Linear Regression

상태	Basic ML
담당자	

Simple Linear Regression

Hypothesis

- 모델 : $H(x) = Wx + b$
- 타겟 : y
- Loss function : $loss(W, b) = \frac{1}{m} \sum_{i=1}^{m} \left(H(x^{(i)}) - y^{(i)} \right)^2$

Loss function을 최소화하는 파라미터 W, b를 찾는 것이 linear regression의 목적

PyTorch

```
import torch
import torch.nn as nn # 신경망을 만들 수 있는 베이스
import torch.nn.functional as F # 신경망에 필요한 각종 함수들이 모여 있는 모듈
import torch.optim as optim # 신경망에 필요한 최적화 알고리즘이 모여 있는 모듈
```

```
# 데이터
x_train = torch.FloatTensor([[1], [2], [3]])
y_train = torch.FloatTensor([[1], [2], [3]])
# 모델 초기화
W = torch.zeros(1, requires_grad=True) # 계산할 때 W에 대한 미분값을 W.grad에 저장
b = torch.zeros(1, requires_grad=True) # 계산할 때 W에 대한 미분값을 b.grad에 저장
# optimizer 설정
optimizer = optim.SGD([W, b], lr=0.01) # 최적화 알고리즘 중 확률적 경사 하강법 사용

nb_epochs = 1000
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()
        ))
```

▼ 참고로 `LinearRegressionModel` 클래스를 다음과 같이 작성할 수 있다.

```
class LinearRegressionModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(1, 1)
    def forward(self, x):
        return self.linear(x)
```

`super().__init__()`을 하는 이유 : `nn.Module`을 상속한 `LinearRegressionModel`이 `nn.Module`의 속성을 사용하고자 할 때, 이를 위해 초기화를 해 주는 것

PyTorch Linear Regression 학습 과정 요약

1. `X_train`과 `y_train`을 준비

2. 모델 초기화 (`requires_grad=True` 옵션을 켜 줘서 미분 과정을 저장할 수 있게 한다)
   일단 크기가 맞지 않아도 tensor manipulation으로 크기가 자동으로 맞추어지는 듯 하다.
   a. 가중치 `W`
   b. Bias `b`
3. 최적화 방법 설정 (여기서는 gradient descent를 사용) : `optim.SGD([params], lr)`
4. 에포크 설정
   a. 모델 계산
   b. Loss function 계산
   c. 모델 업데이트
      i. `optimizer.zero_grad()` : 에포크마다 새로운 경사값을 설정할 것이므로 0으로 초기화
      ii. `loss.backward()` : gradient 계산
      iii. `optimizer.step()` : 최적화 알고리즘 개선 (이때 `W`, `b`의 개선이 이루어짐)

Multivariable Linear Regression

Hypothesis

- 모델 : $H(x_1, x_2, x_3) = x_1w_1 + x_2w_2 + x_3w_3 + b$
- 타겟 : y
- Loss function : $loss(W, b) = \frac{1}{m}\sum_{i=1}^{m}\left(H(x^{(i)}) - y^{(i)}\right)^2$

Loss function을 최소화하는 파라미터 W, b를 찾는 것이 linear regression의 목적

PyTorch

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

>>
torch.Size([5, 3])
torch.Size([5, 1])
```

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

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

5 x 3 사이즈의 텐서를 input으로 받아 `nn.Linear`를 통과시켜 `y`와 shape이 같은 5 x 1 텐서를 만들자. `nn.Linear`의 파라미터로 3, 1을 주어서 5 x 3 → [3 x 1] ⇒ 5 x 1 이 되게 만들자.

```
# 모델 초기화
model = MultivariateLinearRegressionModel()
# optimizer 설정
optimizer = optim.SGD(model.parameters(), lr=1e-5) # init 덕분에 파라미터들은 초기화

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

    # H(x) 계산
    prediction = model(x_train)

    # cost 계산
    cost = F.mse_loss(prediction, y_train) # loss function을 쉽게 쓸 수 있음

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

    # 20번마다 로그 출력
    print('Epoch {:4d}/{} Cost: {:.6f}'.format(
        epoch, nb_epochs, cost.item()
    ))
```