오렌지 질병 분류를 위한 VGG 및 ResNet 사전학습 모델의 성능 비교 연구

이름 : 서재석(Haebo)
소속 : 카카오테크 부트캠프 생성형 AI 과정
날짜 : 2025.03.26

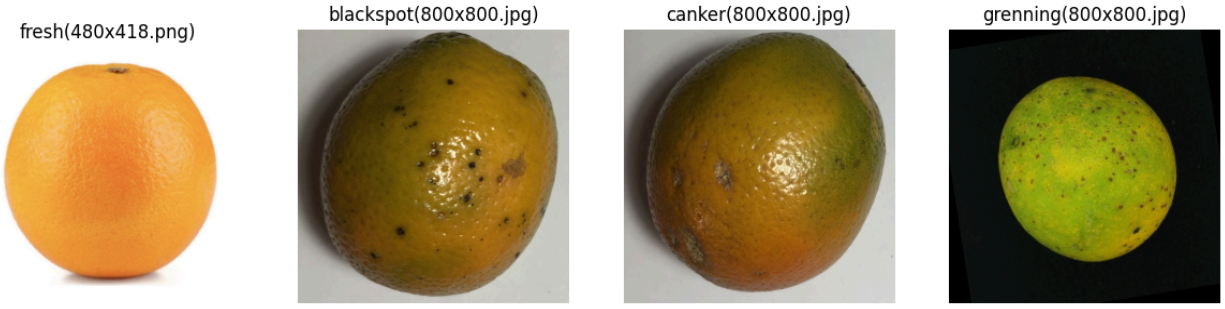
Abstract

본 연구는 오렌지 질병 유형을 이미지 데이터를 이용하여 정확하게 분류하는 데 초점을 맞추고 있으며, 이는 효과적인 치료 적용을 위해 매우 중요한 작업이다. 기존의 수동 분류 방식은 시간과 비용이 많이 소요되므로, 이미지 기반 분류를 통해 질병 유무와 특정 질병 유형을 판별하고자 한다. 연구 대상 질병은 시각적으로 뚜렷한 특징을 갖는 Citrus Black Spot, Citrus Canker, Citrus Greening Disease의 세 가지이다. 특징 추출 및 분류를 위해 VGG와 ResNet 사전 학습 모델을 미세 조정하고, 비교, 학습 및 평가를 수행하였다. 데이터 세트는 정상 및 질병에 걸린 오렌지 이미지로 구성되었으며, 학습/테스트 세트 비율은 약 10:1이다. 모델의 성능은 정확도 및 손실과 같은 지표를 사용하여 평가되었다. 모델 중에서 ResNet 101 모델이 98.99%의 가장 높은 정확도를 달성하여 뛰어난 성능을 보였으며, 이는 ResNet의 스킵 연결 구조가 깊은 네트워크에서 발생할 수 있는 기울기 소실 문제를 효과적으로 완화하기 때문으로 분석된다.

1. 서론

이 연구의 목적은 이미지 데이터 만으로 오렌지의 질병 유형을 정확하게 분류해 내는 데에 있다. 질병의 종류를 정확하게 분류해 내는 것은 정확한 치료를 적용하기 위해 중요한 작업이다. 각각의 오렌지에 대해 직접 눈으로 분류하는 것도 가능하나, 그렇게 되면 많은 시간과 비용이 들어간다. 따라서 오렌지의 이미지만을 가지고 해당 오렌지 개체가 질병인지 아닌지, 더 나아가 어떤 유형의 질병인지까지 분류해 내는 것을 목표로 한다. 분류를 목표하는 질병의 종류는 Citrus Black Spot, Citrus Canker, Citrus Greening Disease 이다. 이 오렌지 질병들은 발병 여부가 시각적으로 구분되는 것이 특징이다. 이번 연구에서는 특징을 추출하고 분류하기 위한 사전학습 모델로 VGG와 ResNet 모델을 중점적으로 비교하여 학습 및 평가를 수행하였다.

Citrus류의 3대 질병은 시각적으로 뚜렷하게 구별된다. Citrus Black Spot은 과일 표면에 검거나 갈색의 불규칙한 반점을 형성하며, 초기에는 작은 점으로 시작하여 점차 확장되고 합쳐지면서 과일의 외관을 심각하게 손상시킨다. 잎에서는 황색 점으로 시작된 병변이 갈색 또는 검은색으로 변하며, 심한 경우 낙엽을 유발한다. Citrus Canker는 잎, 줄기, 과일에 코르크 모양의 융기된 병변을 특징으로 하며, 특히 병변 주변에 황색 테두리가 나타나는 것이 일반적이다. 과일의 경우, 융기된 반점은 더욱 크고 뚜렷하게 나타나며 상품 가치를 크게 떨어뜨린다. Citrus Greening Disease (HLB)는 잎의 황화 현상이 가장 두드러지며, 잎맥 주변부터 황색으로 변하기 시작하여 점차 잎 전체로 확산된다. 과일은 작고 불규칙하며, 녹색을 띠거나 불균일하게 착색되어 외관뿐만 아니라 맛과 과즙에도 심각한 영향을 미쳐 상품 가치를 상실하게 한다. (link)



(Figure 1-A) 오렌지 질병 데이터의 클래스 별 대표 이미지

VGG 모델은 ImageNet 데이터 기반의 사전학습 모델로, 네트워크 깊이를 16-19개 까지 증가시켜 정확도를 크게 향상 시켰다. 또한 3x3 크기의 작은 컨볼루션 필터를 사용하였고 작은 필터를 여러 겹 쌓음으로써 큰 필터와 동일한 성능을 내면서도 파라미터 수를 줄였다. 이러한 구조 덕분에 비선형성을 증가시켜 성능을 증가시켜 뛰어난 일반화 성능을 가질 수 있었다.(link) VGG 모델중에서 VGG16은 13개의 convolution layer와 3개의 FC(fully connected) layer로 구성되어 있으며, 이미지 분류와 객체 탐지 부분에서 단순함과 효율성을 가지면서 좋은 성능을 보이는 것으로 잘 알려져 있다. 그러나 138 million 개의 많은 파라미터로 학습 시간이 오래걸리며, 깊은 네트워크 층으로 인해 Gradients 문제를 야기할 가능성도 있다. (link)

이러한 VGG16 모델의 gradient 문제를 해결하고자 ResNets 모델이 개발되었다. 기존의 모델의 경우 back-propagation의 과정에서 개별적인 뉴런이 gradient 계산의 오차에 어느정도 기여하는 지를 계산하게 되는데, 이때 활성화 함수를 반복해서 적용하다보니 앞쪽 뉴런들의 가중치가 작아져서 가중치 업데이트가 되지 않는 문제가 있었다. (Vanishing Gradient) 이 문제는 단순히 batch normalization을 통해서도 해결할 수 있었으나, ResNet model은 layer의 활성화 함수를 다음 layer의 출력부분에 더하는 'skip connection' 아이디어를 제시했다. 이 구조는 네트워크가 성능에 기여하지 않는 layer를 건너뛸수 있도록 하여, gradient가 앞쪽 레이어로 직접 전달될수 있도록 하였다. 이를 통해 Gradient vanishing을 최소화 하면서 깊은 layer 구조를 가능하게 했다. (link)

2. 데이터셋

클래스 비율 확인

Orange diseases dataset(link)은 fresh(정상), citrus canker, black spot, greening citrus(질병)의 4가지 클래스로 구성된 오렌지 질병 분류 데이터셋이다. 데이터는 약 10:1 비율의 Train/Test 셋으로 분리된다. Train 셋은 black spot 184개(18.6%), canker 179개(18.1%), fresh 281개(28.3%), greening 347개(35.0%)로 총 991개로 구성된다. Test 셋은 black spot 22개(22.2%), canker 22개(22.2%), fresh 33개(33.3%), greening 22개(22.2%)로 총 99개로 구성되어, Train 셋에 비해 클래스별 데이터 분포가 비교적 균일하다.

이미지 형식 비교

그러나 이미지의 사이즈의 경우 차이가 있다. 3개의 disease type의 경우 800x800 사이즈의 .jpg 형식인 반면, nomal type의 경우 이미지 사이즈가 서로 다르고 .png 파일이다. 현재 모델을 돌리기 위해 입력 사이즈 변환 및 tensor 형태로의 변환을 적용하기 때문에 실행에서의 문제는 없지만, 모델의 성능에 이런 데이터의 특성이 어떠한 영향을 미칠지 알기 어렵다.

클래스(Class)	학습 데이터셋(Train Dataset)	테스트 데이터셋(Test Dataset)
fresh	281개 (28.35%)	33개 (33.33%)
blackspot	184개 (18.57%)	22개 (22.22%)
canker	179개 (18.06%)	22개 (22.22%)
grenning	347개 (35.02%)	22개 (22.22%)
총 데이터 수	991개	99개

(Figure 2-A) 데이터의 클래스별 이미지 개수 및 비율

3. 다중분류(4-Classes)모델 성능 비교

사용한 모델 설명

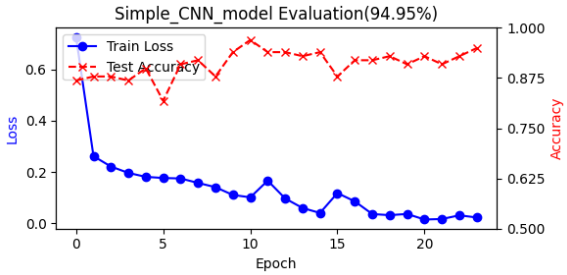
데이터의 기본 형태가 4개의 클래스이기 때문에 사전 학습 모델을 Fine-Tuning하여 성능을 비교하였다. 기본적인 분류 성능을 보기위해 기본 CNN 모델을 만들었고, VGG와 ResNet 모델을 베이스 모델(Fine-tuning의 대상이 되는 사전학습 모델)로 사용하였다. 이때 사전 학습 모델의 Feature extraction 부분의 구조와 가중치를 그대로 사용하기 위해 두 모델 모두 뒤쪽 Classifier layer 부분을 제외하고 Freeze(가중치가 업데이트 되지 않음) 한 뒤 모델 학습 및 평가를 진행하였다.

