pytorch 시작

학습 목표

- pytorch를 활용하여 첫 모델을 구축해 본다.

목차

01 라이브러리 임포트

- 설치가 안되어 있을 경우, 설치하기
- pip install torch

목차로 이동하기

In [1]:
```
import torch

print(torch.__version__)
```

```
2.4.0+cu121
```

02 pytorch의 구성요소

목차로 이동하기

- torch : 텐서를 생성하는 라이브러리
- torch.autograd : 자동 미분 기능을 제공하는 라이브러리
- torch.nn : 신경망을 생성하는 라이브러리
- torch.multiprocessing : 병렬처리 기능을 제공하는 라이브러리
- torch.utils : 데이터 조작 등 유틸티 기능 제공
- torch.legacy : Torch로부터 포팅해온 코드

In [3]:
```
import numpy as np
import pandas as pd

import torch
import torch.nn as nn

from sklearn.preprocessing import RobustScaler
from sklearn.model_selection import train_test_split
```

03. 데이터 불러오기 및 데이터 나누기

data url : https://www.kaggle.com/datasets/yasserh/wine-quality-dataset

목차로 이동하기

In [4]:
```
wine = pd.read_csv("WineQT.csv")
wine
```

Out[4]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulph
0	7.4	0.700	0.00	1.9	0.076	11.0	34.0	0.99780	3.51	
1	7.8	0.880	0.00	2.6	0.098	25.0	67.0	0.99680	3.20	
2	7.8	0.760	0.04	2.3	0.092	15.0	54.0	0.99700	3.26	
3	11.2	0.280	0.56	1.9	0.075	17.0	60.0	0.99800	3.16	
4	7.4	0.700	0.00	1.9	0.076	11.0	34.0	0.99780	3.51	
...	...	...	...	...	...	...	...	...	...	
1138	6.3	0.510	0.13	2.3	0.076	29.0	40.0	0.99574	3.42	
1139	6.8	0.620	0.08	1.9	0.068	28.0	38.0	0.99651	3.42	
1140	6.2	0.600	0.08	2.0	0.090	32.0	44.0	0.99490	3.45	
1141	5.9	0.550	0.10	2.2	0.062	39.0	51.0	0.99512	3.52	
1142	5.9	0.645	0.12	2.0	0.075	32.0	44.0	0.99547	3.57	

1143 rows × 13 columns

In [5]:
```
wine.columns
```

Out[5]:
```
Index(['fixed acidity', 'volatile acidity', 'citric acid', 'residual sugar',
       'chlorides', 'free sulfur dioxide', 'total sulfur dioxide', 'density',
       'pH', 'sulphates', 'alcohol', 'quality', 'Id'],
      dtype='object')
```

In [6]:
```
wine['quality'].value_counts()
```

Out[6]:

quality	count
5	483
6	462
7	143
4	33
8	16
3	6

dtype: int64

데이터 나누기

In [7]:
```
df_train, df_val = train_test_split(wine, test_size=0.2)
```

In [8]:
```
X_train = df_train.drop(['quality', 'Id'], axis=1)
X_val = df_val.drop(['quality', 'Id'], axis=1)

y_train = df_train['quality']
y_val = df_val['quality']
```

데이터 전처리 - RobustScaler를 이용

In [9]:
```
# Scaling
# RobustScaler는 중앙값(median)과 사분위 범위(interquartile range)를 사용하여 데
scaler = RobustScaler()

X_train = scaler.fit_transform(X_train)
X_val = scaler.transform(X_val)
```

- 데이터 자료형 (TORCH.TENSOR)
- https://pytorch.org/docs/stable/tensors.html

In [10]:
```
# To Tensor
X_train_ts = torch.FloatTensor(X_train)
X_val_ts = torch.FloatTensor(X_val)

y_train_ts = torch.LongTensor(y_train.values)
y_val_ts = torch.LongTensor(y_val.values)
```

In [11]:
```
# Hyperparameter
LR = 1e-3
N_EPOCH = 500
DROP_PROB = 0.3
```

04. 모델 구축 및 학습

목차로 이동하기

- 모델에는 2개의 은닉층(hidden layer), ReLU 활성화 함수
- Xavier 가중치 초기화를 사용.
- 과적합(overfitting)을 막기 위해 Dropout 레이어를 추가.

- nn.Linear(입력차원, 출력차원, bias = True)
  - bias가 만약 False이면 layer는 bias를 학습하지 않음. 기본값은 True
  - device는 CPU, GPU를 고르는 것.
  - dtype는 자료형의 타입 정하기
- nn.Dropout()
  - 드롭아웃은 신경망에서 유닛을 제거(또는 드롭아웃)하여 과적합될 가능성을 획기적으로 줄일 수 있는 것.

