Apple leaf disease discovery with Deep Learning and Keras using a Plant Pathology dataset from Kaggle

Apples are one of the most important temperate fruit crops in the world. Leaf diseases can be a large problem for apple orchards because the diseased plants can have direct repercussions on the number of apples that can be produced and the quality of those apples. Currrently, apple leaf diseases are discovered by manual labor which can take considerable amounts of time that can be used for production and raise labor costs.

Leaf color and leaf morphology, the age of infected tissues, non-uniform image background, and different light illumination during imaging pose significant limitations to plant disease classification and identification with computer vision models. A deep learning model using Keras will be developed and trained on part of the training data and subsequently tested on a smaller part of the training data as the Kaggle test set is has only three images. The Keras model will attempt to classify a given leaf image from the newly created test dataset to a particular disease category. An effective model could also be used in other plant pathology environments as well and potentially improve plant quality of life as well as fruit quality and labor costs.

The training set images will have to be formatted, resized and reduced as there is not enough RAM for all 18000 images. 10240 of the original images will be used as the training set and coordinated with the 'train/csv' file. The resulting set of images will be called "trainset."

Exploratory Data Analysis

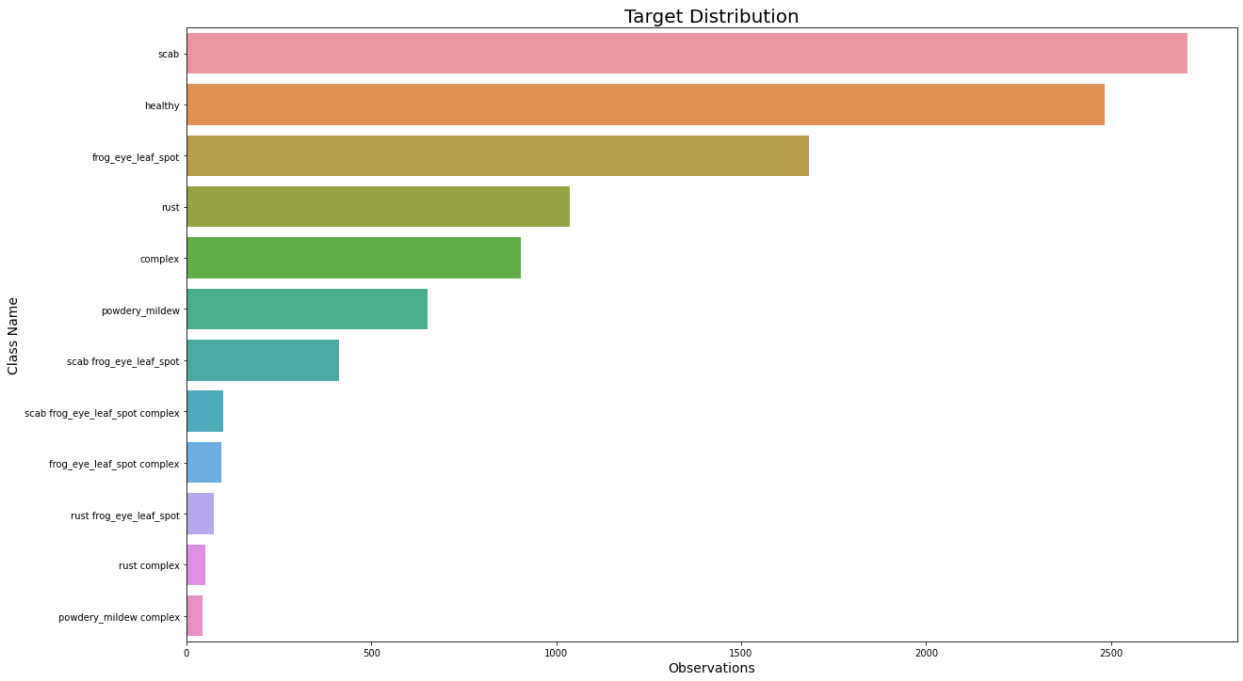
	image	labels
0	/content/train_images/800113bb65efe69e.jpg	healthy
1	/content/train_images/8002cb321f8bfcdf.jpg	scab frog_eye_leaf_spot complex
2	/content/train_images/80070f7fb5e2ccaa.jpg	scab
3	/content/train_images/80077517781fb94f.jpg	scab
4	/content/train_images/800cbf0ff87721f8.jpg	complex
5	/content/train_images/800edef467d27c15.jpg	healthy
6	/content/train_images/800f85dc5f407aef.jpg	rust
7	/content/train_images/801d6dcd96e48ebc.jpg	healthy
8	/content/train_images/801f78399a44e7af.jpg	complex
9	/content/train_images/8021b94d437eb7d3.jpg	healthy
10	/content/train_images/802291cee9fec9f4.jpg	complex
11	/content/train_images/80230a9a3f7a9f6b.jpg	scab
12	/content/train_images/8023c3f31f875b6c.jpg	healthy
13	/content/train_images/80261f473eafb92c.jpg	scab
14	/content/train_images/80273091d9e9bddb.jpg	frog_eye_leaf_spot
15	/content/train_images/802962dc3ecdbb8d.jpg	scab
16	/content/train_images/802969daaddbbc8c.jpg	scab
17	/content/train_images/802b34badefa2ed0.jpg	scab
18	/content/train_images/802b59956a7aa5e7.jpg	healthy
19	/content/train_images/802f4bbd295063fe.jpg	scab
20	/content/train_images/802f7439ec1ef0cd.jpg	powdery_mildew

	image	labels
count	10240	10240
unique	10240	12
top	/content/train_images/a0e33c9d0c7792e9.jpg	scab
freq	1	2706

The shape of the dataframe is to be expected as here is where the labels will come from. The summary above states that there are 12 unique labels. The first 21 labels of the dataframe was displayed to show all of the potential disease categories.

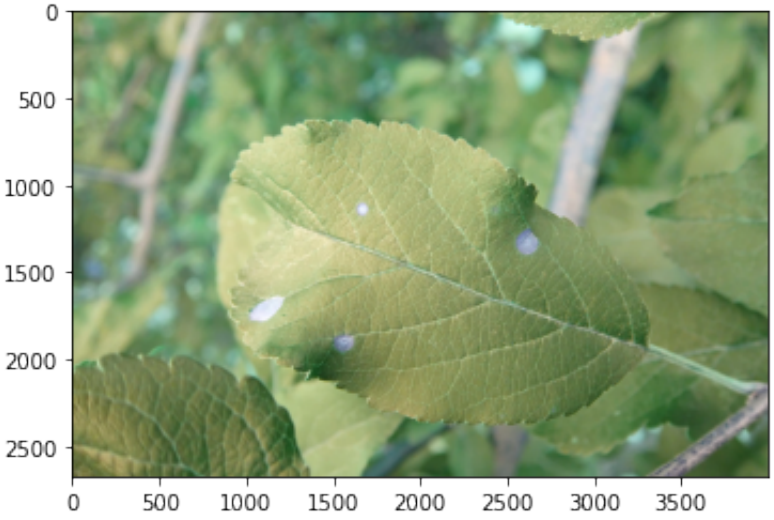
Below is a visualization of the occurrence of different types of disease in the training set using the seaborn package. The labels are not evenly distributed. More importantly, some labels consist of spaces with will be a problem for a deep neural network. Only six of the most common diseases will be used since the other labels look to be conglomerations of other diseases.

```
<Figure size 432x288 with 0 Axes>
```



