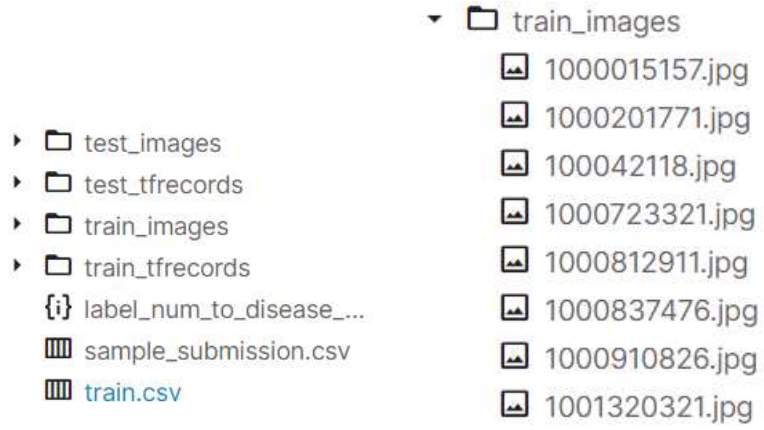
Cassava Leaf Disease Classification

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 field 는 이렇게 구성되어 있으며 train.csv 에는 image_id 와 label 로 이루어져 있음을 알 수 있다. 그리고 필사한 노트북에서는 tfrecord 파일을 이용하지 않는다.

EfficientNet+Aug (tf.keras) for Cassava Diseases

* google colab 으로 작업하려 했으나 dataset 을 Kaggle api 로 다운받는 과정에서 image 파일이 다운되지 않아 Kaggle notebook 으로 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 folder 에 있는 image filepath 와 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 data 를 80 : 20 비율로 train data 와 validation data 로 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 = 0~4 로 설정한다. 원래 image shape 이 600,800,3 인데 efficientnet b3 의 input 은 300,300,3 으로 resolution scaling 하는데, 노트북 작성자는 512,512,3 으로 dimension 을 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_df 와 validation_df 를 tensorflow 의 dataset 으로 변환하는 과정이다. 이때 dataset 은 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 | label 로 구성된 dataset 을 Image | label 로 변환하는 과정이다. 여기서 num_parallel_calls = AUTOTUNE 은 dataset 에서 training, loading 할 때 가져올 element 의 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)
```

Prefetch 는 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 net 이 imagenet 데이터셋에 적합한 normalization layer 를 이 cassava leaf 에 맞는 layer 로 바꾸는 과정이다. 즉 image 의 r,g,b feature 를 0~255 에서 0~1 로 rescaling 한다.

```
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 layer 로 데이터를 회전, 뒤집기, 잘라내기를 이용하여 데이터 증강하는 layer 이다. 이 augmentation 은 training 과정에서만 사용되고 inference 에선 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 efficientnet 의 weight 를 가져와서 쓰지만 layer 들을 freeze 하지 않고 사용한다. 그 이유는 efficientnet 이 cassava leaf dataset 과 다른 imagenet dataset 을 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
```

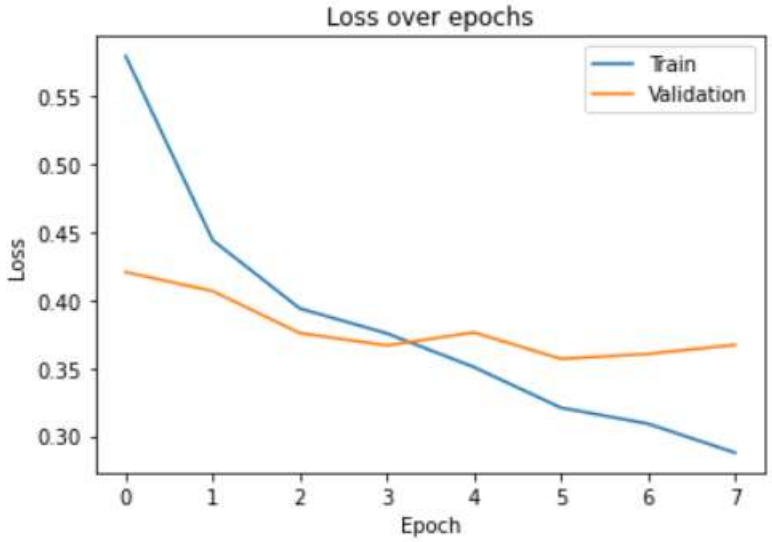
efficientnet 의 3 번째 layer 는 normalization layer 다. 이를 cassava leaf dataset 에 적합한 normalization layer 로 변환하는 과정이다.