모델 학습 손실값 및 테스트 정확도

epochs는 30번으로 설정하고 Early Stop 함수를 추가하여 매번 epoch 마다 train loss값을 비교하여 3회 이상 떨어지지 않을 시 학습을 종료하는 방식을 선택했다.(patience=3, min_delta=0.001). 데이터가 작다 보니 쉽게 과적합 될것을 우려한 조치이다. 아래 자료는 모델 별 학습 및 평가 성능 결과이다.

Simple_CNN_model

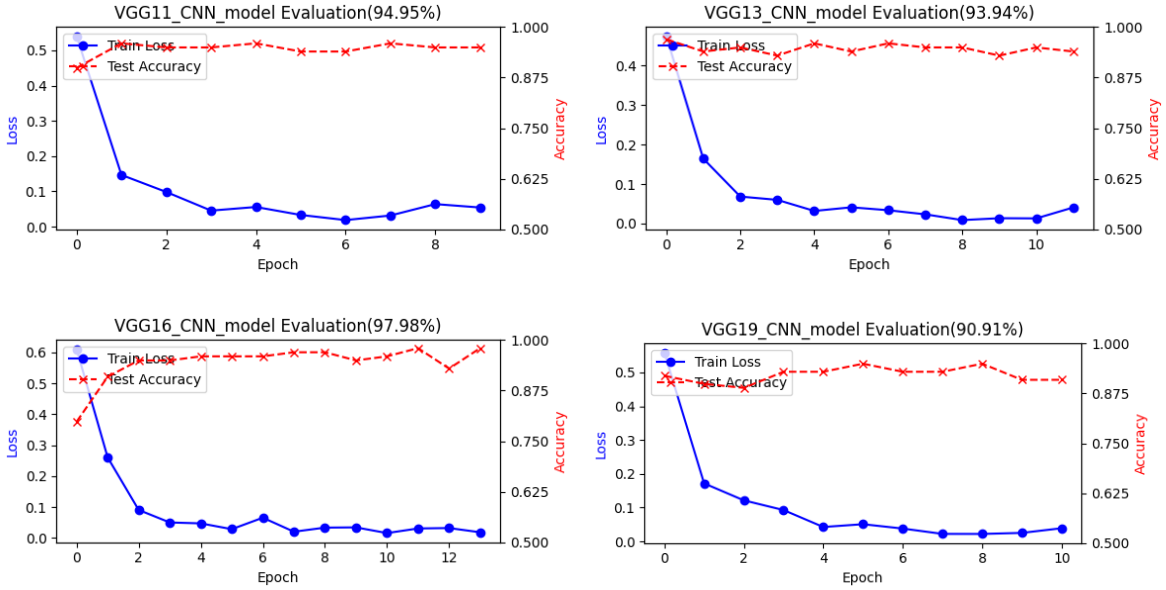
다른 사전학습 모델의 성능의 비교를 위해 3개의 Convolution Layer를 가진 CNN 모델을 제작했다. 성능의 변동성이 크다보니 모델 학습 및 평가 과정에 따라 달라지는 양상을 보인다. 여러번의 학습 및 평가에서 가장 좋은 성능을 보인 수행의 경우 94.95%의 높은 성능에서 조기 종료 되었다. 그러나 평균적으로는 85-90% 사이의 성능을 보인다.



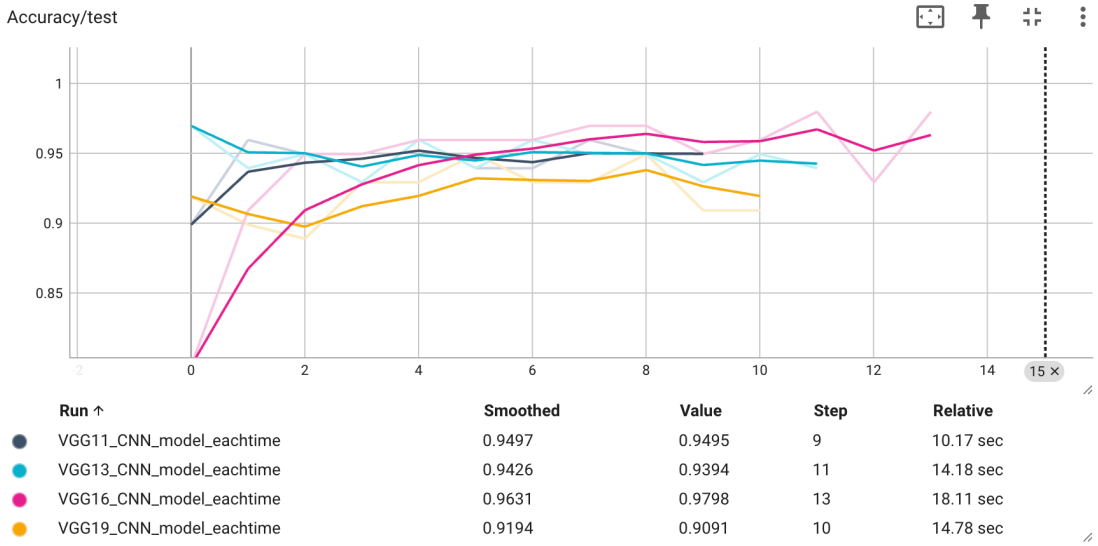
(Figure 3-A) Simple_CNN_model의 성능 평가 (Loss: 0.0228, Accuracy : 0.9495)

VGG model

좀 더 안정적으로 높은 성능을 얻기 위해 사전학습 모델인 VGG를 사용하여 부분 파인튜닝을 진행했다. 이번 연구에서는 기본적으로 VGG16, VGG19를 중심으로 진행했으며, 층의 수에 따른 성능의 비교를 위해 VGG11, VGG13 모델도 함께 사용했다. 단순 최종 정확도를 비교해보았을 때, VGG16 모델이 가장 97.98%의 가장 좋은 성능을 보였다. VGG11과 VGG13은 94% 내외의 정확도를 보였고, 특이적으로 VGG19의 성능이 떨어지는 것을 확인하였다.



(Figure 3-B) VGG_CNN_model의 성능 평가 (max Accuracy : 0.9798)

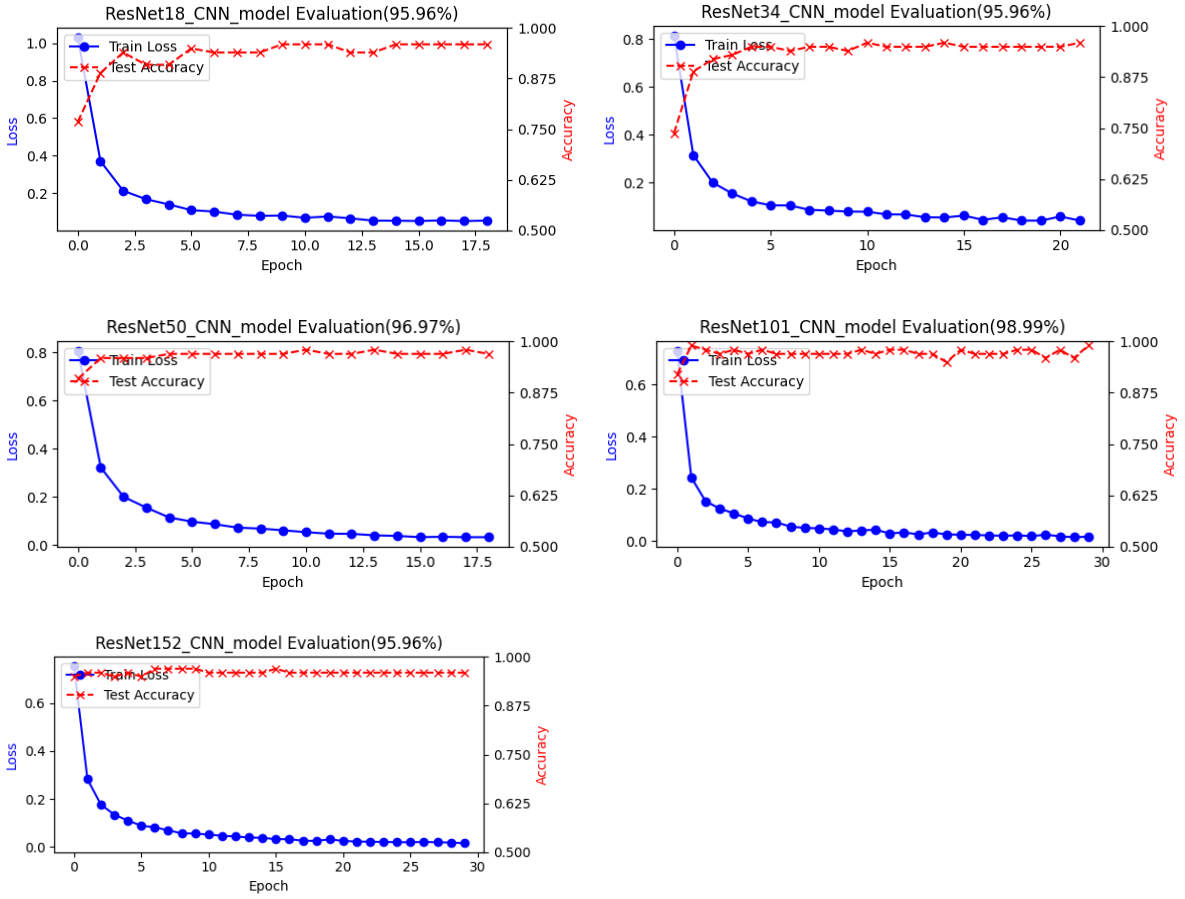


Run	Smoothed	Value	Step	Relative
VGG11_CNN_model_eachtime	0.9497	0.9495	9	10.17 sec
VGG13_CNN_model_eachtime	0.9426	0.9394	11	14.18 sec
VGG16_CNN_model_eachtime	0.9631	0.9798	13	18.11 sec
VGG19_CNN_model_eachtime	0.9194	0.9091	10	14.78 sec