In [12]:
```
# Model
class DNN(nn.Module):
    def __init__(self):
        super(DNN, self).__init__()

        self.fc1 = nn.Linear(X_train_ts.shape[1], 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 32)
        self.fc4 = nn.Linear(32, 11)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(p=DROP_PROB)

        for m in self.modules():
            if isinstance(m, nn.Linear):
                nn.init.xavier_normal_(m.weight.data)

    # 실제로 데이터가 거치는 부분
    def forward(self, x):
        # 은닉층
        x = self.fc1(x)
        x = self.relu(x)
        x = self.dropout(x)

        # 은닉층
        x = self.fc2(x)
        x = self.relu(x)
        x = self.dropout(x)


        x = self.fc3(x)
        x = self.relu(x)
        x = self.dropout(x)

        output = self.fc4(x)
        return output
```

- torch.nn.CrossEntropyLoss : 다중 분류에 사용되는 손실함수

- 예제

```
loss = nn.CrossEntropyLoss()
...
out = loss(y_pred, y_true)
out.item()
```

In [13]:
```
model = DNN()

optimizer = torch.optim.Adam(model.parameters(), lr=LR)
loss_fn = nn.CrossEntropyLoss()
```

In [14]:
```
for epoch in range(1, N_EPOCH+1):
    model.train()
    out = model(X_train_ts)
    loss = loss_fn(out, y_train_ts)

    # backpropagation을 하기 전에 gradients를 zero로 만든다.
    optimizer.zero_grad()

    # 역전파 계산
    loss.backward()

    # 가중치 업데이트
    optimizer.step()

    acc = (torch.argmax(out, dim=1) == y_train_ts).float().mean().item()

    model.eval()
    with torch.no_grad():
        out_val = model(X_val_ts)
        loss_val = loss_fn(out_val, y_val_ts)
        acc_val = (torch.argmax(out_val, dim=1) == y_val_ts).float().mean().item

    if epoch % 20 == 0:
        print('Epoch : {:3d} / {}, Loss : {:.4f}, Accuracy : {:.2f} %, Val Loss
            epoch, N_EPOCH, loss.item(), acc*100, loss_val.item(), acc_val*100))
```

```
Epoch :  20 / 500, Loss : 1.6255, Accuracy : 46.61 %, Val Loss : 1.5950, Val Accu
racy : 50.66 %
Epoch :  40 / 500, Loss : 1.3145, Accuracy : 50.77 %, Val Loss : 1.2628, Val Accu
racy : 52.84 %
Epoch :  60 / 500, Loss : 1.1900, Accuracy : 51.09 %, Val Loss : 1.1004, Val Accu
racy : 53.71 %
Epoch :  80 / 500, Loss : 1.1305, Accuracy : 54.16 %, Val Loss : 1.0277, Val Accu
racy : 57.21 %
Epoch : 100 / 500, Loss : 1.0927, Accuracy : 56.56 %, Val Loss : 1.0007, Val Accu
racy : 56.77 %
Epoch : 120 / 500, Loss : 1.0480, Accuracy : 60.18 %, Val Loss : 0.9725, Val Accu
racy : 58.95 %
Epoch : 140 / 500, Loss : 1.0025, Accuracy : 61.05 %, Val Loss : 0.9539, Val Accu
racy : 58.95 %
Epoch : 160 / 500, Loss : 0.9932, Accuracy : 60.07 %, Val Loss : 0.9457, Val Accu
racy : 58.52 %
Epoch : 180 / 500, Loss : 0.9551, Accuracy : 60.28 %, Val Loss : 0.9433, Val Accu
racy : 58.08 %
Epoch : 200 / 500, Loss : 0.9599, Accuracy : 60.94 %, Val Loss : 0.9310, Val Accu
racy : 58.95 %
Epoch : 220 / 500, Loss : 0.9518, Accuracy : 63.24 %, Val Loss : 0.9290, Val Accu
racy : 58.95 %
Epoch : 240 / 500, Loss : 0.9426, Accuracy : 61.71 %, Val Loss : 0.9270, Val Accu
racy : 59.39 %
Epoch : 260 / 500, Loss : 0.9277, Accuracy : 61.60 %, Val Loss : 0.9260, Val Accu
racy : 60.70 %
Epoch : 280 / 500, Loss : 0.8949, Accuracy : 62.80 %, Val Loss : 0.9234, Val Accu
racy : 60.26 %
Epoch : 300 / 500, Loss : 0.9088, Accuracy : 62.25 %, Val Loss : 0.9151, Val Accu
racy : 59.83 %
Epoch : 320 / 500, Loss : 0.8989, Accuracy : 65.43 %, Val Loss : 0.9167, Val Accu
racy : 60.26 %
Epoch : 340 / 500, Loss : 0.8748, Accuracy : 63.24 %, Val Loss : 0.9241, Val Accu
racy : 60.26 %
Epoch : 360 / 500, Loss : 0.8705, Accuracy : 64.44 %, Val Loss : 0.9199, Val Accu
racy : 61.57 %
Epoch : 380 / 500, Loss : 0.8573, Accuracy : 64.77 %, Val Loss : 0.9179, Val Accu
racy : 62.01 %
Epoch : 400 / 500, Loss : 0.8462, Accuracy : 64.33 %, Val Loss : 0.9189, Val Accu
racy : 62.01 %
Epoch : 420 / 500, Loss : 0.8444, Accuracy : 64.22 %, Val Loss : 0.9258, Val Accu
racy : 62.01 %
Epoch : 440 / 500, Loss : 0.8170, Accuracy : 66.19 %, Val Loss : 0.9318, Val Accu
racy : 61.14 %
Epoch : 460 / 500, Loss : 0.8283, Accuracy : 66.08 %, Val Loss : 0.9256, Val Accu
racy : 62.01 %
Epoch : 480 / 500, Loss : 0.8244, Accuracy : 68.27 %, Val Loss : 0.9302, Val Accu
racy : 62.01 %
Epoch : 500 / 500, Loss : 0.8227, Accuracy : 65.65 %, Val Loss : 0.9325, Val Accu
racy : 62.01 %
```

결과적으로 60~65%의 검증 정확도 얻음.