Term Paper
Data Science and Machine Learning, University of Tokyo

Name	: Hansen Hendra
Student ID	: 37205127
Department	: EEIS (Electrical Engineering and Information Systems)
Faculty	: Graduate School of Engineering

Abstract

Precision agriculture is the utilization of sensors and computer to improve the efficiency and productivity of agriculture works. There are many diseases on crops that can be classified by leaves condition. Unlike fruits condition classification (rotten/fresh) that can be easily manually separated by human visual due to its extreme change in color, structure, even smell, defining crops condition by it leaves is more challenging.

Introduction

Our lab (Oishi Lab) recently started to have research on agricultural field. There are many application on agricultural robotic, from seeding, soil monitoring, crop growth monitoring, to harvesting. One of interesting part is crop condition monitoring to define the health of current crop, so mapping and the needed countermeasure can be done earlier to prevent inefficiency.

Objective

To find the best leaf based plant disease algorithm by exploring:

- Few architecture: shallow CNN, ResNet based
- Few available neural network platform : Tensorflow based Keras, Pytorch
- Performance metric of F1 Score, for unbiased accuracy

Dataset Overview (More information on Data Exploration Section)

Data is provided by Kaggle (https://www.kaggle.com/vipoooool/new-plant-diseases-dataset). [1]

Total data	: 87K images (JPG), with augmentation; 38 classes
Crops	: apple, corn, blueberry, cherry, corn, grape, orange, peach, potato, tomato
Condition	: healthy, scab, rot, rust, blight, mold, virus, etc.

Figure 1 Samples of apple leaves and condition[1]

Train	Validation	Test
70,295 images	17,572 images	33 images (taken later time)

My contribution on this project:
Whole working directories are uploaded on Google Drive :
https://drive.google.com/drive/folders/1GhYnOS_AFat5bCxdtUlFJBjl0BXtB680
Hardware : GPU (Nvidia RTX 2080)

- Implementation of ResNet (Pytorch) and Shallow CNN (Keras) for classification of plant disease:

  1. Do class distribution analysis then define better split for train, validation, and test dataset
  2. Make custom early stopping function inside training function, to stop the training if the loss does not decrease in specific iteration (the origin use a fixed epoch number)
  3. Add the multiclass metric F1 score and do analyze the normal accuracy, macro, weighted macro average. (the origin just use precision)

- Comparison of using Tensorflow Based Keras and Pytorch.
- Provide 2 Python Notebooks:
  1. CNN.ipynb : whole codes from data exploration, data batching, training, and evaluation
  2. Validation_Test_Split.ipynb : code for splitting validation dataset to validation and test dataset

Dataset	Folder containing all dataset
Tests	Dataset for evaluation
Train	Dataset for training
Valid	Dataset for validation
Result	Folder containing models
resnet-model-complete-10-epochs.pth	ResNet Pytorch Model
shallow_cnn_best_model.h5	CNN Keras Model
CNN.ipynb	All CNN Codes
Validation_Test_Split.ipynb	Validation-Test- Split Code

Table of Contents:

The description on this term paper will be based on the Table of Contents of CNN.ipynb as the left figure

Hansen Hendra (37205127)

Data Exploration:

The original dataset from Kaggle the format of classes label as,
[Plant name]__[Disease], by using simple text splitting, a Dataframe is made to review dataset by plant name, disease sorted alphabetically.

