Plant Seedlings Classification

Cheuk Hang Ng (a1821087)

University of Adelaide, a1821087@student.adelaide.edu.au

This project aimed to build a prediction model which is capable to classify the species of a plant from a photo. Near 5,000 images of plants belonging to 12 species were provided as training data. The images were preprocessed with Gaussian Blur and color mask before feeding into the models. Six Convolutional Neural Network models were selected to be trained and performed the task. A model that was build from scratch and five pre-trained models, namely the VGG16, ResNet50, Inception-v3, Xception and EfficientNetB0 were trained with un-augmented images and augmented images. Pre-trained models were fine-tuned after trained with augmented images. The EfficientNetB0 performed as expected while other models encountered over-fitting problems. Continuous parameters tuning is needed to improve the models.

Computing methodologies • Machine learning • Learning paradigms • Unsupervised learning

Additional Keywords and Phrases: Image classification, Convolutional Neural Network

1 INTRODUCTION

The project aims to create a classifier that is capable of determining a plant's species from a photo. The dataset contains images of plants belonging to 12 species at varied growth stages. This project started with exploratory data analysis, followed by data preprocessing, and then prediction models were built. Convolutional Neural Network (CNN) is an excellent deep learning algorithm for analysis of visual images. 6 CNN models were built in this project, in which 1 of the 6 models was built from scratch, and the other 5 were pre-trained models, which is, VGG16, ResNet50, Inception-v3, Xception and EfficientNetB0. Each of the models was trained with un-augmented images and augmented images successively, and the pre-trained models were further fine-tuned after trained with augmented images. Loss and accuracy were chosen to be the metrics for evaluating the models, and a confusion matrix was plotted to visualize the performance of each model. Fine-tined EfficientNetB0 was used to predict the species of plants and the result would be submitted to Kaggle.

2 METHODOLOGY

2.1 Exploratory Data Analysis

The dataset was downloaded from Kaggle and the structure of the folders were examined first. All image files in the "train" folder were then read, resized and stored as a list. The names of the folders containing the image files were stored in another list. The images were then transformed into numpy arrays object, and the folder names list was converted to pandas dataframe object. The dimensions of the dataset were shown to see if all the files were read and resized correctly. In addition, the first 12 images in the list were visualized as samples of input files. The numbers of files contained in each folder against the folders name, equivalent to the name of the 12 species were plotted as bar chart.

2.2 Data Preprocessing

Gaussian Blur and HSV color space masks were applied to the images to isolate the plants in the frame and the masked images were stored in a new list. The new image list was also transformed to numpy arrays object afterwards. Examples of masked images were then displayed. Next, the dataset was split into train-test set.

2.2.1 Gaussian Blur

A low-pass filter that convolves image with Gaussian function. Value of each pixel will be replaced by the weighted average of the color values of the pixels in the kernel. `GaussianBlur()` from OpenCV was called to smooth the image. Kernel size was chosen to be 5x5.

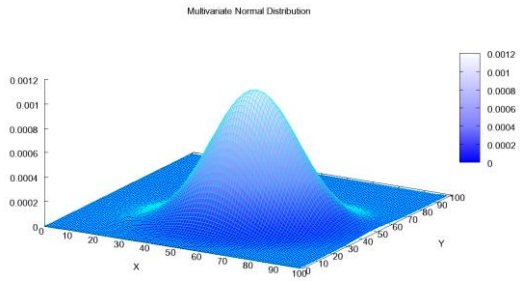

Figure 1: A 2-dimensional Gaussian function, via Wikimedia Commons. (https://t.ly/GeQn)

2.2.2 HSV Color Space

A certain range of color could be indicated easily in HSV (hue, saturation, value) color space. As the images were read with `cv.imread()` which decoded the images in BGR (blue, green, red) order, `cvtColor()` from OpenCV was used to convert the images to HSV representation.

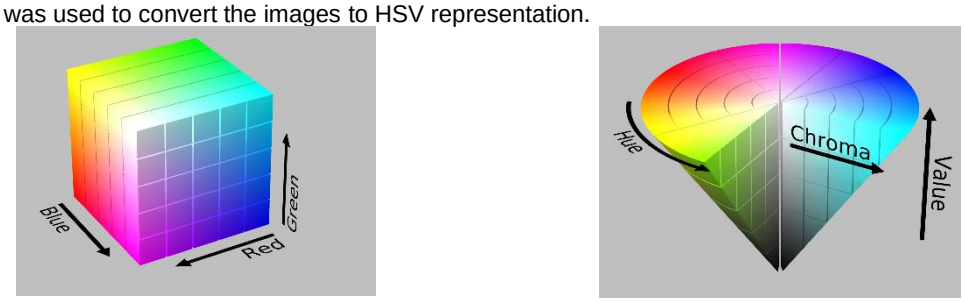

Figure 2: Left: RGB color space, via Wikimedia Commons. (https://t.ly/ee4I); Right: HSV color space, via Wikimedia Commons. (https://t.ly/Afdr)

2.2.3 Masking

To remove everything other than the plants in the frame, the range of green color was chosen to be (25, 30, 40) to (85, 255, 255) in the terms of HSV. `cv2.inRange()` was used to detect the color in the above range in the image, and `cv2.getStructuringElement(cv2.MORPH_ELLIPSE())` with parameter `(11,11)` was used to detect leaf-like shape. The above two functions were applied to the images to remove everything other than the plants.

