Capstone 2: Milestone Report 2

From previous model, it can be seen that training takes a long time, especially due to lack of GPU. One epoch takes around 67 seconds and becomes severely limited on the specs of the underlying laptop. This means that there is a long wait to experiment on different parameters and also little scope of trying out different things. Here is where Google's COLAB offering comes to the rescue.

Google Colab democratizes Machine Learning for the masses. It provides a platform where people can freely use 350 GB of disk and 12 GB of GPU RAM per project to train/test their models. A connection lasts for 12 hours and resources get reset after that. That should give us enough room to take this model to Colab environment and try out different things. But before we do that, lets dive in to the different offering of Keras to see if we can overcome the overfitting problem we saw.

Step 5: Data Augmentation using ImageDataGenerator

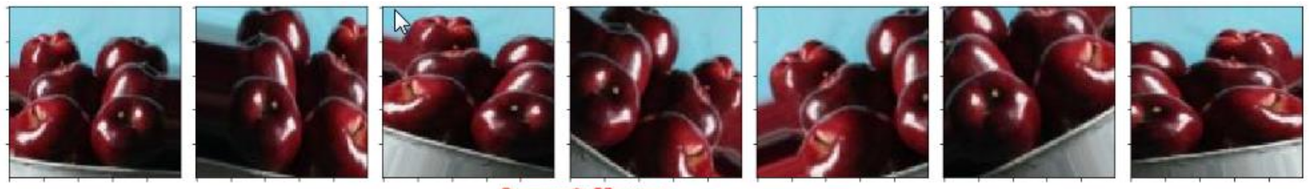
Keras library has a feature for generating variations of a base image by applying numerous modifications to it such as rotation, width shift, height shift, brightness range, shearing, zoom, channel shift etc. More information can be found at the official documentation page of the library at https://keras.io/preprocessing/image/. For the purposes of this project, we will be using these options to generate additional training images, and in effect 'Augment' the training dataset. These options have been chosen after testing a few times, and manually examining the output of the image generator. Options that produce reasonably recognizable images were retained to produce the training set. For example, it doesn't make sense to rotate images 180 degrees (or vertical flip) if there are very less changes of test images (unseen) to be that way. Also, we wouldn't want to shift images more than 20% if that meant partially visible fruits would be taken complete out of frame.

```
imgGen = ImageDataGenerator(
    rotation_range = 40,
    width_shift_range = 0.2,
    height_shift_range = 0.2,
    rescale = 1./255,
    shear_range = 0.2,
    zoom_range = 0.2,
    horizontal_flip = True)

i = 1
for batch in imgGen.flow(x, batch_size=1, save_to_dir=path_keras_output, save_format='jpg'):
    i += 1
    if i > 7:
        break
```

They have been normalized by 'rescale' parameter, which takes care of applying the normalization across all 3 channels. Also, zoom more than 20% doesn't make sense and also a horizontal flip would provide a very good duplication of the image. All parameters take effect in a combined way and the generator randomly chooses the ones to apply at a given point. This generator then needs to be called by flow or one of the options where Keras uses the output of the generator to train data. Let's have a look at what the generator produced from above code for the 7 augmented images.

Augmented Images



Flows -

There are 3 main type of flow methods that are documented in Keras online documentation. We will be exploring all 3 and choosing one of them for the project.

1. flow
2. flow_from_directory
3. flow_from_dataframe

1. **flow:** Takes data & label arrays, generates batches of augmented data. In this method, images need to be loaded, converted to arrays, reshaped and only then they are can be used by the ImageDataGenerator. Example of this is already covered in the code mentioned in data augmentation section. Drawback of this method is that additional processing is needed for the steps of loading the images and there is not any way to use it besides that. This will take up considerable resources, especially if you are loading 1000s of images. So, we won't be using this method for now.
2. **flow_from_directory:** This method is good in the sense that it can read the images directly from the directory and it takes labels as the directory names. So the images need to be organized in folders so that categorical variables are applies to classes automatically. There is also an option to mention the 'validation_split' parameter which splits the input set in to train and test set. Although this option worked very well on my laptop, this did not work well in Google's Colab environment. It was observed that the train-test split was very uneven and sometimes has produced very high class imbalance. That is one of reasons this method was not chosen for the project. Also, the train test split was done using matplotlib's preprocessing method.