	Plant_Disease	Plant	Disease	Number of Images
0	Apple___Apple_scab	Apple	Apple_scab	2016
1	Apple___Black_rot	Apple	Black_rot	1987
2	Apple___Cedar_apple_rust	Apple	Cedar_apple_rust	1760
3	Apple___healthy	Apple	healthy	2008

Figure 2 Dataframe of groups of plant and disease

Another representation of class distribution is by using bar plot as follows:

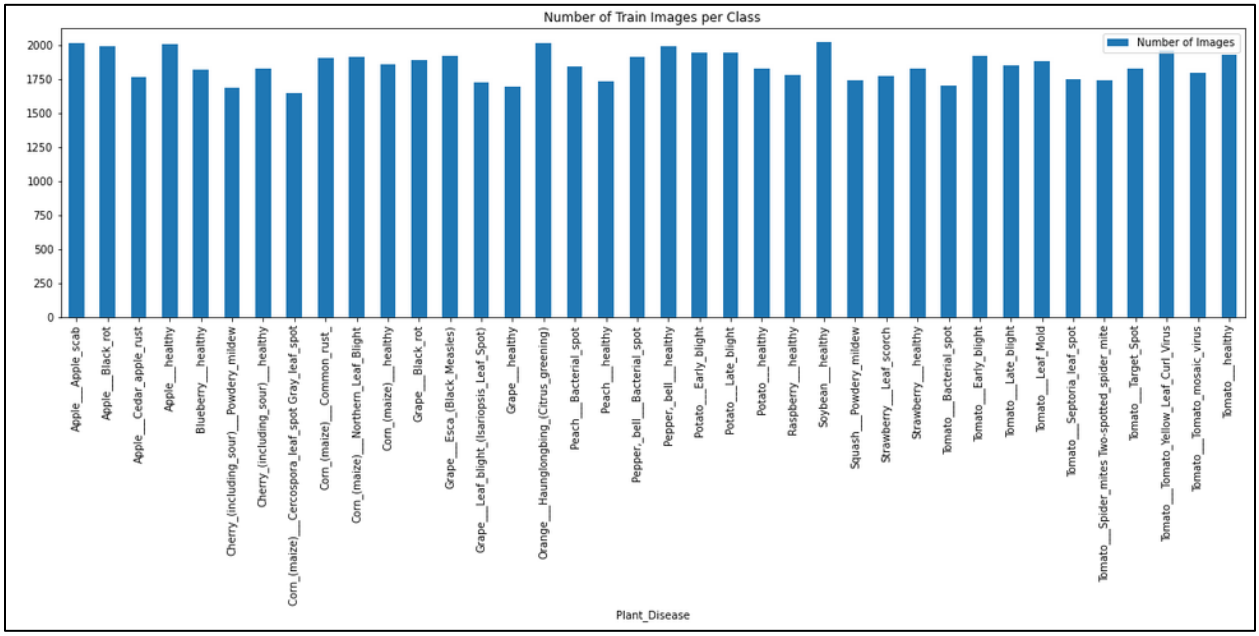


Figure 3 Bar plot of class distribution (overall balanced dataset)

It is really nice that the training dataset has pretty balanced class distribution. It may be balanced because of the augmentation by the Kaggle dataset provider. There are 38 classes to be classified, training a model through balanced dataset is really crucial for a good accuracy.

Split dataset:

Split dataset: we need to correctly split the training, validation, and test dataset.

- The most important thing is **we do not want to leak any information from test dataset to training process**, therefore test dataset cannot be fed to training in any form of information.
- **Validation dataset can be used as only input for evaluation** while model is learning from training dataset, this is to see the generalization ability per iteration training.

It is given from Kaggle dataset this dataset configuration: (All images are 256x256 pixels resolution)

Train	Validation	Test
70,295 images	17,572 images	33 images (taken later time)

By knowing that **there 38 classes overall, we can see that obviously 33 images is insufficient** for final evaluation. Therefore, validation dataset is taken half as test dataset as follows:

Train	Validation	Test
70,295 images	8795 images	8777 images

There is no timestamps or additional information for the files in validation dataset, this will not guarantee that splitting validation into halves of validation and test dataset will not leak information to model training. The most important thing is not to take any training data for test evaluation as it is considered as cheating.

