20.03.30

주제 **Regression**

Lab 03

Linear Regression

이론 말고 코딩하는 방법을 배울 것임.

예시로는 공부한 시간(Input; x)에 대비한 점수(Output; y)를 예측하는 프로그램.

Data definition

```
x_train = torch.FloatTensor([[1],[2],[3]])
y_train = torch.FloatTensor([[2],[4],[6]])
```

Hypothesis (Linear Regression)

```
# Weight, Bias를 0으로 초기화 -> zeros
# requires_grad = True 로 해줌으로써 학습해나가야 한다는 것을 명시

# 참고 : Parameters of torch.zeros
# torch.zeros(*size, out=None, dtype=None, layout=torch.strided, device=None, requires_grad=False)

w = torch.zeros(1, requires_grad = True)
b = torch.zeros(1, requires_grad = True)

hypothesis = x_train * W + b
```

Compute Loss

- Mean Squared Error(MSE)

$$\frac{1}{m}\sum_{i=1}^{m}(H(x^{(i)}-y^{(i)}))^2$$

이걸 수식으로 구현하면

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

정도로 간단히 표현 가능하다.

Gradient Descent

- torch.optim 라이브러리 사용
  - parameter : [W, b] 는 학습할 tensor들, lr은 learning rate

```
optimizer = optim.SGD([W,b], lr=0.01)

# 항상 붙어다니는 3줄
optimizer.zero_grad()   # gradient 초기화
cost.backward()         # gradient 계산
optimizer.step()        # W,b 개선
```

- 실전 코드

```
# setup :
x_train = torch.FloatTensor([[1],[2],[3]])
y_train = torch.FloatTensor([[2],[4],[6]])          # Data 정의

w = torch.zeros(1, requires_grad = True)
b = torch.zeros(1, requires_grad = True)            # Hypothesis 초기화

optimizer = optim.SGD([W,b], lr=0.01)               # Optimizer 정의

nb_epochs = 1000                                    # 학습 횟수
 # loop :
for epoch in range(1, nb_epochs + 1):
    hypothesis = x_train * W + b                    #Hypothesis 예측
    cost = torch.mean((hypothesis - y_train) **2)   # Cost 계산

    # 학습
    optimizer.zero_grad()               # gradient 0으로 초기화
    cost.backward()                     # cost function 미분 - gradient 계산
    optimizer.step()                    # Gradient Descent 수행
```

Why Gradient Descent(경사하강법)?

- cost(W)를 그래프로 나타내면 대략 아래로 볼록한 2차 함수가 나온다.

이 2차 함수의 꼭지점(cost가 가장 적은 지점)을 기계적으로 찾아내는 것이
*경사하강법(Gradient Descent Algoritm)*이다.

- 경사가 어떤지 보고 그 방향으로 W를 조금씩 움직이는 기술.

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

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

- 수식에서 alpha는 W를 얼마나 움직일지 정하는 상수, Learning rate(lr) 과 같음
- 어떤 점에서 시작하든 간에 항상 최소점에 도달할 수 있다.
- 이 방법을 위해서 cost(W)에서 1/m 대신 1/2m로 정의하기도 한다.
(미분했을 때 상수를 없애기 위해)
- In code ...

```
gradient = 2* torch.mean((W * x_train - y_train) * x_train)
lr = 0.1
W -= lr*gradient
```

- Full Code (Without 'optim' Funtion)

```
x_train = torch.FloatTensor([[1],[2],[3]])
y_train = torch.FloatTensor([[1],[2],[3]])

W = torch.zeros(1)
lr = 0.1

nb_epoches = 10
for epoch in range(nb_epoches + 1):
    htpothesis = x_train * W

    cost = torch.mean((hypothesis - x_train) **2)
    gradient = torch.sum((W * x_train - y_train) * x_train)

    print('Epoch {:4d}/{} W: {:.3f}, Cost {:.6f}'.format(epoch, nb_epochs,W.item(),cost.item()))

    W -= lr*gradient
```

Lab 04 - 1

Multivariable Linear Regression

저번까진 하나의 정보만으로 예측을 하는 것을 배웠는데, 이번에는 복수의 정보를 기반으로 하나의 추측값을 도출하는 다항 선형 회귀를 해볼 것이다.

- Data (example)

```
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

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

Data input의 가지 수에 따라 w의 수도 똑같이 맞춰주는 게 인지상정.

- In code...

```
# Since there are so many arguments...
# We will use 'matmul()'
hypothesis = x_train.matmul(W) + b
```

간결할 뿐만 아니라 더욱 빠르기도 하다고 함.

- Cost function : MSE
  - Same as Simple Linear Regression

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

- Gradient Descent with torch.optim
  - Same as Simple Linear Regression

Full Code Example with torch.optim

```
import torch
import torch.optim as optim

# 1. Data Initialization
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]])

# 2. Model Initialization
W = torch.zeros((3,1), requires_grad = True)
b = torch.zeros(1, requires_grad = True)

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

nb_epochs = 20
for epoch in range(nb_epochs + 1):
    # 4. Hypothesis
    hypothesis = x_train.matmul(W) + b # or .mm or @
    # 5. Cost
    cost = torch.mean((hypothesis - y_train)**2)
    # Gradient Descent
    optimizer.zero_grad()
    cost.backward()
    optimizer.step()

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

결과로는 점점 Cost가 작아지고 점점 y에 가까워지는 H(x)를 볼 수 있음

lr 잘못 설정하면 발산할 수도 있음

▼ 실행결과

```
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: 2915.712646
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
```

nn.Module

- **모델 초기화** 과정을 간편하게 만들기 위해 있는 모듈.

```
# 2. Model Initialization
W = torch.zeros((3,1), requires_grad = True)
b = torch.zeros(1, requires_grad = True)

# ...

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

이 부분을

```
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)
```

로 표현할 수도 있음.

- nn.Module을 상속해서 모델 생성
- nn.Linear(3,1) : (입력차원, 출력차원) 을 파라미터로 넣기
- Hypothesis 계산은 forward 함수에 어떻게 하는지만 알려주기
- Gradient 계산은 PyTorch에서 알아서 해줌 backward()

PyTorch 제공 Cost Funtion

- 왜 쓸까?

  다른 Cost Function으로 전환할 때 편리함

  계산 오류를 피할 수 있어 디버깅할 때 편리함

- Code

```
import torch.nn.function as F

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

- 제공되는 다른 cost funtion 예:
  - l1_loss
  - smooth_l1_loss
  - etc.