```
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_decay 는 learning Rate 를 cosine 함수를 따라서 변경하는 것으로 ReduceLROnPlateau 와 같은 효과를 낸다. Callbacks 는 val_loss 를 보고 최적의 model 만 저장하게 설정하였다. 마지막으로 model 을 compile 할 때 loss 는 sparse_categorical_crossentropy, optimizer 는 Adam 으로 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 epoch 부터 val_loss 가 train_loss 보다 낮은 걸 확인할 수 있는데, 그 이유는 학습할 때 이미 학습된 network 와 weight 으로 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 TTA 를 위한 test data 를 flip, zoom, contrast 하는 layer 이다.

```
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
```

TTA 를 진행하고 inference 부분으로 test data 를 바로 위의 layer 를 통과시켜 4 번 반복한 후, 이에 대한 predict 를 진행한다. 이때 한 이미지당 4 개를 predict 하여 각 이미지에 대한 prediction 을 더한 후, np.argmax 를 이용하여 prediction 을 return 한다.

```
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 append 하여 prediction 을 리스트로 return 한다.

```
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 data 로 prediction 을 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 data 의 accuracy 를 구하고 label 과 prediction 의 개수를 저장한다.

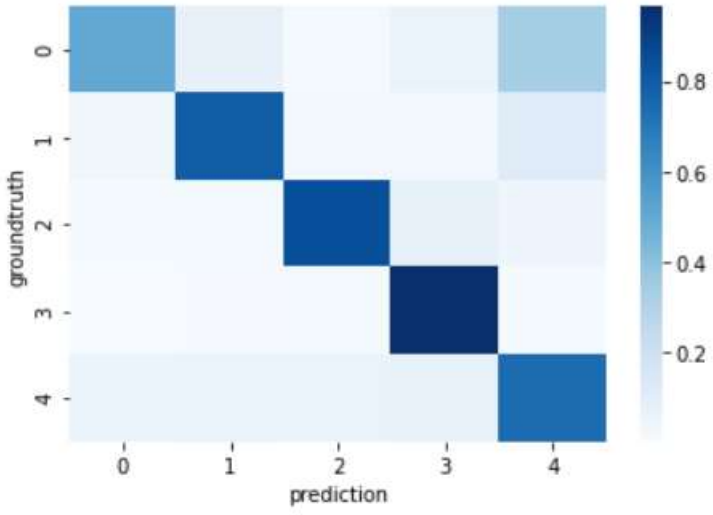
```
prediction_distribution_per_class
```

```
{'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 label 과 prediction 의 결과 값들을 볼 수 있다. 각 label 별 accuracy 는

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

Score = 0.888

Step-by-Step Guide to Denoising Your Labels

Key = denoising label, efficient net b0, image augmentation, TTA(test time augmentation), mixed precision

import

