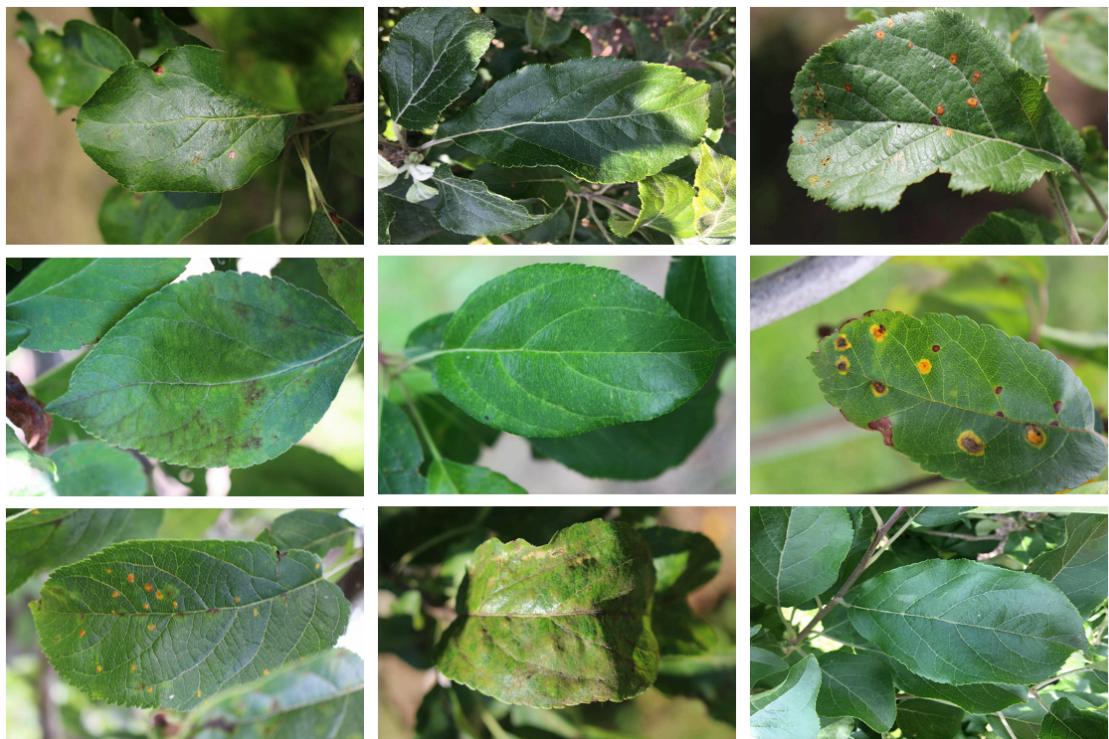


기계학습 최종 보고서

Leaf disease classification



Team 9

14109344 성연석

16101414 조용진

19100598 손정미

19101292 한혜정

Contents

1. 문제 정의
2. 데이터 확보 전략
3. 적용할 기계학습 알고리즘
 - 3.1. ResNet
 - 3.2. Keras CNN
 - 3.3. VGG19
4. 학습 과정 및 코드 분석
 - 4.1. 데이터 시각화 및 전처리
 - 4.2. 모델 구축 및 훈련
 - 4.3. 최종 성능 평가
5. 결과 분석

1. 문제 정의

농작물에 영향을 끼치는 여러 질병들에 대한 오진은 화학물질의 오용으로 이어진다. 이는 해당 식물 병 원체 변종에 저항력이 생기게 하고 비용이 많이들게 되며 환경에도 역시 영향을 끼치게 된다. 현재는 식물의 질병 진단을 대부분 사람이 직접 하는데 이는 시간이 매우 많이 들고 비용 또한 많이 들게 된다. 질병에 대해 보다 이르게 발견하는 것은 해당 질병에 대한 적절한 대처를 계획하고 손실의 최소화하는 데 결정적인 역할을 하기 때문에 이는 굉장히 중요하다.

본 리포트는 이에 착안하여 사과 나뭇잎의 사진이 주어졌을 때 이 나뭇잎이 건강한지, apple rust에 감염되었는지, apple scab을 가지고 있는지, 혹은 이 중 하나 이상의 질병이 있는지 기계학습을 통해 모델을 세워 분류해보고, 해당 결과에 대해 분석을 내린다.

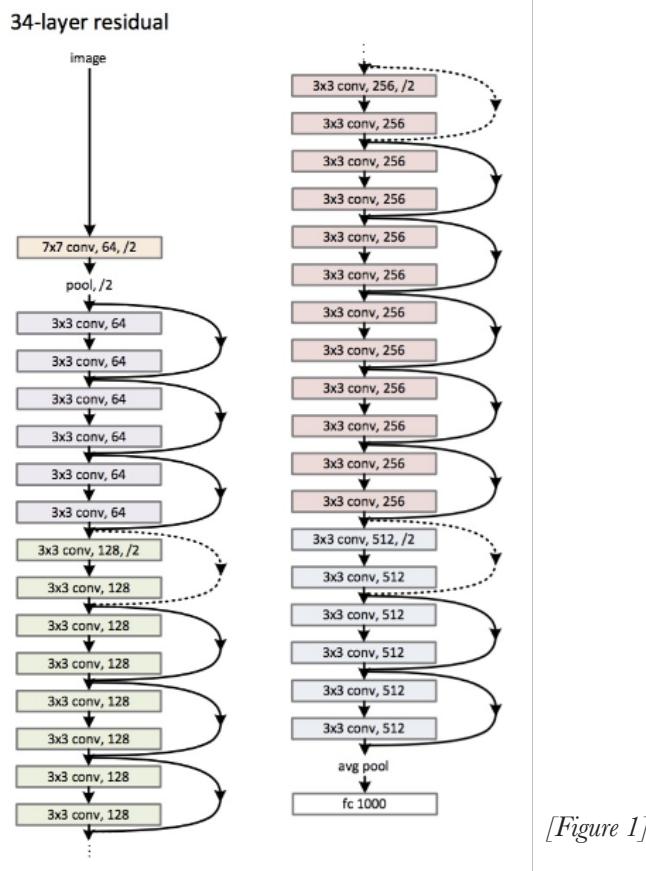
2. 데이터 확보 전략

Kaggle API를 사용하여 plant-pathology-2020 competition 데이터셋을 다운로드 받아 사용한다. 해당 데이터셋은 총 3642개의 이미지와 test.csv, train.csv로 이루어져있다. 해당 데이터셋에 대한 설명은 4. 학습 과정 및 코드 분석에서 다룰 예정이다.

3. 적용할 기계학습 알고리즘

3.1. ResNet

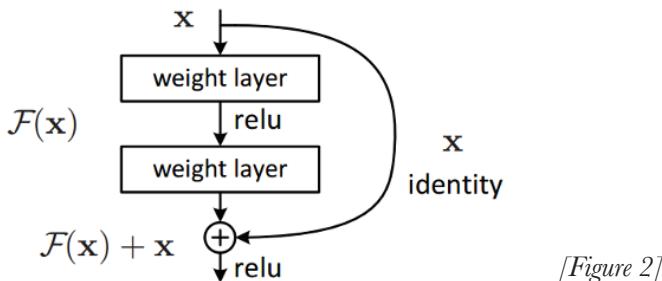
3.1.1. ResNet 모델 구조



프로젝트에서 사용한 모델은 ResNet50이다. [Figure 1]에서 보듯이 dimension을 점차 늘려가면서 (64, 128, 256, 512) Convolution layer를 쌓은 것을 알 수 있다.

Residual Block

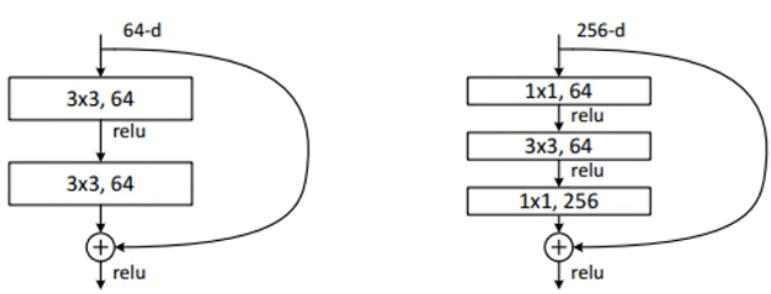
ResNet에서 핵심적인 layer은 Residual Block이다. Block 구조는 다음과 같다.



이전의 weight layer만 추가되어 있던 model에 skip connection을 추가함으로써 최종적인 output은 $H(x) = F(x) + x$ 가 되도록 만들었다. 따라서 weight layer의 output은 $F(x) = H(x) - x$ 의 항등함수를 학습하여 빠르게 학습이 가능하다. 또한 학습이 이루어지지 않은 층이 있을 때 그 이전에 있는 layer들이 학습에 방해를 받게 되는데 이 또한 방지할 수 있다.

BottleNeck

학습하는 parameter를 줄이기 위해 BottleNeck을 사용한다.

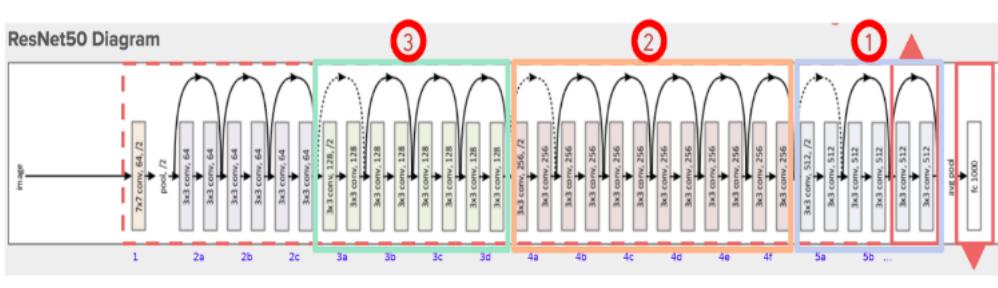


[Figure 3]

[Figure 3]을 보면 BottleNeck의 Input Dimension이 256인 것을 알 수 있다. 여기서 Conv2D 3×3 filter 256을 학습하려고 한다면 parameter의 개수가 $3 \times 3 \times 256 = 2,304$ 가 된다. 하지만 BottleNeck을 사용하여 학습을 한다면 $1 \times 1 \times 64 + 3 \times 3 \times 64 + 1 \times 1 \times 256 = 665$ 개의 parameter만 학습을 하면 된다. 정보의 손실은 어느정도 발생할 수 있다.

3.1.2. 학습 진행 순서

Keras에서 제공하는 pretrained된 ResNet50을 사용하여 학습을 진행하였다.



[Figure 4]

학습 순서는 다음과 같다.

1. Pretrained weight를 가진 ResNet50을 모두 freeze 시킨 후 output layer를 학습시킨다.
2. [Figure 4]에서 1번 부분의 freeze를 해제하고 학습을 시킨다.
3. [Figure 4]에서 2번 부분의 freeze를 해제하고 학습을 시킨다.
4. [Figure 4]에서 3번 부분의 freeze를 해제하고 학습을 시킨다.

