PROJECT TITLE

FRUIT CATEGORY PREDICTION

PARTICIPANT NAMES

PILLUTLA, BHAMA KRISHNA
PULLABHOTLA, KAMESWARA
PARIMI, OOHA

OBJECTIVE:

- Use a custom dataset and split it into train, validation and test.
- Predict a given fruit belongs to good or bad category.
- Rank it on a scale of 1 to 5 depending on quality of the fruit.

PROPOSED ARCHITECTURE:

The current model is developed from the scratch and using four-layer convolutional neural network.

<u>PreProcessing:</u>

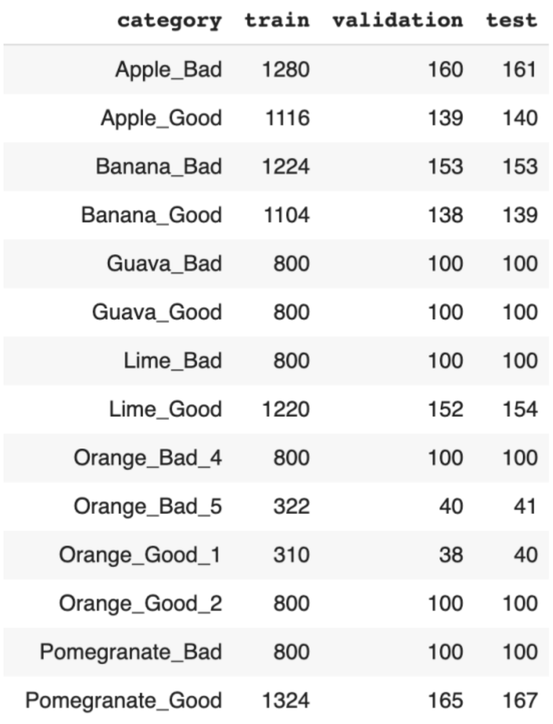
Preprocessed image of size 256 X 256 and reshaped into 45X45X3, normalized the image and provisioned the reshaped image to the convolutional layers.

<u>Dataset Balancing:</u>

We are using a custom dataset for training the model where the base data is picked from the datasets present in IEEE web portal. Later we have added few more images to this dataset in-order to improve the model prediction.

Scaling is been done for the fruit category “Orange” where we have collection sample data for rating the predicted image on a scale of 1 to 4 where **1** - **Best**, **2** - **Good**, **3** - **Bad** and **4** - **Worst** classifications of the fruit.

The dataset is 20% im-balanced as some of the dataset is built for scaling purpose.



category	train	validation	test
Apple_Bad	1280	160	161
Apple_Good	1116	139	140
Banana_Bad	1224	153	153
Banana_Good	1104	138	139
Guava_Bad	800	100	100
Guava_Good	800	100	100
Lime_Bad	800	100	100
Lime_Good	1220	152	154
Orange_Bad_4	800	100	100
Orange_Bad_5	322	40	41
Orange_Good_1	310	38	40
Orange_Good_2	800	100	100
Pomegranate_Bad	800	100	100
Pomegranate_Good	1324	165	167

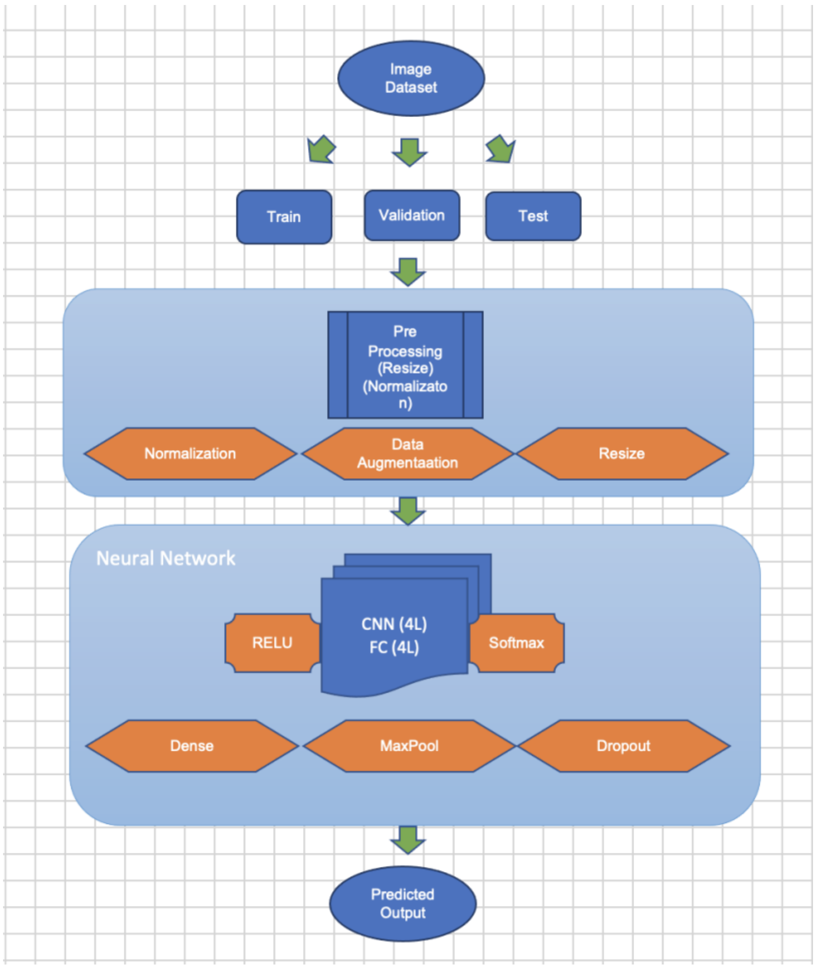
DESCRIPTION:

The current model is a four-layer convolutional network which uses regularization techniques, hyperparameter optimizations, activation functions and data visualizations.

Steps:

- Preprocessing custom dataset and splitting the dataset into train, validation, and test datasets.
- Resize images from 256 X 256 to 45 X 45 X 3 and normalizing them.
- Passing through four-layer neural network having kernel of 3 X 3 and stride 1.
- Applied MaxPool after traversing through two hidden layers which uses RELU.
- Used four forward connected layers where we applied softmax as activation function in the final layer.

Project Flow:



Proposed Model:

Analysis:

Approach 1:

```
model_cnn.add(Conv2D(512, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=(45, 45, 3)))

model_cnn.add(Flatten())
model_cnn.add(Dense(14, activation='softmax'))
model_cnn.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=tensorflow.keras.optimizers.Adam(),
              metrics=['accuracy'])
```

Observation:

Got shape incompatible errors so, adding flatten layer resolved it. Achieved **98**% on train set accuracy and **89%** of validation set accuracy. Model overfitted.

Approach 2:

```
model_cnn.add(Conv2D(512, kernel_size=(2, 2),
                 activation='relu',
                 input_shape=(45, 45, 3)))

model_cnn.add(Flatten())
model_cnn.add(Dense(14, activation='softmax'))
model_cnn.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=tensorflow.keras.optimizers.Adam(),
              metrics=['accuracy'])
```

Observation:

Accuracy: **99%** on train set and **92%** on validation set. Still model is overfitted.
Even after using RMSprop both train and validation accuracies remained same. To overcome overfitting on train data, now adding dropout.

Approach 3:

With

```
model_cnn.add(Dropout(0.5))
```

Observation:

Adding a dropout before flattening layers improved validation set accuracy to **93%** and train set to **98%**.
Adding a dropout layer after flatten layer reduced train set accuracy to **96%** and increased validation set to **92%**. To improve the validation accuracy, now adding another Conv2D layer.

Approach 4:

```
model_cnn = Sequential()

model_cnn.add(Conv2D(512, kernel_size=(2, 2),
                 activation='relu',
                 input_shape=(45, 45, 3)))
model_cnn.add(Conv2D(256,kernel_size=(2, 2), activation='relu'))
model_cnn.add(Dropout(0.5))


model_cnn.add(Flatten())
model_cnn.add(Dropout(0.5))
model_cnn.add(Dense(14, activation='softmax'))

model_cnn.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=tensorflow.keras.optimizers.RMSprop(),
              metrics=['accuracy'])
```

Observation:

Added another Conv2D layers and changed optimizer to RMSProp now, accuracy for train set is **96%** and validation set is **93%**.
With Adam train accuracy is **97%** and validation set is **93%**.

Approach 5:

```
model_cnn.add(Conv2D(512, kernel_size=(2, 2),
                 activation='relu',
                 input_shape=(45, 45, 3)))
model_cnn.add(Conv2D(256,kernel_size=(2, 2), activation='relu'))
model_cnn.add(MaxPooling2D(pool_size=(2, 2)))
model_cnn.add(Dropout(0.5))


model_cnn.add(Flatten())
model_cnn.add(Dropout(0.5))
model_cnn.add(Dense(14, activation='softmax'))

model_cnn.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=tensorflow.keras.optimizers.Adam(),
              metrics=['accuracy'])
```

Observation:

MAXPOOL layer made loss reduction more consistent and brought down the accuracy of train set and validation set **96%** and **95**% respectively. Also, the prediction results are good.

Approach 6:

```
model_cnn.add(Conv2D(128, kernel_size=(2, 2),
                 activation='relu',
                 input_shape=(45, 45, 3)))
model_cnn.add(Conv2D(64,kernel_size=(2, 2), activation='relu'))
model_cnn.add(MaxPooling2D(pool_size=(2, 2)))
model_cnn.add(Dropout(0.5))


model_cnn.add(Flatten())

model_cnn.add(Dropout(0.5))
model_cnn.add(Dense(14, activation='softmax'))

model_cnn.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=tensorflow.keras.optimizers.Adam(),
              metrics=['accuracy'])
```

Observations:

512 Filter size took more time for iterations so, changed it to 128,64, accuracy for train and validation set is **96%** and **95%**. Also, changed kernel size to (3,3) accuracies of train and validation sets are **95%** and **94%**. Let’s focus now on increasing accuracy and reducing epoch time.

