Portable Apple Leaf Disease Detection System using Convolutional Neural Networks

Luu Thanh Hieu

A thesis submitted to Auckland University of Technology
in partial fulfilment of the requirements for the degree of
Master of Computer and Information Sciences (MCIS)

2021

School of Engineering, Computer and Mathematical Sciences

Abstract

A major issue in agriculture is the need to cope with plant diseases. Adequate expertise and domain knowledge are required to handle diseases effectively in the field. The concurrence of the increasingly portable device ownership, such as smartphones and drones, and the major advances in Machine Learning opens a potential measure for portable devices in disease diagnosis at a low cost. This research studies the challenge and solution of building an apple leaf disease detection system using Deep Learning, which can operate in the field without requiring a plant expert presence. The core of the system adopts recent convolutional neural networks, build up two-stage machine learning application pipeline to diagnose apple leaf disease directly from images captured on the field without causing any damage to plant. In the research, transfer learning and fine turning techniques are considered and applied effectively, helping to reduce training time while pre-trained knowledge is reused. Other factors that are dealt with in this system are optimisation, made it feasible for mobile and edge device running at limited resources. Both original and optimised versions of disease detection model are yielding impressive accuracy score which is more than 0.99. The leaf segmentation model is able to work within the system pipeline as providing visually validated results. The entire system should be built and wrapped in a portable, cross-platform application, as well as deployable and serve as cloud services. This research was encouraged and oriented by the design science research process framework [3].

Keywords: Machine Learning; Deep Learning; Image Recognition; Software Application Development; Apple Disease.

Table of Contents

List of Figures

List of Tables

List of Acronyms

API: Application Programming Interface

BN: Batchnorm

CNN: Convolutional Neural Networks

DSRM: Design Science Research Methodology

DL: Deep Learning

DNN: Deep Neural Network

FC: Fully Connected

FPN: Feature Pyramid Networks

GPS: Global Positioning System

GPU: Graphics Processing Unit

IoU: Intersection over Union

mAP: Mean Average Precision

ML: Machine Learning

R-CNN: Region Based Convolutional Neural Networks

ReLU: Rectified Linear Units

RGB: Red Green Blue

RoIs: Regions of Interest

RPN: Regional Proposal Network

SGD: Stochastic Gradient Descent

SSD: Single Shot MultiBox Detector

TPU: Tensor Processing Unit

Attestation of Authorship

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person (except where explicitly defined in the acknowledgments), nor material which to a substantial extent has been submitted for the award of any other degree or diploma of a university or other institution of higher learning.

Signature: Date: 10 July 2021

Acknowledgment

This research work was completed as one part of the Master of Computer and Information Sciences (MCIS) course at the School of Computer and Mathematical Sciences (SCMS) in the Faculty of Design and Creative Technologies (DCT) at the Auckland University of Technology (AUT) in New Zealand. I would like to deeply thank my wife and family members for their dedicated support and sacrifice for my study and research. I also would like to thank Henry for his crucial suggestion in terms of research direction, and Hector Group gave me chance to experience how an actual machine learning application works in production.

I would first like to express deepest thanks to my supervisor Dr Minh Nguyen at the School of Engineering, Computing and Mathematical Sciences at AUT, for his support throughout the thesis. He has not only provided practical advice to overcome research problems but also inspired me a strong source of creativity and enthusiasm.

Luu Thanh Hieu
Auckland, New Zealand
July 2021

Chapter 1

INTRODUCTION

The first chapter of this thesis consists of five paragraphs. In the first paragraphs, background and motivation of this thesis will be introduced. The aim of this research is to apply ML to deal with the issue of apple leaf disease which impacting food production. The third paragraph discusses the rapid development of ML and DL recently. The fourth paragraph focuses on the benefit if applying DL to diagnose apple leaf disease. Finally, this chapter proposes main research questions after in-depth comprehensive understanding of the relevant literatures and research background.

Plant disease management heavily influences food production, and is becoming more critical [5]. In the crop management category, disease detection significantly impacts to quality and productivity of harvest. However, it requires domain knowledge to identify particular types of disease and prescribe precisely. This experiment is considering Machine Learning (ML) as an adoption to deal with this issue.

In recent decades, there are considerable advances ML and Deep Learning (DL). ML is known as an area of artificial intelligence (AI), which studies computer algorithms to improve automatically for systems. In Computer Vision, their applications brought more improvement in the field of Image Recognition (IR). IR, in the context of Computer Vision, is the ability of systems to analyse images and extract information of objects, places, people, actions for desired answers. Basically, an IR system built up by pretrained models is able takes in new images to analyse and produce desired answers [6-9], as depicted in Figure 1.

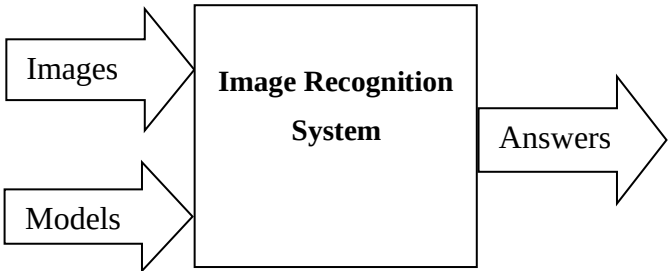

Figure 1: Image Recognition System Process. Input: images, pretrained models. Output: desired answers.

Recently, ML and DL has been applied in various fields, such as self-driving car in transportation [10-12], disease identification in medicine [13-15], automation in surveillance [16-18] and many others [19]. There are many tasks in plant disease management can be bolstered by applying ML and DL, namely analysing apple leaf diseases.

Diagnosing apple leaf diseases benefits both growers and practitioners. Considering the problem context of plant diseases, if diseases are labelled by experts and organised in a

dataset structure, they can be fed into DL models for training. Once the training is completed, the information and knowledge is wrapped and able to assemble into a portable application and deploy to any cloud hosting services. The assembled and published application could be accessed and employed by any growers in the field to analyse diseases without background knowledge required. This research could also be considered as a simple approach suggestion for practitioners to deploy DL models to mobile devices and possible cloud deployment approaches, for either research or practical purposes.

Due to the particularities of plants and fruits, an approach solution for one species may not fit in with others. This research focuses on the portable, cross platform detection system for apple leaf diseases using CNN. Essentially, the followings are questions for our research:

- Solution Reliability and Accuracy: how effective is the DL when applying to detect apple leaf disease? could the plant disease detection model yield precise results in field settings with variation of illumination and background noise?

- Approach Solution: How to utilise theoretical ML models and lab dataset to deal with practical apple leaf disease?

- System Architecture: how to implement a practical apple leaf disease detection system in production using ML?

- Accuracy Preservation: does the portable ML application operate at equivalent accuracy with the full original model?

Chapter 2

RELATED WORK

This chapter describers the background and architectures of convolutional neural network models as well as challenges and difficulties when applying to plant disease detection. The core structure of the apple leaf disease detection system is analysed too and following by undertaken relevant technologies stacks.

2.1 Convolutional Neural Networks

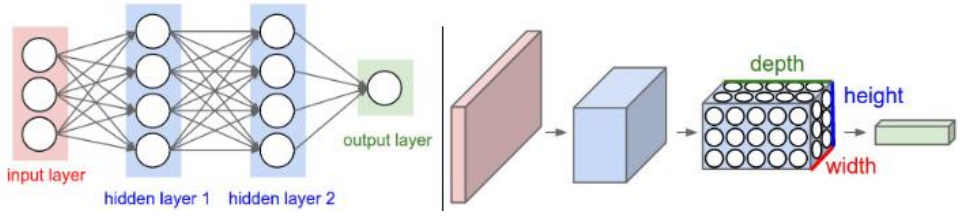

Figure 2: Left: Regular Neural Network. Right: ConvNet [2]

Convolutional Neural Networks (CNN or ConvNet) are popularly used in image classification, produces a crucial breakthrough to computer vision. As the meaning from its name, the network applies a mathematical linear operation called convolution. Unlike a regular neural network which has three layers (input, hidden, and output), a CNN has neurons organized in three dimensions (width, height, depth) [2] as compared in Figure 2. Main types of layers that are usually combined to form a ConvNet architecture are: Convolutional Layer, Rectified Linear Unit, Pooling Layer, and Fully Connected Layer. An example of ConvNet is presented in Figure 3. Some commonly used CNN architectures are AlexNet [20], VGG [21], GoogLeNet [22], ResNet [23], MobileNets [24] and others. MobileNets are lightweight deep neural networks, specifically designed for mobile and embedded devices [24].

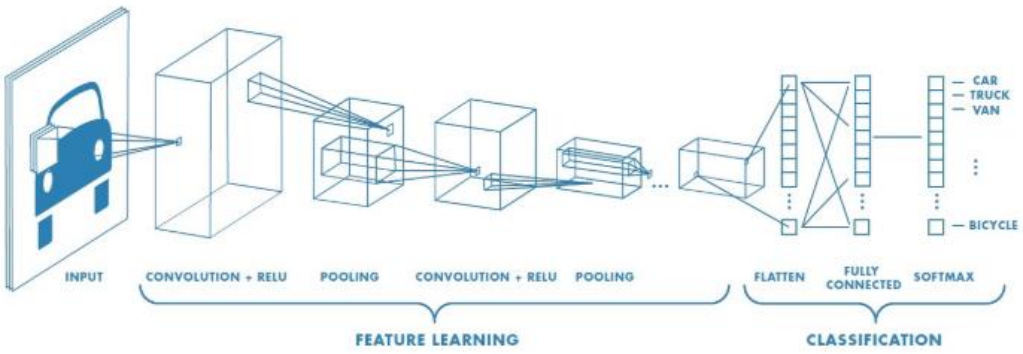

Figure 3: A CNN example [1] formed by various types of layers

2.1.1 Input Image

An image is matrixes of pixels values. The image resolution represents its size and depth, also called dimension or channels. For example, an RGB image comprises of three channels: Red, Green, and Blue, described as a tensor in Figure 4.

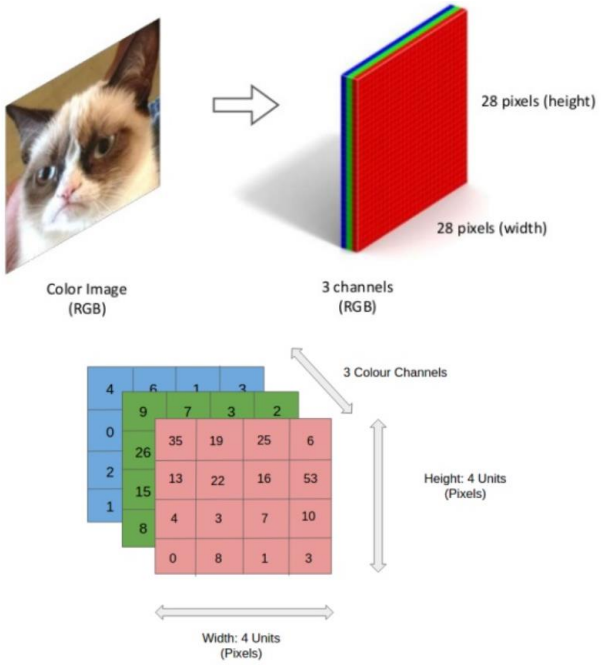

Figure 4: RGB Image Tensor comprises 3 channels: Red, Green, and Blue

2.1.2 Convolutional Layer

Convolutional operation is the first step of a CNN to extract features of the input image by using filters. The filters perform various distinct operations such as edge detection, sharpen, and blur. Each filter in a convolution layer is a collection of kernels and produces only one output channel [25]. When the kernel slides over input channels, the mathematical operation is computed and return corresponding output channels as depicted in Figure 5. These output channels are then summed together to create a single output channel.

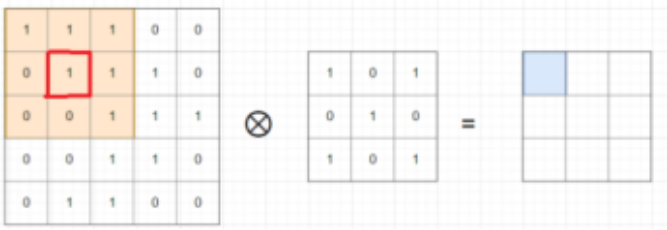

Figure 5: Convolutional Computation. Input and kernel are on the left and right of operator respectively. Single output on the right of equation.

2.1.3 Rectified Linear Units (ReLU) Layer

The most common activation function deployed for the output of the CNN neurons is Rectifier Unit. The purpose of ReLU is a non-linearity operation. The activation function of the input x is defined as in Equation 1.

Equation 1: Activation function

$$f(x) = \max(0, x)$$

The above ReLU function directly return the input if it is positive, otherwise 0 is resulted. This behaviour makes it hard to differentiate at the origin with backpropagation training. That is the reason to utilise a smoother version, namely Softplus in Equation 2. ReLU is a popular choice due to its performance advantage.

Equation 2: Softplus function

$$f(x) = \ln(1 + e^{x})$$

2.1.4 Pooling Layer

Large images give a large number of parameters for computation, severely affect the performance. Spatial pooling technical resolves this problem by reducing the dimensionality of each map while retaining key information. Other purpose of pooling

layer is to reduce overfitting between convolutional layers. The common types of pooling layer in practice:

- Max Pooling: extract the max value of each subregion, described in Figure 6.

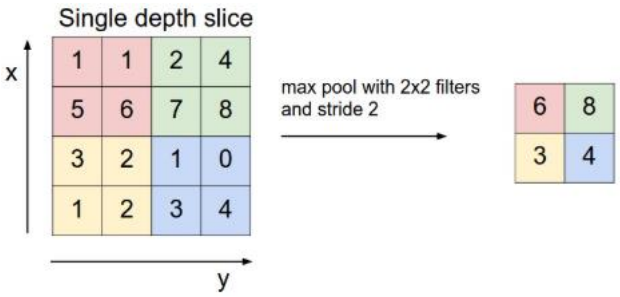

Figure 6: Max Pooling Example and Extraction Process [2]

- Average Pooling: calculate the average value of each subregion.
- Sum Pooling: taking the sum of the input values of subregion.

Figure 7 is a demonstration of pooling an input volume spatially from [224x224x64] to [112x112x64] by a [2x2] filter.

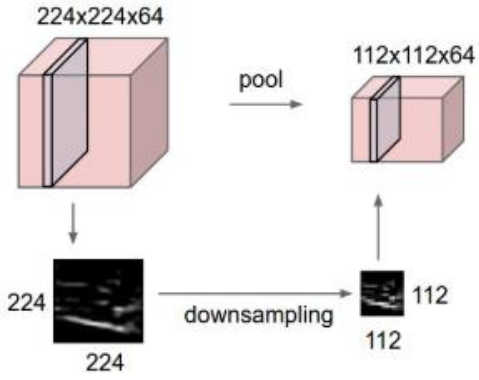

Figure 7: Pooling Example by Down Sampling

2.1.5 Fully Connected (FC) Layer

Before this stage, the feature has been learned, and the output matrix has been extracted. The matrix is then flattened, converted as a vector, and fed into a FC layer like a regular neural network. Finally, the activation function such as softmax or sigmoid performs and gives the outputs.

2.2 MobileNetV1

The main constrain when building CNN models for mobile and other edge devices is the limited compute resources. The first version of MobileNet [24] was introduced in 2017 to tackle this limitation. In MobileNetV1, depthwise separable convolution is used to reduce computation cost and model size. A depthwise separable convolution comprises of a depthwise convolution and a pointwise convolution. When kernel size is 3×3, computational cost is saved by 8 to 9 times, with just only small reduction in accuracy [24]. The MobileNetV1 architecture is presented in Table 1.

Table 1: MobileNetV1 Body Architecture [24]

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5\times$ Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
$5\times$ Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

In MobileNetV1, all layers are followed by a batchnorm [26] and ReLU nonlinearity, except the final fully connected layer. Figure 8 compares the applications of batchnorm and ReLU in regular convolutions and depthwise separable convolutions. In order to build more appropriate and efficient MobileNets, two hyper parameters for adjustment are width multiplier and resolution multiplier [24].

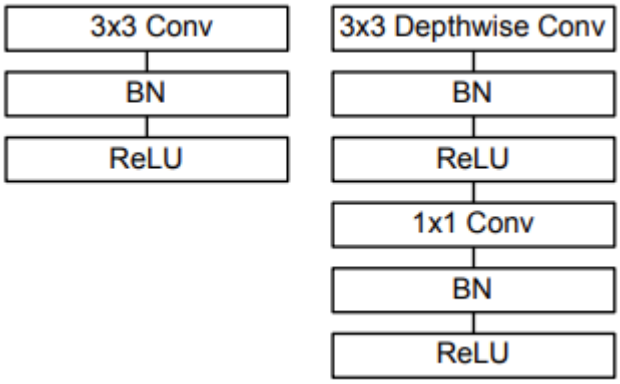

Figure 8: Batchnorm and ReLU applied after each convolutional layer in Standard (Left) and Depthwise Separable (Right)

Comparing to popular models during that same time, the number of multi-adds and parameters of MobileNet-224 version 1.0 is much fewer, while having relatively good performance, details in Table 2. In this comparison, GoogleNet [22] is a 22-layer deep CNN, a winner in classification task at the ILSVRC 2014 classification competition [27]. Similarly, VGGNet [21] is the first runner in classification task, also won the localization task at the same competition.

Table 2: MobileNet Comparison to Popular Models [24]

Model	ImageNet Accuracy	Million Mult-Adds	Million Parameters
1.0 MobileNet-224	70.6%	569	4.2
GoogleNet	69.8%	1550	6.8
VGG 16	71.5%	15300	138

Similar metrics recorded in Table 3 when it was put among smaller networks, Squeezenet [28] and AlexNet [20]. Those are winners of ILSVRC 2012 [27].

Table 3: Smaller MobileNet Comparison to Popular Models [24]

Model	ImageNet Accuracy	Million Mult-Adds	Million Parameters
0.50 MobileNet-160	60.2%	76	1.32
Squeezenet	57.5%	1700	1.25
AlexNet	57.2%	720	60

An extended comparison with Inception V3 [29] was carried out for fine grained recognition on Standard Dogs dataset [30]. Inception V3 is the first runner up for image classification in ILSVRC 2015 [27]. The results in Table 4 convince that MobileNet gained comparable accuracy scores, all above 80%, while having much smaller numbers of mult-adds and parameters.

Table 4: MobileNet for Stanford Dogs [24]

Model	Top-1 Accuracy	Million Mult-Adds	Million Parameters
Inception V3	84%	5000	23.2
1.0 MobileNet-224	83.3%	569	3.3
0.75 MobileNet-224	81.9%	325	1.9
1.0 MobileNet-192	81.9%	418	3.3
0.75 MobileNet-192	80.5%	239	1.9

The MobileNet were also applied on other datasets and use cases to prove its effectiveness counting object detection, face attributes and large-scale geolocation. In large scale geolocation, localisation tasks are transformed into classification of millions of images. PlaNet [31] is an adept at this task and it outperforms Im2GPS [24, 32, 33]. Moreover, when replacing Inception V3, original backend architecture of PlaNet, by MobileNet, competent results are reported with the same task, details in Table 5. Specifically, the PlaNet modified with MobileNet backend got lower score at Country and Region scales only, but better than original PlaNet at other scales. It is definitely better than Im2GPS. Impressively, the MobileNet model has just 13 million parameters and 0.58 million mult-adds compared to 52 million parameters and 5.74 billion mult-adds of the original one [24].

Table 5: Performance of PlaNet using MobileNet architecture [24]

Scale	Im2GPS	PlaNet	PlaNet MobileNet
Continent (2500 km)	51.9%	77.6%	79.3%
Country (750 km)	35.4%	64.0%	60.3%
Region (200 km)	32.1%	51.1%	45.2%
City (25 km)	21.9%	31.7%	31.7%
Street (1 km)	2.5%	11.0%	11.4%

In face attribute classification task, the author of MobileNet tried to compress a large classifier which comprises of 75 million parameters and 1600 mult-adds. The combination of distillation [34] and simplified MobileNet prove the ability to reserve comparable mean average precision (mAP) with just 1% mult-adds consumed as reported in Table 6.

Table 6: Face attribute classification using the MobileNet architecture with different hyper-parameter settings (width multiplier α and image resolution) [24]

Width Multiplier / Resolution	Mean AP	Million Mult-Adds	Million Parameters
1.0 MobileNet-224	88.7%	568	3.2
0.5 MobileNet-224	88.1%	149	0.8
0.25 MobileNet-224	87.2%	45	0.2
1.0 MobileNet-128	88.1%	185	3.2
0.5 MobileNet-128	87.7%	48	0.8
0.25 MobileNet-128	86.4%	15	0.2
Baseline	86.9%	1600	7.5

In object detection measurement, MobileNet, VGG, and Inception V2 [26] are deployed as base models of Faster-RCNN [35] and SSD [36] and are trained on COCO data [24]. Similarly, MobileNet gained comparable scores even running on much smaller model as detailed in Table 7.

Table 7: COCO object detection results comparison using different frameworks and network architectures [24]

Framework Resolution	Model	mAP	Billion Mult-Adds	Million Parameters
SSD 300	deeplab-VGG	21.1%	34.9	33.1
	Inception V2	22.0%	3.8	13.7
	MobileNet	19.3%	1.2	6.8
Faster-RCNN 300	VGG	22.9%	64.3	138.5
	Inception V2	15.4%	118.2	13.3
	MobileNet	16.4%	25.2	6.1
Faster-RCNN 600	VGG	25.7%	149.6	138.5
	Inception V2	21.9%	129.6	13.3
	Mobilenet	19.8%	30.5	6.1

In short, by applying depthwise separable convolution to significantly reduce the model complexity and size, MobileNetV1 make it possible for mobile or other low computational devices.

2.3 MobileNetV2

The improvements in MobileNetV2 are an inverted residual structure and removal of non-linearities in the narrow layer [37]. In terms of architecture, MobileNetV2 comprises of a fully connected layer with 32 filters and 19 residual bottleneck layers as listed in Table 8. The non-linearity ReLU6 and standard kernel size 3×3 are utilised.

Table 8: MobileNetV2 layers (n: repeated times; c: the same number of output channer; s: stride of the first layer of each sequence (line), others use stride 1; t: expansion factor) [37]

Input	Operator	t	c	n	s
$224^2 \times 3$	conv2d	-	32	1	2
$112^2 \times 32$	bottleneck	1	16	1	1
$112^2 \times 16$	bottleneck	6	24	2	2
$56^2 \times 24$	bottleneck	6	32	3	2
$28^2 \times 32$	bottleneck	6	64	4	2
$14^2 \times 64$	bottleneck	6	96	3	1
$14^2 \times 96$	bottleneck	6	160	3	2
$7^2 \times 160$	bottleneck	6	320	1	1
$7^2 \times 320$	conv2d 1x1	-	1280	1	1
$7^2 \times 1280$	avgpool 7x7	-	-	1	-
$1 \times 1 \times 1280$	conv2d 1x1	-	k	-	

In MobileNetV2, the basic building block is a bottleneck depth-separable convolution with residuals, detailed in Table 9, where bottleneck residual block transforms k to k' channels.