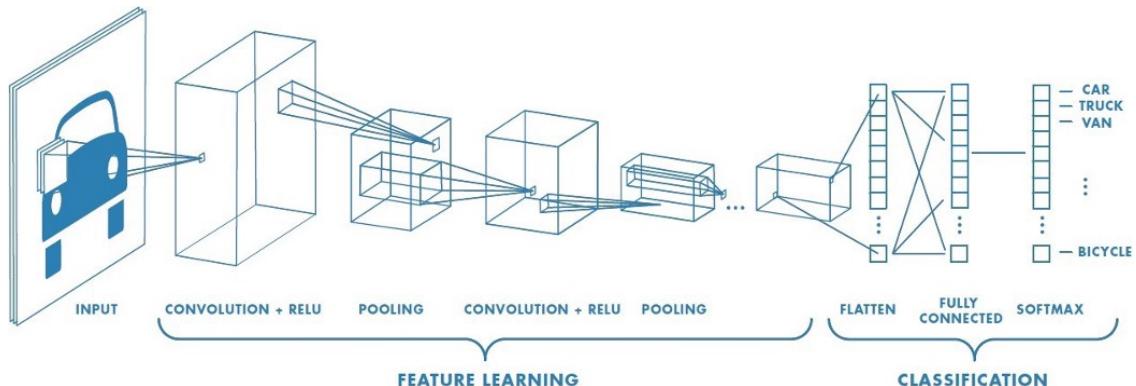
그 다음 layer들까지 학습을 시키면 오히려 검증(valid) 정확도가 감소하고 overfitting 되는 결과가 나타났다.

3.2. Keras CNN

3.2.1. Keras CNN의 모델 구조

CNN은 영상 데이터를 처리할 때 입력되는 이미지의 모든 영역에서 같은 필터를 적용하여 패턴을 찾아내 처리하는 것이 목적이다. Keras CNN의 경우 기본적인 CNN의 구조를 따르고 있어 크게 Convolution

layer, Pooling layer, Dense layer로 구성되어 있다. Input image를 Convolution layer와 Pooling layer에 반복하여 통과시키고 마지막으로 Dense layer를 통과시켜 이미지를 처리한다.



[Figure 5]

Convolution layer

Input image와 필터가 존재하고 필터를 통해 이미지의 특징을 추출하는 과정이 진행된다. 필터를 통과하고 나면 하나의 feature map이 생성되는데, 이는 전체 이미지를 stride 값만큼 이동하며 픽셀과 연산을 거쳐 나온 결과값이다. 이때, 필터를 통과하고 나면 이미지의 크기가 줄어들게 되어 손실이 발생하게 되는데 이를 방지하기 위한 기법으로 Padding이 사용된다. Padding은 input image의 가장자리를 특정 값으로 채운 픽셀을 추가하여 필터를 적용해도 output image의 크기가 input image의 크기와 비슷해지도록 만든다. 주로 픽셀의 값을 0으로 채우는 zero-padding을 사용한다.

이와 같이 필터를 여러 번 통과해서 나온 패턴은 매우 큰 값이 있을 수도 있고 0에 가까운 부분이 있을 수도 있다. 중요한 것은 값이 크기가 아닌 패턴의 여부이기 때문에 이를 위한 과정으로 activation function이 사용된다. 최종적으로 activation function을 적용한 데이터가 convolution layer의 output이 된다.

Pooling layer

주로 Convolution layer 다음에 배치되는 layer이다. Pooling layer은 Convolution layer로 만들어진 결과값을 축소한다. 일정한 범위 내의 픽셀 중 대표값을 추출하는 방식으로 패턴을 추출해낸다. Max Pooling, Min Pooling, Average Pooling과 같은 세 가지 방식이 있다. 각각은 해당 범위의 최댓값, 최솟값, 평균을 대표값으로 추출한다.

Pooling layer은 Convolution layer와 달리 패딩을 통해 output image의 크기를 보존하지 않기 때문에 이미지의 크기가 줄어들게되어 손실이 발생한다. 그러나 하나의 픽셀로부터 형태를 분석하기 쉬워지므로 이미지의 크기를 적절히 줄이면서 특정 Feature를 강조할 수 있다. 따라서 노이즈는 감소하지만 속도는 빨라지고 분별력도 좋아지게 만들 수 있다.

Dense layer

이미지를 분류하는 인공신경망이다. Dense layer의 경우 1차원의 데이터로 바꿔 학습이 되어야하기 때문에 해당 layer를 통과하기 전에 Flatten Layer를 통과시켜 2차원 데이터를 1차원 데이터로 바꾼다. 그리고 Denser layer에 앞선 layer를 통과한 output을 input으로 사용하여 우리가 원하는 분류를 수행한다.

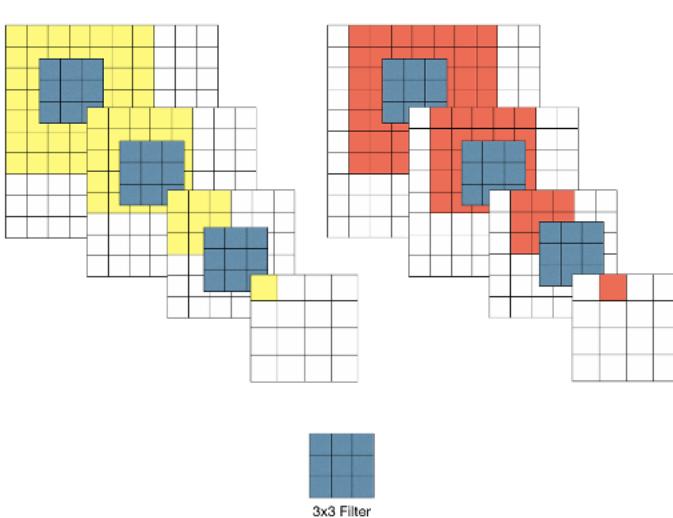
3.3. VGG19

3.3.1. VGG19 모델 개요

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224×224 RGB image)					
conv3-64	conv3-64 LRN	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 conv1-256	conv3-256 conv3-256 conv3-256	conv3-256 conv3-256 conv3-256 conv3-256
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 conv1-512	conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512 conv3-512
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 conv1-512	conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512 conv3-512
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

[Figure 6]

VGGNet은 옥스포드 대학의 연구팀 VGG에 의해 개발된 모델로써 2014년 ImageNet 이미지 인식대회에서 준우승을 한 모델이다. A부터 E까지의 모델이 있는데 그 중 E가 VGG19이고 19는 19개의 layer로 구성되어있기 때문에 19이다. VGGNet은 그 당시 이전의 우승한 모델들보다 네트워크의 깊이가 늘어났다.

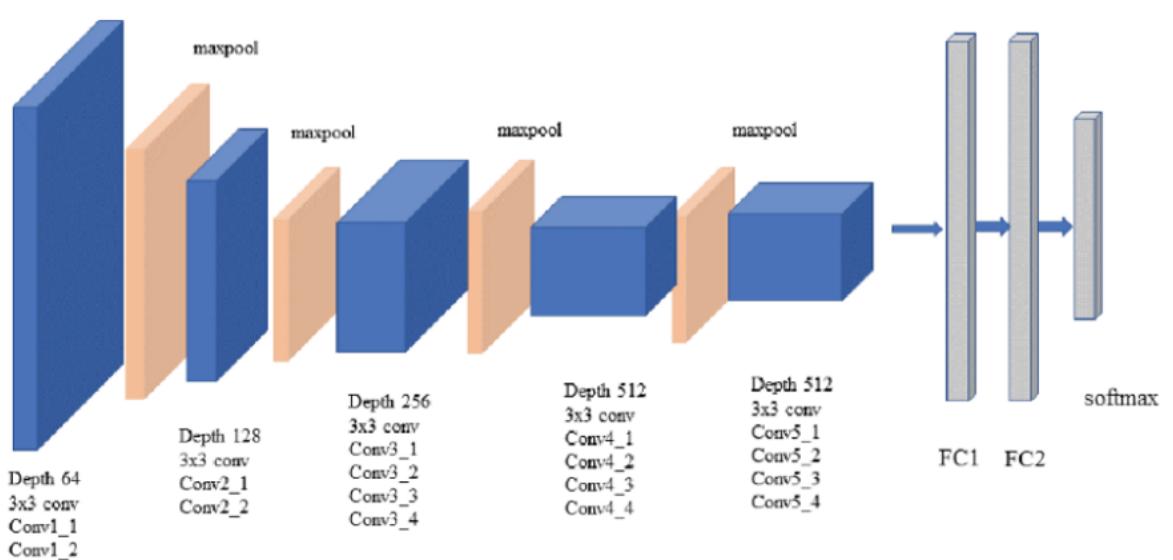


[Figure 7]

이는 VGGNet의 특징 중 하나인 3×3 의 작은 필터를 사용했기에 가능했던 것이었는데 모든 Convolution layer에 작은 필터를 적용하여 이미지의 사이즈가 금방 축소되지 않아 네트워크의 깊이를 충분히 깊게 만들 수 있었던 것이다. 또한 parameter나 weight의 개수가 줄어들어 훈련의 속도가 빨라진다

는 장점도 가지고 있었다. 예를 들어 3×3 필터로 두 차례 convolution을 하는 것과 5×5 필터로 한 번 convolution을 하는 것이 결과적으로 동일한 사이즈의 feature map을 산출하고 3×3 필터로 세 차례 convolution을 하는 것은 7×7 필터로 한 번 convolution 하는 것과 동일한 feature map을 산출한다. 이 때 3×3 필터가 3개면 총 27개의 weight를 갖는다. 반면 7×7 필터는 49개의 weight를 갖는다. CNN에서 weight는 모두 훈련이 필요한 것들이므로, weight가 적다는 것은 그만큼 훈련시켜야 할 개수가 작아진다는 의미이고 이에 따라 학습의 속도가 빨라진다. 동시에 layer의 개수가 늘어나면서 activation function에 더 많은 ReLU 함수를 사용하여 비선형성을 더 증가시킴으로써 이미지의 feature를 특정하는데 더 유용하다.

3.3.2. VGG19 모델 구조



[Figure 8]

VGG19는 $224 \times 224 \times 3$ 이미지(224×224 RGB 이미지)를 입력받는다. 5개의 block이 있는데 각각 2, 2, 4, 4개의 Convolution layer, 각 block의 convolution layer는 64, 128, 256, 512, 512개의 필터를 가지고 있다. 각 block의 마지막 단에는 MaxPooling이 적용되어 있고, 이후 4096개의 hidden layer 두 층, 1000개의 output을 softmax activation function으로 출력한다.