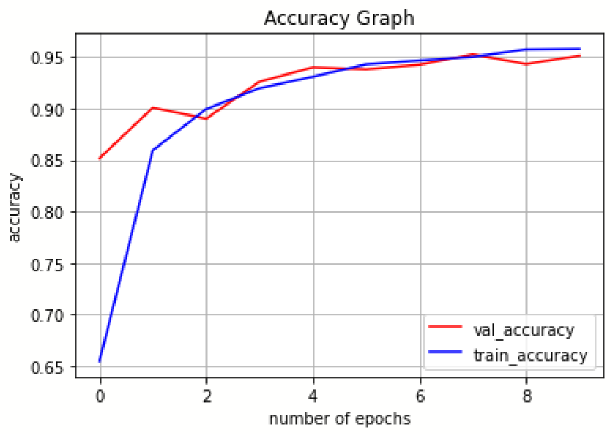
Reduced filter size to 64 and 32 for input layer, accuracy for train set is **96%** and validation set is **94%**.

Approach 7:

```
model_cnn.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=(45, 45, 3)))
model_cnn.add(Conv2D(64,kernel_size=(3, 3), activation='relu'))
model_cnn.add(MaxPooling2D(pool_size=(2, 2)))
model_cnn.add(Dropout(0.5))
model_cnn.add(Conv2D(32, kernel_size=(3, 3), activation='relu'))
model_cnn.add(Flatten())
model_cnn.add(Dropout(0.5))
model_cnn.add(Dense(14, activation='softmax'))
model_cnn.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=tensorflow.keras.optimizers.Adam(),
              metrics=['accuracy'])
```

Observation:

Added Conv2D layer for the model both train and validation accuracies are **95%**. Pooling layer after Conv2D layer did not help with accuracy.



Approach 8:

```
model_cnn.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=(45, 45, 3)))
model_cnn.add(Conv2D(64,kernel_size=(3, 3), activation='relu'))
model_cnn.add(MaxPooling2D(pool_size=(2, 2)))
model_cnn.add(Dropout(0.5))
model_cnn.add(Conv2D(32, kernel_size=(3, 3), activation='relu'))



model_cnn.add(Flatten())
model_cnn.add(Dropout(0.5))
model_cnn.add(Dense(128, activation='relu'))
model_cnn.add(Dense(14, activation='softmax'))

model_cnn.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=tensorflow.keras.optimizers.Adam(),
              metrics=['accuracy'])
```

Observation:

Adding a dense layer before the final layer increased train and validation accuracies to **97%** and **96%**.

Approach 9:

```
model_cnn.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=(45, 45, 3)))

model_cnn.add(Conv2D(64,kernel_size=(3, 3), activation='relu'))
model_cnn.add(MaxPooling2D(pool_size=(2, 2)))
model_cnn.add(Dropout(0.5))

model_cnn.add(Conv2D(32, kernel_size=(3, 3), activation='relu'))
model_cnn.add(MaxPooling2D(pool_size=(2, 2)))

model_cnn.add(Conv2D(32, kernel_size=(3, 3), activation='relu'))
model_cnn.add(MaxPooling2D(pool_size=(2, 2)))


model_cnn.add(Flatten())
model_cnn.add(Dropout(0.5))
model_cnn.add(Dense(128, activation='relu'))
model_cnn.add(Dense(14, activation='softmax'))
```

Observation:

Adding pooling layer to third Conv2D layer reduced **90%** for train and **91%** for validation. Increasing third and fourth layer filter size to 64 reduced validation set accuracy and increased train set accuracy.

So, changing filter size for fourth layer to 128 and third layer to 64 equaled both train and test set accuracy to **95%**. Since observed dense layer improved accuracy in our previous step, added another dense layer.

Approach 10:

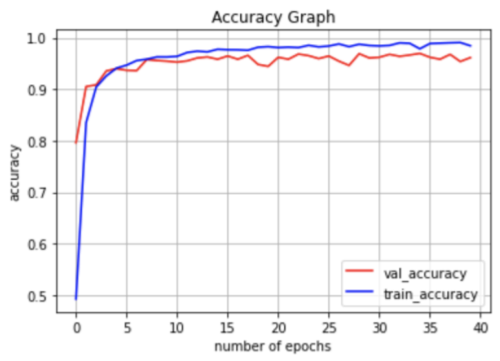
```
model_cnn.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=(45, 45, 3)))

model_cnn.add(Conv2D(64,kernel_size=(3, 3), activation='relu'))
model_cnn.add(MaxPooling2D(pool_size=(2, 2)))
model_cnn.add(Dropout(0.5))

model_cnn.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
model_cnn.add(MaxPooling2D(pool_size=(2, 2)))
model_cnn.add(Dropout(0.5))

model_cnn.add(Conv2D(128, kernel_size=(3, 3), activation='relu'))
model_cnn.add(MaxPooling2D(pool_size=(2, 2)))
model_cnn.add(Dropout(0.5))


model_cnn.add(Flatten())
model_cnn.add(Dense(512*4, activation='relu'))
model_cnn.add(Dropout(0.5))
model_cnn.add(Dense(1024, activation='relu'))
model_cnn.add(Dense(128, activation='relu'))
model_cnn.add(Dense(14, activation='softmax'))
```

Observation:

Added multiple dense layers and MAXPOOL and dropout for Conv2D layers but these reduced the train accuracy and increased validation accuracy. Later removing MAXPOOL and dropout layers settled down the accuracy graph and train and validation losses are appeared to be reducing with every iteration. Though train accuracy is **98%** the validation accuracy is less with **96%**, model generalized well with the test images. So, our proposed model is four Conv2D layers with optimizer Adam. We also believed playing with more Conv2D layers and kernel sizes would also improve the accuracy for both the sets.

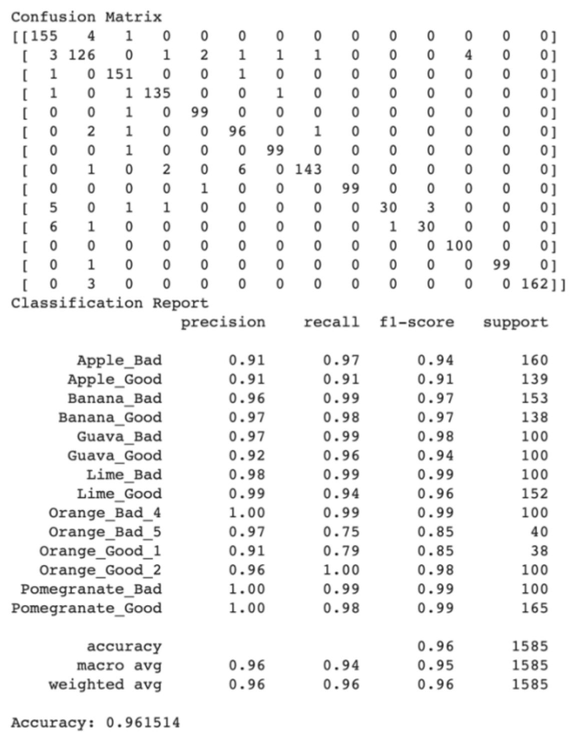
FINAL RESULT:

Jittering is observed as the validation dataset is smaller for few of the classes. Achieved an overall accuracy of **98%** on the training, **96%** on the validation and **97%** on the test datasets.

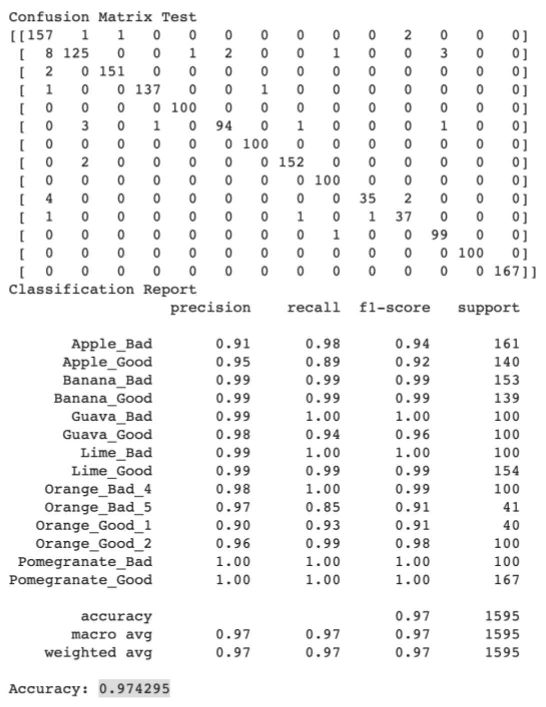


METRICS:

VALIDATION ACCURACY


```
Confusion Matrix
[[155   4   1   0   0   0   0   0   0   0   0   0   0   0]
 [  3 126   0   1   2   1   1   1   0   0   0   4   0   0]
 [  1   0 151   0   0   1   0   0   0   0   0   0   0   0]
 [  1   0   1 135   0   0   1   0   0   0   0   0   0   0]
 [  0   0   1   0  99   0   0   0   0   0   0   0   0   0]
 [  0   2   1   0   0  96   0   1   0   0   0   0   0   0]
 [  0   0   1   0   0   0  99   0   0   0   0   0   0   0]
 [  0   1   0   2   0   6   0 143   0   0   0   0   0   0]
 [  0   0   0   0   1   0   0   0  99   0   0   0   0   0]
 [  5   0   1   1   0   0   0   0   0  30   3   0   0   0]
 [  6   1   0   0   0   0   0   0   0   1  30   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0 100   0   0]
 [  0   1   0   0   0   0   0   0   0   0   0   0  99   0]
 [  0   3   0   0   0   0   0   0   0   0   0   0   0 162]]
Classification Report
                  precision    recall  f1-score   support

        Apple_Bad       0.91      0.97      0.94       160
       Apple_Good       0.91      0.91      0.91       139
       Banana_Bad       0.96      0.99      0.97       153
      Banana_Good       0.97      0.98      0.97       138
        Guava_Bad       0.97      0.99      0.98       100
       Guava_Good       0.92      0.96      0.94       100
         Lime_Bad       0.98      0.99      0.99       100
        Lime_Good       0.99      0.94      0.96       152
     Orange_Bad_4       1.00      0.99      0.99       100
     Orange_Bad_5       0.97      0.75      0.85        40
    Orange_Good_1       0.91      0.79      0.85        38
    Orange_Good_2       0.96      1.00      0.98       100
  Pomegranate_Bad       1.00      0.99      0.99       100
 Pomegranate_Good       1.00      0.98      0.99       165

         accuracy                           0.96      1585
        macro avg       0.96      0.94      0.95      1585
     weighted avg       0.96      0.96      0.96      1585

Accuracy: 0.961514
```

TEST ACCURACY


```
Confusion Matrix Test
[[157   1   1   0   0   0   0   0   0   0   2   0   0   0]
 [  8 125   0   0   1   2   0   0   1   0   0   3   0   0]
 [  2   0 151   0   0   0   0   0   0   0   0   0   0   0]
 [  1   0   0 137   0   0   1   0   0   0   0   0   0   0]
 [  0   0   0   0 100   0   0   0   0   0   0   0   0   0]
 [  0   3   0   1   0  94   0   1   0   0   0   1   0   0]
 [  0   0   0   0   0   0 100   0   0   0   0   0   0   0]
 [  0   2   0   0   0   0   0 152   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0 100   0   0   0   0   0]
 [  4   0   0   0   0   0   0   0   0  35   2   0   0   0]
 [  1   0   0   0   0   0   0   1   0   1  37   0   0   0]
 [  0   0   0   0   0   0   0   0   1   0   0  99   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0 100   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0 167]]
Classification Report
                  precision    recall  f1-score   support

        Apple_Bad       0.91      0.98      0.94       161
       Apple_Good       0.95      0.89      0.92       140
       Banana_Bad       0.99      0.99      0.99       153
      Banana_Good       0.99      0.99      0.99       139
        Guava_Bad       0.99      1.00      1.00       100
       Guava_Good       0.98      0.94      0.96       100
         Lime_Bad       0.99      1.00      1.00       100
        Lime_Good       0.99      0.99      0.99       154
     Orange_Bad_4       0.98      1.00      0.99       100
     Orange_Bad_5       0.97      0.85      0.91        41
    Orange_Good_1       0.90      0.93      0.91        40
    Orange_Good_2       0.96      0.99      0.98       100
  Pomegranate_Bad       1.00      1.00      1.00       100
 Pomegranate_Good       1.00      1.00      1.00       167

         accuracy                           0.97      1595
        macro avg       0.97      0.97      0.97      1595
     weighted avg       0.97      0.97      0.97      1595

Accuracy: 0.974295
```

TESTING ON GOOGLE IMAGES (AS MODEL INPUT):

Also took had done an edge case testing where we have downloaded few images from google and processed the same as a test input to the trained model.

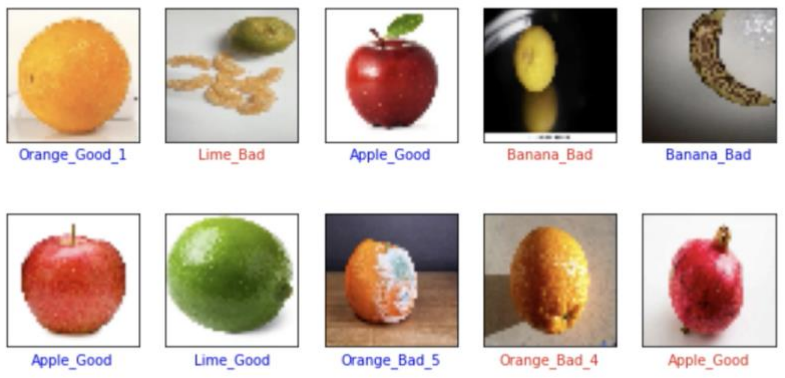
Observation:

Model has performed better by identifying images correctly 80% of times when tested. Approximately 70% of times the model predicted the right label for a scaled image (i.e., Oranges).

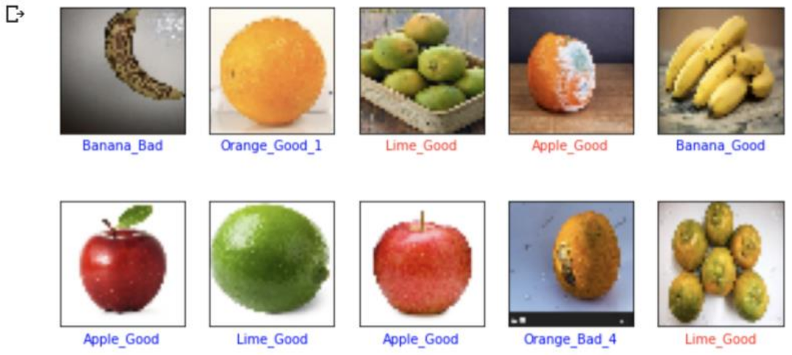
It is also observed that the model can predict more effectively by leveraging the training dataset.

RESULT ON PREDICTING GOOGLE IMAGES:

RUN 1: First Test



RUN 2: Reset GoogleColab and Re-Trained the model



Note:

The above images are not the ones included in the training/validation/test dataset. They were directly downloaded from **GOOGLE** and provided as a test input to the model for prediction.

CONCLUSION:

Applied transfer learning techniques and CNN with multiple layers on the given datasets and the as the current dataset is having skewness with respect to the data, we have chosen to develop a model from scratch by applying the learnings that were learnt from the ones that are previously developed.

TECHNICAL DISCUSSIONS

Team Member 1: (Kameswara Pullabhotla)

Model 1:

Processed image of size 256 X 256 and converted it to tensor. Split the dataset into train, validation and test. Used transferred learning technique with **RESNET18** model.

[0.485, 0.456, 0.406], [0.229, 0.224, 0.225]

Applied data augmentation techniques on the training dataset using data transforms as the code is build using PyTorch. Transforms for validation and testing dataset are as follows:

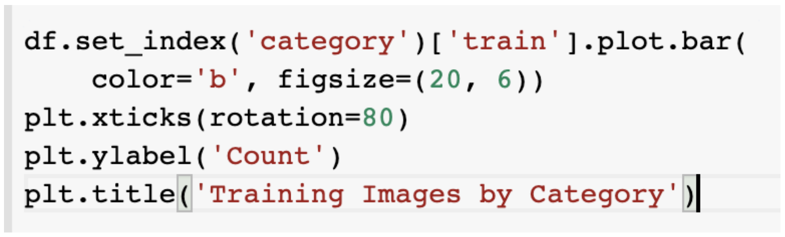
- Resize (Used in case the downloaded image size varies)
- Center Crop - (256 x 256)
- Convert to a tensor
- Normalize with mean and standard deviation

Imported custom dataset using **torch.utils.data** packages and applied the below techniques while training the model:

- Used RES18 for training on the custom dataset.
- Scheduling learning rate using lr_scheduler.
- Saved best model obtained in the training.
- Processed test data using the same model.

Randomly visualized the dataset used

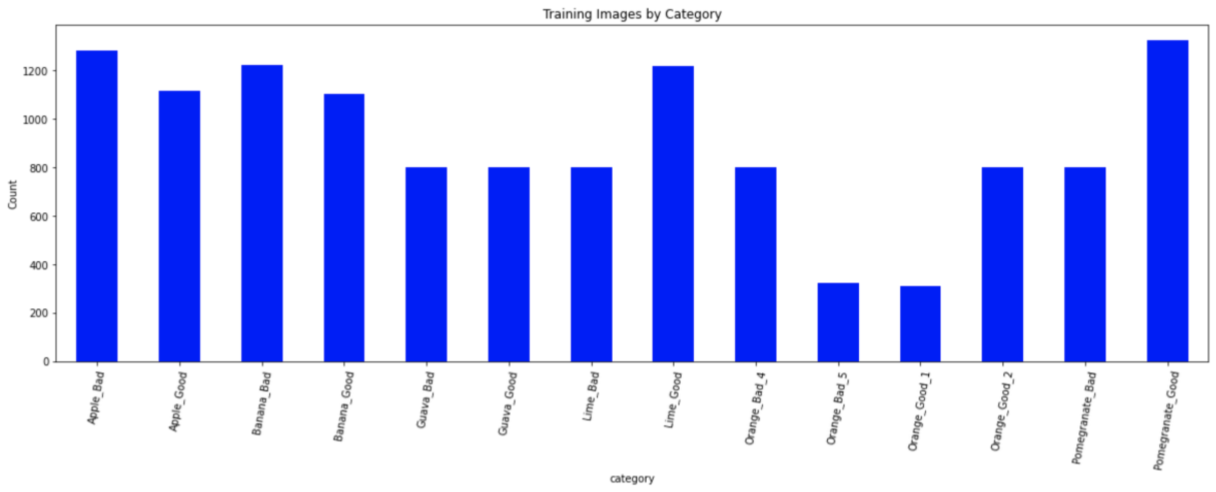
Images per category:



```
df.set_index('category')['train'].plot.bar(
    color='b', figsize=(20, 6))
plt.xticks(rotation=80)
plt.ylabel('Count')
plt.title('Training Images by Category')
```

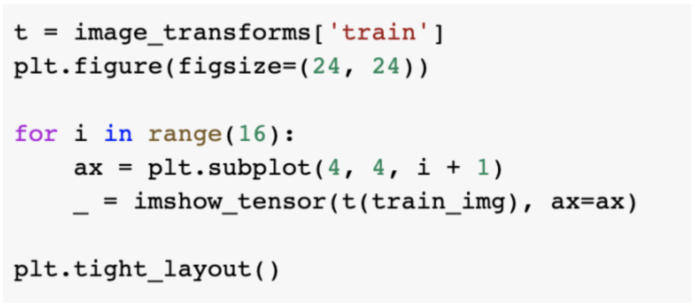
Below table displaces the number of images per class basis:

Here scaling is only considered for the "orange" fruit category.