3. **flow_from_dataframe:** This method takes in a pandas DataFrame containing the names of the files and classes as input and generates the data from there. We could even have absolute and relative paths in here, making this method one of the most flexible ones. Also, due to very less information and examples present on how to use it, it was thought to help other flow learners guide their way by using it for the project.

Step 6: Building Model, training and test findings

A basic CNN was chosen as the number of classes is not big (only five) and different parameters were tried on. As this needed experimentation, the local resources on my laptop were not sufficient and Google Colab was used (Base Model):

https://colab.research.google.com/drive/1wCAVxGcwGVJvJr3Kw8ADouNYDJx64jAd

The way Colab works is that it temporarily installs a python shell on Google Drive, where it can read files from your drive and use them in a temporary work space. In order to make use of this construct, the training and test data (folders with combined images) were zipped and uploaded to Google Drive as 2 separate files. They were then accessed and unzipped in the code for training and testing purposes.

Parameters that were experimented-on are listed below. Both training parameters (loss and accuracy) and validation parameters (val_loss and val_acc) were taken in to account during decision making.

a. ImageDataGenerator’s ‘vertical_flip’: It was observed that turning this parameter ‘on’ stalled the model’s accuracy rate at around 86% and did not lower the loss below 0.1.
b. Increasing filters: Current model has filters in layers as (32, 32, 64, 64). When more filters were added in layers beyond 64, for example 128, it increased the training time but no significant benefit was observed. These filters were changed at first 2 and last 2 levels to no improvement in val_loss parameter.
c. Increasing convolutional layers: More layers were added beyond the 4, but no evident benefit was observed. For example, 2 more convolutional layers were added of 64 each but the accuracy levels did not change. In fact, number of epochs to get the same result slightly increased. This was due to additional layers capturing more info and contributing towards the final classification parameter. Scaling down and keeping it 4 convolutional layers seems to be optimum.
d. Changing Dropouts: The current dropouts are set to 0.25 for the convolutional layers and 0.5 for the dense layer. Increasing the dropout seems to increase the training time and reduce the overall accuracy, whereas decreasing them seemed to go towards overfitting problem.
e. Changing kernel_size: The current kernel_size is set to 3x3 in the convolutional layers and this seems to be optimum for 2 reasons: Increasing it caused increase in the training accuracy and loss, there by increasing the validation accuracy and loss. Whereas decreasing it to 2x2 probably

caused it to be more sensitive to small variations in the image edges. Not all images are taken from similar distances to object and this seemed to play a part in reducing the kernel size. In the end, 3x3 seems to give the best results.

f. Increasing dense layer size: The dense layer is set to 128. Increasing that number to 256 or 512 seems to increase the parameters exponentially (in model summary) and cause model to give sub optimum results. My guess is that it would take more parameters to feed to the next dense layers and weights adjustment for back propagation would also take more time.

It can be concluded that perhaps more sophistication or complexity of a model comes at a computational and timing cost and may act in detriment of a simple classification neural network. If images were larger or complex, a more deeper and denser CNN would have been required.

Step 7: Advanced Model tuning and testing

Checkpoint callback:

In the base model, it was observed during the evaluation of the model, the parameters from the last epoch played significant role. In the overall model training exercise, the training accuracy and loss did not linearly improve. They eventually improved, but in some epochs they become worse temporarily, may be due to some nonstandard images from the training set or ones generated by the ImageDataGenerator. For example, the accuracy from 84th epoch came down from 0.9307 to a low of 0.9113 and went back up in 91th epoch. This nonlinearity can result in the model not being the best state at the end of training epochs. For example, the below values on evaluation of validation dataset and test dataset (respectively) at the end of 250 epochs of model training may not be the best:

```
Validation set evaluation = [0.8309193852904718, 0.9]
Test set evaluation       = [0.7641155040264129, 0.8599999904632568]
```

