Softmax Regression

2017010715허지혜

01. One-Hot Encoding

[1,0,0]

토끼

[0,1,0]

곰

[0,0,1]

공룡

원-핫 벡터
(One-Hot vector)

단어 집합의 크기를 벡터의 차원으로 하고,
표현하고 싶은 단어의 인덱스 = 1
그 외 다른 인덱스 = 0 을 부여하는 단어의 벡터 표현 방식.

01. One-Hot Encoding

5.1 순서가 없는 범주형 특성 인코딩하기

```
# 원핫인코딩 = LabelBinarizer
import numpy as np
from sklearn.preprocessing import LabelBinarizer, MultiLabelBinarizer
```

```
feature = np.array([["Texas"],["California"],["Texas"],["Delaware"],["Texas"]])
```

```
#원핫인코더를 만든다
one_hot = LabelBinarizer()
```

```
#특성을 원핫인코딩하기
one_hot.fit_transform(feature)
```

```
array([[0, 0, 1],
       [1, 0, 0],
       [0, 0, 1],
       [0, 1, 0],
       [0, 0, 1]])
```

```
#classes_ 속성 확인
one_hot.classes_
```

```
array(['California', 'Delaware', 'Texas'], dtype='<U10')
```

```
#원핫인코딩에서 원래대로 되돌리기
one_hot.inverse_transform(one_hot.transform(feature))
```

```
array(['Texas', 'California', 'Texas', 'Delaware', 'Texas'], dtype='<U10')
```

=

```
#판다스를 이용해서도 가능
import pandas as pd
pd.get_dummies(feature[:,0])
```

	California	Delaware	Texas
0	0	0	1
1	1	0	0
2	0	0	1
3	0	1	0
4	0	0	1

01. One-Hot Encoding

```
from konlpy.tag import
Okt okt=Okt()
token=okt.morphs("나는 자연어 처리를 배운다")
print(token)

['나', '는', '자연어', '처리', '를', '배운다']
```

```
word2index={}
for voca in token:
        if voca not in word2index.keys():
        word2index[voca]=len(word2index)
print(word2index)

{'나': 0, '는': 1, '자연어': 2, '처리': 3, '를': 4, '배운다': 5}
```

```
def one_hot_encoding(word, word2index):
        one_hot_vector = [0]*(len(word2index)) index=word2index[word]
        one_hot_vector[index]=1
        return one_hot_vector

one_hot_encoding("자연어",word2index)
# 토큰 입력시 토큰에 대한 원-핫 벡터를 만들어냄
```

[0, 0, 1, 0, 0, 0]

01. One-Hot Encoding

원-핫 인코딩의 무작위성

대부분의 다중 클래스 분류 문제가 각 클래스 간의 관계가 균등하다는 점에서 원-핫 벡터는 이러한 점을 표현할 수 있는 적절한 표현 방법.