4. 학습 과정 및 코드 분석

4.1. 데이터 시각화 및 전처리

```
import pandas as pd
import matplotlib as mpl
import matplotlib.pyplot as plt
%matplotlib inline

# load data
data_path = '/Users/sjjmi/plant-pathology-2020-fgvc7/'

train_set = pd.read_csv(data_path + 'train.csv', index_col = 0)
test_set = pd.read_csv(data_path + 'test.csv', index_col = 0)
```

```
train_set.shape  
(1821, 4)
```

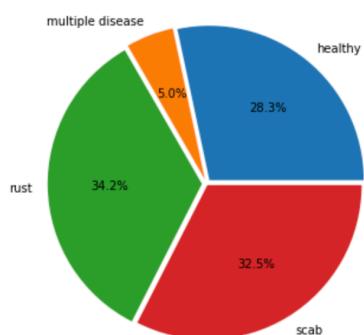
```
train_set.head()
```

	healthy	multiple_diseases	rust	scab
image_id				
Train_0	0	0	0	1
Train_1	0	1	0	0
Train_2	1	0	0	0
Train_3	0	0	1	0
Train_4	1	0	0	0

데이터를 load하고 data structure을 간단히 살펴보면 위와 같다. Train set의 instance는 총 1821개이고 healthy, multiple_diseases, rust, scab 와 같은 4개의 label이 존재함을 알 수 있다.

```
train_healthy = train_set.loc[train_set['healthy']==1]
train_multiple_diseases = train_set.loc[train_set['multiple_diseases']==1]
train_rust = train_set.loc[train_set['rust']==1]
train_scab = train_set.loc[train_set['scab']==1]
```

```
# data visualization
plt.figure(figsize=(6, 6))
label = ['healthy', 'multiple disease', 'rust', 'scab']
explode = [0.03, 0.03, 0.03, 0.01]
plt.pie([len(train_healthy), len(train_multiple_diseases), len(train_rust), len(train_scab)],
       labels=label,
       autopct='%.1f%%',
       explode=explode)
plt.show()
```

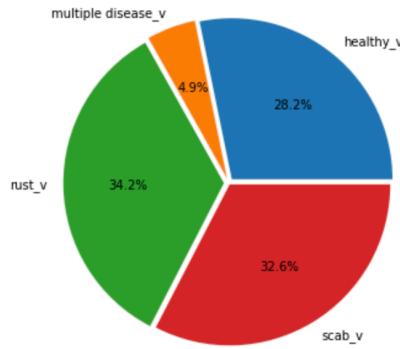


이제 데이터를 시각화하여 각 label별 차지하는 비율을 살펴보자. healthy, rust, scab의 경우 비슷한 비율로 존재하지만 multiple diseases의 경우 비교적 적은 비율로 존재함을 알 수 있다.

```
from sklearn.model_selection import train_test_split
train_set, valid_set = train_test_split(train_set,
                                        test_size=0.2,
                                        stratify=train_set[['healthy', 'multiple_diseases', 'rust', 'scab']],
                                        random_state=42)

healthy_v = valid_set.loc[valid_set['healthy']==1]
multiple_diseases_v = valid_set.loc[valid_set['multiple_diseases']==1]
rust_v = valid_set.loc[valid_set['rust']==1]
scab_v = valid_set.loc[valid_set['scab']==1]

plt.figure(figsize=(6, 6))
label = ['healthy_v', 'multiple disease_v', 'rust_v', 'scab_v']
explode = [0.03, 0.03, 0.03, 0.01]
plt.pie([len(healthy_v), len(multiple_diseases_v), len(rust_v), len(scab_v)],
        labels=label,
        autopct='%.1f%%',
        explode=explode)
plt.show()
```



Train set과 validation set을 나누어보자. Stratified sampling을 통해 label 별 비율을 유지하면서 train set과 validation set을 나눈 뒤 validation set의 각 label 별 비율을 살펴보니 앞의 train set 그래프의 비율과 같음을 확인할 수 있다.

```
from matplotlib import gridspec
from PIL import Image

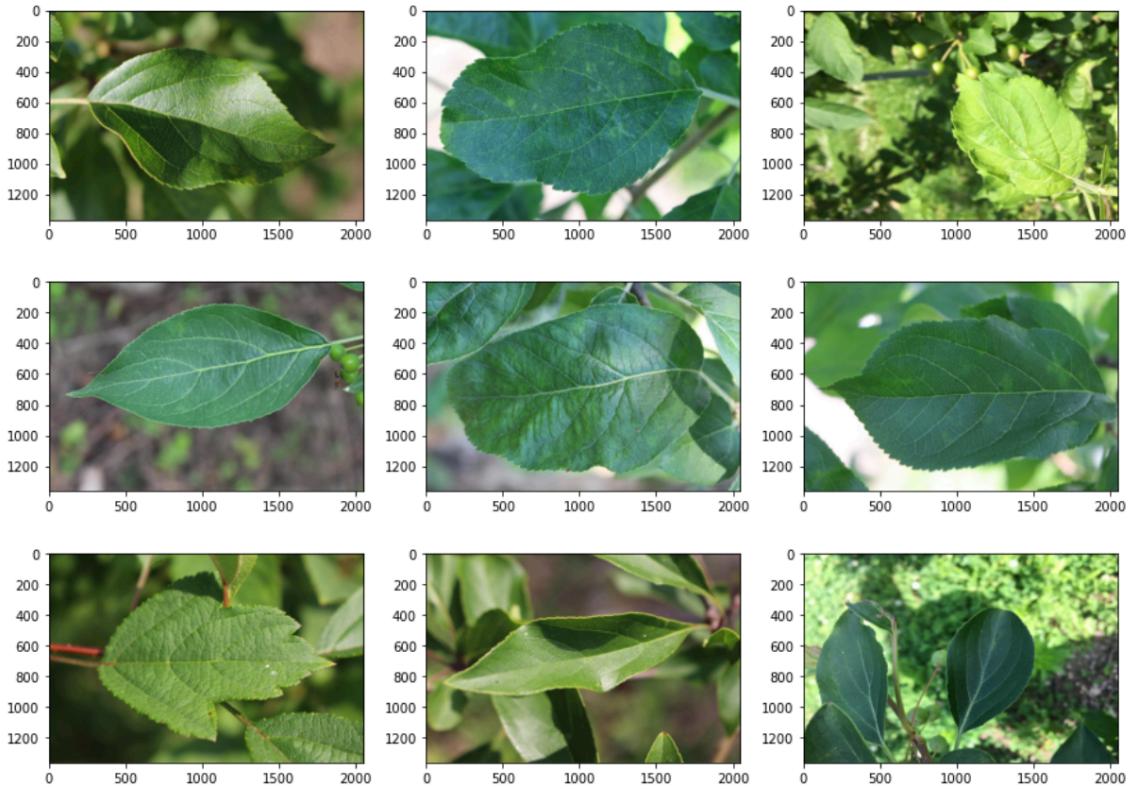
def show_image(img_ids, rows=3, cols=3):
    assert len(img_ids) <= rows*cols

    plt.figure(figsize=(15, 15))
    grid = gridspec.GridSpec(rows, cols)

    for idx, img_id in enumerate(img_ids):
        img_path = f'{data_path}/images/{img_id}.jpg'
        image = Image.open(img_path)
        ax = plt.subplot(grid[idx])
        ax.imshow(image)
```

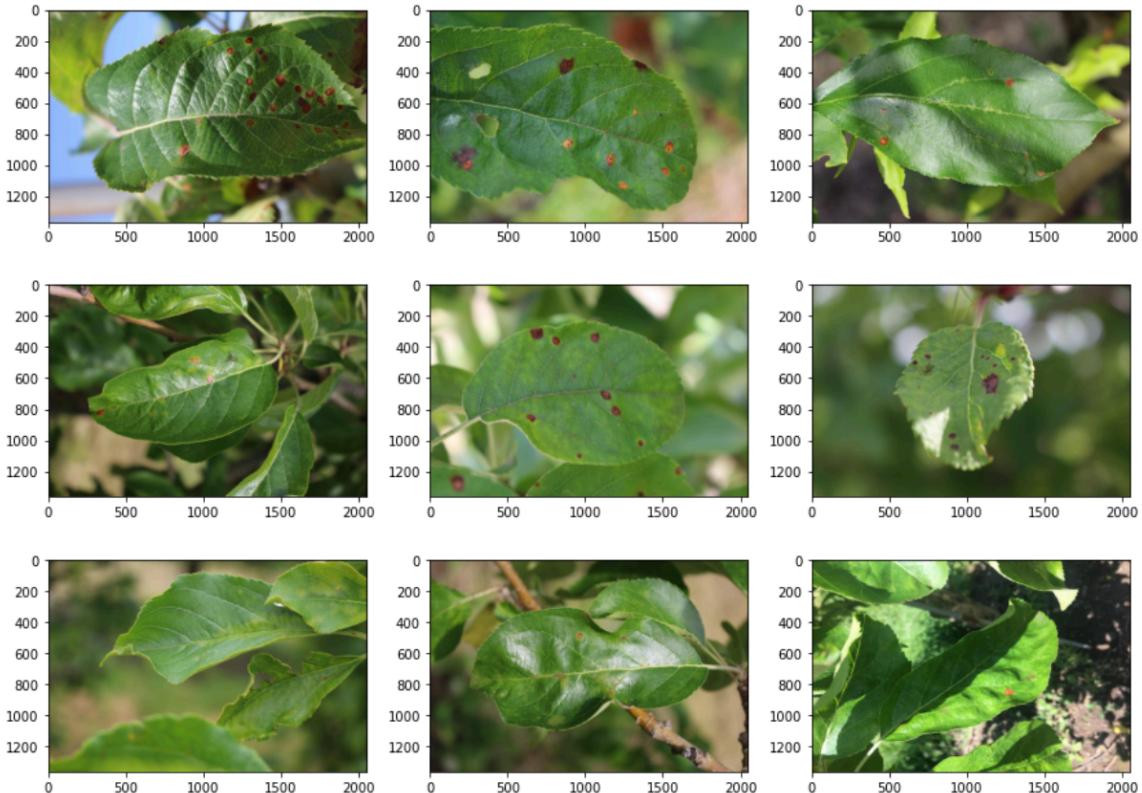
```
healthy_img_ids = train_healthy.index[-9:]
multiple_diseases_img_ids = train_multiple_diseases.index[-9:]
rust_img_ids = train_rust.index[-9:]
scab_img_ids = train_scab.index[-9:]
```

```
show_image(healthy_img_ids) # show healthy leaf
```



