6203 Machine Learning II Final Project Report

Plant Pathology 2020 – FGVC7

Identify the category of foliar diseases in apple trees

Kang Huang, Xin Ma, Yongchao Qiao

04/22/2020

Abstract

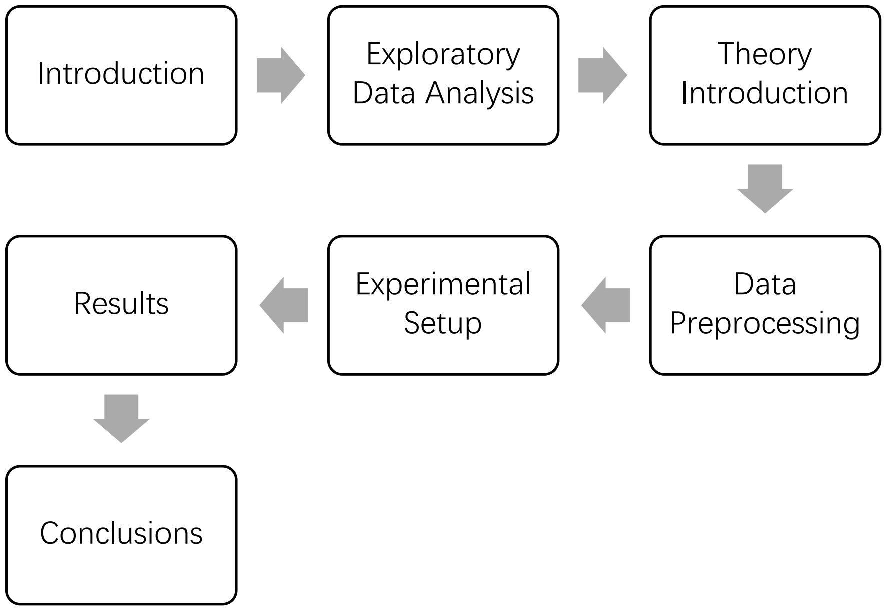
Plant disease diagnosis based on human scouting is time-consuming and expensive. Computer-vision based on models have the promise to increase efficiency and accuracy. This report shows the whole process of building and training our own Convolutional Neural Network (CNN) model to classify a given image from test dataset into different diseased category or a healthy leaf. The performance of the model is evaluated by the average of individual AUCs of each predicted class. The score of the best model is 0.946.

Keywords: Plant Disease, Classification, Convolutional Neural Network (CNN)

1.Introduction

Misdiagnosis of the many diseases impacting agricultural crops can lead to many serious consequences such as misusing of chemicals leading to the emergence of resistant pathogen strains, increasing input costs and more outbreaks with significant economic loss and environment impacts. Current diseases diagnosis based on human scouting is time-consuming and expensive. With the increase of computing power, especially in the past decade, we can build deep learning model to help us diagnose. The computer-vision based on deep learning models have the promise to increase efficiency. However, the great variance in symptoms due to age of infected tissues, genetic variations, and light conditions within trees might increase the complexity of this problem and decreases the accuracy of detection.

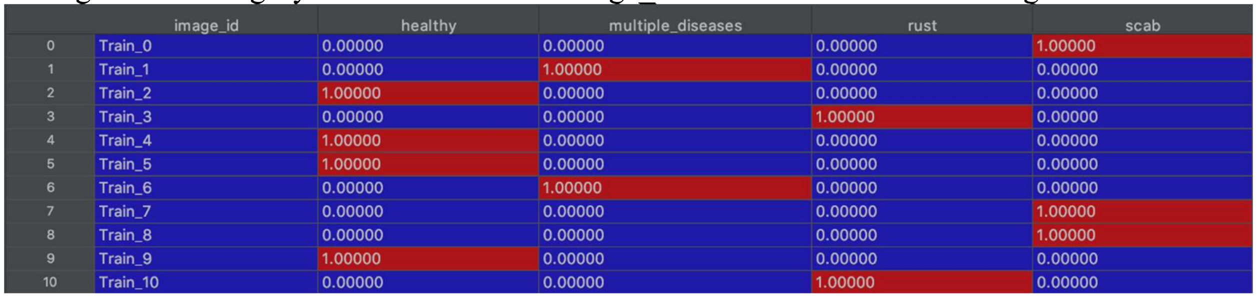
The purpose of our project is to identify the category of foliar diseases in apple trees with our own CNN model instead of the pretrained network. We participated in a competition from Kaggle where we downloaded the datasets. The detail of this competition and datasets will be discussed in next section. Before we start building model, we do some exploratory data analysis (EDA) of datasets. Through EDA, we find there are some problems of metadata such as imbalanced classes, inconsistent image size and duplicated data. To deal with these problems, we use different preprocessing methods, for example, resize, flipping, rotation to augment the data. After data preprocessing, we briefly introduce some essential theories and algorithms that we use in our model. Then, we show the architecture of our model and the process of tuning hyperparameters. Finally, we show performance of our model and summarize the results we obtained.



2.Exploratory Data Analysis

2.1 Metadata

The metadata is downloaded from Kaggle (link: https://www.kaggle.com/c/plant-pathology-2020-fgvc7/data). It contains 1821 train images and corresponding labels and 1821 test images. There are four targets of this dataset: ‘healthy’, ‘multiple_diseases’, ‘rust’ and ‘scab’. Each image only belongs to one category. The data frame of image_id and its label is shown in figure 1.



	image_id	healthy	multiple_diseases	rust	scab
0	Train_0	0.00000	0.00000	0.00000	1.00000
1	Train_1	0.00000	1.00000	0.00000	0.00000
2	Train_2	1.00000	0.00000	0.00000	0.00000
3	Train_3	0.00000	0.00000	1.00000	0.00000
4	Train_4	1.00000	0.00000	0.00000	0.00000
5	Train_5	1.00000	0.00000	0.00000	0.00000
6	Train_6	0.00000	1.00000	0.00000	0.00000
7	Train_7	0.00000	0.00000	0.00000	1.00000
8	Train_8	0.00000	0.00000	0.00000	1.00000
9	Train_9	1.00000	0.00000	0.00000	0.00000
10	Train_10	0.00000	0.00000	1.00000	0.00000

Figure 1: The data frame of image_id and its label

2.2 Targets Distribution

Since we have four targets, we need to check whether the distribution of these four classes is uniform. As figure 2 shown below, the dataset is imbalanced. There is only 5% of training data comes from ‘multiple_diseases’ class. In preprocessing section, we will show how we deal with this problem.

