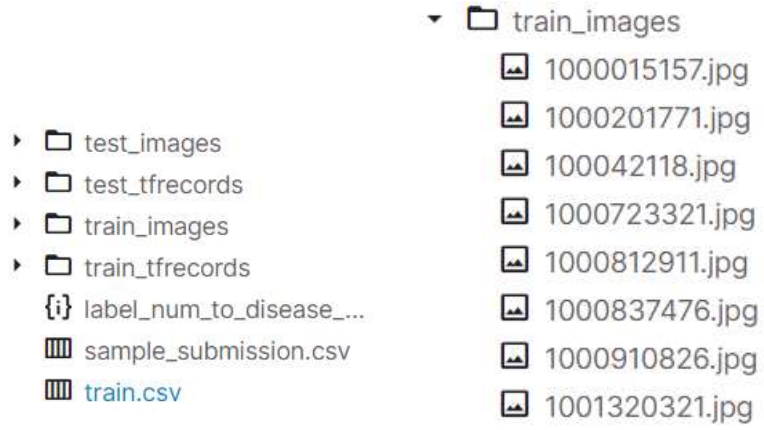
Cassava Leaf Disease Classification

한현수

Problem

Your task is to classify each cassava image into four disease categories or a fifth category indicating a healthy leaf. With your help, farmers may be able to quickly identify diseased plants, potentially saving their crops before they inflict irreparable damage.

Data Fields



train.csv head

	image_id	label
0	1000015157.jpg	0
1	1000201771.jpg	3
2	100042118.jpg	1
3	1000723321.jpg	1
4	1000812911.jpg	3

label_num_to_disease_map.json (172 B)


```
"root" : { 5 items
  "0" : string "Cassava Bacterial Blight (CBB)"
  "1" : string "Cassava Brown Streak Disease (CBSD)"
  "2" : string "Cassava Green Mottle (CGM)"
  "3" : string "Cassava Mosaic Disease (CMD)"
  "4" : string "Healthy"
}
```


Data field는 이렇게 구성되어 있으며 train.csv 에는 image_id와 label로 이루어져 있음을 알 수 있다. 그리고 필사한 노트북에서는 tfrecord 파일을 이용하지 않는다.

* google colab으로 작업하려 했으나 dataset을 Kaggle api로 다운받는 과정에서 image파일이 다운되지 않아 Kaggle notebook으로 진행하였다.

import

```
import numpy as np
import pandas as pd
from PIL import Image
import os
import matplotlib.pyplot as plt
import seaborn as sns
from tqdm import tqdm
from sklearn.utils import shuffle
from sklearn.utils import class_weight
from sklearn.preprocessing import minmax_scale
import random
import cv2
from imgaug import augmenters as iaa
import warnings
warnings.filterwarnings('ignore')

import tensorflow as tf
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.layers import Dense, Dropout, Activation, Input, BatchNormalization, GlobalAveragePooling2D
from tensorflow.keras import layers
from tensorflow.keras.callbacks import ModelCheckpoint, ReduceLROnPlateau, EarlyStopping
from tensorflow.keras.experimental import CosineDecay
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.applications import EfficientNetB3
from tensorflow.keras.layers.experimental.preprocessing import RandomCrop,CenterCrop, RandomRotation
```

Prepare the training and validation data generators

```
training_folder = '../input/cassava-leaf-disease-classification/train_images/'

samples_df = pd.read_csv("../input/cassava-leaf-disease-classification/train.csv")
samples_df = shuffle(samples_df, random_state=42)
samples_df["filepath"] = training_folder+samples_df["image_id"]
samples_df.head()
```

	image_id	label	filepath
9134	2615227158.jpg	4	../input/cassava-leaf-disease-classification/t...
1580	1277648239.jpg	3	../input/cassava-leaf-disease-classification/t...
7304	2305895487.jpg	3	../input/cassava-leaf-disease-classification/t...
13196	336299725.jpg	2	../input/cassava-leaf-disease-classification/t...
5333	1951270318.jpg	2	../input/cassava-leaf-disease-classification/t...

training folder에 있는 image filepath와 train.csv 를 결합한다.

```
training_percentage = 0.8
training_item_count = int(len(samples_df)*training_percentage)
validation_item_count = len(samples_df)-int(len(samples_df)*training_percentage)
training_df = samples_df[:training_item_count]
validation_df = samples_df[training_item_count:]
```

Training data를 80 : 20 비율로 train data와 validation data로 분할한다.

```
batch_size = 8
image_size = 512
input_shape = (image_size, image_size, 3)
dropout_rate = 0.4
classes_to_predict = sorted(training_df.label.unique()) # label
```