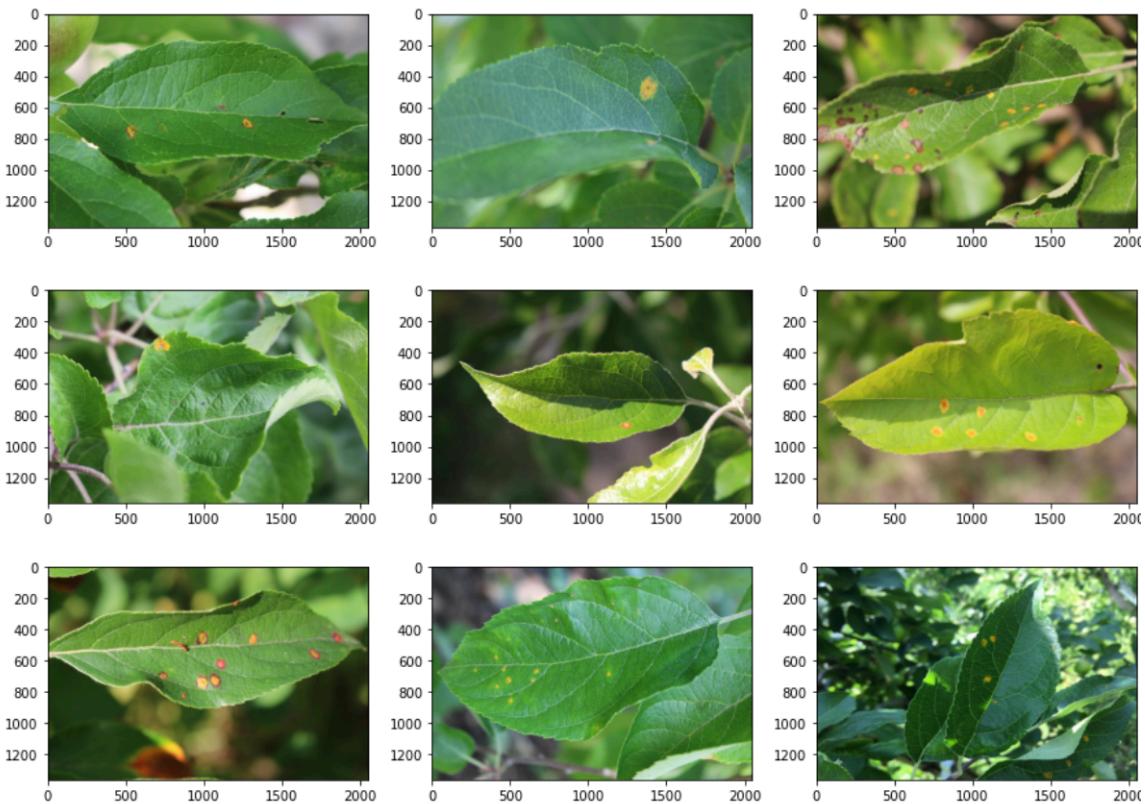
각 label 별 이미지를 살펴보자. 먼저 healthy에 속하는 나뭇잎을 살펴보면 위와 같다.

```
show_image(multiple_diseases_img_ids) # show leaf with multiple diseases
```



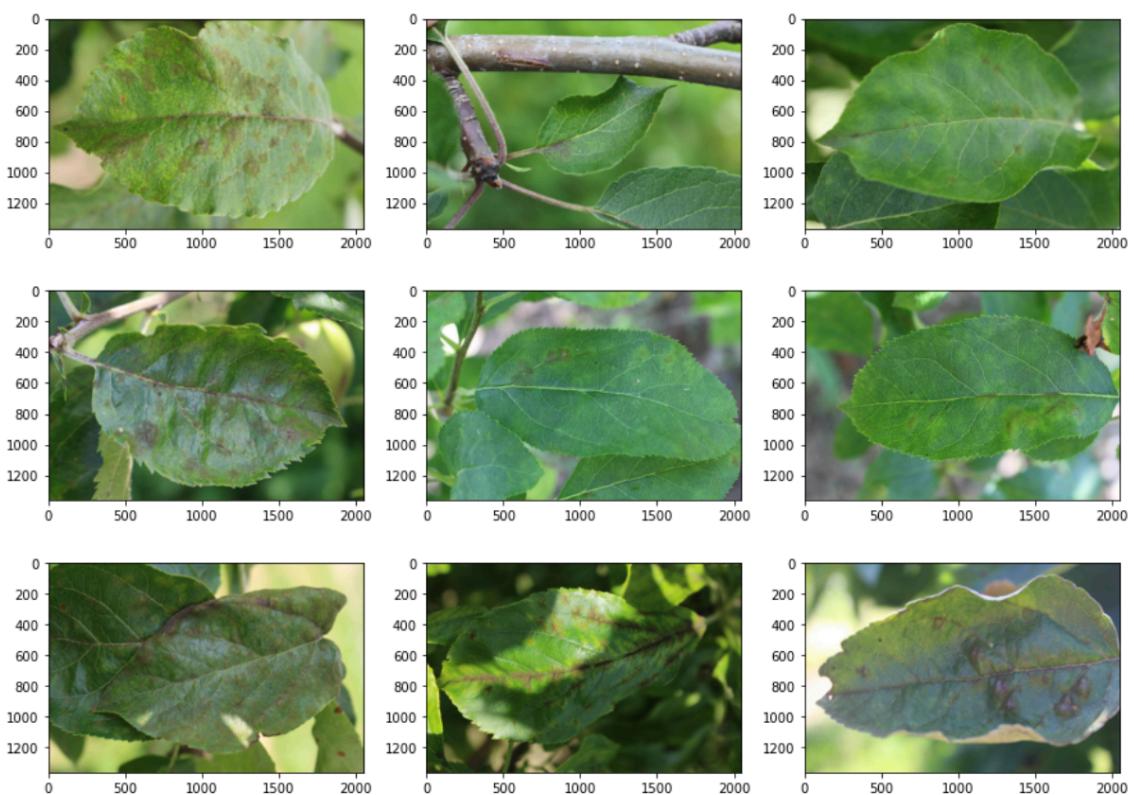
Multiple diseases에 속하는 나뭇잎을 살펴보면 위와 같다.

```
: show_image(rust_img_ids) # show leaf with rust
```



rust에 속하는 나뭇잎을 살펴보면 위와 같다.

```
: show_image(scab_img_ids) # show leaf with scab
```



scab에 속하는 나뭇잎을 살펴보면 위와 같다.

```
import shutil
import os
from shutil import copyfile

if os.path.exists(data_path+'temp/'):
    shutil.rmtree(data_path+'temp/')

os.mkdir(data_path+'temp/')

# train directory
os.mkdir(data_path+'temp/train')
os.mkdir(data_path+'temp/train/healthy')
os.mkdir(data_path+'temp/train/multiple_diseases')
os.mkdir(data_path+'temp/train/rust')
os.mkdir(data_path+'temp/train/scab')

# validation directory
os.mkdir(data_path+'temp/valid')
os.mkdir(data_path+'temp/valid/healthy')
os.mkdir(data_path+'temp/valid/multiple_diseases')
os.mkdir(data_path+'temp/valid/rust')
os.mkdir(data_path+'temp/valid/scab')
```

후에 ImageDataGenerator의 flow_from_directory function을 사용할 예정이기 때문에 train과 valid 폴더를 생성하고 각 label별로 label 이름의 하위폴더를 생성하였다.

```
import numpy as np
SOURCE = data_path+'images/'

TRAIN_DIR = data_path+'temp/train/'

for index, data in train_set.iterrows():
    label = train_set.columns[np.argmax(data)]
    filepath = os.path.join(SOURCE, index + ".jpg")
    destination = os.path.join(TRAIN_DIR, label, index + ".jpg")
    copyfile(filepath, destination)

for subdir in os.listdir(TRAIN_DIR):
    print(subdir, len(os.listdir(os.path.join(TRAIN_DIR, subdir))))
```

multiple_diseases 73
healthy 413
rust 497
scab 473

Train의 하위폴더에 각 data를 옮기면 위와 같다.

```
VALID_DIR = data_path+'temp/valid/'

for index, data in valid_set.iterrows():
    label = train_set.columns[np.argmax(data)]
    filepath = os.path.join(SOURCE, index + ".jpg")
    destination = os.path.join(VALID_DIR, label, index + ".jpg")
    copyfile(filepath, destination)

for subdir in os.listdir(VALID_DIR):
    print(subdir, len(os.listdir(os.path.join(VALID_DIR, subdir))))
```

multiple_diseases 18
healthy 103
rust 125
scab 119

Valid의 하위 폴더에 각 data를 옮기면 위와 같다.

```
import random
import numpy as np
import os
import cv2
import glob
from PIL import Image
import PIL.ImageOps

num_augmented_images = 350

file_path = data_path+'temp/train/multiple_diseases/'
file_names = os.listdir(file_path)
total_origin_image_num = len(file_names)
augment_cnt = 1

for i in range(1, num_augmented_images):
    change_picture_index = random.randrange(1, total_origin_image_num-1)
    file_name = file_names[change_picture_index]

    origin_image_path = data_path+'temp/train/multiple_diseases/' + file_name
    image = Image.open(origin_image_path)
    random_augment = random.randrange(1,3)

    if(random_augment == 1):
        augment1_image = image.transpose(Image.FLIP_LEFT_RIGHT)
        augment1_image = augment1_image.rotate(random.randrange(-20, 20))
        augment1_image.save(file_path+ str(augment_cnt) + '.jpg')

    elif(random_augment == 2):
        augment2_image = image.transpose(Image.FLIP_TOP_BOTTOM)
        augment2_image = augment2_image.rotate(random.randrange(-20, 20))
        augment2_image.save(file_path + str(augment_cnt) + '.jpg')

    augment_cnt += 1

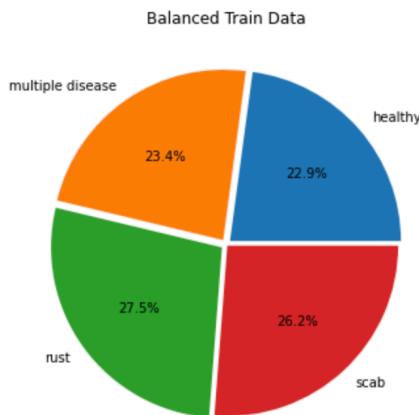
for subdir in os.listdir(TRAIN_DIR):
    print(subdir, len(os.listdir(os.path.join(TRAIN_DIR, subdir))))
```

```
multiple_diseases 422
healthy 413
rust 497
scab 473
```

앞서 multiple diseases의 data가 적어 데이터들이 불균형함을 확인했다. 따라서 data augmentation을 통해 train set의 각 label 별 data 비율을 균등하게 한다.

```
healthy_b = len(os.listdir(os.path.join(TRAIN_DIR, 'healthy')))
multiple_diseases_b = len(os.listdir(os.path.join(TRAIN_DIR, 'multiple_diseases')))
rust_b = len(os.listdir(os.path.join(TRAIN_DIR, 'rust')))
scab_b = len(os.listdir(os.path.join(TRAIN_DIR, 'scab')))

plt.figure(figsize=(6, 6))
label = ['healthy', 'multiple disease', 'rust', 'scab']
explode = [0.03, 0.03, 0.03, 0.01]
plt.pie([healthy_b, multiple_diseases_b, rust_b, scab_b],
        labels=label,
        autopct='%.1f%%',
        explode=explode)
plt.title("Balanced Train Data")
plt.show()
```