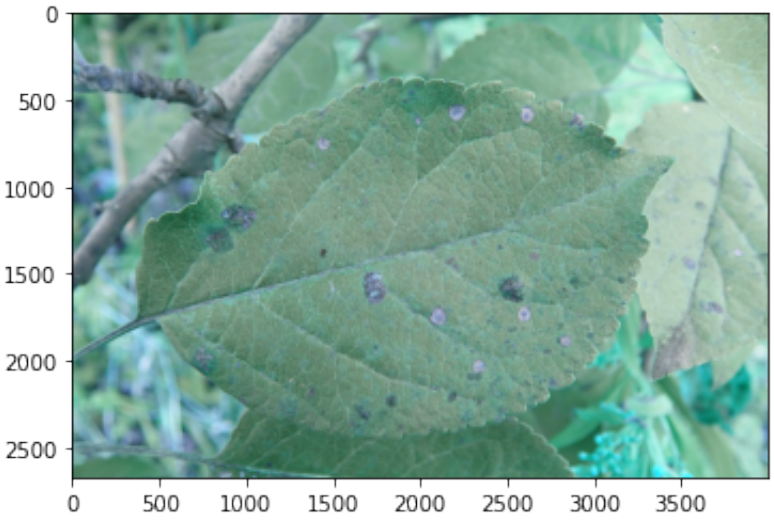
The test images will be read into an array for display since there is only three.

```
<matplotlib.image.AxesImage at 0x7fbbdf1e80d0>
```



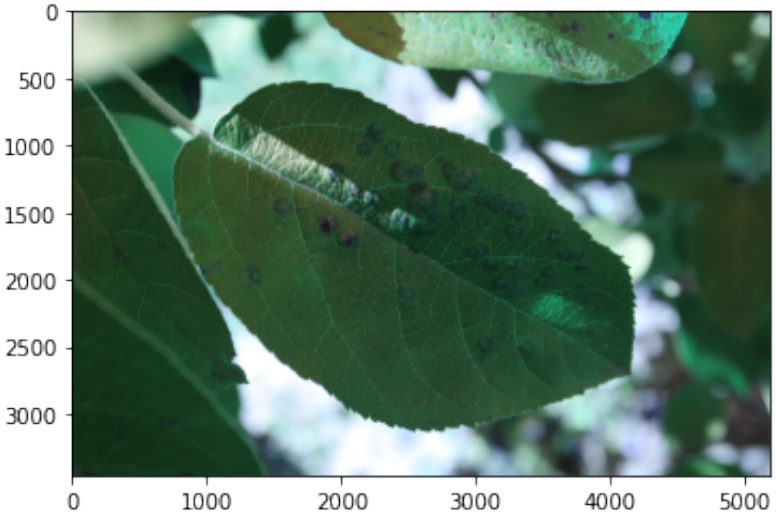
The first image has a length of 2672 and a width of 4000 with a depth of three channels indicating a color image.

```
<matplotlib.image.AxesImage at 0x7fbbdf0f0610>
```



The second image has the same length, width and depth as the second.

```
<matplotlib.image.AxesImage at 0x7fbbdf0dab10>
```



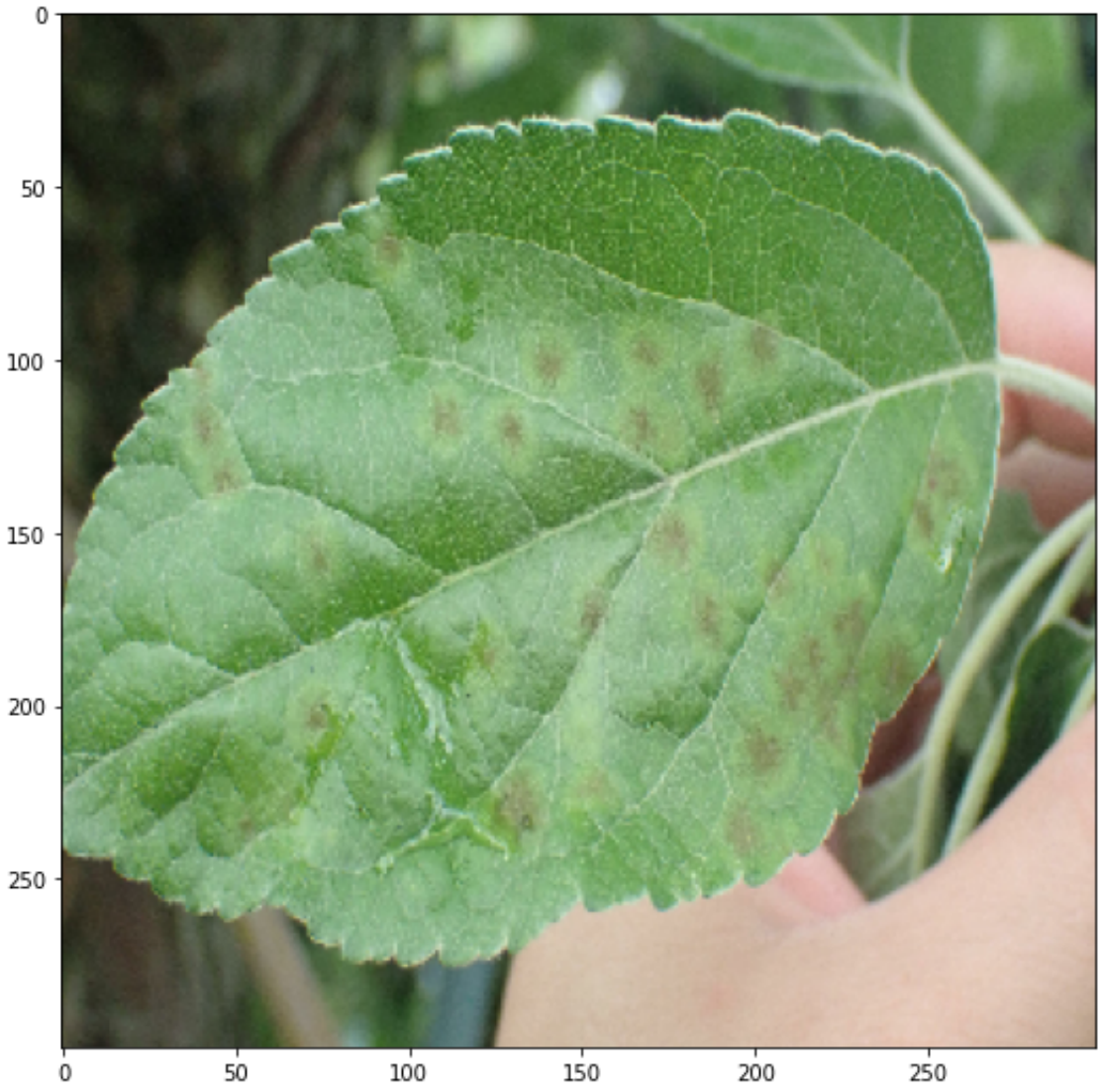
The third image has a length of 3456 and a width of 5184 !

As mentioned above, we will be using the six truly unique classes of apple leaf disease. We will take a look at each in detail.