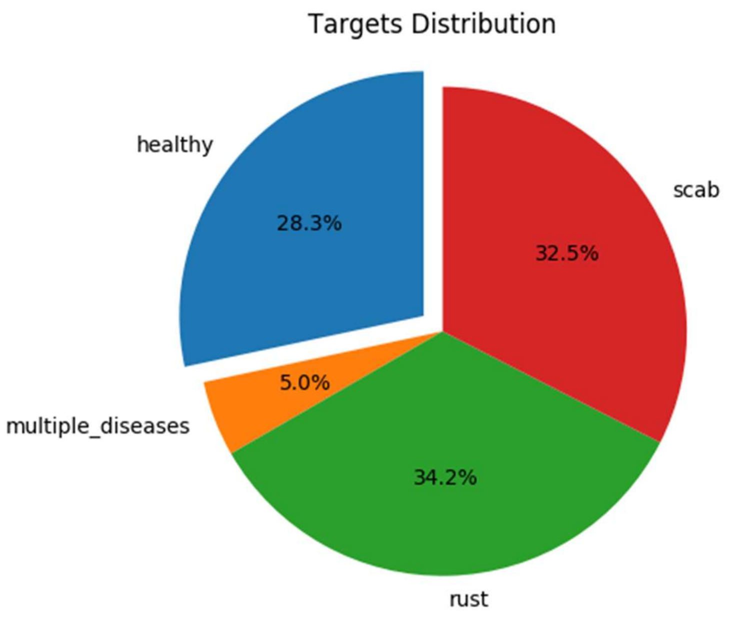


Figure 2: Targets Distribution

2.3 Visualize Sample Images

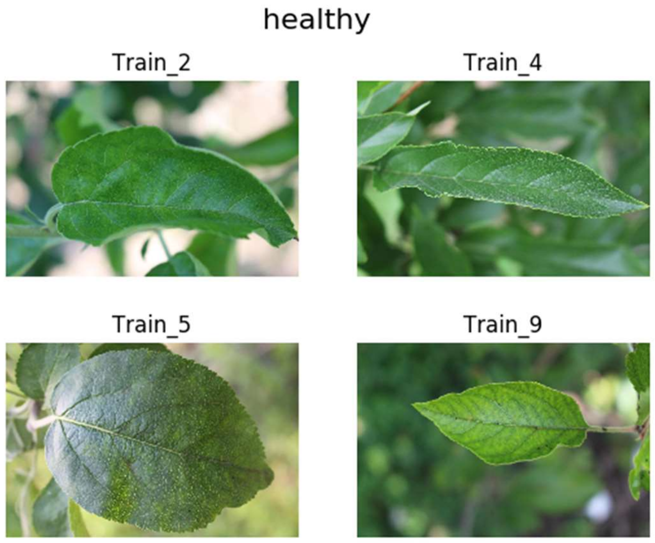


Figure 3: Healthy Leaves

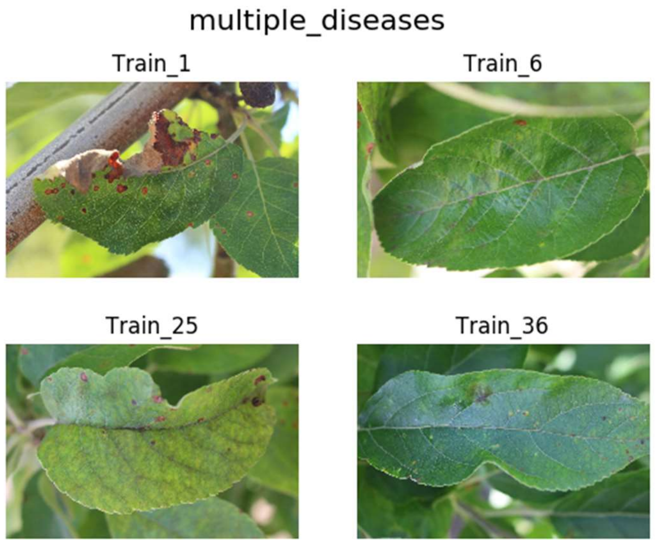


Figure 4: Multiple_Diseases Leaves

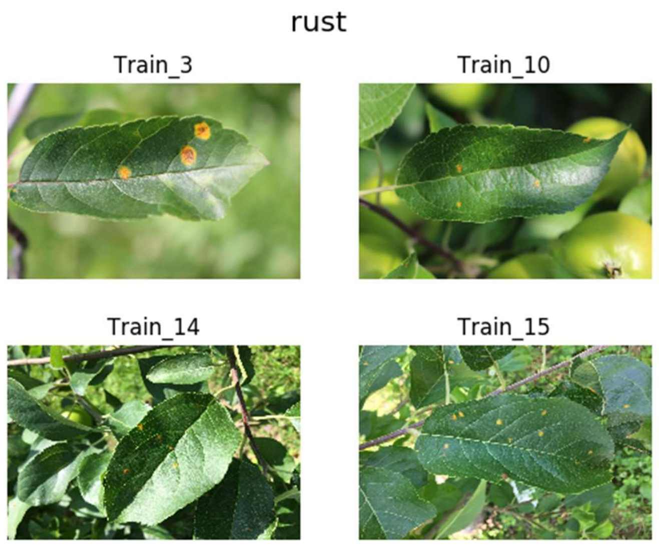


Figure 5: Rust Leaves

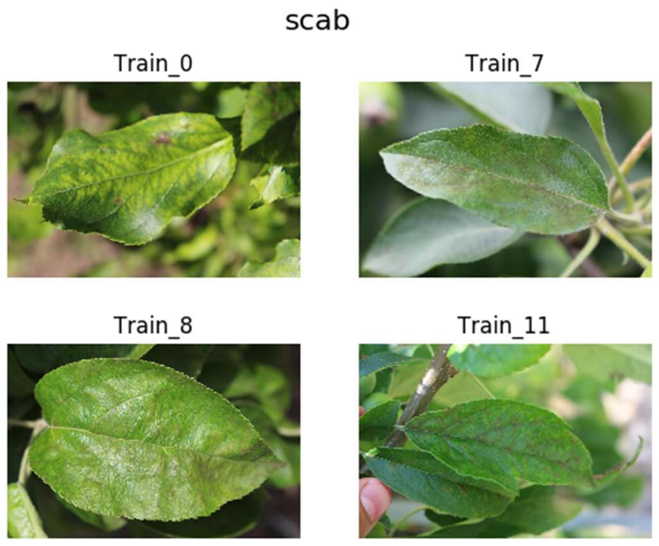


Figure 6: Scab Leaves

3.Theory

Convolution neural networks are a specialized kind of neural network, most commonly applied to analyzing visual imagery. Examples include time-series data, which can be thought of as a 1D grid taking samples at regular time intervals, and image data, which can be thought of as a 2D grid of pixels. The name “convolutional neural network” indicates that the network employs a mathematical operation called convolution. Convolution is a specialized kind of linear operation. Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers.

In its most general form, convolution is an operation on two functions of a real-valued argument. To explain what convolution is, we provide an example. $x(t)$ is the location of the spaceship at time t provided by laser sensor. We suppose that the laser sensor is somewhat noisy. In order to get a less noisy estimate of the spaceship's position, we would like to average together several measurements. Of course, more recent measurements are more relevant, so we will want this to be a weighted average that gives more weight to recent measurements. We can do this with a weighting function w(a), where a is the age of a measurement. If we apply such a weighted average operation at every moment, we obtain a new function s providing a smoothed estimate of the position of the spaceship:

$$s(t) = \int x(a)w(t-a)da$$

This operation is called convolution. We can also denote it as $s(t) = (x * w)(t)$. In convolution network, the convolution layer is the principal layer. In this layer, x in the last equation means input and w refers to kernel. The output s sometimes refers to feature map. For example, let the image input be a $R_r \times R_c$ matrix V. And kernel is represented by a $r \times c$ matrix W. Then the output is:

$$z_{ij} = \sum_{k=1}^{r} \sum_{1=1}^{c} w_{k,l} v_{i+k-1,j+l-1}$$

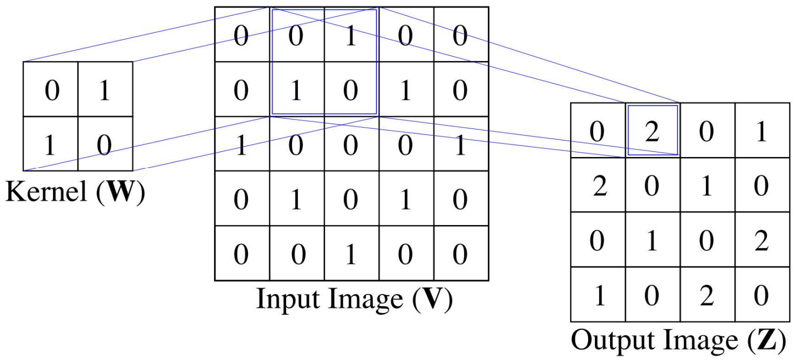


Figure 7: Convolution operation