균등해진 train set 분포를 살펴보면 위와 같다.

```

import tensorflow as tf
import keras.preprocessing
from keras.preprocessing import image
from keras.preprocessing.image import ImageDataGenerator

training_datagen = ImageDataGenerator(rescale = 1./255,
                                       rotation_range=30,
                                       width_shift_range=0.2,
                                       height_shift_range=0.2,
                                       shear_range=0.2,
                                       zoom_range=0.2,
                                       horizontal_flip=True,
                                       fill_mode='nearest')

validation_datagen = ImageDataGenerator(rescale = 1./255)

train_generator = training_datagen.flow_from_directory(TRAIN_DIR, target_size=(224,224), class_mode='categorical', batch_size=32)
validation_generator = validation_datagen.flow_from_directory(VALID_DIR, target_size=(224,224), class_mode='categorical', batch_size=32)

Found 1805 images belonging to 4 classes.
Found 365 images belonging to 4 classes.

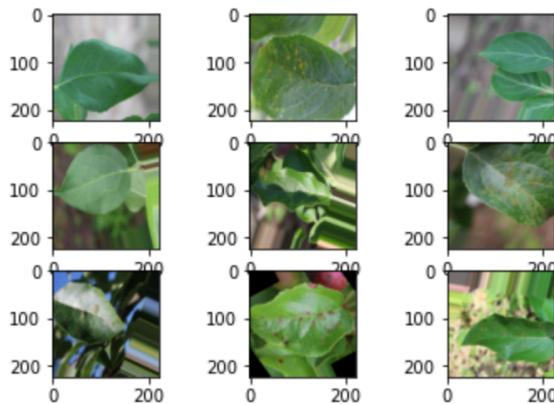
```

앞서 train set의 instance 양이 1805개임을 확인했다. 더 좋은 성능을 얻기 위해 data augmentation을 통해 양을 늘려보았다. ImageDataGenerator을 사용하여 training data generator와 validation data generator를 생성했다. 이때 validation data generator의 경우 255로 나누어서 rescale만 적용했다. 후에 적용할 model의 input size가 224×224인것을 감안하여 resize 또한 적용했다. 후에 fit_generator method를 통해 data generator를 불러와 원하는 만큼 데이터를 생성할 수 있다.

```

# augmentation 후 이미지 확인
for i in range(9):
    batch = train_generator.next()
    plt.subplot(3, 3, i+1)
    plt.imshow(batch[0][i])

```



Augmentation 후 이미지를 확인하면 위와 같다.

4.2. 모델 구축 및 훈련

4.2.1. ResNet

Model Summary

```
def Model():
    model = tf.keras.applications.ResNet50(
    ):

        base = model(weights = "imagenet", include_top = False)
        avg = keras.layers.GlobalAveragePooling2D()(base.output)
        output = keras.layers.Dense(4, activation = "softmax")(avg)
        model = keras.Model(inputs = base.input, outputs=output)

        for layer in base.layers:
            layer.trainable = False

        print('Model complete!')
        return model
```

위와 같이 pre-train 된 keras의 ResNet50 모델을 불러온 다음 output layer를 작성하고 pre-train된 ResNet layer를 freeze하였다.

Callback

```
def callbacks(dir_name = None, model_name='model.h5', patience = 10):
    checkpoint_cb = keras.callbacks.ModelCheckpoint(os.path.join(dir_name, model_name), save_best_only = True)
    early_stopping_cb = keras.callbacks.EarlyStopping(patience = patience, restore_best_weights = True)
    tensorboard_cb = keras.callbacks.TensorBoard(os.path.join(dir_name, "tensorboard"), histogram_freq=1)
    scheduler = keras.callbacks.ReduceLROnPlateau(factor = 0.5, patience = 5)

    return [checkpoint_cb, early_stopping_cb, tensorboard_cb, scheduler]
```

1. checkpoin_cb는 log directory에 한 epoch마다 저장되도록 만들었다. val_loss에 따라 가장 좋은 모델만 저장되도록 하였다.
2. early_stopping_cb는 일정 epochs 동안 val_loss가 더 나아지지 않으면 가장 좋은 weights를 복구하고 학습을 멈춘다.

3. tensorboard_cb는 좋은 시각화 도구인 tensorboard를 활용하기 위해서 사용하였다.
4. scheduler는 학습률을 잘 스케줄링해서 global optimal에 도달할 수 있도록 만들었다.

Train

```
def train(model,
          epochs = 500,
          optimizer = keras.optimizers.Adam(),
          dir_name = None,
          model_name = "model.h5",
          batch_size = 32,
          patience = 10):

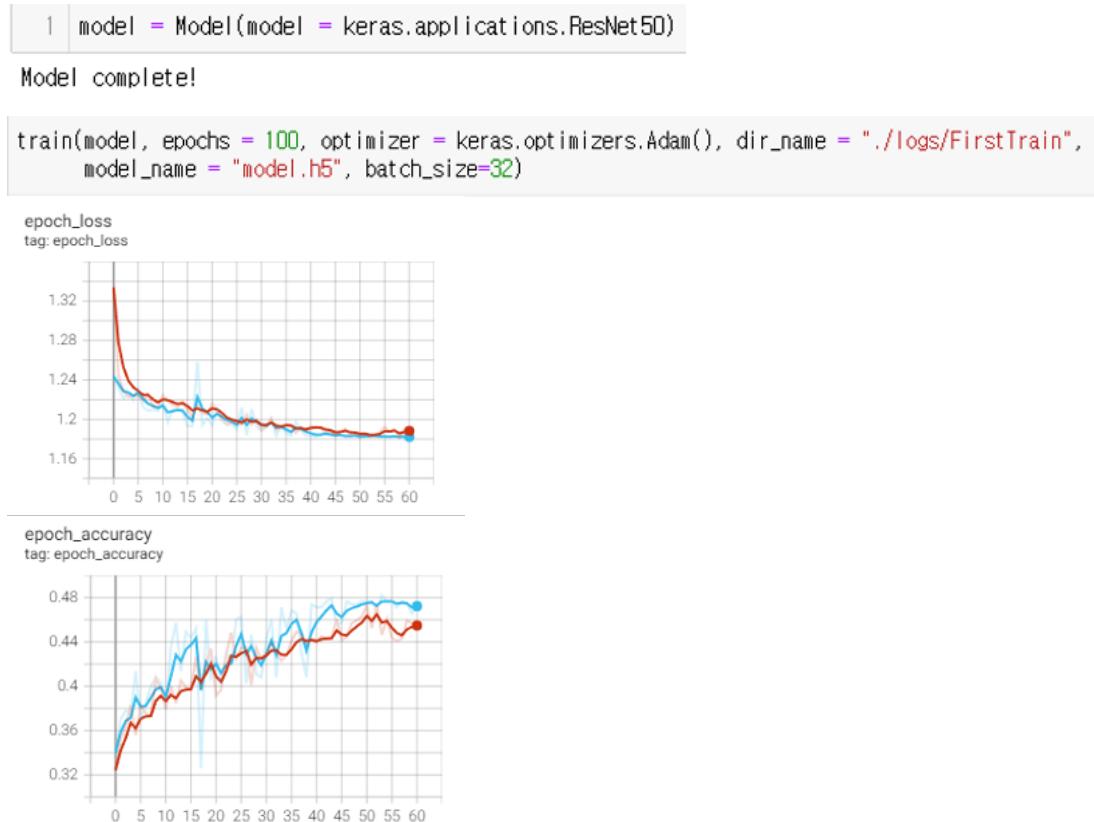
    if dir_name == None:
        dir_name = get_run_logdir()
        os.mkdir(dir_name)

    model.compile(loss="categorical_crossentropy", optimizer = optimizer, metrics=['accuracy'])
    train_generator, validation_generator = DataGenerator(batch_size = batch_size)

    history = model.fit(train_generator, epochs=epochs, steps_per_epoch=56,
                         validation_data = validation_generator, validation_steps=12, callbacks=callbacks(dir_name, model_name, patience))

    return history
```

다음과 같이 train 함수를 만들고 compile과 fit, data generator를 선언하고 사용하였다. 이때 fit을 사용하더라도 generator와 steps_per_epoch을 변수로 주어 fit_generator을 쓴 것과 같은 결과를 낸다. 또 한 앞서 선언한 callback을 사용하였다.



위와 같이 Model 함수로 model을 선언하고 학습한다. 이때 ResNet 층들이 모두 freezing 상태로 반환된다.

```

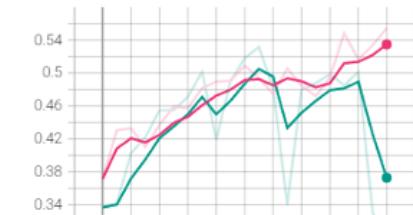
1 model = keras.models.load_model('./logs/FirstTrain/model.h5')

1 # conv5를 모두 풀기
2 for layer in model.layers[143:]:
3     layer.trainable = True

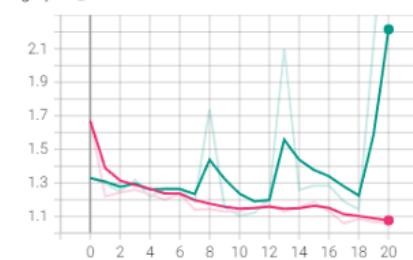
1 train(model, epochs = 100, optimizer = keras.optimizers.Adam(learning_rate=0.0001),
2      dir_name = "./logs/SecondTrain", model_name = "model.h5", batch_size=32)

```

epoch_accuracy
tag: epoch_accuracy



epoch_loss
tag: epoch_loss



처음에 저장한 모델을 불러오고 앞서 3.1.2. 학습 진행순서에서 설명한 ①의 freeze를 풀고 학습한다.

```

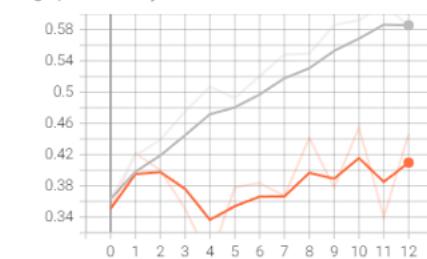
1 model = keras.models.load_model('./logs/SecondTrain/model.h5')

1 # conv4
2 for layer in model.layers[81:]:
3     layer.trainable = True

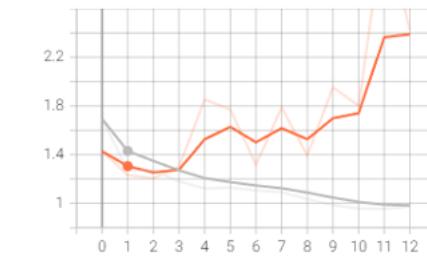
1 train(model, epochs = 100, optimizer = keras.optimizers.Adam(learning_rate=0.0001),
2      dir_name = "./logs/ThirdTrain", model_name = "model.h5", batch_size=32)

```

epoch_accuracy
tag: epoch_accuracy



epoch_loss
tag: epoch_loss

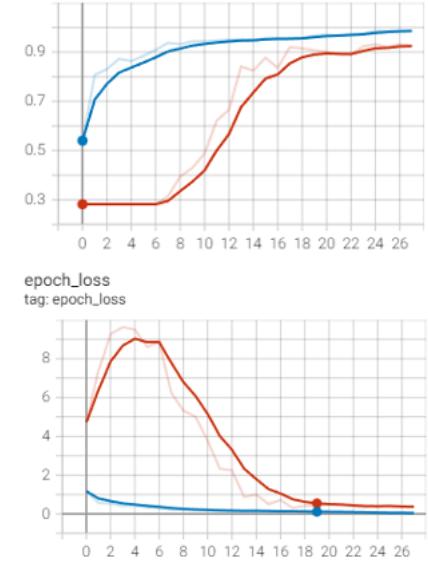


②의 freeze를 풀고 학습하면 위와 같다.

```
model = keras.models.load_model('./logs/ThirdTrain/model.h5')

# conv 3
for layer in model.layers[39:]:
    layer.trainable = True

train(model, epochs = 500, optimizer = keras.optimizers.Adam(learning_rate=1e-5),
      dir_name = "./logs/Fourth_train", model_name = "model.h5", batch_size=32, patience = 10)
```



③의 freeze를 풀고 학습하면 위와 같다.