The procedure of splitting validation dataset to validation and training dataset is made on the Python Notebook , "**split valid train.ipynb**"

Overall methodology for Both Shallow CNN and ResNet:

Both ResNet and Shallow CNN is implemented with standard machine learning flowchart:

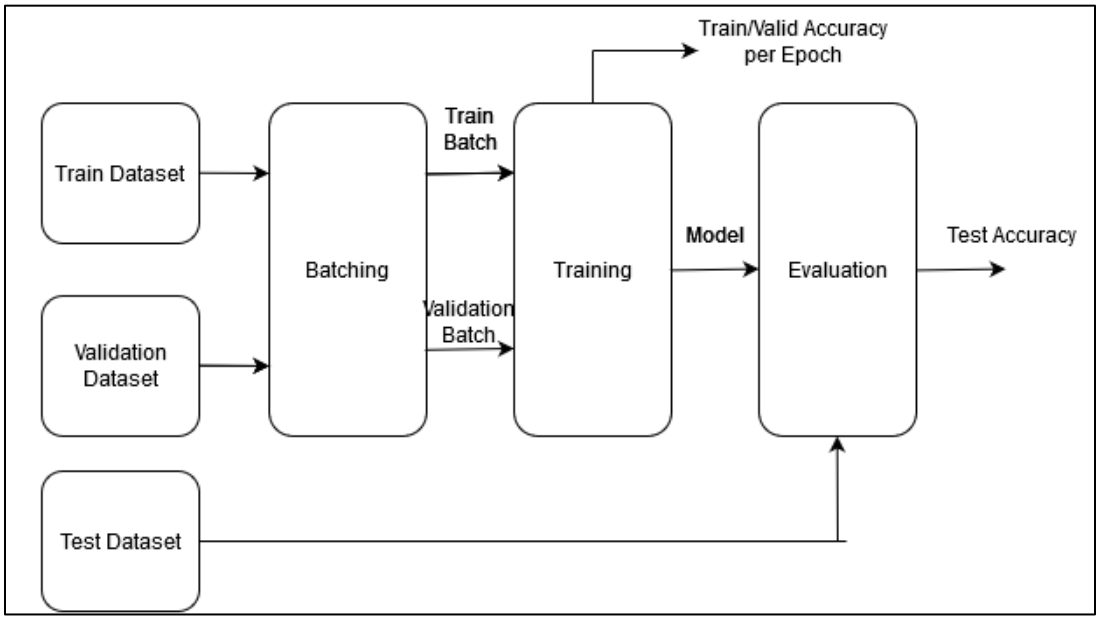
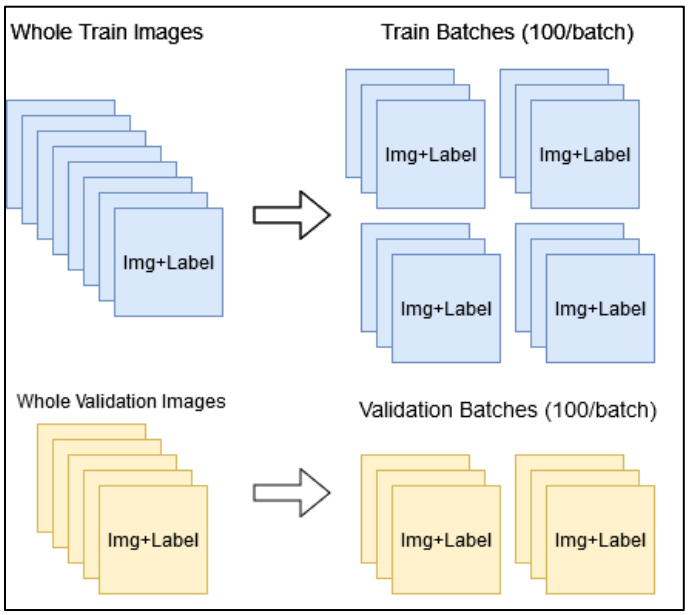


Figure 4 Overall Flowchart for Image Classification

Batching:



Number of images per batch is customized depends on how many data we want to feed the CNN in one iteration of error evaluation.