2.2.4 Train-test Split

The masked images were split into train set and test set to train the models. `train_test_split()` from Sklearn was used to perform the task. Test set size was selected to be 0.15 of the original dataset size, and random state was fixed at 42.

2.3 Convolutional Neural Network (CNN)

Convolutional Neural Network (CNN) is a deep learning algorithm which is able to learn the importance of various objects and differentiating object from the others. There are three main types of layers, namely, Convolutional layer, Pooling layer and Fully-connected layer.

2.3.1 Convolutional Layer

The convolutional layer is made up of an input, a kernel and the convolved feature. The kernel acts as a feature detector which scans the entire image and maps the feature in the input to the convolved feature. An activation function is then applied to the transform the feature map (a typical choice of activation function is the rectified linear activation function (ReLU), which introduce non-linearity to the model). It is common to see a convolutional layer is followed by another convolutional layer. Consecutive convolutional layers create a hierarchical structure to the model, where the later layers extract higher-level patterns.

2.3.2 Pooling Layer

The pooling layer reduces the dimension and the number of parameters in the input. Similar to the convolutional layer, it consists of a kernel that sweeps across the input which applies an aggregation function to populate the output array. A common choice of pooling method is Max pooling, where the maximum value within the kernel is selected to map to the output array.

2.3.3 Fully-connected Layer

The fully-connected layer is responsible for performing classifications based on the features extracted in former layers. The softmax activation function is usually used to classify the input image from multiple classes.

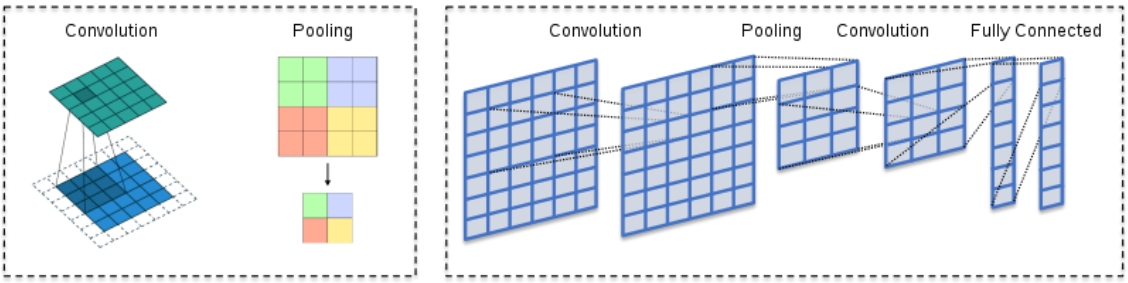

Figure 3: Example of the structure of a CNN, via Wikimedia Commons. (https://t.ly/XbRN)

2.4 The Models

6 CNN models were trained with the dataset. All functions and models used in this part are from the Keras library unless specified otherwise. In which the first one is built from scratch and the others were pre-trained VGG16, ResNet50, Inception-v3, Xception and EfficientNetB0 models. The 5 pre-trained models are the

competitors and even winners in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC), which requires the models to classify 1000 classes of objects. As the models are already trained with the ImageNet dataset which contains millions of images, they can efficiently learn and extract features from new input of images. To increase the performance of the pre-trained models on this project's dataset, fine-tuning is performed to allow the layers in the pre-trained models learn from the dataset. Two `ImageDataGenerator()` objects were declared before each model, for un-augmented input and augmented input. For instance, the generators for VGG16 are declared as follow.

```
datagen = ImageDataGenerator(preprocessing_function = vgg_preprocess_input)
aug_datagen = ImageDataGenerator(rotation_range = 90, width_shift_range = 0.2, height_shift_range = 0.2, zoom_range = 0.2, horizontal_flip = True, vertical_flip = True, preprocessing_function = vgg_preprocess_input)
```

The parameter `preprocessing_function` for each pre-trained models were assigned with the corresponding `preprocess_input` from the library. In the from scratch model, it was replaced with `rescale = 1./255` to rescale pixel values from [0, 255] to [0, 1].