Table 9: Bottleneck residual block (s: stride; t: expansion factor) [37]

Input	Operator	Output
$h \times w \times k$	1x1 conv2d , ReLU6	$h \times w \times (tk)$
$h \times w \times tk$	3x3 dwise s=s, ReLU6	$\frac{h}{s} \times \frac{w}{s} \times (tk)$
$\frac{h}{s} \times \frac{w}{s} \times tk$	linear 1x1 conv2d	$\frac{h}{s} \times \frac{w}{s} \times k'$

There are two types of blocks, stride 1 and stride 2, as described in Figure 9. When both blocks are added a linear 1×1 conv2d, the depthwise and pointwise convolution layers still remain as the previous version of MobileNet. In bottleneck residual blocks, a direct shortcut is used between the bottlenecks [37]. The paper also reported that the bottleneck shortcut connection outperforms the shortcut connection of expanded layers as compared in Figure 10.

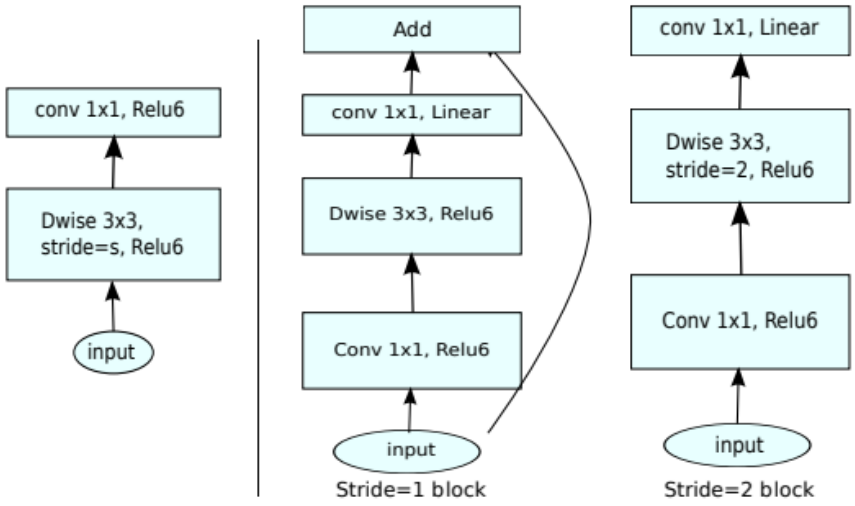

Figure 9: MobileNetV1 (Left) and MobieNetV2 (Right) [37]

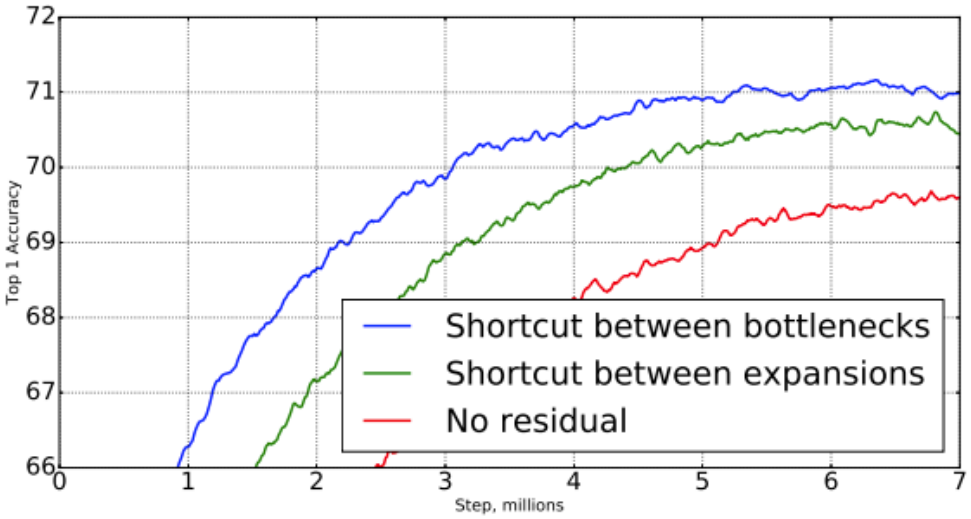

Figure 10: Impact of variations in residual blocks [37]

The author of MobileNetV2 also evaluated trade-offs between accuracy and performance through adjust hyper parameters, input image resolution and width multiplier. With the resolution 224×224 and width multiplier 1, the computational cost is 300 million mult-adds and 3.4 million parameters [37]. For input resolution ranged from 96 to 224 and width multiplier of 0.35 to 1.4, the computational cost varied from 7 to 585 million mult-adds, and 1.7 to 6.9 million parameters [37]. To improve performance for smaller models, the multiplier less than one is not applied to their last convolutional layer [37].

Similarly, the experiments of MobileNetV2 are setup and measured on ImageNet classification, object detection, and semantic segmentation tasks. The performance comparison of ImageNet classification is reported in Figure 11. In that experiment, the MobileNetV2 uses multiplier 0.35, 0.5, 0.75, 1.0 for all resolutions, for 224, additional multiplier 1.4 is applied.

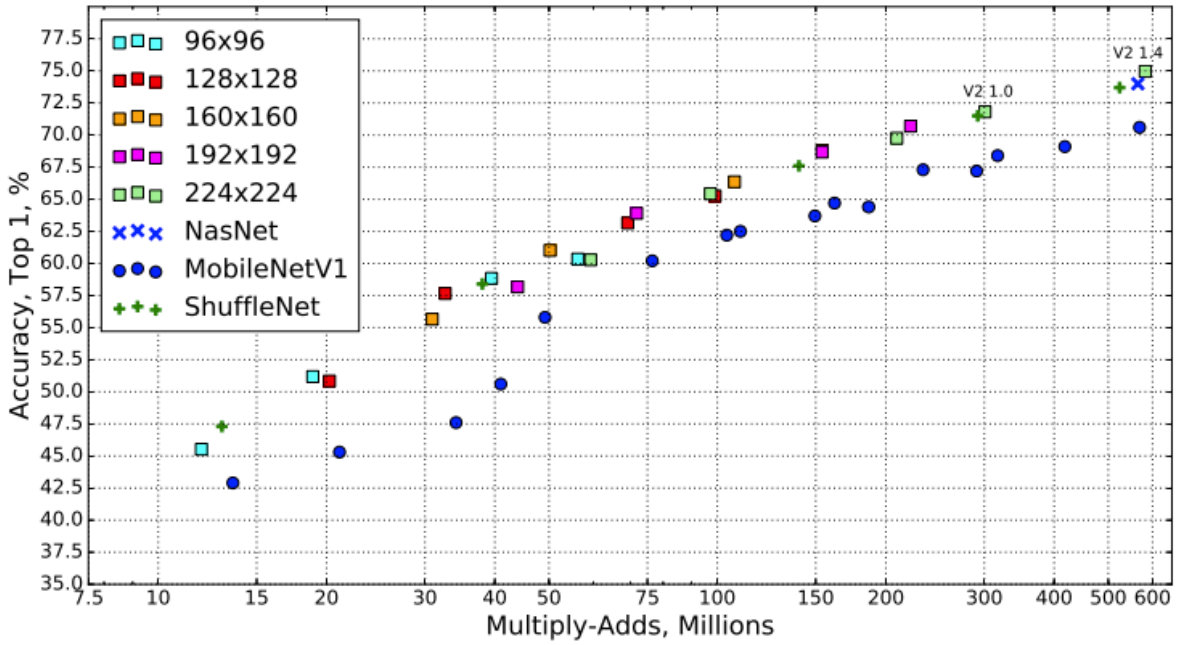

Figure 11: Performance curve of MobileNetV2 vs MobileNetV1, ShuffleNet, NAS [37]

2.4 Transfer Learning

In transfer learning, a fully pretrained model of a similar task is chosen to start with. The pretrained models could be selected from a range of published literature such as VGG, Inception, MobileNet.

Depending on the purpose and the situation, the pretrained model could be utilized in various ways:

- Freeze some layers and train others: in case of a small dataset and a large number of parameters, more layers should be frozen to avoid overfitting [38]. In contrast, for the case of large dataset and small parameters, more layers should be trained to improve the model without being prone to overfitting [38].

- Only train the final layer: the convolutional base is frozen, pretrained weights are extracted. This approach is useful for the case of small dataset and problems to be solved are similar [38]. It is the case of this practice context because there is only limited dataset acquired for training and small parameters.

For the first classification model of the research application pipeline, after trying various transfer learning approaches, the combination turned out to be the best by yielding highest accuracy among other. The pretrained model Mobilenetv2 [37] developed by Google team and trained on the ImageNet dataset [39] is selected for experiment. Comparing to the apple leaf disease classification, the solving problems could be considered as similar to some extent. Furthermore, the data set using in this research, a subset of Plant Village dataset [40], has just a few hundred samples for each class. Those give us the point of reference to start the first step of training on the final layer only. Subsequently, the number of parameters in millions suggest us to train some more layers to improve accuracy and avoid overfitting. In fact, we are also applying other fine-tuning techniques to control train process which will be explained in more detail under the model implementation section. Similarly, transfer learning is also applied on building of our second model which responsible for leaf segmentation. The second model training starts with weights which were trained on COCO dataset [41].

2.5 Optimisation Method for Training

Stochastic gradient descent (SGD) [42] is core function in utilisation of deep neural network. There are several popular SGD-based optimization methods developed such as RMSProp [43], Adagrad [44], Adadelta [45], AMSGrad [46] and Adam [47] and others. Both RMSProp and Adam is reported having good performance in some cases [48]. Specifically, Adam is more recent and popular and be chosen in training of the apple leaf disease classification model. It actually advances RMSProp by compute the learning rates based on two vectors so called the first and second moments [49] defined as below:

$$m_{t,i} = \beta_1 m_{t-1,i} + (1 - \beta_1) g_{t,i} \qquad [50]$$

$$v_{t,i} = \beta_2 v_{t-1,i} + (1 - \beta_2) g_{t,i}^2 \qquad [51]$$

As proposed in Adam algorithms, the initial settings are $\alpha \in [10^{-2}, 10^{-4}]$, $\beta_1 = 0.9$ and $\beta_2 = 0.999$ is good for many models [47].

2.6 Multi Class Cross Entropy Loss for Classification

The loss function is a key required component to monitor and evaluate training progress of deep learning models. After each iteration of training, the loss function computes error as the difference between ground truth labels and predictions based on the weights of current state of the model. In order to improve prediction accuracy, the calculated error needs to be minimized by modify weights and bias [52]. Weight modification is responsible by the optimisation function [52].

Different lost functions would return different error for the same predictions, thus extremely impact on the model performance. There are several loss functions designed to deal with various tasks such as regression, binary classification, and multi class classification. It is essential to choose an applicable loss function to train the desired model.

In this experiment, the classification model is multi class, and the default loss function is cross entropy, defined in the following formula:

$$\mathcal{L}(\theta) = \sum_{n=1}^{N} - \log(\sum_{i}^{c} p(\tilde{y} = \tilde{y}_n | y = i) p(y = i | x_n, \theta)) \qquad [53]$$

2.7 Data Augmentation

Deep convolution neural networks have excellently done many tasks in computer vision. In fact, it usually requires a large number of training data to reduce overfitting, improve model performance and generalise learning knowledge [54]. To deal with the major hindrance from limited dataset while working with images, we could consider several common augmentation methods such as vertical and horizontal flipping, vertical and horizontal flipping shifting, colour space transformation, random zooming, random cropping. Other advance augmentation techniques have also been utilised recently are adversarial training and GAN [54].

2.8 R-CNN Regions with CNN features

In dealing with object detection problem by using CNN models, the unknown number of objects in the image makes it hard to design an efficient output layer. To overcome the difficulty of selecting a huge number of regions, Girshick et al. [55] propose using selective search algorithm [56] to extract only 2000 possible regions then work with them instead of classifying a huge number of regions [57], depicted in Figure 12.

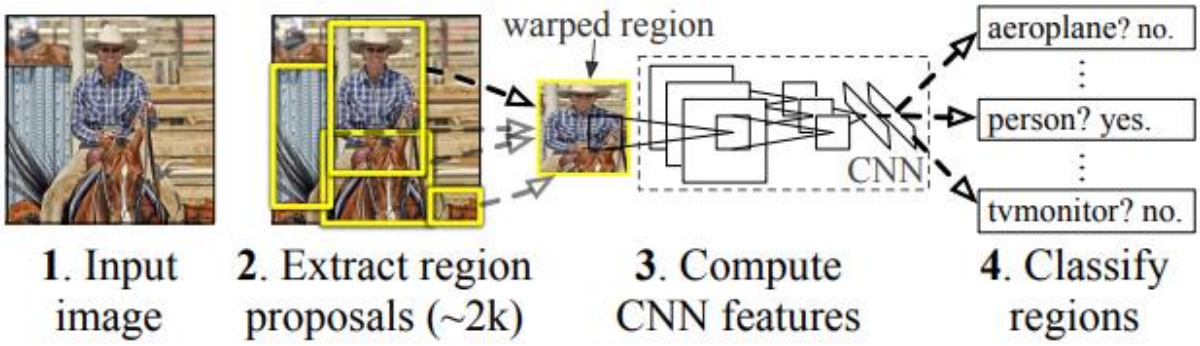

Figure 12: R-CNN Regions with CNN features [55]

However, R-CNN still has the problems of time consuming for classification of 2000 region proposals, taking 47 seconds for each test image and poor learning ability due to inflexible selective search algorithm [57]. After about one year and a half, the same author of R-CNN built Fast R-CNN [58] to deal with above problems of R-CNN and improve performance. Fast R-CNN still adopts selective search to extract region proposals, but not from the original image. The input image is fed into the CNN to create a convolutional feature map. Subsequently, regional proposals are selected from the convolutional feature map. The RoI pooling layer then reshapes these regions of interest (RoIs) before fully connected layers map them to a feature vector. Each vector (Roi) produces two outputs: softmax prediction and bounding box offset.

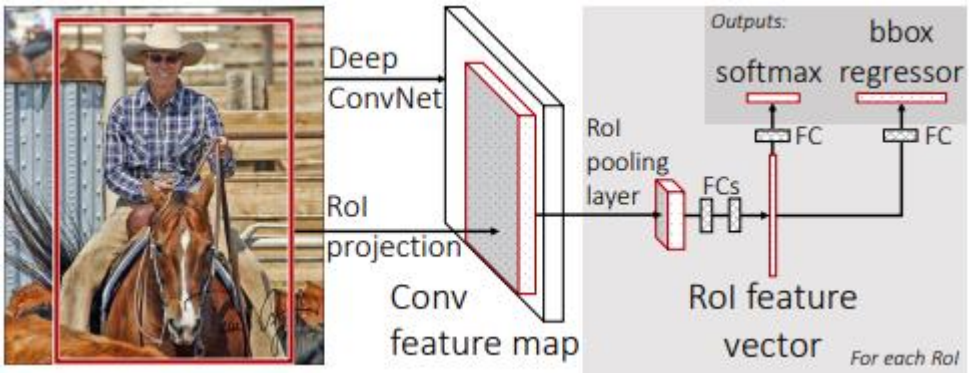

Figure 13: Fast R-CNN architecture [58]

Compared to R-CNN, depicted in Figure 14, the performance of Fast R-CNN is improved significantly since the CNN compute is performed on the feature map once per image instead of 2000 RoIs. Moreover, when excluding regional proposal task catering by the selective search we can see a big difference. That is the area for Faster R-CNN [35] jumping in and replace selective search by a separate network.

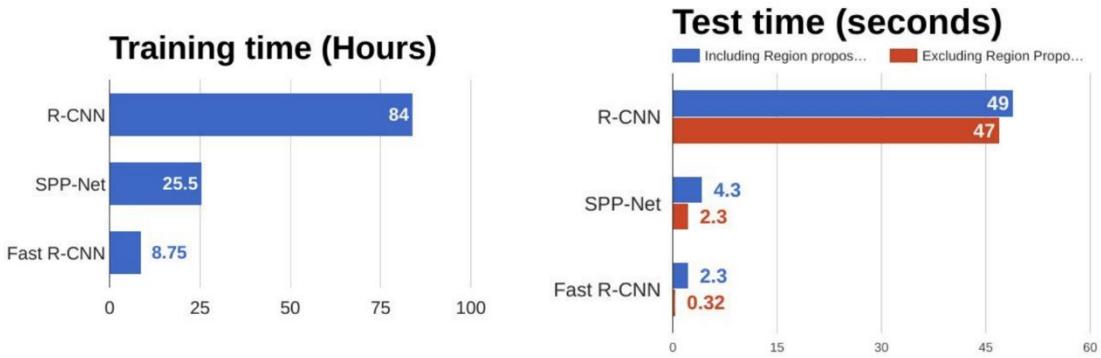

Figure 14: Performance of object detection algorithms [57] in Train and Test

Like Fast R-CNN, the convolutional feature map is still used to feed to regional proposals in Faster R-CNN [35]. But on top that, selective search is replaced by a region proposal network which is claimed to be nearly cost-free for detection network computation. The predicted region proposals of any size are reshaped using a RoI pooling layer before feeding to a classifier and predicting the offset values of the bounding boxes [57]. The process of Fast R-CNN is represented in Figure 15.

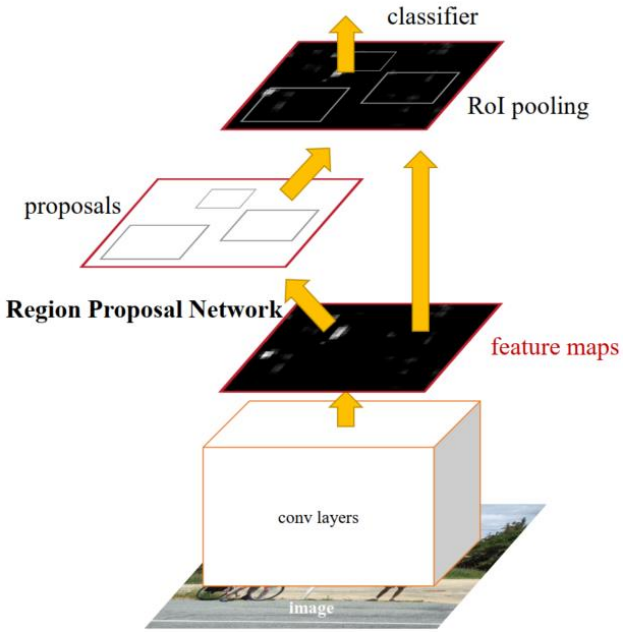

Figure 15: Faster R-CNN [35] using Feature Maps and Region Proposals

2.8.1 RPN

A Regional Proposal Network (RPN) takes in input image and outputs several rectangular proposals with objectness score [35].While generating region proposals, a small network is slid over the input convolution feature map and grab $n \times n$ spatial windows as inputs. Each sliding window is projected to lower-dimensional feature (256-d/512-d), then fed into a pair of fully connected layers for box regression (reg) and box classification (cls) as in Figure 16.

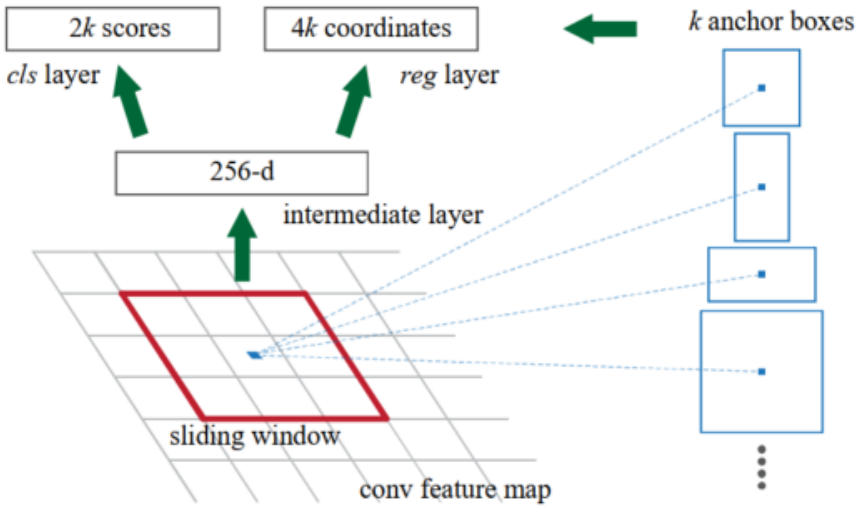

Figure 16: RPN in Faster R-CNN [35] using Sliding Windows to generate Region Proposals

2.8.2 Anchors

At each sliding step, maximum k boxes are taken as region proposals for simultaneous predictions [35]. The reg layer and cls layer return $4k$ encoded coordinates of k boxes and $2k$ probability scores of objectness respectively. These reference boxes are parameterised in relation to region proposals and called anchors [35]. An anchor uses the same centre with the sliding window and associated with a scale and aspect ratio. In Faster R-CNN, 3 scales and 3 ratios are set, producing $k = 9$ anchors. A convolutional feature map with size $W \times H$ has $W \times H$ sliding windows, yields $W \times H \times k$ anchors in total.

2.8.3 Intersection over Union (IoU)

IoU is a popular method to evaluate object detector benchmarks. It is calculated based on ground-truth bounding boxes and predicted bounding boxes as represented in Figure 17.

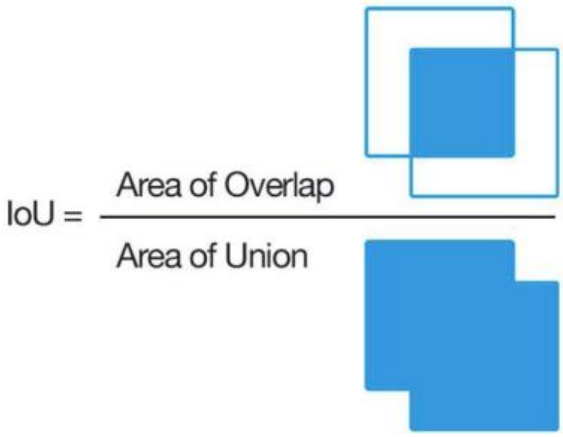

Figure 17: IoU Formula [2]

IoU encodes the shape properties of comparing objects, make it invariant to the scale of the problem under consideration [59]. By that, it is appealing and being used widely in segmentation [41, 60-62], object detection [41, 63], and tracking tasks [64, 65] [59].

2.9 Instance Segmentation

Instance segmentation is a typical task of Computer Vision to identify multiple instances of each object with the image at the pixel level. In other words, the task of image segmentation is to locate the position of objects in the image, shape them by predicting to which object each pixel belongs. Instance segmentation is considered as the combination of semantic segmentation and object detection as illustrated by Ladický et al. [66]. Instance segmentation depends on the ability of both object detection (to separate between instance) and semantic segmentation (to produce pixel-wise predictions) to delineate the shape of the objects [67]. There are several common applications of instance segmentation such as medical imaging, self-driving cars, satellite imaging to name a few [68]. In our experimentation, we manage to count the number of the same object, segment and locate them within the image.

2.10 Mask R-CNN

Facebook AI Research (FAIR) built Detectron, a high performance codebase for object detection algorithms including Mask R-CNN [69, 70]. It is modification of Faster R-CNN [35] by including an extra branch for predicting object mask alongside the existing branch for bounding box recognition [70] (Figure 18). The mask branch predict a segmentation

mask at the pixel level by using Fully Convolutional Network (FCN) [71] applied to each Regional of Interest (RoI). Authors claimed that the additional branch only consumes a light computational overhead, about 20% on typical model, running at 5 fps on the top 100 detection boxes only [70]. Overall, Mask R-CNN is easy to implement and train, flexible design, enabling streamlined experimentation.