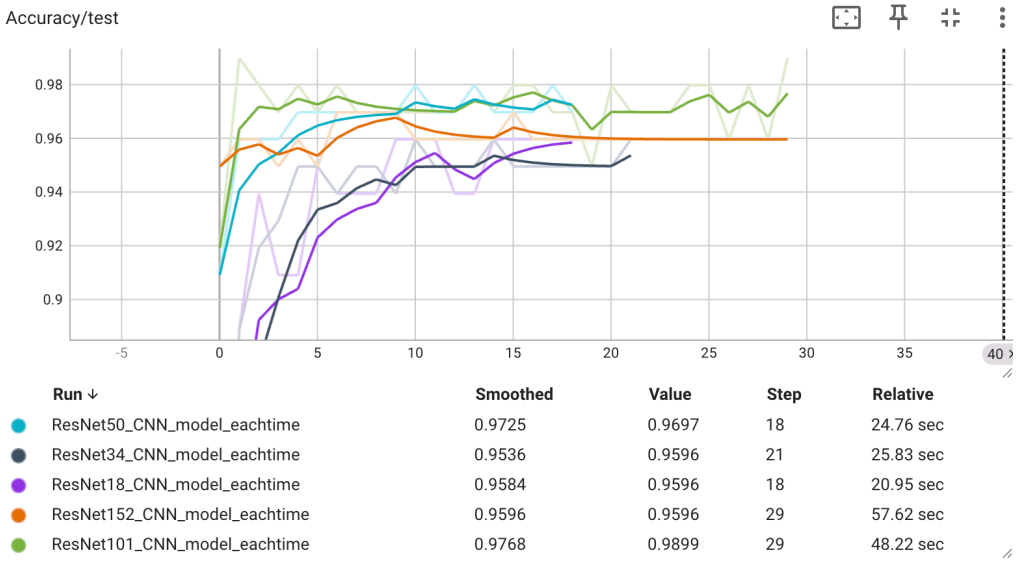
(Figure 3-C) VGG_CNN_model의 성능 비교

ResNet model

앞선 연구인 VGG 파인튜닝 모델을 활용한 분류에서, 초기 성능이 잘 나오기는 했지만 epoch가 증가함에 따라 loss값이 떨어지지 못하고 횡보 혹은 증가하여 빠르게 조기종료되는 모습을 보였다. 이것은 모델이 학습 데이터에 과적합 되었을 가능성을 시사한다. 이러한 결과를 바탕으로 과적합 문제를 해결하기 위해 ResNet 모델을 사용하였다. 모델 학습 및 검사 결과 ResNet 101 모델에서 최대 98.99%, ResNet 50 모델에서 96.97%의 높은 성능을 보인 것을 확인하였다. 다른 ResNet 모델에서도 95% 이상의 성능을 보이며, VGG16을 제외한 다른 VGG 모델들 보다 전반적으로 좋은 정확도가 나왔음을 알 수 있었다. 초기 성능은 VGG에 비해 낮은 경우가 많았지만 loss값이 꾸준히 감소하여 많은 epoch를 수행하면서 높은 정확도에 가까워지는 것을 확인할 수 있었다.



(Figure 3-D) ResNet_CNN_model의 성능 평가 (max Accuracy : 0.9899)

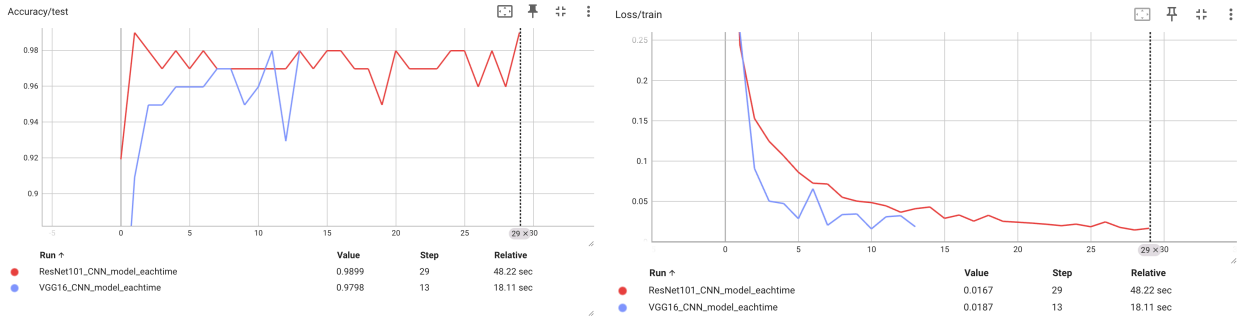


Run ↓	Smoothed	Value	Step	Relative
ResNet50_CNN_model_eachtime	0.9725	0.9697	18	24.76 sec
ResNet34_CNN_model_eachtime	0.9536	0.9596	21	25.83 sec
ResNet18_CNN_model_eachtime	0.9584	0.9596	18	20.95 sec
ResNet152_CNN_model_eachtime	0.9596	0.9596	29	57.62 sec
ResNet101_CNN_model_eachtime	0.9768	0.9899	29	48.22 sec

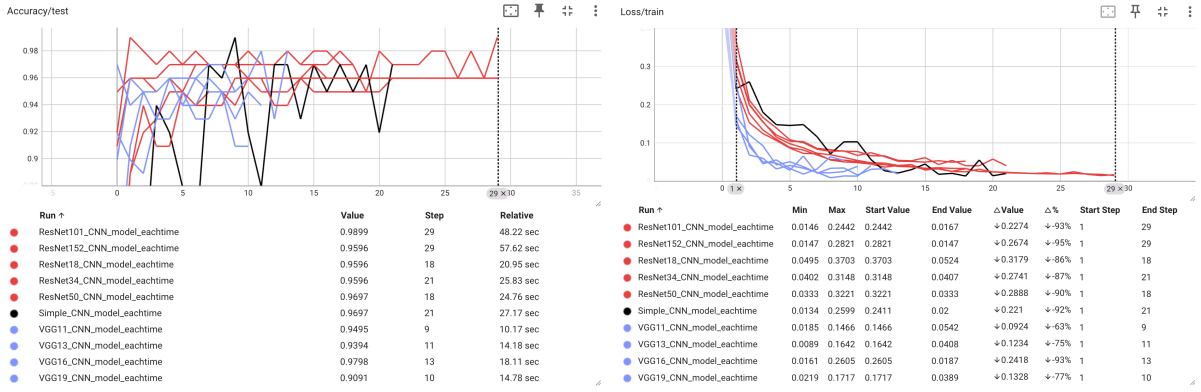
(Figure 3-E) ResNet_CNN_model 성능 비교

VGG16 vs ResNet 101

VGG와 ResNet 모델 중에서 성능이 가장 좋았던 두 모델을 비교해보았을 때, ResNet 101모델이 VGG16 모델에 보다 초기 정확도와 최종 정확도 모두 높은 것을 알수 있다. 다만 이것은 초기 가중치 초기화 단계 또는 초기 학습의 과정이 ResNet 101모델이 좀 더 잘 되어서 좋은 결과가 나왔을 가능성이 있다. 다만 두 모델의 Loss 값을 확인하였을 때, VGG의 모델의 경우가 ResNet 모델 보다 좀 더 빠르게 낮은 값으로 떨어지고 변동이 있는 것을 알수 있었다. ResNet과 VGG 모델 전체의 Loss값을 비교해봤을 때도 전반적으로 VGG 모델의 Loss값이 낮은데도 성능의 개선이 명확하지 않음을 생각해볼 때, VGG 모델의 경우 데이터가 학습데이터에 과적합(과도하게 맞춰짐)의 가능성을 시사한다.



(Figure 3-F) VGG 16과 ResNet 101 모델의 정확도(좌)와 손실값(우) 비교

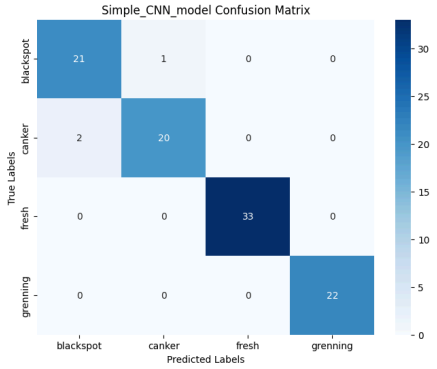


(Figure 3-G) 전체적인 VGG 과 ResNet 모델의 정확도(좌)와 손실값(우) 비교

4. **다중분류(4-Classes) 모델 클래스 예측 결과 비교**

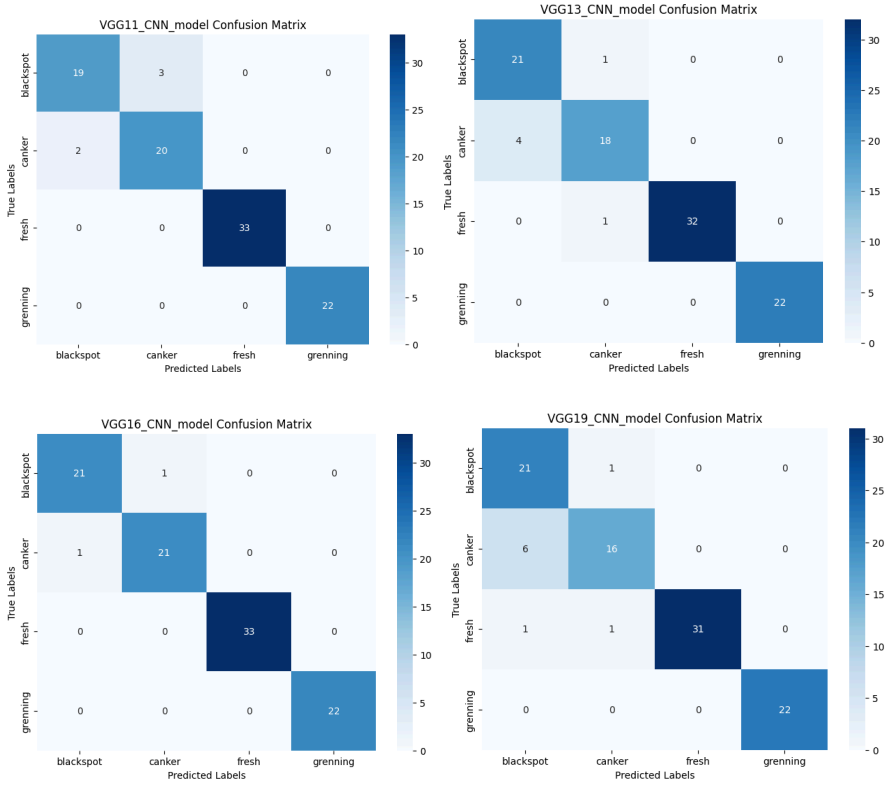
기본적으로 Confusion matrix를 통해 클래스 예측 결과를 확인하는 방식으로 진행하였다. 가장 기본적인 형태의 CNN 모델임에도 클래스를 잘 예측했다. 잘못 예측한 클래스를 보면 canker와 blacksplot의 한두개 데이터를 잘못 예측한 것을 확인했다. 이러한 사실은 클래스를 구분하는 특징이 복잡하고 고차원적이라기 보단 단순한 하고 저차원적인 시각적으로 구분되는 것일 가능성을 의미한다. 실제 이미지를 보았을 때도 클래스별로 시각적인 차이가 두드러지게 나타나며, 다시 말해 각각의 질병에 의한 오렌지 개체의 반응 결과가 시각적인 결과로서 구분되어 나타난다는 것이다. 이번 연구에서 모델들이 혼동한 클래스의 경우에는 이미지가 육안으로 확연하게 구분된다기 보다는 미세한 병변 패턴들의 차이(상대적으로 고차원적인 특징)가 있기에 두 클래스를 혼동하는 것으로 판단된다. 또한 실제가 fresh(normal type)인 경우에 예측을 disease type으로 한 경우(False Positive Error, FP)가 몇몇 있었다. 다만 실제가 disease 인 이미지를 normal type으로 예측한 경우(False Negative Error, FN)는 없다. 당연하게도 질병과 관련된 예측에서 False Negative Error는 질병 치료 또는 질병 인지에서의 심각한 문제를 야기한다. FP Error를 보인 모델은 VGG13(acc = 93.94%), VGG19(acc = 90.91%), ResNet 34(acc = 95.96%) 모델 이다.

