베이스 코드 코드 해석

1.

```
import random
import os
import pandas as pd
import numpy as np
from tqdm.auto import tqdm
from sklearn.preprocessing import LabelEncoder

import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
```

▼

물론! 주어진 코드는 Python에서 데이터 분석 및 딥 러닝을 위해 자주 사용되는 라이브러리들을 임포트하는 코드입니다. 각각의 라이브러리와 모듈에 대해 설명드리겠습니다.

1. `import random`: 무작위 난수를 생성하기 위한 Python 기본 라이브러리인 `random`을 임포트합니다. 이를 통해 난수를 생성하거나 리스트 등에서 무작위로 항목을 선택할 수 있습니다.
2. `import os`: 파일 및 디렉토리와 관련된 작업을 수행하기 위한 Python 기본 라이브러리인 `os`를 임포트합니다. 파일 경로를 조작하거나 파일/디렉토리를 생성, 삭제, 이동하는 등의 작업을 수행할 수 있습니다.
3. `import pandas as pd`: 데이터 분석과 조작을 위한 라이브러리인 `pandas`를 임포트하고, 이를 `pd`라는 별칭으로 사용하겠다는 의미입니다. `pandas`는 표 형태의 데이터를 쉽게 다룰 수 있는 자료구조와 다양한 데이터 처리 기능을 제공합니다.
4. `import numpy as np`: 수치 계산과 배열 처리를 위한 라이브러리인 `numpy`를 임포트하고, 이를 `np`라는 별칭으로 사용하겠다는 의미입니다. `numpy`는 고성능 수치 계산 기능을 제공하여 배열과 행렬 연산을 효율적으로 수행할 수 있습니다.
5. `from tqdm.auto import tqdm`: 반복 작업의 진행 상태를 시각화하는 라이브러리인 `tqdm`의 `auto` 모듈에서 `tqdm` 함수를 가져옵니다. `tqdm`은 for 루프와 같은 반복문에서 진행

상태 바를 표시하여 작업이 얼마나 진행되었는지를 시각적으로 확인할 수 있습니다.

6. `from sklearn.preprocessing import LabelEncoder`: 사이킷런(Scikit-learn) 라이브러리의 `LabelEncoder` 클래스를 임포트합니다. 이 클래스는 문자열 레이블을 숫자로 인코딩하는 데 사용됩니다. 주로 범주형 데이터를 머신 러닝 알고리즘에 입력으로 제공하기 위해 사용됩니다.
7. `import torch`: 파이토치(PyTorch) 딥 러닝 라이브러리를 임포트합니다. 파이토치는 텐서 연산과 딥 러닝 모델 구축을 위한 기능을 제공합니다.
8. `import torch.nn as nn`: 파이토치의 신경망 모듈들이 정의되어 있는 `torch.nn` 모듈에서 `nn`이라는 별칭으로 신경망 관련 기능들을 가져옵니다. 이를 통해 다양한 뉴럴 네트워크 레이어를 사용할 수 있습니다.
9. `import torch.optim as optim`: 파이토치의 최적화 알고리즘들이 정의되어 있는 `torch.optim` 모듈에서 `optim`이라는 별칭으로 최적화 알고리즘 관련 기능들을 가져옵니다. 이를 통해 SGD(Stochastic Gradient Descent) 등의 최적화 방법을 사용할 수 있습니다.
10. `import torch.nn.functional as F`: 파이토치의 함수형 API들이 정의되어 있는 `torch.nn.functional` 모듈에서 `F`라는 별칭으로 함수형 API를 가져옵니다. 이를 통해 활성화 함수, 손실 함수 등을 사용할 수 있습니다.
11. `from torch.utils.data import Dataset, DataLoader`: 파이토치에서 데이터 로딩을 위한 클래스인 `Dataset`과 데이터 배치 처리를 위한 클래스인 `DataLoader`를 가져옵니다. `Dataset`을 상속받아 커스텀 데이터셋을 만들고, `DataLoader`를 사용하여 데이터를 미니 배치로 나누어 모델 학습 시에 사용할 수 있습니다.