```
#Seed everything for reproducible results
import random
from numpy.random import seed
from tensorflow.random import set_seed

seed_value = 42
random.seed(seed_value)
seed(seed_value)
set_seed(seed_value)
```

Random seed 를 42 로 고정한다.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
import random
import os
import cv2
import sys
from pylab import rcParams
from PIL import Image
from tqdm import tqdm
warnings.filterwarnings('ignore')

import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras import layers
from tensorflow.keras import Input
from tensorflow.keras.models import Model, load_model
from tensorflow.keras.layers import Dense, Flatten, Dropout, Activation, Input, GlobalAveragePooling2D
from tensorflow.keras.callbacks import ModelCheckpoint, ReduceLROnPlateau, EarlyStopping
from tensorflow.keras.applications import EfficientNetB0
from tensorflow.keras.mixed_precision import experimental as mixed_precision
from sklearn.model_selection import StratifiedKFold
```

Mixed precision

```
policy = mixed_precision.Policy('mixed_float16')
mixed_precision.set_policy(policy)
```

Mixed precision 은 모델 학습 시 FP16, FP32 부동 소수점 유형을 상황에 따라 유연하게 사용하여 학습을 더 빠르게 실행하고 메모리를 적게 사용하는 방법이다.

참고 : https://brstar96.github.io/dl%20training%20tip/event&seminar/NVIDIAAIConf_session1/

Augmentation

```
batch_size=32
image_size=300

input_shape = (image_size, image_size, 3)
target_size = (image_size, image_size)
```

```
img_augmentation = tf.keras.Sequential(
    [
        tf.keras.layers.experimental.preprocessing.RandomCrop(image_size, image_size),
        tf.keras.layers.experimental.preprocessing.RandomFlip("horizontal_and_vertical"),
        tf.keras.layers.experimental.preprocessing.RandomRotation(0.25),
        tf.keras.layers.experimental.preprocessing.RandomContrast(0.2)
    ])
```

이는 Augmentation layer 로 image data 를 augmentation 하는 layer 이다.

Data Generator

```
def DataGenerator(train_set, val_set):

    train_datagen = ImageDataGenerator().flow_from_dataframe(
                    dataframe = train_set,
                    directory='../input/cassava-leaf-disease-classification/train_images',
                    x_col='image_id',
                    y_col='label',
                    target_size=target_size,
                    batch_size=batch_size,
                    shuffle=True,
                    class_mode='sparse',
                    seed=seed_value)

    val_datagen = ImageDataGenerator().flow_from_dataframe(
                dataframe = val_set,
                directory='../input/cassava-leaf-disease-classification/train_images',
                x_col='image_id',
                y_col='label',
                target_size=target_size,
                batch_size=batch_size,
                shuffle=False,
                class_mode='sparse',
                seed=seed_value)

    return train_datagen, val_datagen
```