Simple_CNN_model



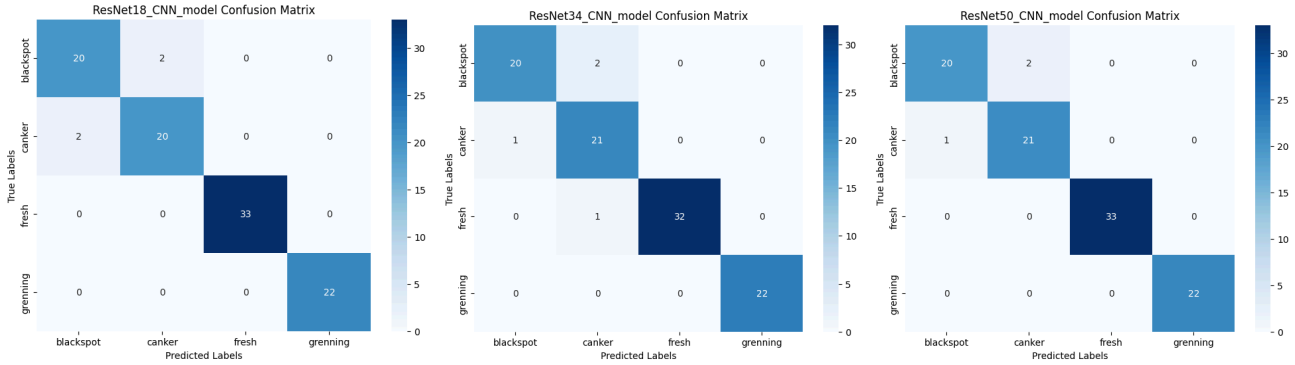
(Figure 4-A) Simple CNN model의 Test data 예측 결과 Confusion matrix

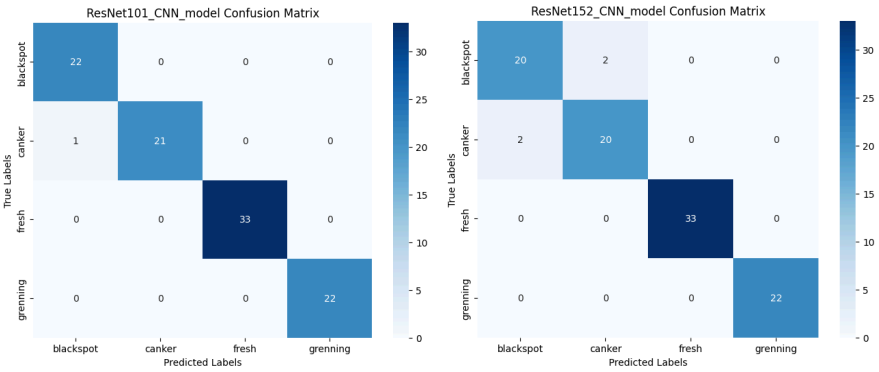
VGG_CNN_model



(Figure 4-B) VGG CNN model의 Test data 예측 결과 Confusion matrix

ResNet_CNN_model





(Figure 4-C) ResNet CNN model의 Test data 예측 결과 Confusion matrix

5. Discussion

이번 연구에서는 오렌지 질병 분류를 위해 VGG와 ResNet 사전학습 모델을 파인튜닝하여 성능을 비교 분석했다. 실험 결과, ResNet 101 모델이 98.99%의 최고 정확도를 달성하며 VGG16 모델보다 우수한 성능을 보였다. 이는 ResNet의 skip connection 구조가 깊은 네트워크 학습 시 발생할 수 있는 기울기 소실 문제를 효과적으로 해결하고, 네트워크의 깊이를 늘려도 안정적인 학습이 가능하게 했기 때문으로 분석된다. 그러나 VGG19 모델에서 성능 저하와 학습 과정에서 나타난 과적합 가능성은 깊은 네트워크 구조가 항상 성능 향상을 보장하지 않음을 시사한다. 또한 VGG 모델들의 Loss값이 ResNet 모델들 보다 낮았음에도 성능 개선이 명확하지 않았는데. 이는 VGG 모델이 학습 데이터에 과적합 되었을 가능성을 뒷한다.

Confusion Matrix를 분석한 결과, 모델들이 canker와 blackspot 클래스를 혼동하는 경향을 보였는데, 이는 질병 클래스 간의 병변 패턴차이가 모델의 분류 능력을 저해하는 요인으로 작용했을 가능성을 나타낸다. 나타난 오류는 False Positive Error로 정상 오렌지를 질병으로 오인하는 문제를 야기 할수 있으므로, 모델 개선을 한다면 이러한 오류를 줄이는 데 초점을 맞춰야 한다. 또한 데이터 셋 측면에서, normal type 이미지(.png)와 disease type(.jpg) 간의 파일 형식 및 크기 차이는 모델 성능에 영향을 미칠 수 있으므로, 데이터 셋의 불균형 해소 및 데이터 증강 기법 등을 통해 모델의 일반화 성능을 향상시키는 것이 필요하다.

결론적으로 이번연구에서는 데이터의 개수가 적은 오렌지 질병 데이터를 활용하여 ImageNet 데이터셋 기반의 사전학습 CNN 모델인 ResNet과 VGG 모델을 Fine-Tuning 하였다. 그리고 파인튜닝된 모델을 가지고 적은 데이터를 학습 시켰을 때도 충분한 분류 성능이 나오는 지를 확인했다. 두개의 사전학습 모델의 학습 및 평가 성능이 어떤식으로 나타나는지를 알아보았다. 후속 연구 방향으로는 효율화와 최신 모델 적용이 있다. 효율화를 위해 EfficientNetB0이나 GoogLeNet(inception) 모델 등을 파인튜닝하여 성능을 확인해보는 작업을 진행해볼수 있다. 치명적인 오류가 아니라면 효율적인 모델을 사용하여 진행하고, 데이터 전처리 또는 하이퍼파라미터 튜닝 등을 통해 성능을 개선한다면 더 좋은 분류 모델이 만들어 질것이다. 또한 동일한 데이터에 최신 모델을 적용해 볼수 있는데, 예를 들면 ViT(Visual Transformer, 이미지 분류에 transformer 구조를 적용한 모델) 등을 적용해보며 기존 모델과 성능을 비교해 보고 오렌지 질병 데이터와 같이 개수가 적은 데이터에서도 잘 작동하는 지 등을 종합적으로 평가해 볼수 있다.

6. Method

코랩 드라이브 마운트

데이터를 미리 동일한 구글 계정의 드라이브에 저장해 두고, 세션을 시작한 뒤 드라이브에 마운트 하여 경로를 지정하여 원하는 데이터에 접근할 수 있도록 하였다. 이때, 사용하는 하드웨어 가속기(CPU, CUDA) 상태를 확인하여 device 설정하는 코드와 CPU 코어수를 확인하여 이후 DataLoader의 num_workers나 prefetch_factor 옵션값을 설정하기 위해 변수에 할당하는 작업을 수행했다. 그 외는 이후 데이터 로드 및 저장을 위해 경로를 설정하는 코드이다.

```
from google.colab import drive
import json, os
import torch

# google drive에 mount
drive.mount('/content/drive', force_remount=True)
base_path = '/content/drive/MyDrive/Orange_CNN'
train_path = os.path.join(base_path, 'train')
test_path = os.path.join(base_path, 'test')
kaggle_path = os.path.join(base_path, 'kaggle.json')
with open(kaggle_path, 'r') as f:
   kaggle_config = json.load(f)
print(kaggle_config)

# 디바이스 선택
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f'using device :{device} ')

# CPU 코어수 확인
cpu_count = os.cpu_count()
print(f"CPU 코어 수: {cpu_count}")
```