Module 이용 Full Code

```
# Package
import torch
import torch.optim as optim

# minibatch 생성하기

from torch.utils.data import Dataset
# 원하는 Dataset을 지정할 수 있게 됨

class CustomDataset(Dataset):
    def __init__(self):
        self.x_data = [[73, 80, 75],
                       [93, 88, 93],
                       [89, 91, 90],
                       [96, 98, 100],
                       [73, 66, 70]]
        self.y_data = [[152],[185],[180],[196],[142]]
    # __len__() : 이 데이터셋의 총 데이터수
    def __len__(self):
        return len(self.x_data)
    # __getitem__() :  index를 받았을 때 그에 상응하는 입출력 데이터 반환
    def __getitem__(self, idx):
        x = torch.FloatTensor(self.x_data[idx])
        y = torch.FloatTensor(self.y_data[idx])

        return x, y

dataset = CustomDataset()

from torch.utils.data import DataLoader

dataloader = DataLoader(
      dataset,
    batch_size = 2,     # Size of each minibatch
    shuffle = True      # 프로그램이 순서 자체를 학습할 위험이 있어, Batch 생성시마다 순서를 바꿔줌.
)

# lab 04 -1 실습 코드 중 클래스 선언하는 쪽 이용.
import torch.nn as nn
import torch.nn.functional as F

class MultivariateLinearRegressionModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(3,1)  # w,b 포함 linear layer
```

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

# Model
model = MultivariateLinearRegressionModel()

# Optimizer
optimizer = optim.SGD(model.parameters(), lr=1e-5)

nb_epochs = 20
for epoch in range(nb_epochs + 1):
    for batch_idx, samples in enumerate(dataloader):
            # enumerate(dataloader):    minibatch 인덱스와 데이터를 받음
        x_train, y_train = samples

        prediction = model(x_train)
        cost = F.mse_loss(prediction, y_train)

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

        print('Epoch {:4d}/{} Batch{}/{} Cost: {:.6f}'.format(epoch, nb_epochs, batch_idx+1, len(dataloader), cost.item()))
```

▼ 실행결과

```
Epoch    0/20 hypothesis: tensor([14.9085, 25.6099, 21.2488, 23.7558, 20.8729]) Cost: 22748.216797
Epoch    1/20 hypothesis: tensor([73.8074, 96.4009, 91.0009, 99.7140, 74.8683]) Cost: 7132.474121
Epoch    2/20 hypothesis: tensor([106.7831, 136.0339, 130.0527, 142.2403, 105.0982]) Cost: 2237.765869
Epoch    3/20 hypothesis: tensor([125.2452, 158.2228, 151.9164, 166.0493, 122.0225]) Cost: 703.533813
Epoch    4/20 hypothesis: tensor([135.5818, 170.6454, 164.1572, 179.3791, 131.4975]) Cost: 222.632568
Epoch    5/20 hypothesis: tensor([141.3691, 177.6001, 171.0105, 186.8420, 136.8020]) Cost: 71.894333
Epoch    6/20 hypothesis: tensor([144.6095, 181.4936, 174.8475, 191.0203, 139.7716]) Cost: 24.645065
Epoch    7/20 hypothesis: tensor([146.4240, 183.6733, 176.9957, 193.3596, 141.4339]) Cost: 9.833952
Epoch    8/20 hypothesis: tensor([147.4401, 184.8934, 178.1985, 194.6694, 142.3643]) Cost: 5.190468
Epoch    9/20 hypothesis: tensor([148.0093, 185.5763, 178.8721, 195.4027, 142.8849]) Cost: 3.733967
Epoch   10/20 hypothesis: tensor([148.3282, 185.9584, 179.2492, 195.8134, 143.1762]) Cost: 3.276457
Epoch   11/20 hypothesis: tensor([148.5071, 186.1722, 179.4605, 196.0433, 143.3390]) Cost: 3.132051
Epoch   12/20 hypothesis: tensor([148.6075, 186.2917, 179.5788, 196.1721, 143.4299]) Cost: 3.085807
Epoch   13/20 hypothesis: tensor([148.6640, 186.3584, 179.6452, 196.2442, 143.4806]) Cost: 3.070328
Epoch   14/20 hypothesis: tensor([148.6959, 186.3955, 179.6824, 196.2847, 143.5087]) Cost: 3.064468
Epoch   15/20 hypothesis: tensor([148.7141, 186.4161, 179.7034, 196.3074, 143.5242]) Cost: 3.061669
Epoch   16/20 hypothesis: tensor([148.7245, 186.4274, 179.7152, 196.3201, 143.5326]) Cost: 3.059777
Epoch   17/20 hypothesis: tensor([148.7307, 186.4336, 179.7218, 196.3273, 143.5371]) Cost: 3.058201
Epoch   18/20 hypothesis: tensor([148.7344, 186.4369, 179.7257, 196.3314, 143.5394]) Cost: 3.056733
Epoch   19/20 hypothesis: tensor([148.7367, 186.4385, 179.7279, 196.3338, 143.5404]) Cost: 3.055282
Epoch   20/20 hypothesis: tensor([148.7383, 186.4392, 179.7293, 196.3351, 143.5407]) Cost: 3.053836
```

Lab 04 - 2

Minibatch Gradient Descent

복잡한 머신러닝 모델을 학습하려면 엄청난 양(수십만개)의 데이터를 다룬다.

문제는 이 엄청난 양의 데이터를 한번에 학습시키려면 시간적, 하드웨어적 자원이 부족하다.

따라서 일부만 데이터로 학습하는 방식을 써보겠다.

- 개념

  전체 데이터를 작은 Minibatch 단위로 쪼개어 학습.

  모든 데이터에 대한 Cost를 계산하지 않고 각 minibatch에 있는 데이터만 계산

- 효과

  한 번의 업데이트 때마다 계산할 cost 양이 더 적어 업데이트 주기가 빨라짐

  하지만 전체 데이터를 쓰지 않게 때문에 잘못된 방향으로 업데이트를 할 수도 있음(매끄럽지 못함)

- PyTorch Dataset 설정