코드 분석 시작

—

필요한 라이브러리를 임포트합니다

```
import random  # 난수 생성을 위한 라이브러리
import os  # 파일 및 디렉토리 작업을 위한 라이브러리
```

import pandas as pd # 데이터 분석을 위한 pandas 라이브러리
import numpy as np # 수치 계산을 위한 numpy 라이브러리
from tqdm.auto import tqdm # 반복 작업의 진행 상태를 시각화하는 tqdm 라이브러리
from sklearn.preprocessing import LabelEncoder # 문자열 레이블을 숫자로 인코딩하는 라이브러리

import torch # 파이토치 딥 러닝 라이브러리
import torch.nn as nn # 파이토치의 신경망 모듈들
import torch.optim as optim # 파이토치의 최적화 알고리즘들
import torch.nn.functional as F # 파이토치의 함수형 API
from torch.utils.data import Dataset, DataLoader # 파이토치의 데이터 로딩과 배치 처리를 위한 클래스들

학습 및 예측 파라미터를 설정하는 딕셔너리를 정의합니다

CFG = {
'TRAIN_WINDOW_SIZE': 90, # 학습에 사용할 윈도우 크기 (90일치)
'PREDICT_SIZE': 21, # 예측에 사용할 기간 크기 (21일치)
'EPOCHS': 10, # 학습 반복 횟수 (10번의 epoch로 학습)
'LEARNING_RATE': 1e-4, # 학습 속도 (0.0001)
'BATCH_SIZE': 4096, # 미니 배치 크기 (한 번에 4096개의 데이터를 처리)
'SEED': 41 # 난수 생성 시드 (랜덤한 결과를 재현하기 위해 41을 사용)
}

랜덤 시드를 설정하여 실험 결과를 재현 가능하도록 하는 함수 `seed_everything`와 해당 함수를 호출하여 시드를 고정하는 부분

def seed_everything(seed):
# Python의 random 라이브러리의 시드를 설정합니다.
random.seed(seed)
# 환경 변수 'PYTHONHASHSEED'를 지정된 시드로 설정합니다.
os.environ['PYTHONHASHSEED'] = str(seed)
NumPy 라이브러리의 시드를 설정합니다.

np.random.seed(seed)
# PyTorch의 CPU 연산 시드를 설정합니다.
torch.manual_seed(seed)
# PyTorch의 GPU 연산 시드를 설정합니다.
torch.cuda.manual_seed(seed)
# PyTorch의 cuDNN을 deterministic 모드로 설정하여 동일한 입력이 주어질 때마다 동일한 결과를 얻도록 합니다.
torch.backends.cudnn.deterministic = True
# PyTorch의 cuDNN을 benchmark 모드로 설정하여 연산 시간을 최적화합니다.
torch.backends.cudnn.benchmark = True

CFG 딕셔너리에 저장된 SEED 값을 사용하여 시드를 고정합니다.

seed_everything(CFG['SEED'])

train_data = pd.read_csv('./train.csv').drop(columns=['ID', '제품'])

스케일링을 위한 최대값과 최소값을 저장하는 딕셔너리를 초기화합니다.

scale_max_dict = {}
scale_min_dict = {}

훈련 데이터의 각 행에 대해 스케일링을 수행합니다. tqdm 라이브러리를 사용하여 진행 상태를 시각화합니다.

for idx in tqdm(range(len(train_data))):
# 현재 행의 최대값과 최소값을 구합니다.
maxi = np.max(train_data.iloc[idx, 4:]) # 4번째 열부터 마지막 열까지의 최대값을 구합니다.
mini = np.min(train_data.iloc[idx, 4:]) # 4번째 열부터 마지막 열까지의 최소값을 구합니다.

```
# 최대값과 최소값이 같은 경우 (전체 값이 동일한 경우)
if maxi == mini:
    # 해당 행의 데이터를 모두 0으로 설정합니다.
    train_data.iloc[idx, 4:] = 0
else:
    # 최대값과 최소값을 이용하여 스케일링을 수행합니다. (0과 1 사이의 값으로 정규화)
    train_data.iloc[idx, 4:] = (train_data.iloc[idx, 4:] - mini) / (maxi - mini)

# 해당 행의 스케일링을 위한 최대값과 최소값을 딕셔너리에 저장합니다.
scale_max_dict[idx] = maxi
scale_min_dict[idx] = mini
```

Label Encoding을 위해 LabelEncoder 객체를 생성합니다.