데이터셋 정의

경로에 각각 저장된 Train과 Test 데이터셋을 불러와서 transform을 적용하고, 데이터 로더로 저장하는 과정을 거쳤다. 이때 입력 사이즈는 VGG와 ResNet에서 공통적으로 사용되는 (224, 224)로 설정하였다. 추가적인 과정으로 GPU의 성능을 최대한으로 사용하기 위해 데이터 셋을 tensor로 변환한 결과를 캐싱처리하고, 데이터 로더 옵션을 설정하는 등의 코드를 추가하였다.

```
import matplotlib.pyplot as plt
from PIL import Image
import torchvision
import torch
from torchvision.datasets import ImageFolder
from torch.utils.data import DataLoader

def orange_data(in_size : tuple = (224, 224), info_view : bool = True, img_view : bool = True):
   # test와 train 폴더 내 각각의 클래스의 이미지 개수 확인
   if info_view == True:
       ...
   # train 데이터에서 각 class별 첫번째 이미지(사이즈, 형식) 확인
   if img_view == True:
       ...
   # model에 적용하기 위해 데이터 형식 변환
   transform = torchvision.transforms.Compose([
       torchvision.transforms.Resize(in_size), # VGG, ResNet, Inception input size : (224, 224)
       torchvision.transforms.ToTensor(),
   ])

   # CUDA 환경인 경우 GPU 최적화 세팅
   if str(device) == 'cuda':
       train_set = ImageFolder(root = train_path, transform = None) # PiL file
       test_set = ImageFolder(root = test_path, transform = None)
       from tqdm import tqdm
       def tensor_transform(data_set, transform):
           images, labels = [], []
           for idx in tqdm(range(len(data_set)), desc = "caching"):
               ...
           return torch.utils.data.TensorDataset(images, labels)

       train_set_cache = tensor_transform(train_set, transform)
       test_set_cache = tensor_transform(test_set, transform)

       num_workers = cpu_count//2
       pin_memory = True
```

```
        prefetch_factor = cpu_count

        train_loader = DataLoader(train_set_cache, batch_size = 32,
                                  shuffle = True, num_workers= num_workers,
                                  pin_memory= pin_memory, prefetch_factor = prefetch_factor)
        test_loader = DataLoader(test_set_cache, batch_size = 32,
                                 shuffle = False, num_workers= num_workers,
                                 pin_memory= pin_memory, prefetch_factor = prefetch_factor)

    # CPU 환경인 경우 기본값 사용
    else:
        train_set = ImageFolder(root = train_path, transform = transform)
        test_set = ImageFolder(root = test_path, transform = transform)

        train_loader = DataLoader(train_set, batch_size = 32, shuffle = True)
        test_loader = DataLoader(test_set, batch_size = 32, shuffle = False)

    # 클래스 및 데이터 셋 정보 출력
    if info_view == True:
        ...
    print("\nusing dataset : orange_data")
    return train_set, test_set, train_loader, test_loader

orange_data(in_size = (224, 224), info_view = True, img_view= True)
```

```
### classes ###
['canker', 'blackspot', 'grenning', 'fresh']

### train-dataset count ###
canker : 179
blackspot : 184
grenning : 347
fresh : 281

## test-dataset count ###
canker : 22
fresh : 33
blackspot : 22
grenning : 22
```

모델 구조 정의

VGG 모델을 비교하기 위해 **VGG11, VGG13, VGG16, VGG19 모델의 구조**를 각각 정의하였다. 이때 weights는 VGG 각각의 모델에 해당하는 것을 불러와 사용하였다. 그리고 classifier 앞쪽 부분은 Freeze 설정을 하여 Fine-tuning 과정에서 weight가 수정되지 않도록 설정해주었다. classifier layer는 이진 분류 등 다른 클래스 개수의 분류를 진행 할 것을 고려하여 변수로 받아 들어가도록 하였다. 이번 연구에서는 4개의 클래스 분류가 되도록 출력 개수를 설정하였다. 정리하자면 특징을 뽑는 부분은 사전학습 모델의 구조와 가중치 모두 그대로 사용하도록 했고, 특징 추출 층이 끝났을 때 (7, 7) 사이즈로 항상 맞춰지도록 AdaptiveAvgpool2d((7,7))을 적용해주었다. 이 사이즈는 원래의 것과 동일하지만 구조의 안정성을 위해 추가하였다. 또한 모델 파라미터는 전체 개수 27,692,612개에서 12,977,924개만이 학습 가능하다. 이것은 classifier 부분의 파라미터 개수로 이해할 수 있다.

```
# class pretrained_model(nn.Module):
from torchvision.models.vgg import vgg16, VGG16_Weights
from torchinfo import summary

class VGG16_CNN_model(nn.Module):
    def __init__(self, class_num):
        super(VGG16_CNN_model, self).__init__()
        self.base_model = vgg16(weights = VGG16_Weights.DEFAULT)
        self.model_name = "VGG16_CNN_model"
        self.avgpool = nn.AdaptiveAvgPool2d((7, 7))
        self.flatten = nn.Flatten()

        # Freeze feature extraction layers
        for param in self.base_model.features.parameters():
            param.requires_grad = False

        self.base_model.classifier = nn.Sequential(
            nn.Linear(512 * 7 * 7, 512),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(512,256),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(256,class_num),
```

```
        )

    def forward(self,x):
        x = self.base_model(x)
        return x

summary(VGG16_CNN_model(class_num=4), input_size=(32, 3, 224, 224))
```

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
VGG16_CNN_model                          [32, 4]                   --
├─VGG: 1-1                               [32, 4]                   --
│    └─Sequential: 2-1                   [32, 512, 7, 7]           --
│    │    └─Conv2d: 3-1                  [32, 64, 224, 224]        (1,792)
│    │    └─ReLU: 3-2                    [32, 64, 224, 224]        --
│    │    └─Conv2d: 3-3                  [32, 64, 224, 224]        (36,928)
│    │    └─ReLU: 3-4                    [32, 64, 224, 224]        --
...
│    │    └─Conv2d: 3-29                 [32, 512, 14, 14]         (2,359,808)
│    │    └─ReLU: 3-30                   [32, 512, 14, 14]         --
│    │    └─MaxPool2d: 3-31              [32, 512, 7, 7]           --
│    └─AdaptiveAvgPool2d: 2-2            [32, 512, 7, 7]           --
│    └─Sequential: 2-3                   [32, 4]                   --
│    │    └─Linear: 3-32                 [32, 512]                 12,845,568
│    │    └─ReLU: 3-33                   [32, 512]                 --
│    │    └─Dropout: 3-34                [32, 512]                 --
│    │    └─Linear: 3-35                 [32, 256]                 131,328
│    │    └─ReLU: 3-36                   [32, 256]                 --
│    │    └─Dropout: 3-37                [32, 256]                 --
│    │    └─Linear: 3-38                 [32, 4]                   1,028
==========================================================================================
Total params: 27,692,612
Trainable params: 12,977,924
Non-trainable params: 14,714,688
Total mult-adds (Units.GIGABYTES): 491.94
==========================================================================================
Input size (MB): 19.27
Forward/backward pass size (MB): 3468.36
Params size (MB): 110.77
Estimated Total Size (MB): 3598.40
==========================================================================================
```

다음으로는 ResNet 모델의 구조를 정의하였다. ResNet도 마찬가지로 모델 비교를 위해 **ResNet 18, ResNet 34, ResNet 50, ResNet 101, ResNet 152 모델의 구조**를 각각 정의해 주었다. ResNet도 VGG와 마찬가지로 Classifier 부분만 가중치가 업데이트 될수 있도록 Freeze를 주었으며 출력 FC layer 부분에서 클래스 개수(4개)에 맞게 값이 나올수 있도록 설정해주었다. 모델의 파라미터 개수는 42,508,356이고, 이중에 학습 가능한 것은 8,196개인것을 알수 있다. VGG와 비교했을 때 많은 Residual Block을 사용한 ResNet 모델이 전체 파라미터 개수는 더 많지만, 분류기(classifier) 부분의 복잡도는 VGG가 ResNet보다 크다는 것을 알수 있다. 단순히 비교해도 3개의 fully connected layer로 구성된 VGG의 분류기 층 구조가 단순 Linear 층으로 구성된 ResNet 보다 복잡함을 이해할 수 있다.

```
# ResNet101 model
from torchinfo import summary
from torchvision.models import resnet101, ResNet101_Weights

class ResNet101_CNN_model(nn.Module):
    def __init__(self, class_num):
        super(ResNet101_CNN_model,self).__init__()
        self.base_model = resnet101(weights=ResNet101_Weights.DEFAULT)
        self.model_name = "ResNet101_CNN_model"
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.flatten = nn.Flatten()

        # Freeze feature extraction layers
        for name, param in self.base_model.named_parameters():
            if "fc" not in name:
                param.requires_grad = False

        self.base_model.fc = nn.Linear(self.base_model.fc.in_features, class_num)
        torch.nn.init.xavier_uniform_(self.base_model.fc.weight)

    def forward(self, x):
        x = self.base_model(x)
        return x
```

```
summary(ResNet101_CNN_model(class_num=4), input_size=(32, 3, 224, 224))
```

```
==========================================================================================
Layer (type:depth-idx)                             Output Shape              Param #
==========================================================================================
ResNet101_CNN_model                                [32, 4]                   --
├─ResNet: 1-1                                      [32, 4]                   --
│    └─Conv2d: 2-1                                 [32, 64, 112, 112]        (9,408)
│    └─BatchNorm2d: 2-2                            [32, 64, 112, 112]        (128)
│    └─ReLU: 2-3                                   [32, 64, 112, 112]        --
│    └─MaxPool2d: 2-4                              [32, 64, 56, 56]          --
│    └─Sequential: 2-5                             [32, 256, 56, 56]         --
│    │    └─Bottleneck: 3-1                        [32, 256, 56, 56]         (75,008)
│    │    └─Bottleneck: 3-2                        [32, 256, 56, 56]         (70,400)
│    │    └─Bottleneck: 3-3                        [32, 256, 56, 56]         (70,400)
│    └─Sequential: 2-6                             [32, 512, 28, 28]         --
│    │    └─Bottleneck: 3-4                        [32, 512, 28, 28]         (379,392)
│    │    └─Bottleneck: 3-5                        [32, 512, 28, 28]         (280,064)
│    │    └─Bottleneck: 3-6                        [32, 512, 28, 28]         (280,064)
│    │    └─Bottleneck: 3-7                        [32, 512, 28, 28]         (280,064)
│    └─Sequential: 2-7                             [32, 1024, 14, 14]        --
│    │    └─Bottleneck: 3-8                        [32, 1024, 14, 14]        (1,512,448)
│    │    └─Bottleneck: 3-9                        [32, 1024, 14, 14]        (1,117,184)
│    │    └─Bottleneck: 3-10                       [32, 1024, 14, 14]        (1,117,184)
…
│    │    └─Bottleneck: 3-30                       [32, 1024, 14, 14]        (1,117,184)
│    └─Sequential: 2-8                             [32, 2048, 7, 7]          --
│    │    └─Bottleneck: 3-31                       [32, 2048, 7, 7]          (6,039,552)
│    │    └─Bottleneck: 3-32                       [32, 2048, 7, 7]          (4,462,592)
│    │    └─Bottleneck: 3-33                       [32, 2048, 7, 7]          (4,462,592)
│    └─AdaptiveAvgPool2d: 2-9                      [32, 2048, 1, 1]          --
│    └─Linear: 2-10                                [32, 4]                   8,196
==========================================================================================
Total params: 42,508,356
Trainable params: 8,196
Non-trainable params: 42,500,160
Total mult-adds (Units.GIGABYTES): 249.58
==========================================================================================
Input size (MB): 19.27
Forward/backward pass size (MB): 8310.75
Params size (MB): 170.03
Estimated Total Size (MB): 8500.05
==========================================================================================
```

코드의 실행 테스트 및 사전 학습 모델과의 비교를 위해 Simple_CNN_model 모델도 정의했다.