4.2.2. Keras CNN

```
import keras
from keras.models import Model, Sequential
from keras.layers import Dense, Conv2D, MaxPooling2D, Flatten, Dropout, Activation, BatchNormalization

model = Sequential()

model.add(Conv2D(64, (3,3), activation='relu', input_shape=(224, 224, 3)))
model.add(BatchNormalization())
model.add(MaxPooling2D((2, 2), padding='SAME'))
model.add(Conv2D(64, (3,3), activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D((2,2), padding='SAME'))
model.add(Conv2D(64, (3,3), activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D((2,2), padding='SAME'))
model.add(Dropout(.5))

model.add(Conv2D(128, (3,3), activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D((2,2), padding='SAME'))
model.add(Conv2D(128, (3,3), activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D((2,2), padding='SAME'))
model.add(Dropout(.5))

model.add(Flatten())
model.add(Dense(300, activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(.5))
model.add(Dense(200, activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(.5))
model.add(Dense(100, activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(.5))
model.add(Dense(4, activation='softmax'))
```

Convolution layer와 Pooling layer를 각각 5번씩 사용하는 모델이다. Convolution layer 중 3개의 layer은 64개의 필터를 지나가고 2개의 layer은 128개의 필터를 통과한다. 모든 Convolution layer의 필터 사이즈는 3×3 이며 activation function의 경우 ReLU를 사용한다. Pooling layer의 경우 MaxPooling 방식을 사용했으며 필터 사이즈는 2×2 를 사용했다. 이때 Padding 설정을 SAME으로 두어 zero padding이 이루어지도록 했다.

또한 모든 Convolution layer의 뒤에 BatchNormalization() layer를 추가해 성능이 향상되도록 하였고 3개, 2개의 Convolution layer의 뒤에 Dropout을 배치하여 overfitting의 가능성을 줄였다, 이때 dropout 될 확률은 50%로 설정하였다.

Flatten() layer를 통하여 1차원 배열로 변경한 데이터를 Dense layer 4개를 통하여 배치시켰다. 각각의 layer 뒤에 BatchNormalization과 Dropout을 배치했고 마지막 output layer의 경우 softmax 함수를 적용시켰다.

Model: "sequential"		
Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 222, 222, 64)	1792
batch_normalization (BatchNormalization)	(None, 222, 222, 64)	256
max_pooling2d (MaxPooling2D)	(None, 111, 111, 64)	0
conv2d_1 (Conv2D)	(None, 109, 109, 64)	36928
batch_normalization_1 (BatchNormalization)	(None, 109, 109, 64)	256
max_pooling2d_1 (MaxPooling2D)	(None, 55, 55, 64)	0
conv2d_2 (Conv2D)	(None, 53, 53, 64)	36928
batch_normalization_2 (BatchNormalization)	(None, 53, 53, 64)	256
max_pooling2d_2 (MaxPooling2D)	(None, 27, 27, 64)	0
dropout (Dropout)	(None, 27, 27, 64)	0
conv2d_3 (Conv2D)	(None, 25, 25, 128)	73856
batch_normalization_3 (BatchNormalization)	(None, 25, 25, 128)	512
max_pooling2d_3 (MaxPooling2D)	(None, 13, 13, 128)	0
conv2d_4 (Conv2D)	(None, 11, 11, 128)	147584
batch_normalization_4 (BatchNormalization)	(None, 11, 11, 128)	512
max_pooling2d_4 (MaxPooling2D)	(None, 6, 6, 128)	0
dropout_1 (Dropout)	(None, 6, 6, 128)	0
flatten (Flatten)	(None, 4608)	0
dense (Dense)	(None, 300)	1382700
batch_normalization_5 (BatchNormalization)	(None, 300)	1200
dropout_2 (Dropout)	(None, 300)	0
dense_1 (Dense)	(None, 200)	60200
batch_normalization_6 (BatchNormalization)	(None, 200)	800
dropout_3 (Dropout)	(None, 200)	0
dense_2 (Dense)	(None, 100)	20100
batch_normalization_7 (BatchNormalization)	(None, 100)	400
dropout_4 (Dropout)	(None, 100)	0
dense_3 (Dense)	(None, 4)	404
<hr/>		
Total params: 1,764,684		
Trainable params: 1,762,588		
Non-trainable params: 2,096		

Model summary를 살펴보면 위와 같다.

```
from keras.callbacks import ReduceLROnPlateau
from keras.callbacks import EarlyStopping

LR_reduce=ReduceLROnPlateau(monitor='val_accuracy',
                           factor=.5,
                           patience=10,
                           min_lr=.000001,
                           verbose=1)

ES_monitor=EarlyStopping(monitor='val_loss',
                        patience=20)
```

Callback 함수는 위와 같다.

ReduveLRONPlateaU는 모델의 개선이 없을 경우 learning rate를 조절하여 모델의 개선을 유도하는 콜백함수이다. Val accuracy가 10번의 epoch동안 증가하지 않는 경우 해당 함수를 적용하도록 했다. 또한 min_lr을 설정해 learning rate의 하한선을 지정했다. EarlyStopping은 val_loss를 관찰해 20번의 epoch 동안 증가하지 않는 경우 멈추도록 설정했다.

```
model.compile(loss = 'categorical_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])

checkpoint_cb = tf.keras.callbacks.ModelCheckpoint("cnn.h5", save_best_only=True)

history = model.fit(train_generator,
                     epochs=80,
                     steps_per_epoch=56,
                     validation_data = validation_generator,
                     validation_steps=12,
                     callbacks=[LR_reduce, ES_monitor, checkpoint_cb])
```

Model checkpoint는 모델이 가장 성능이 좋을 때를 저장하도록 했다. loss의 경우 categorical_crossentropy를 사용했고 optimizer로는 RMSProp을 사용했다. 또한 accuracy를 통해 훈련을 모니터링한다. epoch은 총 80번 돌도록 했고 steps_per_epoch과 validation_steps의 경우 전체 데이터수/배치 사이즈, validation 데이터의 수/배치 사이즈로 구한 값이다.

모델의 학습이 끝난 후, 가장 좋은 모델을 불러오기 위해 load_model 함수를 사용했다.

4.2.3. VGG19

```
base_model = VGG19(include_top=False, weights='imagenet', input_shape=(224, 224, 3))
base_model.trainable = False
model = keras.Sequential(base_model)
model.add(layers.Flatten())
model.add(layers.Dense(1024, activation='relu'))
model.add(layers.BatchNormalization())
model.add(layers.Dense(512, activation='relu'))
model.add(layers.BatchNormalization())
model.add(layers.Dense(4, activation='softmax'))
optimizerAdam = Adam(learning_rate=0.00001, amsgrad=True)
model.compile(optimizer=optimizerAdam, loss = 'categorical_crossentropy', metrics=['accuracy'])
model.summary()
print("freeze 후 훈련되는 가중치 수 : ", len(model.trainable_weights))
```

Layer (type)	Output Shape	Param #
<hr/>		
vgg19 (Functional)	(None, 7, 7, 512)	20024384
flatten (Flatten)	(None, 25088)	0
dense (Dense)	(None, 1024)	25691136
batch_normalization (BatchN	(None, 1024)	4096
ormalization)		
dense_1 (Dense)	(None, 512)	524800
batch_normalization_1 (Batch	(None, 512)	2048
hNormalization)		
dense_2 (Dense)	(None, 4)	2052
<hr/>		
Total params: 46,248,516		
Trainable params: 26,221,060		
Non-trainable params: 20,027,456		
<hr/>		
freeze 후 훈련되는 가중치 수 : 10		

Model summary는 위와 같다. pre_train된 keras의 VGG19 모델을 불러 온 다음 Pre_train된 모델은 freeze했다. 더 나은 학습을 위해 2개의 Dense layer를 추가하고 activation function은 ReLU로 설정했다. 마지막 output layer는 4개의 class를 구분한다. optimizer는 RMSProp와 Momentum의 장점을 가진 Adam을 사용했고 이때의 learning rate는 0.00001로 설정하였다. Loss는 Multi-class classification 을 구분하는 one-hot encoding이기 때문에 categorical_crossentropy를 사용했다.

```
earlystopping = tf.keras.callbacks.EarlyStopping(monitor='val_loss',
                                                patience=20)

checkpoint = tf.keras.callbacks.ModelCheckpoint(filepath = data_path + 'best_model.h5',
                                                monitor='val_loss',
                                                verbose=1,
                                                save_best_only=True,
                                                mode='min')
```

callback 함수는 위와 같다. overfitting을 방지하고 가장 좋은 성능(가장 낮은 validation loss)를 가진 모델을 학습시키기 위해 early stopping과 checkpoint 함수를 구현했다. Early stopping은 validation loss가 가장 적은 후 20번 이후로 더 성능의 발전이 없으면 중단하도록 구현하였고, checkpoint는 가장 좋은 validation loss를 가진 모델을 저장하도록 구현하였다.

```
history = model.fit(
    train_generator,
    steps_per_epoch = train_generator.samples // batch_size,
    validation_data=validation_generator,
    validation_steps = validation_generator.samples // batch_size,
    epochs=15)

Epoch 1/15
56/56 [=====] - 472s 8s/step - loss: 1.5860 - accuracy: 0.3768 - val_loss: 1.3389 - val_accuracy: 0.3807
Epoch 2/15
56/56 [=====] - 94s 2s/step - loss: 1.1524 - accuracy: 0.5324 - val_loss: 1.2203 - val_accuracy: 0.4801
Epoch 3/15
56/56 [=====] - 98s 2s/step - loss: 1.0915 - accuracy: 0.5505 - val_loss: 1.0894 - val_accuracy: 0.5568
Epoch 4/15
56/56 [=====] - 98s 2s/step - loss: 1.0286 - accuracy: 0.5956 - val_loss: 0.9924 - val_accuracy: 0.6108
Epoch 5/15
56/56 [=====] - 95s 2s/step - loss: 0.9583 - accuracy: 0.6385 - val_loss: 0.9441 - val_accuracy: 0.6222
Epoch 6/15
56/56 [=====] - 97s 2s/step - loss: 0.9015 - accuracy: 0.6610 - val_loss: 0.9475 - val_accuracy: 0.6278
Epoch 7/15
56/56 [=====] - 96s 2s/step - loss: 0.8569 - accuracy: 0.6723 - val_loss: 0.9297 - val_accuracy: 0.6222
Epoch 8/15
56/56 [=====] - 96s 2s/step - loss: 0.8636 - accuracy: 0.6678 - val_loss: 0.9648 - val_accuracy: 0.6250
Epoch 9/15
56/56 [=====] - 94s 2s/step - loss: 0.8101 - accuracy: 0.6887 - val_loss: 0.9964 - val_accuracy: 0.6222
Epoch 10/15
56/56 [=====] - 94s 2s/step - loss: 0.7985 - accuracy: 0.6926 - val_loss: 1.0635 - val_accuracy: 0.6051
Epoch 11/15
56/56 [=====] - 96s 2s/step - loss: 0.8232 - accuracy: 0.6932 - val_loss: 1.0316 - val_accuracy: 0.6335
Epoch 12/15
56/56 [=====] - 96s 2s/step - loss: 0.7813 - accuracy: 0.7022 - val_loss: 1.0186 - val_accuracy: 0.6136
Epoch 13/15
56/56 [=====] - 96s 2s/step - loss: 0.8107 - accuracy: 0.6825 - val_loss: 0.9874 - val_accuracy: 0.6335
Epoch 14/15
56/56 [=====] - 95s 2s/step - loss: 0.7204 - accuracy: 0.7202 - val_loss: 1.0399 - val_accuracy: 0.6278
Epoch 15/15
56/56 [=====] - 94s 2s/step - loss: 0.7611 - accuracy: 0.7016 - val_loss: 0.9110 - val_accuracy: 0.6648
```