1

2

3

균등

$$((1,0,0)-(0,1,0))^2=(1-0)^2+(0-1)^2+$$

$$((1,0,0)-(0,0,1))^2=(1-0)^2+(0-0)^2+(0-1)^2=2$$

정수 인코딩 사용시 MSE

실제값이 곰, 예측값이 토끼 $(2-1)^2=1$

실제값이 공룡, 예측값이 토끼 $(3-1)^2=4$

→ 곰이 공룡보다 토끼와 더 가깝다

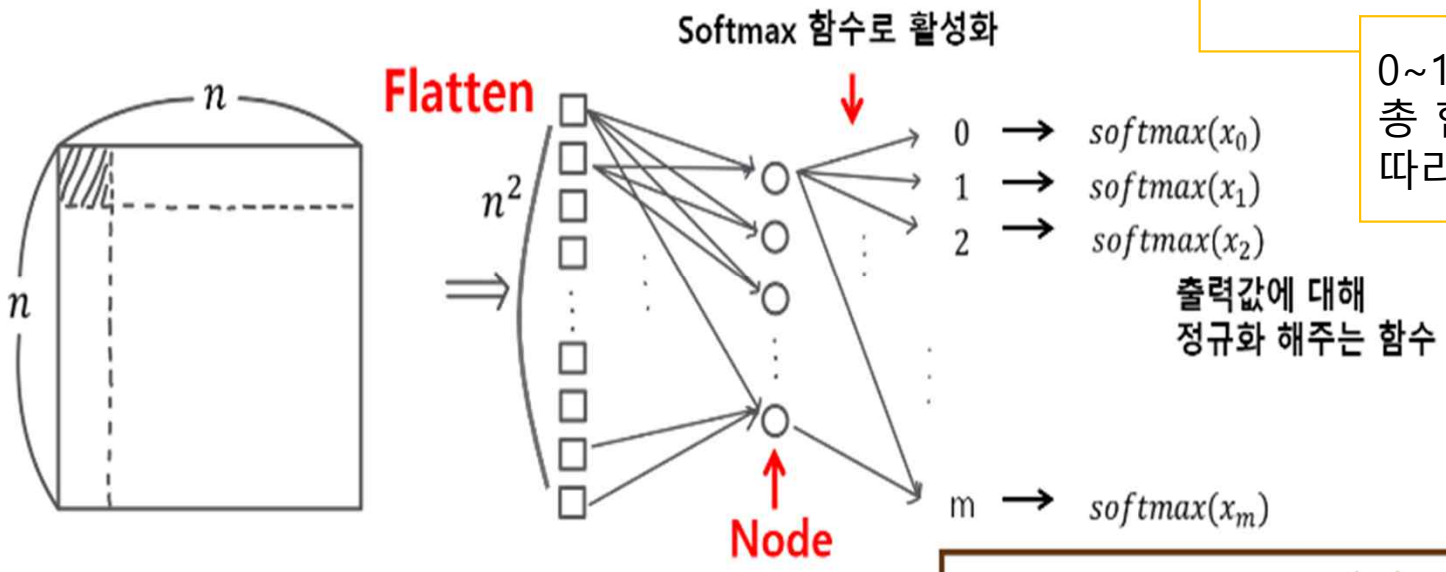
02. Softmax Regression

신경망 활성화 함수

FC층(완전 연결 계층) Fully connected -> 최종 분류를 위한 층

분류 문제를 풀 때 점수 벡터를
클래스 별 확률로 변환하기 위해
흔히 사용하는 함수

0~1 사이의 실수
총 합은 1
따라서 확률로 해석이 가능

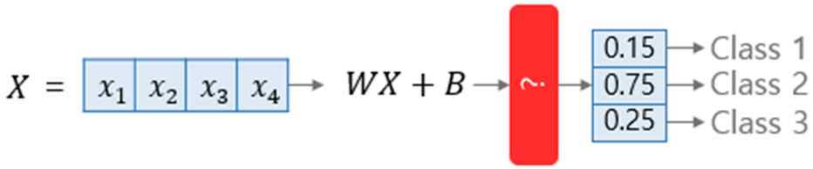


출력값에 대해
정규화 해주는 함수

$$softmax(x_i) = \frac{\exp(x_i)}{\sum_{j=1}^{n} \exp(x_j)}$$

```
def softmax(arr):
    m = np.argmax(arr)
    arr = arr - m
    arr = np.exp(arr)
    return arr / np.sum(arr)
```