label_encoder = LabelEncoder()

Label Encoding을 수행할 범주형 변수들의 열 이름을 리스트로 저장합니다.

categorical_columns = ['대분류', '중분류', '소분류', '브랜드']

각 범주형 변수에 대해 Label Encoding을 수행합니다.

for col in categorical_columns:
# label_encoder를 현재 열의 데이터에 맞게 학습합니다.
label_encoder.fit(train_data[col])
# 학습된 label_encoder를 사용하여 현재 열의 데이터를 숫자로 변환합니다.
train_data[col] = label_encoder.transform(train_data[col])

주어진 일별 판매량 데이터를 학습 기간과 예측 기간 블럭의 세트로 변환하는 함수인 `make_train_data`를 정의하는 부분

def make_train_data(data, train_size=CFG['TRAIN_WINDOW_SIZE'], predict_size=CFG['PREDICT_SIZE']):
'''
학습 기간 블럭, 예측 기간 블럭의 세트로 데이터를 생성
data : 일별 판매량
train_size : 학습에 활용할 기간
predict_size : 추론할 기간
'''
num_rows = len(data)
window_size = train_size + predict_size

```
# 학습 데이터와 타겟 데이터를 담을 배열을 초기화합니다.
input_data = np.empty((num_rows * (len(data.columns) - window_size + 1), train_size, l
en(data.iloc[0, :4]) + 1))
target_data = np.empty((num_rows * (len(data.columns) - window_size + 1), predict_siz
e))

# 각 행별로 학습 데이터와 타겟 데이터를 생성합니다.
```

```
for i in tqdm(range(num_rows)):
    # encode_info에는 현재 행의 대분류, 중분류, 소분류, 브랜드 데이터가 저장됩니다.
    encode_info = np.array(data.iloc[i, :4])
    # sales_data에는 현재 행의 판매량 데이터가 저장됩니다.
    sales_data = np.array(data.iloc[i, 4:])

    # 해당 행의 판매량 데이터를 기간 블럭 단위로 나누어 학습 데이터와 타겟 데이터를 생성합니다.
    for j in range(len(sales_data) - window_size + 1):
        # window에는 기간 블럭에 해당하는 판매량 데이터가 저장됩니다.
        window = sales_data[j: j + window_size]
        # temp_data에는 기간 블럭에 해당하는 대분류, 중분류, 소분류, 브랜드 데이터와 판매량 데이터가 저
장됩니다.
        temp_data = np.column_stack((np.tile(encode_info, (train_size, 1)), window[:tr
ain_size]))
        # input_data에는 학습 데이터가 저장됩니다.
        input_data[i * (len(data.columns) - window_size + 1) + j] = temp_data
        # target_data에는 타겟 데이터가 저장됩니다.
        target_data[i * (len(data.columns) - window_size + 1) + j] = window[train_siz
e:]

return input_data, target_data
```

def make_predict_data(data, train_size=CFG['TRAIN_WINDOW_SIZE']):
'''
평가 데이터(Test Dataset)를 추론하기 위한 Input 데이터를 생성합니다.
data : 일별 판매량 데이터
train_size : 추론을 위해 필요한 일별 판매량 기간 (= 학습에 활용할 기간)
'''
num_rows = len(data)

```
# 추론을 위한 Input 데이터를 저장할 배열을 초기화합니다.
input_data = np.empty((num_rows, train_size, len(data.iloc[0, :4]) + 1))

# 각 행별로 추론을 위한 Input 데이터를 생성합니다.
for i in tqdm(range(num_rows)):
    # encode_info에는 현재 행의 대분류, 중분류, 소분류, 브랜드 데이터가 저장됩니다.
    encode_info = np.array(data.iloc[i, :4])
    # sales_data에는 현재 행에서 학습에 활용할 기간에 해당하는 판매량 데이터가 저장됩니다.
    sales_data = np.array(data.iloc[i, -train_size:])
```

```
    # 추론을 위한 Input 데이터를 생성합니다.
    window = sales_data[-train_size:]
    temp_data = np.column_stack((np.tile(encode_info, (train_size, 1)), window[:train_
size]))
    input_data[i] = temp_data

return input_data
```

학습 데이터 생성

train_input, train_target = make_train_data(train_data)

평가 데이터 생성

test_input = make_predict_data(train_data)

data_len = len(train_input) # 전체 학습 데이터의 개수를 구합니다.

