# 딥러닝(CNN)을 활용한 식물 잎 사진으로 질병 분류하기

# 디지털미래융합서비스협동과정

# 석사과정 최정철

# 0. 기초 셋팅

# 구글 드라이브 마운트(쥬피터노트북 환경에서는 gpu사용불가로 모델학습에서 오류)

import os

from google.colab import drive

drive.mount('/content/drive')

# 프로젝트 폴더 하위에 "dataset" 폴더가 만들어져있어야합니다.

os.chdir("/content/drive/MyDrive/jc\_deeplearning\_project/dataset")

# 압축풀기위해 데이터셋폴더로 지정

# "dataset"폴더안에 "dataset.zip"파일이 있어야합니다.

!unzip -qq "/content/drive/MyDrive/jc\_deeplearning\_project/dataset/dataset.zip"

# 데이터셋 압축해제(5분정도 소요),

import shutil

src\_path1 = '/content/drive/MyDrive/jc\_deeplearning\_project/dataset/dataset.zip'

dst\_path1 = '/content/drive/MyDrive/jc\_deeplearning\_project/dataset.zip'

shutil.move(src\_path1, dst\_path1)

# 기존 data셋 내에 있는 dataset.zip파일을 프로젝트 폴더로 옮기기(오류방지)

os.chdir("/content/drive/MyDrive/jc\_deeplearning\_project")

# 압축해제 완료후 작업폴더로 다시 이동

# 1-1 데이터 분할을 위한 폴더 생성

import os

import shutil

original\_dataset\_dir = './dataset'

# 원본데이터 셋이 위치한 경로를 지정

classes\_list = os.listdir(original\_dataset\_dir

# 해당폴더 하위에 있는 모든폴더 목록 가져옴

# 폴더목록 = 클래스목록 => class\_list로 저장

base\_dir = './splitted'

# 나눈데이터를 저장할 폴더를 생성

os.mkdir(base\_dir)

train\_dir = os.path.join(base\_dir, 'train'

# 분리후에 각 데이터를 저장할 하위폴더 생성(train, val, test)

os.mkdir(train\_dir)

validation\_dir = os.path.join(base\_dir, 'val')

os.mkdir(validation\_dir)

test\_dir = os.path.join(base\_dir, 'test')

os.mkdir(test\_dir)

for cls in classes\_list:

# train, validation, test 폴더 하위에 각각 클래스 목록 폴더를 재생성

    os.mkdir(os.path.join(train\_dir, cls))

    os.mkdir(os.path.join(validation\_dir, cls))

    os.mkdir(os.path.join(test\_dir, cls))

# 1-2 데이터 분할과 클래스별 데이터 수 확인

import math

for cls in classes\_list:

# for문을 통해 모든클래스에서 작업 반복

    path = os.path.join(original\_dataset\_dir, cls)

    fnames = os.listdir(path)

# path의치에 존재하는 모든 이미지 파일의 목록수를 frame 변수에 저장

    train\_size = math.floor(len(fnames) \* 0.6)

# 훈련, 검증, 테스트 셋트의 비율 지정(6:2:2)

    validation\_size = math.floor(len(fnames) \* 0.2)

    test\_size = math.floor(len(fnames) \* 0.2)

    train\_fnames = fnames[:train\_size]

# train 데이터에 해당하는 파일이름은 frain\_franmes로 저장

    print("Train size(",cls,"): ", len(train\_fnames))

    for fname in train\_fnames:

# 모든 train 데이터에 대해 for문의 내용 반복

        src = os.path.join(path, fname)

# 복사할 파일 경로지정

        dst = os.path.join(os.path.join(train\_dir, cls), fname)

# 복사한 후 저장할 파일 경로 지정

        shutil.copyfile(src, dst)

# scr의 경로에 해당하는 파일을 dst경로에 저장

    validation\_fnames = fnames[train\_size:(validation\_size + train\_size)]

# 반복

    print("Validation size(",cls,"): ", len(validation\_fnames))

    for fname in validation\_fnames:

        src = os.path.join(path, fname)

        dst = os.path.join(os.path.join(validation\_dir, cls), fname)

        shutil.copyfile(src, dst)

    test\_fnames = fnames[(train\_size+validation\_size):(validation\_size + train\_size +test\_size)]

    print("Test size(",cls,"): ", len(test\_fnames))

    for fname in test\_fnames:

        src = os.path.join(path, fname)

        dst = os.path.join(os.path.join(test\_dir, cls), fname)

        shutil.copyfile(src, dst)

# 1-3 학습을 위한 준비

import torch

import os

USE\_CUDA = torch.cuda.is\_available()

# GPU를 사용할수 있는지 확인(True - 사용, Fale - 불가)

DEVICE = torch.device("cuda" if USE\_CUDA else "cpu")