In addition to convolution layer, convolution neural networks always have another layer: sample layer. In this layer we use pooling function. The pooling function replaces the output of the net at a certain location with a summary statistic of the nearby outputs. For instance, the max pooling operation reports the maximum output within a rectangular neighborhood.

$$z_{i,j} = \max\{v_{r(i-1)+k,c(j-1)+l} | k = 1, \dots, r; l = 1, \dots, c\}$$

In general, pooling helps to make the representation become approximately invariant to small translations of the input. Invariance to translation means that if we translate the input by a small amount, the values of most of the pooled outputs do not change. Invariance to local translation can be a very useful property if we care more about whether some feature is present than exactly where it is.

We choose this network to do our image classification because convolution networks can help improve a machine learning system in three ways: sparse interactions, parameter sharing and equivariant representations. Convolution networks usually have sparse interactions. Just as the figure shows, the output image is smaller than the input because of the convolution operation. This really This means that we need to store fewer parameters, which both reduces the memory requirements of the model and improves its statistical efficiency. It also means that computing the output requires fewer operations. These improvements in efficiency are usually quite large. Parameter sharing refers to using the same parameter for more than one function in a model. The parameter sharing used by the convolution operation means that rather than learning a separate set of parameters for every location, we learn only one set. It can reduce the storage of the model. And parameter sharing causes the layer to have a property called equivariance to translation. This is useful for when we know that some function of a small number of neighboring pixels is useful when applied to multiple input locations.

And how to train this model? Here we also use forward pass to get the output and use back propagation to update the weights. And we validate this updated model to get roc auc score. If we get a higher score, then we save this model. Thus, we can have a best validation score.