1. Scab

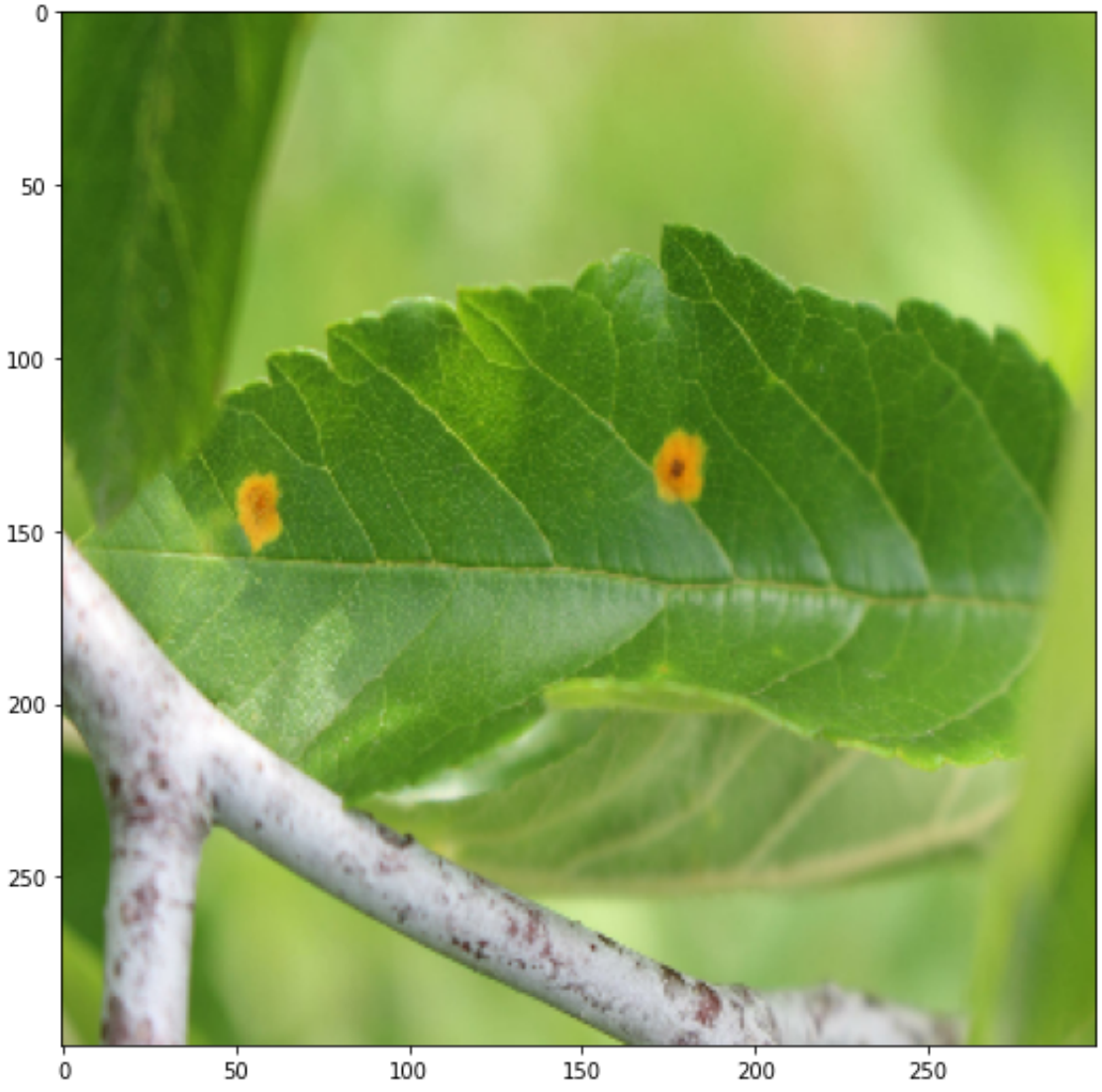
Scab is serious disease of apples and ornamental crabapples, apple scab (Venturia inaequalis) attacks both leaves and fruit. The fungal disease forms pale yellow or olive-green spots on the upper surface of leaves. Dark, velvety spots may appear on the lower surface. Severely infected leaves become twisted and puckered and may drop early in the summer.

Symptoms on fruit are similar to those found on leaves. Scabby spots are sunken and tan and may have velvety spores in the center. As these spots mature, they become larger and turn brown and corky.



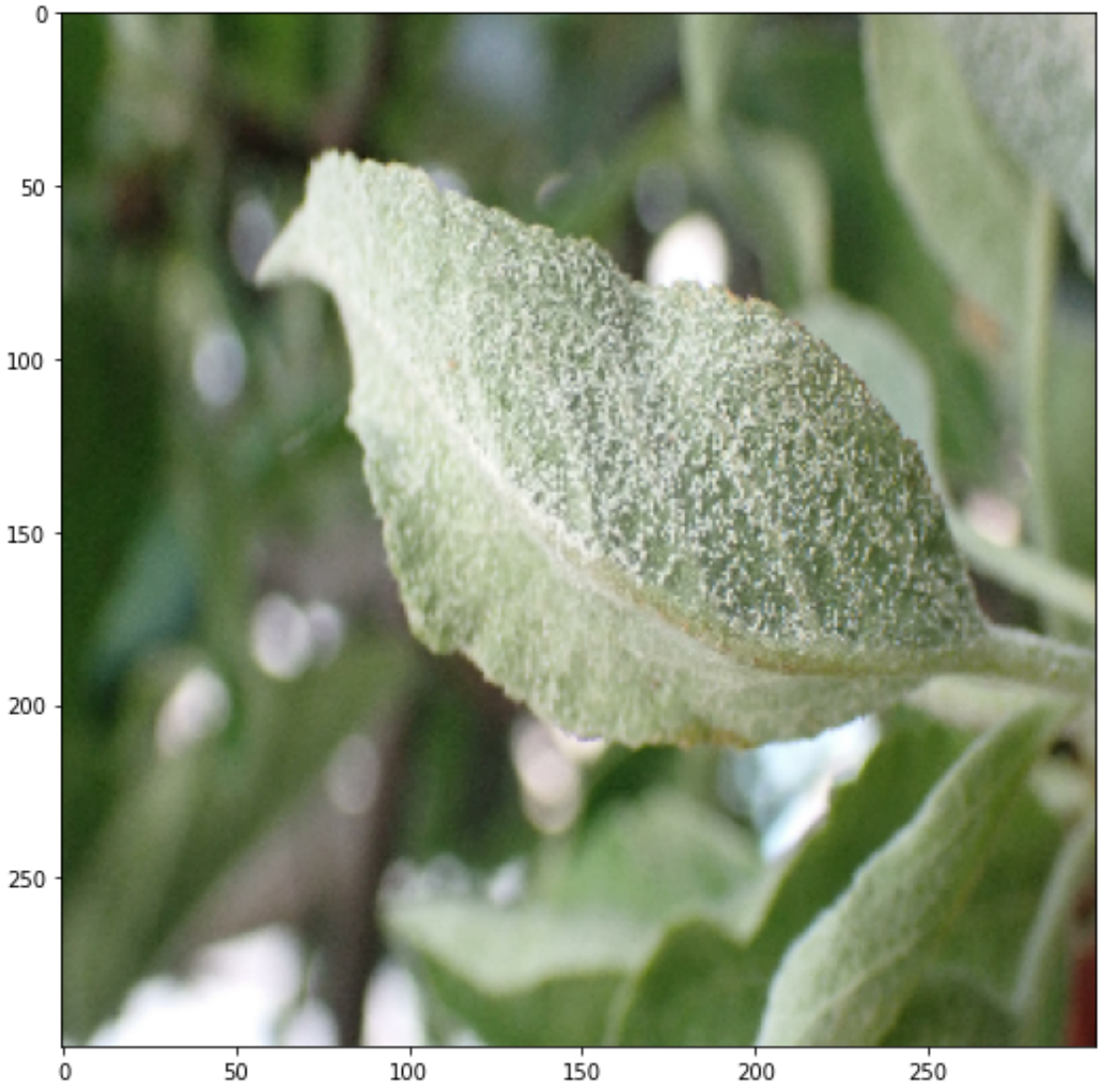
2. Rust

Circular, yellow spots (lesions) appear on the upper surfaces of the leaves shortly after bloom. In late summer, brownish clusters of threads appear beneath the yellow leaf spots or on fruits and twigs. The spores associated with the threads the leaves during wet, warm weather.



3. Powdery Mildew

Powdery mildew is a fungal disease that affects a wide range of plants and is one of the easier plant diseases to identify, as its symptoms are quite distinctive. Infected plants display white powdery spots on the leaves and stems. The lower leaves are the most affected, but the mildew can appear on any above-ground part of the plant. As the disease progresses, the spots get larger and denser as large numbers of asexual spores are formed, and the mildew may spread up and down the length of the plant.