It would be prudent to save a copy of model that performed the best in the intermediate states of training. That is where the 'checkpoint' functionality of Keras is used to save the best performing model weights on the basis of least amount of 'val_loss' parameter. This checkpoint is saved by 'callback' option given during training and later loaded to check the validation. On loading the intermediate checkpoint file ('weights_cap2_3.hdf5') of best weights, it can be seen that it performs better on the test (unseen) data, with accuracy of 90% vs that of 86% above.

```
Validation set evaluation = [0.2686952355752389, 0.97]
Test set evaluation       = [0.2596906507015228, 0.9000000023841858]
```

Batch normalization

On further research online for advanced tuning and mentor feedback, it came up that 'BatchNomalization' on the model layers makes the model train faster. To try this and to be able to compare to the original trained model, a new Colab notebook was setup at the below location:

https://colab.research.google.com/drive/1q_Rt6op3rv6cmog8dH_0hPMLjdG2mwoS

It can be seen how the model is changed significantly to introduce ‘BatchNomalization’ on the four convolutional layers and also on the dense layers.

```
from keras import layers
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),input_shape=(100,100,3), padding='same', use_bias
= 'False'))
model.add(layers.BatchNormalization(axis=3))
model.add(layers.Activation('relu'))
model.add(Conv2D(32, (3, 3), use_bias = 'False'))
model.add(layers.BatchNormalization(axis=3))
model.add(layers.Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Conv2D(64, (3, 3), use_bias = 'False', padding='same'))
model.add(layers.BatchNormalization(axis=3))
model.add(layers.Activation('relu'))
model.add(Conv2D(64, (3, 3), use_bias = 'False'))
model.add(layers.BatchNormalization(axis=3))
model.add(layers.Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, use_bias = 'False'))
model.add(layers.BatchNormalization())
model.add(layers.Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(5, use_bias = 'False'))
model.add(layers.BatchNormalization())
model.add(layers.Activation('softmax'))
```

An additional argument of ‘use_bias=False’ needs to be used in the layers preceding BatchNomalization. Also axis=3 is chosen due to apply it on the channels axis as Keras is running on TensorFlow with input tensor of [b, h, w, c] meaning batch_size, height, width and channels. It was noticed that when this model is trained for the same number of epochs, it gave better accuracy on validation and test data. The actual time it took for each epoch slightly increased by 1s/epoch but final accuracy on unseen data (test set) increased by 4%.

```
Validation set evaluation = [0.14419285039727886, 0.9533333333333334]
Test set evaluation       = [0.15934765964746475, 0.94]
```

It can be clearly seen how BatchNomalization proved to be effective in increasing not only the accuracy, but also bringing down the val_loss to 0.1593.

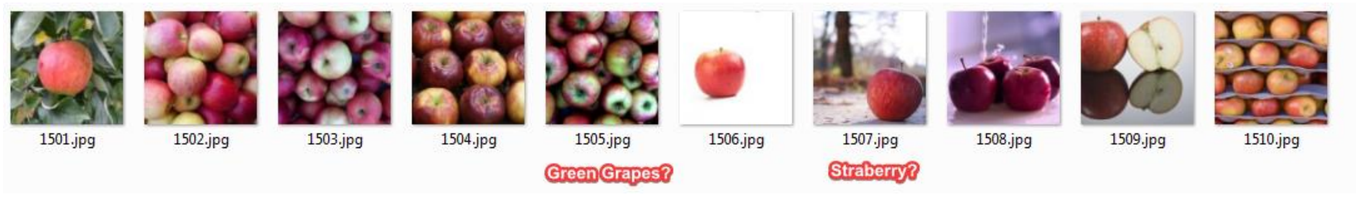
It can also be inferred from the model summary that the most number of parameters were introduced by the Dense layer that takes the convolutional input from the last one of 64 layers. Almost 4.3 million parameters are introduced here, which are then pass to the next Dense layer with the last Batch Normalization introducing only 20. Overall, BatchNomalization introduced 1300 new parameters, but 650 non-trainable but gave a better performance than the one without it.