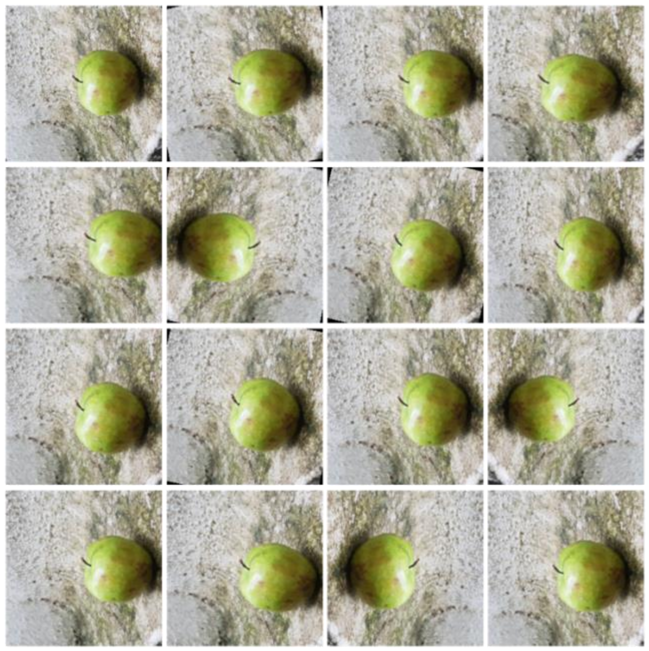


Ran model with the below configuration and recorded the train and validation dataset accuracies.

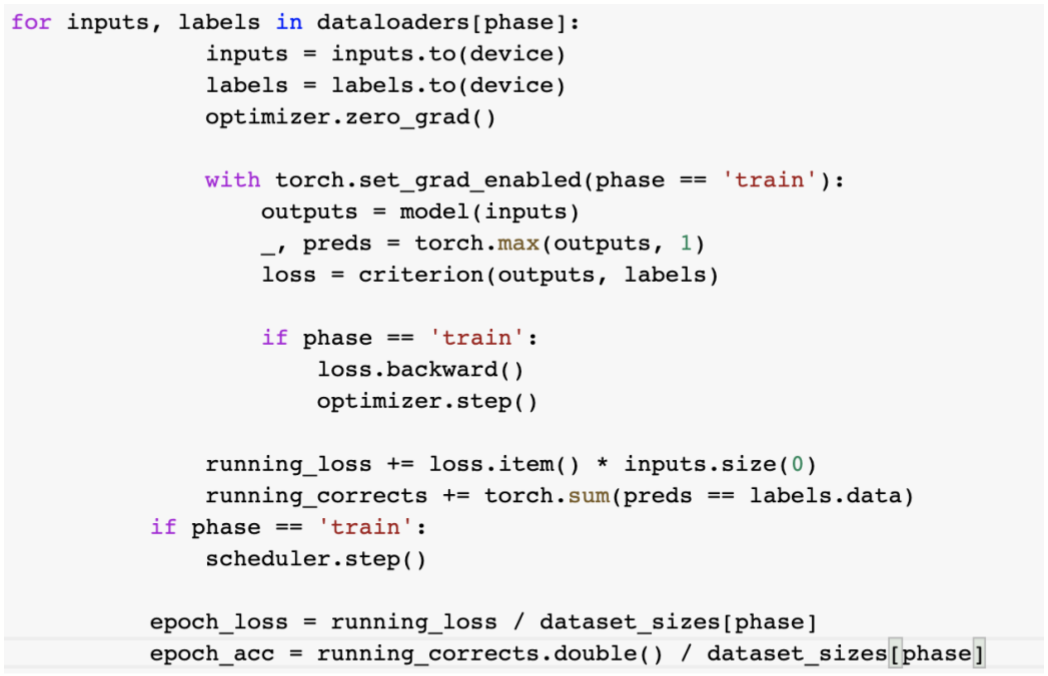
Data Augmentation:



```
t = image_transforms['train']
plt.figure(figsize=(24, 24))

for i in range(16):
    ax = plt.subplot(4, 4, i + 1)
    _ = imshow_tensor(t(train_img), ax=ax)

plt.tight_layout()
```

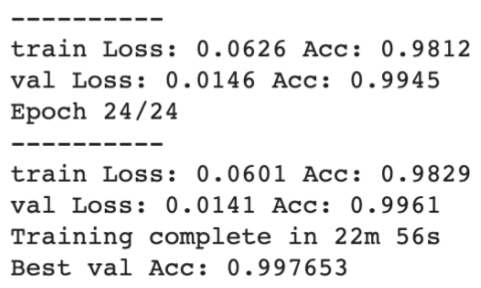


<u>Observation:</u>
Tested accuracy for the above trained model is given below



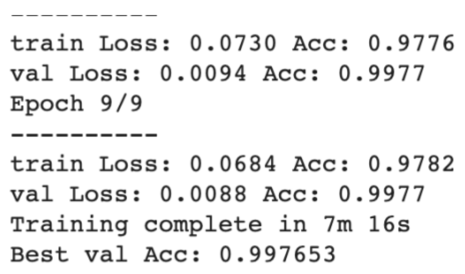
```
for inputs, labels in dataloaders[phase]:
              inputs = inputs.to(device)
              labels = labels.to(device)
              optimizer.zero_grad()

              with torch.set_grad_enabled(phase == 'train'):
                  outputs = model(inputs)
                  _, preds = torch.max(outputs, 1)
                  loss = criterion(outputs, labels)

                  if phase == 'train':
                      loss.backward()
                      optimizer.step()

              running_loss += loss.item() * inputs.size(0)
              running_corrects += torch.sum(preds == labels.data)
          if phase == 'train':
              scheduler.step()

          epoch_loss = running_loss / dataset_sizes[phase]
          epoch_acc = running_corrects.double() / dataset_sizes[phase]
```

```
model_ft = train_model(model_ft, criterion, optimizer_ft,
exp_lr_scheduler,num_epochs=10)
```



```
----------
train Loss: 0.0626 Acc: 0.9812
val Loss: 0.0146 Acc: 0.9945
Epoch 24/24
----------
train Loss: 0.0601 Acc: 0.9829
val Loss: 0.0141 Acc: 0.9961
Training complete in 22m 56s
Best val Acc: 0.997653
```

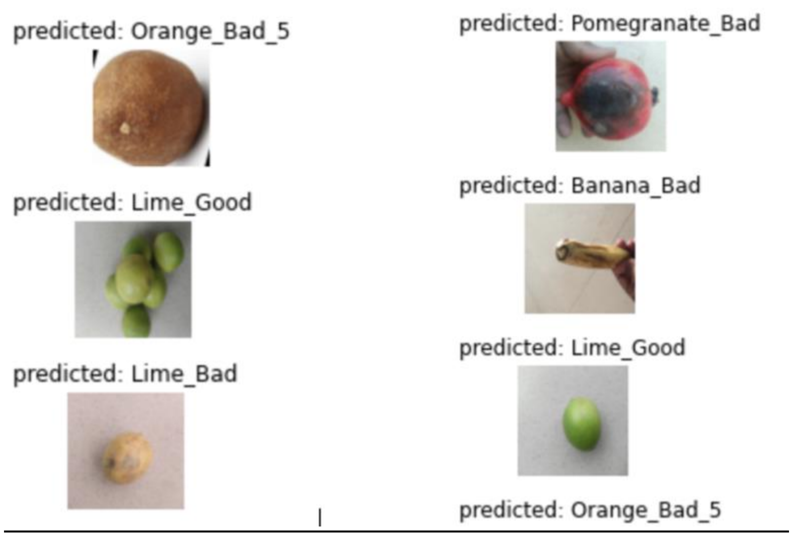
Ran for 10 Epochs and observed the accuracy.



```
----------
train Loss: 0.0730 Acc: 0.9776
val Loss: 0.0094 Acc: 0.9977
Epoch 9/9
----------
train Loss: 0.0684 Acc: 0.9782
val Loss: 0.0088 Acc: 0.9977
Training complete in 7m 16s
Best val Acc: 0.997653
```

Validation accuracy seems higher compared to train as the dataset is having lesser images compared to the train set.