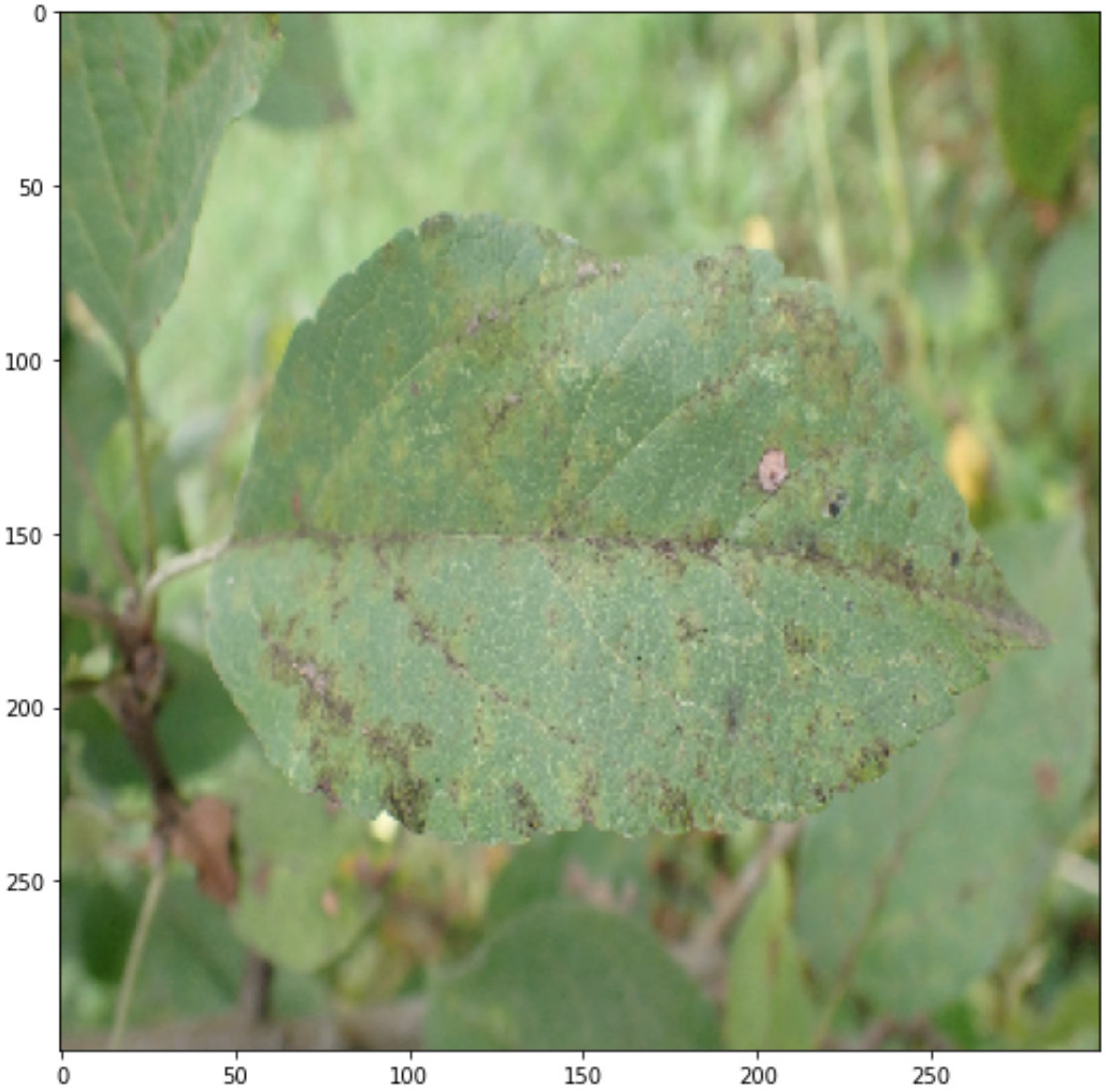


4. Frog eye leaf spot

First, small purple spots form on the leaves. These spots gradually enlarge and eventually develop into lesions with a light tan interior, surrounded by a dark purple perimeter. Heavy infections of frog-eye leaf spot can cause leaves to turn yellow and drop.

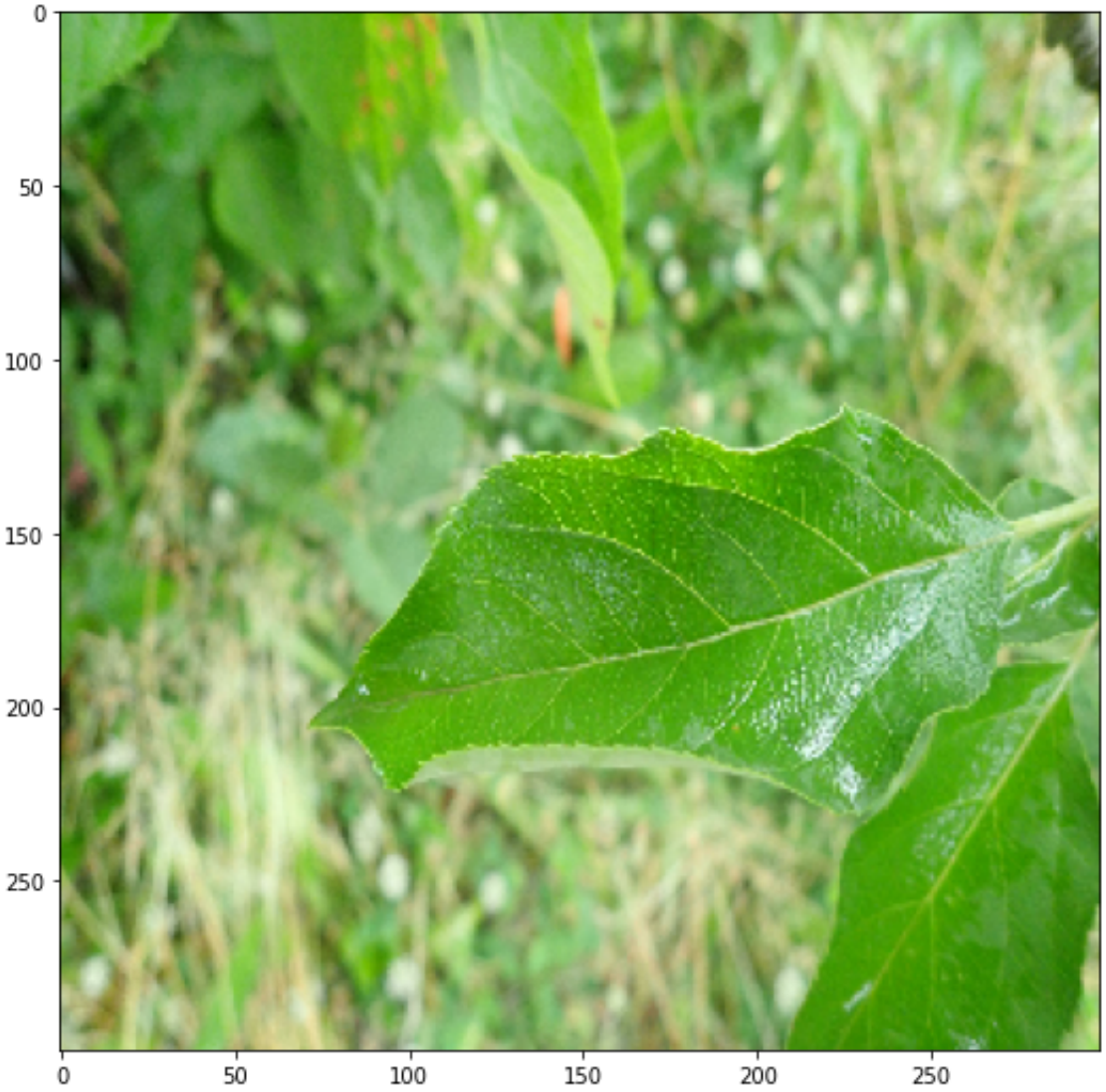
5. Complex

According to data description - Unhealthy leaves with too many diseases to classify visually will have the complex class, and may also have a subset of the diseases identified.