The larger size of batch will lead to faster training time and much more generalize the pattern of the dataset. Usually, the limitation of batch number is the memory, especially when GPU is used.

The smaller size of batch will take longer time, this is called stochastic gradient descent and even be implemented in online learning when new individual data is often updated and trained to the current model.

Figure 5 Dataset Batching for training model

The number of batch that is used on this project as follows:

ResNet (Pytorch)	Shallow CNN (Keras)
16	**16**

Batching in Keras Tensorflow:

```
## loading training set
training_batch = tf.keras.preprocessing.image_dataset_from_directory(
    './dataset/train',
    seed=42,
    image_size= (img_height, img_width),
    batch_size=batch_size
)
```

Batching in Pytorch:

```
batch_size = 16
train = ImageFolder(train_dir, transform=transforms.ToTensor())
train_dl = DataLoader(train, batch_size, shuffle=True, num_workers=4, pin_memory=True
```

Both batching in Keras and Pytorch take input of directory, size of batch, and some randomizer (seed/shuffle Boolean). There is a difference as follows:

Keras Batching	Pytorch Batching
Does not have option for acceleratiom	Have option to select the core number by setting the num_workers, pin_memory to set to move data to GPU

Shallow CNN:[3]

Training:

Model Architecture (Shallow CNN):

Batch Normalization	
Convolution Layer	Filter : 32 ; Kernel: (3x3); Stride (1,1); Padding : valid
MaxPooling	Kernel : (2x2)
Convolution Layer	Filter : 64 ; Kernel: (3x3) ; Stride (1,1); Padding : valid
MaxPooling	Kernel : (2x2)
Convolution Layer	Filter : 128 ; Kernel: (3x3) ; Stride (1,1); Padding : valid
MaxPooling	Kernel : (2x2)
Flatten	
Dense Layer	256 , ReLu activation
Dense Layer	#Classes, Softmax activation

Adam optimizer is often recommended for CNN training as it is well used in many computer vision papers. The loss categorical crossentropy is a template when we are dealing with multi class classificiation with accuracy metric.

```
MyCnn.compile(optimizer='adam',loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

```
# simple early stopping
es = EarlyStopping(monitor='val_loss', mode='min', verbose=1, patience=2)
PATH_model = './result/shallow_cnn_best_model_second.h5'
mc = ModelCheckpoint(PATH_model, monitor='val_accuracy', mode='max', verbose=1, save_best_only=True)
```

