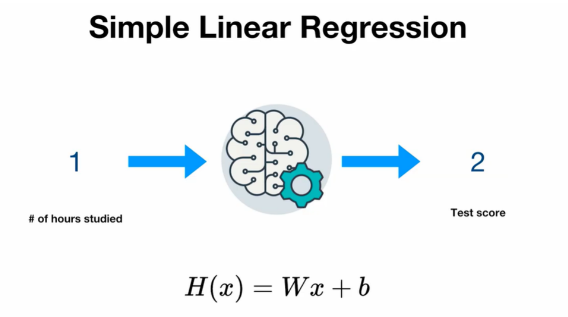
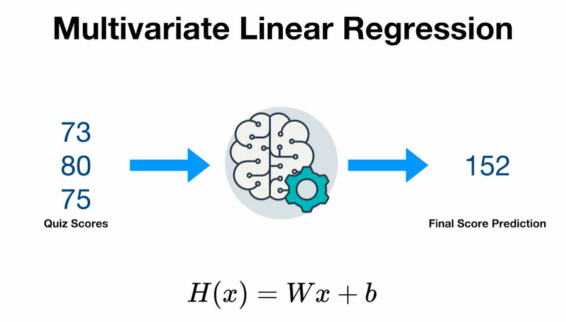
Multivariable Linear regression



하나의 정보로부터 하나의 결론을 짓는 모델



Multivariable Linear regression : 복수의 정보가 존재할 때 하나의 결론을 짓는 모델

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

단순하게는 이렇게 표현할 수 있지만 x의 개수가 많아지면 불가능

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

```
hypothesis = x_train.matmul(W) + b
```

- **matmul()** 로 한번에 계산
    - 더 간결함
    - x의 길이가 바뀌어도 코드를 바꿀 필요가 없음
    - 속도가 더 빠름

Cost Function

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

기존 Simple Linear Regression과 동일한 공식

학습방식

Gradient Descent with `torch.optim`

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

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

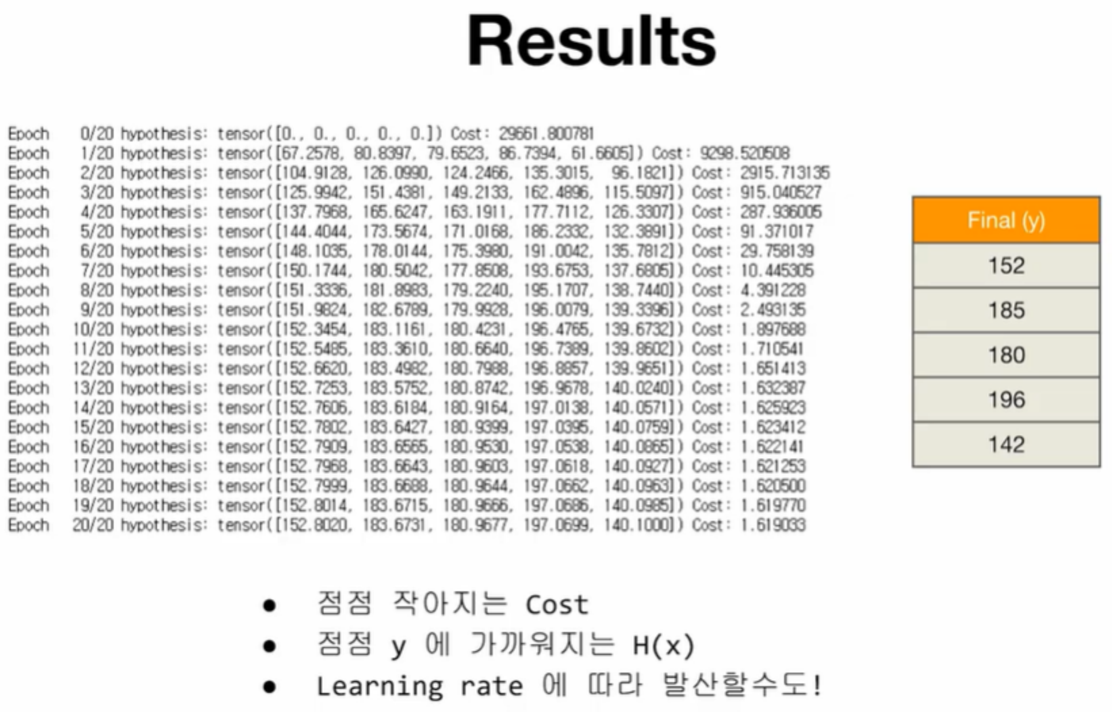
Optimizer을 설정한 후 Cost를 구할때마다 Optimizer의 Gradient에 저장한 후 Gradient Descent를 실행한다.

```
#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)
```

Simple Linear Regression과 달라진 점은 데이터와 W를 정의하는 부분밖에 없다.

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

hypothesis = x_train.matmul(W) + b

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

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

print('Epoch {:4d}/{} Batch {}/{} Cost: {:.6f}'.format(
            epoch, nb_epochs, hypothesis.spueeze().detach(),
            cost_item()
        ))
```

학습 부분은 동일하다.



nn.Module

```
#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
```

```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 Pytorch가 알아서 해준다 - backward()

F.mse_loss

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

import torch.nn.functional as F
cost = F.mse_loss(prediction, y_train)
```

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