```
from torch.utils.data import Dataset
# 원하는 Dataset을 지정할 수 있게 됨

class CustomDataset(Dataset):
    def __init__(self):
        self.x_data = [[73, 80, 75],
                       [93, 88, 93],
                       [89, 91, 90],
                       [96, 98, 100],
                       [73, 66, 70]]
        self.y_data = [[152],[185],[180],[196],[142]]
    # __len__() : 이 데이터셋의 총 데이터수
    def __len__(self):
        return len(self.x_data)
    # __getitem__() :  index를 받았을 때 그에 상응하는 입출력 데이터 반환
    def __getitem__(self, idx):
        x = torch.FloatTensor(self.x_data[idx])
        y = torch.FloatTensor(self.y_data[idx])

        return x, y

dataset = CustomDataset()
```

- PyTorch DataLoader 사용

```
from torch.utils.data import DataLoader

dataloader = DataLoader(
    dataset,
    batch_size =2,      # Size of each minibatch
    shuffle = True      #
)
```

  Dataset을 설정한 후에는 이렇게 DataLoader라는 걸 이 데이터셋에 사용할 수 있다.

  - batch_size는 통상 2의 제곱수로 설정하는 편임.
  - 권장: Shuffle = True로 할 시, Epoch마다 데이터셋을 섞어 학습되는 순서를 바꾼다.
    (순서 자체를 학습해버릴 위험이 있음.)

Full Code with Dataset and DataLoader

```
from torch.utils.data import Dataset
# 원하는 Dataset을 지정할 수 있게 됨

class CustomDataset(Dataset):
    def __init__(self):
        self.x_data = [[73, 80, 75],
                       [93, 88, 93],
                       [89, 91, 90],
                       [96, 98, 100],
                       [73, 66, 70]]
        self.y_data = [[152],[185],[180],[196],[142]]
    # __len__() : 이 데이터셋의 총 데이터수
    def __len__(self):
        return len(self.x_data)
    # __getitem__() :  index를 받았을 때 그에 상응하는 입출력 데이터 반환
```

```
    def __getitem__(self, idx):
        x = torch.FloatTensor(self.x_data[idx])
        y = torch.FloatTensor(self.y_data[idx])

        return x, y

dataset = CustomDataset()

from torch.utils.data import DataLoader

dataloader = DataLoader(
      dataset,
    batch_size = 2,     # Size of each minibatch
    shuffle = True      #
)


nb_epochs = 20
for epoch in range(nb_epochs + 1):
    for batch_idx, samples in enumerate(dataloader):
        # enumerate(dataloader)     minibatch 인덱스와 데이터를 받음
        x_train, y_train = samples

        prediction = model(x_train)
        cost = F.mse_loss(prediction, y_train)

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

        print('Epoch {:4d}/{} Batch{}/{} Cost: {:.6f}'.format(epoch, nb_epochs, batch_idx+1, len(dataloader), cost.item()))
```

▼ 실행결과

```
Epoch    0/20 Batch1/3 Cost: 12157.089844
Epoch    0/20 Batch2/3 Cost: 4924.210938
Epoch    0/20 Batch3/3 Cost: 641.451294
Epoch    1/20 Batch1/3 Cost: 342.544586
Epoch    1/20 Batch2/3 Cost: 234.793259
Epoch    1/20 Batch3/3 Cost: 66.887321
Epoch    2/20 Batch1/3 Cost: 9.590186
Epoch    2/20 Batch2/3 Cost: 5.412623
Epoch    2/20 Batch3/3 Cost: 0.029994
Epoch    3/20 Batch1/3 Cost: 0.679318
Epoch    3/20 Batch2/3 Cost: 0.175700
Epoch    3/20 Batch3/3 Cost: 1.130171
Epoch    4/20 Batch1/3 Cost: 0.237743
Epoch    4/20 Batch2/3 Cost: 0.330704
Epoch    4/20 Batch3/3 Cost: 0.649463
Epoch    5/20 Batch1/3 Cost: 0.763309
Epoch    5/20 Batch2/3 Cost: 0.127478
Epoch    5/20 Batch3/3 Cost: 0.444166
Epoch    6/20 Batch1/3 Cost: 0.058704
Epoch    6/20 Batch2/3 Cost: 0.742591
Epoch    6/20 Batch3/3 Cost: 0.364084
Epoch    7/20 Batch1/3 Cost: 0.294895
Epoch    7/20 Batch2/3 Cost: 0.099529
Epoch    7/20 Batch3/3 Cost: 0.835008
Epoch    8/20 Batch1/3 Cost: 0.152616
Epoch    8/20 Batch2/3 Cost: 0.299804
Epoch    8/20 Batch3/3 Cost: 0.704902
Epoch    9/20 Batch1/3 Cost: 0.069351
Epoch    9/20 Batch2/3 Cost: 0.328606
Epoch    9/20 Batch3/3 Cost: 0.802508
Epoch   10/20 Batch1/3 Cost: 0.525486
Epoch   10/20 Batch2/3 Cost: 0.228544
Epoch   10/20 Batch3/3 Cost: 0.193374
Epoch   11/20 Batch1/3 Cost: 0.396457
Epoch   11/20 Batch2/3 Cost: 0.232007
Epoch   11/20 Batch3/3 Cost: 0.458104
```