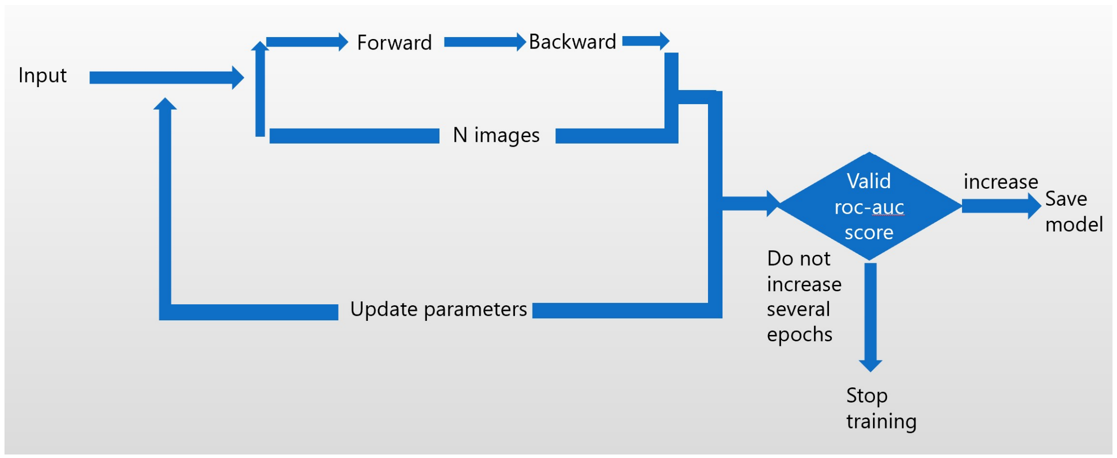


Figure 8: Training algorism

4.Experiment setup

4.1 Data preprocessing

4.1.1 Training part:

There are 1821 observations in the train set which is not that big. In order to make the model more robust, we decided to make the data augmentation. First, we downloaded the original train set: plant-pathology-2020-fgvc7.zip to the computer, uploaded it to cloud and read images as well as target with the one-hot encoding format. Then we saved images of original train set into a tuple group by the matched file keyword, like “Train” and transformed the one-hot encoding target csv file into numpy array. There are total 1821 images, 1819 of whose size are (1365, 2048, 3) with 2 belonging to (2048, 1365, 3). So we transformed the 2 images in (2048, 1365, 3) to (1365, 2048, 3) first. Since the diseases pattern in these images are not that big, we resized them into one same size as (110, 164, 3) to avoid losing much information, fortunately this size was verified as the most suitable one for our model and the memory of cloud CPU and GPU which helped us train the model precisely. Based on the GPU memory, we tried many times and found that augmenting the original dataset to 10926 observations was suitable. That is, each image must generate 5 other new images by nine reasonable random methods like rotation within 30 degrees, horizontal flip, vertical flip, width_shift_range within 0.2, height_shift_range with 0.2, feature-wise_center(Set input mean to 0 over the dataset), feature-wise_std_normalization(Divide inputs by std of the dataset), brightness adjustment in the range [0.8, 1.2] and fill_mode as “reflect”. Some augmented images are shown as below.

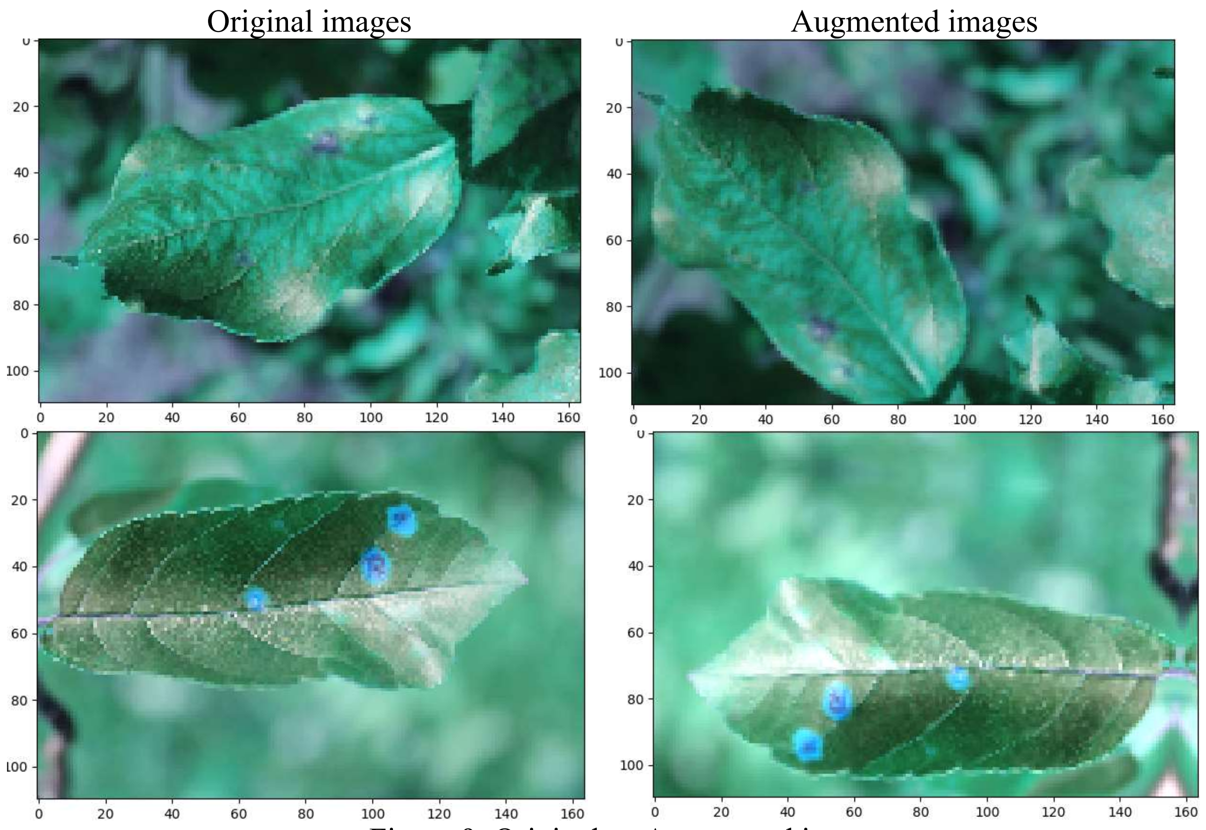


Figure 9: Original vs Augmented images

Also we needed to switch the one-hot encoding target into four numbers representing four categories as the strategy to split the original data set. After augmenting the data, we needed to split the augmented data into train and test set by the generated strategy. Here based on the memory of the GPU, the best randomly stratified separation is: train : test = 0.7 : 0.3 = 7648 : 3278.