02. Softmax Regression

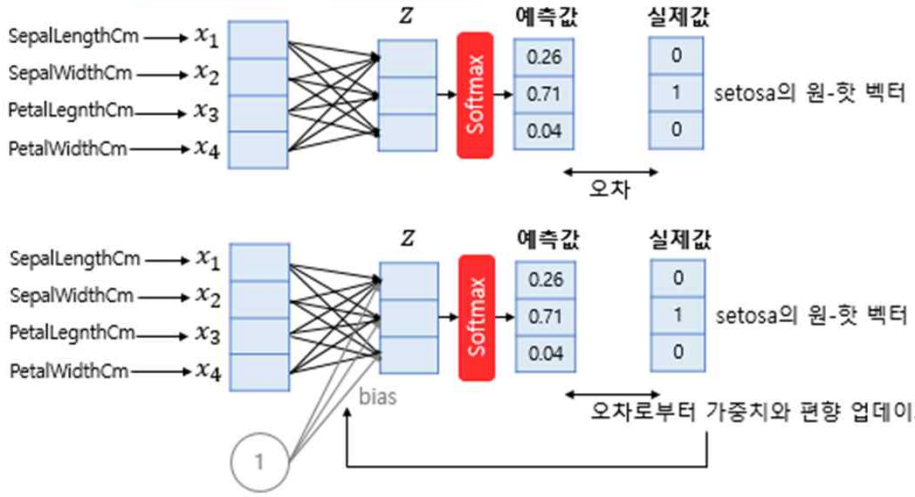


선택지 개수만큼 차원을 가지는 벡터를 만든다.

$$softmax(z) = [\frac{e^{z_1}}{\sum_{j=1}^{3} e^{z_j}} \quad \frac{e^{z_2}}{\sum_{j=1}^{3} e^{z_j}} \quad \frac{e^{z_3}}{\sum_{j=1}^{3} e^{z_j}}] = [p_1, p_2, p_3] = \hat{y} = \text{예측값}$$

$= P_{\text{토끼}}, P_{\text{곰}}, P_{\text{공룡}}$

토끼 곰 공룡

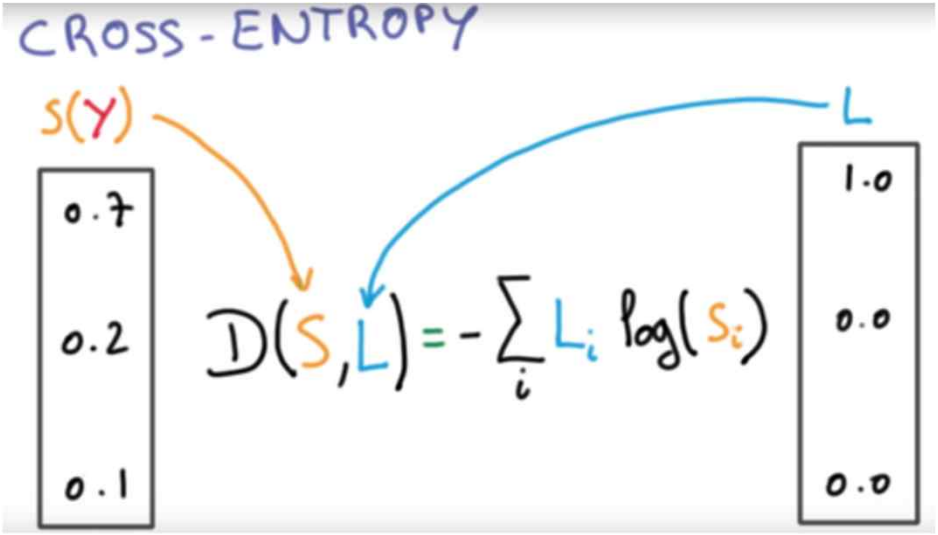


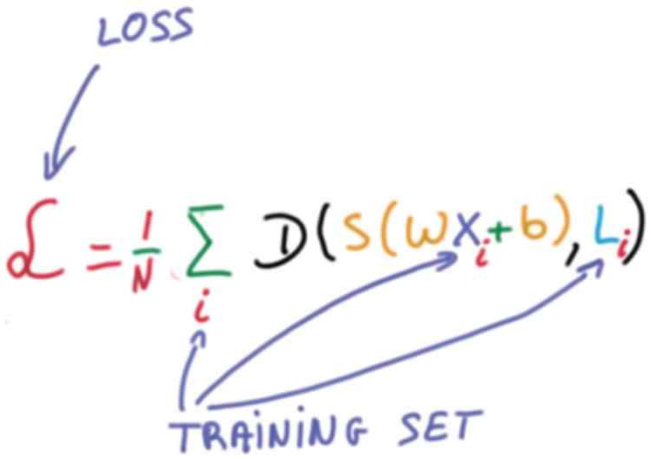
$$J(\Theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(\hat{p}_k^{(i)})$$