검증 데이터 개수를 전체 학습 데이터 개수의 20%로 설정합니다.

val_size = int(data_len * 0.2)

검증 데이터를 전체 학습 데이터의 뒷부분에서 잘라냅니다.

val_input = train_input[-val_size:]
val_target = train_target[-val_size:]

학습용 데이터를 전체 학습 데이터에서 검증 데이터를 제외한 부분으로 설정합니다.

train_input = train_input[:-val_size]
train_target = train_target[:-val_size]

train_input.shape : (학습 데이터 개수, 학습 기간, 입력 데이터의 특성 수)
train_target.shape : (학습 데이터 개수, 예측 기간)
val_input.shape : (검증 데이터 개수, 학습 기간, 입력 데이터의 특성 수)
val_target.shape : (검증 데이터 개수, 예측 기간)
test_input.shape : (평가 데이터 개수, 학습 기간, 입력 데이터의 특성 수)

class CustomDataset(Dataset):
def **init**(self, X, Y):
# 데이터셋의 입력과 타겟을 인스턴스 변수로 저장합니다.
self.X = X
self.Y = Y

```
def __getitem__(self, index):
    # 주어진 인덱스에 해당하는 데이터를 반환합니다.
    # 입력 데이터와 타겟 데이터를 함께 반환합니다.
    if self.Y is not None:
        return torch.Tensor(self.X[index]), torch.Tensor(self.Y[index])
    # 만약 타겟 데이터가 없을 경우 입력 데이터만 반환합니다.
    return torch.Tensor(self.X[index])

def __len__(self):
    # 데이터셋의 크기(데이터 개수)를 반환합니다.
    return len(self.X)
```

학습 데이터로 DataLoader를 생성합니다.

train_dataset = CustomDataset(train_input, train_target)
train_loader = DataLoader(train_dataset, batch_size=CFG['BATCH_SIZE'], shuffle=True, num_workers=0)

검증 데이터로 DataLoader를 생성합니다.

val_dataset = CustomDataset(val_input, val_target)
val_loader = DataLoader(val_dataset, batch_size=CFG['BATCH_SIZE'], shuffle=False, num_workers=0)

모델 선언

class BaseModel(nn.Module):
def **init**(self, input_size=5, hidden_size=512, output_size=CFG['PREDICT_SIZE']):
super(BaseModel, self).**init**()
self.hidden_size = hidden_size
# LSTM 레이어를 정의합니다.
self.lstm = nn.LSTM(input_size, hidden_size, batch_first=True)
# Fully Connected 레이어를 정의합니다.
self.fc = nn.Sequential(
nn.Linear(hidden_size, hidden_size//2),
nn.ReLU(),
nn.Dropout(),
nn.Linear(hidden_size//2, output_size)
)

# 활성화 함수를 정의합니다.
self.actv = nn.ReLU()

```
def forward(self, x):
    # x shape: (B, TRAIN_WINDOW_SIZE, 5)
    batch_size = x.size(0)
    # LSTM 레이어를 위한 초기 hidden state와 cell state를 초기화합니다.
    hidden = self.init_hidden(batch_size, x.device)

    # LSTM 레이어를 통해 입력 데이터를 처리합니다.
    lstm_out, hidden = self.lstm(x, hidden)

    # 마지막 시퀀스의 출력만 사용합니다.
    last_output = lstm_out[:, -1, :]

    # Fully Connected 레이어를 통해 최종 예측 결과를 얻습니다.
    output = self.actv(self.fc(last_output))

    return output.squeeze(1)

def init_hidden(self, batch_size, device):
    # LSTM 레이어의 초기 hidden state와 cell state를 초기화합니다.
    return (torch.zeros(1, batch_size, self.hidden_size, device=device),
            torch.zeros(1, batch_size, self.hidden_size, device=device))
```

모델 학습

def train(model, optimizer, train_loader, val_loader, device):
model.to(device) # 모델을 지정한 디바이스(GPU 또는 CPU)로 이동합니다.
criterion = nn.MSELoss().to(device) # 평균 제곱 오차(Mean Squared Error, MSE) 손실 함수를 생성하고, 해당 디바이스로 이동합니다.
best_loss = 9999999 # 현재까지 최소 검증 손실 값을 저장할 변수를 초기화합니다. 초기