Batch size = 8, image shape = 512, 512, 3 dropout rate = 0.4 label = 0~4로 설정한다. 원래 image shape이 600,800,3 인데 efficientnet b3의 input 은 300,300,3으로 resolution scaling하는데, 노트북 작성자는 512,512,3으로 dimension을 설정하는 것이 좋은 결과가 나왔다고 한다.

```
training_data = tf.data.Dataset.from_tensor_slices((training_df.filepath.values, training_df.label.values))
validation_data = tf.data.Dataset.from_tensor_slices((validation_df.filepath.values, validation_df.label.values))
```

이는 training_df와 validation_df를 tensorflow의 dataset으로 변환하는 과정이다. 이때 dataset은 filepath | label 로 구성된다.

```
def load_image_and_label_from_path(image_path, label):
    img = tf.io.read_file(image_path)
    img = tf.image.decode_jpeg(img, channels=3)
    return img, label

AUTOTUNE = tf.data.experimental.AUTOTUNE

training_data = training_data.map(load_image_and_label_from_path, num_parallel_calls=AUTOTUNE)
validation_data = validation_data.map(load_image_and_label_from_path, num_parallel_calls=AUTOTUNE)
```

이는 filepath | label로 구성된 dataset을 Image | label 로 변환하는 과정이다. 여기서 num_parallel_calls = AUTOTUNE 은 dataset에서 training, loading 할 때 가져올 element의 수를 동적으로 조정한다는 매개변수이다.

```
training_data_batches = training_data.shuffle(buffer_size=1000).batch(batch_size).prefetch(buffer_size=AUTOTUNE)
validation_data_batches = validation_data.shuffle(buffer_size=1000).batch(batch_size).prefetch(buffer_size=AUTOTUNE)
```

Prefetch는 데이터 로딩과 학습이 병렬적으로 이루어져서 데이터 로딩에서 병목 현상이 일어나는 것을 방지할 수 있기에 사용되었다.

```
adapt_data = tf.data.Dataset.from_tensor_slices(training_df.filepath.values)
def adapt_mode(image_path):
    img = tf.io.read_file(image_path)
    img = tf.image.decode_jpeg(img, channels=3)
    img = layers.experimental.preprocessing.Rescaling(1.0 / 255)(img)
    return img

adapt_data = adapt_data.map(adapt_mode, num_parallel_calls=AUTOTUNE)
adapt_data_batches = adapt_data.shuffle(buffer_size=1000).batch(batch_size).prefetch(buffer_size=AUTOTUNE)
```

이는 efficient net이 imagenet 데이터셋에 적합한 normalization layer를 이 cassava leaf에 맞는 layer로 바꾸는 과정이다. 즉 image의 r,g,b feature를 0~255에서 0~1로 rescaling한다.

```
data_augmentation_layers = tf.keras.Sequential(
    [
        layers.experimental.preprocessing.RandomCrop(height=image_size, width=image_size),
        layers.experimental.preprocessing.RandomFlip("horizontal_and_vertical"),
        layers.experimental.preprocessing.RandomRotation(0.25),
        layers.experimental.preprocessing.RandomZoom((-0.2, 0)),
        layers.experimental.preprocessing.RandomContrast((0.2,0.2))
    ]
)
```

Data augmentation layer로 데이터를 회전, 뒤집기, 잘라내기를 이용하여 데이터 증강하는 layer이다. 이 augmentation은 training과정에서만 사용되고 inference에선 비활성화된다.

Build the model

```
efficientnet = EfficientNetB3(weights="../input/keras-efficientnetb3-no-top-weights/efficientnetb3_notop.h5",
                              include_top=False,
                              input_shape=input_shape,
                              drop_connect_rate=dropout_rate)

inputs = Input(shape=input_shape)
augmented = data_augmentation_layers(inputs) #Augmentation
efficientnet = efficientnet(augmented) #efficientnet
pooling = layers.GlobalAveragePooling2D()(efficientnet) #globalaveragepooling
dropout = layers.Dropout(dropout_rate)(pooling) #dropout
outputs = Dense(len(classes_to_predict), activation="softmax")(dropout)
model = Model(inputs=inputs, outputs=outputs)

model.summary()
```

```
Model: "functional_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_8 (InputLayer)         [(None, 512, 512, 3)]     0
_________________________________________________________________
sequential (Sequential)      (None, 512, 512, 3)       0
_________________________________________________________________
efficientnetb3 (Functional)  (None, 16, 16, 1536)      10783535
_________________________________________________________________
global_average_pooling2d_1 ( (None, 1536)              0
_________________________________________________________________
dropout_1 (Dropout)          (None, 1536)              0
_________________________________________________________________
dense_1 (Dense)              (None, 5)                 7685
=================================================================
Total params: 10,791,220
Trainable params: 10,703,917
Non-trainable params: 87,303
_________________________________________________________________
```