```
%%time
history_shallow_CNN = MyCnn.fit(training_batch,validation_data= validation_batch,epochs = 30,callbacks=[es, mc])
```

I added **implementation of early stopping method**, this will stop the training process when the model did not have a better validation loss on 2 consecutive epochs. This is important as we do not want the model to only strictly memorize the training data structure (overfit) while getting worse on validation/test data evaluation.

Early stopping will be really helpful especially when we have a really large dataset that may take days to for training to converge, rather than waiting to stop the iteration manually we may use the early stopping to automatically stop and save the best model (**implemented easily on Keras with ModelCheckpoint function**), then directly implement other set of hyperparameter to train another model.

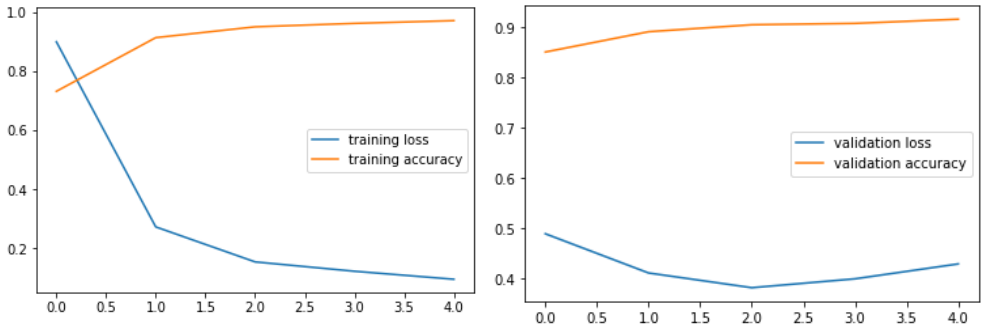


Figure 6 Training performance over epochs (Duration of 14 minutes)

It is clear that the model is learning from training dataset as we see the training loss and accuracy get better as the epoch increases. Meanwhile by looking at the right figure of validation evaluation, we can see that the loss is the lowest on epoch 2, then it got worse on the epoch 3 and 4. The early stopping stops the training as the patience parameter was set as 2.

Evaluation (on test dataset):

	precision	recall	f1-score	support
0	0.22	0.19	0.20	252
1	0.69	0.51	0.59	248
2	0.65	0.09	0.16	220
3	0.14	0.57	0.23	251
4	0.67	0.15	0.25	227
5	0.52	0.07	0.13	210
6	0.24	0.02	0.04	228
7	0.83	0.19	0.30	205
8	0.99	0.46	0.63	238
9	1.00	0.02	0.03	238
10	0.52	1.00	0.69	232
11	0.48	0.39	0.43	236
12	0.57	0.03	0.06	240
13	0.29	0.02	0.03	215
14	0.36	0.85	0.51	211
15	0.00	0.00	0.00	251
16	0.37	0.13	0.19	229
17	0.83	0.75	0.79	216
18	1.00	0.00	0.01	239
19	0.66	0.52	0.58	248
20	0.00	0.00	0.00	242
21	0.25	0.19	0.22	242
22	0.19	0.43	0.27	228
23	0.82	0.04	0.08	222
24	0.08	0.65	0.14	252
25	0.32	0.18	0.23	217
26	0.00	0.00	0.00	222
27	0.00	0.00	0.00	228
28	0.15	0.04	0.06	212
29	0.13	0.69	0.22	240
30	0.23	0.13	0.16	231
31	0.33	0.34	0.34	235
32	0.20	0.11	0.14	218
33	0.47	0.29	0.36	217
34	0.50	0.51	0.51	228
35	0.25	0.00	0.01	245
36	0.88	0.92	0.90	224
37	0.82	0.21	0.34	240
accuracy			0.28	8777
macro avg	0.44	0.28	0.26	8777
weighted avg	0.44	0.28	0.26	8777

Figure 7 F1 Score of Evaluation on Test Dataset (Shallow CNN)

In the term of performance metric of multiclass classification. It may not be the best to just use accuracy in term of correct prediction per prediction number as it does not consider the distribution.
One of a good metric of classification F1 score, however it is more common in binary classification. In the case of multiclass, 1 VS ALL scheme can be adapted to F1 score calculation.