```
Epoch   12/20 Batch1/3 Cost: 0.115202
Epoch   12/20 Batch2/3 Cost: 0.567447
Epoch   12/20 Batch3/3 Cost: 0.748050
Epoch   13/20 Batch1/3 Cost: 0.071090
Epoch   13/20 Batch2/3 Cost: 0.324147
Epoch   13/20 Batch3/3 Cost: 0.793566
Epoch   14/20 Batch1/3 Cost: 0.373485
Epoch   14/20 Batch2/3 Cost: 0.052910
Epoch   14/20 Batch3/3 Cost: 0.766159
Epoch   15/20 Batch1/3 Cost: 0.278453
Epoch   15/20 Batch2/3 Cost: 0.342148
Epoch   15/20 Batch3/3 Cost: 0.558046
Epoch   16/20 Batch1/3 Cost: 0.080471
Epoch   16/20 Batch2/3 Cost: 0.864004
Epoch   16/20 Batch3/3 Cost: 0.395734
Epoch   17/20 Batch1/3 Cost: 0.488057
Epoch   17/20 Batch2/3 Cost: 0.108816
Epoch   17/20 Batch3/3 Cost: 0.342036
Epoch   18/20 Batch1/3 Cost: 0.500481
Epoch   18/20 Batch2/3 Cost: 0.116562
Epoch   18/20 Batch3/3 Cost: 0.350133
Epoch   19/20 Batch1/3 Cost: 0.181985
Epoch   19/20 Batch2/3 Cost: 0.270600
Epoch   19/20 Batch3/3 Cost: 0.646172
Epoch   20/20 Batch1/3 Cost: 0.544204
Epoch   20/20 Batch2/3 Cost: 0.145107
Epoch   20/20 Batch3/3 Cost: 0.307509
```

Lab 05

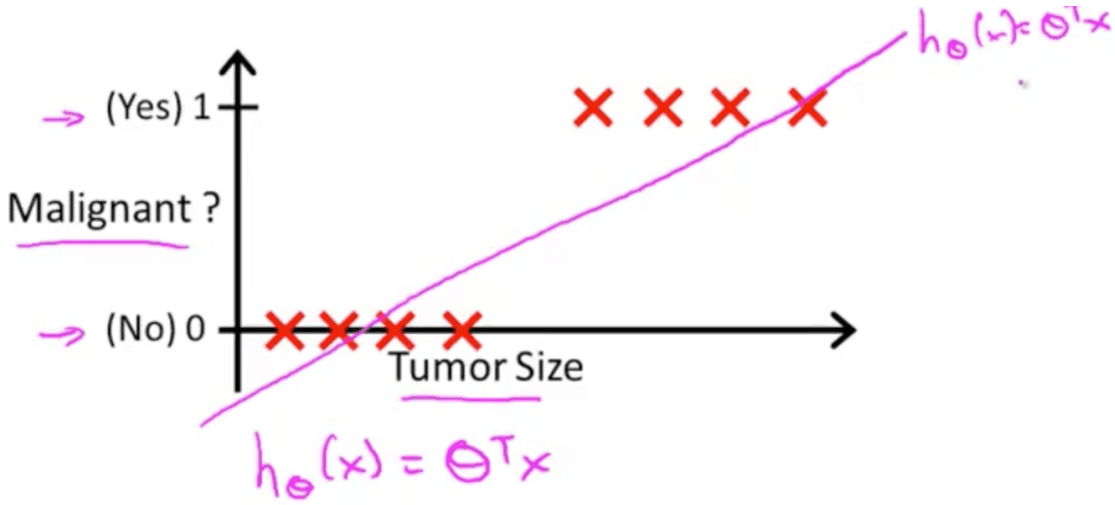
Logistic Regression

이론

- Logistic Classification

  : 정해진 2가지의 카테고리에 분류(Classify)하는 방식
  (예 : 이메일 스팸 검출 / 페이스북 피드에 띄울 글 정하기 / 신용카드 도용 검출 / 주식 동향 등)

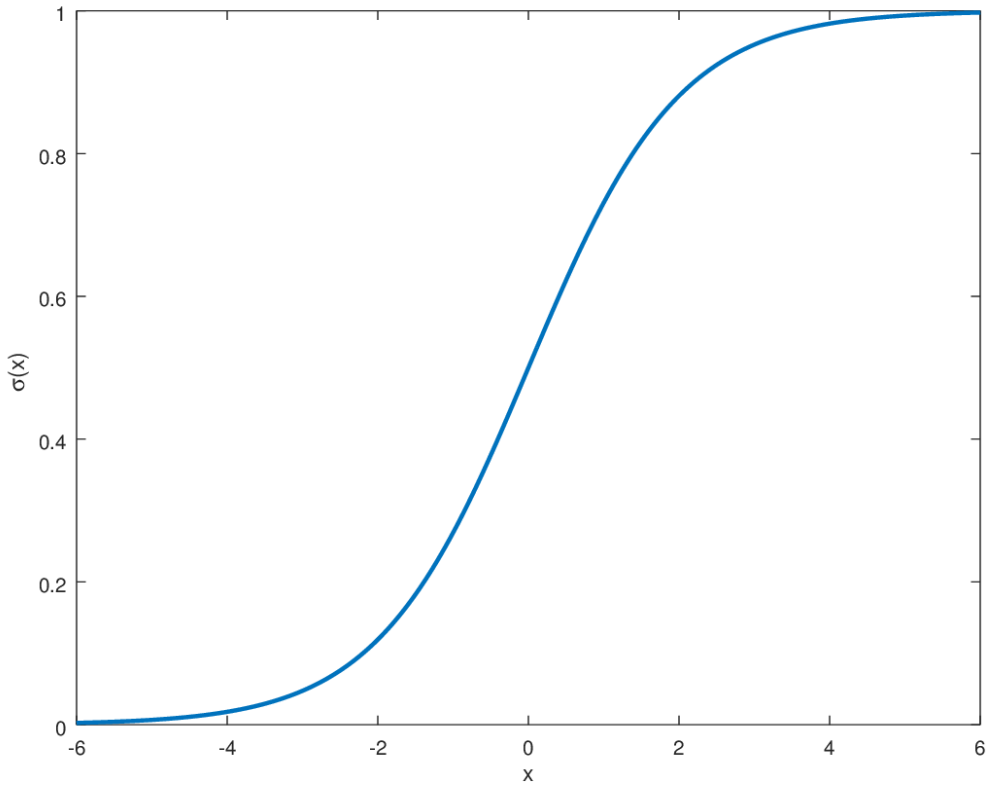
  - 각 카테고리를 0, 1로 표현
  - Linear Regression을 사용하지 못하는 이유:
    선형 그래프로는 0과 1을 가르기에 적합하지 않음.



경계도 제대로 못 지을 뿐 아니라, y가 꼭 0과 1 사이로 나오지도 않음

- Sigmoid function

$$g(z) = \frac{1}{1 + e^{-}z}$$



0과 1 사이의 값을 갖는 것이, Binary classification에 적합하다.

- Cost Function
  - Linear로 hypothesis를 주었을 땐 cost(w) 모양은 2차함수였음.
    하지만 sigmoid 모양으로 cost를 기존의 제곱을 써버리면 굉장히 난해한 곡선이 나온다.
    따라서 경사하강법을 사용할 경우 시작점에 따라 local minimum을 찾게 된다.
  - Hypothesis에 맞춰 새로 도입하는 cost function:

$$cost(W) = \frac{1}{m}\sum c(H(x), y)$$

$$c(H(x), y) = \begin{cases} -\log(H(x)) & if \;\; y = 1 \\ -\log(1 - H(x)) & if \;\; y = 0 \end{cases}$$

  - Understanding Cost function

Hypothesis에 exponential 함수가 들어갔기 때문에 log를 쓴다.
함수 구조상, 예측을 맞출 경우 cost가 0이고 틀릴 경우 cost 거의 무한대로 커진다.

자세한 설명은 서술하기 힘드니 모딥 lec 5-2 참조

- Minimize (Gradient Descent)

  역시나 경사하강법을 쓴다. 이전과 비슷하게

$$W := W - \alpha \frac{\partial}{\partial W} cost(W)$$

  이다.

  이 공식은 실전에선 그냥 라이브러리를 쓰면 된다.

실습

- Hypothesis
  - H(x)는 주어진 x값에 대한 예측... 이자 X 가 1일 확률.
  - cost(W,b)는 H(x)가 얼마나 잘 예측했는지 나타내는 지표 (작을 수록 좋다)

$$H(X) = \frac{1}{1 + e^{-W^T X}}$$

$$cost(W) = -\frac{1}{m} \sum y \log((H(x) + (1-y)(\log(1 - H(x))$$

- In code
  - Settings

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

# Torch Seed 부여
torch.manual_seed(1)
```