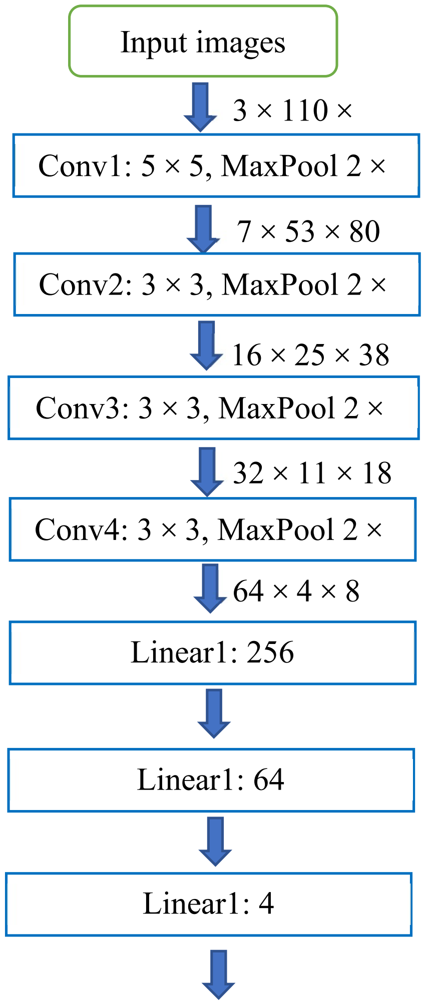
4.1.2 Test part:
Just like the processing of training part, we read images and saved images of original train set into a tuple group by the matched file keyword, like “Test”. There are total 1821 images, 1801 of whose size are (1365, 2048, 3) with 20 belonging to (2048, 1365, 3). So I transformed the 2 images in (2048, 1365, 3) to (1365, 2048, 3) and then resized these test images into (110, 164, 3) to match the model input size.

4.2 Model construction

In this project, we have tried many convolution neural networks and played with the convolution layer and pooling layer as what we said, like more layers or less layers. Finally, we found one best single CNN model which is shown as below. This is a deep network containing 11 convolution layers and 3 linear layers, with activation function as "relu" and 14 batchnormalization layers. Also we will show an original convolution neural network that modified from Exam 2.

4.2.1 The original CNN

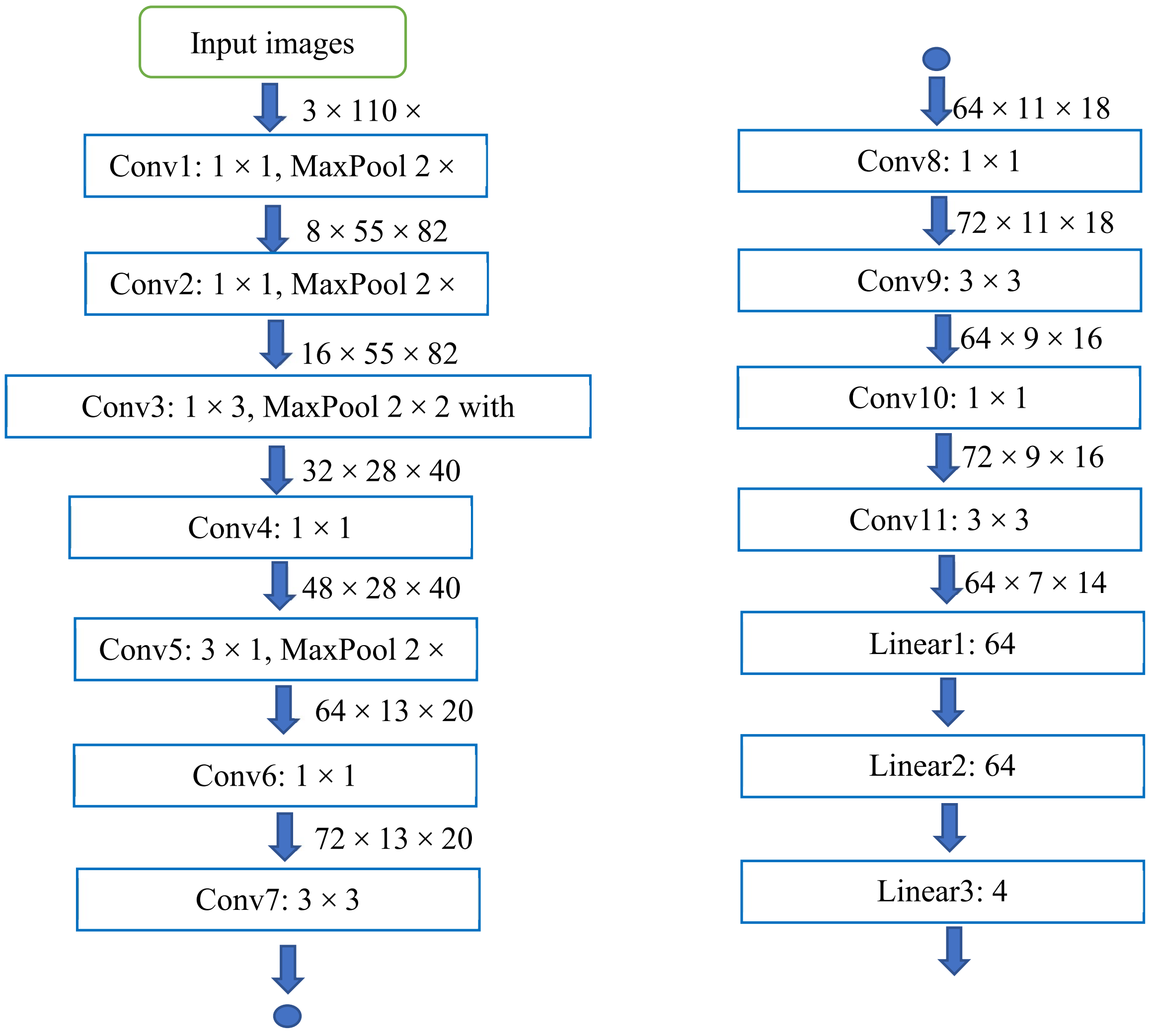
Figure 10: The original CNN


Input images
3 × 110 ×
Conv1: 5 × 5, MaxPool 2 ×
7 × 53 × 80
Conv2: 3 × 3, MaxPool 2 ×
16 × 25 × 38
Conv3: 3 × 3, MaxPool 2 ×
32 × 11 × 18
Conv4: 3 × 3, MaxPool 2 ×
64 × 4 × 8
Linear1: 256
Linear1: 64
Linear1: 4

There are 4 general convolution layers and 3 linear layers in this CNN, also we add the batch normalization layers after each of these 7 layers. And the activation function is "relu".