2.4.1 A CNN Model from Scratch

A CNN model is built from scratch using `Model.Sequential()`. Three convolutional blocks were built in a similar structure. There were 2 convolutional layers followed by batch normalization in each block, and before sending to the next convolutional block, max pooling and drop out were applied for regularization. It was then connected to a `Flatten()` layer, followed by two set of `Dense()-BatchNormalization()-Dropout()` layers and the classification layer with softmax as the activation function. The model summary is listed below.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 76, 76, 64)        4864
_________________________________________________________________
batch_normalization (BatchNo (None, 76, 76, 64)        256
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 72, 72, 64)        102464
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 36, 36, 64)        0
_________________________________________________________________
batch_normalization_1 (Batch (None, 36, 36, 64)        256
_________________________________________________________________
dropout (Dropout)            (None, 36, 36, 64)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 32, 32, 128)       204928
_________________________________________________________________
batch_normalization_2 (Batch (None, 32, 32, 128)       512
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 28, 28, 128)       409728
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 14, 14, 128)       0
_________________________________________________________________
batch_normalization_3 (Batch (None, 14, 14, 128)       512
_________________________________________________________________
dropout_1 (Dropout)          (None, 14, 14, 128)       0
_________________________________________________________________
```

```
conv2d_4 (Conv2D)            (None, 10, 10, 256)       819456
_________________________________________________________________
batch_normalization_4 (Batch (None, 10, 10, 256)       1024
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 6, 6, 256)         1638656
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 3, 3, 256)         0
_________________________________________________________________
batch_normalization_5 (Batch (None, 3, 3, 256)         1024