# GPU사용가능하면 CUDA사용, 아니면 CPU사용

BATCH\_SIZE = 256

# 배치사이즈 한번에 바꾸기위해 변수로 등록

EPOCH = 16

# 에포크횟수도 미리 변수로 등록

import torchvision.transforms as transforms

from torchvision.datasets import ImageFolder

transform\_base = transforms.Compose([transforms.Resize((64,64)), transforms.ToTensor()])

# transforms.Compose() : 이미지 전처리 및 Augmentation 매서드

# transforms.Resize((64,64)) : 사이즈 조정

# transforms.ToTensor() : 이미지를 텐서형태로 변환, 모든값을 0~1사이로 정규화

train\_dataset = ImageFolder(root='./splitted/train', transform=transform\_base)

# 데이터셋 불러오는 메서드

val\_dataset = ImageFolder(root='./splitted/val', transform=transform\_base)

# 검증데이터를 val에서 불러옴

from torch.utils.data import DataLoader

train\_loader = torch.utils.data.DataLoader(train\_dataset, batch\_size=BATCH\_SIZE, shuffle=True, num\_workers=4)

# 불러온 이미지 데이터를 주어진 조건에 따라 미니 배치 단위로 분리하는 역할 수행

# 셔플 = True로 하면 데이터의 순서가 섞여 모델이 학습할때 순서를 기억하는 것을 방지함

val\_loader = torch.utils.data.DataLoader(val\_dataset, batch\_size=BATCH\_SIZE, shuffle=True, num\_workers=4)

# 검증데이터도 동일하게 셋팅

# 2-1 모델 설계

import torch.nn as nn

import torch.nn.functional as F

import torch.optim as optim

class Net(nn.Module):

# nn.Module클래스 상속

    def \_\_init\_\_(self):

        super(Net, self).\_\_init\_\_()

# nn.Module 내에 있는 매서드 상속받아 사용

        self.conv1 = nn.Conv2d(3, 32, 3, padding=1)

# 첫번째 conv2d 정의(채널수,출력채널수,커널크기)

        self.pool = nn.MaxPool2d(2,2)

# maxPooling 정의 커널크기, 스트라이드

        self.conv2 = nn.Conv2d(32, 64, 3, padding=1)

# 입력채널 수 32, 출력채널수 64, 커널크기 3인 2d conv 정의

        self.conv3 = nn.Conv2d(64, 64, 3, padding=1)

# 입력채널 수 64, 출력채널수 64, 커널크기 3인 2d conv 정의

        self.fc1 = nn.Linear(4096, 512)

# 플래튼 이후에 사용될 첫번째 fully cennected 레이어 정의

        self.fc2 = nn.Linear(512, 33)

# 플래튼 이후에 사용될 두번째 fully cennected 레이어 정의

    def forward(self, x):

        x = self.conv1(x)

# 첫번째 conv2d를 이용해 conv연산진행후 feature map 생성

        x = F.relu(x)

# 렐루함수 적용

        x = self.pool(x)

# 맥스 풀링 적용

        x = F.dropout(x, p=0.25, training=self.training)

# 맥스풀링 결과값에 드랍아웃 적용(25%)

        x = self.conv2(x)

# 두번째 conv2d를 이용해 conv연산진행후 feature map 생성

        x = F.relu(x)

        x = self.pool(x)

        x = F.dropout(x, p=0.25, training=self.training)

        x = self.conv3(x)

# 세번째 conv2d를 이용해 conv연산진행후 feature map 생성

        x = F.relu(x)

        x = self.pool(x)

        x = F.dropout(x, p=0.25, training=self.training)

        x = x.view(-1, 4096)

# 생성된 feature map을 1차원으로 펼침(Flatten)

        x = self.fc1(x)

# Flatten 된 1차원 tensor를 fc1에 통과

        x = F.relu(x)

        x = F.dropout(x, p=0.5, training=self.training)

        x = self.fc2(x)

# 모델의 마지막 레이어

        return F.log\_softmax(x, dim=1)

# 마지막 레이어 33개 결과값에 소프트맥스 합수 적용

# 각 클래스에 속할 확률을 output값으로 출력

model\_base = Net().to(DEVICE)

# 정의한 cnn모델 net()의 새로운 객체를 생성

optimizer = optim.Adam(model\_base.parameters(), lr=0.001)

# 옵티마지저 = 아담, 러닝레이트 0,001

# 2-2 모델 학습을 위한 함수

def train(model, train\_loader, optimizer):

    model.train()

# 입력받은 모델을 학습모드로 설정

    for batch\_idx, (data, target) in enumerate(train\_loader):

# batch인덱스,(data, target) 형태로 for문 실행

        data, target = data.to(DEVICE), target.to(DEVICE)

        optimizer.zero\_grad()

# 옵티마이저 초기화

        output = model(data)