```
!pip install torchinfo
# 가장 단순한 형태의 CNN 모델 구조
import torch.nn as nn
from torchinfo import summary

class Simple_CNN_model(nn.Module):
    def __init__(self, class_num):
        super().__init__()
        self.model_name = "Simple_CNN_model"
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(128 * 28 * 28, 512) # after flatten
        self.fc2 = nn.Linear(512, class_num)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.pool(self.relu(self.conv1(x))) # conv2d > ReLU > MaxPool2d
        x = self.pool(self.relu(self.conv2(x)))
        x = self.pool(self.relu(self.conv3(x)))
        x = x.view(-1, 128 * 28 * 28) # flatten
        x = self.relu(self.fc1(x)) # fc layer > ReLU
        x = self.fc2(x)
        return x

summary(Simple_CNN_model(class_num=4), input_size=(32, 3, 224, 224))
```

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
Simple_CNN_model                         [32, 4]                   --
├─Conv2d: 1-1                            [32, 32, 224, 224]        896
├─ReLU: 1-2                              [32, 32, 224, 224]        --
├─MaxPool2d: 1-3                         [32, 32, 112, 112]        --
├─Conv2d: 1-4                            [32, 64, 112, 112]        18,496
├─ReLU: 1-5                              [32, 64, 112, 112]        --
├─MaxPool2d: 1-6                         [32, 64, 56, 56]          --
├─Conv2d: 1-7                            [32, 128, 56, 56]         73,856
├─ReLU: 1-8                              [32, 128, 56, 56]         --
├─MaxPool2d: 1-9                         [32, 128, 28, 28]         --
├─Linear: 1-10                           [32, 512]                 51,380,736
├─ReLU: 1-11                             [32, 512]                 --
├─Linear: 1-12                           [32, 4]                   2,052
==========================================================================================
Total params: 51,476,036
Trainable params: 51,476,036
Non-trainable params: 0
Total mult-adds (Units.GIGABYTES): 17.92
==========================================================================================
Input size (MB): 19.27
Forward/backward pass size (MB): 719.46
Params size (MB): 205.90
Estimated Total Size (MB): 944.63
==========================================================================================
```

모델 학습 및 평가 클래스 및 함수 정의

모델 학습에서 **조기종료 조건**을 넣기 위해 EarlyStopping 클래스를 만들었다. 조기종료의 조건은 기본값으로 3번 연속으로 학습의 Loss값이 0.001 이상으로 떨어지지 않을 시 종료되도록 설정하였다. 또한 따로 비동기로 처리하여 학습 Loss가 코드 내에서 입력되면 값을 확인하여 자동으로 실행될 수 있도록 구현했다.

```python
# early stop 함수 정의
class EarlyStopping:
    def __init__(self, patience=3, min_delta=0.001):
        self.patience = patience
        self.min_delta = min_delta
        self.counter = 0
        self.best_loss = None
        self.early_stop_TF = False

    def __call__(self, val_loss):
        if self.best_loss is None:
            self.best_loss = val_loss
        elif val_loss > self.best_loss - self.min_delta:
            self.counter += 1
            if self.counter >= self.patience:
                self.early_stop_TF = True
        else:
            self.best_loss = val_loss
            self.counter = 0
```

다음으로는 학습, 평가, 시각화 등 전반적인 수행을 진행할 **Evaluation 클래스를 구현**하였다. 해당 클래스의 속성으로는 디바이스과 모델부터, 데이터 로더, 컴파일 함수, 에포크 등을 인자로 받아 처리하였고, 모델 수행 과정에서의 결과를 얻기 위해 여러 값들의 초기값을 None으로 할당하여 정의하였다.

```python
import matplotlib.ticker as ticker
import seaborn as sns
from sklearn.metrics import confusion_matrix
from torch.utils.tensorboard import SummaryWriter

class Evaluation:
    def __init__(self, device, model, criterion, optimizer,
                 train_dataloader, test_dataloader, epochs=30):
        self.device = device
        self.model = model
        self.criterion = criterion
        self.optimizer = optimizer
        self.train_dataloader = train_dataloader
        self.test_dataloader = test_dataloader
        self.epochs = epochs
        self.train_losses = None
        self.test_accuracies = None
        self.train_test_option = None
        self.all_labels = None
        self.all_predicts = None
```

그리고 Evaluation 클래스 내에 **학습과 평가를 수행할 기본 함수를 정의**하였다. train_model 함수와 test_model 함수의 경우 모두 한번의 epoch에서의 학습과 평가를 수행하는 코드로서 정의되어있다. 이론상 두 함수 모두 dataloader의 개수만큼 실행되면서, 학습의 경우 그때의 running loss를 누적해서 초기값 0.0의 변수에 더해가는 방식으로 진행되고 평가의 경우에도 그때의 예측값과 라벨을 저장하고 비교하면서 정확도를 얻는 방식으로 구현했다.

```
         # train model 함수 정의
        def train_model(self):
            self.model.train()
            running_loss = 0.0

            for inputs, labels in self.train_dataloader:
                inputs, labels = inputs.to(self.device), labels.to(self.device)
                self.optimizer.zero_grad()
                outputs = self.model(inputs)
                loss = self.criterion(outputs, labels)
                loss.backward()
                self.optimizer.step()
                running_loss += loss.item()
            return running_loss

        # test model 함수 정의
        def test_model(self):
            self.model.eval()
            all_predicts = []
            all_labels = []
            correct = 0
            total = 0

            # 데이터 로더를 사용하여 모델 테스트 진행
            with torch.no_grad():
                for inputs, labels in self.test_dataloader:
                    inputs, labels = inputs.to(self.device), labels.to(self.device)
                    outputs = self.model(inputs)
                    loss = self.criterion(outputs, labels)

                    _, predicted = torch.max(outputs.data, 1)
                    total += labels.size(0)
                    correct += (predicted == labels).sum().item()
                    accuracy = (predicted == labels).sum().item() / len(labels)

                    # 예측 라벨 저장
                    all_predicts.append(predicted)
                    all_labels.append(labels)

            self.all_predicts = all_predicts
            self.all_labels = all_labels

            test_accuracy = correct / total
            print(f'Accuracy : {test_accuracy : .4f}')
            return test_accuracy
```

이제 앞에서 정의한 학습과 평가 함수를 사용하여 평가를 매번 (epoch 마다) 수행할지, 마지막에 한번만 수행할 지에 따라 다르게 **두가지 방식의 수행 함수**를 정의했다. train_test_eachtime 함수의 경우 epoch 마다 학습을 진행하면서 동시에 그때의 모델로 평가를 진행하여 정확도 데이터를 얻는 방식을 사용하였다. 이와 달리 train_test_lasttime 함수의 경우 마지막에만 평가를 수행한 것을 알 수 있다. 두 함수 모두 EarlyStoping 인스턴스를 사용하여 조기종료 조건을 매 epoch 마다 검사하였다. 추가적으로 시각화를 위해 두 함수 모두 loss 값과 accuracy값을 저장하여 evaluation 속성 값을 갱신해 주었고, TensorBoard로의 시각화를 위한 runs 값을 저장하기 위해 writer 함수로 값을 저장하였다.

```
# epoch를 반복하면서 train+test+early-stop 수행
def train_test_eachtime(self, earlystopping):

    # tensorboard writer 기록 경로 설정
    self.train_test_option = "eachtime"
    writer = SummaryWriter(f'{base_path}/runs/{self.model.model_name}_{self.train_test_option}')
```

```
        # loss - acc 초기 빈 리스트 생성
        train_losses = []
        test_accuracies = []

        for epoch in range(self.epochs):
            # 모델 train 진행하고 loss값 도출
            running_loss = self.train_model()
            train_loss = running_loss / len(self.train_dataloader)
            train_losses.append(train_loss)

            print(f"""
            Epoch {epoch + 1}/{self.epochs},
            Loss: {running_loss / len(self.train_dataloader) :.4f}
                  """)


            # 모델 test 진행하고 accuracy값 도출
            test_accuracy = self.test_model()
            test_accuracies.append(test_accuracy)

            # TensorBoard 기록
            writer.add_scalar('Loss/train', running_loss / len(self.train_dataloader), epoch)
            writer.add_scalar('Accuracy/test', test_accuracy, epoch)

            # early stop 조건 확인
            earlystopping(running_loss/len(self.train_dataloader))
            if earlystopping.early_stop_TF == True:
                print("Early stopping")
                break

        # evalution 속성값 갱신
        self.train_losses = train_losses
        self.test_accuracies = test_accuracies

        # writer 기록 종료
        writer.close()

    # 모든 train 실행이 끝난후 test 수행
    def train_test_lasttime(self, earlystopping):

        # tensorboard writer 기록 경로 설정
        self.train_test_option = "lasttime"
        writer = SummaryWriter(f'{base_path}/runs/{self.model.model_name}_{self.train_test_option}')

        train_losses = []
        last_epoch = 0 # 마지막 epoch를 저장하기 위한 초기값
        for epoch in range(self.epochs):
            # 모델 train 진행하고 loss값 도출
            running_loss = self.train_model()
            train_loss = running_loss / len(self.train_dataloader)
            train_losses.append(train_loss)
            last_epoch = epoch

            # TensorBoard 기록
            writer.add_scalar('Loss/train', running_loss / len(self.train_dataloader), epoch)

            print(f"""
            Epoch {epoch + 1}/{self.epochs},
            Loss: {running_loss / len(self.train_dataloader) :.4f}
                  """)

            # early stop 조건 확인
            earlystopping(running_loss/len(self.train_dataloader))
            if earlystopping.early_stop_TF == True:
                print("Early stopping")
                break
         self.train_losses = train_losses

        # 마지막에 test 한번 진행
        test_accuracy = self.test_model()
        self.test_accuracies = [test_accuracy]
        writer.add_scalar('Accuracy/test', test_accuracy, last_epoch)
```