_________________________________________________________________
dropout_2 (Dropout)          (None, 3, 3, 256)         0
_________________________________________________________________
flatten (Flatten)            (None, 2304)              0
_________________________________________________________________
dense (Dense)                (None, 256)               590080
_________________________________________________________________
batch_normalization_6 (Batch (None, 256)               1024
_________________________________________________________________
dropout_3 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_1 (Dense)              (None, 256)               65792
_________________________________________________________________
batch_normalization_7 (Batch (None, 256)               1024
_________________________________________________________________
dropout_4 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_2 (Dense)              (None, 12)                3084
=================================================================
Total params: 3,844,684
Trainable params: 3,841,868
Non-trainable params: 2,816
_________________________________________________________________
```

2.4.2 VGG16

The model is proposed by K. Simonyan and A. Zisserman [1] and submitted to ILSVRC in 2014. It consists of 5 blocks of varied size convolutional stacks with 3 fully-connected layers, with ReLU activation in each layer.

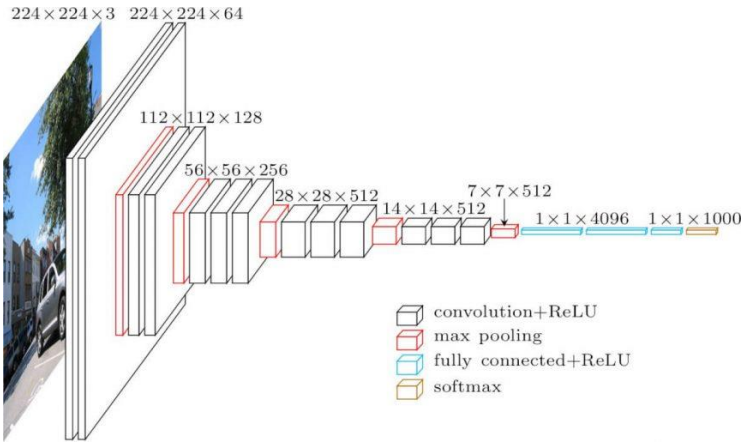

Figure 4: Architecture of the VGG16 model, via Neurohive. (https://t.ly/lfcg)

2.4.3 ResNet50

Resnet50 is a variant of the ResNet model which has 48 convolutional layers with 1 max pooling layer and an average pool layer [2]. The authors of the ResNet model introduced shortcut connections that perform identity mappings which let the layer fit a residual mapping. By the above modification, the performance of the model is raised without additional parameters added.

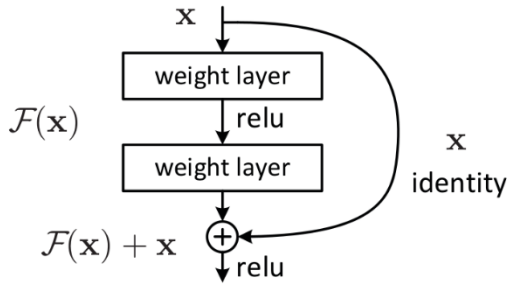

Figure 5: Shortcut connection for identity mapping [2]

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Figure 6: Architectures of ResNet variants [2]

2.4.4 InceptionV3