값은 매우 큰 값인 9999999로 설정합니다.
best_model = None # 최적의 모델을 저장할 변수를 초기화합니다.

```
for epoch in range(1, CFG['EPOCHS']+1):  # 주어진 epoch 수만큼 학습을 수행합니다.
    model.train()  # 모델을 학습 모드로 설정합니다.
    train_loss = []  # 각 미니배치에 대한 훈련 손실 값을 저장할 리스트를 초기화합니다.
    train_mae = []

    for X, Y in tqdm(iter(train_loader)):  # 학습용 데이터 로더를 이터레이션하면서 미니배치 단위로
학습을 진행합니다.
        X = X.to(device)  # X와 Y를 지정한 디바이스(GPU 또는 CPU)로 이동시킵니다.
        Y = Y.to(device)

        optimizer.zero_grad()  # 옵티마이저의 그라디언트를 초기화합니다.

        output = model(X)  # 모델의 예측 값을 얻습니다.
        loss = criterion(output, Y)  # 예측 값과 타겟 값 사이의 손실 값을 계산합니다.

        loss.backward()  # 역전파를 수행하여 그라디언트를 계산합니다.
        optimizer.step()  # 파라미터를 업데이트합니다.

        train_loss.append(loss.item())  # 현재 미니배치의 훈련 손실 값을 저장합니다.

    val_loss = validation(model, val_loader, criterion, device)  # 검증 데이터를 사용하여
 검증 손실 값을 계산합니다.
    print(f'Epoch : [{epoch}] Train Loss : [{np.mean(train_loss):.5f}] Val Loss : [{va
l_loss:.5f}]')  # 현재 epoch에서의 학습 손실과 검증 손실 값을 출력합니다.

    if best_loss > val_loss:  # 현재 epoch에서의 검증 손실 값이 이전 최소 검증 손실 값보다 작을 경
우 최적의 모델을 업데이트합니다.
        best_loss = val_loss
        best_model = model
        print('Model Saved')

return best_model  # 학습이 끝난 후 최적의 모델을 반환합니다.
```

def validation(model, val_loader, criterion, device):
model.eval() # 모델을 평가 모드로 설정합니다.
val_loss = [] # 각 미니배치에 대한 검증 손실 값을 저장할 리스트를 초기화합니다.

```
with torch.no_grad():  # 그라디언트 계산이 필요 없으므로, 역전파 비활성화를 위해 torch.no_grad()
 블록을 사용합니다.
    for X, Y in tqdm(iter(val_loader)):  # 검증용 데이터 로더를 이터레이션하면서 미니배치 단위로
 검증을 진행합니다.
        X = X.to(device)  # X와 Y를 지정한 디바이스(GPU 또는 CPU)로 이동시킵니다.
        Y = Y.to(device)
```

```
        output = model(X)  # 모델의 예측 값을 얻습니다.
        loss = criterion(output, Y)  # 예측 값과 타겟 값 사이의 손실 값을 계산합니다.

        val_loss.append(loss.item())  # 현재 미니배치의 검증 손실 값을 저장합니다.

return np.mean(val_loss)  # 모든 미니배치에 대한 검증 손실 값을 평균하여 반환합니다.
```

model = BaseModel() # 새로운 BaseModel 인스턴스를 생성합니다. 이 모델은 학습에 사용될 딥 러닝 모델을 나타냅니다.

optimizer = torch.optim.Adam(params=model.parameters(), lr=CFG["LEARNING_RATE"]) # Adam 옵티마이저를 생성합니다.

Adam 옵티마이저는 모델의 파라미터를 업데이트하는데 사용되며, 학습률(learning rate)은 CFG["LEARNING_RATE"] 변수로 지정한 값으로 설정합니다.

infer_model = train(model, optimizer, train_loader, val_loader, device)

생성한 모델을 주어진 학습용 데이터(train_loader)로 학습하고, 검증용 데이터(val_loader)를 사용하여 모델의 성능을 평가합니다.

학습이 끝난 후, 검증 데이터를 기준으로 성능이 가장 좋은 최적의 모델을 반환합니다.

이 최적의 모델은 infer_model 변수에 저장됩니다.