여기서 efficientnet의 weight를 가져와서 쓰지만 layer들을 freeze하지 않고 사용한다. 그 이유는 efficientnet이 cassava leaf dataset과 다른 imagenet dataset을 사용하여 학습했기 때문에 다시 학습시키는 것이 타당하다 생각해서 다시 학습시켰다고 한다.

```
%%time
model.get_layer('efficientnetb3').get_layer('normalization').adapt(adapt_data_batches)
```

```
CPU times: user 13min 56s, sys: 2min 8s, total: 16min 4s
Wall time: 12min 46s
```

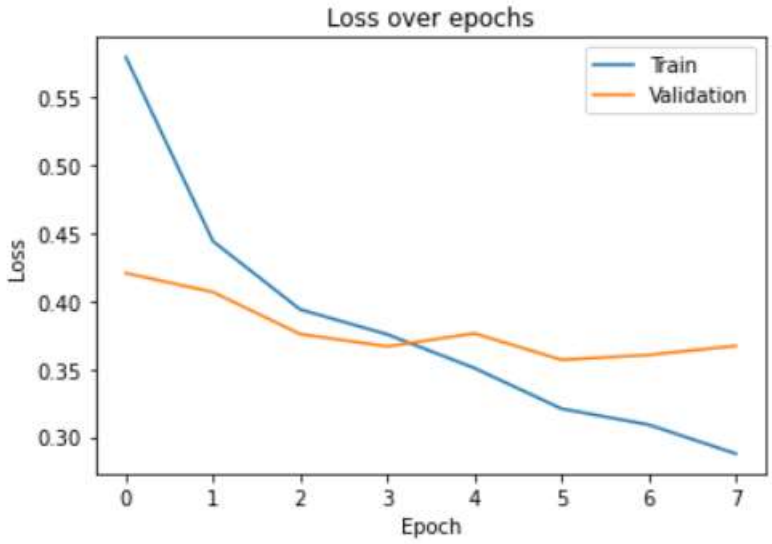
efficientnet의 3번째 layer는 normalization layer다. 이를 cassava leaf dataset에 적합한 normalization layer로 변환하는 과정이다.

```
epochs = 8
decay_steps = int(round(len(training_df)/batch_size))*epochs
cosine_decay = CosineDecay(initial_learning_rate=1e-4, decay_steps=decay_steps, alpha=0.3)

callbacks = [ModelCheckpoint(filepath='best_model.h5', monitor='val_loss', save_best_only=True)]

model.compile(loss="sparse_categorical_crossentropy", optimizer=tf.keras.optimizers.Adam(cosine_decay), metrics=["accuracy"])
```

```
history = model.fit(training_data_batches,
                    epochs = epochs,
                    validation_data=validation_data_batches,
                    callbacks=callbacks)
```

이는 epochs = 8 로 설정하고, cosin_decay는 learning Rate를 cosine 함수를 따라서 변경하는 것으로 ReduceLROnPlateau와 같은 효과를 낸다. Callbacks는 val_loss를 보고 최적의 model만 저장하게 설정하였다. 마지막으로 model을 compile 할 때 loss는 sparse_categorical_crossentropy, optimizer는 Adam으로 설정하였다.

```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Loss over epochs')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='best')
plt.show()
```



학습과정을 시각화 한 것으로 첫 epoch부터 val_loss가 train_loss보다 낮은 걸 확인할 수 있는데, 그 이유는 학습할 때 이미 학습된 network와 weight으로 학습을 시작했기 때문으로 보인다.

Prediction on test images

```
test_time_augmentation_layers = tf.keras.Sequential(
    [
        layers.experimental.preprocessing.RandomFlip("horizontal_and_vertical"),
        layers.experimental.preprocessing.RandomZoom((-0.2, 0)),
        layers.experimental.preprocessing.RandomContrast((0.2,0.2))
    ]
)
```

이는 TTA를 위한 test data를 flip, zoom, contrast 하는 layer이다.

```
model.load_weights("best_model.h5")

def predict_and_vote(image_filename, folder, TTA_runs=4):

    #apply TTA to each of the 4 images and sum all predictions for each local image
    localised_predictions = []
    local_image_list = scan_over_image(folder+image_filename)
    for local_image in local_image_list:
        duplicated_local_image = tf.convert_to_tensor(np.array([local_image for i in range(TTA_runs)]))
        augmented_images = test_time_augmentation_layers(duplicated_local_image)

        predictions = model.predict(augmented_images)
        localised_predictions.append(np.sum(predictions, axis=0))

    #sum all predictions from all 4 images and retrieve the index of the highest value
    global_predictions = np.sum(np.array(localised_predictions),axis=0)
    final_prediction = np.argmax(global_predictions)

    return final_prediction
```