InceptionV3 is an improved version of the previous Inception model by reducing the demand of computational power mainly by factorized convolutions and parallelized computations [3]. The highlights of the InceptionV3 are as follows. The model adapted factorized convolutions, smaller convolutions filter sizes and asymmetric convolutions. The above modifications enhance the efficiency of the model while maintaining the performance. In addition, auxiliary classifier and grid size reduction are used during training to act as regularizer and to further reduce computational costs.

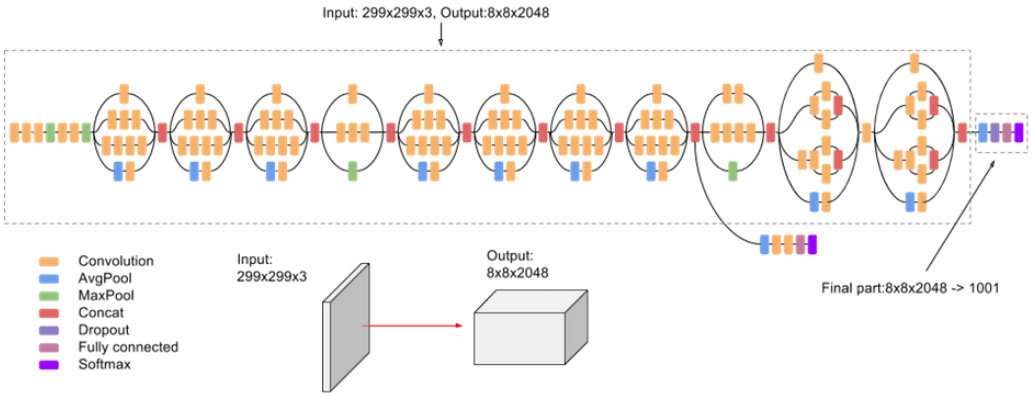

Figure 7: Architectures of InceptionV3, via Paperswithcode (https://t.ly/zcCE)

2.4.5 Xception

Xception is a model proposed after the InceptionV3. It adapts modified depthwise separable convolution, which is a pointwise convolution followed by depthwise separable convolution. It also removes intermediate ReLU non-linearity activation [4].

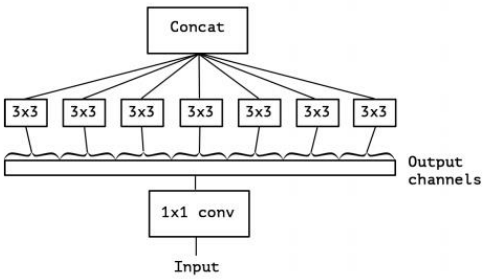

Figure 8: “Extreme” version of Inception module [4]

2.4.6 EfficientNet

EfficientNet is a computational cost efficient CNN model. Instead of arbitrarily scaling model dimensions, such as width, depth and resolution, the EfficientNet family uniformly scales each dimension with a fixed set of scaling coefficients [5]. The compound scaling method is inspired by that if the input image is larger, then the model needs more layers to increase the receptive field and need more channels to learn more detailed patterns.

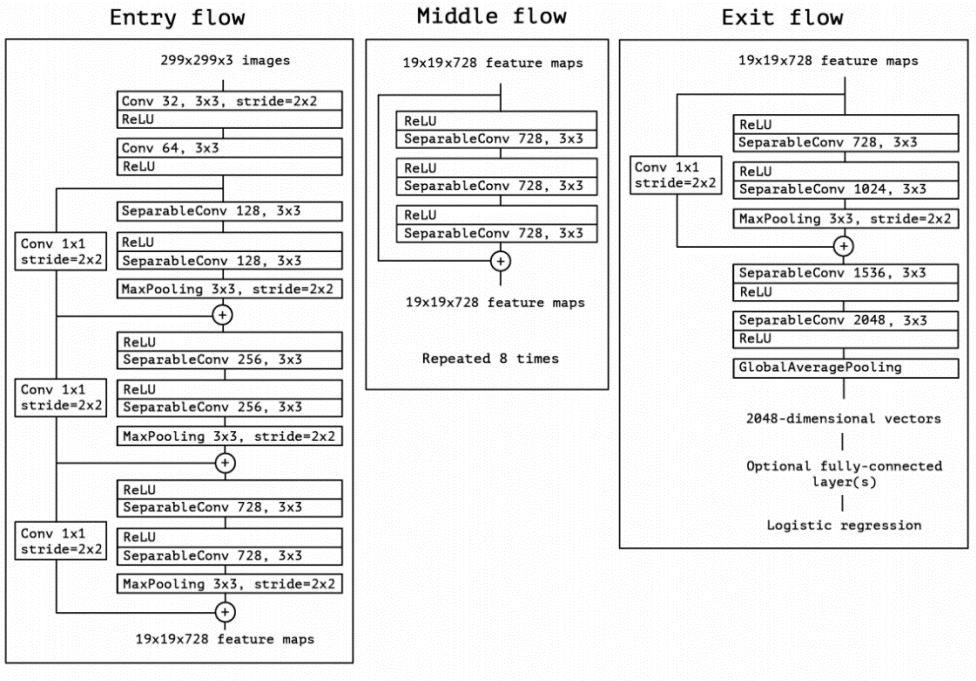

Figure 9: Overall architecture of an Xception model [4]

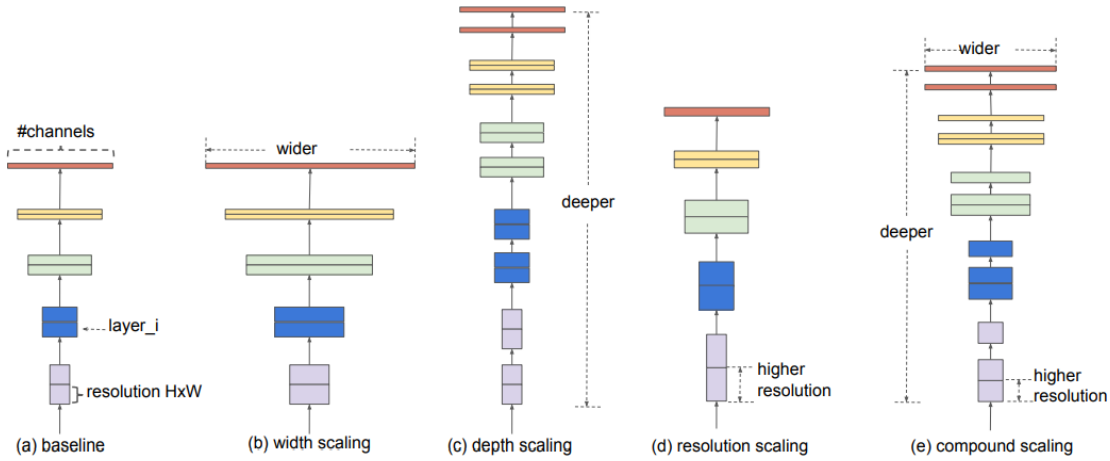

Figure 10: Different types of model scaling. EfficientNet is using the compound scaling on the right [5]

2.5 Metrics

2.5.1 Loss and Accuracy

Loss and accuracy were chosen to be the metrics for evaluating the models. Loss function chose to be `categorical_crossentropy()` as `LabelEncoder()` from Sklearn was used to encode the species. Accuracy

is calculated by how often predictions equal the classes. The metrics from train set and test set were plotted on the same graph to visualize the performance.

2.5.2 Confusion Matrix

The true classes of the test set were plotted against the predicted classes in a confusion matrix style, which visualize the number of correct and wrong predictions for each class.

3 EXPERIMENTS AND RESULTS

3.1 Exploratory Data Analysis