Visualized Images After Training the Model:



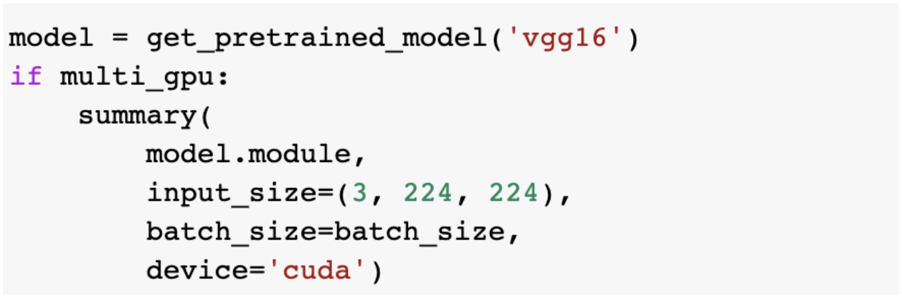
Observed Test Accuracy for the current model is **98.1%**

Model 2:

Tried transferred learning technique with VGG16

Steps Followed:

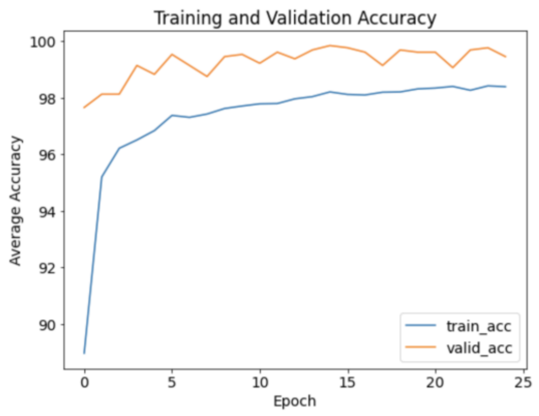
- Load CNN model for training dataset
- Resized image to 224 X 224
- Freeze parameters in lower convolutional layers
- Fine-tune hyper-parameters



```
model = get_pretrained_model('vgg16')
if multi_gpu:
    summary(
        model.module,
        input_size=(3, 224, 224),
        batch_size=batch_size,
        device='cuda')
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1       [128, 64, 224, 224]           1,792
              ReLU-2       [128, 64, 224, 224]               0
            Conv2d-3       [128, 64, 224, 224]          36,928
              ReLU-4       [128, 64, 224, 224]               0
         MaxPool2d-5       [128, 64, 112, 112]               0
            Conv2d-6      [128, 128, 112, 112]          73,856
              ReLU-7      [128, 128, 112, 112]               0
            Conv2d-8      [128, 128, 112, 112]         147,584
              ReLU-9      [128, 128, 112, 112]               0
        MaxPool2d-10        [128, 128, 56, 56]               0
           Conv2d-11        [128, 256, 56, 56]         295,168
             ReLU-12        [128, 256, 56, 56]               0
           Conv2d-13        [128, 256, 56, 56]         590,080
             ReLU-14        [128, 256, 56, 56]               0
           Conv2d-15        [128, 256, 56, 56]         590,080
             ReLU-16        [128, 256, 56, 56]               0
        MaxPool2d-17        [128, 256, 28, 28]               0
           Conv2d-18        [128, 512, 28, 28]       1,180,160
             ReLU-19        [128, 512, 28, 28]               0
           Conv2d-20        [128, 512, 28, 28]       2,359,808
             ReLU-21        [128, 512, 28, 28]               0
           Conv2d-22        [128, 512, 28, 28]       2,359,808
             ReLU-23        [128, 512, 28, 28]               0
        MaxPool2d-24        [128, 512, 14, 14]               0
           Conv2d-25        [128, 512, 14, 14]       2,359,808
             ReLU-26        [128, 512, 14, 14]               0
           Conv2d-27        [128, 512, 14, 14]       2,359,808
             ReLU-28        [128, 512, 14, 14]               0
           Conv2d-29        [128, 512, 14, 14]       2,359,808
             ReLU-30        [128, 512, 14, 14]               0
        MaxPool2d-31          [128, 512, 7, 7]               0
AdaptiveAvgPool2d-32          [128, 512, 7, 7]               0
           Linear-33               [128, 4096]     102,764,544
             ReLU-34               [128, 4096]               0
          Dropout-35               [128, 4096]               0
           Linear-36               [128, 4096]      16,781,312
             ReLU-37               [128, 4096]               0
          Dropout-38               [128, 4096]               0
           Linear-39                [128, 256]       1,048,832
             ReLU-40                [128, 256]               0
          Dropout-41                [128, 256]               0
           Linear-42                 [128, 14]           3,598
       LogSoftmax-43                 [128, 14]               0
================================================================
Total params: 135,312,974
Trainable params: 1,052,430
Non-trainable params: 134,260,544
----------------------------------------------------------------
Input size (MB): 73.50
Forward/backward pass size (MB): 28003.78
Params size (MB): 516.18
Estimated Total Size (MB): 28593.46
----------------------------------------------------------------
```

Observation:
Got an overall accuracy of 98% but it seems the model got overfitted as the validation accuracy is higher compared to train. This might also because of less data present in validation dataset.



<u>Scaling:</u>

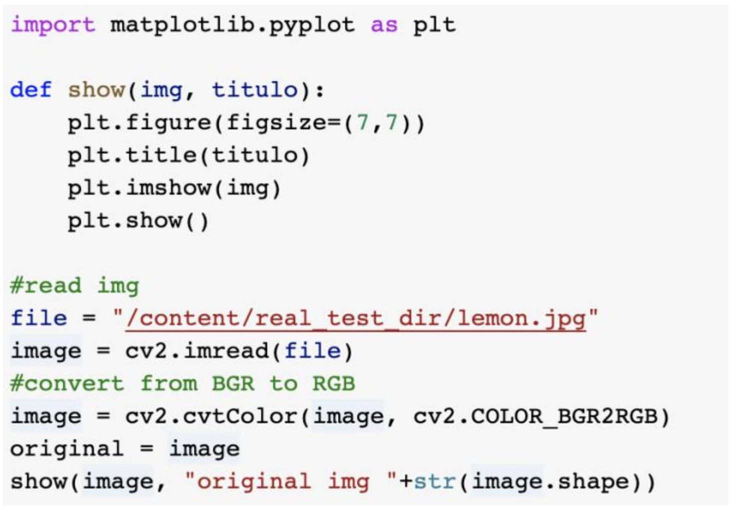
Method 1:

Tried extracting the defective image parts after categorizing the fruit amongst the two classes i.e., Good or Bad.

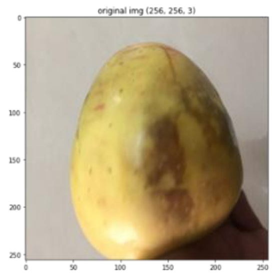
Referred few papers which are mentioned in References section for finding the defective areas on a particular fruit. Applied **OpenCV** pre-defined libraries in order to identify the fruit.

Below are the observations:

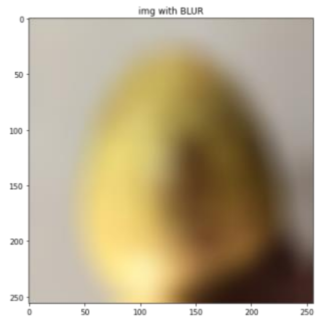
Step1: Picked an image



```
import matplotlib.pyplot as plt

def show(img, titulo):
    plt.figure(figsize=(7,7))
    plt.title(titulo)
    plt.imshow(img)
    plt.show()

#read img
file = "/content/real_test_dir/lemon.jpg"
image = cv2.imread(file)
#convert from BGR to RGB
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
original = image
show(image, "original img "+str(image.shape))
```

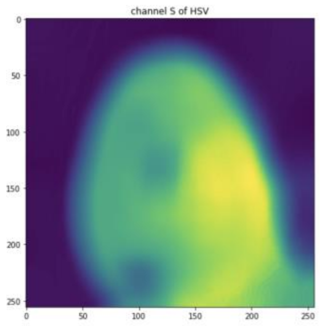


Step 2: Blurred the image



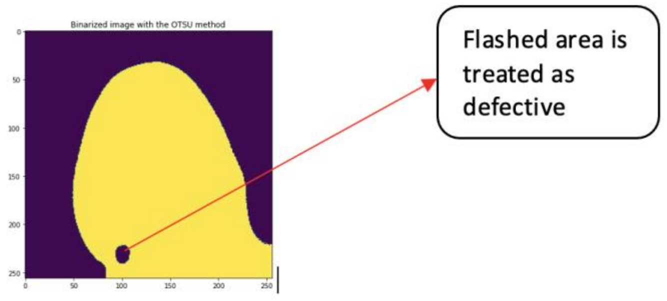
Step 3: Processed the image via an HSV channel

```
hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
#get hsv channels
h, s, v = cv2.split(hsv)
show(s, "channel S of HSV")
```



Step 4: Used OTSU method on the above image

```
#OTSU
_, thr = cv2.threshold(s, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)
show(thr, "Binarized image with the OTSU method")
```



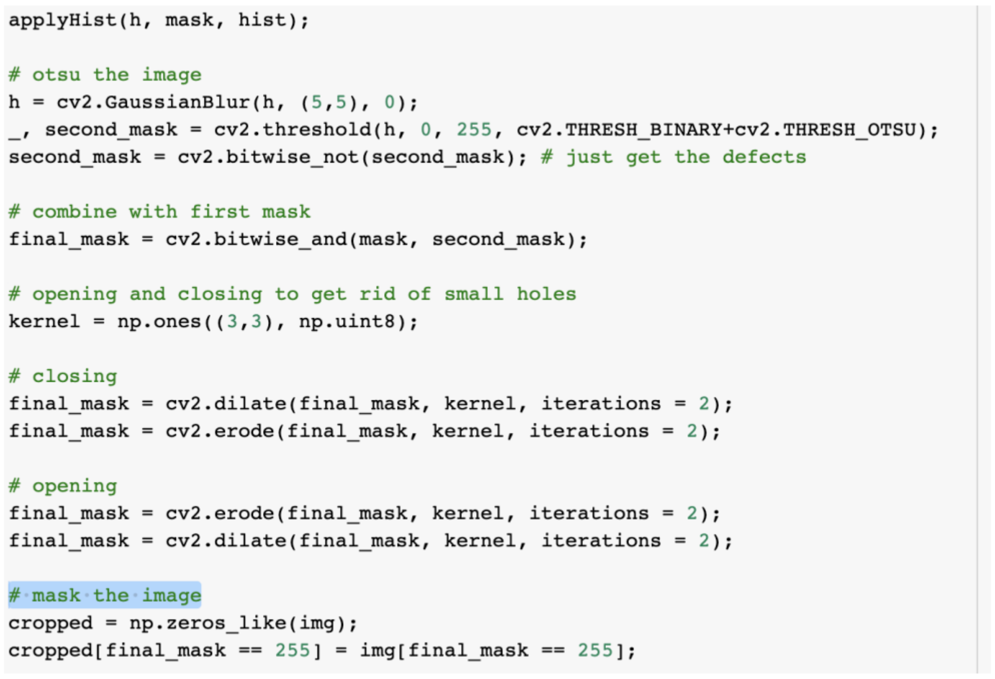
Observation:

Observed in the above two steps that even though the fruit is good the flashy areas are getting spotted as defective and other object areas in the image such as hand as we see in the input image are getting considered for further evaluations.