크로스 엔트로피 비용함수를 최소화

타깃 범주에 대해 낮은 확률을 예측하는 모델을 억제하므로 목적과 부합

02. Softmax Regression





02. Softmax Regression

필요한 패키지 불러오기

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
# conda install PyTorch -c PyTorch

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from IPython.display import display, Math, Latex
torch.manual_seed(1)
```

```
<torch._C.Generator at 0x1965a846370>
```

텐서 => 소프트맥스 함수 입력

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

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

```
tensor([0.0900, 0.2447, 0.6652])
```

```
hypothesis.sum()
```

```
tensor(1.)
```

비용 함수 구현

```
z = torch.rand(3, 5, requires_grad=True)
```

```
hypothesis = F.softmax(z, dim=1)
print(hypothesis)
```

각 샘플에 대한 적용이므로 dim =1

```
tensor([[0.2645, 0.1639, 0.1855, 0.2585, 0.1277],
        [0.2430, 0.1624, 0.2322, 0.1930, 0.1694],
        [0.2226, 0.1986, 0.2326, 0.1594, 0.1868]], grad_fn=<SoftmaxBackward>)
```

```
y = torch.randint(5, (3,)).long()
print(y)
```

```
tensor([0, 2, 1])
```

임의의 레이블에 대한 원핫 인코딩

```
y = torch.randint(5, (3,)).long()
print(y)
```

```
tensor([0, 2, 1])
```

```
# 모든 원소가 0의 값을 가진 3 × 5 텐서 생성
y_one_hot = torch.zeros_like(hypothesis)
y_one_hot.scatter_(1, y.unsqueeze(1), 1)
```

```
tensor([[1., 0., 0., 0., 0.],
        [0., 0., 1., 0., 0.],
        [0., 1., 0., 0., 0.]])
```

```
print(y.unsqueeze(1))
```

```
tensor([[0],
        [2],
        [1]])
```

```
print(y_one_hot)
```

```
tensor([[1., 0., 0., 0., 0.],
        [0., 0., 1., 0., 0.],
        [0., 1., 0., 0., 0.]])
```

02. Softmax Regression

1. F.softmax() + torch.log() = F.log_softmax()

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

```
tensor([[-1.3301, -1.8084, -1.6846, -1.3530, -2.0584],
        [-1.4147, -1.8174, -1.4602, -1.6450, -1.7758],
        [-1.5025, -1.6165, -1.4586, -1.8360, -1.6776]], grad_fn=<LogBackward>)
```

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

```
tensor([[-1.3301, -1.8084, -1.6846, -1.3530, -2.0584],
        [-1.4147, -1.8174, -1.4602, -1.6450, -1.7758],
        [-1.5025, -1.6165, -1.4586, -1.8360, -1.6776]],
       grad_fn=<LogSoftmaxBackward>)
```

2. F.log_softmax() + F.nll_loss() = F.cross_entropy()

```
cost = (y_one_hot * -torch.log(hypothesis)).sum(dim=1).mean()
print(cost)
```

```
tensor(1.4689, grad_fn=<MeanBackward0>)
```

```
# Low level
# 첫번째 수식
(y_one_hot * -torch.log(F.softmax(z, dim=1))).sum(dim=1).mean()
```

```
tensor(1.4689, grad_fn=<MeanBackward0>)
```

```
# 두번째 수식
(y_one_hot * - F.log_softmax(z, dim=1)).sum(dim=1).mean()
```

```
tensor(1.4689, grad_fn=<MeanBackward0>)
```

```
# High level
# 세번째 수식
F.nll_loss(F.log_softmax(z, dim=1), y)
```

```
tensor(1.4689, grad_fn=<NllLossBackward>)
```

```
# 네번째 수식
F.cross_entropy(z, y)
```

```
tensor(1.4689, grad_fn=<NllLossBackward>)
```

02. Softmax Regression

1) low-level softmax regression

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

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

```
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]]
y_train = [2, 2, 2, 1, 1, 1, 0, 0]
x_train = torch.FloatTensor(x_train)
y_train = torch.LongTensor(y_train)
```

8 x 4

8 x 1

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