4.2.2The best single model

Figure 11: The best CNN



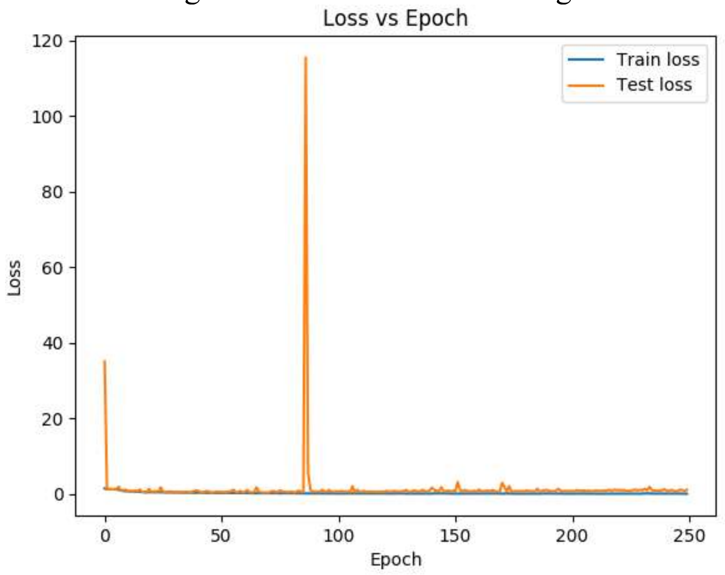
There are 11 general convolution layers and 3 linear layers in this CNN, also we add the batch normalization layers after each of these 14 layers. And the activation function is “relu”.

4.3 Parameter tuning

4.3.1 Training operations

We used the train set and test set which are split from the original train dataset, to train the model. we fed the model with train set and test it on the test set during the training procedure. The performance index is cross entropy loss but we will use the AUC score on the test set to check the performance of the model. Also, we added the early stopping operation to prevent the overfitting and save the model weight which own the highest AUC score during the training step. Besides, there are other methods that I used to prevent the overfitting, like the batch normalization layers in the model, dropout and specify the group parameters in the convolution layers. For detecting the overfitting, we just plot the train loss and test loss against the epochs as shown below.

Figure 12: Detect the overfitting



From the plot, it is still hard to detect the overfitting pattern, but if we zoom the bottom part, there may exist overfitting pattern since the test loss has a slight upward tendency while the train loss is decreasing.

4.3.2 Minibatches

Based on the memory of the GPU, the max value of minibatch is 430 so we tried several different values of the minibatch size, like 200, 412, 430. The comparison is shown in the results part.

4.3.3 Learning rate

For the learning rate, based on the experience of Exam 1 and Exam 2, first we needed to check which form of the learning rate schedular is the best. So with other parameters being the same, we tried one static leaning rate as 0.1 for the whole training loop and one dynamic learning rate as cosine annealing schedular with max value as 0.1. Then we found the dynamic learning rate is better than the static learning rate so we began to tuning the specific value of the max learning rate in the cosine annealing schedular. Besides, we also tried other dynamic learning rate schedule like, ReduceLROnPlateau, CosineAnnealingWarmRestarts and so on. However, for my best model the best dynamic learning schedule was cosine annealing learning rate schedular so we decide to tune the max value of it. The comparison will be shown in the results section.

4.3.4 Initial weight method

In this project, we did not use the default initial weight for my CNN model but checked four different initial weight methods. They are xavier_normal, xavier_uniform, kaiming_normal and kaiming_uniform. For initial biases, we just let them follow the Norm(0, 0.02) distribution which is also the experience from Exam 2. Then the comparison is shown in the result section.

4.3.5 Optimizers

In this project, we only tried Adam and SGD, since SGD is the first optimizer we learnt from this class and Adam is a quite fast optimizer. Also, the comparison is shown in the results section.

4.3.6 Image size

In this project, we tried 2 types of image size: (110, 164, 3) and (82, 120, 3). To make the results comparable, we changed the image size and rebuilt the model to match the input size with other parameters being the same. And the comparison is shown in the results section.

4.3.7 Original CNN and deeper CNN
In this project, we tried many types of convolution neural network. For example, the original CNN and the best single model as a deeper one. To make the results comparable, we only changed the model with other parameters being the same. And the comparison is shown in the results section.

4.3.8 Confusion matrix and classification report
Based on the confusion matrix and the classification report, we may find the problem in the trained model.

4.3.9 Class weights
Based on the results of confusion matrix and the classification report and the imbalanced data fact, we may check the influence of class weights specified in the cross entropy loss function.

4.3.10 Ensemble model
In this project, we tried to combine different model weights with high AUC score to get the highest AUC score on the Kaggle test set.

