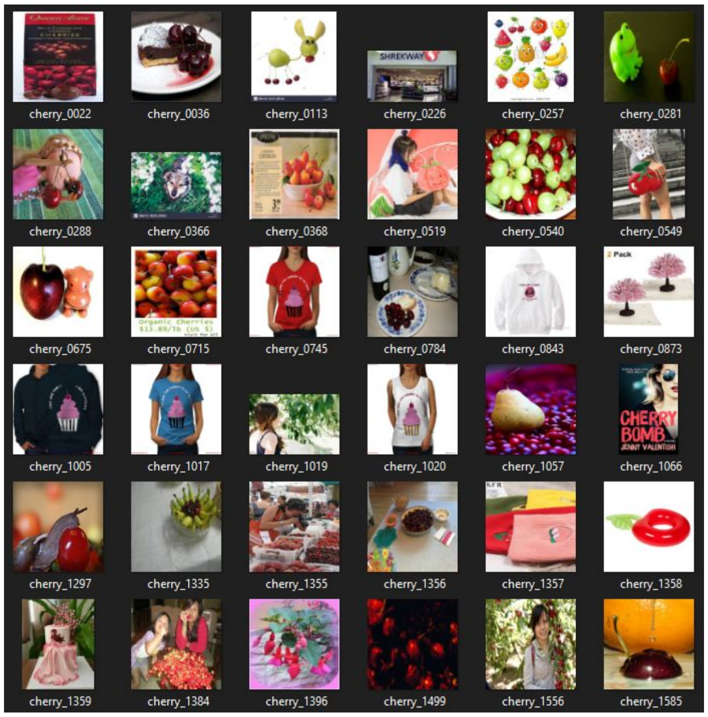
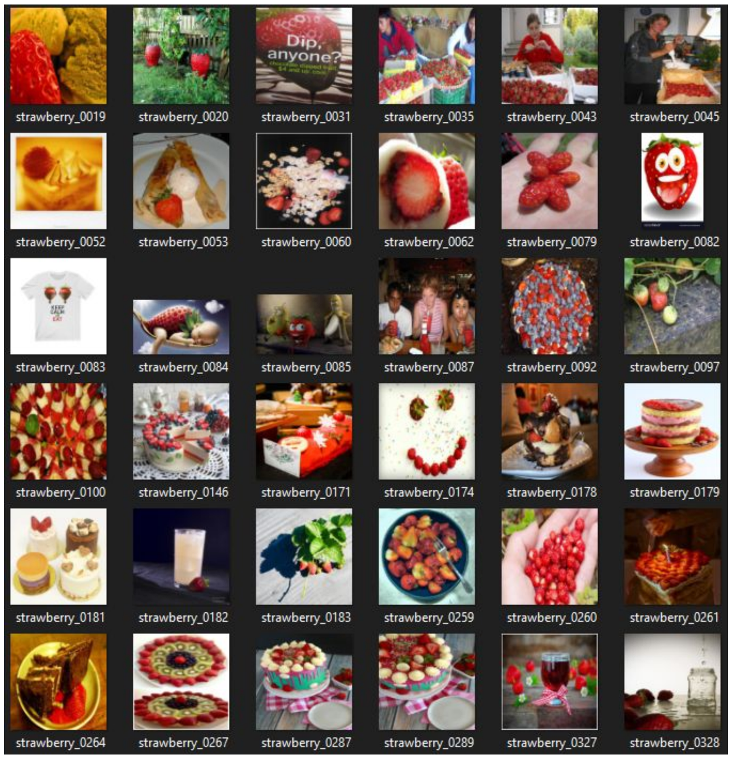
COMP309 Project - David Burrell : 300209541

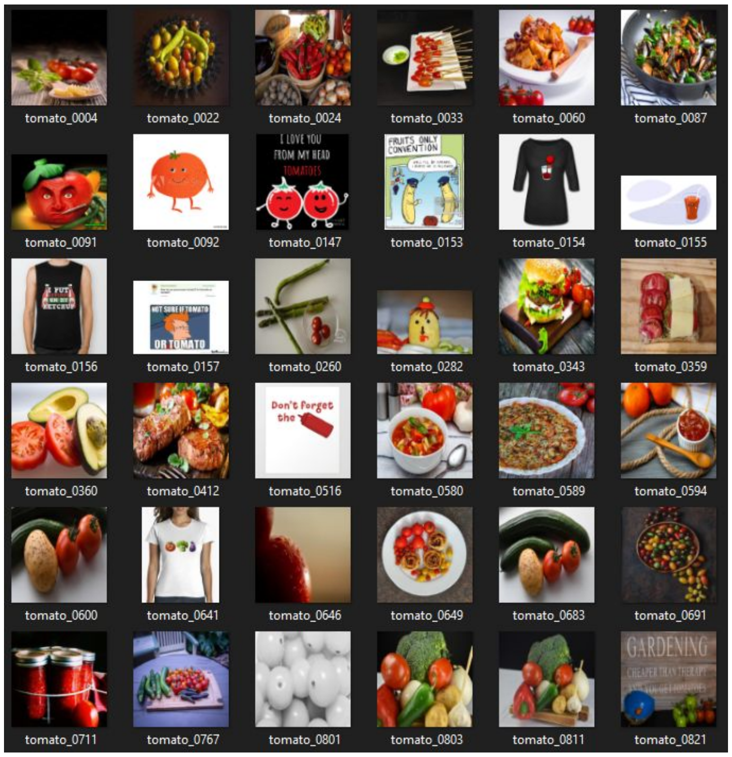
Step 1: Conduct exploratory data analysis (EDA).
Looking through the datasets, cherry, strawberry, and tomato, during exploratory data analysis made it clear that the training data was not perfect. Many of the images provided in all three sets were not appropriate to train from, most of the problematic images did not have the given fruit as the dominant subject of the image, if even having the fruit in it at all.
Three main factors were looked at when examining the datasets.
Firstly the Illumination conditions of the given image, as this affects the pixel level of images.
Second is the environmental clutter in the images, causing the subject to either blend into the background in ways that it is no longer the dominant subject.
Lastly occlusion of the subject, either caused by other contents of the image, or the framing (such as for artistic purposes)
These images would provide bad training information, as they do not have a clear size and shape of the fruit to learn from, a subset of each of the bad images are shown below.