  - Training data

```
# Training Data
x_data = [[1,2],[2,3],[3,1],[4,3],[5,3],[6,2]]  # |x_data| = 6X2
y_data = [[0], [0], [0], [1], [1], [1]]         # |y_data| = 6X1


x_train = torch.FloatTensor(x_data)
y_train = torch.FloatTensor(y_data)
```

  - Hypothesis

```
# Hypothesis
W = torch.zeros((2,1), requires_grad = True)
b = torch.zeros(1, requires_grad = True)

# 수식 그대로 표현
hypothesis = 1 / (1 + torch.exp(-(x_train.matmul(W) + b)))
# torch 제공함수로 계산
hypothesis = torch.sigmoid(x_train.matmul(W)+ b)
```

- Cost Function

```
# 수식 그대로 표현
losses = -(y_train[0]*torch.log(hypothesis[0]) +
        (1-y_train) * torch.log(1-hypothesis[0]))
cost = losses.mean()

# torch 제공함수로 계산 : BCE(Binary Cross Entropy)
cost = F.binary_cross_entropy(hypothesis, y_train)
```

- Whole Training Process

```
# 모델 초기화
W = torch.zeros((2,1), require_grad = True)
b = torch.zeros(1, require_grad = True)

# Optimizer 설정
optimizer = optim.SGD([W,b], lr = 1)

nb_epochs = 1000
for epoch in range(nb_epochs + 1):
    # Cost 계산
    hypothesis = torch.sigmoid(x_train.matmul(W) + b)
    cost = F.binary_cross_entropy(hypothesis, y_train)

    # Calculate H(x)
    optimizer.zero_grad()
    cost.backward()
    optimizer.step()

    # 100번마다 로그 출력하게
    if epoch % 100 == 0:
        print('Epoch {:4d}/{} Cost: {:.6f}'.format(
            epoch,nb_epochs, cost.item()))
```