The size of image input was decided to be 80x80. As 80x80 is a valid input size for all the models used and it is suitable for running image classification on personal computer in terms of computational power. The shape of image list was (4750, 80, 80, 3) implying that there were 4,750 image files in the list with size 80x80 and was in 3-channels representation, which matched the number of image files in the train folder and the images were resized properly. The label list was (4750, 1) which was as expected. The first 12 images as the samples of input images were shown below. The number of files with each label (i.e., counts of files under each sub-folder in the dataset) was also shown below.

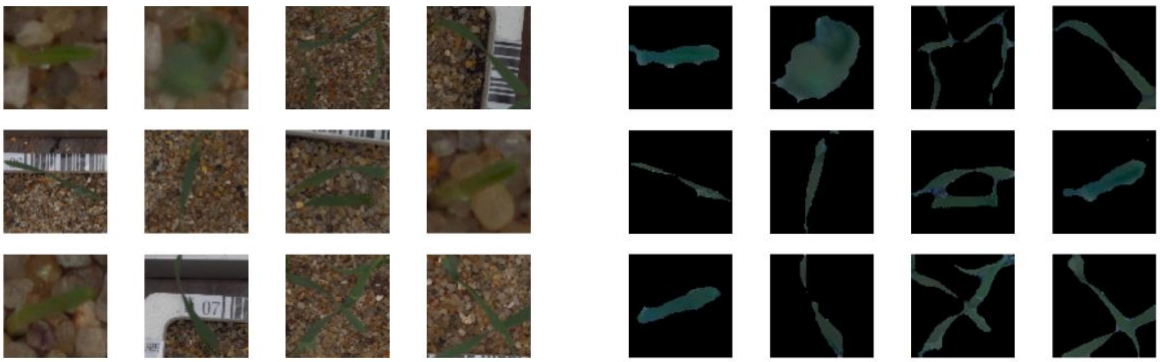

Figure 11: Left: samples of input images; Right: masked input images

3.2 Data Preprocessing

All images in the list were masked as described in section 2.2. Samples of masked input images were shown above. Number of images in train set and test set were 4,037 and 713 respectively.

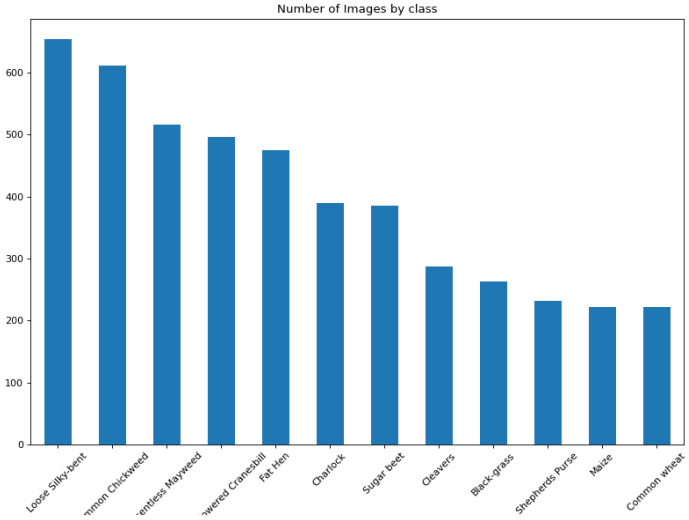

Figure 10: Number of files in each sub-folder

3.3 The models

3.3.1 From scratch CNN model

The metrics plotted for the from scratch model with un-augmented training data and augmented data were shown below.

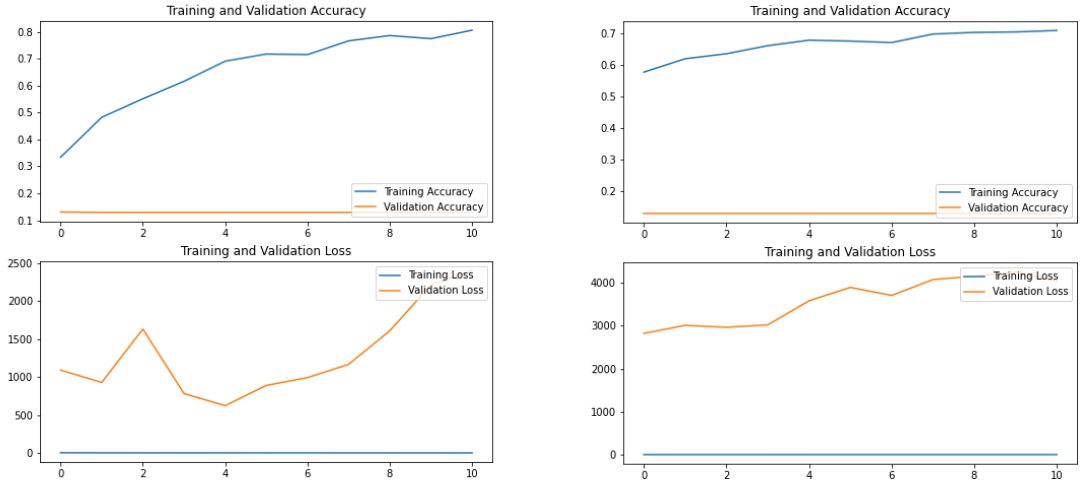

Figure 11: Metrics plotted for from scratch model with normal data (left) and augmented data (right).

3.3.2 Pre-trained Models

The metrics plotted for pre-trained models with un-augmented training data, augmented data and augmented data with fine-tuning were shown below in the order of VGG16, ResNet50, InceptionV3, Xception and EfficientNetB0. Fine-tunings were performed as unfreezing (i.e., set as trainable) shallower layers or blocks in the first 4 pre-trained models, and unfreezing all layers in the EfficientNetB0 model.

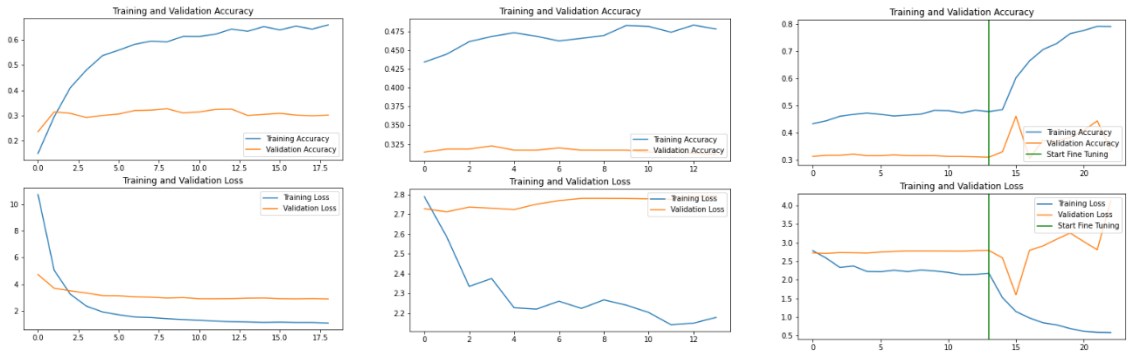

Figure 12: Metrics plotted for VGG16 with normal data (left), augmented data (middle) and with fine-tuning (right).

Figure 13: Metrics plotted for ResNet50 with normal data (left), augmented data (middle) and with fine-tuning (right).

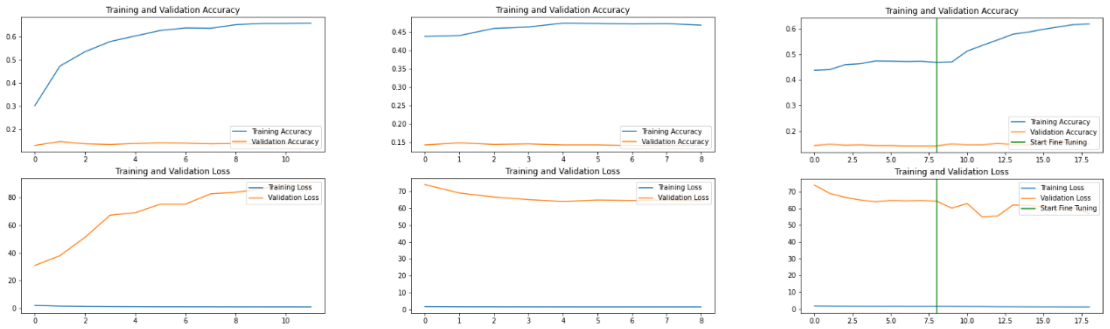

Figure 14: Metrics plotted for InecptionV3 with normal data (left), augmented data (middle) and with fine-tuning (right).

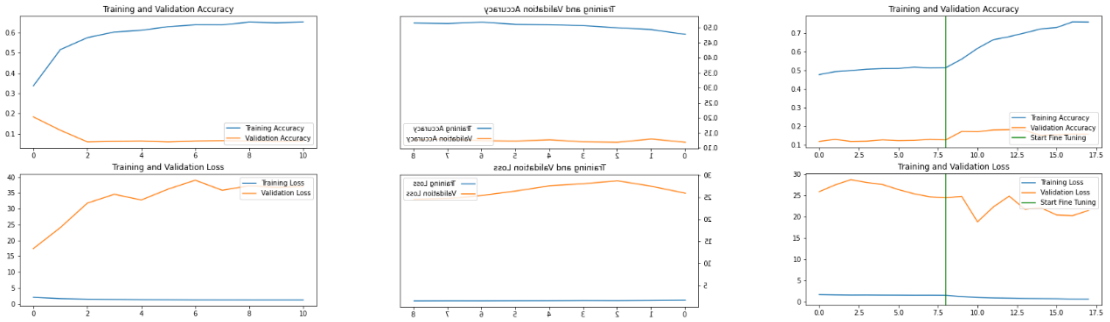

Figure 15: Metrics plotted for Xception with normal data (left), augmented data (middle) and with fine-tuning (right).