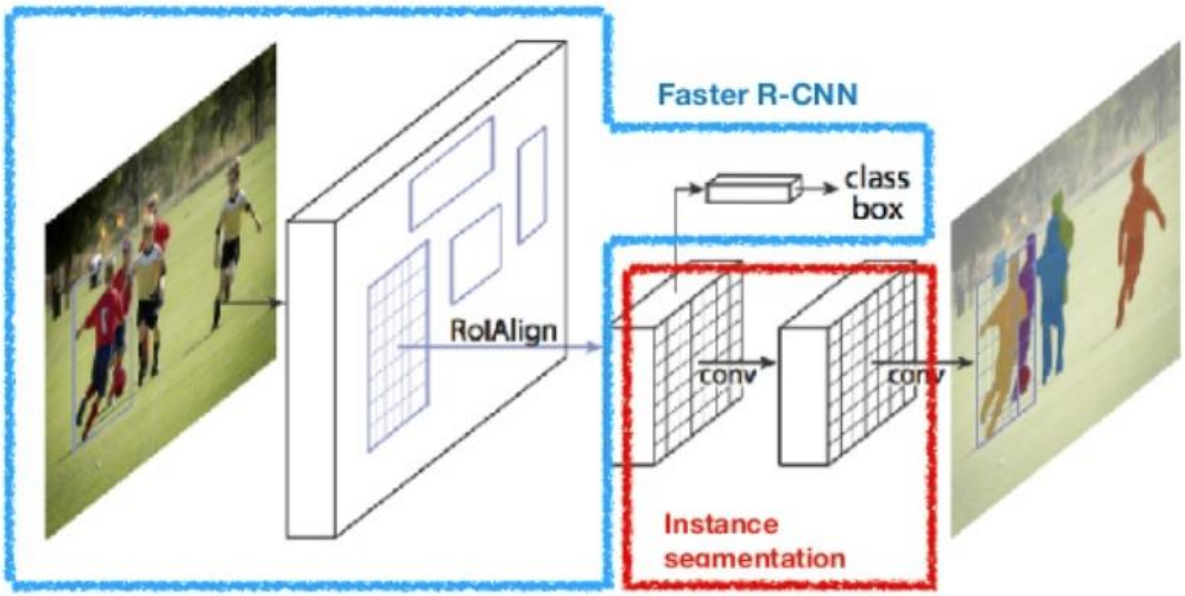

Figure 18: The Mask R-CNN framework: Faster R-CNN + Instance segmentation [72]

Similar to Faster R-CNN, Mask R-CNN procedure comprises of two stages [70]. In the first stage, input image is scanned to propose regions which possibly contain an object. In the second stage, proposed regions are classifier, bounding boxes and masks are generated concurrently [70].

In the network architecture of Mask R-CNN, Figure 19, several modules are mentioned. Compared to Faster R-CNN, mask branch component is added.

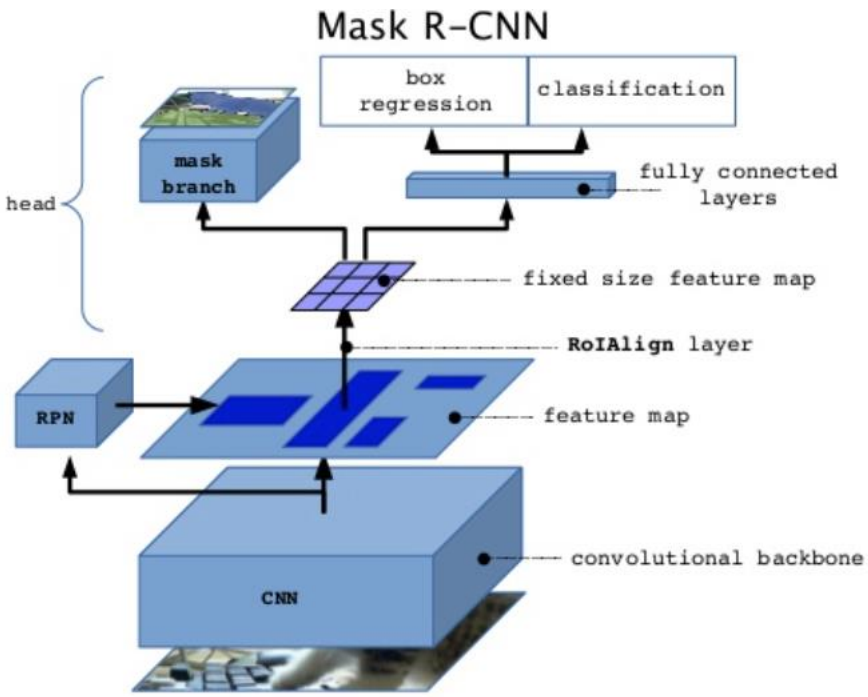

Figure 19: Mask R-CNN Architecture [72]

One main module is the convolutional backbone architecture which is used as a feature extractor. The convolutional backbone architecture could be ResNet [23] or ResNeXt [73]. In our leaf segmentation model, Mask R-CNN framework is used with the backbone of ResNet network of 101 layers. Additionally, the Mask R-CNN paper reported that the combination of ResNet and FPN [74] yield excellent results for both accuracy and speed. FPN improves feature extraction by adding a second pyramid, adopting top-down and skipping connections [74]. In FPN, single high level feature map from the first pyramid is passed to the second one where predictions could be made independently on each level as shown in Figure 20.

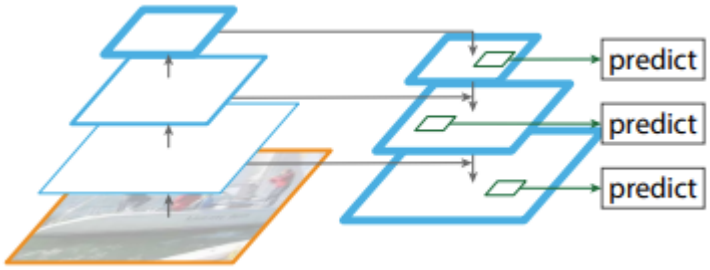

Figure 20: FPN architecture [74] with additional pyramid adopting top-down and skipping connection

In Faster R-CNN [35], the standard operation RoIPool extracts a feature map for each RoI. RoIPool has the issue of misalignment between RoI and extracted feature because of quantization on RoI. RoIAlign, simple and quantization-free, is introduced to address the misaligned issue by preserving spatial locations [70]. It replaces quantization linear interpolation [75]. It claims to improve mask accuracy by 10% to 50% even it is just a minor change. Specifically, it gains 3 point of AP over RoIPool with improvement at high IoU (AP_{75}) [70].

The other module, network head, is responsible for bounding-box recognition and mask prediction on each RoI. In Mask R-CNN, the box heads are extended from the previous work Faster R-CNN by adding a mask branch to ResNet, ResNeXt and FPN, depicted in Figure 21. Both are having two heads. However, in the case of FPN, the network is splitting directly without running any common convolutions. Therefore, more convolutions are performed on the mask head of FPN.

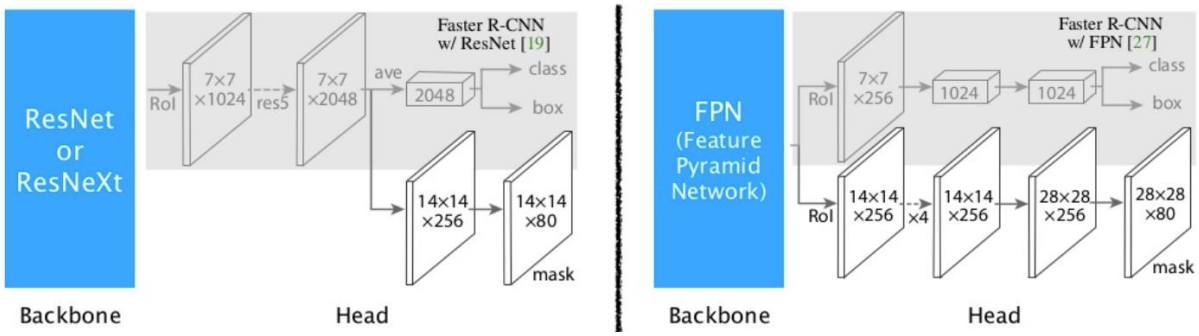

Figure 21: Head Architecture [72] in cases of ResNet/ResNeXt (Left) and FPN (Right)

Mask R-CNN was trained on COCO dataset [41] and measured performance following the standard COCO metric, in order to compare with MNC [76] and FCIS [77], the winners of the COCO 2015 and 2016 segmentation challenges respectively [70]. Based on the report in Figure 22, Mask R-CNN surpassed them at various scale.

	backbone	AP	AP_{50}	AP_{75}	AP_S	AP_M	AP_L
MNC	ResNet-101-C4	24.6	44.3	24.8	4.7	25.9	43.6
FCIS +OHEM	ResNet-101-C5-dilated	29.2	49.5	-	7.1	31.3	50.0
FCIS+++ +OHEM	ResNet-101-C5-dilated	33.6	54.5	-	-	-	-
Mask R-CNN	ResNet-101-C4	33.1	54.9	34.8	12.1	35.6	51.1
Mask R-CNN	ResNet-101-FPN	35.7	58.0	37.8	15.5	38.1	52.4
Mask R-CNN	ResNeXt-101-FPN	**37.1**	**60.0**	**39.4**	**16.9**	**39.9**	**53.5**

Figure 22: Instance segmentation mask AP on COCO test-dev [70]

The visual output comparison also means Mask R-CNN archive good results, having no systematic artifacts on overlapping objects.

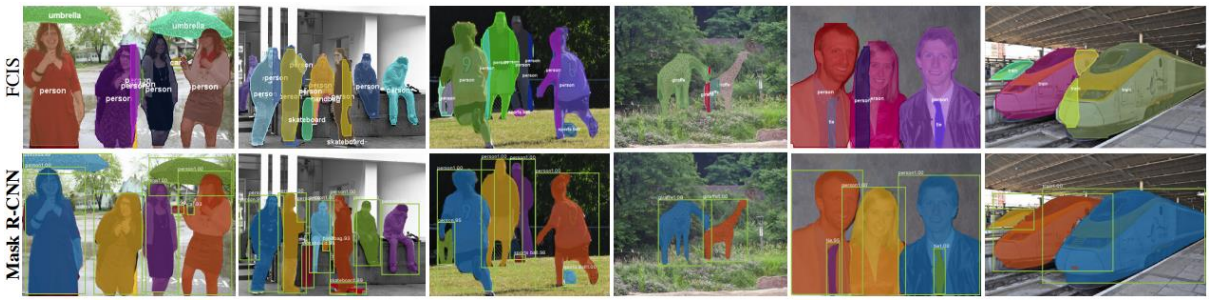

Figure 23: FCIS+++ (top) vs. Mask R-CNN (bottom, ResNet-101-FPN) [70]

In terms of inference timing, Mask R-CNN with ResNet-101-C4 backbone consumes more time and is not recommended to use in practice [70]. In the paper, the Mask R-CNN author claimed it is fast, but not specifically designed for speed.

2.11 Training ML Models on The Could

In general, edge devices are often built with comparable camera but light compute resources. That is reason some activities in the machine learning application life cycle that do not require a strong compute unit, could be performed on the edge device such as image or video capturing and labelling. Model training is an intensive task and usually be run on backend servers or cloud services with strong compute combined with GPU or TPU resources. Cloud approach seems to be more convenient and flexible. It enables us to promptly initiate necessary resources for training whenever needed with much lower cost compared to preparing on-premises servers. The cloud-based resources are not only expandable but also could be released once the job is done.

The typical machine learning services are Amazon ML, Azure ML, Google Cloud AI, and IMB Watson, detail comparison in Figure 24. In Federated [78] or Privacy-Preserving [79] training, the individual data could be avoided sending to the cloud, but the client needs to upload updated weights to the server for training model.

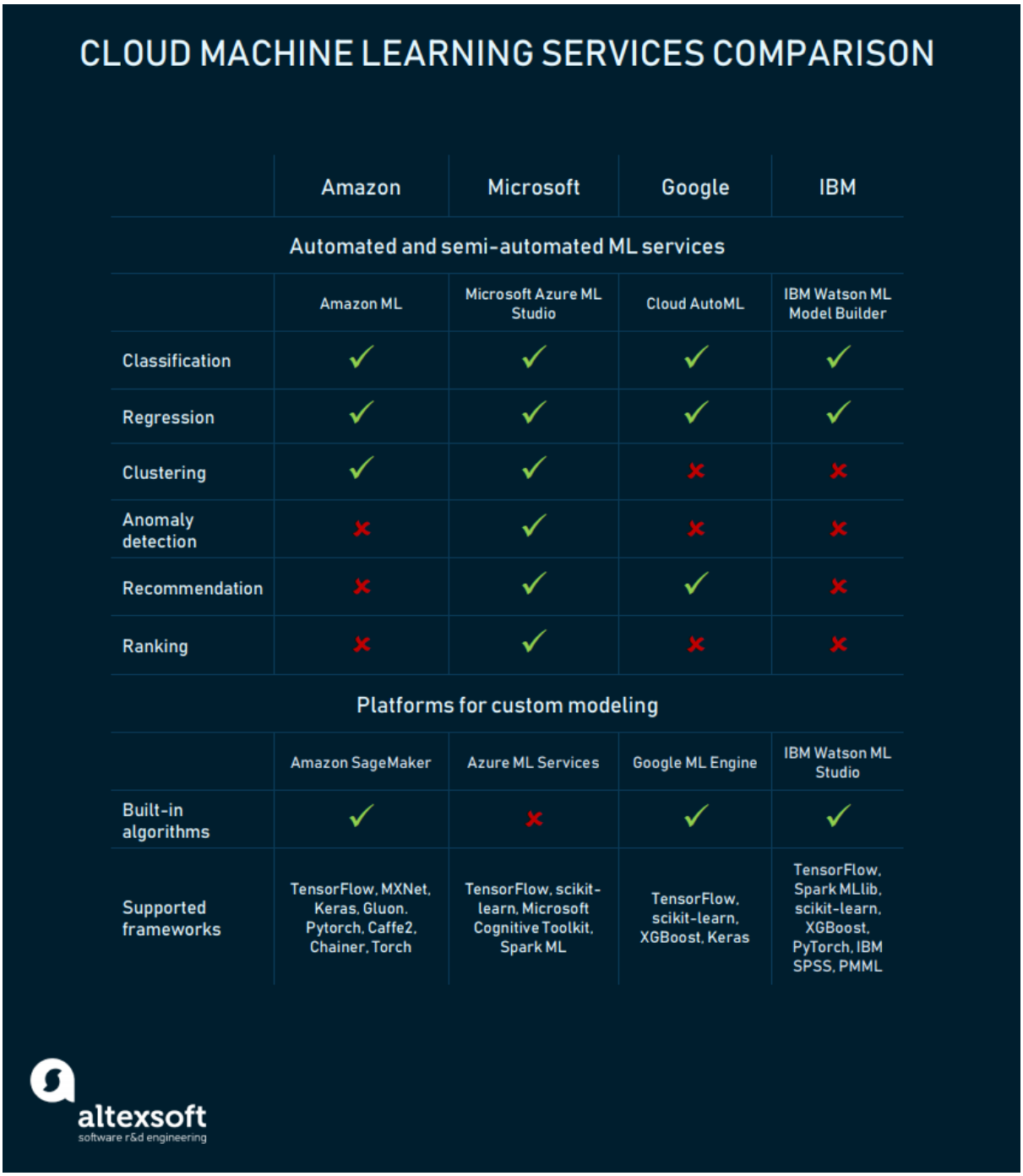

Figure 24: ML services comparison [80]

In this experiment, we mainly use the utilities of the TensorFlow ecosystem and other related cloud services provided by Google.

2.12 Running Deep Learning Applications on Mobile

Mobiles are usually designed with optimised computing resources which are not ideal for conducting the whole life cycle of machine learning application. There are several studies recently looked into this aspect and proposed approaches to operate DL models on mobile devices.

Wang et al. [81] studied challenges and achievements in running DL on mobile devices. They stated that it is impossible to train DL models on the mobile devices due to resource constraint. Training still be taken on the cloud or robust computers. For the inference task, the research [81] also mentioned two choices: on the cloud or on the local mobile device. Even it is possible to run full models on mobile devices and inference locally, the inference task of full DNN models would consume considerable amount of power and local resources [81]. Thus, to ease running trained DNN models on mobile devices, especially to handle complex inference tasks, the models should be compressed and optimised [81]. In fact, there are several development suites helping to bring ML models to mobile devices such as Core ML [82], Create ML [83], PyTorch Mobile [84], Firebase ML [85], TensorFlow Lite [86].

2.13 Model Optimization for Edge Devices

Edge devices usually have limited storage, memory, and computational resources. To deploy and serve trained models on edge devices, practitioners have to deal with these tight constraints. Using TensorFlow suite, it is easy to make good use of TensorFlow Lite [86] and Model Optimization Toolkit [87] to enable models working within these major constraints.

At this writing, TensorFlow Lite supports three optimization types which are quantization, pruning, and clustering [88]. The default option of TensorFlow Lite Converter is quantization. For the purposes of optimization, quantization is the most common and comprehensive solution. It is implemented by reducing the precision of a model's parameters. This technique results in helping decline model size and increase computational speed without losing much accuracy. These other two options are solely improving model compression effectiveness. As a result, they benefit model deployment by smaller model download size. In pruning solution, parameters of a model which have light impact on predictions are trimmed. Similarly, to reduce the number of unique

parameters and simplify the model, clustering groups weights of each layer into predefined clusters with sharing centroid values.

2.14 TensorFlow Serving [89]

Once the training is completed, there is a need to deploy the ideal trained model and publish it for consumers, other people, or external services, to use. While many practitioners find deploying machine learning model cumbersome, TensorFlow serving emerges as an ease for this task. It has been used within Google for many years. TensorFlow Serving is a popular serving system for production uses due to its high performance and convenience such as handling requests or utilising GPU effectively for multiple running models. The same server architecture and APIs are conserved by just simple deployments for new algorithms and experiments [89]. When the request increases, puts the system under pressure of high workload, it could leverage on the Kubernetes architecture to orchestrate and scale up as needed. Furthermore, as part of TensorFlow extended ecosystem, it provides out-of-the-box integration with TensorFlow models, as well as extends easily to serve other type of models and data.

TensorFlow Serving architecture comprises of the following key concepts: Servables, Loaders, Sources, Managers and Core.

2.14.1 Servables

The core objects of TensorFlow serving are Servables [89]. It is invoked by clients to perform desired tasks such as lookup or inference.

Size and scale of Servables could be varied, depending on requirements. It may be as simple as an additional function or as complex as a deep learning inference model. Servables could contains any type or interface, allowing flexibility and upgrades such as streaming results, experimental APIs, asynchronous modes of operation. The lifecycle of Servables is not handled by itself, more details are discussed in the next sections. Typical Servables consist of two common components: a TensorFlow SavedModelBundle and a lookup table for embedding or dictionary lookup.

TensorFlow Serving is capable of managing multiple versions of a servable with only one single server instance, as well as loading them concurrently for serving different

version request from clients. Servable versions are sorted sequentially by version numbers and stored in a servable stream.

A model in TensorFlow Serving is represented as one or more servables. Similarly, a composite model could comprise of either multi-independent servables or single composite servable. Likewise, a servable may be part of a model.

2.14.2 Loaders

Loaders manage life cycle of a servable by using standard APIs to load or unload a servable. This help isolates common infrastructure from various algorithms, data, or product use cases.

2.14.3 Sources

Sources are plugin modules which get and supply servables from file system to Loaders. Each Loader instance is created separately by a Source for each version of servable.

2.14.4 Aspired Versions

Aspired versions are referred to as a set of servable versions when a Source communicate and handle it to the Manager.

2.14.5 Manager

Managers manage the full life cycle of Servables, as well as loading, serving, and unloading them.

2.14.6 Core

Overall, TensorFlow Serving Core manage life cycle and metrics of servables by its standard Serving APIs, detailed in Figure 25.

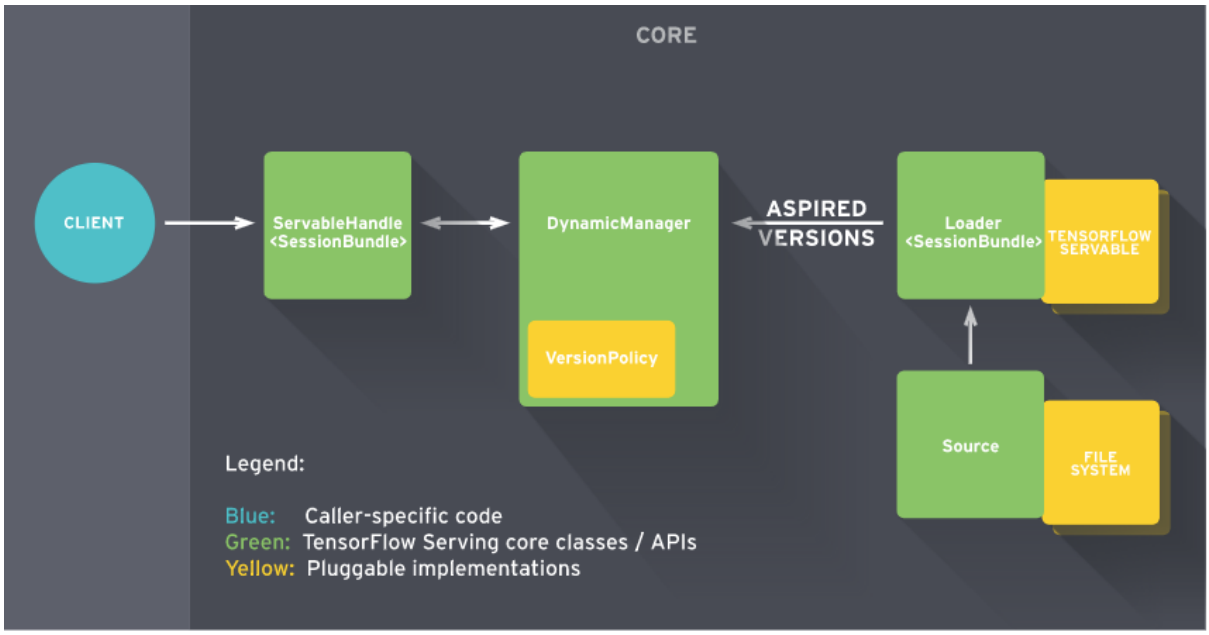

Figure 25: Life of a Servable

2.15 DL Application Approach for Mobile

The implementation of a DL system for mobiles usually involves these steps: prepare training data, build the model, train and finetune the model, deploy the model to cloud, hosted servers or mobile itself, invoke the model in the mobile application to solve the target problems.

CNN models are really effective for various computer vision tasks. However, due to their complex architecture, CNN models need to be trained on a large amount of dataset and take a long duration to complete training. Hence, it is not wise to train CNN models locally on mobile device. Even inference by CNN also requires heavy computation. Recently, some platforms have exposed to undertake or alleviate these burdens.