다음으로는 loss값과 accuracy 값을 tensorboard를 사용하지 않고 직접적으로 evaluation 인스턴스에 리스트 형태로 저장한 값을 사용하여 시각화 하는 visualize_loss_acc 함수를 작성하였다. 앞에서 수행함수의 방식에 따라 시각화 방법이 다르게 작동하도록 설정하였다. 매번 평가을 수행한 경우(eachtime) loss와 acc가 좌 우 측에 값을 y축으로 각각 사용하며 시각화 되도록 설정하였고, 마지막에만 평가를 수행한 경우(lasttime) loss값만으로 시각화를 수행하도록 설정했다.

```python
# loss와 accuracy 변화 시각화
def visualize_loss_acc(self):
    if self.train_test_option == "eachtime":
        plt.figure(figsize=(6, 3))

        # 손실 값은 왼쪽 y축에, 정확도는 오른쪽 y축에 표시
        ax1 = plt.gca()
        ax2 = ax1.twinx()

        ax1.plot(self.train_losses, label='Train Loss',
                 color='blue', linestyle='-', marker='o')
        ax2.plot(self.test_accuracies, label='Test Accuracy',
                 color='red', linestyle='--', marker='x')


        ax1.set_xlabel('Epoch')
        ax1.set_ylabel('Loss', color='blue')
        ax2.set_ylabel('Accuracy', color='red')
        ax2.set_ylim(0.5, 1.0) # accuracy 값 범주 설정

        # 범례 표시
        lines, labels = ax1.get_legend_handles_labels()
        lines2, labels2 = ax2.get_legend_handles_labels()
        ax1.legend(lines + lines2, labels + labels2, loc='upper left')
        ax2.yaxis.set_major_locator(ticker.LinearLocator(numticks=5))

        plt.title(f'{self.model.model_name} Evaluation({self.test_accuracies[-1]*100:.2f}%)')
        plt.tight_layout()

        save_path = f'{base_path}/result/{self.model.model_name}_{self.train_test_option}.png'
        os.makedirs(os.path.dirname(save_path), exist_ok=True)
        plt.savefig(save_path)
        print(f"the figure saved to {save_path}")

        plt.show()

    elif self.train_test_option == "lasttime":
        plt.figure(figsize=(6, 3))

        plt.plot(self.train_losses, label='Train Loss', color='blue', linestyle='-', marker='o')

        plt.xlabel('Epoch')
        plt.ylabel('Loss', color='blue')

        plt.title(f'{self.model.model_name} Evaluation({self.test_accuracies[-1]*100:.2f}%)')
        plt.tight_layout()
        plt.legend(loc='upper left')

        save_path = f'{base_path}/result/{self.model.model_name}_{self.train_test_option}.png'
        os.makedirs(os.path.dirname(save_path), exist_ok=True)
        plt.savefig(save_path)
        print(f"the figure saved to {save_path}")

        plt.show()

    else:
        print("train or test were not completed")
```

Evaluation 클래스 내 함수의 마지막 함수로 Confusion Matrix를 그리기 위한 visualize_confusion 함수를 구현하였다. 이를 위해 필요한 데이터는 evaluation 인스턴스에 저장해둔 tensor 값을 numpy 객체로 바꾸면서 시작되고, sklearn.metrics 라이브러리에서 confusion_matrix 함수를 불러와 히트맵을 그리는 방식으로 시각화 하였다.

```python
# confusion matrix
def visualize_confusion(self):
    # Confusion Matrix 계산
    all_predicts = torch.cat(evaluation.all_predicts).cpu().numpy()
    all_labels = torch.cat(evaluation.all_labels).cpu().numpy()

    conf_matrix = confusion_matrix(all_labels, all_predicts)

    # Confusion Matrix 시각화
    plt.figure(figsize=(8, 6))
    sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',
                xticklabels=class_names, yticklabels=class_names)

    plt.xlabel('Predicted Labels')
    plt.ylabel('True Labels')
    plt.title(f'{self.model.model_name} Confusion Matrix')
```

```
        #
        save_path = f'{base_path}/result/{self.model.model_name}_confusion.png'
        os.makedirs(os.path.dirname(save_path), exist_ok=True)
        plt.savefig(save_path)
        print(f"the confusion figure saved to {save_path}")
        plt.show()
```

여러 기능의 독립적인 함수 정의

평가를 수행하는 함수 외 보조적인 기능을 구현하기 위해 여러 함수를 구현했다. 먼저 model_comile 함수의 경우 컴파일을 위한 Loss 함수(Criterion)와 Optimizer 함수를 정의하기 위해 사용했다. 이번 연구에서는 4개의 클래스로 분류하는 모델만을 사용했지만, 클래스 개수에 따른 컴파일 변수를 구분하여 명확히 하기 위해 제작되었다. 그리고 loss_acc_save 함수의 경우 특정 경로에 학습과 평가를 수행한 뒤의 loss값과 accuracy 값을 evaluation 인스턴스에서 불러와 JSON 형태로 정해진 경로에 저장하기 위해 제작되었다. 마지막으로 model_save 함수의 경우에도 loss_acc_save 함수와 유사하게, 학습과 평가를 수행한 최종 모델의 가중치 정보를 pth 형태로 경로에 저장하기 위해 제작되었다.

```
# 모델 컴파일 변수 함수
def model_comile(model, classes: int = 'multi'):
    """
    classes = multi  : four-classes model (Default)
    classes = binary : binary-claaes model
    """
    if classes == 'multi':
        criterion = nn.CrossEntropyLoss()
        optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
    elif classes == 'binary':
        criterion = nn.BCEWithLogitsLoss()
        optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
    return criterion, optimizer

# 모델의 loss와 accuracy 저장
def loss_acc_save(evaluation):
    result_dir = f'{base_path}/result/loss_acc.json'
    os.makedirs(os.path.dirname(result_dir), exist_ok=True)

    data = {
        evaluation.model.model_name: {
            "train_losses": evaluation.train_losses,
            "test_accuracies": evaluation.test_accuracies
        }
    }

    try:
        if os.path.exists(result_dir):  # 파일이 존재하면 로드
            with open(result_dir, 'r') as f:
                existing_data = json.load(f)
            existing_data.update(data)
            data_to_write = existing_data
        else:
            data_to_write = data

        with open(result_dir, 'w') as f: # 파일 쓰기
            json.dump(data_to_write, f, indent=4)
        print(f"Losses and accuracies saved to {result_dir}")

    except Exception as e:
        print(f"Error saving to JSON: {e}")

# 모델을 pth 파일로 저장
def model_save(model,version = ''):
    model_dir = f"{base_path}/model"

    if not os.path.exists(model_dir):
        os.makedirs(model_dir)
    try:
        torch.save(model.state_dict(), os.path.join(model_dir, f'{model.model_name}_{version}.pth'))
        print(f"the model is saved at {model_dir}/{model.model_name}.pth")
    except Exception as e:
        print(f"An error occurred while saving the model: {e}")
```

데이터 셋 선택

사용할 데이터 셋과 데이터 로더가 정의된 함수에서 각각의 변수를 할당받아 이후 작업에서 사용하기 위해 해당 코드를 작성했다. 이번 연구에서는 orange_data를 사용했는데 입력 사이즈와 정보 및 이미지를 볼껀 지 아닌지 bool 의 값을 입력받아 선택할 수 있게 하였다. 데이터 확인을 할 것이 아니고 변수를 할당하기 위한 것이라면 기본적으로는 False로 하여 사용한다. 또한 이번에는 사용하지 않았지만 만약 데이터 증강 또는 다른 데이터를 불러와 사용할 가능성을 위해 이러한 방식을 선택했다.

```
# select dataset
train_set, test_set, train_loader, test_loader = orange_data(in_size=(224, 224),
                                                             info_view = False, img_view= False)
```

```
using dataset : orange_data
```

학습 및 평가 수행

이 과정은 **앞에서 정의한 클래스와 함수를 사용하여 학습 및 평가를 진행**하는 과정이다. 일단 사용할 모델을 정의한 다음 Compile 정의, Evaluation 인스턴스 생성, EarlyStopping 인스턴스 생성 등의 작업을 수행한다. 그 다음음 기본적으로 Loss값과 Accuracy 값을 Epoch 마다 확인하기 위해 train_test_eachtime으로 진행하였다. 그 다음 시각화를 하는 과정과 저장을 하는 과정이 이어진다. 아래의 코드는 **VGG16**의 예시이다.