▼ 실행결과

```
Epoch    0/1000 Cost: 0.693147
Epoch  100/1000 Cost: 0.134722
Epoch  200/1000 Cost: 0.080643
Epoch  300/1000 Cost: 0.057900
Epoch  400/1000 Cost: 0.045300
Epoch  500/1000 Cost: 0.037261
Epoch  600/1000 Cost: 0.031673
Epoch  700/1000 Cost: 0.027556
Epoch  800/1000 Cost: 0.024394
Epoch  900/1000 Cost: 0.021888
Epoch 1000/1000 Cost: 0.019852
```

- Evaluation (내가 만든 모델의 성능이 얼마나 좋을까?)

```
# 실제로는 x_train이 아니라 x_test로 해줘야함
hypothesis = torch.sigmoid(x_train.matmul(W) + b)
print(hypothesis[:5])
```

를 실행하면 몇의 확률로 1이 될지 알 수 있음.

```
prediction = hypothesis >= torch.FloatTensor([0.5])
print(prediction[:5])
# prediction은 ByteTensor
```

이렇게 하면 확률이 0.5 이상 되는 애들을 1로 예측하도록 찍을 수 있음.

이후에는 prediction과 y_train값을 비교하면 된다.

```
correct_prediction = prediction.float() == y_train
```

으로 예측과 train 값이 일치하는지 확인할 수 있음.

실전 : Higher Implementation with Class

- 클래스 선언

```
class BinaryClassifier(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(8,1)
        self.sigmoid = nn.Sigmoid()

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

model = BinaryClassifier()
```

- Full Code

```
# optimizer 설정
optimizer = optim.SGD(model.parameters(), lr=1)

nb_epochs = 100
for epoch in range(nb_epochs + 1):

    # H(x) 계산
    hypothesis = model(x_train)
  # 돌려보면 여기서 오류가 나는데... size가 6x2와 8x1로 안 맞는다고 함
    # 뭔갈 빠트린 걸까?

    # cost 계산
    cost = F.binary_cross_entropy(hypothesis, y_train)

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

    # 20번마다 로그 출력
    if epoch % 10 == 0:
        prediction = hypothesis >= torch.FloatTensor([0.5])
        correct_prediction = prediction.float() == y_train
        accuracy = correct_prediction.sum().item() / len(correct_prediction)
        print('Epoch {:4d}/{} Cost: {:.6f} Accuracy {:2.2f}%'.format(
            epoch, nb_epochs, cost.item(), accuracy * 100,
      ))
```

▼ 실행결과

```
Epoch    0/100 Cost: 0.897528 Accuracy 50.00%
Epoch   10/100 Cost: 0.546400 Accuracy 83.33%
```

```
Epoch   20/100 Cost: 0.588509 Accuracy 83.33%
Epoch   30/100 Cost: 0.478054 Accuracy 83.33%
Epoch   40/100 Cost: 0.392104 Accuracy 83.33%
Epoch   50/100 Cost: 0.311146 Accuracy 83.33%
Epoch   60/100 Cost: 0.238634 Accuracy 83.33%
Epoch   70/100 Cost: 0.184601 Accuracy 100.00%
Epoch   80/100 Cost: 0.156515 Accuracy 100.00%
Epoch   90/100 Cost: 0.143471 Accuracy 100.00%
Epoch  100/100 Cost: 0.133744 Accuracy 100.00%
```

Lab 06

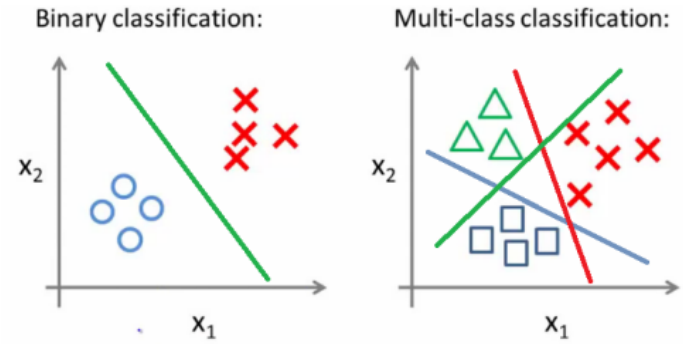
Softmax Regression

Logistic Classification을 통해서 Binary Classification을 한다는 걸 알았으니,
이제 Multinomial Classification의 경우를 생각 해보자.

이론

- **개념**

  만약 data를 A, B, C로 나누어야 하는 상황이라면,
  A, B에 속하는 데이터를 'C가 아님'으로 생각하고, C에 속하는 데이터를 'C임' 으로 생각하면
  Binary Classification과 동일하게 된다.



  따라서 모델은 'A or not?', 'B or not?', 'C or not?' 세가지 모델을 쓰는게 된다.

  즉, 동일한 X 벡터를 세 가지 W 벡터에 대해 연산을 하게 되는 건데,
  이 건 행렬곱의 원리를 이용해 하나의 연산으로 합칠 수가 있다.