Table 10: Platforms and supported mobile OS [90]

TensorFlow	iOS, Android
Caffe2	iOS, Android
CoreML	iOS
Snapdragon	Android
DeepLearningKit	iOS

There are different approach solutions to build DL models for mobile applications:

- Leveraging on Cloud AI services: all procedures including training ML models, inference is supported by cloud services. Some well-known service providers are Microsoft Azure Cognitive Service, Google Cloud Vision, IBM Watson Services, Amazon Rekognition, Baidu EZDL [90]. This list is increasing quickly since more providers are coming into this field.

- Supporting edge devices: at least the inference step could be implemented on edge devices without relying on cloud services. Some popular platforms supporting implementation of DL for mobile OS are listed in Table 10.

Our experiment context is the application of DL in the field setting where usually has no internet connection. Hence the application is expected to be able to operate offline on mobile devices.

TensorFlow Lite [4] is an easy option to execute TensorFlow models on mobile, inferences with low latency and small binary size. Especially, it does not require internet connection, reduce power consumption [4], convenient for operation on the field where internet and power sources are usually unavailable.

A TensorFlow Lite must be converted from a full TensorFlow model, but not all models are compatible and supported. This work is catered by TensorFlow Lite converter. A TensorFlow Lite FlatBuffer file (.tflite) is generated and deployed to client devices and run locally as in Figure 26. The steps involved to deploy a TensorFlow Lite model are: Pick a model, convert the model, deploy to your device, optimise your model. TensorFlow Lite currently is not trainable, but it is stated under its roadmap plan at the time of writing [4].

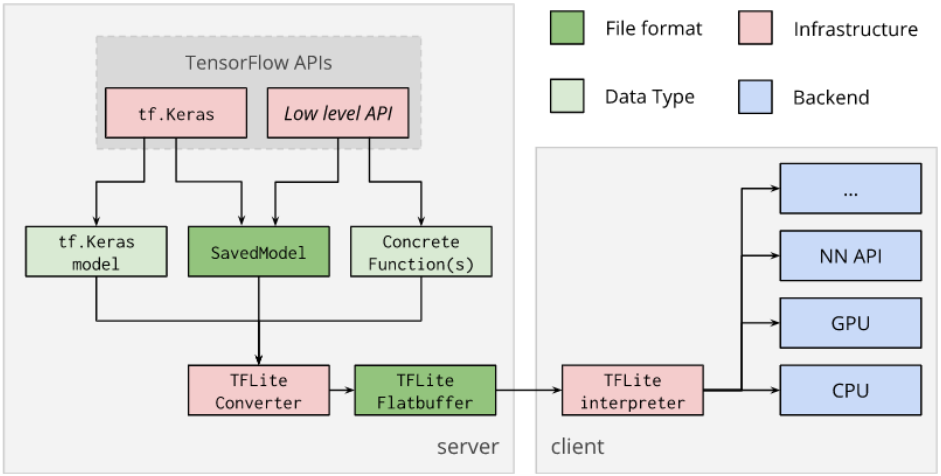

Figure 26: TensorFlow Lite Conversion Process [4]

Once the model is optimised, it could be used in the native mobiles by using:

- Core ML for iOS apps.
- TensorFlow Lite Android Support Library for Android apps.

Some common contemporary platforms support building ML mobile apps are: Custom Vision (Microsoft), AutoML, Flutter, ML Kit. The Flutter framework [91] supports building cross platform application including iOS and Android from a single codebase.

2.16 CNN in Plant Leaf Disease Detection

In the Agriculture realm, contemporary applications of ML could be categorised in four management groups: crop, livestock, water, soil [92]. Plant disease detection tasks belong to the crop management group.

Recently, there are some studies about applications of DL on plant disease detection, focusing on figuring out a better approach to the classification puzzle, maximizing the accuracy of the architecture at the same time. Those experiments proved that applying DL CNN for the apple leaf disease detection is possible. Several existing related types of research are listed as follows, as well as their technique details, dataset information and final results.

Baranwal, Khandelwal, and Arora [93] employed DL CNN to classify and analyse apple leaf diseases. The model proposed in this research is based on the GoogLeNet architecture, which consists of 22 hidden layers with 5 million parameters. The dataset utilized in this research is a subset of Plant Village dataset, which comprises 1000 healthy leaf images and 1526 diseased leaf images. The highest accuracy achieved by the proposed architecture is 98.42% on overall samples with the dropout is 0.2.

Mohanty, Hughes, and Salathé in their research 'Using Deep Learning for Image-Based Plant Disease Detection' [94], trained a deep convolutional neural network, which could achieve 99.35% accuracy, to recognise 14 kinds of crops and 26 diseases. This research utilises 54,306 healthy and diseased plant leaves images provided by the PlantVillage project [40] to feed the proposed model. By comparing and evaluating two popular CNN models AlexNet and GoogLeNet, it comes out that GoogLeNet persistently performs better than AlexNet among different training-testing ratios. They suggested that implementation of CNN on mobiles to analyse crop disease is possible.

In the paper 'Deep neural networks based recognition of plant diseases by leaf image classification' [95], Sladojevic et al. proposed a model to implement thirteen different genera of plant disease classification by using CaffeNet [96], which is a type of deep Convolutional Neural Network with five convolutional layers followed by fully connected layers. The best accuracy of their model is 96.3% after fine-tuning. This experiment proves that other kind of CNN architecture is also applicable to diagnose plant leaf diseases.

Another application of DL is the approach with a new proposed model INAR-SSD for real time detection of apple leaf diseases [97]. The application performance reaches 78.80% mean average precision while detecting five types of diseases.

Applying computer vision in plant leaf detection could save a huge number of manual efforts. The common challenges and difficulties when applying CV in plant and fruit quality control could be known as illuminance, background noise, model scalability, and time-consuming constraint on accuracy.

- Model Scalability: coming to ML, knowledge is data. Often having inadequate data for training, the detection system may yield inaccurate and deviant results in orchard settings. More extensive training data collected under various condition could improve accuracy when applying in real condition.

- Time-Consuming: The training of the model usually takes quite long. The significant amount of time for computation and inference also needs to be reduced under acceptable manner.

Chapter 3

METHODOLOGY

This chapter explain how this research is carried out and followed a research method. It outlines the research flow as well as objectives and detail activities in each step.

According to surveys conducted in 2.16 and discussions among ML skilled personnel in New Zealand, applying of DL into agriculture especially apple growing is common these days. However, there is a lack of an efficient portable system helping growers dealing with apple leaf diseases within their orchard condition. The ideation ended up targeting portable apple leaf disease detection system backed by ML.

There are indeed various frameworks to carry out research. However, the aim of this research is creating a portable application using DL to diagnose apple leaf diseases, a practical problem. Hence, the design science methodology [3] guidelines are appropriate for following systematically to create the artifact which is the apple leaf disease detection application and system. The context consists of IT infrastructure, cloud services, orchards, farms, growers, apple leaf diseases and others. The artifact itself would not be enough to resolve the problem. It requires the interaction of both artifact and context [98].

This research was conducted and inspired by the design science research methodology (DSRM) [3]. The DSRM process [3] lists four possible entry points for conducting researches and demonstrate their good effect by four different project cases. For this research, the motivation with background of Design and Development leads to the respective trigger activity, forming a Design and Development-Centred Approach, depicted in Figure 27. The artifact to be developed and built is the mobile application to detect diseases using DL.

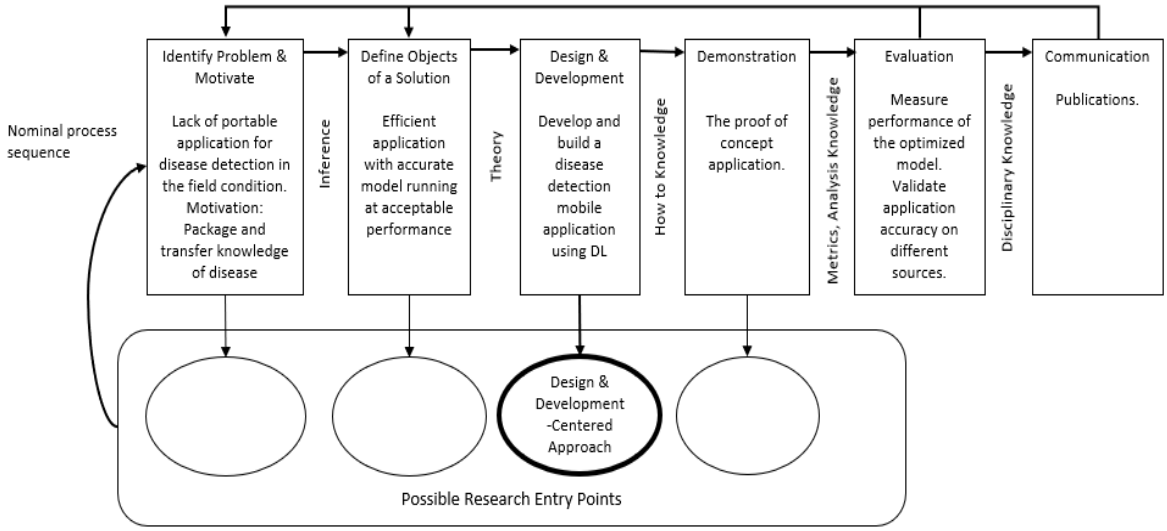

Figure 27: Design Science Research Methodology Process for Apple Leaf Disease Detection Project [3] with Design and Development Approach

At the first stage, the practical problem is expected to be resolved is detecting apple leaf diseases which can operate in the field. The main requirement to be addressed is packaging pathologists' insight into the application and making it feasible support growers in orchards without requiring interaction of pathologists and IT personnel every time. This high-level requirement introduces next stage with broken down objectives.

At the second stage, main objectives are an accurate optimised model and efficient application which can be distributed to multiple platforms, Android, and iOS, for example. The solution was designed by surveys conducted in Chapter 2 and advice from the researcher's supervisor and ML practitioner colleagues.

After the development stage, a prototype application is presented, demonstrated, and validated against datasets from both lab and other sources. The research is planned to be communicated via both scholarly research publications and feedbacks of users in application store.

The core model of apple leaf disease detection system is CNN. Training a CNN model from the scratch requires a huge amount of data and a robust platform to train the model in a long run. This heavy burden is inflicted on the small project with the limited dataset. Transfer learning is a good solution to cope with this situation. On top of the transfer learning, other optimisation and augmentation methods were applied in training. Instance segmentation technique is also be involved in the effort of eliminating background noise and extracting single objects for easier processing.

Chapter 4

DESIGN AND DEVELOPMENT

This chapter proposes the system and functional designs and development process to build up a system covering research objectives. It also describes implementation of ML models including preparing data, setting up models and other training settings, model evaluation and optimisation.

4.1 System Design

Regarding the leaf diseases, specimens labelled for research and analysing often be instances of individual leaves. There also be several open datasets of individual leaf labelled by their disease. However, in reality, growers will need to diagnose diseases directly from apple trees without cutting or capturing every single leaf. That could damage their plants, or it must be time-consuming and rather troublesome work carried out. To reduce the effort of capturing every single leaf and avoid impact to plants, the system is designed to capture the images of foliage directly from apple trees for disease analysis. The core components of the system would be:

- Portable application: it is cross-platform application installed on edge devices where system functions are built in and exposed for end users. With the client application installed, edge devices are able to capture foliage images on the orchard, perform pre-processing on images then send them to backend for analysing. Another function of client app is reporting recorded data such as disease by areas or areas by disease affected level. In the cases of no internet connectivity, application is pre-configured to capture single leaf image and analyse on spot. In this case, the optimised model is deployed to edge devices and no comprehensive report function provided.

- Leaf segmentation model: to extract individual leaves from the foliage image to pass to disease detection model for analysing. This model is deployed on cloud backend only.

- Leaf disease detection model: to identify any disease of individual or segmented leaves. This model is deployed and serving on both cloud and local edge devices.

4.1.1 High Level Architecture

Once the models are trained, they need to be deployed, published, and enable working in practise.

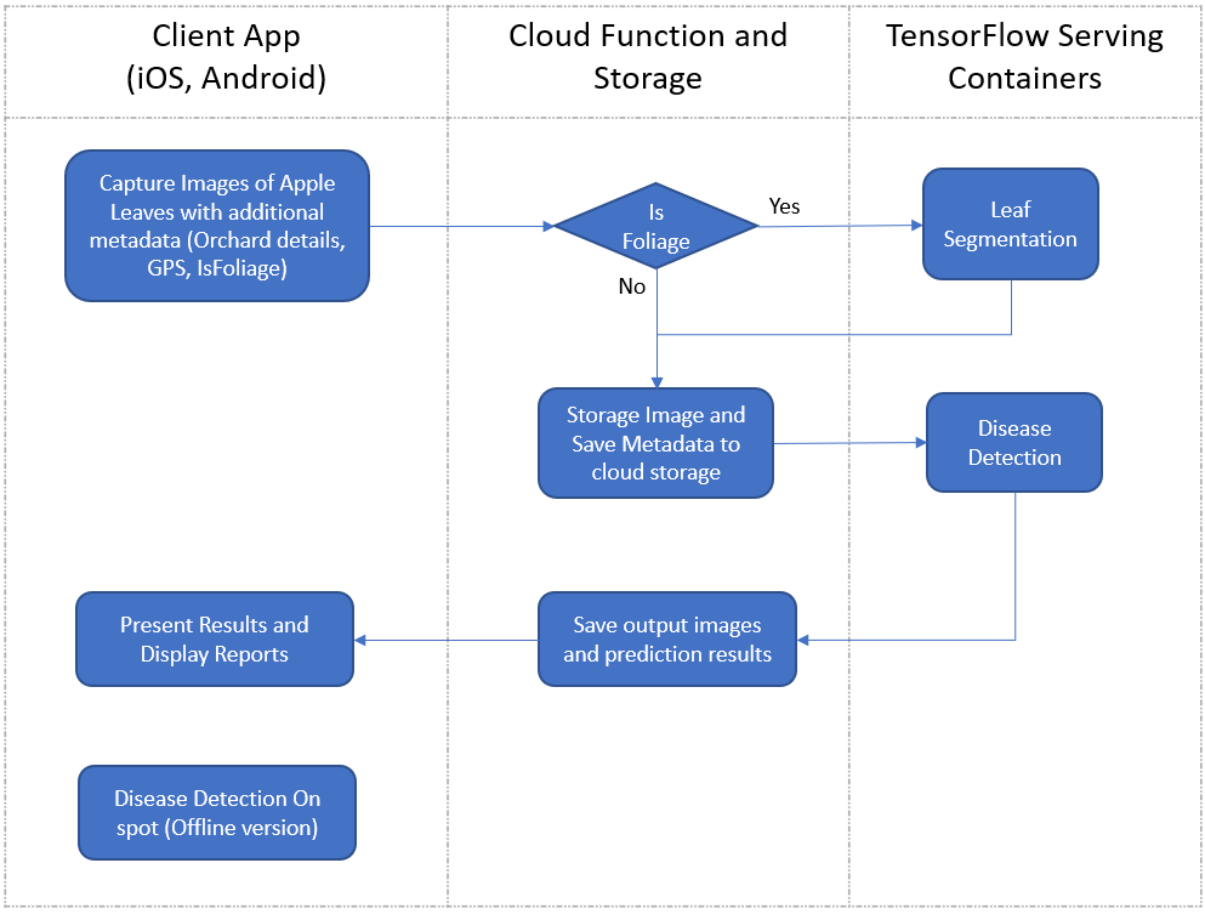

Figure 28: System Process consists of three layers: Client App, Cloud Function/Storage, and TensorFlow Serving Container

The whole system, as depicted in Figure 28, comprises of two parts, frontend, and backend. The main system functionalities are analysing apple leaves images and disease detection. These are responsible by the two core models, leaf segmentation and disease detection, which are deployed on two separate TensorFlow Serving containers. One point in design to reduce workload, improve system performance is the check of foliage. Since leaf segmentation process is time consuming and intensive computation, the single leaf image cases should avoid going through segmentation model.

4.1.2 Frontend

Frontend includes client app and serverless cloud function. Client app is developed by cross platform framework flutter, communicating with backend via cloud functions. Serverless cloud function could be implemented by using either Azure function, AWS lambda function, or any other similar solution.

Client apps could be installed on iOS or Android devices such as mobile phones, tablets, or drones. After installing app, client devices are able to pre-process captured images, tag them with additional details called metadata such as GPS, Orchard details, timestamp before sending to backend. Captured images are stored at a cloud blob storage while their metadata is saved into sequence database to make it more convenient for reporting functions. Depending on whether the input image is single leaf or foliage, the cloud function will send it to disease detection model or leaf segmentation, respectively.

4.1.3 Disease Detection on spot

As part of frontend, offline application version is released to deal with the cases where there is no internet connection, no access to cloud services. The offline version is packaged with the built-in optimised disease detection model. This version can only handle single leaf images together with the sacrifice a bit of model accuracy and reporting functions.

4.1.3.1 Portable Application

Photos of foliage could be captured effectively by drones or mobile devices. A mobile application is covered in the scope of this experiment.

The mobile application was built by using the Flutter Development toolkit [91] for cross platform support while maintaining a single codebase. An example of project structure is presented in Figure 29. This framework also enables the integration of optimised model, TFLite for example, to run & serve locally where has no facilities to connect to a robust backend or web APIs.

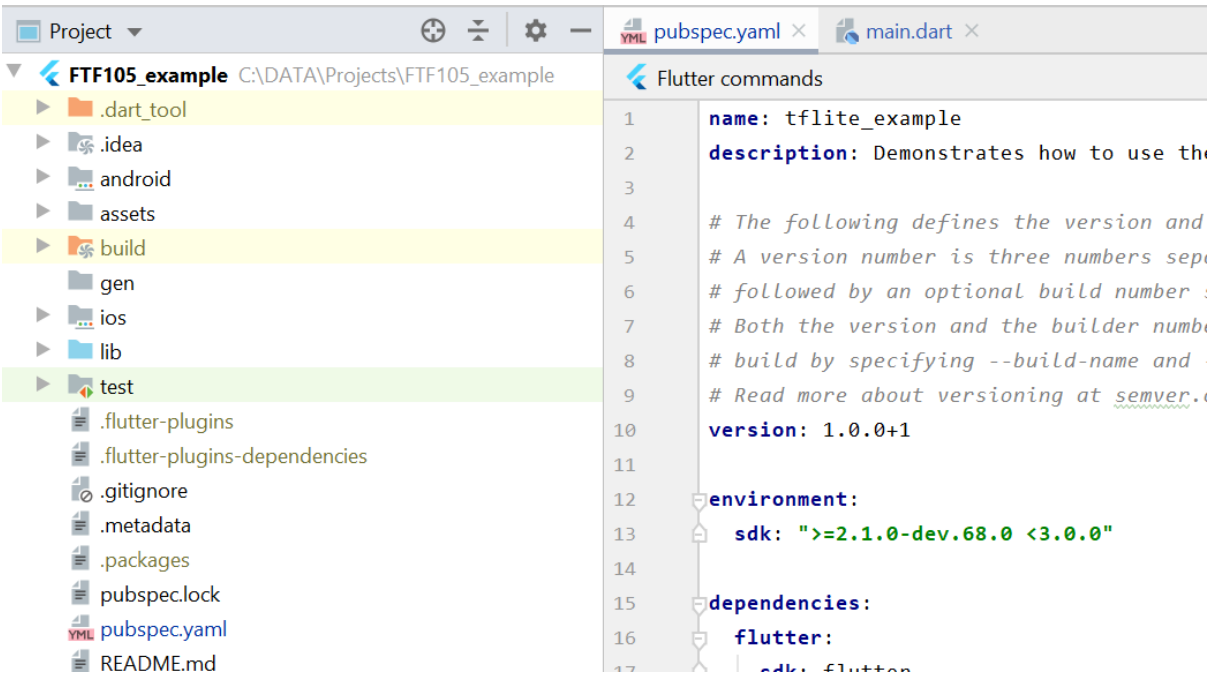

Figure 29: Flutter Project Structure Example

4.1.4 Backend

Backend comprises of cloud storage, database, and TensorFlow Serving containers. Azure and AWS both offer analogous cloud storage and database services that we could choose based on our preference. Similarly, TensorFlow Serving docker containers are able to be spin up on either Azure or AWS compute instances. The trained models are deployed on separate containers to make it more convenient for scalability respectively depending on how frequent they are requested from the front end.

Backend models are receiving input image from the frontend function. In the case that the input image is single leaf, not foliage, it will be passed directly to the classification model for disease detection. Otherwise, the leaf segmentation model will extract individual leaves from the input image then send back to cloud function for saving extracted images their metadata. Only after that, the extracted images are fit into the disease detection model for classifying. The final output images and prediction results are saved and return to client devices for reporting.

4.2 Functionality

4.2.1 Use Case Diagram

This use case diagram in Figure 30 describes a high-level overview of the relationship between use cases, actors, and systems. The boundary box at the centre defines a scope of use cases of is the Apple Leaf Disease Detection system. An actor represents a role played by an outside object, symbolized by a person figure. A use case is a series of events occurred when actors use a system to complete a process. Use cases are represented by horizontally shaped ovals. Solid lines match actors and their accessible use cases, namely associations. Dash lines represent the relationship between use cases.

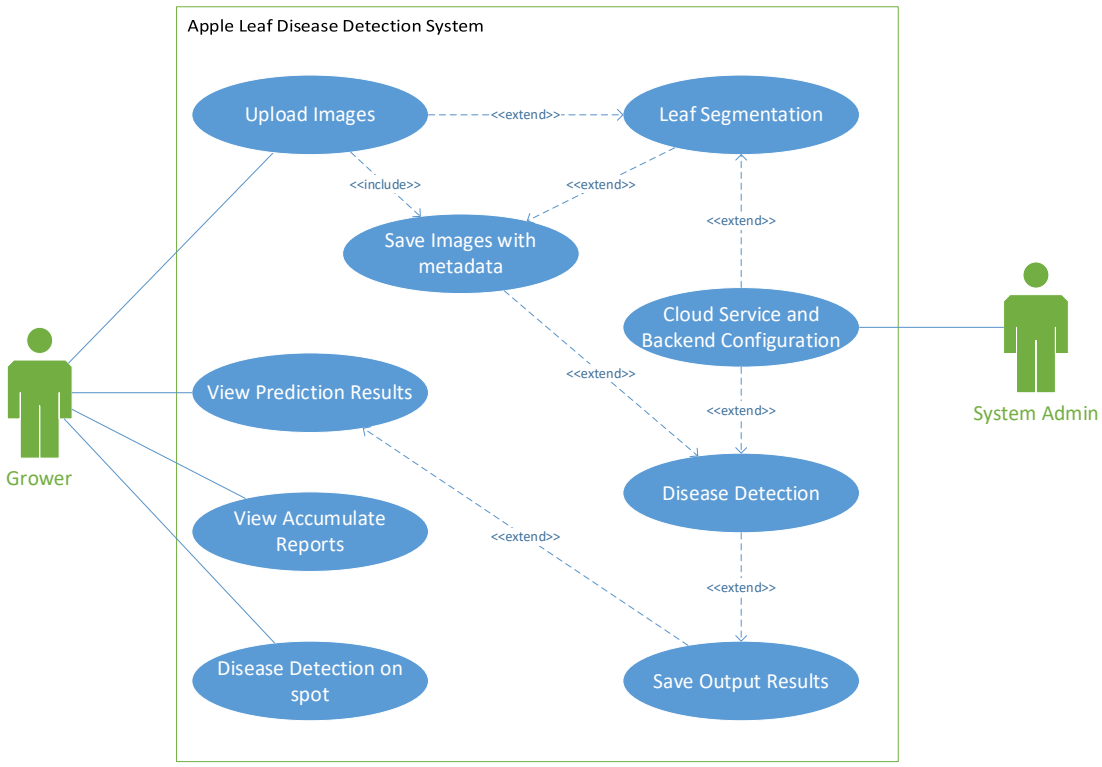

Figure 30: Use Case Diagram of Apple Leaf Disease Detection system

4.2.2 Sequence Diagram

Sequence diagrams, also known as event scenarios, are popular modelling solution which describe interaction among objects during the sequence. It also demonstrates message exchanged between objects while performing a function before the lifeline ends. The Figure 31 describe the process of the Apple Leaf Disease Detection system. First, growers capture images of either a single leaf or a foliage. In case the captured image is a

single leaf and detection type is on spot, the mobile application with built-in optimised model will classify and predict disease directly. Otherwise, captured images will be sent from the mobile app to cloud backend storage via cloud functions. The image reference with metadata of respective orchard is returned for further processing. If the image is foliage, it will be segmented to extract into images of single leaf before sending to backend ML serving model for disease prediction. The prediction outputs are returned to cloud functions to perform saving into the respective orchard storage, make it ready for report functions. Depending upon whether selected report type is simple prediction or accumulate report to view, mobile app will invoke cloud functions to load data from orchard storage and display the report respectively.