# 데이터를 모델에 입력하여 output 값 계산

        loss = F.cross\_entropy(output, target)

# 모델에서 계산한 output값인 예측값과 타겟값 사이의 loss 계산

        loss.backward()

# 계산한 loss값을 바탕으로 그레디언트값을 각 파라미터에 할당

        optimizer.step()

# 각 파라미터에 할당된 그레디언트 값을 이용해 모델의 파라미터 업데이트

# 2-3 모델 평가를 위한 함수

def evaluate(model, test\_loader):

    model.eval()

# 입력받은 모델을 평가모델로 설정

    test\_loss = 0

# 미니 배치별로 loss를 합산에서 저장할변수 생성

    correct = 0

# 올바르게 예측한 데이터 세는 변수 생성

    with torch.no\_grad():

# 모델을 평가 할때는 파라미터 업데이트를 하지 않아야됨, 업데이트 중단하는 메서드 생성

        for data, target in test\_loader:

# 데이터와 대응하는 label값 접근

            data, target = data.to(DEVICE), target.to(DEVICE)

            output = model(data)

# 데이터를 모델에 입력하여 output 계산

            test\_loss += F.cross\_entropy(output,target, reduction='sum').item()

# 모델에서 계산한 output값인 예측값과 타겟값 사이의 로스 계산

            pred = output.max(1, keepdim=True)[1]

# 모델에 입력된 test데이터가 33개의 클래스에 속할 각각의 확률값이 output으로 출력

# 이중 가장 높은 값을 가진 인덱스를 예측값으로 저장

            correct += pred.eq(target.view\_as(pred)).sum().item()

# target tensor의 구조를 pred tensor와 같은 모양으로 정렬

    test\_loss /= len(test\_loader.dataset)

# 모든 미니 배치에서 합한 loss값을 배치수로 나누어 미니배치마다 계산된 loss값의 평균을 구함

    test\_accuracy = 100. \* correct / len(test\_loader.dataset)

# 모든 미니배치에서 합한 정확도 값을 batch수로 나누어 미니 배치마다 계산된 정확도 값의 평균을 구함

    return test\_loss, test\_accuracy

# 측정한 test loss와 정확도를 반환

# 2-3 모델 학습 실행 및 결과 확인

import time

import copy

def train\_baseline(model ,train\_loader, val\_loader, optimizer, num\_epochs = 30):

    best\_acc = 0.0

# 가장 높은 모델 저장하는 변수 설정

    best\_model\_wts = copy.deepcopy(model.state\_dict())

# 가장 정확도가 높은 모델 저장하는 변수 설정

    for epoch in range(1, num\_epochs + 1):

        since = time.time()

# 에포크당 소요시간 측정위해 시각 저장

        train(model, train\_loader, optimizer)

# 모델 학습

        train\_loss, train\_acc = evaluate(model, train\_loader)

# 트레인 데이터 로스 정확도 계산

        val\_loss, val\_acc = evaluate(model, val\_loader)

# 검증데이터 로스 정확도 계산

        if val\_acc > best\_acc:

# 에포크에서 최고 정확도 저장

            best\_acc = val\_acc

            best\_model\_wts = copy.deepcopy(model.state\_dict())

        time\_elapsed = time.time() - since                                # 에포크 소요시간 계산

        print('-------------- epoch {} ----------------'.format(epoch))

        print('train Loss: {:.4f}, Accuracy: {:.2f}%'.format(train\_loss, train\_acc))

# 트레인 로스 정확도 출력

        print('val Loss: {:.4f}, Accuracy: {:.2f}%'.format(val\_loss, val\_acc))

# 검증 로스 정확도 출력

        print('Completed in {:.0f}m {:.0f}s'.format(time\_elapsed // 60, time\_elapsed % 60))

# 에포크당 소요시간 출력

    model.load\_state\_dict(best\_model\_wts)

# 최종적으로 정확도가 가장 높은 모델 불러온뒤 반환

    return model

base = train\_baseline(model\_base, train\_loader, val\_loader, optimizer, EPOCH)

# baseline 모델 학습

torch.save(base,'baseline.pt')

# 학습된 모델을 저장

# 4-1 베이스라인 모델 평가를 위한 전처리하기

transform\_base = transforms.Compose([transforms.Resize([64,64]),transforms.ToTensor()])

test\_base = ImageFolder(root='./splitted/test',transform=transform\_base)

test\_loader\_base = torch.utils.data.DataLoader(test\_base, batch\_size=BATCH\_SIZE, shuffle=True, num\_workers=4)

# 4-2 베이스라인 모델 성능 평가하기

baseline=torch.load('baseline.pt')

baseline.eval()

test\_loss, test\_accuracy = evaluate(baseline, test\_loader\_base)

print('baseline test acc:  ', test\_accuracy)