Method 2:

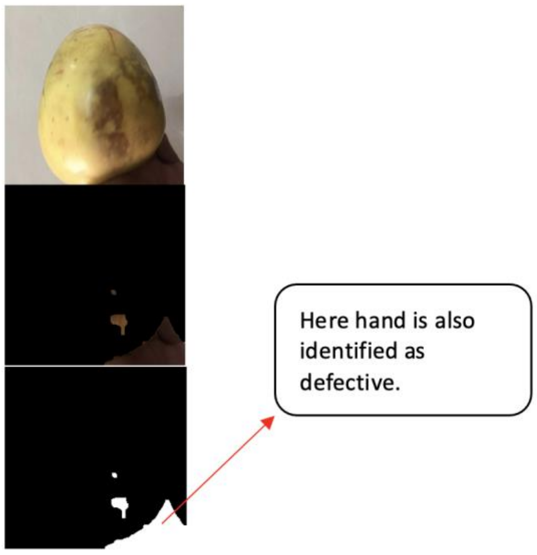
This is also similar to the above method, but few researchers stated that masking imaging and applying masks and dilating the image after applying OTSU method might exactly spot the defect area of an image.

This process is also done using OpenCV libraries but haven't considered for the proposed solution as there this identification is as simple as identifying the pixels closer to dark pixels and as we have hand holding fruits in the images the method had led to inappropriate conclusions.(Took reference code from a research discussion in stack overflow)



```
applyHist(h, mask, hist);

# otsu the image
h = cv2.GaussianBlur(h, (5,5), 0);
_, second_mask = cv2.threshold(h, 0, 255, cv2.THRESH_BINARY+cv2.THRESH_OTSU);
second_mask = cv2.bitwise_not(second_mask); # just get the defects

# combine with first mask
final_mask = cv2.bitwise_and(mask, second_mask);

# opening and closing to get rid of small holes
kernel = np.ones((3,3), np.uint8);

# closing
final_mask = cv2.dilate(final_mask, kernel, iterations = 2);
final_mask = cv2.erode(final_mask, kernel, iterations = 2);

# opening
final_mask = cv2.erode(final_mask, kernel, iterations = 2);
final_mask = cv2.dilate(final_mask, kernel, iterations = 2);

# mask the image
cropped = np.zeros_like(img);
cropped[final_mask == 255] = img[final_mask == 255];
```

Output:



Works involved in proposed model:

- Preparing custom dataset
- Development and fine-tuning the current model
- Regression testing the model with images outside the scope. (Random images from google)
- Adding Data Visualizations for the current dataset.

Team Member 2: (Bhama Krishna Pillutla)

PreProcessing:

For this image classification problem images are resized to (45 * 45 * 3). Later we normalized the color values to between 0 and 1, for this we divided the data by 255, as we know the maximum RGB value is 255. We also used flattened versions of train and test datasets for some of our models. Y- array conversion i.e., 1D array with 12 classes are converted to 12 arrays with each one class using one hot encoding as part of preprocessing.
Here is an example of train image.

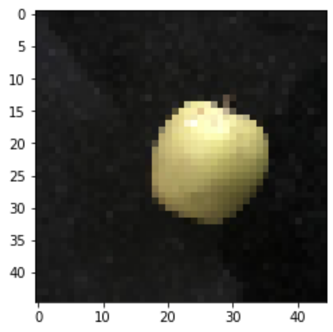


Figure 1: Apple Image from Trainset

Figure 2: Random images of train set are displayed below along with labels.

Model 1: with dense layers

This is a two layer network with dense layers.

```
model_dense_nodp = Sequential()
model_dense_nodp.add(Dense(512, activation='relu', input_shape=(X_flat_train.shape[1],)))
model_dense_nodp.add(Dense(224, activation='relu'))
model_dense_nodp.add(Dense(12, activation='softmax'))
model_dense_nodp.summary()
model_dense_nodp.compile(loss='categorical_crossentropy',
            optimizer=SGD(),
            metrics=['accuracy',])

history_dense = model_dense_nodp.fit(X_flat_train, Y_train,
                      batch_size=128,
                      epochs=30,
                      verbose=1,
                      validation_data=(X_flat_test, Y_test))
score = model_dense_nodp.evaluate(X_flat_test, Y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

Observation:

Train and test accuracies are 90 and 93 precents.
Train and test loss is 0.31 and 0.266 respectively. Accuracy wise the model is pretty good even though it is a two layer network. Overall the filters played a crucial role to learn the patterns in the data.

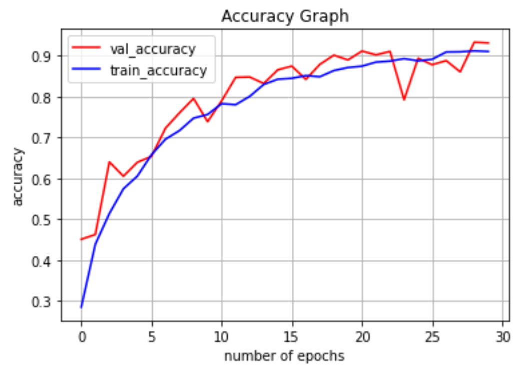


Figure 3: train and val accuracy graph for two layer dense network

Model 2: Dense layers with dropout

```
model_dense = Sequential()
model_dense.add(Dense(64, activation='relu', input_shape=(X_flat_train.shape[1],)))
model_dense.add(Dropout(0.05))
model_dense.add(Dense(32, activation='relu'))
model_dense.add(Dropout(0.05))
model_dense.add(Dense(12, activation='softmax'))
model_dense.summary()
model_dense.compile(loss='categorical_crossentropy',
            optimizer=SGD(),
            metrics=['accuracy',])
history_dense = model_dense.fit(X_flat_train, Y_train,
                    batch_size=64,
                    epochs=10,
                    verbose=1,
                    validation_data=(X_flat_test, Y_test))
score = model_dense.evaluate(X_flat_test, Y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

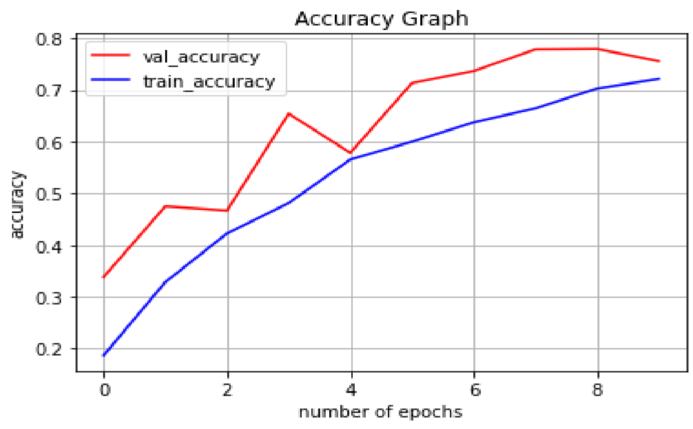


Figure 4: Train and Validation accuracy graph for dense layer network with dropout

```
model_dense.add(Dense(512, activation='relu', input_shape=(X_flat_train.shape[1],)))
model_dense.add(Dropout(0.05))
model_dense.add(Dense(128, activation='relu'))
model_dense.add(Dropout(0.05))
model_dense.add(Dense(64, activation='relu'))
model_dense.add(Dropout(0.05))
model_dense.add(Dense(12, activation='softmax'))
```

Observation :

though the final accuracy scores for train and test sets after 10 epochs are 72 and 75 there is large difference between the scores throughout 1 to 10 epochs. Here using dropout really dropped the performance of the model. Increasing the filters to 512 and 128 improved accuracy by 10 percent. Adding an extra dense layer and increasing the epochs to 20 has improved test accuracy(94%) but train accuracy is still low (89%).