6. Healthy

These leaves should exhibit no evidence of disease exclusive of the other classes that we have discussed.



A little bit of Feature Engineering and Data Wrangling

A class will be created to compare labels against as only the six most relevant classes of disease will be used.

The MultiLabelBinarizer is used to remove whitespace in the labels and create a format usable for a Keras neural network model.

image	complex	frog_eye_leaf_spot	powdery_mildew	rust
/content/train_images/800113bb65efe69e.jpg	0	0	0	0
/content/train_images/8002cb321f8bfcdf.jpg	1	1	0	0
/content/train_images/80070f7fb5e2ccaa.jpg	0	0	0	0
/content/train_images/80077517781fb94f.jpg	0	0	0	0
/content/train_images/800cbf0ff87721f8.jpg	1	0	0	0

The dataframe still has 10240 rows and now, 6 columns.

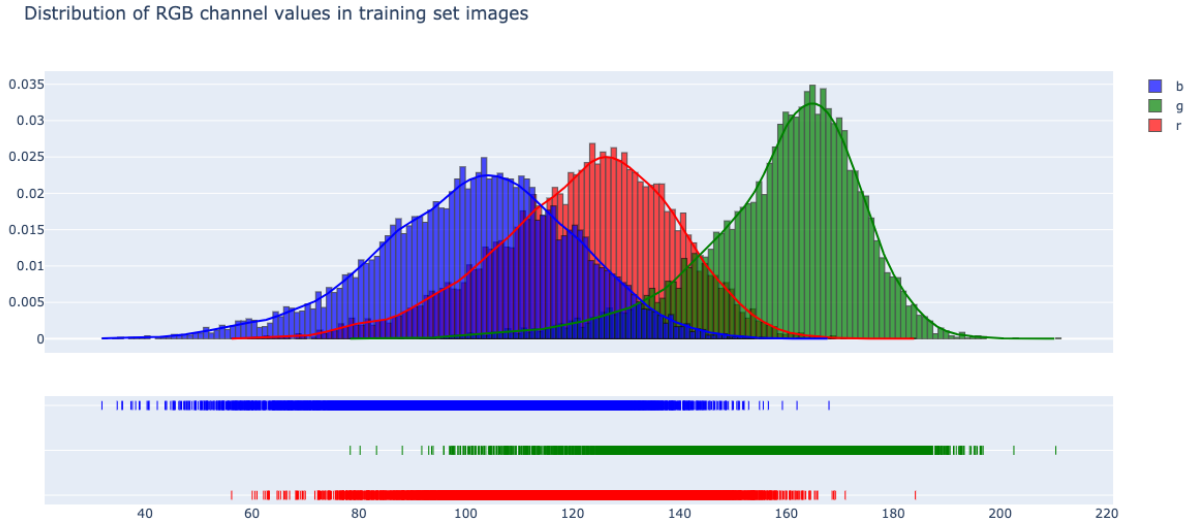
A test set will now be created from the training set using the same methods so that there are plenty of images to predict.

The MultiLabelBinarizer is used again remove whitespace in our labels for the test set.

image	complex	frog_eye_leaf_spot	powdery_mildew	rust
/content/train_images/c6b460353d77c362.jpg	0	0	0	0
/content/train_images/c6b4b491f27e4349.jpg	1	0	0	0
/content/train_images/c6b50cf4bd087374.jpg	0	0	0	0
/content/train_images/c6b833780fe5c34a.jpg	1	0	0	0
/content/train_images/c6bc9352f4466627.jpg	0	0	0	0

Separating images into red, green and blue pixel values

Creating a plot of the distribution of average color channel values for each RGB value in the training set images.



It is not surprising that green has a higher median RGB value around 165 and is more pronounced since these images contain mostly healthy leaves with scabs. These values could possibly range from 0 to 255 but no values appear to be above 220. The green values are highly skewed to the left. Red values are slightly more distributed than blue but both red and blue have fairly normal distributions. Next, we will change the images to a numpy array before being input into various convolutional neural networks.

Final inspection of the data

```
(10240, 299, 299, 3)
```

```
(2559, 299, 299, 3)
```

```
(10240, 6)
```

```
(2559, 6)
```

The shape of the train set and test set with their respective labels are as they should be.

The Keras convolutional neural network (CNN)

This CNN consists of three convolutional layers, a dense hidden layer and an output layer. Relu activation is utiilized with an 'adam' optimizer. Each convolutional layer has a kernel size of 3 and is followed by batch normalization, max pooling of 2 and a drop out layer with a value of 0.5 since padding was not utilized. The model was flattened after the last convolutional layer.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 224, 224, 8)       80
_________________________________________________________________
batch_normalization (BatchNo (None, 224, 224, 8)       32
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 112, 112, 8)       0
_________________________________________________________________
dropout (Dropout)            (None, 112, 112, 8)       0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 110, 110, 16)      1168
_________________________________________________________________
batch_normalization_1 (Batch (None, 110, 110, 16)      64
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 55, 55, 16)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 55, 55, 16)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 53, 53, 32)        4640
_________________________________________________________________
batch_normalization_2 (Batch (None, 53, 53, 32)        128
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 26, 26, 32)        0
_________________________________________________________________
dropout_2 (Dropout)          (None, 26, 26, 32)        0
_________________________________________________________________
flatten (Flatten)            (None, 21632)             0
_________________________________________________________________
dense (Dense)                (None, 64)                1384512
_________________________________________________________________
dense_1 (Dense)              (None, 6)                 390
=================================================================
Total params: 1,391,014
Trainable params: 1,390,902
Non-trainable params: 112
_________________________________________________________________
None
```

```
Epoch 1/3
410/410 [==============================] - 5s 12ms/step - loss: 1.788
6 - accuracy: 0.2832 - val_loss: 1.8561 - val_accuracy: 0.3091
Epoch 2/3
410/410 [==============================] - 4s 10ms/step - loss: 1.674
6 - accuracy: 0.3491 - val_loss: 1.8833 - val_accuracy: 0.3037
Epoch 3/3
410/410 [==============================] - 4s 10ms/step - loss: 1.634
0 - accuracy: 0.3711 - val_loss: 1.7711 - val_accuracy: 0.3369
```

```
<tensorflow.python.keras.callbacks.History at 0x7f70d00f96d0>
```

```
256/256 [==============================] - 1s 3ms/step - loss: 1.7664
- accuracy: 0.3337
```