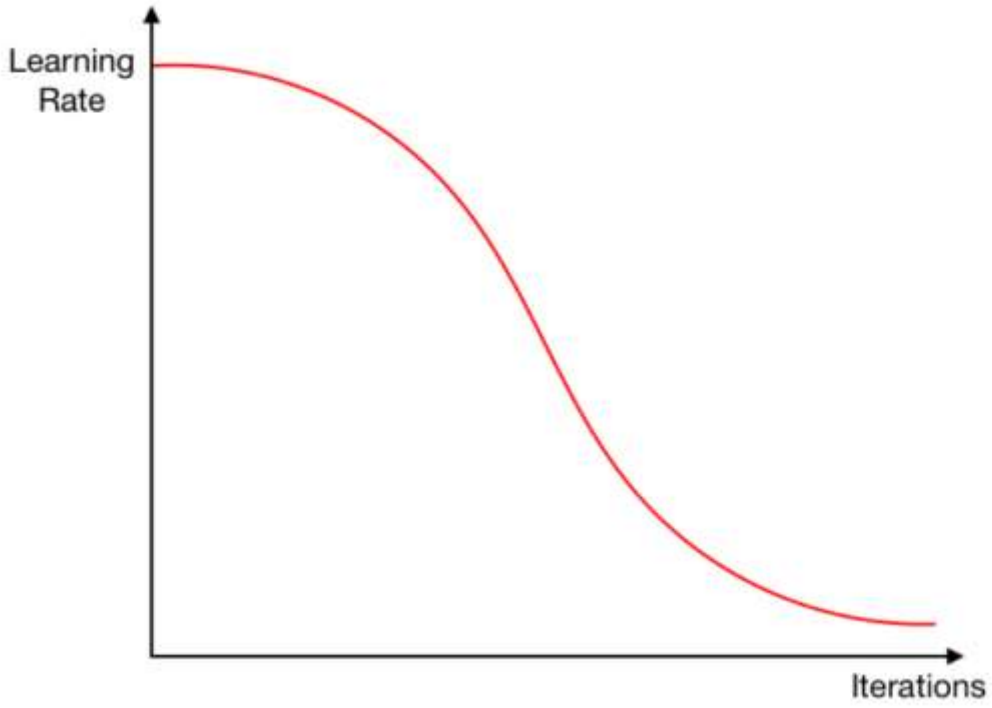
Build model

```
epochs = 3
total_steps = (int(len(df_train)*0.8/batch_size)+1)*epochs

lr = tf.keras.experimental.CosineDecay(initial_learning_rate=1e-3, decay_steps=total_steps)
```

```
def build_model():
    base_model = EfficientNetB0(include_top=False, weights='imagenet', input_shape=input_shape)

    # Rebuild top
    inputs = Input(shape=input_shape)
    base = base_model(inputs)
    pooling = GlobalAveragePooling2D()(base)
    outputs = Dense(5, activation='softmax', dtype='float32')(pooling) #necessary for mixed-precision training to work properly

    # Compile
    model = Model(inputs=inputs, outputs=outputs)
    optimizer = tf.keras.optimizers.Adam(learning_rate=lr)
    model.compile(optimizer=optimizer, loss='sparse_categorical_crossentropy', metrics=['accuracy'])
    return model
```

이 모델에서 기본 base model 은 efficient net b0 를 사용하였고, 특이한 점은 outputs layer 에서 dtype 을 float32 로 설정하는데, 이 이유는 mixed precision 을 사용하여 원활한 학습을 위해서이다. 그리고 learning rate 는 cosine decay , loss 는 sparse_categorical_crossentropy 로 모델을 컴파일한다.

Cosine Decay



Cosine Decay 란 learning rate 를 위와 같은 cosine 함수에 맞춰 줄여 나가는 learning rate 설정 방법이다.

Stratified 5-Fold Cross Validation

```
fold_number = 0
n_splits = 5
oof_accuracy = []

tf.keras.backend.clear_session()
skf = StratifiedKFold(n_splits=n_splits, random_state=seed_value)
for train_index, val_index in skf.split(df_train["image_id"], df_train["label"]):
    train_set = df_train.loc[train_index]
    val_set = df_train.loc[val_index]
    train_datagen, val_datagen = DataGenerator(train_set, val_set)
    model = build_model()
    print("Training fold no.: " + str(fold_number+1))

    model_name = "effnetb0 "
    fold_name = "fold.h5"
    filepath = model_name + str(fold_number+1) + fold_name
    callbacks = [ModelCheckpoint(filepath=filepath, monitor='val_accuracy', save_best_only=True)]

    history = model.fit(train_datagen, epochs=epochs, validation_data=val_datagen, callbacks=callbacks)
    oof_accuracy.append(max(history.history["val_accuracy"]))
    fold_number += 1
    if fold_number == n_splits:
        print("Training finished!")
```

학습시에, 학습 데이터를 5 개로 나누어 학습하고 각 fold 별로 모델을 저장한다.

```
print("Average Out-Of-Fold Accuracy: {:.2f}".format(np.mean(oof_accuracy)))
```

```
Average Out-Of-Fold Accuracy: 0.87
```

이렇게 학습시에 각 fold 당 average accuracy 는 0.87 로 나왔다.

