 hypothesis: tensor([152.8020, 183.6731, 180.9677, 197.0699, 140.1000]) Cost: 1.619033
```

Final (y)
152
185
180
196
142

- 점점 작아지는 Cost
- 점점 y 에 가까워지는 H(x)
- Learning rate 에 따라 발산할수도!

nn.Module

```python
# 모델 초기화
W = torch.zeros((3, 1), requires_grad=True)
b = torch.zeros(1, requires_grad=True)

# H(x) 계산
hypothesis = x_train.matmul(W) + b # or .mm or @
```

```python
import torch.nn as nn

class MultivariateLinearRegressionModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(3, 1)

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

hypothesis = model(x_train)
```

- nn.Module 을 상속해서 모델 생성
- nn.Linear(3, 1)
    - 입력 차원: 3
    - 출력 차원: 1
- Hypothesis 계산은 forward() 에서!
- Gradient 계산은 PyTorch 가 알아서 해준다 backward()

F.mse_loss

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

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


# cost 계산
cost = F.mse_loss(prediction, y_train)
```

- torch.nn.functional 에서 제공하는 loss function 사용
- 쉽게 다른 loss와 교체 가능! (l1_loss, smooth_l1_loss 등...)

Full Code with `torch.optim` (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]])

# 모델 초기화
W = torch.zeros((3, 1), requires_grad=True)
b = torch.zeros(1, requires_grad=True)
model = MultivariateLinearRegressionModel()


# optimizer 설정
optimizer = optim.SGD([W, b], lr=1e-5)
```

1. 데이터 정의

2. 모델 정의

3. `optimizer` 정의

Full Code with `torch.optim` (2)

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

    # H(x) 계산
    hypothesis = x_train.matmul(W) + b # or .mm or @
    Hypothesis = model(x_train)

    # cost 계산
    cost = torch.mean((hypothesis - y_train) ** 2)
    cost = F.mse_loss(prediction, y_train)

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

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

4. Hypothesis 계산

5. Cost 계산 (MSE)

6. Gradient descent

What’s Next?

- 지금까지 적은 양의 데이터를 가지고 학습했습니다.
- 하지만 딥러닝은 많은 양의 데이터와 함께할 때 빛을 발합니다.
- PyTorch 에서는 많은 양의 데이터를 어떻게 다룰까요?