$$F1\ Score = \frac{2x\ Precision\ x\ Recall}{Precision + Recall}$$

$$Precision = \frac{True\ Positive}{True\ Positive + False\ Positive}$$

$$Recall = \frac{True\ Positive}{True\ Positive + False\ Negative}$$

We may choose F1 Score as our metric to be optimized if we want a balance performance. In some cases we may need to focus on either precision/recall as the risk is different on miss detection or wrong accusation.

In this project, the evaluation on dataset is really bad on the three metric. It may be caused by overfitting as the training accuracy is high, or the model does not generalize the essential features of the class distribution.

ResNet:[2]

(based on Kaggle IPYNB : https://www.kaggle.com/vanvalkenberg/cnn-for-plant-disease-detection-92-val-accuracy)

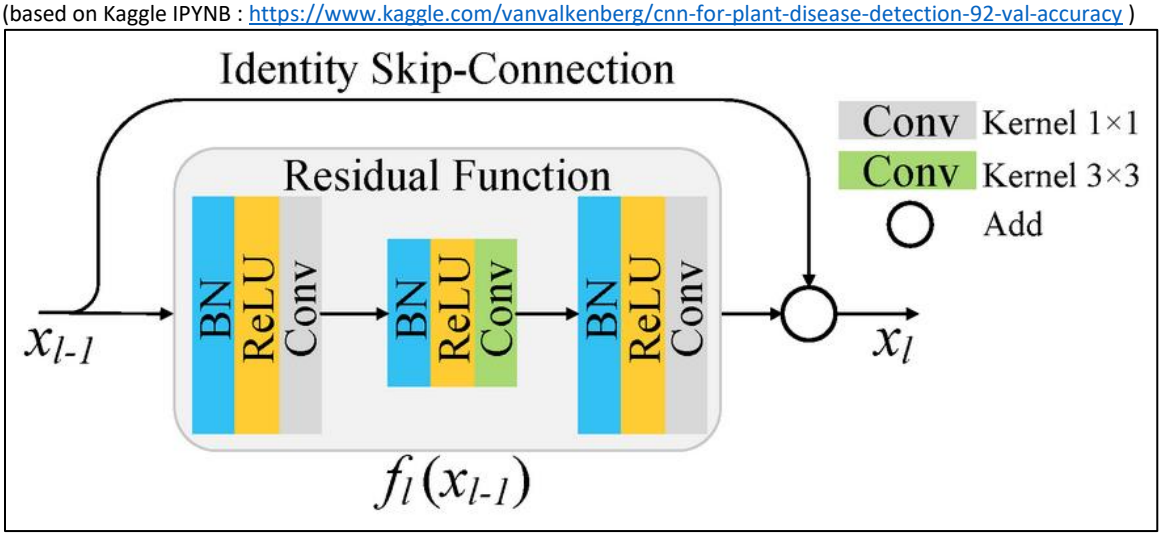


Figure 8 F1 ResNet Residual Function Block Diagram

The special block of ResNet is the Residual Function with Identity Skip-Connection as the pictures above: When the weight is really small, the additional identity is much bigger than forwarded network. It behaves as identity. Meanwhile when the weight is really big, the identity is negligible and the normal behavior of neural network is forwarded.[4]

Training:

Model Architecture (Resnet):

Residual Block:

Convolution Layer	Filter :In1; Kernel: (3x3); Stride (1,1); Padding : 1
Batch Normalization+ReLU	
Convolution Layer	Filter : In2 ; Kernel: (3x3); Stride (1,1); Padding : 1
Batch Normalization+ReLU	
Addition + Input	

Total Architecture:

Convolution Layer	Filter :64; Kernel: (3x3); Stride (1,1); Padding : 1
Batch Normalization+ReLU	
Convolution Layer	Filter : 128 ; Kernel: (3x3); Stride (1,1); Padding : 1
Batch Normalization+ReLU	
MaxPooling	Kernel : (2x2)
Residual Block	Filter (128,128)
Convolution Layer	Filter :256; Kernel: (3x3); Stride (1,1); Padding : 1
Batch Normalization+ReLU	
Convolution Layer	Filter : 512 ; Kernel: (3x3); Stride (1,1); Padding : 1
Batch Normalization+ReLU	
Residual Block	Filter (512,512)
MaxPooling	Kernel : (2x2)
Flatten	
Dense + Softmax	512 – number of classes

More advanced implementation of training process is implmented with ResNet architecture as follows:

- Fit One Cycle method : training with a small starting learning rate, gradually increase to higher learning rate for around 30% of batch, then gradually decrasing to really low value on the remaining training set.
- Weight decay: regularization method to prevent weight from becoming too large by adding weight value to loss minimization
- Gradient Clipping: cut the out of boundary from large gradient value

All the method above can be implemented by the built in function from pytorch:

```
optimizer = opt_func(model.parameters(), max_lr, weight_decay=weight_decay)
# scheduler for one cycle learniing rate
sched = torch.optim.lr_scheduler.OneCycleLR(optimizer, max_lr, epochs=epochs, steps_per_epoch=len(train_loader))
```

Early stopping was implemented again with patience of 2. The model stopped at 10 epochs.

```
if result['val_loss']<min_val_loss:
    min_val_loss = result['val_loss']
    epoch_no_improve=0

if result['val_loss']>min_val_loss:
    epoch_no_improve +=1
    if epoch_no_improve == patience:
        print('Early stopping!' )
        break
```

Unlike Keras which has the functioning early stopping callback, in Pytorch it is recommended to use own implementation in the training process code. I implemented the early stopping as the left snip of code where if the validation loss is not decreasing on 2 (patience) consecutive epochs, the training will stop.

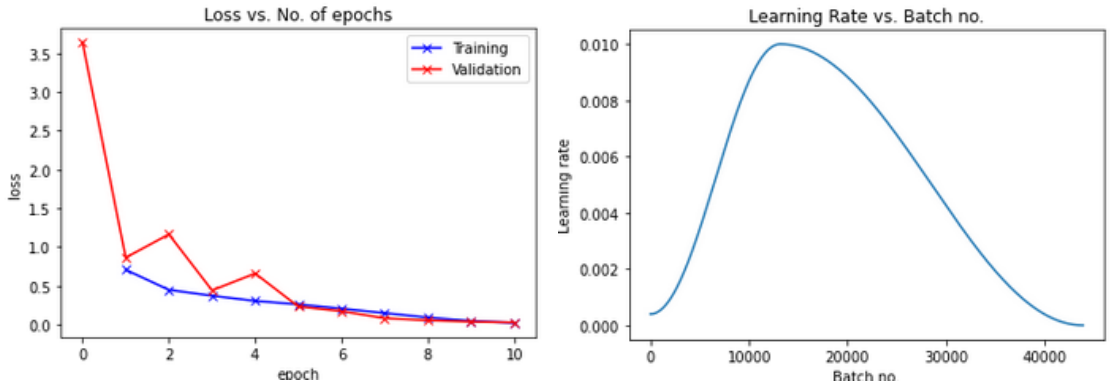


Figure 9 Loss per epoch (ResNet Training) and Learning Rate per Epoch on Fit One Cycle