Load model

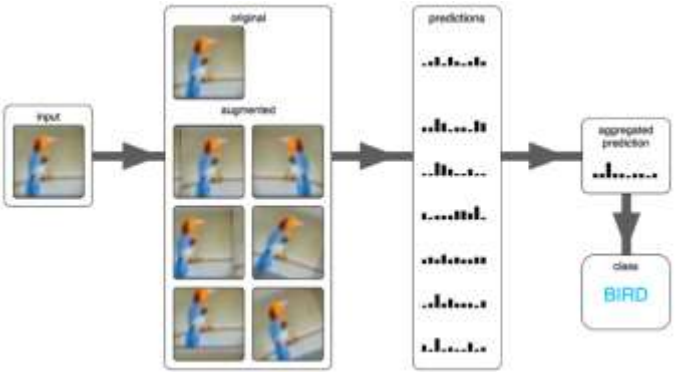
```
# First we load our models
models = []
for i in range(5):
    effnet = load_model("./effnetb0 " + str(i+1) + "fold.h5")
    models.append(effnet)

model_one = models[0]
model_two = models[1]
model_three = models[2]
model_four = models[3]
model_five = models[4]
```

각 fold 별로 model 을 load 한다.

```
# Then we get our validation data
df = pd.read_csv("../input/cassava-leaf-disease-classification/train.csv")
val_list = []

skf = StratifiedKFold(n_splits=5, random_state=seed_value)
for train_index, val_index in skf.split(df["image_id"], df["label"]):
    val_list.append(val_index)

one_fold = df.loc[val_list[0]]
two_fold = df.loc[val_list[1]]
three_fold = df.loc[val_list[2]]
four_fold = df.loc[val_list[3]]
five_fold = df.loc[val_list[4]]
```

각 fold 별로 val_data 를 형성한다.

Test Time augmentation



TTA 는 test data 를 augmentation 하여 predict 를 진행하는 것이다.

```
tta = tf.keras.Sequential(
    [
        tf.keras.layers.experimental.preprocessing.RandomFlip("horizontal_and_vertical"),
        tf.keras.layers.experimental.preprocessing.RandomRotation(0.25),
        tf.keras.layers.experimental.preprocessing.RandomContrast(0.2)
    ]
)
```

```
def duplicate_image(img_path, image_size=image_size, tta_runs=2):

    img = Image.open(img_path)
    img = img.resize((image_size, image_size))
    img_height, img_width = img.size
    img = np.array(img)

    img_list = []
    for i in range(tta_runs):
        img_list.append(img)

    return np.array(img_list)
```

TTA layer 를 만들고 TTA predict 를 진행하기 위해 image 를 np.array 로 return 하는 함수이다.

```
def predict_with_tta(image_filename, folder, tta_runs=2):

    #apply TTA to each of the 3 images and sum all predictions for each local image
    localised_predictions = []
    local_image_list = duplicate_image(folder+image_filename)
    for local_image in local_image_list:
        local_image = tf.expand_dims(local_image,0)
        augmented_images = [tta(local_image) for i in range(tta_runs)]
        predictions = model.predict(np.array(augmented_images[0]))
        localised_predictions.append(np.sum(predictions, axis=0))

    #sum all predictions from all 3 images and retrieve the index of the highest value
    global_predictions = np.sum(np.array(localised_predictions),axis=0)
    max_value = max(global_predictions)
    final_prediction = np.argmax(global_predictions)

    return [final_prediction, max_value, global_predictions]
```

TTA predict 하는 함수로, image2 개를 TTA layer 에 넣어 predict 값을 더하여 argmax 로 final_predcition 을 결정한다. max_value 는 이후 label denoising 을 위한 값이다.

```
train_folder = "../input/cassava-leaf-disease-classification/train_images/"
train_image = "1000015157.jpg"
predictions = predict_with_tta(train_image, train_folder)

print("Predicted Label: ", predictions[0])
print("Predicted Label Value: ", predictions[1])
print("Predicted One-Hot Label: ", predictions[2])
```

```
Predicted Label:  0
Predicted Label Value:  1.0710019
Predicted One-Hot Label:  [1.0710019  0.23761785 0.07196406 0.12470569 0.49471045]
```