TTA를 진행하고 inference 부분으로 test data를 바로 위의 layer를 통과시켜 4번 반복한 후, 이에 대한 predict를 진행한다. 이때 한 이미지당 4개를 predict하여 각 이미지에 대한 prediction을 더한 후, np.argmax를 이용하여 prediction을 return한다.

```
def run_predictions_over_image_list(image_list, folder):
    predictions = []
    with tqdm(total=len(image_list)) as pbar:
        for image_filename in image_list:
            pbar.update(1)
            predictions.append(predict_and_vote(image_filename, folder))
    return predictions
```

이 함수는 각 이미지를 predict 하고 append하여 prediction을 리스트로 return한다.

```
validation_df["results"] = run_predictions_over_image_list(validation_df["image_id"], training_folder)
```

```
100%|          | 4280/4280 [25:44<00:00,  2.77it/s]
```

이는 위의 함수들을 이용하여 validation data로 prediction을 만든다.

```
true_positives = 0
prediction_distribution_per_class = {"0":{"0": 0, "1": 0, "2":0, "3":0, "4":0},
                                     "1":{"0": 0, "1": 0, "2":0, "3":0, "4":0},
                                     "2":{"0": 0, "1": 0, "2":0, "3":0, "4":0},
                                     "3":{"0": 0, "1": 0, "2":0, "3":0, "4":0},
                                     "4":{"0": 0, "1": 0, "2":0, "3":0, "4":0}}
number_of_images = len(validation_df)
for idx, pred in validation_df.iterrows():
    if int(pred["label"]) == pred.results:
        true_positives+=1
    prediction_distribution_per_class[str(pred["label"])][str(pred.results)] += 1
print("accuracy: {}%".format(true_positives/number_of_images*100))
```

```
accuracy: 88.29439252336448%
```

이는 validation data의 accuracy를 구하고 label과 prediction의 개수를 저장한다.

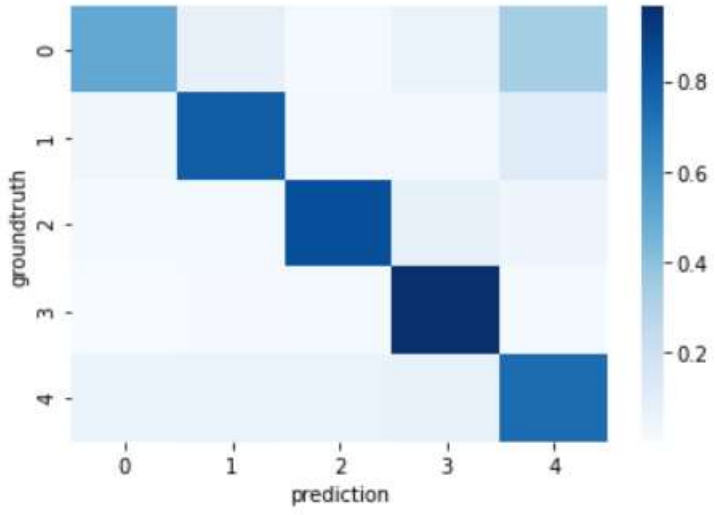
```
prediction_distribution_per_class
```

```
{'0': {'0': 114, '1': 17, '2': 4, '3': 14, '4': 76},
 '1': {'0': 17, '1': 342, '2': 9, '3': 12, '4': 51},
 '2': {'0': 5, '1': 9, '2': 395, '3': 36, '4': 21},
 '3': {'0': 3, '1': 15, '2': 50, '3': 2533, '4': 24},
 '4': {'0': 32, '1': 35, '2': 33, '3': 38, '4': 395}}
```

그 결과 각 label과 prediction의 결과 값들을 볼 수 있다. 각 label 별 accuracy는

0 = 50.7 , 1 = 80.0 , 2 = 84.8 , 3 = 96.5 , 4 = 74.1 로 확인 할 수 있다.

```
heatmap_df = pd.DataFrame(columns={"groundtruth","prediction","value"})
for key in prediction_distribution_per_class.keys():
    for pred_key in prediction_distribution_per_class[key].keys():
        value = prediction_distribution_per_class[key][pred_key]/validation_df.query('label==@key').count()[0]
        heatmap_df = heatmap_df.append({"groundtruth":key,"prediction":pred_key,"value":value}, ignore_index=True)

heatmap = heatmap_df.pivot(index='groundtruth', columns='prediction', values='value')
sns.heatmap(heatmap,cmap="Blues")
```



Conclusion

이를 heatmap을 그려 시각화 하였는데, 확인하니 3,2,1,4,0 순으로 accuracy가 높음을 확인할 수 있었는데, 이 차이는 각 label의 개수가 고르지 않고 각 개수가 3,4,2,1,0 순이기 때문임을 알 수 있다. 하지만 label 3을 제외한 다른 0,1,2의 경우 label 4로 predict한 경우가 많았기 때문에 이 문제를 해결할 방법을 알아 봐야 할 것 같다.

Score = 0.888