```
torch.Size([8, 4])
torch.Size([8])
```

```
y_one_hot = torch.zeros(8, 3)
y_one_hot.scatter_(1, y_train.unsqueeze(1), 1)
print(y_one_hot.shape)
```

```
torch.Size([8, 3])
```

최종 레이블은 y_train을 원-핫 인코딩한 형태이므로 8x3 이다

```
# 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)
```

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

    # 가설
    hypothesis = F.softmax(x_train.matmul(W) + b, dim=1)

    # 비용 함수
    cost = (y_one_hot * -torch.log(hypothesis)).sum(dim=1).mean()

    # 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()
        ))
```

```
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
```

02. Softmax Regression

2) high-level softmax regression

```
# 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)

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)

    # 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()
        ))
```

```
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
```

3) softmax regreesion nn.Module

```
# 모델을 선언 및 초기화. 4개의 특성을 가지고 3개의 클래스로 분류. input_dim=4, output_dim=3.
model = nn.Linear(4, 3)
```

```
# 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)

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

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

    # 20번마다 로그 출력
    if epoch % 100 == 0:
        print('Epoch {:4d}/{} Cost: {:.6f}'.format(
            epoch, nb_epochs, cost.item()
        ))
```

```
Epoch    0/1000 Cost: 1.849513
Epoch  100/1000 Cost: 0.689894
Epoch  200/1000 Cost: 0.609259
Epoch  300/1000 Cost: 0.551218
Epoch  400/1000 Cost: 0.500141
Epoch  500/1000 Cost: 0.451947
Epoch  600/1000 Cost: 0.405051
Epoch  700/1000 Cost: 0.358733
Epoch  800/1000 Cost: 0.312911
Epoch  900/1000 Cost: 0.269521
Epoch 1000/1000 Cost: 0.241922
```

02. Softmax Regression

4) softmax regression class

```
class SoftmaxClassifierModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(4, 3) # Output이 3!

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

```
model = SoftmaxClassifierModel()
```

```
# 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)

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

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

    # 20번마다 로그 출력
    if epoch % 100 == 0:
        print('Epoch {:4d}/{} Cost: {:.6f}'.format(
            epoch, nb_epochs, cost.item()
        ))
```

```
Epoch    0/1000 Cost: 1.845720
Epoch  100/1000 Cost: 0.647150
Epoch  200/1000 Cost: 0.568868
Epoch  300/1000 Cost: 0.515699
Epoch  400/1000 Cost: 0.471727
Epoch  500/1000 Cost: 0.432486
Epoch  600/1000 Cost: 0.395879
Epoch  700/1000 Cost: 0.360507
Epoch  800/1000 Cost: 0.325227
Epoch  900/1000 Cost: 0.289217
Epoch 1000/1000 Cost: 0.254086
```

03. MNIST Classification

```
import torch
#conda install torchvision -c pytorch
import torchvision.datasets as dsets
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
import torch.nn as nn
import matplotlib.pyplot as plt
import random
```

```
USE_CUDA = torch.cuda.is_available() # GPU를 사용가능하면 True, 아니라면 False를 리턴
device = torch.device("cuda" if USE_CUDA else "cpu") # GPU 사용 가능하면 사용하고 아니면 CPU 사용
print("다음 기기로 학습합니다:", device)
```

```
다음 기기로 학습합니다: cpu
```

```
# for reproducibility
random.seed(777)
torch.manual_seed(777)
if device == 'cuda':
    torch.cuda.manual_seed_all(777)
```

```
# hyperparameters
training_epochs = 15
batch_size = 100
```

```
# MNIST dataset
mnist_train = dsets.MNIST(root='MNIST_data/',
                          train=True,
                          transform=transforms.ToTensor(),
                          download=True)

mnist_test = dsets.MNIST(root='MNIST_data/',
                         train=False,
                         transform=transforms.ToTensor(),
                         download=True)
```

03. MNIST Classification

```
# dataset loader
data_loader = DataLoader(dataset=mnist_train,
                                          batch_size=batch_size, # 배치 크기는 100
                                          shuffle=True,
                                          drop_last=True)
```

```
# MNIST data image of shape 28 * 28 = 784
linear = nn.Linear(784, 10, bias=True).to(device)
```