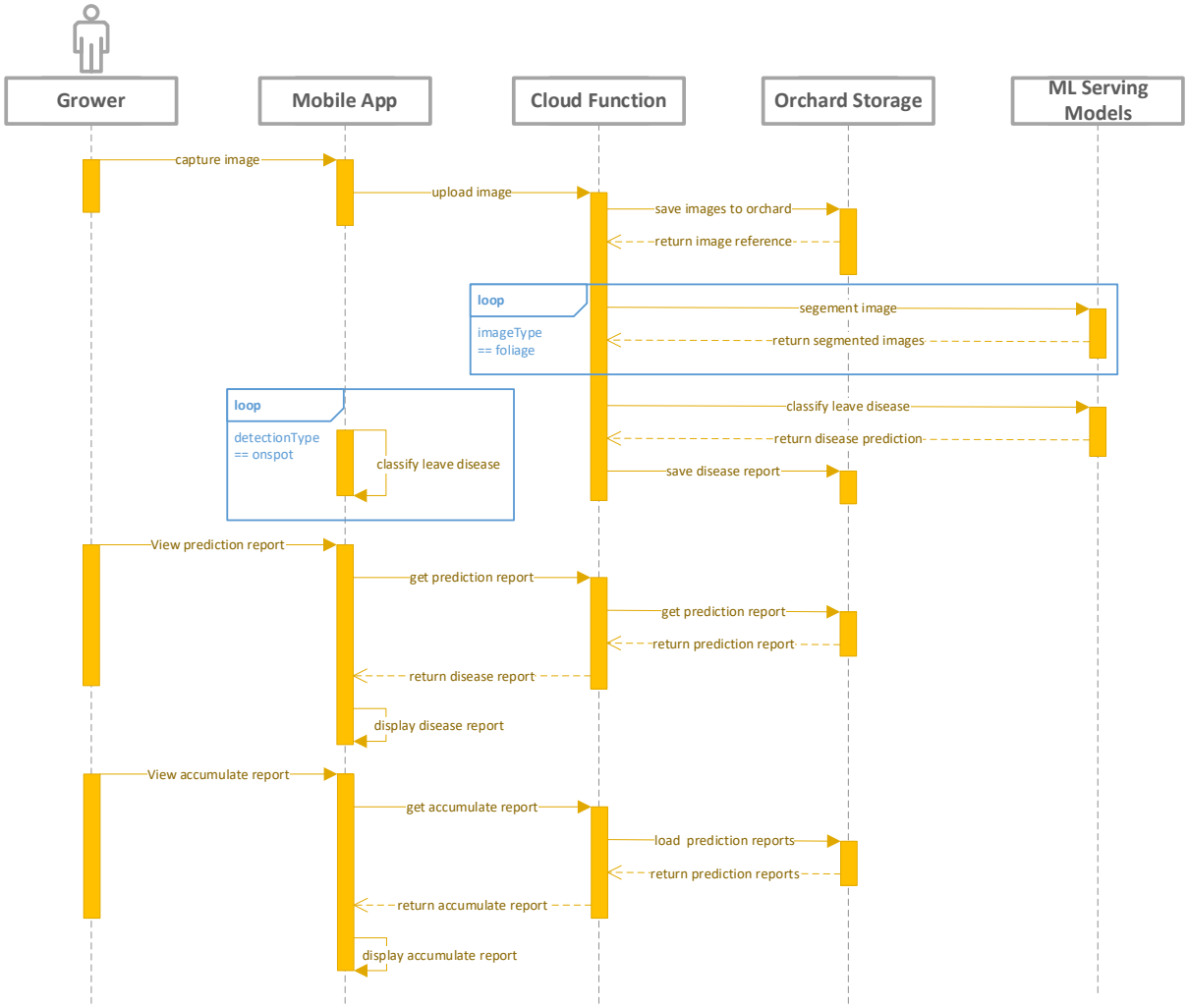

Figure 31: Sequence Diagram of Apple Leaf Disease Detection system which starts from Grower state

4.3 Development

4.3.1 Deep Learning Pipeline

The pipeline with two model steps is chosen for this research. First, Mask R-CNN analyses input images and extract individual leaves out for next step. Individual leaves are then be classified by the second classification model to identify healthy and other types of disease leaves. We have considered the approach of using one single model which outputs mashes or boxes. But necessary data set and required expert labelling effort is out of reach. Furthermore, that approach would require fully retrain process whenever new kinds of disease are added.

4.4 ML Model Implementation

4.4.1 Disease Detection

The disease detection module is built on the backbone of the CNN classification model. The main problem resolved by the disease detection module is to identify whether an input apple leaf is healthy. If a leaf is not healthy, the module would suggest what kind of disease is presenting on it.

4.4.1.1 Dataset for Classification

In this experiment, the apple leaf disease data set, a subset of Plant Village dataset [40], was used. The dataset consists of 3171 apple leaf images divided into 4 classes as listed in Table 11. The Figure 32 presents some sample images with labels.

Table 11: Images of Apple Leaf Diseases

Type	**Sample**
Gymnosporangium juniperivirginianae (rust)	275
Venturia insequalis (scrab)	630
Botryospaeria obtuse (rot)	621
Healthy	1645

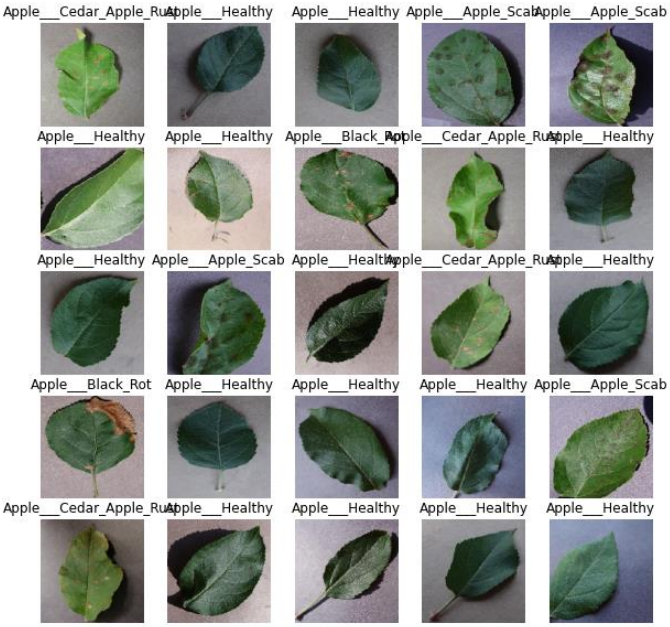

Figure 32: Samples with apple leaf disease labels

The selected dataset is split into three sets of train, validation and test detailed in Figure 33.

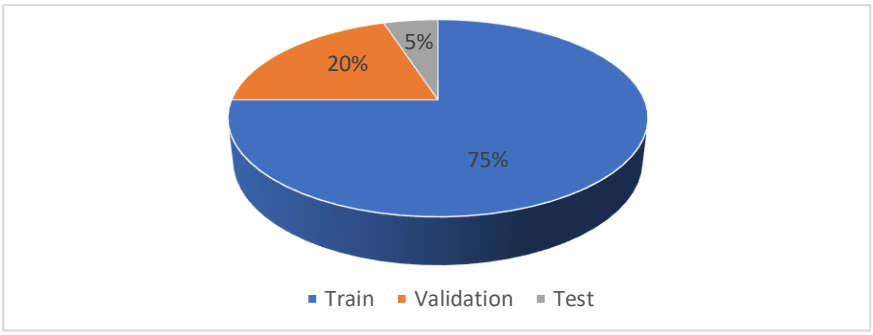

Figure 33: Split Image Dataset

To fit and train the model, the input images are converted to the size of (224, 224) and also be rescaled to the model expected float range of [-1, 1].

Since the dataset we have is not large enough, the simple augmentation techniques are applied, namely random flip and rotation. The Figure 34 present how the augmentation looks like on the same image.

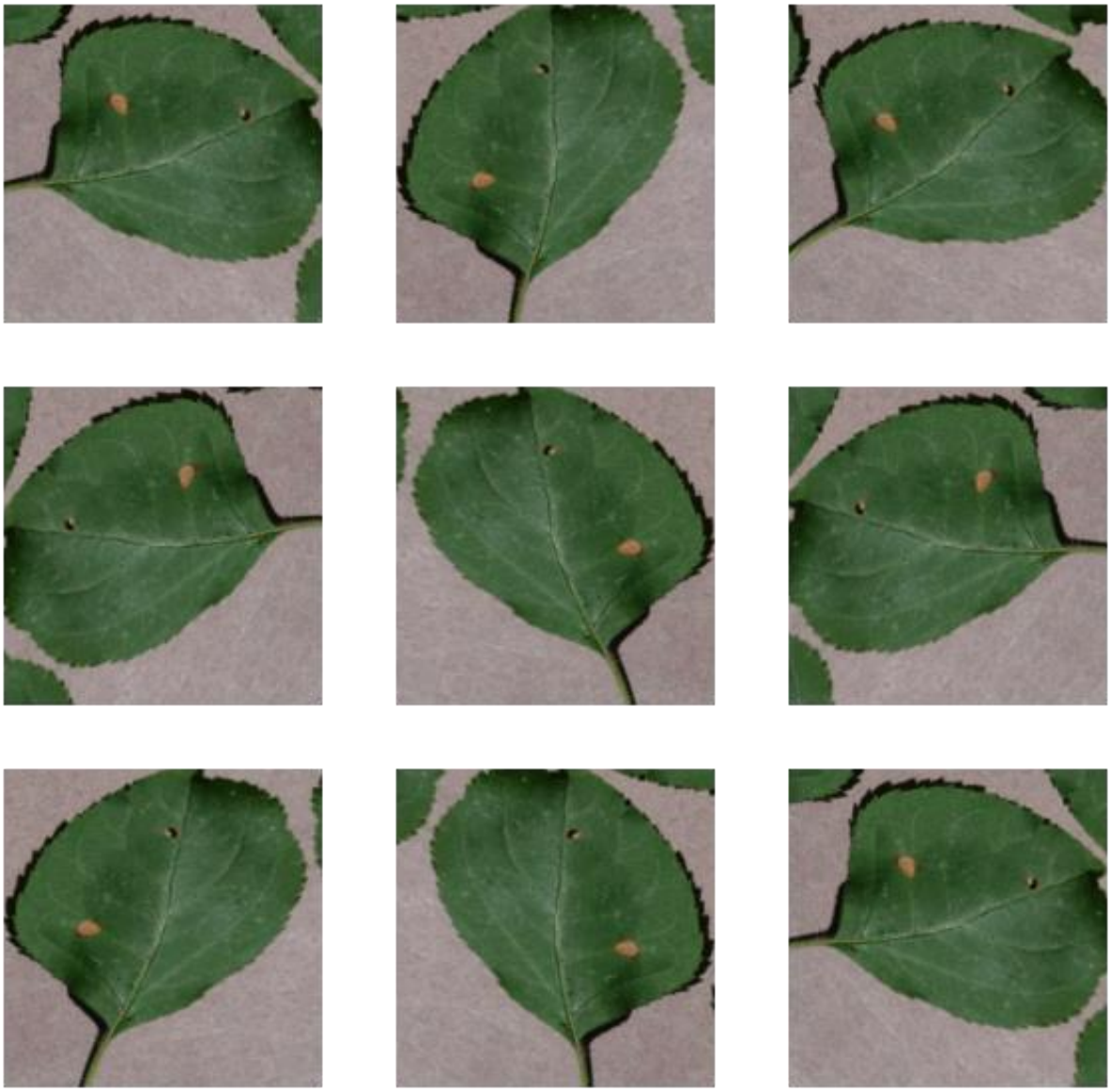

Figure 34: Flip and Rotation Augmentation on the same original image

4.4.1.2 Model Setup and Training

To train a CNN model from the scratch with random initialization, a large data set is required, and the training time would take very long. Those are the rationale of coming up with the utilisation of a pretrained model and transfer learning method. All of these are supported on the TensorFlow ecosystem. For the mobile optimisation purpose, the MobileNet V2 model [37] developed at Google is chosen. It is pretrained on the research training dataset ImageNet [39] which consists of 1.4M images and 1000 classes. In terms of feature extraction, it is a common practice to rely on the very last layer where retains more generality, before the flatten. The extracted feature, so-called base model, with pre-

trained weights is frozen and untrainable, only newly added output layer would be trained. The summary of the base model is presented in APPENDIX A.

On top of this base model, we add a classification head which is the only trainable layer at the first stage of training. It comprises 5124 parameters of weights and bias. The base model with its more than 2.2M parameters are frozen and will not be updated during the first stage. The architecture and summary of the model is described in Figure 35 and Table 12 respectively.

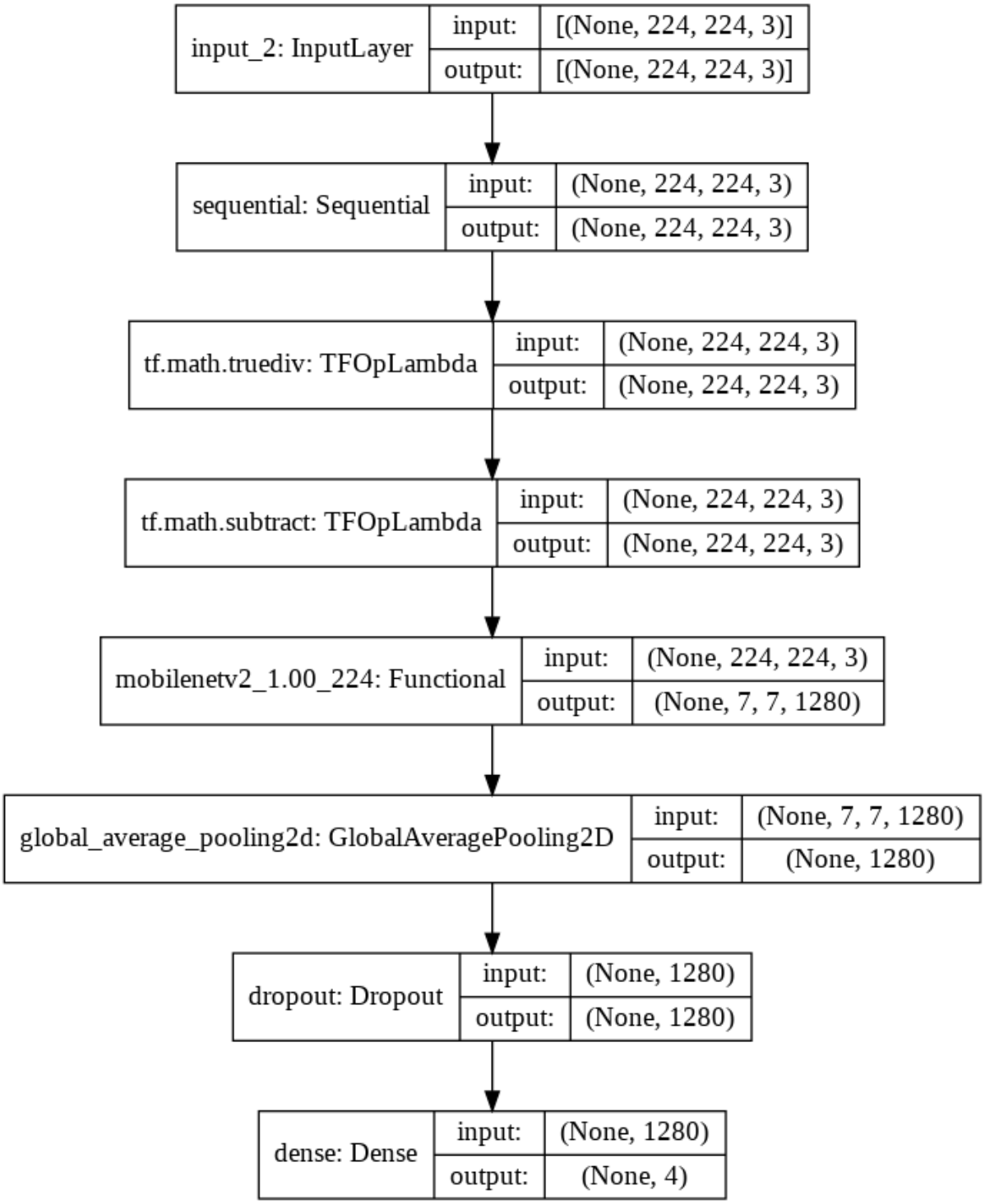

Figure 35: The Classification Model Architecture

Table 12: The Classification Model Summary

Layer (type)	**Output Shape**	**Param #**
input_2 (InputLayer)	[(None, 224, 224, 3)]	0
sequential (Sequential)	(None, 224, 224, 3)	0

tf.math.truediv (TFOpLambda)	(None, 224, 224, 3)	0
tf.math.subtract (TFOpLambda	(None, 224, 224, 3)	0
mobilenetv2_1.00_224 (Functi	(None, 7, 7, 1280)	2257984
global_average_pooling2d (Gl	(None, 1280)	0
dropout (Dropout)	(None, 1280)	0
dense (Dense)	(None, 4)	5124
Total params: 2,263,108 Trainable params: 5,124 Non-trainable params: 2,257,984		

For the model settings, Adam [47] is chosen as an optimiser with learning rate 0.001, and SparseCategoricalCrossentropy [4] is selected as the loss function. The classification metric, namely accuracy, is specified to report on accuracy.

Before training, the classification model with pretrained weights is loaded and evaluated against the validation dataset. The initial loss and accuracy are recorded at 1.6625 and 0.2799 respectively.

The training terminal is just a host on the cloud, with following hardware specification:

- GPU 0: Tesla T4 15GB GDDR5 VRAM
- CPU: 2 threads x 1 core - Intel(R) Xeon(R) CPU @ 2.20GHz
- Memory: 13GB

The first stage of training is set to run with only 10 epochs of 75 steps. The main purpose of this stage is to train the newly added classification head and make it work with the new set of classes instead of the original classification task. The training was completed within less than 20 minutes. The details of training performance and training scores are recorded in Figure 36 and Table 13 respectively.

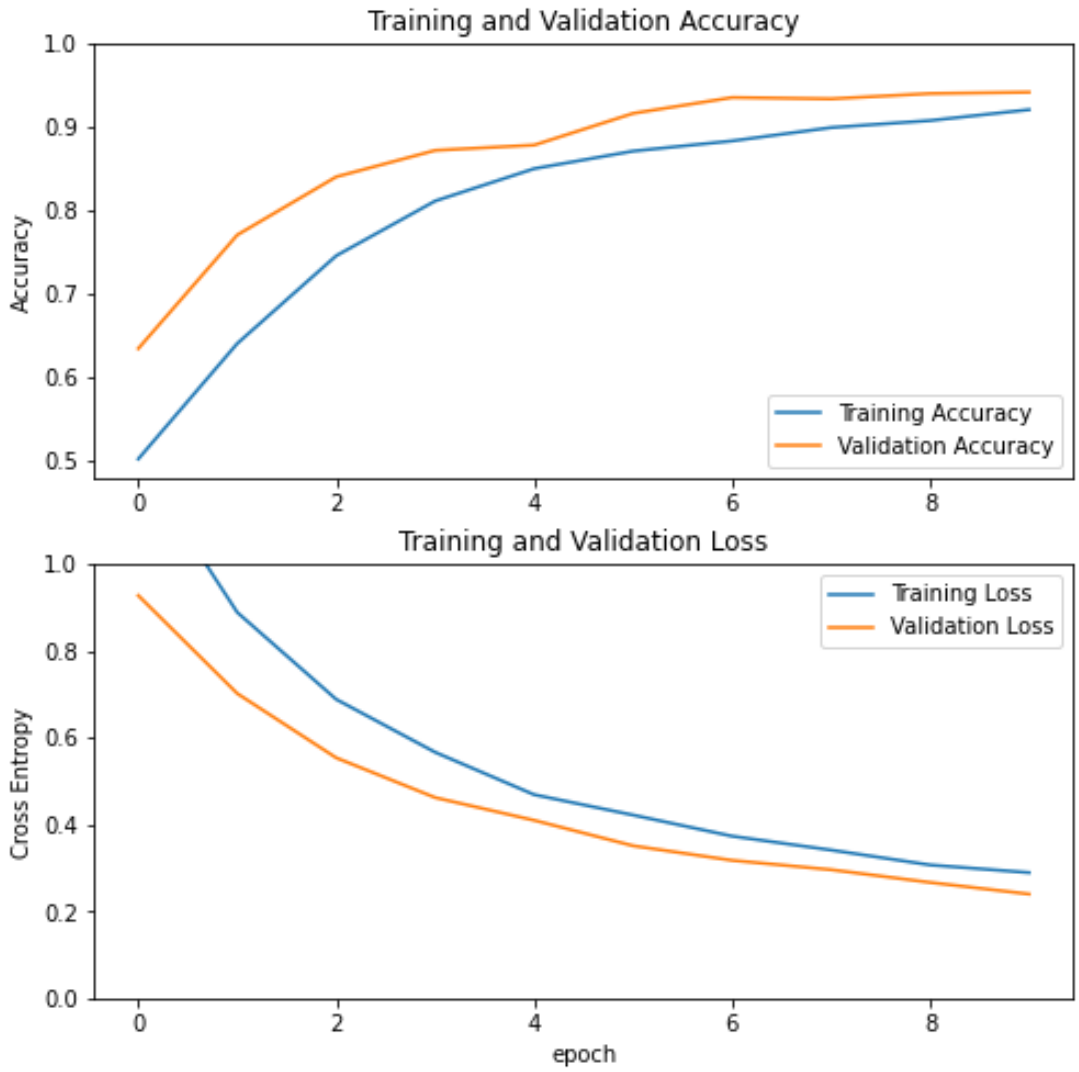

Figure 36: First Stage Training Performance

Table 13: First Stage Training Score

Dataset	**Loss**	**Accuracy**
Train	0.2892	0.9206
Validation	0.2397	0.9415

The outcome of the first training stage looks quite positive. The validation scores, loss, and accuracy, always be better than that of training. This trend might be the case when the validation dataset contains easier samples than training samples. However, the validation, training, and test datasets are split randomly in this experiment. Furthermore, the same trend persists after many tries. It means that the model is not overfitting. The main factor

of this trend is because the layers BatchNormalization and Dropout influence accuracy in training. Those are turned off when calculating validation loss. It is also because the training metrics are calculated as the average of an epoch while the validation metrics are evaluated only at the end of an epoch, when the model is better by having a longer training time.