  참고 :

$$[w_1 \ w_2 \ w_3]\begin{bmatrix}x_1\\x_2\\x_3\end{bmatrix}=[w_1x_1+w_2x_2+w_3x_3]=H(x)$$

$$[w_1 \ w_2 \ w_3]\begin{bmatrix}x_1\\x_2\\x_3\end{bmatrix}=[w_1x_1+w_2x_2+w_3x_3]=H(x)$$

$$[w_1 \ w_2 \ w_3]\begin{bmatrix}x_1\\x_2\\x_3\end{bmatrix}=[w_1x_1+w_2x_2+w_3x_3]=H(x)$$

$$\begin{bmatrix} w_{A1} & w_{A2} & w_{A3} \\ w_{B1} & w_{B2} & w_{B3} \\ w_{C1} & w_{C2} & w_{C3} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} w_{A1}x_1 + w_{A2}x_2 + w_{A3}x_3 \\ w_{B1}x_1 + w_{B2}x_2 + w_{B3}x_3 \\ w_{C1}x_1 + w_{C2}x_2 + w_{C3}x_3 \end{bmatrix} = \begin{bmatrix} \overline{Y_A} \\ \overline{Y_B} \\ \overline{Y_C} \end{bmatrix} = \begin{bmatrix} H_A(x) \\ H_B(x) \\ H_C(x) \end{bmatrix}$$

- Softmax의 필요성
  - 0과 1 사이의 값이 나왔으면 좋겠다.
  - A에 대한, B에 대한, C에 대한 y output의 Sum이 1이 되었으면 좋겠다
    (꼭 확률처럼)

  과 같은, Sigmoid를 썼을 때와 비슷한 이유로 softmax가 필요하다.
- Cost function : **Cross - Entropy**

  예측 값과 실제 값의 차이가 얼마인지 구하는 cost function까지 완성되어야 함!
  일단 생김새는 이렇다.

$$D(S, L) = -\sum_i L_i \log(S_i)$$

  이게 왜 설득력 있는 cost function이 되는가?(강의영상)
  요약하자면 맞으면 그냥 넘어가고 하나 틀리면 어마무시하게 값이 커지는 뭐 그런 거라고 한다.
- Logistic cost VS cross entropy
  - Logistic Cost

$$C : (H(x), y) = y \log(H(x)) - (1 - y) \log(1 - H(x))$$

  이 것이 사실상 Cross Entropy와 같다.
  H(x) == S, y ==L 이렇게 등치됨.
- Cost function (full version)

$$\mathcal{L} = \frac{1}{N} \sum_i D(S(wx_i + b)L_i)$$

  이렇게 다 더해서 평균을 하는 것까지 하면 Cost Function의 완성이다.
  - Minimalize (Gradient Descent)

    경사면을 미분하는 것은 다루지 않겠다. 암튼 gradient 값이 최소값을 향하게 해준다는 걸 알면 된다...

실습

- Settings
  - Package

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

  - For reproducibility / 같은 결과를 보장하기 위해서 -?

```
torch.manual_seed(1)
```

- Softmax 함수란?
  - 개념

    [1, 2, 3]을 일반적인 max에 넣는다면, [0, 0, 1]이 결과로 나올 것이다.
    하지만 softmax의 경우 좀더 명확하지 않은 '비율'로 max를 계산한다.
    앞선 예시와 같이 [1, 2, 3]을 넣었다면 softmax의 결과는 [0.0900, 0.2447, 0.6652] 뭐 이런 식

    셋을 다 합치면 1이 나오니, 어떻게 보면 이산확률분포처럼 생각할 수 있다.
  - 형태

$$P(class = i) = \frac{e^i}{\sum e^i}$$

  - Code

```
hypothesis = F.softmax(z, dim=0)
```

- Cross Entropy
  - 수식

$$H(P, Q) = -\mathbb{E}_{x \sim P(x)}[logQ(x)] = -\sum_{x \in X} P(x) \log(Q(x))$$

  - :: 설명이 있긴 한데 이해가 안 간다... 추가 조사가 필요함
  - Cross Entropy Loss **(Low - Level)**

$$L = -\frac{1}{N} \sum -y \log(\hat{y})$$

    y hat은 예측한 확률값, y는 실제 확률값(0 또는 1)

```
z = torch.rand(3, 5, requires_grad = True)
# softmax의 예시로 그냥 랜덤한 수.
# 3X5니깐 class는 5개, sample은 3개
hypothesis = F.softmax(z, dim = 1)
# dim=1: --> 방향으로 softmax를 하라.

# 예를 드는 거니깐 정답도 임의로 생성하자.
y = torch.randint(5,(3,)).long()

y_one_hot = torch.zeros_like(hyothesis)     # H와 같은 크기(3,5)의 텐서 선언
y_one_hot.scatter_(1,y.unsqueeze(1),1)
# 첫 parameter는 dim을 나타내고, 마지막 parameter는 어떤 값을 넣을지 정하는 것.
# 참고로 unsqueeze의 parameter도 dim. 명시한 dim 방향에 따라 차원 추가함. 3 -> (3,1)

# 결과로 쉽게 말하자면, 원래 [0, 2, 1]이던 걸 이용해
# [[1, 0, 0, 0, 0],
#  [0, 0, 1, 0, 0],
#  [0, 1, 0, 0, 0]]
# 으로 바꾸는 뭐 그런 과정이다.
# 암튼 이렇게 정답 벡터를 임의로 만들어냈다.

cost = (y_one_hot * -torch.log(hypothesis)).sum(dim=1).mean()
# |y_one_hot|== (3,5) and |hypothesis| == (3,5)
```

```
# sum(dim=1)을 해주니 둘을 곱한 값 (3,5)를 -> 방향으로 합해줌. 결과로 (3,)
# 그리고 그걸 평균냄 (3,) --> Scalar
```

- PyTorch 제공 함수를 이용

```
# Low level
torch.log(F.softmax(z,dim=1))

# High level
F.log_softmax(z, dim=1)

# 둘이 동일한 값을 낸다는 걸 이용해서 High level스럽게 수정을 해보자.

#기존
hypothesis = F.softmax(z, dim = 1)
cost = (y_one_hot * -torch.log(hypothesis)).sum(dim=1).mean()
# 1차 수정
cost = (y_one_hot * -F.log_softmax.sum(dim=1).mean()
# 2차 수정
cost = F.nll_loss(F.log_softmax(z , dim=1), y)
# 3차 수정
cost = F.cross_entropy(z, y)
```

저기서 nll은 Negative Log Likelihood의 약자다.