```
[1.7664055824279785, 0.33372411131858826]
```

The validation accuracy is very low because the convolutional neural network needs more data than we can provide due to memory limitations. Here, we must use a neural network that is already trained on "big data." This means implementing **transfer learning**. Let's start with VGGNet. This network is characterized by its simplicity, using only 3×3 convolutional layers stacked on top of each other in increasing depth. Reducing volume size is handled by max pooling. Limitations are it's ridiculously long time to train and the large netwrok architecture weights. Let's see how it performs with a batch size of 8.

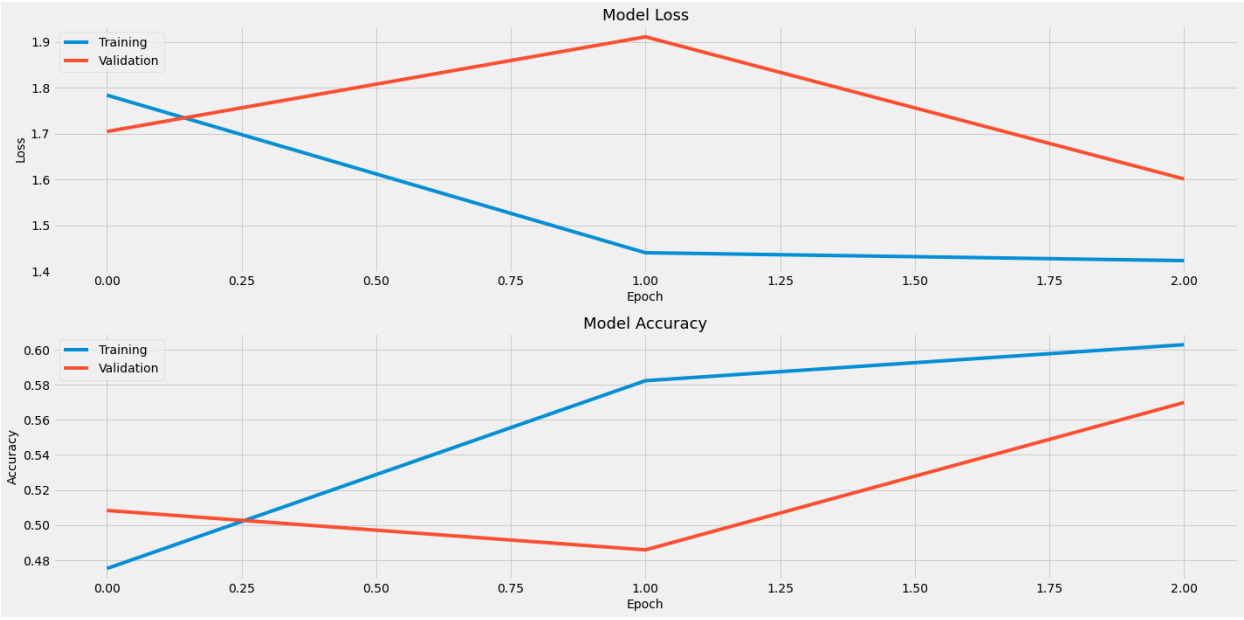
```
Model: "VGG19"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_3 (InputLayer)         [(None, 224, 224, 3)]     0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 224, 224, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 224, 224, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 112, 112, 64)      0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 112, 112, 128)     73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 112, 112, 128)     147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 56, 56, 128)       0
```

```
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 56, 56, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_conv4 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 28, 28, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 28, 28, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_conv4 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 14, 14, 512)       0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv4 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 7, 7, 512)         0
_________________________________________________________________
global_average_pooling2d_3 ( (None, 512)               0
_________________________________________________________________
dense_8 (Dense)              (None, 512)               262656
_________________________________________________________________
dense_9 (Dense)              (None, 6)                 3078
=================================================================
Total params: 20,290,118
Trainable params: 265,734
Non-trainable params: 20,024,384
_________________________________________________________________
```

```
Epoch 1/3
1024/1024 [==============================] - 435s 423ms/step - loss:
2.1984 - accuracy: 0.3931 - precision: 0.4513 - recall: 0.2993 - val_
loss: 1.7046 - val_accuracy: 0.5083 - val_precision: 0.5867 - val_rec
all: 0.3956
Epoch 2/3
1024/1024 [==============================] - 429s 419ms/step - loss:
1.4120 - accuracy: 0.5851 - precision: 0.6678 - recall: 0.4879 - val_
loss: 1.9113 - val_accuracy: 0.4858 - val_precision: 0.5544 - val_rec
all: 0.4521
Epoch 3/3
1024/1024 [==============================] - 429s 419ms/step - loss:
1.3614 - accuracy: 0.6149 - precision: 0.6860 - recall: 0.5389 - val_
loss: 1.6011 - val_accuracy: 0.5698 - val_precision: 0.6479 - val_rec
all: 0.4839
```

```
80/80 [==============================] - 102s 1s/step - loss: 1.3396
- accuracy: 0.5893 - precision: 0.6759 - recall: 0.5182

[1.339553952217102, 0.5892927050590515, 0.6759214997291565, 0.5181651
11541748]
```



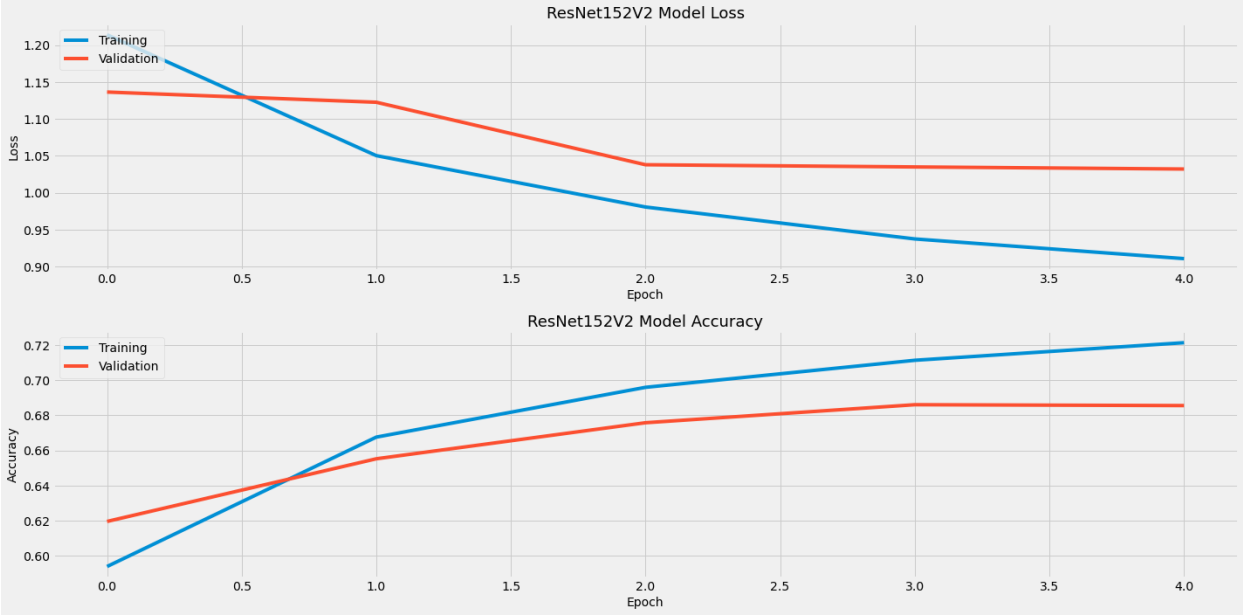
This looks pretty mediocre as more epochs appear to be needed. This is particulary cumbersome especially since VGG19 takes a painfully long time to train. I am searching for better performance from some other transfer learining networks. Let's take a look at ResNet. ResNet relies on micro-architecture modules called network-in-network architectures that are much deeper than VGG19 and the model size is smaller. ResNet stands for the use of residual modules that can train extremely deep networks with stochastic gradient descent. Here, we will try ResNet152V2 as our model.

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/resnet/resnet152v2_weights_tf_dim_ordering_tf_kernels_notop.h5 (https://storage.googleapis.com/tensorflow/keras-applications/resnet/resnet152v2_weights_tf_dim_ordering_tf_kernels_notop.h5)
234553344/234545216 [==============================] - 5s 0us/step
```

We are not showing the summary of the ResNet model as it is monstrously large. Total params: 59,383,814 - Trainable params: 1,052,166 leaving the balance as non-trainable.