5.Results

5.1 Minibatch comparison

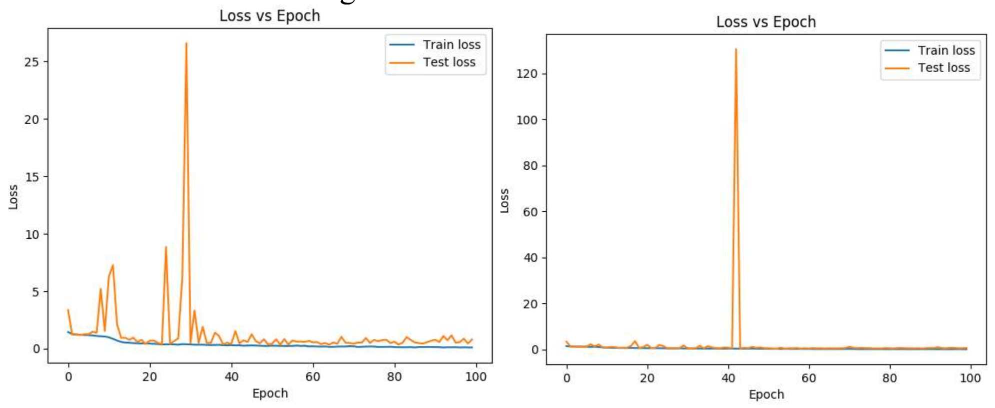
The test AUCs for minibatch comparison on the Kaggle are shown in table 1, which indicates that the there is no obvious relationship between the minibatch size and test AUC score. Based on the results and with the consideration of the training time, we chose 412 as the minibatch size since it provided the highest AUC score in this comparison.

Table 1: The influence of minibatch size

Minibatch Size	200	412	430
Test AUC$_{(kaggle)}$	0.935	0.937	0.915

5.2 Learning rate comparison

Figure 13: Test for the Adam



Before comparing, we did a test on the Adam optimizer to find answer of: If I use Adam do we still need a learning rate schedular, since Adam is an adaptive learning rate optimizer? The figure 7, show the train loss and test loss of under different learning rate situation: Left one does not have the learning schedular and right one has the Cosine learning schedular. From the plot, we can see the loss are different which means learning rate schedular can still impact the loss curves when the optimizer is Adam.

Table 2: The influence of learning rate

Static LR	0.1	Cosine LR	0.15	0.1	0.01
TestAUC$_{(kaggle)}$	0.911	Test AUC$_{(kaggle)}$	0.923	0.937	0.909

The test AUCs for Learning rate comparison on the Kaggle are shown in table 2. Firstly, when the initial learning rate is 0.1, the dynamic learning rate schedular performed better than static learning rate. So we decided to use the dynamic learning rate schedular to find the best weight of the model. Then we chose the annealing schedular, since its values are quite wider. Besides, we changed the max value of the cosine annealing schedular to find the best learning rate. However, there is still no obvious relationship between the learning rate and test AUC score. Based on the results and with the consideration of the training time, we chose 0.1 as the max value of the cosine annealing schedular since it provided the highest AUC score in this comparison and would reduce our running time.

5.3 Weights initialization method comparison

The test AUCs for Initial weight method comparison on the Kaggle are shown in table 3. The result clearly indicates that Xavier normal is the best method of weights initialization.

Table 3: The influence of weights initialization

Weights Initialization	Xavier normal	Xavier uniform	Kaiming normal	Kaiming uniform
Test AUC$_{(kaggle)}$	0.937	0.928	0.927	0.922

5.4 Optimizers comparison

The test AUCs for optimizers comparison on the Kaggle are shown in table 4. The result clearly indicates that Adam's performance is better than SGD.

Table 4: The influence of optimizers

Initial weight method	SGD	Adam
Test AUC$_{(kaggle)}$	0.901	0.937

5.5 Image size comparison

The test AUCs for image size comparison on the Kaggle are shown in table 5. The result clearly indicates that the bigger size of image will lead to the higher AUC score.

Table 5: The influence of image size

Image size	(110, 164, 3)	(82, 120, 3)
Test AUC$_{(kaggle)}$	0.937	0.912

5.6 The original CNN and the deeper CNN

The test AUCs for different CNNs' comparison on the Kaggle are shown in table 6. The result clearly indicates that the deeper convolution neural network, the higher AUC score, in some this case.

Table 6: The influence of image size

CNN	The original one	Deeper one
Test $AUC_{(kaggle)}$	0.920	0.937

5.7 Confusion matrix and classification report

Table 7: Confusion matrix

	Healthy	Multiple diseases	Rust	Scab
Healthy	847	28	2	59
Multiple diseases	11	98	26	29
Rust	6	53	1059	2
Scab	38	21	0	1006

Table 8: Classification report

	Precision	Recall	F1-score	Support
Healthy	0.94	0.91	0.93	929
Multiple diseases	0.49	0.60	0.54	164
Rust	0.97	0.95	0.96	1120
Scab	0.92	0.94	0.93	1065
Accuracy			0.92	3278
Macro avg	0.83	0.85	0.84	3278
Weighted avg	0.92	0.92	0.92	3278

Table 7 and table 8 show the confusion matrix and the classification report of the best single model with 0.937 on the Kaggle. From these two tables, we can diagnose the best single model. One possible improvement direction may exist in the category "Multiple diseases". If we can improve the accuracy of category "Multiple diseases", the model may perform better on Kaggle. Since there are least observations for "Multiple diseases" in these four categories, one class weights or more augmented images in this category may be needed.

5.8 Class weights

The test AUCs on the Kaggle are shown in table 9. The result clearly indicates that If we specify the class weights as (3.53, 19.99, 2.93, 3.08) based on the counts of each category to the cross entropy loss function, the previous best model will be improved.

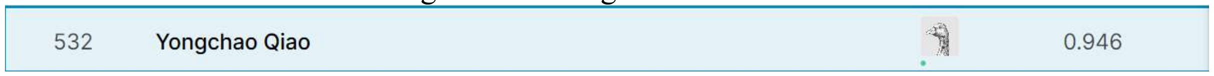
Table 9: The influence of class weights

Class weights	No specified weights	Specified weight
Test $AUC_{(kaggle)}$	0.937	0.938

5.9 Ensemble model

This method really worked well. After combining top 6 highest AUC score's model weights, we got the best AUC in this project which is 0.946.