처음 15 epoch은 pre_trained된 모델의 weight가 exploding 우려가 있어 freeze 후 훈련시켰다. train 과 validation 모두 accuracy가 0.65 ~ 0.70 정도로 형성되었다.

```

base_model.trainable = True
model.compile(optimizer=optimizerAdam, loss = 'categorical_crossentropy', metrics=['accuracy'])
model.summary()
print("unfreeze 후 훈련되는 가중치 수 : ", len(model.trainable_weights))

Model: "sequential"

Layer (type)          Output Shape         Param #
=================================================================
vgg19 (Functional)    (None, 7, 7, 512)     20024384
flatten (Flatten)     (None, 25088)        0
dense (Dense)         (None, 1024)         25691136
batch_normalization (BatchN (None, 1024)     4096
ormalization)
dense_1 (Dense)       (None, 512)          524800
batch_normalization_1 (Bac (None, 512)        2048
hNormalization)
dense_2 (Dense)       (None, 4)            2052
=====
Total params: 46,248,516
Trainable params: 46,245,444
Non-trainable params: 3,072
=====
unfreeze 후 훈련되는 가중치 수 : 42

```

그 후 모델의 fine-tuning을 위해 pre_train된 모델을 unfreeze 후 callback 함수를 사용하여 재학습했다.

```

history = model.fit(
    train_generator,
    steps_per_epoch = train_generator.samples // batch_size,
    validation_data=validation_generator,
    validation_steps = validation_generator.samples // batch_size,
    epochs=epochs,
    callbacks = [earlystopping, checkpoint]
)

Epoch 1/100
56/56 [=====] - ETA: 0s - loss: 0.8493 - accuracy: 0.7005
Epoch 00001: val_loss improved from inf to 4.44959, saving model to /content/drive/MyDrive/colab/plant pathology/best_model.h5
56/56 [=====] - 118s 2s/step - loss: 0.8493 - accuracy: 0.7005 - val_loss: 4.4496 - val_accuracy: 0.3466
Epoch 2/100
56/56 [=====] - ETA: 0s - loss: 0.5727 - accuracy: 0.8105
Epoch 00002: val_loss improved from 4.44959 to 1.74820, saving model to /content/drive/MyDrive/colab/plant pathology/best_model.h5
56/56 [=====] - 106s 2s/step - loss: 0.5727 - accuracy: 0.8105 - val_loss: 1.7482 - val_accuracy: 0.5312
Epoch 3/100
56/56 [=====] - ETA: 0s - loss: 0.4186 - accuracy: 0.8658
Epoch 00003: val_loss improved from 1.74820 to 0.81332, saving model to /content/drive/MyDrive/colab/plant pathology/best_model.h5
56/56 [=====] - 105s 2s/step - loss: 0.4186 - accuracy: 0.8658 - val_loss: 0.8133 - val_accuracy: 0.7443
Epoch 4/100
56/56 [=====] - ETA: 0s - loss: 0.3591 - accuracy: 0.8821
Epoch 00004: val_loss improved from 0.81332 to 0.33168, saving model to /content/drive/MyDrive/colab/plant pathology/best_model.h5
56/56 [=====] - 101s 2s/step - loss: 0.3591 - accuracy: 0.8821 - val_loss: 0.3317 - val_accuracy: 0.8920
Epoch 5/100
56/56 [=====] - ETA: 0s - loss: 0.3089 - accuracy: 0.9058
Epoch 00005: val_loss did not improve from 0.33168
56/56 [=====] - 96s 2s/step - loss: 0.3089 - accuracy: 0.9058 - val_loss: 0.5636 - val_accuracy: 0.7841
Epoch 6/100
56/56 [=====] - ETA: 0s - loss: 0.2901 - accuracy: 0.9075
Epoch 00006: val_loss did not improve from 0.33168
56/56 [=====] - 94s 2s/step - loss: 0.2901 - accuracy: 0.9075 - val_loss: 0.4922 - val_accuracy: 0.8693
Epoch 7/100
56/56 [=====] - ETA: 0s - loss: 0.2664 - accuracy: 0.9058
Epoch 00007: val_loss did not improve from 0.33168
56/56 [=====] - 95s 2s/step - loss: 0.2664 - accuracy: 0.9058 - val_loss: 0.4229 - val_accuracy: 0.8551

```

```

Epoch 00024: val_loss improved from 0.21450 to 0.16378, saving model to
/content/drive/MyDrive/colab/plant pathology/best_model.h5
56/56 [=====] - 101s 2s/step - loss: 0.0830 -
accuracy: 0.9724 - val_loss: 0.1638 - val_accuracy: 0.9489

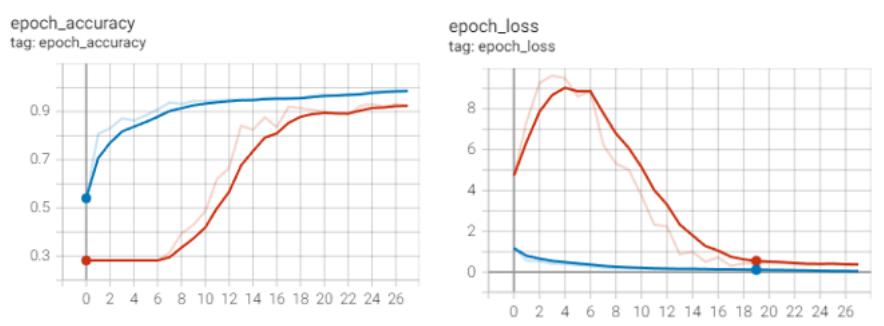
```

100번의 epoch을 설정하였으나 24번째 epoch에서 가장 낮은 validation loss를 가졌고 early stopping으로 인해 44번째 epoch에서 학습이 완료되었으며 checkpoint 함수로 인해 가장 validation loss가 낮은 모델이 저장되었다.

4.3. 최종 성능 평가

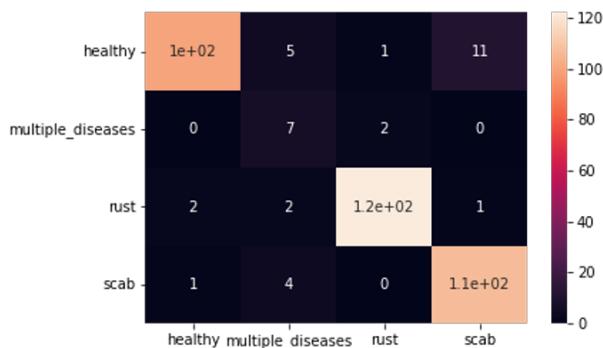
4.3.1. ResNet

Accuracy



위와 같이 validation accuracy는 93%정도가 나왔다. loss도 overfitting 되지 않고 잘 줄어들었다는 것을 볼 수 있다.

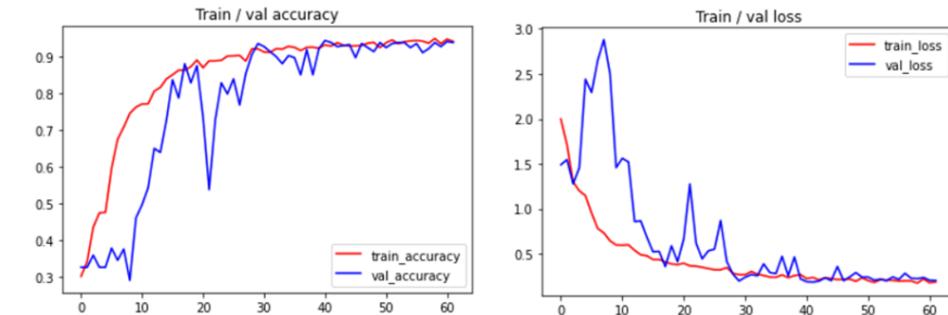
Confusion Matrix



위와 같이 multiple diseases를 잘 분류하지 못하고 있다.

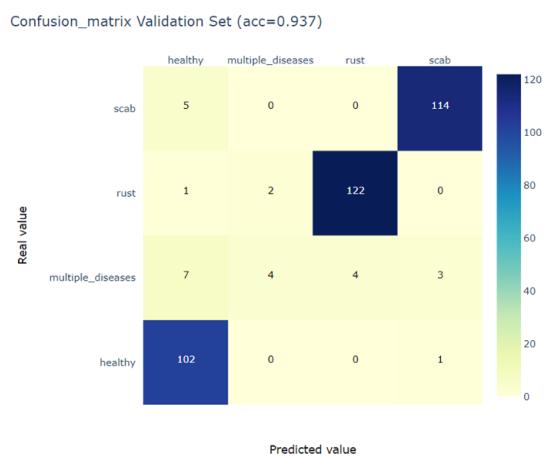
4.3.2. Keras CNN

Accuracy



Validation accuracy는 93.7% 정도가 나왔다.

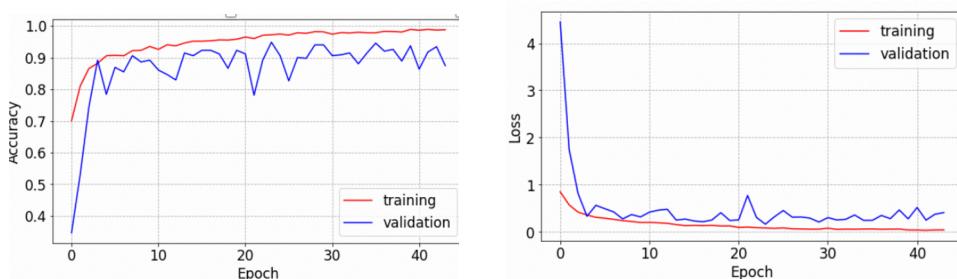
Confusion matrix



Keras CNN 또한 multiple diseases를 잘 분류하지 못했다.