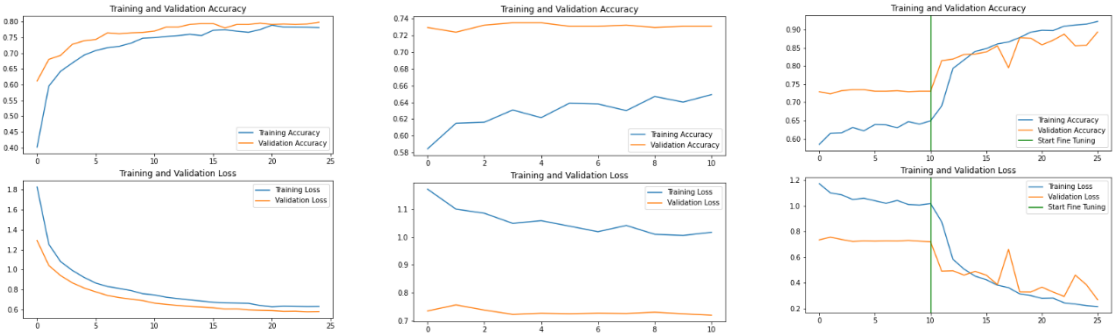

Figure 16: Metrics plotted for EfficientNetB0 with normal data (left), augmented data (middle) and with fine-tuning (right).

3.4 Confusion Matrix

The confusion matrix of the predictions from the EfficientNetB0 model was shown below as an example.

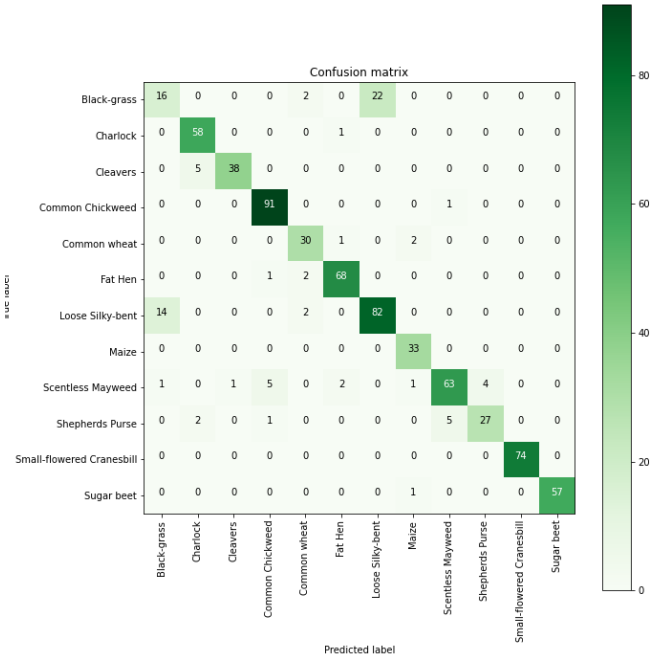

Figure 17: Confusion matrix of the predictions of the EfficientNetB0 model with fine-tuning.

3.5 The Test Data

The images in the test folder were preprocessed with blurring and masking before feeding into the fine-tuned EfficientNetB0 model. Samples of masked test images were shown below.

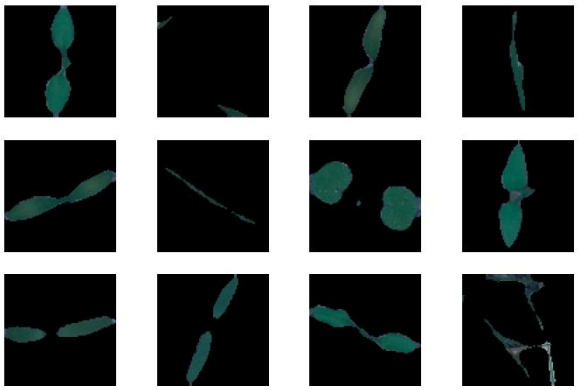

Figure 18: Samples of masked test images.

The results were written into “Submission.csv” and the file was submitted to Kaggle. The submission score is 0.86397, which is close to the accuracy of the fine-tuned EfficientNetB0 model.

4 DISCUSSION AND CONCLUSIONS

This project was conducted on a personal notebook with Google Colab using GPU, total running time was around an hour. Image classification problems require great computational power which is a tough task for personal computers. This project was my first image classification problem with deep learning algorithms, the results shown above clearly indicate that there are room for improvements.

First of all, the masks for preprocessing the images could be tuned. Data preprocessing have always been the most important steps in all aspects of machine learning problems. For example, the combination of the parameter of the ellipse element and the range of green color could be further tested for better results.

The from scratch model was not performing as expected. The quality of preprocessing of the input images may had a great impact on its performance. The model could also be improved by changing the units for the convolutional layers and adding more layers with regularizations.

For pre-trained models, from the graphs of the metrics plotted, most of them were subjected to the over-fitting problem. The problem is caused by varied reasons. Under the context of this project, the first reason that led to over-fitting is a small dataset used on a very deep model. The pre-trained models are trained with millions of images and with transfer learning, the models may easily learn all features, or even more than needed, in the