위 함수에 임의의 data 를 넣어 사용했을 때 output 결과이다.

```
def predict_image_list(image_list, folder):
    predictions = []
    values = []
    with tqdm(total=len(image_list)) as pbar:
        for image_filename in image_list:
            pbar.update(1)
            predictions.append(predict_with_tta(image_filename, folder)[0])
            values.append(predict_with_tta(image_filename, folder)[1])
    return [predictions, values]
```

이 함수는 predict_with_tta 가 image 1 개를 predict 하지만 image list 를 predict 하는 함수다.

```
model = model_one
placeholder = predict_image_list(one_fold["image_id"], train_folder)
one_fold["pred"] = placeholder[0]
one_fold["value"] = placeholder[1]
one_fold.head()
```

```
100%|          | 4280/4280 [19:14<00:00,  3.71it/s]
```

	image_id	label	pred	value
0	1000015157.jpg	0	4	1.588433
1	1000201771.jpg	3	3	1.999337
2	100042118.jpg	1	4	1.946424
3	1000723321.jpg	1	1	1.985121
4	1000812911.jpg	3	3	1.999931

위 함수의 결과는 이런 형식으로 각 fold 에 저장한다.

Denoising Data

```
threshold = 2*0.8

mask1 = (one_fold["label"] != one_fold["pred"]) & (one_fold["value"] >= threshold)
one_list = one_fold[mask1].index.to_list()

mask2 = (two_fold["label"] != two_fold["pred"]) & (two_fold["value"] >= threshold)
two_list = two_fold[mask2].index.to_list()

mask3 = (three_fold["label"] != three_fold["pred"]) & (three_fold["value"] >= threshold)
three_list = three_fold[mask3].index.to_list()

mask4 = (four_fold["label"] != four_fold["pred"]) & (four_fold["value"] >= threshold)
four_list = four_fold[mask4].index.to_list()

mask5 = (five_fold["label"] != five_fold["pred"]) & (five_fold["value"] >= threshold)
five_list = five_fold[mask5].index.to_list()

combined_list = list(np.unique(one_list + two_list + three_list + four_list + five_list))
```

여기서 threshold 는 label 이 다를 때 2 개의 이미지의 prediction value 중 가장 높은 값이 threshold 보다 높을 때 그 이미지를 label 이 잘못 설정 되었다고 하여 train data 에서 삭제한다.

```
df = df_train.drop(combined_list, axis="index")

df.reset_index(drop=True, inplace=True)

fold_number = 0
n_splits = 5
oof_accuracy = []

tf.keras.backend.clear_session()
skf = StratifiedKFold(n_splits=n_splits, random_state=seed_value)
for train_index, val_index in skf.split(df["image_id"], df["label"]):
    train_set = df.loc[train_index]
    val_set = df.loc[val_index]
    train_datagen, val_datagen = DataGenerator(train_set, val_set)
    model = build_model()
    print("Training fold no.: " + str(fold_number+1))

    model_name = "denoised effnetb0 "
    fold_name = "fold.h5"
    filepath = model_name + str(fold_number+1) + fold_name
    callbacks = [ModelCheckpoint(filepath=filepath, monitor='val_accuracy', save_best_only=True)]

    history = model.fit(train_datagen, epochs=epochs, validation_data=val_datagen, callbacks=callbacks)
    oof_accuracy.append(max(history.history["val_accuracy"]))
    fold_number += 1
    if fold_number == n_splits:
        print("Training finished!")
```

마지막으로 학습하는 부분으로, 위의 기준으로 잘못 labeling 된 data 를 지우고 새로 학습을 한다.

Conclusion

이 노트북에선 labeling 이 잘못된 data 를 이런 방식으로도 추출할 수 있는 방법도 알게 되었고, 간단하지만 경우에 따라 이런 방식으로도 acc 를 높일 수 있다는 것을 알게 되었다.