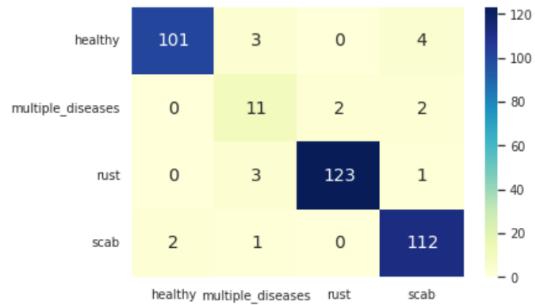
4.3.2. VGG19

Accuracy



Validation accuracy는 94.8% 정도가 나왔다.

Confusion Matrix



VGG19 또한 Multiple diseases class는 제대로 분류하지 못하는 것을 확인할 수 있다.

최종적으로 각 모델 별 성능을 통합해서 살펴보자. Accuracy는 ResNet50의 경우 약 93%, Keras CNN의 경우 93.7%, VGG19의 경우 94.8%로 VGG19가 가장 좋은 성능을 보인다. Confusion matrix를 살펴보아도 healthy, rust, scab의 경우 세 모델 모두 비슷한 수준으로 분류했지만 multiple_diseases의 경우 VGG19가 비교적 잘 분류한 것으로 보인다. 이에 대한 분석을 해보자면 다음과 같다.

VGG19는 Keras CNN과 ResNet50 사이의 복잡도를 가진다. Keras CNN의 경우, 모델 자체의 구조를 직접 만들고 하이퍼파라미터를 적당히 바꿔가면서 가장 좋은 모델을 찾아냈다. VGGNet과 ResNet50의 경우 비슷한 조건을 미리 좋은 성능으로 pre-train 된 모델을 사용하기 때문에 결과에 대한 정확도가 Keras CNN보다 좋게 나올 것으로 예상했다. 하지만 ResNet50과 Keras CNN이 비슷한 성능이 나온 것을 보아 해당 문제가 복잡한 문제가 아니기 때문에 복잡한 ResNet50 모델과 비교적 단순한 Keras CNN의 정확도가 비슷하게 나온 것으로 추정된다. VGG19의 경우 Keras CNN보다는 복잡하지만 ResNet50보다는 단순하기 때문에 더 좋은 성능을 보였다는 결론을 내렸다.

	image_id	healthy	multiple_diseases	rust	scab
0	Test_0	0.000299	1.195602e-03	0.998488	0.000017
1	Test_1	0.012086	1.534922e-01	0.821772	0.012650
2	Test_2	0.000031	2.766452e-05	0.000003	0.999939
3	Test_3	0.999995	4.835730e-10	0.000004	0.000001
4	Test_4	0.045677	6.560407e-01	0.298246	0.000036
5	Test_5	0.999114	2.218740e-04	0.000086	0.000058
6	Test_6	0.999616	1.952261e-06	0.000026	0.000036
7	Test_7	0.000158	1.878625e-04	0.000036	0.999618
8	Test_8	0.000200	1.333534e-03	0.000006	0.998461
9	Test_9	0.000034	1.087651e-05	0.999952	0.000004
10	Test_10	0.000058	3.493321e-02	0.964494	0.000015
11	Test_11	0.998875	7.1117426e-05	0.001032	0.000022
12	Test_12	0.194645	6.822664e-01	0.002786	0.120302
13	Test_13	0.999950	2.638954e-06	0.000024	0.000024
14	Test_14	0.000050	1.825492e-02	0.980964	0.000261
15	Test_15	0.039719	2.709012e-04	0.959957	0.000053
16	Test_16	0.999149	7.038062e-07	0.000847	0.000003
17	Test_17	0.000188	1.063533e-01	0.000112	0.893347
18	Test_18	0.385115	2.279915e-02	0.009222	0.582865
19	Test_19	0.000070	1.377727e-04	0.999788	0.000003

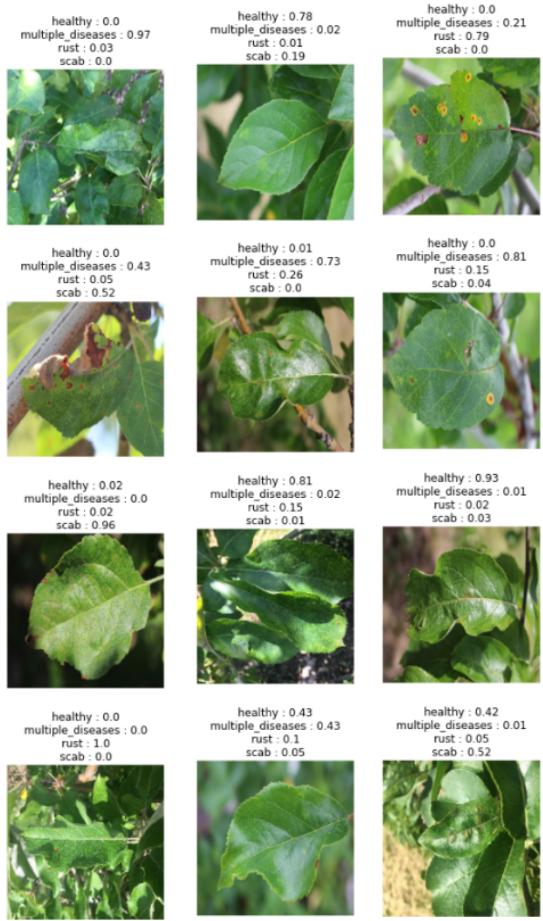
VGG19의 성능이 가장 좋게 나왔기 때문에 해당 모델을 선택했다. Test set에 대한 predict한 것을 살펴보면 위와 같다.

Your most recent submission				
Name	Submitted	Wait time	Execution time	Score
submission.csv	5 minutes ago	1 seconds	0 seconds	0.91559
Complete				
Jump to your position on the leaderboard ▾				

Kaggle Plant-Pathology-2020 competition에서 위 모델의 predict 결과에 대한 score는 0.92가 나왔다.

5. 결과 분석

Multiple Diseases plot



위의 multiple diseases의 image와 ResNet을 이용해 prediction한 class의 확률을 plot해보았다. 위에서 보면 알 수 있듯 healthy의 정도와 rust, scab의 정도의 차이에 의해 classifier가 헛갈릴 수 있다는 생각이 든다. 예를 들어 1열 2행의 그림을 보면 어느정도 healthy 해 보이지만 scab과 rust가 각각 조금 나타난다는 것을 classifier가 잘 나타내고 있는 것을 볼 수 있다. 비슷하게 2행 1열의 그림을 보면 scab이 매우 심하다는 것을 확인할 수 있다. 따라서 scab이 높은 확률로 예측되는 것이 합당해 보인다. 물론 다른 병도 발견이 되었기 때문에 multiple_diseases의 확률이 같이 높게 나온 것 같다.

마지막으로 건강한 잎들과 같이 나왔을 때를 살펴보자. 이 경우는 healthy의 확률이 대단히 올라가는 것을 확인할 수 있다. (ex. 3행 2열, 4행 3열) 이에 따라 과연 multiple diseases의 확률이 1(optimal)하게 되도록 학습을 하는 것이 모델이 정확해지는 것인지, 또는 가능한지에 대한 의문이 들었다.

multiple_diseases class의 데이터 양의 부족 또한 낮은 accuracy에 기여했을 것이다. Validation set에 대해서 data augmentation을 하는 것은 적절하지 못하다고 판단했기 때문에 애초에 multiple_diseases에 속하는 validation data의 양이 적어 비교적 옳게 분류된 data의 개수가 적게 나왔을 수 있다. 하지만 train set에 대해서는 data augmentation을 해서 훈련을 시켰음에도 불구하고 이와 같이 비교적 현저히 낮은 성능이 나온 점을 고려하면 데이터 양의 문제보다는 위에서 언급했듯이 multiple diseases data 자체를 다른

class로 잘못 분류했거나 다른 class의 질병들이 부분적으로 분포해서 각 class 별 추정 확률이 분산되었을 확률이 높다.

위 프로젝트의 목적은 농작물에 영향을 끼치는 질병들에 대해 사람이 직접 판단하여 대응을 하게 되면 비용이 많이 들기 때문에 이를 기계학습으로 판단하여 각 농작물에 대해 적절한 판단을 내리고 그에 따르는 올바른 대응을 내려 해당 문제를 해결하는 비용을 줄일 수 있도록 모델을 만드는 것이었다. 따라서 multiple diseases와 같은 경우 최종 성능 결과만 살펴보았을 때는 유의미한 결과를 도출해내기 힘들 수 있지만 각 class 별 확률을 살펴보면 해당 data에 대한 insight를 얻는데 도움이 될 수 있다. 예를 들어 multiple_diseases class에서 rust와 scab 각각에 대해 하나의 확률이 높게 나온 경우와 rust와 scab 각각 미약하게 나왔을 경우, 각 case에 따라 해당 나뭇잎 문제에 대한 대응의 정도를 확률의 정도에 따라 조절하여 case 별로 적절한 대응을 할 수 있다. 하지만 이와 같은 대응 또한 인건비 상승의 요인이 된다. 앞서 언급 했듯이 우리의 목적은 사람의 개입을 줄여 인건비를 줄이는 것이다. 어떻게 하면 이런 문제를 인건비를 최대한 줄이며 해결할 수 있을까?

이는 Threshold를 설정하여 어느정도 해결할 수 있다. 앞서 살펴보았듯이 mulitple_diseases class의 경우 주변에 건강한 잎들과 같이 나왔을 경우 healthy의 확률이 대단히 올라가는 경우들이 존재함을 확인했다. 이런 경우에서도 사람이 직접 개입하는 것은 비용적인 면에서 효율이 떨어지기 때문에 threshold를 설정하여 모델과 softmax 함수를 통과시킨 data가 healthy의 확률이 90%(threshold) 이하이고 다른 class의 확률이 높은 경우가 나온다면 사람이 개입하여 판단하는 방법을 채택할 수 있다.

본 레포트에서는 VGG19를 이용한 모델의 성능이 가장 좋게 나왔다. 이에 더하여 위와 같이 적절한 threshold를 설정하여 얻은 결과를 이용하면 보다 적은 인건비를 사용하면서도 주어진 식물에 대한 insight를 얻을 수 있다. 이를 확장하여 사과 나뭇잎이 아닌 다른 농작물을 모니터링할 때 역시 사용할 수 있다. 이와 같이 농작물 생산에 기계학습을 활용하여 모니터링함으로써 다양한 농작물 잎의 질병을 판별하거나 이에 더하여 줄기나 흙의 상태 역시 모니터링하여 인건비를 줄이면서도 해당 문제여부 판단에 걸리는 시간을 줄인다면 해당 농작물에 대해 적절한 대처를 보다 빠르게 계획할 수 있어 손실 역시 최소화할 수 있다. 따라서 이를 이용하여 식물 병리학 분야의 발전에 도움이 될 것으로 판단한다.

본 레포트 관련 전체 코드는 아래 주소에서 확인할 수 있다.

GitHub: https://github.com/HJSIFEN/9team_Plant-Patholog