images in the small dataset and hence losing the ability to generalize the features, which results in low prediction accuracy for the test set. One possible solution is to use data augmentation. Data augmentation creates data points from the current small data set, which allows the model to see more images. However, it seemed data augmentation was not enough for the models used. Another solution was to use regularization. I also performed regularization by adding dropout layer in the top layer of the pre-trained models. Yet, except the EfficientNet model, the over-fitting problem still existed. One more reason for the over-fitting problem that I have considered is that I used a global average pooling layer instead of the conventional fully-connected layer.

The results could be improved by continuous parameters and structural tuning. Parameters such as learning rate and optimizer are worth trying, tuning these parameters may lead to significant change in the performance of the models. On the other hand, different types of top layers could be tested for better performance. Another possible way to improve the results is to preprocess the data by different approach. For example, highlighting the plants in the image without removing the background to save more information for the deeper models. Nevertheless, trying out all possible outcomes require great computational power. Hopefully this provides a direction for future works in the same aspect.

REFERENCES

[1] Simonyan, K., & Zisserman, A. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. arXiv:1409.1556. Retrieved from https://arxiv.org/abs/1409.1556

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. 2016. Deep Residual Learning for Image Recognition. 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 770–778. https://doi.org/10.1109/CVPR.2016.90

[3] Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., & Wojna, Z. 2015. Rethinking the Inception Architecture for Computer Vision.Chelsea. arXiv: 1512.00567. Retrieved from https://arxiv.org/abs/1512.00567

[4] Chollet, F. 2016. Xception: Deep Learning with Depthwise Separable Convolutions. arXiv: 1610.02357. Retrieved from https://arxiv.org/abs/1610.02357

[5] Tan, M., & Le, Q. V. 2019. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. arXiv: 1905.11946. Retrieved from https://arxiv.org/abs/1905.11946