In the first the stage of training, we have trained the final classification added on top the base model Mobilenet V2. However, the weights of the base model were not updated during the first stage of training. To improve performance of the model further, we are applying fine-tune technique to train the last 54 layers of the base model alongside with the training of the added classifier, while the first 100 layers of the base model are frozen. Basically, this process is to tune the generic feature maps to adapt with the features associated with the provided dataset. The purpose of freezing the base model in the first training stage and the first 100 of its layers is to minimise the magnitude of the gradient updates, retrain knowledge the pretrained model has learned. It is very common in CNN models that the first layers learn very simple and generic feature to all most all type of images. The higher layers are more specific to the training dataset. Overall, the main target of the fine-tuning stage is those specialized features of the last layers and making them adapt with training dataset, rather than overwriting generic feature learning.

Before starting the fine-tuning stage, the model summary and trainable param is represented in Table 14. Total trainable layers are 56 which consists of the 54 last layers of the base model and two layers of the added classification head.

Table 14: Classification Model Summary for Fine Tune

Layer (type)	**Output Shape**	**Param #**
input_2 (InputLayer)	[(None, 224, 224, 3)]	0
sequential (Sequential)	(None, 224, 224, 3)	0
tf.math.truediv (TFOpLambda)	(None, 224, 224, 3)	0
tf.math.subtract (TFOpLambda	(None, 224, 224, 3)	0
mobilenetv2_1.00_224 (Functi	(None, 7, 7, 1280)	2257984
global_average_pooling2d (Gl	(None, 1280)	0
dropout (Dropout)	(None, 1280)	0
dense (Dense)	(None, 4)	5124
Total params: 2,263,108 Trainable params: 1,866,564 Non-trainable params: 396,544		

This stage of training is performed with almost the same settings with the first stage. The only difference is the reduce of learning rate to 0.0001 to avoid the model being overfit quickly. The reason is scope of the fine-tuning training is larger and we want to readapt the pre-train model.

From the training history plotted in Figure 37, we can see the validation loss is not too far from training loss. It means the model is less overfitting. Overfitting will happen in the case that the validation loss is much higher than training loss. It could also be results of the cases of using a small training dataset and they are similar to the original dataset which is ImageNet in this case.

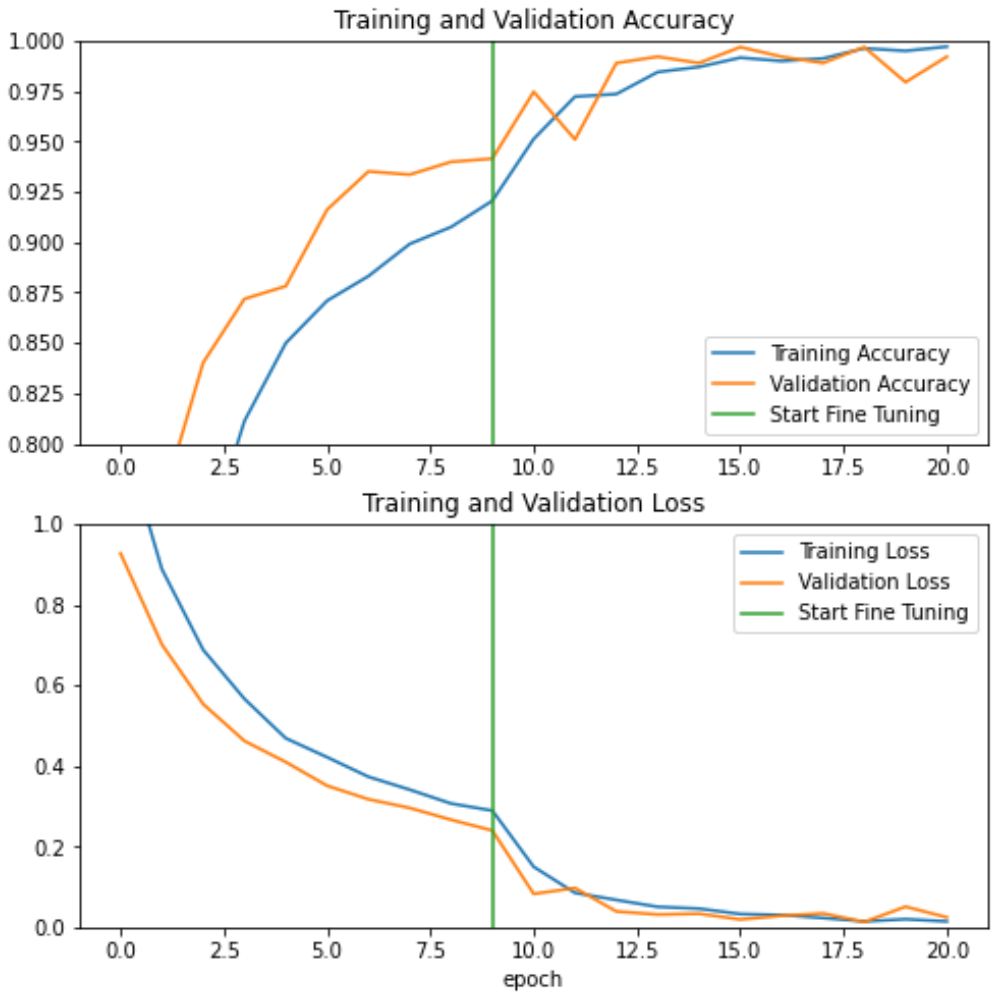

Figure 37: Fine-Tuning Training Performance

After completing the fine-tuning training, the model performance is improved a few percentages, reached more than 99% accuracy on both training and validation dataset. The details of final score are recorded in Table 15.

Table 15: Fine-Tuning Training Score

Dataset	**Loss**	**Accuracy**
Train	0.0123	0.9984
Validation	0.0243	0.9921

Impressively, this final classification model gives the score of more than 99% accuracy when evaluated on the test dataset as details in Table 16. More intuitive results of one prediction batch are present in Table 17 and Figure 38.

Table 16: Classification Score on Test

Dataset	**Loss**	**Accuracy**
Test	0.0158	0.9937

Table 17: Predictions on Test Dataset

Predictions	3 0 3 0 0 0 2 0 3 1 2 3 2 0 0 1 0 1 3 3 3 3 2 2 3 2 0 0 3 3 3 3
Labels	3 0 3 0 0 0 2 0 3 1 2 3 2 0 0 1 0 1 3 3 3 3 2 2 3 2 0 0 3 3 3 3

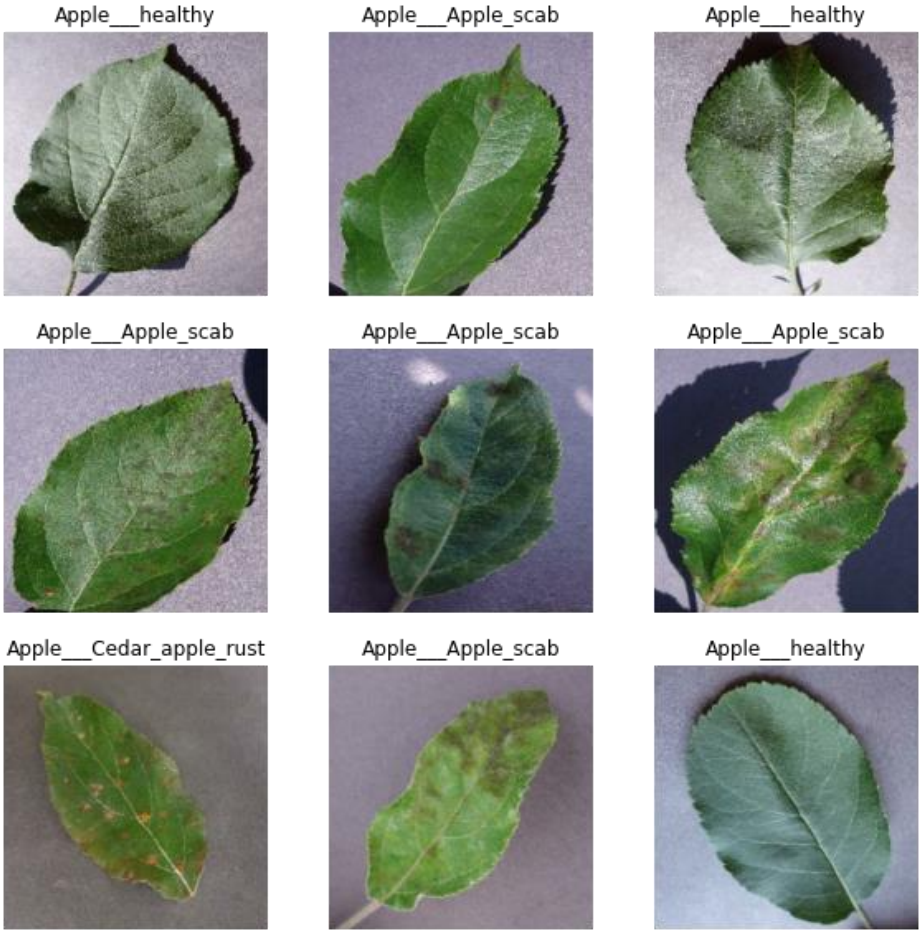

Figure 38: Samples of Test Predictions. Wrong Predictions having red title.

4.4.1.3 Convert Trained Model to TensorFlow Lite

After the full model is trained and saved with a good score, we would need to convert it to a lite model to be able to run locally on mobile or edge devices. Fortunately, as part of TensorFlow ecosystem, TensorFlow Lite Converter [99] is a convenient tool supporting convert a full model to lite model. TensorFlow Lite Converter applies post-training quantization, a conversion technique to reduce model size while also improving CPU and hardware accelerator latency, with little degradation in accuracy [86].

The most comprehensive optimization technique, quantization, is applied on the trained model of disease detection. The input and output shapes of the TensorFlow Lite model is also resized to handle batches of 32 images as details in Table 18.

Table 18: TensorFlow Lite model details

Input Details	**Output Details**
name: serving_default_input_4:0 shape: [32 224 224 3] type: <class 'numpy.float32'>	name: StatefulPartitionedCall:0 shape: [32 4] type: <class 'numpy.float32'>

Based on results, the model size is reduced by about 40% as presented in Figure 39, while the accuracy is kept at around 99% validated on the lab dataset.

```
2.6M Jun  6 01:17 aldd_quant.tflite
4.3M Jun  6 01:17 saved_model.pb
```

Figure 39: Size in Model Optimization

4.4.2 Leaf Segmentation

Using the core Mask R-CNN framework, the leaf segmentation module deal with images of apple foliage. Ideally, single leaves will be extracted by the leaf segmentation module to be used as inputs on the classification module.

4.4.2.1 Dataset of foliage

Total 84 images of apple foliage captured on the field with natural surroundings were annotated and split by the percentage of 80 – 20 for training and validation accordingly. Images are taken from arbitrary directions, such as from top, down or side positions. Number of leaves in each image are different, ranging from a few to about 50. Leaves in the images were then annotated by using VGG Image Annotator [100], as resulted in Figure 40.

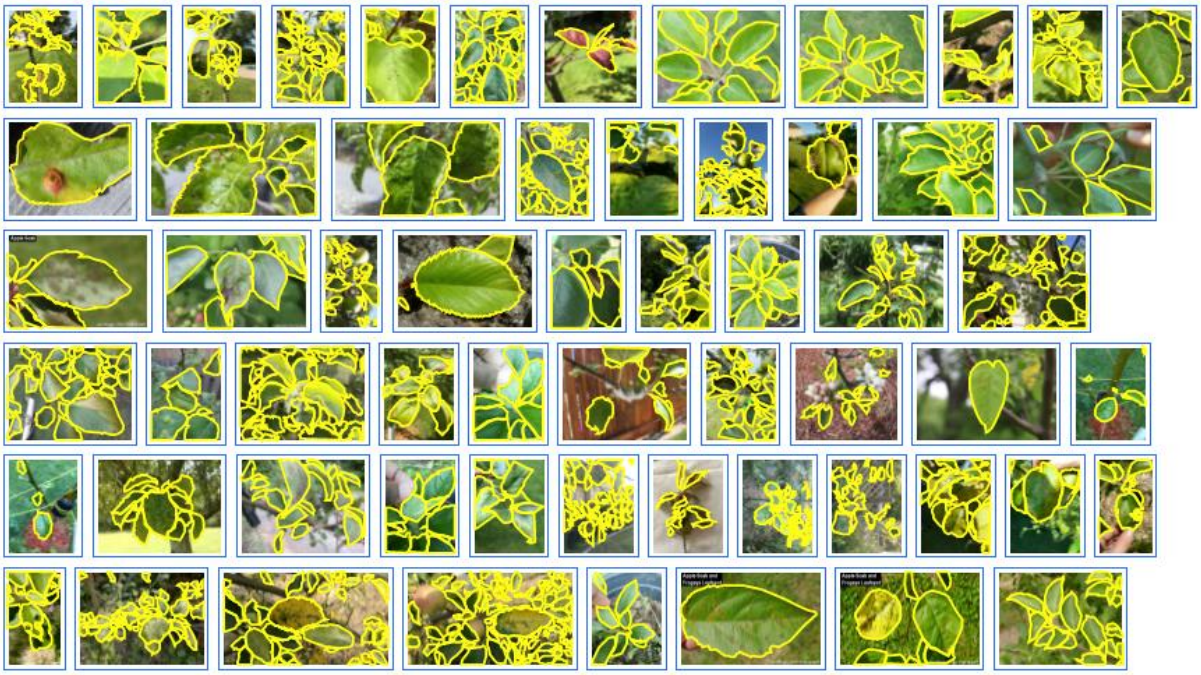

Figure 40: Annotated Images

4.4.2.2 *Setup Mask R-CNN model*

For this research, Mask R-CNN model was extended from the Matterport Mask R-CNN [101]. Ideally, the transfer learning approach is applied. Instead of training from scratch, it starts with weights which were trained on COCO dataset [41]. The backbone network architecture is resnet101.

The model configurations are also be applied appropriately as in Table 19.

Table 19: Leaf Segmentation Model Configuration

BACKBONE	resnet101
BACKBONE_STRIDES	[4, 8, 16, 32, 64]
BATCH_SIZE	2
BBOX_STD_DEV	[0.1 0.1 0.2 0.2]
COMPUTE_BACKBONE_SHAPE	None
DETECTION_MAX_INSTANCES	100
DETECTION_MIN_CONFIDENCE	0.9
DETECTION_NMS_THRESHOLD	0.3
FPN_CLASSIF_FC_LAYERS_SIZE	1024
GPU_COUNT	1
GRADIENT_CLIP_NORM	5
IMAGES_PER_GPU	2
IMAGE_CHANNEL_COUNT	3

IMAGE_MAX_DIM	1024
IMAGE_META_SIZE	14
IMAGE_MIN_DIM	800
IMAGE_MIN_SCALE	0
IMAGE_RESIZE_MODE	square
IMAGE_SHAPE	[1024 1024 3]
LEARNING_MOMENTUM	0.9
LEARNING_RATE	0.001
LOSS_WEIGHTS	{ 'rpn_class_loss': 1.0, 'rpn_bbox_loss': 1.0, 'mrcnn_class_loss': 1.0, 'mrcnn_bbox_loss': 1.0, 'mrcnn_mask_loss': 1.0 }
MASK_POOL_SIZE	14
MASK_SHAPE	[28, 28]
MAX_GT_INSTANCES	100
MEAN_PIXEL	[123.7 116.8 103.9]
MINI_MASK_SHAPE	(56, 56)
NAME	ald
NUM_CLASSES	2
POOL_SIZE	7
POST_NMS_ROIS_INFERENCE	1000
POST_NMS_ROIS_TRAINING	2000
PRE_NMS_LIMIT	6000
ROI_POSITIVE_RATIO	0.33
RPN_ANCHOR_RATIOS	[0.5, 1, 2]
RPN_ANCHOR_SCALES	(32, 64, 128, 256, 512)
RPN_ANCHOR_STRIDE	1
RPN_BBOX_STD_DEV	[0.1 0.1 0.2 0.2]
RPN_NMS_THRESHOLD	0.7
RPN_TRAIN_ANCHORS_PER_IMAGE	256
STEPS_PER_EPOCH	100
TOP_DOWN_PYRAMID_SIZE	256
TRAIN_BN	FALSE
TRAIN_ROIS_PER_IMAGE	200
USE_MINI_MASK	TRUE
USE_RPN_ROIS	TRUE
VALIDATION_STEPS	50
WEIGHT_DECAY	0.0001

The number of classes is set to two, one for apple leaves and one for background. Steps per one of 30 epochs is 100 since we will be training the on a medium compute resource. Detection threshold, minimum probability value to accept a detected instance, is 0.9. Any

ROIs is less than the threshold will be ignored. The learning rate is initiated at relatively low value, 0.001.

Chapter 5

EVALUATION AND DEMONSTRATION

The main focus in this section is performance assessment of machine learning models. The inference time of lite model and costing is also evaluated and reported in detail. Additionally, this chapter shows how the software solution runs and demo some main use cases.

5.1 Disease Detection Model

As discussed in Model Setup and Training, the Disease Detection model yield good performance, accuracy score reaches 99.37% on test dataset. It also less overfitting since the validation loss is not too far from training loss.

5.2 Classification and Portable App

The model and the prototype application were tested by some images from various sources rather than lab images. The image sources are a public community with communal discussions among growers and plant pathologists and direct sharing to researcher from growers. The application was able to load images, transform and pass them to the optimised model for inference. The results are quite positive, captured in Figure 41.

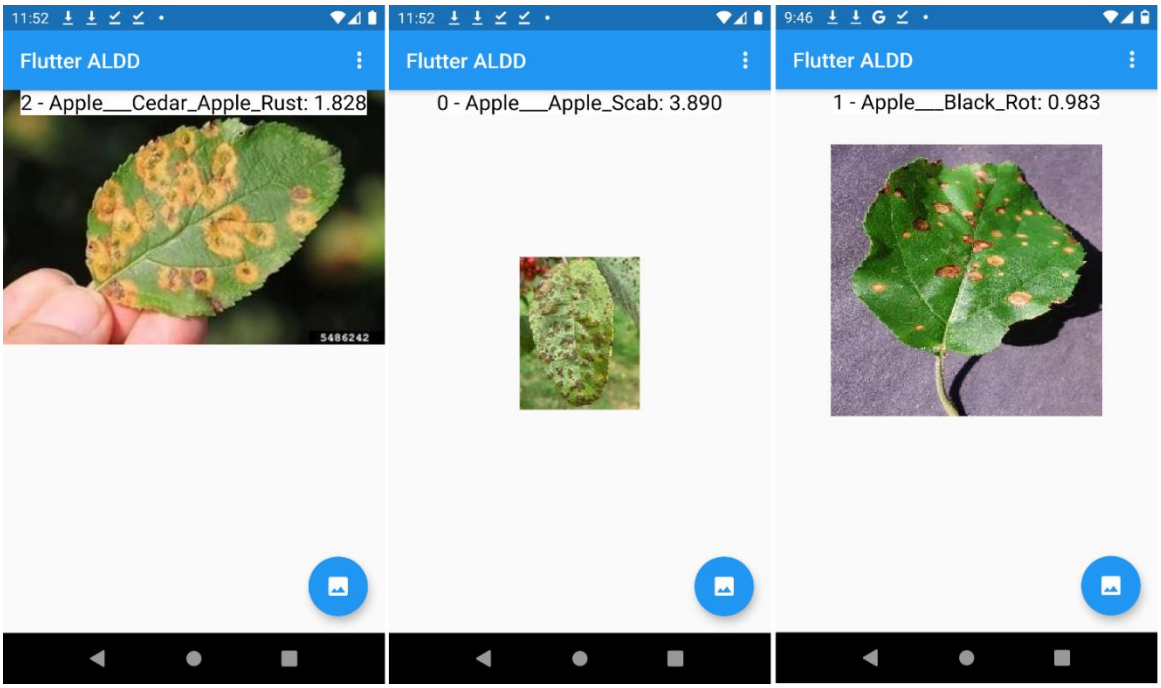

Figure 41: ALDD Testing with Prediction Results printed

The assembled application with the built-in model is another form of diseases insight. Once it is published to mobile app stores, it becomes accessible by numerous growers to assist them in application leaf detection in the field.

In the cases of dealing with new diseases or new plants, only the first step of training data preparation is required, the whole process replicated to produce a new artifact for new requirements.

Overall, our ideal and convenient approach may benefit researchers and practitioners in creation a DL mobile app to solve similar problems or doing other related research matters.

5.3 Lite Model of Disease Detection

The accuracy of the converted and optimised model was measured before packaging into the application project solution. The score was recorded at 0.994 when testing on all lab images, gave 3152 correct results on total 3171 samples. The lite model predictions for one batch of images were captured and visualized in Figure 42.

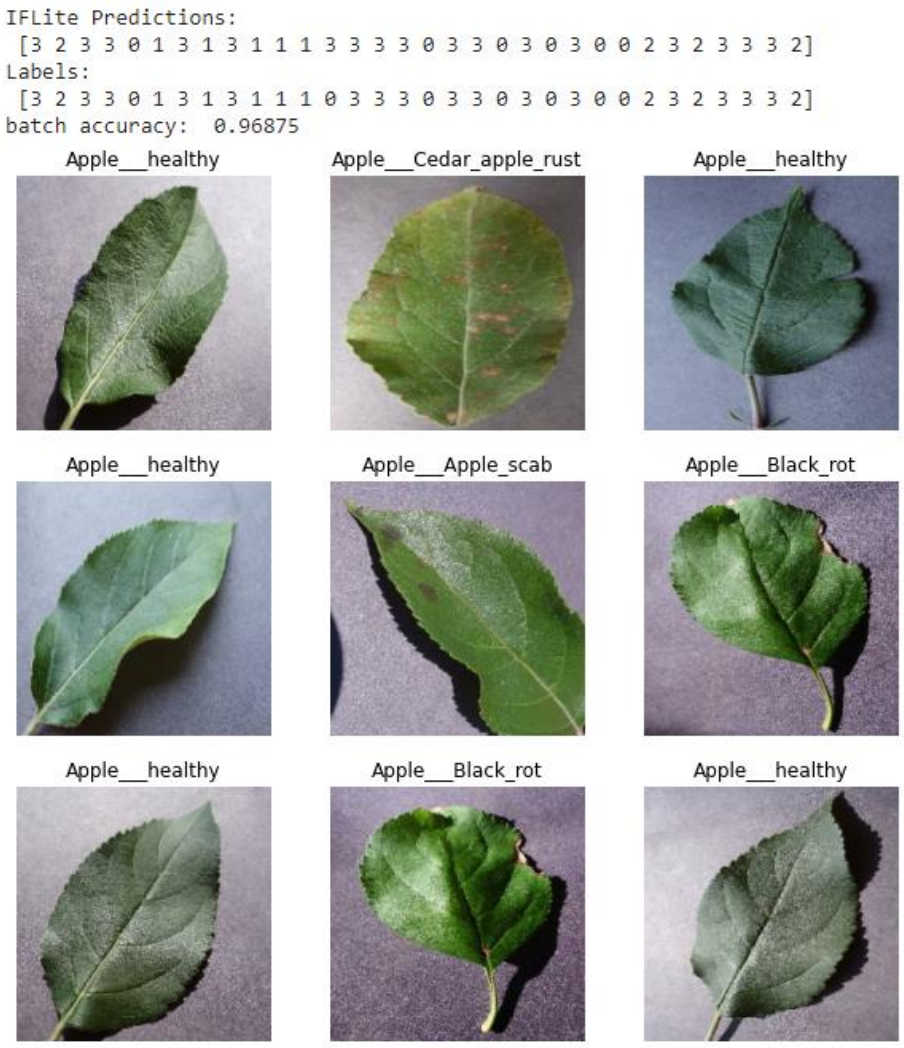

Figure 42: One Prediction Batch of TFLite Model. Wrong Prediction titles in red.