There will be 5 epochs and a batch size of 32 for the ResNet152V2 model.

```
Epoch 1/5
256/256 [==============================] - 424s 2s/step - loss: 1.213
9 - accuracy: 0.5940 - precision: 0.7827 - recall: 0.3628 - val_loss:
1.1364 - val_accuracy: 0.6196 - val_precision: 0.7765 - val_recall: 0
.4328
Epoch 2/5
256/256 [==============================] - 423s 2s/step - loss: 1.050
3 - accuracy: 0.6676 - precision: 0.8027 - recall: 0.4937 - val_loss:
1.1225 - val_accuracy: 0.6553 - val_precision: 0.7981 - val_recall: 0
.4570
Epoch 3/5
256/256 [==============================] - 415s 2s/step - loss: 0.980
7 - accuracy: 0.6959 - precision: 0.8159 - recall: 0.5463 - val_loss:
1.0379 - val_accuracy: 0.6758 - val_precision: 0.8025 - val_recall: 0
.5206
Epoch 4/5
256/256 [==============================] - 426s 2s/step - loss: 0.937
5 - accuracy: 0.7113 - precision: 0.8230 - recall: 0.5788 - val_loss:
1.0350 - val_accuracy: 0.6860 - val_precision: 0.8017 - val_recall: 0
.5345
Epoch 5/5
256/256 [==============================] - 425s 2s/step - loss: 0.910
8 - accuracy: 0.7213 - precision: 0.8254 - recall: 0.6007 - val_loss:
1.0321 - val_accuracy: 0.6855 - val_precision: 0.7939 - val_recall: 0
.5627
```



5 epochs during training may be ideal from looking at the graph above for ResNet152V2. But can we get better accuracy from another model?

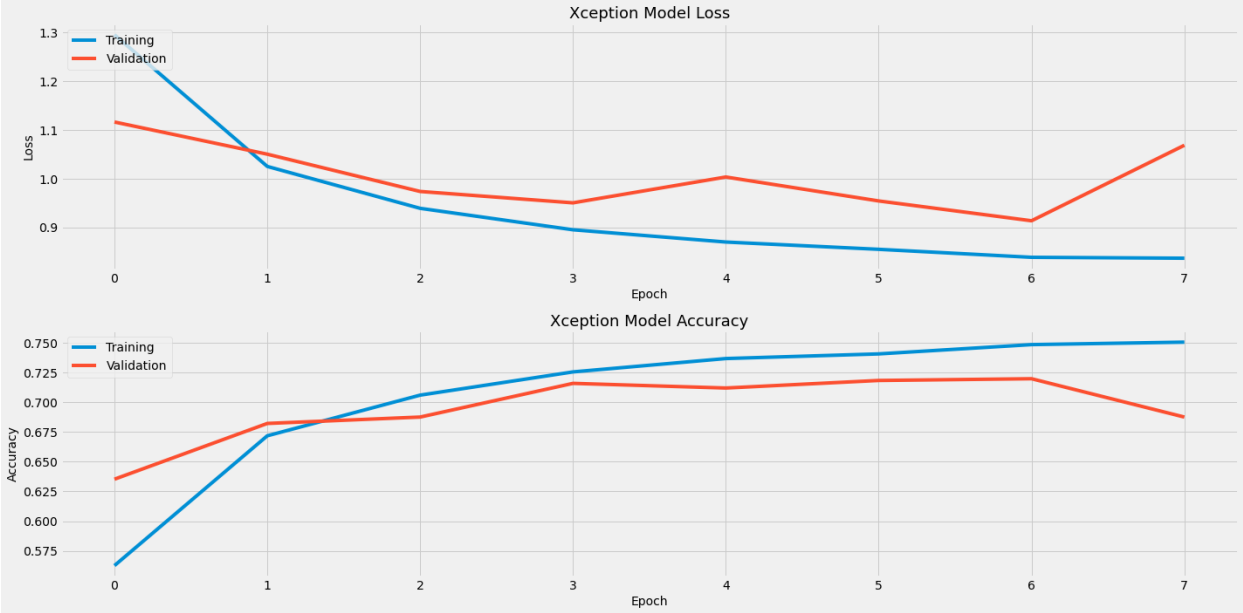
```
80/80 [==============================] - 105s 1s/step - loss: 0.9188
- accuracy: 0.7093 - precision: 0.8122 - recall: 0.5967
```

```
[0.9188497066497803, 0.709261417388916, 0.8121877908706665, 0.5966972
708702087]
```

This is better than the VGG19 model, especially on the validation accuracy. The earlystopping callbacks make a huge difference to the tune of about 25 percent! We are not showing the training results because of the computation time. Let's see if we can do better? Time for Xception, which was developed by the creator of Keras, François Chollet. Xception is an extension of the Inception architecture which replaces the standard Inception modules with depthwise separable convolutions. In case you are wondering why I have not mentioned Inception yet, that is because I could be saving the best for last!

```
Model: "Xception"
__________________________________________________________________________
_______________________________
Layer (type)                    Output Shape         Param #     Conn
ected to
==========================================================================
==============================
input_1 (InputLayer)            [(None, 299, 299, 3) 0
__________________________________________________________________________
_______________________________
block1_conv1 (Conv2D)           (None, 149, 149, 32) 864         inpu
t_1[0][0]
__________________________________________________________________________
_______________________________
block1_conv1_bn (BatchNormaliza (None, 149, 149, 32) 128         bloc
k1_conv1[0][0]
__________________________________________________________________________
_______________________________
block1_conv1_act (Activation)   (None, 149, 149, 32) 0           bloc
```

```
Epoch 1/8
256/256 [==============================] - 46s 102ms/step - loss: 1.4
784 - accuracy: 0.4725 - precision: 0.7602 - recall: 0.1568 - val_los
s: 1.1161 - val_accuracy: 0.6353 - val_precision: 0.7956 - val_recall
: 0.4413
Epoch 2/8
256/256 [==============================] - 25s 98ms/step - loss: 1.03
75 - accuracy: 0.6716 - precision: 0.8216 - recall: 0.4853 - val_loss
: 1.0501 - val_accuracy: 0.6821 - val_precision: 0.8405 - val_recall:
0.4816
Epoch 3/8
256/256 [==============================] - 25s 97ms/step - loss: 0.95
14 - accuracy: 0.7081 - precision: 0.8334 - recall: 0.5492 - val_loss
: 0.9738 - val_accuracy: 0.6875 - val_precision: 0.7956 - val_recall:
0.5685
Epoch 4/8
256/256 [==============================] - 25s 98ms/step - loss: 0.88
92 - accuracy: 0.7307 - precision: 0.8364 - recall: 0.5987 - val_loss
: 0.9505 - val_accuracy: 0.7158 - val_precision: 0.8283 - val_recall:
0.5856
Epoch 5/8
256/256 [==============================] - 25s 98ms/step - loss: 0.86
30 - accuracy: 0.7337 - precision: 0.8413 - recall: 0.6138 - val_loss
: 1.0034 - val_accuracy: 0.7119 - val_precision: 0.8323 - val_recall:
0.5650
Epoch 6/8
256/256 [==============================] - 25s 98ms/step - loss: 0.84
76 - accuracy: 0.7417 - precision: 0.8402 - recall: 0.6219 - val_loss
: 0.9544 - val_accuracy: 0.7183 - val_precision: 0.8080 - val_recall:
0.6053
Epoch 7/8
256/256 [==============================] - 25s 99ms/step - loss: 0.82
71 - accuracy: 0.7562 - precision: 0.8424 - recall: 0.6430 - val_loss
: 0.9137 - val_accuracy: 0.7197 - val_precision: 0.8122 - val_recall:
0.6183
Epoch 8/8
256/256 [==============================] - 25s 98ms/step - loss: 0.83
75 - accuracy: 0.7517 - precision: 0.8381 - recall: 0.6405 - val_loss
: 1.0684 - val_accuracy: 0.6875 - val_precision: 0.7794 - val_recall:
0.5874
```