보통은, 특히 뉴럴 네트워크 이론에선 prediction 단계에서 확률값을 알아야할 때가 있어서,
3차 수정에서 쓴 Cross Entropy 함수를 쓰면 곤란할 수 있다.

따라서 사용자의 판단에 따라서 알맞은 것으로 쓰면 되겠다.

- Training
  - Training Set

```
x_train = [[1, 2, 1, 1],
          [2, 1, 3, 2],
          [3, 1, 3, 4],
          [4, 1, 5, 5],
          [1, 7, 5, 5],
          [1, 2, 5, 6],
          [1, 6, 6, 6],
          [1, 7, 7, 7]]                # |x_train| = (m, 4)
                                       #  즉 sample 개수 m, Class개수 4
y_train = [2, 2, 2, 1, 1, 1, 0, 0]     # |y_train| = (m)
# One Hot vector로 나타냈을 때 1이 있는 자리의 index
x_train = torch.FloatTensor(x_train)
y_train = torch.LongTensor(y_train)    # y_train이 Long형이어야 F.cross_entropy 함수의 parameter로 들어갈 수 있음.
```

  - 모델 초기화

```
W = torch.zeros((4,3), requires_grad = True)
b = torch.zeros(1, requires_grad = True)
```

  - Optimizer 설정

```
optimizer = optim.SGD([W,b], lr = 0.1)
```

  - 학습 (1) - Low Level

```
nb_epochs = 1000
for epoch in range(nb_epochs+1):

    # cost 계산
    hypothesis = F.softmax(x_train.matmul(W) + b, dim=1)
    y_one_hot = torch.zeros_like(hypothesis)
    y_one_hot.scatter_(1, y_train.unsqueeze(1),1)
    cost = (y_one_hot * -torch.log(hypothesis)).sum(dim=1).mean()

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

    # 로그 출력
    if epoch % 100 == 0:
        print('Epoch {:4d}/{} Cost : {:6f}'.format(epoch, nb_epochs, cost.item()))
```

▼ 실행결과

Epoch 0/1000 Cost: 1.098612
Epoch 100/1000 Cost: 0.901535
Epoch 200/1000 Cost: 0.839114
Epoch 300/1000 Cost: 0.807826
Epoch 400/1000 Cost: 0.788472
Epoch 500/1000 Cost: 0.774822
Epoch 600/1000 Cost: 0.764449
Epoch 700/1000 Cost: 0.756191
Epoch 800/1000 Cost: 0.749398
Epoch 900/1000 Cost: 0.743671
Epoch 1000/1000 Cost: 0.738749

- 학습 (2) - with F.cross_entropy

```
nb_epochs = 1000
for epoch in range(nb_epochs+1):

    # cost 계산
    z = x_train.matmul(W) + b
    cost = F.cross_entropy(z, y_train)
    # One hot vector 만드는 과정이 생략된 걸 볼 수 있다

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

    # 로그 출력
    if epoch % 100 == 0:
        print('Epoch {:4d}/{} Cost : {:6f}'.format(epoch, nb_epochs, cost.item()))
```

▼ 실행결과

```
Epoch    0/1000 Cost: 1.098612
Epoch  100/1000 Cost: 0.761050
Epoch  200/1000 Cost: 0.689991
Epoch  300/1000 Cost: 0.643229
Epoch  400/1000 Cost: 0.604117
Epoch  500/1000 Cost: 0.568255
```

```
Epoch  600/1000 Cost: 0.533922
Epoch  700/1000 Cost: 0.500291
Epoch  800/1000 Cost: 0.466908
Epoch  900/1000 Cost: 0.433507
Epoch 1000/1000 Cost: 0.399962
```

실전

High level Implementation with nn.Module

```
# 클래스 정의
class SoftmaxClassifierModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(4, 3)   # 4개의 확률값을 받아 3개의 vector 생성

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

model = SoftmaxClassifierModel()    # 선언

# optimizer 설정
optimizer = optim.SGD(model.parameters(), lr = 0.1)
nb_epochs = 1000
for epoch in range(nb_epochs + 1):
    # H(X) 계산
    prediction = model(x_train)   # |x_train| = (m,4), |prediction| = (m,3)

    # cost 계산
    cost = F.cross_entropy(prediction, y_train)

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

    # 100번마다 로그 출력
    if epoch % 100 == 0:
        print('Epoch {:4d}/{}  Cost: {:.6f}'.format(epoch), nb_epochs, cost.item())
```

▼ 실행결과

```
Epoch    0/1000 Cost: 1.113673
Epoch  100/1000 Cost: 0.714719
Epoch  200/1000 Cost: 0.638072
Epoch  300/1000 Cost: 0.581891
Epoch  400/1000 Cost: 0.532020
Epoch  500/1000 Cost: 0.484810
Epoch  600/1000 Cost: 0.438817
Epoch  700/1000 Cost: 0.393323
Epoch  800/1000 Cost: 0.347979
Epoch  900/1000 Cost: 0.302978
Epoch 1000/1000 Cost: 0.261105
```

강조

Binary Classification의 경우:

Binary Cross Entropy (BCE)

Sigmoid

Multinomial Classification의 경우:

Cross Entropy(CE)
    Softmax

를 쓰는 걸 실전에서 주의하도록 하라.