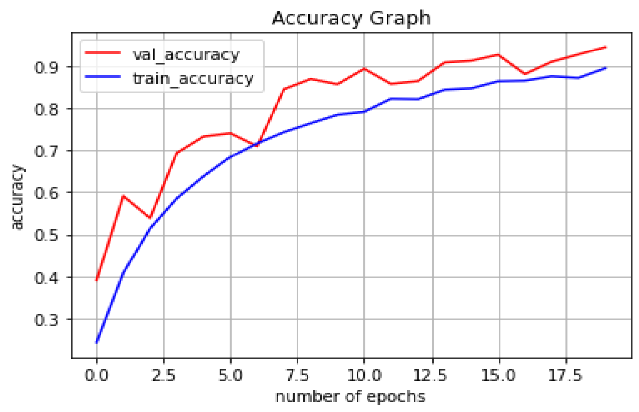


Figure 5: Train and Validation accuracy graph for increased filters

Model 3: Using CNN

```
model_cnn = Sequential()
model_cnn.add(Conv2D(32, kernel_size=(3, 3),
            activation='relu',
            input_shape=(45, 45, 3)))

model_cnn.add(Conv2D(32, (3, 3), activation='relu'))
model_cnn.add(MaxPooling2D(pool_size=(2, 2)))
model_cnn.add(Flatten())
model_cnn.add(Dense(512, activation='relu'))
model_cnn.add(Dense(12, activation='softmax'))
model_cnn.compile(loss=keras.losses.categorical_crossentropy,
          optimizer=tensorflow.keras.optimizers.RMSprop(),
          metrics=['accuracy'])
history_cnn = model_cnn.fit(X_train, Y_train,
        batch_size=64,
        epochs=20,
        verbose=1,
        validation_data=(X_test, Y_test))
score = model_cnn.evaluate(X_test, Y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

Observation:

Training accuracy is noted always 99 percent after 10 epochs until 20 epochs and test accuracy is around 97-98 percent. Here drop out would help to overcome overfit.

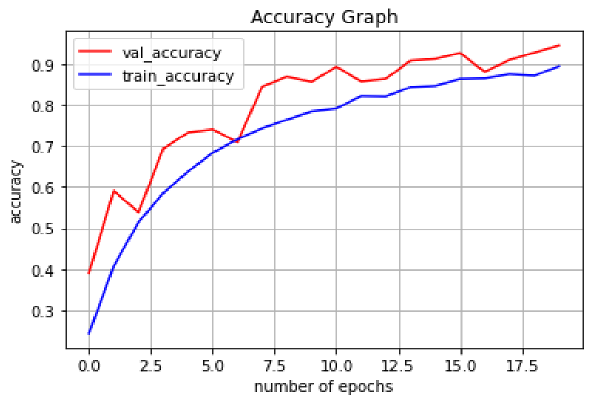


Figure 6: Train and Validation Accuracy graph for CNN

Model 4: Using CNN- dropout

```
model_cnn.add(Conv2D(32, kernel_size=(3, 3),
             activation='relu',
             input_shape=(45, 45, 3)))
model_cnn.add(Conv2D(32, (3, 3), activation='relu'))
model_cnn.add(MaxPooling2D(pool_size=(2, 2)))
model_cnn.add(Dropout(0.2))
model_cnn.add(Flatten())
model_cnn.add(Dense(512, activation='relu'))
model_cnn.add(Dropout(0.5))
model_cnn.add(Dense(12, activation='softmax'))
model_cnn.compile(loss=keras.losses.categorical_crossentropy,
           optimizer=tensorflow.keras.optimizers.RMSprop(),
           metrics=['accuracy'])
model_cnn.fit(X_train, Y_train,
        batch_size=64,
        epochs=20,
        verbose=1,
        validation_data=(X_test, Y_test))
```

Observation:
Both train and test accuracy improved slowly and showed consistency, train accuracy and test accuracy are 99 percent and 98 percent after 20 epochs.

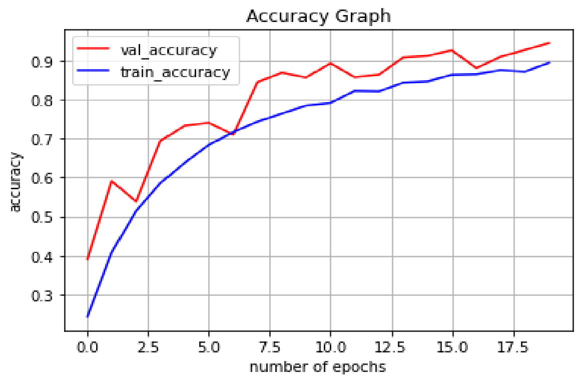
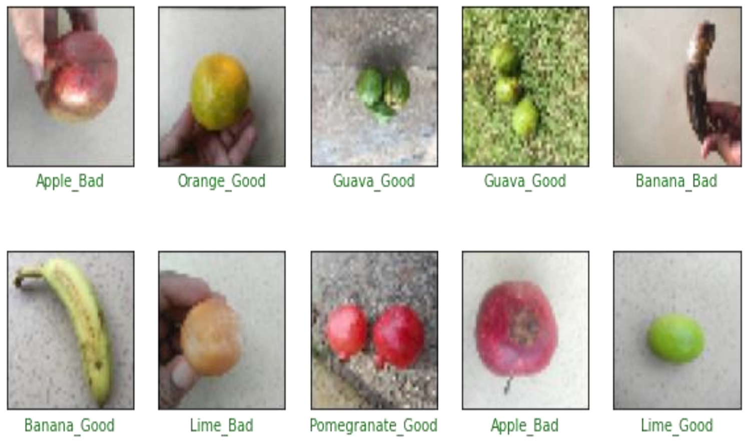


Figure 7: Train and Validation Accuracy graph for CNN with dropout

CNN with dropout gave us the best accuracy of 98 percent on test dataset. Predictions are accurate for the randomly picked images.
Later increasing the filter size to 512, 324 and 128 but accuracy reduced to 97% for both train and test sets.

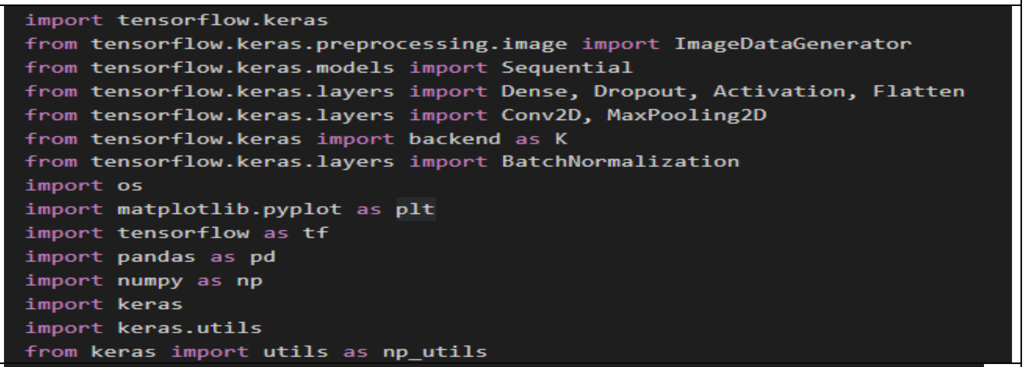


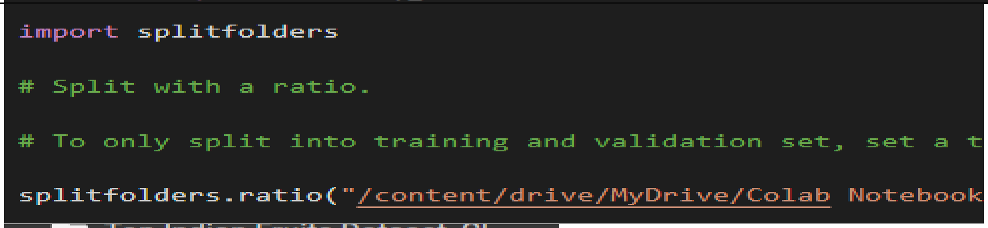
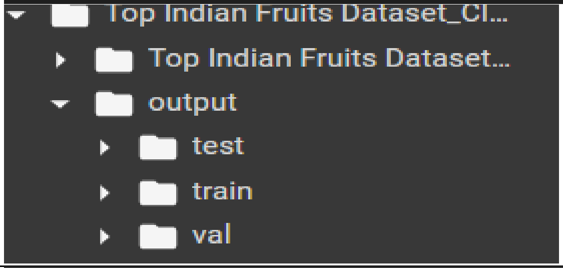
Works involved in proposed model:

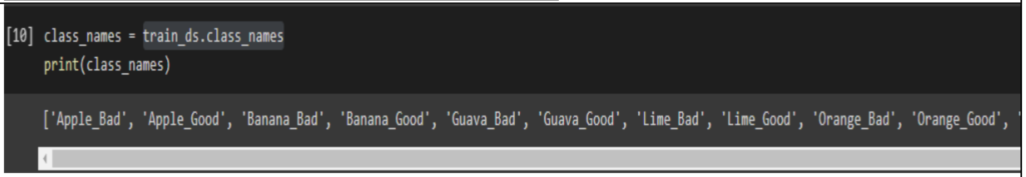
- Used above model as base model.
- Development and fine-tuning the model
- Regression testing the model with images outside the scope. (Random images from google)
- Adding Data Visualizations for the current dataset.

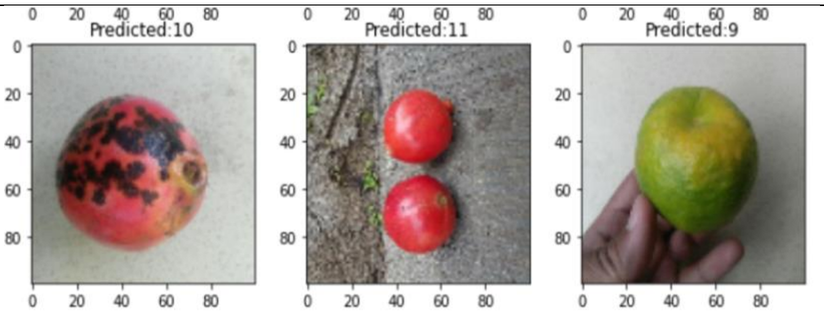
Team Member 3: (Ooha Parimi)

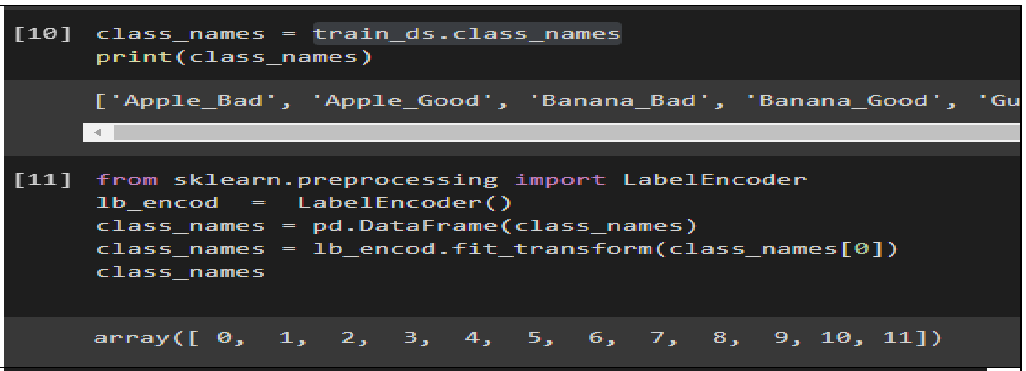
	METHODS	OUTPUT

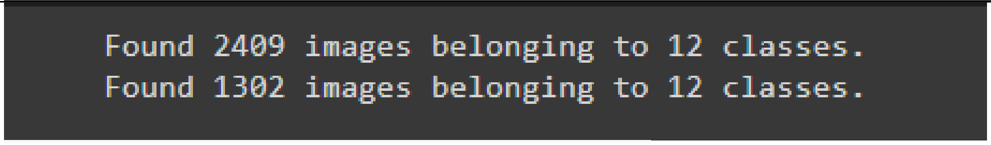
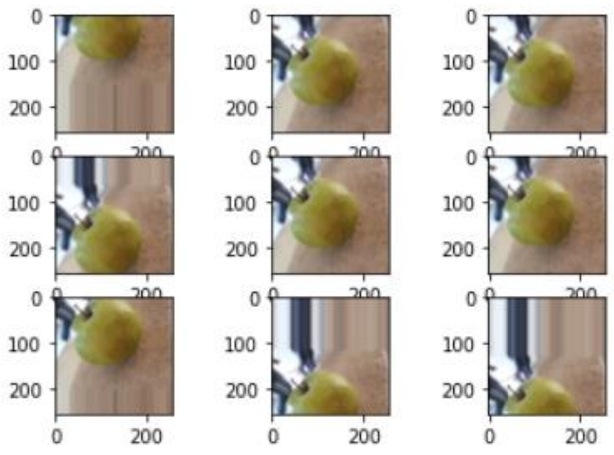
1	Importing Libraries	 `import tensorflow.keras`
		`from tensorflow.keras.preprocessing.image import ImageDataGenerator`
		`from tensorflow.keras.models import Sequential`
		`from tensorflow.keras.layers import Dense, Dropout, Activation, Flatten`
		`from tensorflow.keras.layers import Conv2D, MaxPooling2D`
		`from tensorflow.keras import backend as K`
		`from tensorflow.keras.layers import BatchNormalization`
		`import os`
		`import matplotlib.pyplot as plt`
		`import tensorflow as tf`
		`import pandas as pd`
		`import numpy as np`
		`import keras`
		`import keras.utils`
		`from keras import utils as np_utils`

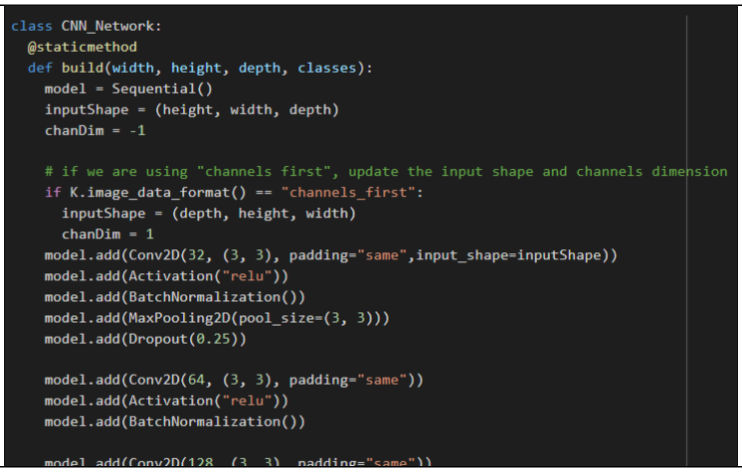
2	Loading the Dataset: Splitting the dataset into test, train, and Val	 `import splitfolders`
		`# Split with a ratio.`
		`# To only split into training and validation set, set a t`
		`splitfolders.ratio("/content/drive/MyDrive/Colab Notebook`
		 Top Indian Fruits Dataset_Cl...
		Top Indian Fruits Dataset...
		output
		test
		train
		val

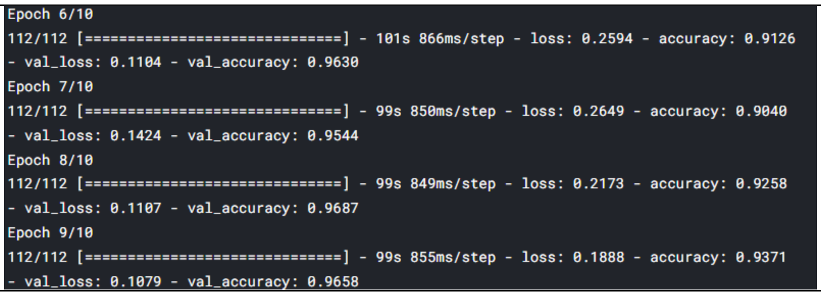
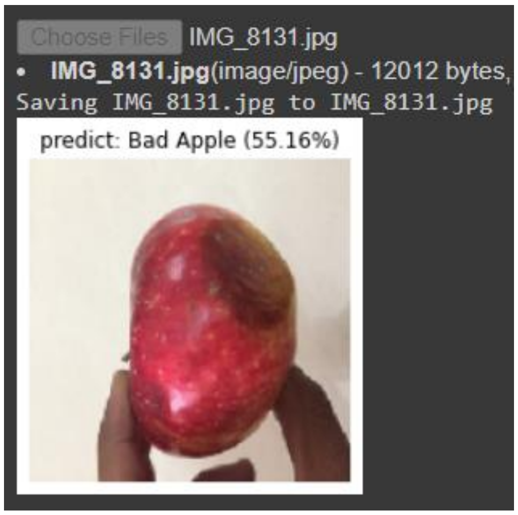
3	Pre-processing the data	 `[10] class_names = train_ds.class_names`
		`print(class_names)`
		`['Apple_Bad', 'Apple_Good', 'Banana_Bad', 'Banana_Good', 'Guava_Bad', 'Guava_Good', 'Lime_Bad', 'Lime_Good', 'Orange_Bad', 'Orange_Good', '`

4	Data Visualization	

		
		`[10] class_names = train_ds.class_names`
		`     print(class_names)`
		`     ['Apple_Bad', 'Apple_Good', 'Banana_Bad', 'Banana_Good', 'Gu`
		`[11] from sklearn.preprocessing import LabelEncoder`
		`     lb_encod  =  LabelEncoder()`
		`     class_names = pd.DataFrame(class_names)`
		`     class_names = lb_encod.fit_transform(class_names[0])`
		`     class_names`
		`     array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])`

5	Loading the images with a generator, Data Augmentation and Normalization	
		`Found 2409 images belonging to 12 classes.`
		`Found 1302 images belonging to 12 classes.`
		

6	Creating CNN to classify fruits	
		`class CNN_Network:`
		`  @staticmethod`
		`  def build(width, height, depth, classes):`
		`    model = Sequential()`
		`    inputShape = (height, width, depth)`
		`    chanDim = -1`
		
		`    # if we are using "channels first", update the input shape and channels dimension`
		`    if K.image_data_format() == "channels_first":`
		`      inputShape = (depth, height, width)`
		`      chanDim = 1`
		`    model.add(Conv2D(32, (3, 3), padding="same",input_shape=inputShape))`
		`    model.add(Activation("relu"))`
		`    model.add(BatchNormalization())`
		`    model.add(MaxPooling2D(pool_size=(3, 3)))`
		`    model.add(Dropout(0.25))`
		
		`    model.add(Conv2D(64, (3, 3), padding="same"))`
		`    model.add(Activation("relu"))`
		`    model.add(BatchNormalization())`
		
		`    model.add(Conv2D(128, (3, 3), padding="same"))`

7	Compilation Results	
8	Prediction Result	