```
# 비용 함수와 옵티마이저 정의
criterion = nn.CrossEntropyLoss().to(device) # 내부적으로 소프트맥스 함수를 포함하고 있음.
optimizer = torch.optim.SGD(linear.parameters(), lr=0.1)
```

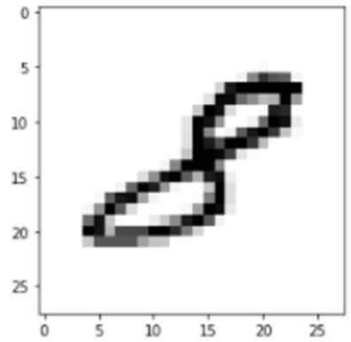
```
for epoch in range(training_epochs): # 앞서 training_epochs의 값은 15로 지정함.
    avg_cost = 0
    total_batch = len(data_loader)

    for X, Y in data_loader:
        # 배치 크기가 100이므로 아래의 연산에서 X는 (100, 784)의 텐서가 된다.
        X = X.view(-1, 28 * 28).to(device)
        # 레이블은 원-핫 인코딩이 된 상태가 아니라 0 ~ 9의 정수.
        Y = Y.to(device)

        optimizer.zero_grad()
        hypothesis = linear(X)
        cost = criterion(hypothesis, Y)
        cost.backward()
        optimizer.step()

        avg_cost += cost / total_batch

    print('Epoch:', '%04d' % (epoch + 1), 'cost =', '{:.9f}'.format(avg_cost))

print('Learning finished')
```

```
Epoch: 0001 cost = 0.534912527
Epoch: 0002 cost = 0.359308630
Epoch: 0003 cost = 0.331088185
Epoch: 0004 cost = 0.316574246
Epoch: 0005 cost = 0.307130307
Epoch: 0006 cost = 0.300207913
Epoch: 0007 cost = 0.294897288
Epoch: 0008 cost = 0.290830463
Epoch: 0009 cost = 0.287419587
Epoch: 0010 cost = 0.284589052
Epoch: 0011 cost = 0.281816214
Epoch: 0012 cost = 0.279919624
Epoch: 0013 cost = 0.277836859
Epoch: 0014 cost = 0.276022345
Epoch: 0015 cost = 0.274443209
Learning finished
```

03. MNIST Classification

```
# 테스트 데이터를 사용하여 모델을 테스트한다.
with torch.no_grad(): # torch.no_grad()를 하면 gradient 계산을 수행하지 않는다.
    X_test = mnist_test.test_data.view(-1, 28 * 28).float().to(device)
    Y_test = mnist_test.test_labels.to(device)

    prediction = linear(X_test)
    correct_prediction = torch.argmax(prediction, 1) == Y_test
    accuracy = correct_prediction.float().mean()
    print('Accuracy:', accuracy.item())

    # MNIST 테스트 데이터에서 무작위로 하나를 뽑아서 예측을 해본다
    r = random.randint(0, len(mnist_test) - 1)
    X_single_data = mnist_test.test_data[r:r + 1].view(-1, 28 * 28).float().to(device)
    Y_single_data = mnist_test.test_labels[r:r + 1].to(device)

    print('Label: ', Y_single_data.item())
    single_prediction = linear(X_single_data)
    print('Prediction: ', torch.argmax(single_prediction, 1).item())

    plt.imshow(mnist_test.test_data[r:r + 1].view(28, 28), cmap='Greys', interpolation='nearest')
    plt.show()
```

```
Accuracy: 0.8867999911308289
Label:  8
Prediction:  3
```



출처

https://ericabae.medium.com/ml-softmax-%EC%86%8C%ED%94%84%ED%8A%B8%EB%A7%A5%EC%8A%A4-%ED%95%A8%EC%88%98-7a8c0362b2a3
https://doongdoongeee.tistory.com/42
https://yganalyst.github.io/ml/ML_chap3-5/
https://m.blog.naver.com/PostView.nhn?blogId=sohyunst&logNo=221586366191&proxyReferer=https:%2F%2Fwww.google.com%2F

END