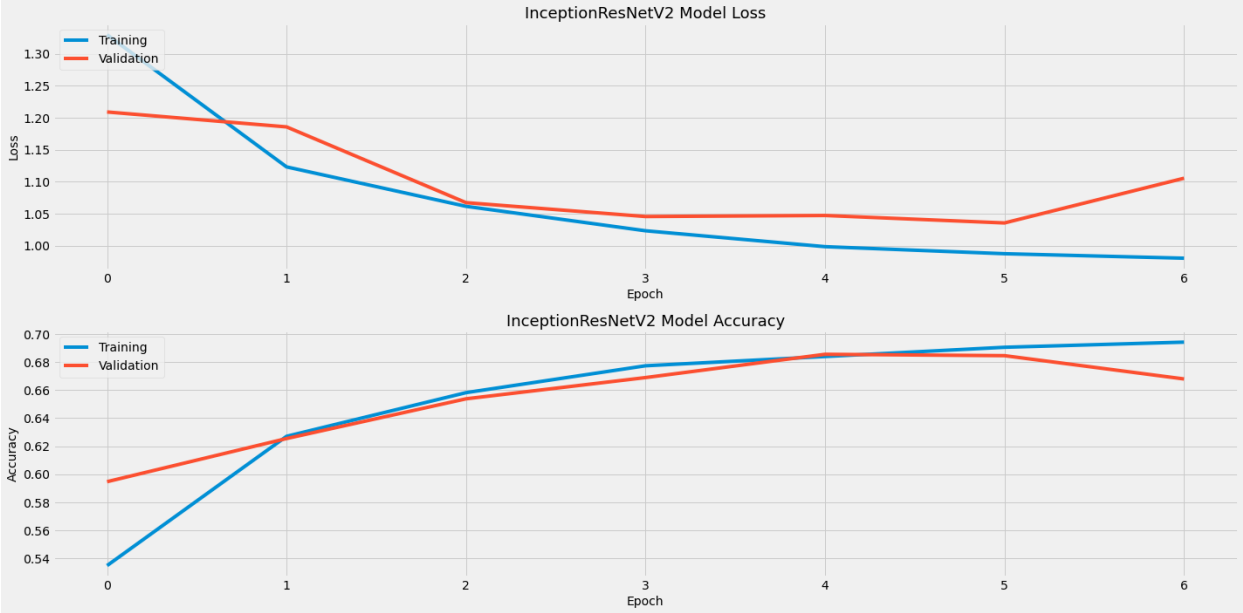

```
80/80 [==============================] - 8s 85ms/step - loss: 0.9698
- accuracy: 0.6956 - precision: 0.7801 - recall: 0.6000

[0.9697695970535278, 0.695584237575531, 0.7800572514533997, 0.6000000
238418579]
```

The Xception model was just a little worse than the ResNet model with an accuracy of 0.696 with 32 for a batch size. The recall was better at 0.6 and the precision value of 0.78 was just a little under the 0.812 value for the ResNet model. The accuracy decreased after 6 epochs unlike the ResNet model where the validation accuracy hit a plateau after 4 epochs. As many things are in data science as in life, adding two good things together generally makes it better. For our last model, we will try InceptionResNetV2 which is a convolutional neural architecture that builds on the Inception family of architectures but incorporates residual connections replacing the filter concatenation stage of the Inception architecture. Fortunately, we can keep the 299 x 299 image size and give it a go.

Again, this model summary is way too large to depict, so we will display a summary of 55,126,758 total parameters, almost as large as the ResNet model we imported. We will train this model with 7 epochs to ensure high accuracy.

```
Epoch 1/7
256/256 [==============================] - 70s 163ms/step - loss: 1.4
661 - accuracy: 0.4747 - precision: 0.7414 - recall: 0.1788 - val_los
s: 1.2089 - val_accuracy: 0.5947 - val_precision: 0.7513 - val_recall
: 0.3884
Epoch 2/7
256/256 [==============================] - 38s 148ms/step - loss: 1.1
423 - accuracy: 0.6216 - precision: 0.7803 - recall: 0.4345 - val_los
s: 1.1856 - val_accuracy: 0.6255 - val_precision: 0.7791 - val_recall
: 0.4014
Epoch 3/7
256/256 [==============================] - 38s 148ms/step - loss: 1.0
622 - accuracy: 0.6576 - precision: 0.7842 - recall: 0.4987 - val_los
s: 1.0672 - val_accuracy: 0.6538 - val_precision: 0.7729 - val_recall
: 0.5246
Epoch 4/7
256/256 [==============================] - 38s 148ms/step - loss: 1.0
245 - accuracy: 0.6732 - precision: 0.7893 - recall: 0.5234 - val_los
s: 1.0455 - val_accuracy: 0.6689 - val_precision: 0.7813 - val_recall
: 0.5556
Epoch 5/7
256/256 [==============================] - 38s 147ms/step - loss: 0.9
833 - accuracy: 0.6871 - precision: 0.7974 - recall: 0.5649 - val_los
s: 1.0471 - val_accuracy: 0.6855 - val_precision: 0.7918 - val_recall
: 0.5556
Epoch 6/7
256/256 [==============================] - 38s 147ms/step - loss: 0.9
601 - accuracy: 0.6958 - precision: 0.7977 - recall: 0.5809 - val_los
s: 1.0355 - val_accuracy: 0.6846 - val_precision: 0.7833 - val_recall
: 0.5829
Epoch 7/7
256/256 [==============================] - 38s 147ms/step - loss: 0.9
823 - accuracy: 0.6959 - precision: 0.7925 - recall: 0.5777 - val_los
s: 1.1053 - val_accuracy: 0.6680 - val_precision: 0.7563 - val_recall
: 0.5909
```




```
80/80 [==============================] - 14s 128ms/step - loss: 0.974
4 - accuracy: 0.6921 - precision: 0.7737 - recall: 0.6187

[0.9744309782981873,
 0.6920672059059143,
 0.7737494111061096,
 0.6187155842781067]
```

An accuracy of 0.692 on new data is pretty good. It is just below the test accuracy of the Xception model. The recall value is higher at 0.619 but the precision score is lower than the Xception model at 0.774 but it is the ResNet152V2 that had an accuracy over 70%! The ResNet152V2 model also had the highest precision score of 0.81 and the lower recall score of just under 0.6 so it would depend on which metric you are most interested in. However, these models perform very similarly on the test data. It was the VGG19 model that performed about 10% lower on accuracy, precision and recall.

Conclusion

While the accuracy of the CNN was not excellent, there may be a number of reasons for this and a number of methods that can be used to improve the model.

The first issue is that there is not enough RAM available to read all of the images provided in the initial Kaggle training set into Google Colab. Neural networks need a large amount of data fed into them to be effective and the initial Keras model did not have sufficient data to perform well.

Additionally, leaf color and leaf morphology, non-uniform image background, and different light illumination during imaging may require that the images be processed in a more thorough way so that some uniformity with the original clarity can be achieved. This may produce a model that could be deployed on larger datasets increasing accuracy for entire orchards.

The last modification may very well be the use of PyTorch as PyTorch is more advanced with some of the object-oriented programming functions that can be utilized. While the accuracy is not excellent, there are many methods that can be employed for improvement of the CNN model.

Transfer learning did lead to a pretty good model as the ResNet152V2 model outperformed InceptionResNetV2, VGG19 and Xception with just over 70% accuracy. Xception and InceptionResNetV2 produced an accuracy of just under 70% on new data in the form of a test set created from the original training set provided by Kaggle that was not used for training. This was due to the fact that the original test set only consisted of three images. ResNet152V2, InceptionResNetV2 and Xception were very close in accuracy, precision and recall. The validation accuracy was very close to the training accuracy for the InceptionResNetV2 through 5 epochs.

This model can be used to identify the five most common classes of disease in apple leaves against healthy leaves with no evidence of disease. This will reduce labor costs and increase production time which will both lower expenses and increase efficiency leading to higher profits. This model could also improve apple tree health and fruit health as well. All of the transfer learning models used were trained on ImageNet which an image database organized according to the WordNet hierarchy (currently only the nouns), in which each node of the hierarchy is depicted by hundreds and thousands of images. This means that these models could also be used in other fields of plant pathology to improve plant health all over the world.