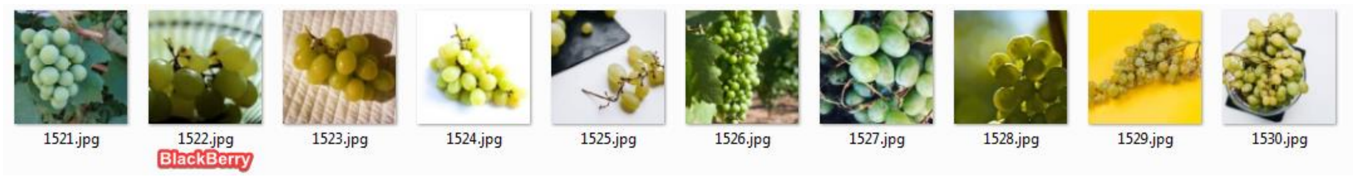
Final evaluation

It can be inferred from the confusion matrix that model was very good at detecting 'Blackberries', 'Kiwi' and 'Strawberry' but got confused with Apples as Green Grapes and Strawberry. Getting Apples mixed with Strawberries was in line of expectation as some images in the set resemble closely with each other and have the same color. But getting Apples confused with Green Grapes was a surprise. This could have been due to presence of some green apples in the training set and color and edge confusion with strawberries. There was also expectation of model getting confused between Green Grapes and Kiwi as they also have same color, but it seems like the model has become very good at distinguishing between the two. Also surprisingly, one Green Grapes was misclassified as Blackberry, perhaps of the shadow and low light conditions of the image.

```
Confusion Matrix
[[ 8  0  1  0  1]
 [ 0 10  0  0  0]
 [ 0  1  9  0  0]
 [ 0  0  0 10  0]
 [ 0  0  0  0 10]]
```

Lets have a look at which images could have been the misclassified ones.





The below report gives details on the recall, precision and f1 score of the model.

```
Classification Report
              precision    recall  f1-score   support

       Apple       1.00      0.80      0.89        10
  Blackberry       0.91      1.00      0.95        10
Green Grapes       0.90      0.90      0.90        10
        Kiwi       1.00      1.00      1.00        10
  Strawberry       0.91      1.00      0.95        10

    accuracy                           0.94        50
   macro avg       0.94      0.94      0.94        50
weighted avg       0.94      0.94      0.94        50
```

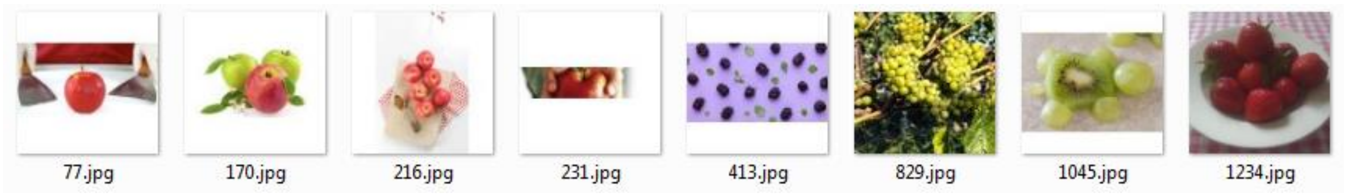
Conclusion

Challenges of accuracy and detection complete.

It can be concluded that the model is good at detecting edges of similar colored fruits and also adept at recognizing the various shapes that a fruit may have (test set of 50 images). For example, variations in shape of apples, grapes and strawberries. The last challenge of recognizing partial fruits also has been completed as partial images were fed to the model, both during training and testing steps with accuracy of 95.3% and 94% respectively.

Further investigations

Some more investigation needs to done to find out what images compromise the accuracy of the model by exposing the model to more test data. This would solidify the rationale behind misdetection due to lighting conditions, proximity of the object or model limitations. It was also observed that the training set has some images that could have caused some confusion to the model whilst training. The below images are either with another fruits or too distant and could cause classification confusion



Next steps

1. Other ImageDataGenerator parameters can be used to improve the brightness adaptability of the model. Some dark images could be made lighter and some bright ones darker for training. This will make the model more robust and tolerant of different light conditions.
2. Keras also makes image resizing easier as it is nothing but a numpy array, to which resizing can be on the fly without any preprocessing needed. This option can be tried out to see if it is any better or worse than an independent preprocessing step. Other libraries can also be tried like ‘imgaug’ so compare their performance.
3. The scope of this classifier can be expanded to 10 or 15 classes to see if the model needs addition of more layers.