```
Epoch 6/10
112/112 [==============================] - 101s 866ms/step - loss: 0.2594 - accuracy: 0.9126
- val_loss: 0.1104 - val_accuracy: 0.9630
Epoch 7/10
112/112 [==============================] - 99s 850ms/step - loss: 0.2649 - accuracy: 0.9040
- val_loss: 0.1424 - val_accuracy: 0.9544
Epoch 8/10
112/112 [==============================] - 99s 849ms/step - loss: 0.2173 - accuracy: 0.9258
- val_loss: 0.1107 - val_accuracy: 0.9687
Epoch 9/10
112/112 [==============================] - 99s 855ms/step - loss: 0.1888 - accuracy: 0.9371
- val_loss: 0.1079 - val_accuracy: 0.9658
```

```
IMG_8131.jpg
• IMG_8131.jpg(image/jpeg) - 12012 bytes,
Saving IMG_8131.jpg to IMG_8131.jpg
predict: Bad Apple (55.16%)
```

References:

https://ieee-dataport.org/open-access/fruitsgb-top-indian-fruits-quality
https://www.irjet.net/archives/V7/i5/IRJET-V7I51254.pdf
https://www.sciencedirect.com/science/article/pii/S131915781830209X
https://stackoverflow.com/questions/66234503/i-cant-get-the-patched-regions-of-a-citrus-fruit-using-otsu-method-with-the-gre
https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7297581/
https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html#further-learning
https://pytorch.org/tutorials/intermediate/quantized_transfer_learning_tutorial.html
https://www.kaggle.com/sangarshanan/age-group-classification-with-cnn/notebook
https://www.kaggle.com/rafetcan/image-classification-using-cnn-fruits/notebook
https://www.kaggle.com/gulsahdemiryurek/image-classification-with-logistic-regression/notebook
https://www.kaggle.com/sshikamaru/fruit-recognition/code