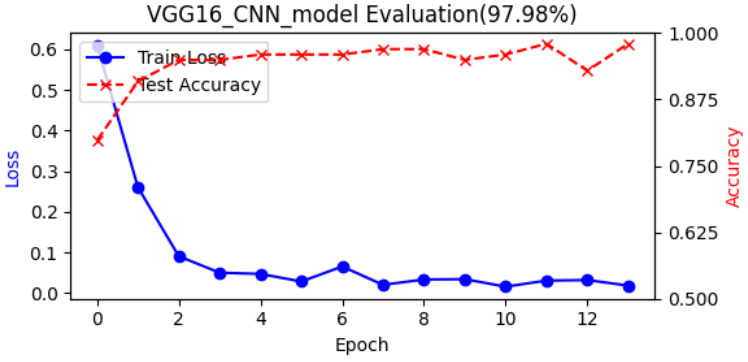
```
# device and model
model = VGG16_CNN_model(class_num=4).to(device)

# train_test class instance
criterion, optimizer = model_comile(model, 'multi')
evaluation = Evaluation(device, model, criterion, optimizer, train_loader, test_loader, epochs = 30)
earlystopping = EarlyStopping(patience=3, min_delta=0.001)

# model train and test
evaluation.train_test_eachtime(earlystopping)
evaluation.visualize_loss_acc()

# save model
loss_acc_save(evaluation)
model_save(model)
```

```
Epoch 1/30, Loss: 0.6103
Accuracy :  0.7980
Epoch 2/30, Loss: 0.2605
Accuracy :  0.9091
Epoch 3/30, Loss: 0.0908
Accuracy :  0.9495
Epoch 4/30, Loss: 0.0504
Accuracy :  0.9495
Epoch 5/30, Loss: 0.0474
Accuracy :  0.9596
...
Accuracy :  0.9495
Epoch 11/30, Loss: 0.0161
Accuracy :  0.9596
Epoch 12/30, Loss: 0.0310
Accuracy :  0.9798
Epoch 13/30, Loss: 0.0323
Accuracy :  0.9293
Epoch 14/30, Loss: 0.0187
Accuracy :  0.9798
Early stopping
the figure saved to /content/drive/MyDrive/Orange_CNN/result/VGG16_CNN_model_eachtime.png
Losses and accuracies saved to /content/drive/MyDrive/Orange_CNN/result/loss_acc.json
the model is saved at /content/drive/MyDrive/Orange_CNN/model/VGG16_CNN_model.pth
```



ResNet 101의 경우에도 모델 정의하는 부분을 제외하고는 동일하게 진행하였다. 이렇게 동일한 코드를 함수화 하여 불러와 사용하지 않은 이유는 학습 및 평가의 과정이 코드를 보고도 직관적으로 이해할 수 있도록 한 이유가 있으며, 아래에 해당 모델의 평가를 수행한 결과를 확인하는 과정이 필요하기에 모델 별로 서로다른 셀에서 작업을 진행했다.

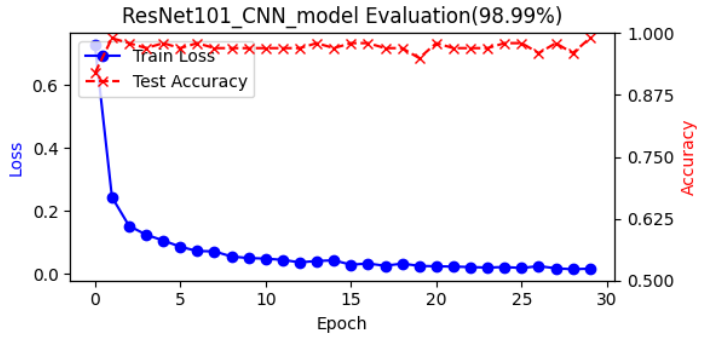
```
# device and model
model = ResNet101_CNN_model(class_num=4).to(device)

# train_test class instance
criterion, optimizer = model_comile(model, 'multi')
evaluation = Evaluation(device, model, criterion, optimizer, train_loader, test_loader, epochs = 30)
earlystopping = EarlyStopping(patience=3, min_delta=0.001)

# model train and test
evaluation.train_test_eachtime(earlystopping)
evaluation.visualize_loss_acc()

# save model
loss_acc_save(evaluation)
model_save(model)
```

```
Epoch 1/30, Loss: 0.7303
Accuracy :  0.9192
Epoch 2/30, Loss: 0.2442
Accuracy :  0.9899
Epoch 3/30, Loss: 0.1529
Accuracy :  0.9798
...
Epoch 28/30, Loss: 0.0178
Accuracy :  0.9798
Epoch 29/30, Loss: 0.0146
Accuracy :  0.9596
Epoch 30/30, Loss: 0.0167
Accuracy :  0.9899
the figure saved to /content/drive/MyDrive/Orange_CNN/result/ResNet101_CNN_model_eachtime.png
Losses and accuracies saved to /content/drive/MyDrive/Orange_CNN/result/loss_acc.json
the model is saved at /content/drive/MyDrive/Orange_CNN/model/ResNet101_CNN_model.pth
```



클래스 별 분류 결과 시각화(Confusion Matrix)

정확도를 확인하는 것 외에 클래스 별로 어떻게 분류를 했는지 정확하게 확인하기 위해 Confusion Matrix를 사용했다. 이러한 작업을 통해 실제 값(True Labels)이 뭐일때 어떤 클래스로 모델이 분류했는지(Predicted Labels) 확인할 수 있었다. 추가적으로 version을 설정한 이유는 만약 다른 버전의 데이터셋(증강 등의 작업을 한 데이터)을 사용하는 경우에 해당 데이터로 학습된 모델을 model_name_v1과 같은 이름으로 저장하여 구분하기 위한 것이다. 그렇기에 데이터를 불러올 때에도 불러오는 모델의 이름에 버전을 추가해주는 과정이 필요하다. 기본적으로는 version = ‘ ‘ 로 빈 문자를 넣어 사용한다.

```
import torch
model = VGG16_CNN_model(class_num=4).to(device) # modified

# 합성곱 레이어의 가중치 값(4차원 tensor) 불러오기
model_name = model.model_name
version = '_v1'
model_path = f'{base_path}/model/{model_name}{version}.pth'
class_names = test_set.classes ; print(class_names)

# 모델에 가중치 적용
model.load_state_dict(torch.load(model_path))

# model test
criterion, optimizer = model_comile(model, 'multi')
evaluation = Evaluation(device, model, criterion, optimizer, train_loader, test_loader, epochs = 30)
evaluation.test_model()
evaluation.visualize_confusion()
```

DB 결과 저장

추가적으로 모델의 평가 정보 등을 NoSQL DB에 저장하여 사용하기 위해 MongoDB atlas 와 연동하여 값이 저장되도록 **Mongo_DB 클래스를 정의**했다. 구조적으로 atlas와 연결을 진행하는 connect 함수, 원하는 컬렉션에 데이터를 넣기 위한 insert_data 함수, 원하는 값을 찾아서 뽑는 find_data 함수, 그리고 DB를 닫아주는 close 함수까지 클래스 내에 정의하였다. 추가적으로 새롭게 넣은 데이터만 입력하기 위해 선택한 DB 안의 컬랙션을 모두 지우는 drop_collection까지 정의하였다.

```
# !pip install pymongo
import urllib.parse
from pymongo.mongo_client import MongoClient
from pymongo.server_api import ServerApi

class Mongo_DB():
    def __init__(self, user_id, password):
        self.user_id = urllib.parse.quote(user_id)
        self.password = urllib.parse.quote(password)
        self.db = None
        self.collection = None
        self.client = None

    def connect(self, db_name, coll_name):
        # Create a new client and connect to the server
        uri = f"""
        mongodb+srv://{self.user_id}:{self.password}
        @fine-tunning.35pqyng.mongodb.net/?retryWrites=true&w=majority&appName=Fine-tunning
        """

        self.client = MongoClient(uri, server_api=ServerApi('1'))
        self.db = self.client[db_name]
        self.collection = self.db[coll_name]

        # Send a ping to confirm a successful connection
        try:
            self.client.admin.command('ping')
            print("Pinged your deployment. You successfully connected to MongoDB!")
        except Exception as e:
            print(e)

    def insert_data(self,data):
        try:
            result = self.collection.insert_one(data)
            print(f"데이터 삽입 성공: {result.inserted_id}")
        except Exception as e:
            print(f"데이터 삽입 실패: {e}")

    def find_data(self, query):
        try:
            results = list(self.collection.find(query))
            if results:
                print("조회된 데이터:")
                for result in results:
                    print(result)
            else:
                print("쿼리에 맞는 데이터가 없습니다.")
        except Exception as e:
            print(f"데이터 조회 실패: {e}")

    def close(self):
        if self.client:
            self.client.close()


    def drop_collection(self, collection_name):
        try:
            self.db.drop_collection(collection_name)
            print(f"컬렉션 '{collection_name}'이 삭제되었습니다.")
            return True
        except Exception as e:
            print(f"컬렉션 삭제 실패: {e}")
            return False
```

그런 다음 모델의 학습 결과(class 예측 및 라벨 정보)를 DB에 모델의 이름으로 구분하여 정의하기 위해 아래의 **평가 및 DB 저장 코드**를 작성하였다. 먼저 모델 이름과 객체로 구성된 딕셔너리를 만들고, 모든 모델에 대해 저장된 가중치를 해당 모델 구조에 적용하고 그 모델로 학습을 수행하였다. 그런다음 얻은 결과를 DB에 저장하고 실행 결과를 확인하는 과정까지 진행하였다.

```python
import torch

model_names = {
    "Simple_CNN_model" : Simple_CNN_model(class_num=4).to(device),
    "VGG11_CNN_model" : VGG11_CNN_model(class_num=4).to(device),
    "VGG13_CNN_model" :  VGG13_CNN_model(class_num=4).to(device),
    "VGG16_CNN_model" : VGG16_CNN_model(class_num=4).to(device),
    "VGG19_CNN_model" : VGG19_CNN_model(class_num=4).to(device),
    "ResNet18_CNN_model" : ResNet18_CNN_model(class_num=4).to(device),
    "ResNet34_CNN_model" : ResNet34_CNN_model(class_num=4).to(device),
    "ResNet50_CNN_model" : ResNet50_CNN_model(class_num=4).to(device),
    "ResNet101_CNN_model" : ResNet101_CNN_model(class_num=4).to(device),
    "ResNet152_CNN_model" : ResNet152_CNN_model(class_num=4).to(device),
}

for model_name, model in model_names.items():
    print("- "*30)
    print(model_name)
    # 합성곱 레이어의 가중치 값(4차원 tensor) 불러오기
    version = ''
    model_name = model.model_name
    model_path = f'{base_path}/model/{model_name}{version}.pth'
    class_names = test_set.classes ; print(class_names)

    # 모델에 가중치 적용
    model.load_state_dict(torch.load(model_path, map_location=device))

    # model test
    criterion, optimizer = model_comile(model, 'multi')
    evaluation = Evaluation(device, model, criterion, optimizer, train_loader, test_loader, epochs = 30)
    test_accuracy = evaluation.test_model()

    # DB save
    db = Mongo_DB('haebo', "wyA3NAea9#AS_x_")
    db.connect(db_name= 'test_result', coll_name= model_name)
    db.drop_collection(model_name)
    data = {
        'model_name' : model_name,
        'version' : version,
        'test_accuracy' : test_accuracy,
        'all_labels': [tensor.tolist() for tensor in evaluation.all_labels],
        'all_predicts': [tensor.tolist() for tensor in evaluation.all_predicts]
    }
    db.insert_data(data)
    db.find_data({})
    db.close()
```