```
Epoch [9], last_lr: 0.00000, train_loss: 0.0221, val_loss: 0.0280, val_acc: 0.9891
CPU times: user 53min 26s, sys: 38min 22s, total: 1h 31min 48s
Wall time: 1h 28min 14s
```

The training process was taking around 9 minutes per epochs, it takes around 1,5 hours of training much more longer than the previous normal CNN with only less than 15 minutes. The validation accuracy of almost 99% seems to be a major improvement as we will see on test dataset evaluation.

Evaluation (on test dataset):

	precision	recall	f1-score	support
0	0.99	0.99	0.99	252
1	0.99	1.00	0.99	248
2	0.98	1.00	0.99	220
3	0.99	0.99	0.99	251
4	1.00	1.00	1.00	227
5	1.00	1.00	1.00	210
6	1.00	1.00	1.00	228
7	0.99	0.96	0.97	205
8	1.00	1.00	1.00	238
9	0.97	0.99	0.98	238
10	1.00	1.00	1.00	232
11	1.00	1.00	1.00	236
12	1.00	1.00	1.00	240
13	1.00	1.00	1.00	215
14	1.00	1.00	1.00	211
15	1.00	0.99	1.00	251
16	1.00	0.99	0.99	229
17	0.99	1.00	1.00	216
18	0.99	0.98	0.99	239
19	0.99	0.97	0.98	248
20	1.00	1.00	1.00	242
21	0.98	1.00	0.99	242
22	0.99	1.00	0.99	228
23	1.00	1.00	1.00	222
24	1.00	1.00	1.00	252
25	1.00	1.00	1.00	217
26	1.00	1.00	1.00	222
27	1.00	1.00	1.00	228
28	0.95	0.99	0.97	212
29	0.96	0.95	0.95	240
30	0.96	0.97	0.96	231
31	0.99	1.00	1.00	235
32	0.98	0.95	0.97	218
33	1.00	0.99	1.00	217
34	0.99	0.98	0.98	228
35	1.00	1.00	1.00	245
36	1.00	1.00	1.00	224
37	1.00	1.00	1.00	240
accuracy			0.99	8777
macro avg	0.99	0.99	0.99	8777
weighted avg	0.99	0.99	0.99	8777

Figure 7 F1 Score of Evaluation on Test Dataset (ResNet)

It is an impressive accuracy for all three performance metric here. All value is above 95%.

Another detail that should be considered is on the last 2 rows, there are macro avg and weighted macro avg.

Macro-average means all the class will be treated as equal weight as if all the number of dataset is the identical.

$$Macro - avg = \frac{\sum_{i=1}^{class\ N} Acc_i}{\#\ class}$$

Micro-average means all the class will be treated as equal weight as if all the number of dataset is the identical.

$$Weighted - avg = \frac{\sum_{i=1}^{class\ N} Acc_i}{\sum_{i=1}^{class\ N} i}$$

It is clear that the macro and weighted average of the test evaluation result has the the same value of 99% as we can see from the last column “support” the class distribution of test dataset is quite balanced.

The test evaluation of ResNet is much more better than the Shallow CNN shows that the combination of Residual Network and Fit One Cycle outperforms the normal Shallow CNN.

Conclusion:

Keras and Pytorch

- Tensorflow Based Keras and Pytorch is machine learning library that can utilized GPU to implement many kind of CNN. Both has roughly the same function on dataset loading and batching for the training data format.
- Keras was developed earlier than Pytorch. From my experience in coding by both libraries, Keras will use the default device (CPU/GPU) based on the Tensorflow on training process, while Pytorch has the flexibility to assign the data and/or model to GPU or CPU so we may adjust based on our computer specification.
- Pytorch is easier to read and has more complex method implemented in one-liner function. I myself prefer to use Pytorch than Tensorflow for DNN development.

Classification: Shallow CNN and ResNet

- A simple CNN is insufficient to make a good classifier especially on large number multi class classification (38 plant diseases). A deep network like ResNet is more suitable for this task (99% accuracy on F1 score).
- F1 score is one of metric to have comprehensive performance understanding on the model evaluation, so we do not trapped by the biased accuracy.
- Early stopping is a method to stop the training before the model strays away from the generalization ability. This is really helpful when the dataset is huge and we want to keep only the best model.

Reference:

[1] Kaggle Dataset on “New Plant Disease Dataset”, https://www.kaggle.com/vipoooool/new-plant-diseases-dataset/tasks
[2] Kaggle notebook examples ResNet on Plant Disease Dataset, https://www.kaggle.com/atharvaingle/plant-disease-classification-resnet-99-2
[3] Kaggle notebook examples CNN on Plant Disease Dataset, https://www.kaggle.com/vanvalkenberg/cnn-for-plant-disease-detection-92-val-accuracy
[4] Understanding and visualizing ResNets, https://towardsdatascience.com/understanding-and-visualizing-resnets-442284831be8#:~:text=ResNet%20Layers,layers%20remains%20the%20same%20%E2%80%94%204.

CNN Components

Batch Normalization
Transformation to change the mean of output close to 0 and standard deviation close to 1. This is to ensure, some range of values in the dataset does not dominate and skew the distribution
Convolution Layer
Convolution filters in image processing to extract features on some regional frame. • Kernel : size of regional frame to be calculated on the image • Stride : step move to the next kernel convolution • Filter : number of layers per convolution, this leads to number of output size • Padding : the action when the stride and kernel size cut the edge of image, usually does not matter much when stride is 1
MaxPooling
Take 1 max value of the kernel size, will significantly reduce the size
Flatten
Convert 2/3 dimensional data format into 1 dimensional layer, this is to be fed into fully connected layer for classification
Dense Layer
Fully connected layer with output of total neuron same as output classes, usually softmax is used to out the highest value class with highest probability