Figure 3: The highest AUC score



532 Yongchao Qiao 0.946

6.Summary

Generally, in this project each parameter in the model can have a big influence on the model performance. A)Specifically, in this project, there is no obvious relationship between minibatch size and the performance of the model. With the consideration of running time, we prefer a bigger minibatch size with high AUC score. B)For the learning rate, using the dynamic learning rate is a good path to find the best AUC score. In this project, the Cosine annealing schedular worked well. Besides, the max value of the learning rate in this schedular is also important. C)Then the weights initialization is also an important part. In this project, the Xavier normal weights initialization method performed well. D)Adam is still a good optimizer to train the model and E)bigger image size as well as F)deeper network will lead to a good performance. G)Finally, the ensemble of some high performance model weights is a good way to get a higher performance.

All these things above are what we learnt in this project. Based on this we found that the experiment is a good way to learn some new parameter tuning rules in this class. Also, if given more time, we want to try the CNN with some branches and find efficient methods to reduce the occupied GPU memory during model training, which means a more efficient method still need to be found. Finally, since there are least observations for "Multiple diseases" in these four categories leading to lowest accuracy and the improvement of specified class weights is limited, another operation that adding more augmented images in this category to balance the train set may be needed.

7.Reference

[1] Mingxing Tan, Quoc V. Le. *EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks*, ICML 2019
[2] Hagan, Demuth etc. *Neural Network Design 2nd edition*.
[3] https://pytorch.org/docs/stable/torch.html
[4] Kaggle notebook: https://www.kaggle.com/pestipeti/plant-pathology-2020-pytorch
[5] Pytorch tutorial: https://pytorch.org/tutorials/
[6]Liu, T., Fang, S., Zhao, Y., Wang, P., & Zhang, J. (2015). *Implementation of training convolutional neural networks*. arXiv preprint arXiv:1506.01195.
[7]Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT press.

8.Appendix

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 8, (1, 1))  #
output (n_examples, 8, 110, 164)
        self.convnorm1 = nn.BatchNorm2d(8)
        self.pool1 = nn.MaxPool2d((2, 2))  # output
(n_examples, 8, 55, 82)
        self.conv2 = nn.Conv2d(8, 16, (1, 1))  #
output (n_examples, 16, 55, 82)
        self.convnorm2 = nn.BatchNorm2d(16)
        self.conv3 = nn.Conv2d(16, 32, (1, 3),
groups=1)  # output (n_examples, 32, 55, 80)
        self.convnorm3 = nn.BatchNorm2d(32)
        self.pool3 = nn.MaxPool2d((2, 2),
padding=(1, 0))  # output (n_examples, 32, 28, 40)
        self.conv4 = nn.Conv2d(32, 48, (1, 1))  #
output (n_examples, 48, 28, 40)
        self.convnorm4 = nn.BatchNorm2d(48)
        self.conv5 = nn.Conv2d(48, 64, (3, 1))  #
output (n_examples, 64, 26, 40)
        self.convnorm5 = nn.BatchNorm2d(64)
        self.pool5 = nn.MaxPool2d((2, 2))  # output
(n_examples, 64, 13, 20)
        self.conv6 = nn.Conv2d(64, 72, (1, 1))  #
output (n_examples, 72, 13, 20)
        self.convnorm6 = nn.BatchNorm2d(72)
        self.conv7 = nn.Conv2d(72, 64, (3, 3),
groups=1)  # output (n_examples, 64, 11, 18)
        self.convnorm7 = nn.BatchNorm2d(64)
        self.conv8 = nn.Conv2d(64, 72, (1, 1))  #
output (n_examples, 72, 11, 18)
        self.convnorm8 = nn.BatchNorm2d(72)
        self.conv9 = nn.Conv2d(72, 64, (3, 3),
groups=1)  # output (n_examples, 64, 9, 16)
        self.convnorm9 = nn.BatchNorm2d(64)
        self.conv10 = nn.Conv2d(64, 72, (1, 1))  #
output (n_examples, 72, 9, 16)
        self.convnorm10 = nn.BatchNorm2d(72)
        self.conv11 = nn.Conv2d(72, 64, (3, 3),
groups=1)  # output (n_examples, 64, 7, 14)
        self.convnorm11 = nn.BatchNorm2d(64)
        self.linear1 = nn.Linear(64 * 7 * 14, 64)  #
input will be flattened to (n_examples, 64*7*14)
        self.linear1_bn = nn.BatchNorm1d(64)
        self.drop1 = nn.Dropout(DROPOUT1)
        self.linear2 = nn.Linear(64, 64)
        self.linear2_bn = nn.BatchNorm1d(64)
        self.drop2 = nn.Dropout(DROPOUT2)
        self.linear3 = nn.Linear(64, 4)
        self.act = nn.ReLU(inplace=True)
        self.act2 = nn.Softmax(dim=1)

    def forward(self, x):
        x =
self.pool1(self.convnorm1(self.act(self.conv1(x.per
mute(0, 3, 1, 2)))))
        x = self.convnorm2(self.act(self.conv2(x)))
        x =
self.pool3(self.convnorm3(self.act(self.conv3(x))))
        x = self.convnorm4(self.act(self.conv4(x)))
        x =
self.pool5(self.convnorm5(self.act(self.conv5(x))))
        x = self.convnorm6(self.act(self.conv6(x)))
        x = self.convnorm7(self.act(self.conv7(x)))
        x = self.convnorm8(self.act(self.conv8(x)))
        x = self.convnorm9(self.act(self.conv9(x)))
        x =
self.convnorm10(self.act(self.conv10(x)))
```

```
        x =
self.convnorm11(self.act(self.conv11(x)))
        x =
self.drop1(self.linear1_bn(self.act(self.linear1(x.
view(len(x), -1)))))
        x =
self.drop2(self.linear2_bn(self.act(self.linear2(x)
)))
        return self.linear3(x)


def weights_init_1(m):
    classname = m.__class__.__name__
    if classname.find('Conv') != -1:
        torch.nn.init.xavier_normal_(m.weight.data,
gain=nn.init.calculate_gain('relu'))
        torch.nn.init.normal_(m.bias.data, 0, 0.02)
    elif classname.find('linear') != -1:
        torch.nn.init.xavier_normal_(m.weight.data,
gain=nn.init.calculate_gain('relu'))
        torch.nn.init.normal_(m.bias.data, 0, 0.02)
```