The entire trail of accuracy evaluation is recorded in Table 20.

Table 20: TFLite Model Accuracy Score

Batch 0 started ... > accuracy: 1.0	Batch 50 started ... > accuracy: 1.0
Batch 1 started ... > accuracy: 1.0	Batch 51 started ... > accuracy: 0.96875
Batch 2 started ... > accuracy: 1.0	Batch 52 started ... > accuracy: 1.0
Batch 3 started ... > accuracy: 1.0	Batch 53 started ... > accuracy: 1.0
Batch 4 started ... > accuracy: 0.96875	Batch 54 started ... > accuracy: 1.0
Batch 5 started ... > accuracy: 1.0	Batch 55 started ... > accuracy: 0.96875
Batch 6 started ... > accuracy: 1.0	Batch 56 started ... > accuracy: 1.0
Batch 7 started ... > accuracy: 1.0	Batch 57 started ... > accuracy: 1.0
Batch 8 started ... > accuracy: 1.0	Batch 58 started ... > accuracy: 1.0
Batch 9 started ... > accuracy: 1.0	Batch 59 started ... > accuracy: 1.0
Batch 10 started ... > accuracy: 1.0	Batch 60 started ... > accuracy: 1.0

Batch 11 started ... > accuracy: 1.0	Batch 61 started ... > accuracy: 0.96875
Batch 12 started ... > accuracy: 0.96875	Batch 62 started ... > accuracy: 1.0
Batch 13 started ... > accuracy: 1.0	Batch 63 started ... > accuracy: 1.0
Batch 14 started ... > accuracy: 1.0	Batch 64 started ... > accuracy: 0.96875
Batch 15 started ... > accuracy: 1.0	Batch 65 started ... > accuracy: 0.96875
Batch 16 started ... > accuracy: 1.0	Batch 66 started ... > accuracy: 1.0
Batch 17 started ... > accuracy: 1.0	Batch 67 started ... > accuracy: 0.96875
Batch 18 started ... > accuracy: 1.0	Batch 68 started ... > accuracy: 1.0
Batch 19 started ... > accuracy: 1.0	Batch 69 started ... > accuracy: 1.0
Batch 20 started ... > accuracy: 1.0	Batch 70 started ... > accuracy: 1.0
Batch 21 started ... > accuracy: 1.0	Batch 71 started ... > accuracy: 0.96875
Batch 22 started ... > accuracy: 1.0	Batch 72 started ... > accuracy: 1.0
Batch 23 started ... > accuracy: 0.96875	Batch 73 started ... > accuracy: 1.0
Batch 24 started ... > accuracy: 1.0	Batch 74 started ... > accuracy: 1.0
Batch 25 started ... > accuracy: 1.0	Batch 75 started ... > accuracy: 1.0
Batch 26 started ... > accuracy: 0.96875	Batch 76 started ... > accuracy: 1.0
Batch 27 started ... > accuracy: 1.0	Batch 77 started ... > accuracy: 1.0
Batch 28 started ... > accuracy: 0.9375	Batch 78 started ... > accuracy: 1.0
Batch 29 started ... > accuracy: 1.0	Batch 79 started ... > accuracy: 0.96875
Batch 30 started ... > accuracy: 1.0	Batch 80 started ... > accuracy: 0.96875
Batch 31 started ... > accuracy: 0.96875	Batch 81 started ... > accuracy: 1.0
Batch 32 started ... > accuracy: 0.96875	Batch 82 started ... > accuracy: 1.0
Batch 33 started ... > accuracy: 1.0	Batch 83 started ... > accuracy: 0.96875
Batch 34 started ... > accuracy: 1.0	Batch 84 started ... > accuracy: 1.0
Batch 35 started ... > accuracy: 1.0	Batch 85 started ... > accuracy: 1.0
Batch 36 started ... > accuracy: 1.0	Batch 86 started ... > accuracy: 1.0
Batch 37 started ... > accuracy: 1.0	Batch 87 started ... > accuracy: 1.0
Batch 38 started ... > accuracy: 1.0	Batch 88 started ... > accuracy: 1.0
Batch 39 started ... > accuracy: 1.0	Batch 89 started ... > accuracy: 1.0
Batch 40 started ... > accuracy: 1.0	Batch 90 started ... > accuracy: 1.0
Batch 41 started ... > accuracy: 1.0	Batch 91 started ... > accuracy: 1.0
Batch 42 started ... > accuracy: 1.0	Batch 92 started ... > accuracy: 1.0
Batch 43 started ... > accuracy: 1.0	Batch 93 started ... > accuracy: 1.0
Batch 44 started ... > accuracy: 1.0	Batch 94 started ... > accuracy: 1.0
Batch 45 started ... > accuracy: 1.0	Batch 95 started ... > accuracy: 1.0
Batch 46 started ... > accuracy: 0.96875	Batch 96 started ... > accuracy: 1.0
Batch 47 started ... > accuracy: 1.0	Batch 97 started ... > accuracy: 1.0
Batch 48 started ... > accuracy: 1.0	Batch 98 started ... > accuracy: 1.0
Batch 49 started ... > accuracy: 1.0	Batch 99 started ... > accuracy: 1.0
TensorFlow Lite model accuracy: 99.40625%	

A limited number of fresh images which were not captured under lab condition have been used to measure overall performance of the mobile application. The image sources

are a public community with communal discussions among growers and plant pathologists and direct sharing to researcher from growers. The test results were positive and captured in Figure 41. All activities including loading an image, reformatting, inference, and present descriptions took less than a second when running on a client device with system details in Table 21.

Table 21: Client Device Specification

Name	Pixel_2
CPU/ABI	Google APIs Intel Atom (x86)
SD Card	512M
hw.ramSize	1536
hw.cpu.ncore	4
vm.heapSize	256
disk.dataPartition.size	6G
hw.gpu.enabled	yes

In terms of implementation solution, all tools and frameworks applied in the project are open source, zero cost for environment setup or cloud service subscription. The development effort was also saved by using the cross-platform framework Flutter. It means that only one time of effort was required to implement single codebase for multiple applications of different platforms. The single codebase of the application also helps to reduce amount of effort needed for maintenance.

At this stage of the project, some places to improve the application performance, main function and UI had been discovered. The project could go back to the design and development activity for another iteration. However, the final decision is to leave it for subsequent projects and continue the next activity of the research.

5.4 Leaf Segmentation

Leaf segmentation model is only evaluated visually and the outcome is reported under this section. The output of leaf segmentation model will be snipped and passed to disease detection model for diagnosis. Main purpose here is to proof that the full ML pipeline solution is feasible.

By using mask object detection, the leaf segmentation was able to highlight and segregate many leaves from the foliage images as depicted in Figure 43. Those individual leaves could be extracted and fed into the disease detection model.

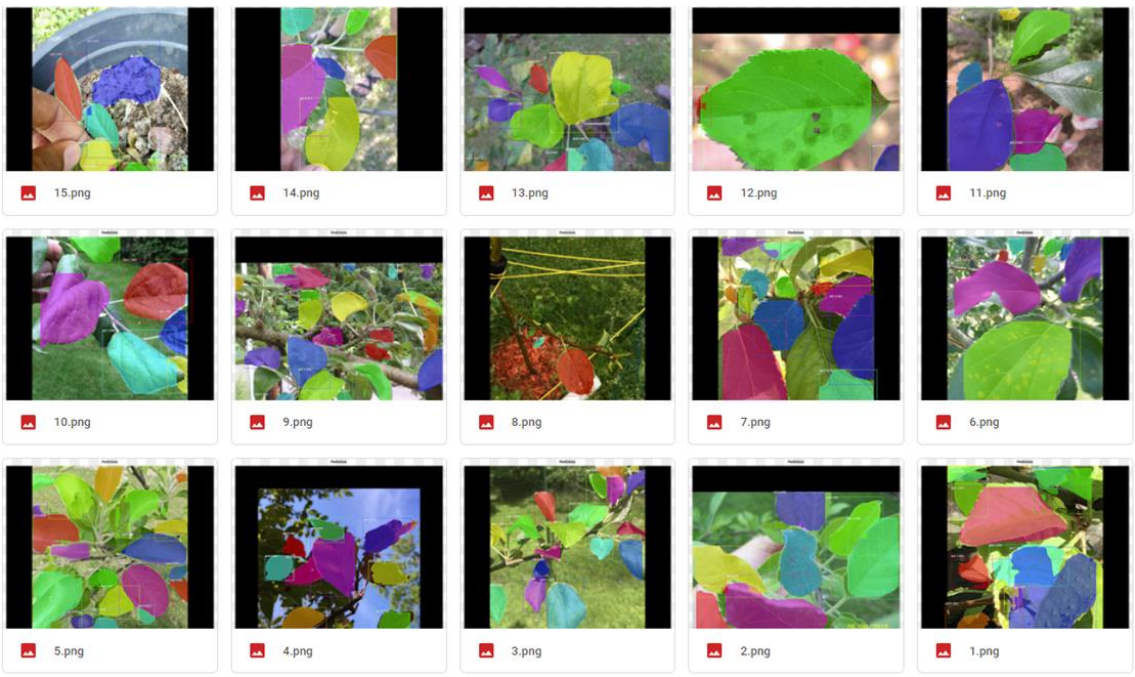

Figure 43: Demonstration of Leaf Segmentation mode. Segmented Leaves are shared in various colour.

The disease detection model and the prototype application were tested by some images from various sources rather than lab images, yielding positive results. The image sources are a public community with communal discussions among growers and plant pathologists and direct sharing to researcher from growers.

The assembled application integrated with the trained model is another collection of disease knowledge. It could be installed on multiple portable devices such as mobile phone or drones to operate on the field by any device owner.

Chapter 6

FINDING, DISCUSSION

This chapter discuss achievements of the solution as well as its drawbacks.

The disease detection model yielded high accuracy results after just short training duration by applying transfer learning technique from a pretrained DL CNN model, combined with appropriate fine-tuning. It is also optimised for deployment on mobile and embedded devices to operate on the field. The case of on spot disease detection eliminates the need of internet connection nor cloud backend service. Thus, it can operate at zero cost. Another advantage of this approach is that client data and activity history is secured on the client device itself. However, TFLite models are not trainable. In the case that there are often new datasets or label classes for the model to be retrained, for example importing new diseases, cloud API approach should be a better option. It is also the solution if a higher accuracy is required. Other functions such as segmentation, disease detection from foliage images, and viewing accumulate report for each orchard are also supplied by cloud service backend.

The instance segmentation approach in our leaf segmentation automatically eliminates background noise challenge posed in research questions. It is a significant relief for our disease detection model. The elimination fully relies on the ability of the leaf segmentation model to extract individual leaves from its foliage. The output of segmentation model are just individual leaves without any background. On the other hand, the instance segmentation model consumes quite amount of time for inference tasks during our test. It would be a problem to deal with if real time experience is required.

The image dataset used to train the disease detection model are mainly captured by top-down direction, resulted in upper epidermis. Thus, to make sure models working at good performance, it is recommended to take pictures with the same surface of leaves, even though the model is still yielding good results on some tested images of lower surface.

Chapter 7

CONCLUSIONS AND FUTURE WORKS

The research is summarized in this chapter, along with the conclusion drawn from the obtained findings in this report. By undertaking future scope proposed in this chapter, a further area of research and improvement could be recommended.

The development of DL gets many complex tasks done simply, faster with higher accuracy compared to using traditional computer vision techniques. It opens more potential applications in agriculture. By applying transfer learning technique with proper fine-tuning, it is possible to have a very good performance model after just short training period. In our experiment, the accuracy score of apple leaf disease detection model remains at more than 99% even after being converted and optimised for mobile deployment. The leaf segmentation proof to be working with visually validated results. Overall, the portable system with combination of ML models and lab dataset is promising in dealing with practical disease in orchards. The outstanding part for future work is the accuracy score of end-to-end solution. This would be done by collecting more input samples with pathologist involvement.

Expanding application broadly to farm and orchard level with more crucial feature may also be a suggestion for future work, such as tracking location of captured disease or inference severity of diseases from images.

Additionally, it is possible to corporate with apple growers and apple leaf disease expertise to collect more images, extend dataset for leaf segmentation, and carry out intensive evaluation.

References

[1] MathWorks. "Convolutional Neural Network." MathWorks. https://www.mathworks.com/solutions/deep-learning/convolutional-neural-network.html (accessed June 15, 2020).

[2] cs231n. "Convolutional Neural Networks (CNNs / ConvNets)." https://cs231n.github.io/convolutional-networks/ (accessed June 15, 2020).

[3] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee, "A design science research methodology for information systems research," *Journal of management information systems*, vol. 24, no. 3, pp. 45-77, 2007.

[4] A. A. Martín Abadi, Paul Barham, Eugene Brevdo, *et al.*, *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*, 2015. [Online]. Available: https://www.tensorflow.org/.

[5] D. He, J. Zhan, and L. Xie, "Problems, challenges and future of plant disease management: from an ecological point of view," *J. Integr. Agric,* vol. 15, pp. 705-715, 2016.

[6] A. Kantarci. "Image Recognition in 2021: In-depth Guide." https://research.aimultiple.com/image-recognition/ (accessed 13 June, 2021).

[7] F. AI. "Image Recognition Guide." https://www.fritz.ai/image-recognition/ (accessed 13 June, 2021).

[8] MathWorks. "What Is Image Recognition?" https://www.mathworks.com/discovery/image-recognition-matlab.html (accessed 13 June, 2021).

[9] Dataman. "What Is Image Recognition?" https://medium.com/dataman-in-ai/module-6-image-recognition-for-insurance-claim-handling-part-i-a338d16c9de0 (accessed 13 June, 2021).

[10] J. Ni, Y. Chen, Y. Chen, J. Zhu, D. Ali, and W. Cao, "A Survey on Theories and Applications for Self-Driving Cars Based on Deep Learning Methods," *Applied Sciences*, vol. 10, no. 8, p. 2749, 2020.

[11] A. I. Maqueda, A. Loquercio, G. Gallego, N. García, and D. Scaramuzza, "Event-based vision meets deep learning on steering prediction for self-driving cars," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 5419-5427.

[12] S. Ramos, S. Gehrig, P. Pinggera, U. Franke, and C. Rother, "Detecting unexpected obstacles for self-driving cars: Fusing deep learning and geometric modeling," in *2017 IEEE Intelligent Vehicles Symposium (IV)*, 2017: IEEE, pp. 1025-1032.

[13] F. Wang, L. P. Casalino, and D. Khullar, "Deep learning in medicine—promise, progress, and challenges," *JAMA internal medicine*, vol. 179, no. 3, pp. 293-294, 2019.

[14] F. Piccialli, V. Di Somma, F. Giampaolo, S. Cuomo, and G. Fortino, "A survey on deep learning in medicine: Why, how and when?," *Information Fusion*, vol. 66, pp. 111-137, 2021.

[15] T. Ching *et al.*, "Opportunities and obstacles for deep learning in biology and medicine," *Journal of The Royal Society Interface*, vol. 15, no. 141, p. 20170387, 2018.

[16] G. Sreenu and M. S. Durai, "Intelligent video surveillance: a review through deep learning techniques for crowd analysis," *Journal of Big Data*, vol. 6, no. 1, pp. 1-27, 2019.

[17] Q. Fang *et al.*, "Detecting non-hardhat-use by a deep learning method from far-field surveillance videos," *Automation in Construction*, vol. 85, pp. 1-9, 2018.

[18] J. Chen, K. Li, Q. Deng, K. Li, and S. Y. Philip, "Distributed deep learning model for intelligent video surveillance systems with edge computing," *IEEE Transactions on Industrial Informatics*, 2019.

[19] L. Deng, "A tutorial survey of architectures, algorithms, and applications for deep learning," *APSIPA Transactions on Signal and Information Processing*, vol. 3, 2014.

[20] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097-1105.

[21] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[22] C. Szegedy *et al.*, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1-9.

[23] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770-778.

[24] A. G. Howard *et al.*, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[25] I. Shafkat. "Intuitively Understanding Convolutions for Deep Learning." https://towardsdatascience.com/intuitively-understanding-convolutions-for-deep-learning-1f6f42faee1 (accessed June 15, 2020).

[26] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International conference on machine learning*, 2015: PMLR, pp. 448-456.

[27] O. Russakovsky *et al.*, "Imagenet large scale visual recognition challenge," *International journal of computer vision*, vol. 115, no. 3, pp. 211-252, 2015.

[28] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size," *arXiv preprint arXiv:1602.07360*, 2016.

[29] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2818-2826.

[30] A. Khosla, N. Jayadevaprakash, B. Yao, and F.-F. Li, "Novel dataset for fine-grained image categorization: Stanford dogs," in *Proc. CVPR Workshop on Fine-Grained Visual Categorization (FGVC)*, 2011, vol. 2, no. 1: Citeseer.

[31] T. Weyand, I. Kostrikov, and J. Philbin, "Planet-photo geolocation with convolutional neural networks," in *European Conference on Computer Vision*, 2016: Springer, pp. 37-55.

[32] J. Hays and A. A. Efros, "Im2gps: estimating geographic information from a single image," in *2008 ieee conference on computer vision and pattern recognition*, 2008: IEEE, pp. 1-8.

[33] J. Hays and A. A. Efros, "Large-scale image geolocalization," in *Multimodal location estimation of videos and images*: Springer, 2015, pp. 41-62.

[34] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *arXiv preprint arXiv:1503.02531*, 2015.

[35] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," in *Advances in neural information processing systems*, 2015, pp. 91-99.

[36] W. Liu *et al.*, "Ssd: Single shot multibox detector," in *European conference on computer vision*, 2016: Springer, pp. 21-37.

[37] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 4510-4520.

[38] P. Marcelino. "Transfer learning from pre-trained models." https://towardsdatascience.com/transfer-learning-from-pre-trained-models-f2393f124751 (accessed June 15, 2020).

[39] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*, 2009: Ieee, pp. 248-255.

[40] D. Hughes and M. Salathé, "An open access repository of images on plant health to enable the development of mobile disease diagnostics," *arXiv preprint arXiv:1511.08060*, 2015.

[41] T.-Y. Lin *et al.*, "Microsoft coco: Common objects in context," in *European conference on computer vision*, 2014: Springer, pp. 740-755.

[42] H. Robbins and S. Monro, "A stochastic approximation method," *The annals of mathematical statistics*, pp. 400-407, 1951.

[43] G. Hinton, N. Srivastava, and K. Swersky, "Neural networks for machine learning lecture 6a overview of mini-batch gradient descent," *Cited on*, vol. 14, no. 8, 2012.

[44] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of machine learning research*, vol. 12, no. 7, 2011.

[45] M. D. Zeiler, "Adadelta: an adaptive learning rate method," *arXiv preprint arXiv:1212.5701*, 2012.

[46] S. J. Reddi, S. Kale, and S. Kumar, "On the convergence of adam and beyond," *arXiv preprint arXiv:1904.09237*, 2019.

[47] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[48] Z. Zhang, "Improved adam optimizer for deep neural networks," in *2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS)*, 2018: IEEE, pp. 1-2.

[49] S. R. Dubey, S. Chakraborty, S. K. Roy, S. Mukherjee, S. K. Singh, and B. B. Chaudhuri, "diffGrad: an optimization method for convolutional neural networks," *IEEE transactions on neural networks and learning systems*, vol. 31, no. 11, pp. 4500-4511, 2019.

[50] R. Collobert and J. Weston, "A unified architecture for natural language processing: Deep neural networks with multitask learning," in *Proceedings of the 25th international conference on Machine learning*, 2008, pp. 160-167.

[51] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber, "LSTM: A search space odyssey," *IEEE transactions on neural networks and learning systems*, vol. 28, no. 10, pp. 2222-2232, 2016.

[52] A. Agrawal. "Loss Functions and Optimization Algorithms. Demystified." https://medium.com/data-science-group-iitr/loss-functions-and-optimization-algorithms-demystified-bb92daff331c (accessed May 18, 2021).

[53] Z. Zhang and M. R. Sabuncu, "Generalized cross entropy loss for training deep neural networks with noisy labels," *arXiv preprint arXiv:1805.07836*, 2018.

[54] C. Shorten and T. M. Khoshgoftaar, "A survey on Image Data Augmentation for Deep Learning," *Journal of Big Data*, vol. 6, no. 1, p. 60, 2019/07/06 2019, doi: 10.1186/s40537-019-0197-0.

[55] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2014, pp. 580-587.

[56] J. R. Uijlings, K. E. Van De Sande, T. Gevers, and A. W. Smeulders, "Selective search for object recognition," *International journal of computer vision*, vol. 104, no. 2, pp. 154-171, 2013.

[57] R. Gandhi. "R-CNN, Fast R-CNN, Faster R-CNN, YOLO — Object Detection Algorithms." https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms-36d53571365e (accessed October 15, 2020).

[58] R. Girshick, "Fast r-cnn," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1440-1448.

[59] H. Rezatofighi, N. Tsoi, J. Gwak, A. Sadeghian, I. Reid, and S. Savarese, "Generalized intersection over union: A metric and a loss for bounding box regression," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 658-666.

[60] M. Cordts *et al.*, "The cityscapes dataset for semantic urban scene understanding," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 3213-3223.

[61] H. A. Alhaija, S. K. Mustikovela, L. Mescheder, A. Geiger, and C. Rother, "Augmented reality meets computer vision: Efficient data generation for urban driving scenes," *International Journal of Computer Vision*, vol. 126, no. 9, pp. 961-972, 2018.

[62] B. Zhou, H. Zhao, X. Puig, S. Fidler, A. Barriuso, and A. Torralba, "Scene parsing through ade20k dataset," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 633-641.

[63] M. Everingham, L. Van Gool, C. K. Williams, J. Winn, and A. Zisserman, "The pascal visual object classes (voc) challenge," *International journal of computer vision*, vol. 88, no. 2, pp. 303-338, 2010.

[64] M. Kristan *et al.*, "The visual object tracking vot2016 challenge results," in *ECCV workshop*, 2016, vol. 2, no. 6, p. 8.

[65] L. Leal-Taixé, A. Milan, I. Reid, S. Roth, and K. Schindler, "Motchallenge 2015: Towards a benchmark for multi-target tracking," *arXiv preprint arXiv:1504.01942*, 2015.

[66] Ľ. Ladický, P. Sturgess, K. Alahari, C. Russell, and P. H. S. Torr, "What, Where and How Many? Combining Object Detectors and CRFs," Berlin, Heidelberg, 2010: Springer Berlin Heidelberg, in Computer Vision – ECCV 2010, pp. 424-437.

[67] B. Romera-Paredes and P. H. S. Torr, "Recurrent Instance Segmentation," Cham, 2016: Springer International Publishing, in Computer Vision – ECCV 2016, pp. 312-329.

[68] Tensorflow. "Image segmentation." https://www.tensorflow.org/tutorials/images/segmentation (accessed October 15, 2020).

[69] Facebook. "Detectron." https://ai.facebook.com/tools/detectron/ (accessed October 14, 2020).

[70] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask r-cnn," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 2961-2969.

[71] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 3431-3440.

[72] C. Lim. "Mask R-CNN." https://www.slideshare.net/windmdk/mask-rcnn (accessed 27 June, 2021).

[73] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 1492-1500.

[74] T.-Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie, "Feature pyramid networks for object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 2117-2125.

[75] M. Jaderberg, K. Simonyan, A. Zisserman, and K. Kavukcuoglu, "Spatial transformer networks," *arXiv preprint arXiv:1506.02025*, 2015.

[76] J. Dai, K. He, and J. Sun, "Instance-aware semantic segmentation via multi-task network cascades," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 3150-3158.

[77] Y. Li, H. Qi, J. Dai, X. Ji, and Y. Wei, "Fully convolutional instance-aware semantic segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 2359-2367.

[78] B. McMahan and D. Ramage, "Federated learning: Collaborative machine learning without centralized training data," 2017. [Online]. Available: https://ai.googleblog.com/2017/04/federated-learning-collaborative.html.

[79] R. Shokri and V. Shmatikov, "Privacy-preserving deep learning," in *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*, 2015, pp. 1310-1321.

[80] altexsoft. "Comparing Machine Learning as a Service: Amazon, Microsoft Azure, Google Cloud AI, IBM Watson." https://www.altexsoft.com/blog/datascience/comparing-machine-learning-as-a-service-amazon-microsoft-azure-google-cloud-ai-ibm-watson (accessed December 08, 2020).

[81] J. Wang, B. Cao, P. Yu, L. Sun, W. Bao, and X. Zhu, "Deep learning towards mobile applications," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, 2018: IEEE, pp. 1385-1393.

[82] Apple. "Core ML." https://developer.apple.com/documentation/coreml (accessed 20 June, 2021).

[83] Apple. "Create ML." https://developer.apple.com/documentation/createml (accessed 20 June, 2021).

[84] FaceBook. "PyTorch Mobile." https://pytorch.org/mobile (accessed 20 June, 2021).

[85] Google. "Firebase ML | Machine learning for mobile developers." https://firebase.google.com/products/ml (accessed 20 June, 2021).

[86] Google. "TensorFlow Lite." https://www.tensorflow.org/lite (accessed June 6, 2021).

[87] Google. "TensorFlow Model Optimization Toolkit." https://github.com/tensorflow/model-optimization (accessed June 6, 2021).

[88] Google. "Model optimization." https://www.tensorflow.org/lite/performance/model_optimization (accessed June 6, 2021).

[89] Tensorflow. "Serving Models." https://www.tensorflow.org/tfx/guide/serving (accessed October 27, 2020).

[90] Y. Deng, "Deep learning on mobile devices: a review," in *Mobile Multimedia/Image Processing, Security, and Applications 2019*, 2019, vol. 10993: International Society for Optics and Photonics, p. 109930A.

[91] *Flutter*. (2020). Google. [Online]. Available: https://flutter.dev/

[92] K. Liakos, P. Busato, D. Moshou, S. Pearson, and D. Bochtis, "Machine Learning in Agriculture: A Review," *Sensors*, vol. 18, p. 2674, 08/14 2018, doi: 10.3390/s18082674.

[93] S. Baranwal, S. Khandelwal, and A. Arora, "Deep Learning Convolutional Neural Network for Apple Leaves Disease Detection," *SSRN Electronic Journal*, 01/01 2019, doi: https://doi.org/10.2139/ssrn.3351641.

[94] S. P. Mohanty, D. P. Hughes, and M. Salathé, "Using Deep Learning for Image-Based Plant Disease Detection," (in English), *Frontiers in Plant Science*, Methods vol. 7, no. 1419, 2016-September-22 2016, doi: https://doi.org/10.3389/fpls.2016.01419.

[95] S. Sladojevic, M. Arsenovic, A. Anderla, D. Culibrk, and D. Stefanovic, "Deep Neural Networks Based Recognition of Plant Diseases by Leaf Image Classification," *Computational Intelligence and Neuroscience*, vol. 2016, p. 3289801, 2016/06/22 2016, doi: 10.1155/2016/3289801.

[96] Y. Jia *et al.*, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*, 2014, pp. 675-678.

[97] P. Jiang, Y. Chen, B. Liu, D. He, and C. Liang, "Real-Time Detection of Apple Leaf Diseases Using Deep Learning Approach Based on Improved Convolutional Neural Networks," *IEEE Access*, vol. 7, pp. 59069-59080, 2019, doi: 10.1109/ACCESS.2019.2914929.

[98] R. J. Wieringa, *Design science methodology for information systems and software engineering*. Springer, 2014.

[99] Google. "TensorFlow Lite converter." https://www.tensorflow.org/lite/convert (accessed June 6, 2021).

[100] A. Dutta and A. Zisserman, "The VIA annotation software for images, audio and video," in *Proceedings of the 27th ACM International Conference on Multimedia*, 2019, pp. 2276-2279.

[101] W. Abdulla, "Mask r-cnn for object detection and instance segmentation on keras and tensorflow," 2017.

APPENDIX A: MOBILENETV2_1.00_224 BASE MODEL SUMMARY

Layer (type)	**Output Shape**	**Param**	**Connected to**
input_1 (InputLayer)	[(None, 224, 224, 3)	0	
Conv1 (Conv2D)	(None, 112, 112, 32)	864	input_1[0][0]
bn_Conv1 (BatchNormalization)	(None, 112, 112, 32)	128	Conv1[0][0]
Conv1_relu (ReLU)	(None, 112, 112, 32)	0	bn_Conv1[0][0]
expanded_conv_depthwise (Depthw	(None, 112, 112, 32)	288	Conv1_relu[0][0]
expanded_conv_depthwise_BN (Bat	(None, 112, 112, 32)	128	expanded_conv_depthwise[0][0]
expanded_conv_depthwise_relu (R	(None, 112, 112, 32)	0	expanded_conv_depthwise_BN[0][0]
expanded_conv_project (Conv2D)	(None, 112, 112, 16)	512	expanded_conv_depthwise_relu[0][0]
expanded_conv_project_BN (Batch	(None, 112, 112, 16)	64	expanded_conv_project[0][0]

block_1_expand (Conv2D)	(None, 112, 112, 96)	1536	expanded_conv_project_BN[0][0]
block_1_expand_BN (BatchNormali	(None, 112, 112, 96)	384	block_1_expand[0][0]
block_1_expand_relu (ReLU)	(None, 112, 112, 96)	0	block_1_expand_BN[0][0]
block_1_pad (ZeroPadding2D)	(None, 113, 113, 96)	0	block_1_expand_relu[0][0]
block_1_depthwise (DepthwiseCon	(None, 56, 56, 96)	864	block_1_pad[0][0]
block_1_depthwise_BN (BatchNorm	(None, 56, 56, 96)	384	block_1_depthwise[0][0]
block_1_depthwise_relu (ReLU)	(None, 56, 56, 96)	0	block_1_depthwise_BN[0][0]
block_1_project (Conv2D)	(None, 56, 56, 24)	2304	block_1_depthwise_relu[0][0]
block_1_project_BN (BatchNormal	(None, 56, 56, 24)	96	block_1_project[0][0]
block_2_expand (Conv2D)	(None, 56, 56, 144)	3456	block_1_project_BN[0][0]
block_2_expand_BN (BatchNormali	(None, 56, 56, 144)	576	block_2_expand[0][0]
block_2_expand_relu (ReLU)	(None, 56, 56, 144)	0	block_2_expand_BN[0][0]
block_2_depthwise (DepthwiseCon	(None, 56, 56, 144)	1296	block_2_expand_relu[0][0]
block_2_depthwise_BN (BatchNorm	(None, 56, 56, 144)	576	block_2_depthwise[0][0]
block_2_depthwise_relu (ReLU)	(None, 56, 56, 144)	0	block_2_depthwise_BN[0][0]
block_2_project (Conv2D)	(None, 56, 56, 24)	3456	block_2_depthwise_relu[0][0]
block_2_project_BN (BatchNormal	(None, 56, 56, 24)	96	block_2_project[0][0]
block_2_add (Add)	(None, 56, 56, 24)	0	block_1_project_BN[0][0] block_2_project_BN[0][0]
block_3_expand (Conv2D)	(None, 56, 56, 144)	3456	block_2_add[0][0]
block_3_expand_BN (BatchNormali	(None, 56, 56, 144)	576	block_3_expand[0][0]
block_3_expand_relu (ReLU)	(None, 56, 56, 144)	0	block_3_expand_BN[0][0]
block_3_pad (ZeroPadding2D)	(None, 57, 57, 144)	0	block_3_expand_relu[0][0]
block_3_depthwise (DepthwiseCon	(None, 28, 28, 144)	1296	block_3_pad[0][0]
block_3_depthwise_BN (BatchNorm	(None, 28, 28, 144)	576	block_3_depthwise[0][0]

block_3_depthwise_relu (ReLU)	(None, 28, 28, 144)	0	block_3_depthwise_BN[0][0]
block_3_project (Conv2D)	(None, 28, 28, 32)	4608	block_3_depthwise_relu[0][0]
block_3_project_BN (BatchNormal	(None, 28, 28, 32)	128	block_3_project[0][0]
block_4_expand (Conv2D)	(None, 28, 28, 192)	6144	block_3_project_BN[0][0]
block_4_expand_BN (BatchNormali	(None, 28, 28, 192)	768	block_4_expand[0][0]
block_4_expand_relu (ReLU)	(None, 28, 28, 192)	0	block_4_expand_BN[0][0]
block_4_depthwise (DepthwiseCon	(None, 28, 28, 192)	1728	block_4_expand_relu[0][0]
block_4_depthwise_BN (BatchNorm	(None, 28, 28, 192)	768	block_4_depthwise[0][0]
block_4_depthwise_relu (ReLU)	(None, 28, 28, 192)	0	block_4_depthwise_BN[0][0]
block_4_project (Conv2D)	(None, 28, 28, 32)	6144	block_4_depthwise_relu[0][0]
block_4_project_BN (BatchNormal	(None, 28, 28, 32)	128	block_4_project[0][0]
block_4_add (Add)	(None, 28, 28, 32)	0	block_3_project_BN[0][0] block_4_project_BN[0][0]
block_5_expand (Conv2D)	(None, 28, 28, 192)	6144	block_4_add[0][0]
block_5_expand_BN (BatchNormali	(None, 28, 28, 192)	768	block_5_expand[0][0]
block_5_expand_relu (ReLU)	(None, 28, 28, 192)	0	block_5_expand_BN[0][0]
block_5_depthwise (DepthwiseCon	(None, 28, 28, 192)	1728	block_5_expand_relu[0][0]
block_5_depthwise_BN (BatchNorm	(None, 28, 28, 192)	768	block_5_depthwise[0][0]
block_5_depthwise_relu (ReLU)	(None, 28, 28, 192)	0	block_5_depthwise_BN[0][0]
block_5_project (Conv2D)	(None, 28, 28, 32)	6144	block_5_depthwise_relu[0][0]
block_5_project_BN (BatchNormal	(None, 28, 28, 32)	128	block_5_project[0][0]
block_5_add (Add)	(None, 28, 28, 32)	0	block_4_add[0][0] block_5_project_BN[0][0]
block_6_expand (Conv2D)	(None, 28, 28, 192)	6144	block_5_add[0][0]
block_6_expand_BN (BatchNormali	(None, 28, 28, 192)	768	block_6_expand[0][0]
block_6_expand_relu (ReLU)	(None, 28, 28, 192)	0	block_6_expand_BN[0][0]

block_6_pad (ZeroPadding2D)	(None, 29, 29, 192)	0	block_6_expand_relu[0][0]
block_6_depthwise (DepthwiseCon	(None, 14, 14, 192)	1728	block_6_pad[0][0]
block_6_depthwise_BN (BatchNorm	(None, 14, 14, 192)	768	block_6_depthwise[0][0]
block_6_depthwise_relu (ReLU)	(None, 14, 14, 192)	0	block_6_depthwise_BN[0][0]
block_6_project (Conv2D)	(None, 14, 14, 64)	12288	block_6_depthwise_relu[0][0]
block_6_project_BN (BatchNormal	(None, 14, 14, 64)	256	block_6_project[0][0]
block_7_expand (Conv2D)	(None, 14, 14, 384)	24576	block_6_project_BN[0][0]
block_7_expand_BN (BatchNormali	(None, 14, 14, 384)	1536	block_7_expand[0][0]
block_7_expand_relu (ReLU)	(None, 14, 14, 384)	0	block_7_expand_BN[0][0]
block_7_depthwise (DepthwiseCon	(None, 14, 14, 384)	3456	block_7_expand_relu[0][0]
block_7_depthwise_BN (BatchNorm	(None, 14, 14, 384)	1536	block_7_depthwise[0][0]
block_7_depthwise_relu (ReLU)	(None, 14, 14, 384)	0	block_7_depthwise_BN[0][0]
block_7_project (Conv2D)	(None, 14, 14, 64)	24576	block_7_depthwise_relu[0][0]
block_7_project_BN (BatchNormal	(None, 14, 14, 64)	256	block_7_project[0][0]
block_7_add (Add)	(None, 14, 14, 64)	0	block_6_project_BN[0][0] block_7_project_BN[0][0]
block_8_expand (Conv2D)	(None, 14, 14, 384)	24576	block_7_add[0][0]
block_8_expand_BN (BatchNormali	(None, 14, 14, 384)	1536	block_8_expand[0][0]
block_8_expand_relu (ReLU)	(None, 14, 14, 384)	0	block_8_expand_BN[0][0]
block_8_depthwise (DepthwiseCon	(None, 14, 14, 384)	3456	block_8_expand_relu[0][0]
block_8_depthwise_BN (BatchNorm	(None, 14, 14, 384)	1536	block_8_depthwise[0][0]
block_8_depthwise_relu (ReLU)	(None, 14, 14, 384)	0	block_8_depthwise_BN[0][0]
block_8_project (Conv2D)	(None, 14, 14, 64)	24576	block_8_depthwise_relu[0][0]
block_8_project_BN (BatchNormal	(None, 14, 14, 64)	256	block_8_project[0][0]
block_8_add (Add)	(None, 14, 14, 64)	0	block_7_add[0][0]
			block_8_project_BN[0][0]

block_9_expand (Conv2D)	(None, 14, 14, 384)	24576	block_8_add[0][0]
block_9_expand_BN (BatchNormali	(None, 14, 14, 384)	1536	block_9_expand[0][0]
block_9_expand_relu (ReLU)	(None, 14, 14, 384)	0	block_9_expand_BN[0][0]
block_9_depthwise (DepthwiseCon	(None, 14, 14, 384)	3456	block_9_expand_relu[0][0]
block_9_depthwise_BN (BatchNorm	(None, 14, 14, 384)	1536	block_9_depthwise[0][0]
block_9_depthwise_relu (ReLU)	(None, 14, 14, 384)	0	block_9_depthwise_BN[0][0]
block_9_project (Conv2D)	(None, 14, 14, 64)	24576	block_9_depthwise_relu[0][0]
block_9_project_BN (BatchNormal	(None, 14, 14, 64)	256	block_9_project[0][0]
block_9_add (Add)	(None, 14, 14, 64)	0	block_8_add[0][0]
			block_9_project_BN[0][0]
block_10_expand (Conv2D)	(None, 14, 14, 384)	24576	block_9_add[0][0]
block_10_expand_BN (BatchNormal	(None, 14, 14, 384)	1536	block_10_expand[0][0]
block_10_expand_relu (ReLU)	(None, 14, 14, 384)	0	block_10_expand_BN[0][0]
block_10_depthwise (DepthwiseCo	(None, 14, 14, 384)	3456	block_10_expand_relu[0][0]
block_10_depthwise_BN (BatchNor	(None, 14, 14, 384)	1536	block_10_depthwise[0][0]
block_10_depthwise_relu (ReLU)	(None, 14, 14, 384)	0	block_10_depthwise_BN[0][0]
block_10_project (Conv2D)	(None, 14, 14, 96)	36864	block_10_depthwise_relu[0][0]
block_10_project_BN (BatchNorma	(None, 14, 14, 96)	384	block_10_project[0][0]
block_11_expand (Conv2D)	(None, 14, 14, 576)	55296	block_10_project_BN[0][0]
block_11_expand_BN (BatchNormal	(None, 14, 14, 576)	2304	block_11_expand[0][0]
block_11_expand_relu (ReLU)	(None, 14, 14, 576)	0	block_11_expand_BN[0][0]
block_11_depthwise (DepthwiseCo	(None, 14, 14, 576)	5184	block_11_expand_relu[0][0]
block_11_depthwise_BN (BatchNor	(None, 14, 14, 576)	2304	block_11_depthwise[0][0]
block_11_depthwise_relu (ReLU)	(None, 14, 14, 576)	0	block_11_depthwise_BN[0][0]
block_11_project (Conv2D)	(None, 14, 14, 96)	55296	block_11_depthwise_relu[0][0]

block_11_project_BN (BatchNorma	(None, 14, 14, 96)	384	block_11_project[0][0]
block_11_add (Add)	(None, 14, 14, 96)	0	block_10_project_BN[0][0]
			block_11_project_BN[0][0]
block_12_expand (Conv2D)	(None, 14, 14, 576)	55296	block_11_add[0][0]
block_12_expand_BN (BatchNormal	(None, 14, 14, 576)	2304	block_12_expand[0][0]
block_12_expand_relu (ReLU)	(None, 14, 14, 576)	0	block_12_expand_BN[0][0]
block_12_depthwise (DepthwiseCo	(None, 14, 14, 576)	5184	block_12_expand_relu[0][0]
block_12_depthwise_BN (BatchNor	(None, 14, 14, 576)	2304	block_12_depthwise[0][0]
block_12_depthwise_relu (ReLU)	(None, 14, 14, 576)	0	block_12_depthwise_BN[0][0]
block_12_project (Conv2D)	(None, 14, 14, 96)	55296	block_12_depthwise_relu[0][0]
block_12_project_BN (BatchNorma	(None, 14, 14, 96)	384	block_12_project[0][0]
block_12_add (Add)	(None, 14, 14, 96)	0	block_11_add[0][0]
			block_12_project_BN[0][0]
block_13_expand (Conv2D)	(None, 14, 14, 576)	55296	block_12_add[0][0]
block_13_expand_BN (BatchNormal	(None, 14, 14, 576)	2304	block_13_expand[0][0]
block_13_expand_relu (ReLU)	(None, 14, 14, 576)	0	block_13_expand_BN[0][0]
block_13_pad (ZeroPadding2D)	(None, 15, 15, 576)	0	block_13_expand_relu[0][0]
block_13_depthwise (DepthwiseCo	(None, 7, 7, 576)	5184	block_13_pad[0][0]
block_13_depthwise_BN (BatchNor	(None, 7, 7, 576)	2304	block_13_depthwise[0][0]
block_13_depthwise_relu (ReLU)	(None, 7, 7, 576)	0	block_13_depthwise_BN[0][0]
block_13_project (Conv2D)	(None, 7, 7, 160)	92160	block_13_depthwise_relu[0][0]
block_13_project_BN (BatchNorma	(None, 7, 7, 160)	640	block_13_project[0][0]
block_14_expand (Conv2D)	(None, 7, 7, 960)	153600	block_13_project_BN[0][0]
block_14_expand_BN (BatchNormal	(None, 7, 7, 960)	3840	block_14_expand[0][0]
block_14_expand_relu (ReLU)	(None, 7, 7, 960)	0	block_14_expand_BN[0][0]
block_14_depthwise (DepthwiseCo	(None, 7, 7, 960)	8640	block_14_expand_relu[0][0]

block_14_depthwise_BN (BatchNor	(None, 7, 7, 960)	3840	block_14_depthwise[0][0]
block_14_depthwise_relu (ReLU)	(None, 7, 7, 960)	0	block_14_depthwise_BN[0][0]
block_14_project (Conv2D)	(None, 7, 7, 160)	153600	block_14_depthwise_relu[0][0]
block_14_project_BN (BatchNorma	(None, 7, 7, 160)	640	block_14_project[0][0]
block_14_add (Add)	(None, 7, 7, 160)	0	block_13_project_BN[0][0]
			block_14_project_BN[0][0]
block_15_expand (Conv2D)	(None, 7, 7, 960)	153600	block_14_add[0][0]
block_15_expand_BN (BatchNormal	(None, 7, 7, 960)	3840	block_15_expand[0][0]
block_15_expand_relu (ReLU)	(None, 7, 7, 960)	0	block_15_expand_BN[0][0]
block_15_depthwise (DepthwiseCo	(None, 7, 7, 960)	8640	block_15_expand_relu[0][0]
block_15_depthwise_BN (BatchNor	(None, 7, 7, 960)	3840	block_15_depthwise[0][0]
block_15_depthwise_relu (ReLU)	(None, 7, 7, 960)	0	block_15_depthwise_BN[0][0]
block_15_project (Conv2D)	(None, 7, 7, 160)	153600	block_15_depthwise_relu[0][0]
block_15_project_BN (BatchNorma	(None, 7, 7, 160)	640	block_15_project[0][0]
block_15_add (Add)	(None, 7, 7, 160)	0	block_14_add[0][0]
			block_15_project_BN[0][0]
block_16_expand (Conv2D)	(None, 7, 7, 960)	153600	block_15_add[0][0]
block_16_expand_BN (BatchNormal	(None, 7, 7, 960)	3840	block_16_expand[0][0]
block_16_expand_relu (ReLU)	(None, 7, 7, 960)	0	block_16_expand_BN[0][0]
block_16_depthwise (DepthwiseCo	(None, 7, 7, 960)	8640	block_16_expand_relu[0][0]
block_16_depthwise_BN (BatchNor	(None, 7, 7, 960)	3840	block_16_depthwise[0][0]
block_16_depthwise_relu (ReLU)	(None, 7, 7, 960)	0	block_16_depthwise_BN[0][0]
block_16_project (Conv2D)	(None, 7, 7, 320)	307200	block_16_depthwise_relu[0][0]
block_16_project_BN (BatchNorma	(None, 7, 7, 320)	1280	block_16_project[0][0]
Conv_1 (Conv2D)	(None, 7, 7, 1280)	409600	block_16_project_BN[0][0]
Conv_1_bn (BatchNormalization)	(None, 7, 7, 1280)	5120	Conv_1[0][0]
out_relu (ReLU)	(None, 7, 7, 1280)	0	Conv_1_bn[0][0]

```
Total params: 2,257,984
Trainable params: 0
Non-trainable params: 2,257,984
```