Cherry



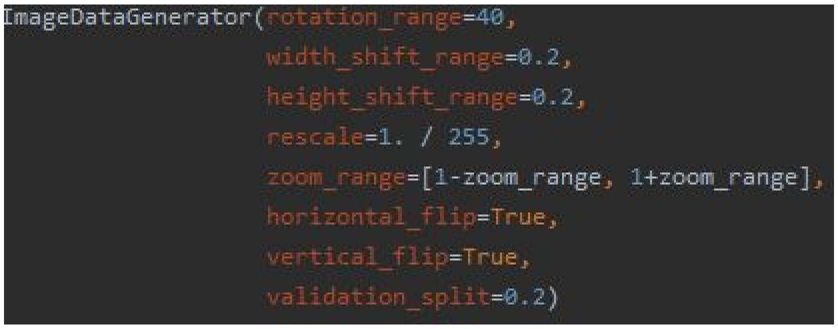
Strawberry



Tomato

Data Pre-Processing:
Utilising the ImageDataGenerator class from keras which allows generation of batches of tensor image data with real-time data augmentation, giving the ability to do the data preprocessing on the images as they are loaded into tensor data.
Augmentation of data during preprocessing can resolve common issues that arise with image classification, namely image inconsistency and overfitting, by allowing the system to deal with consistent new information supplied by modified versions of already assessed images.

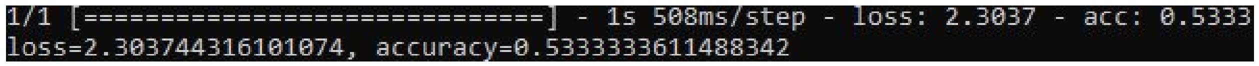
Re-sizing:
Resizing of the images is done when loading the data with the ImageDataGenerator, by setting this value to 128 this resizes the supplied images from 300x300 to 128x128, significantly reducing computation time over the default image size, at the cost of accuracy due to having more information available.
The arguments passed to the constructor of the ImageDataGenerator allow the augmentation of the variables I set are:



```
ImageDataGenerator(rotation_range=40,
                   width_shift_range=0.2,
                   height_shift_range=0.2,
                   rescale=1. / 255,
                   zoom_range=[1-zoom_range, 1+zoom_range],
                   horizontal_flip=True,
                   vertical_flip=True,
                   validation_split=0.2)
```

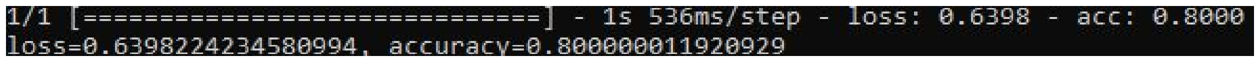
rotation_range: the image is randomly rotated clockwise or anti-clockwise by and amount in the range of 0 to 40
width_shift_range and *height_shift_range*: shifts the image vertically or horizontally by a fraction given, in this case 0.2
rescale: this is the value that applies the normalization to the pixel values, reducing them from a 0-255 scale, to a simple floating point value between 0 and 1. Leaving values as they were would cause the model processing to take far too long.
zoom_range: randomly applies a zoom between the range given (0.8-1.2) to images
horizontal_flip and *vertical_flip*: do as they say, randomly flip images both horizontally and vertically

Improvements from preprocessing
The preprocessing methods applied increase the testing accuracy from 53%



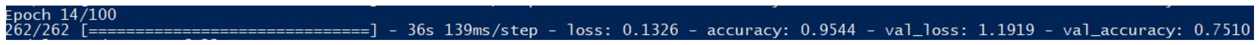
```
1/1 [==============================] - 1s 508ms/step - loss: 2.3037 - acc: 0.5333
loss=2.303744316101074, accuracy=0.5333333611488342
```

To 80% with the methods applied



```
1/1 [==============================] - 1s 536ms/step - loss: 0.6398 - acc: 0.8000
loss=0.6398224234580994, accuracy=0.800000011920929
```

Also noteworthy is the increase in training accuracy all the way up to 95% without preprocessing



```
Epoch 14/100
262/262 [==============================] - 36s 139ms/step - loss: 0.1326 - accuracy: 0.9544 - val_loss: 1.1919 - val_accuracy: 0.7510
```

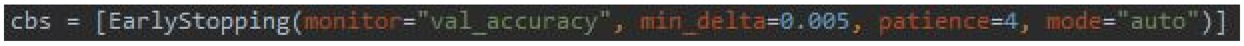
Methodology:
How you used the given images
The supplied images were used as a training set and a validation set, at a split of 80/20. This was done during the preprocessing step by specifying the *validation_split* variable of the ImageDataGenerator. The validation set supplies an estimation of model properties as it is learning, and changing the classifier accordingly to provide better results, this is a form of cross-validation.

Loss Functions
Loss functions act to represent some cost associated with an event, and is used to help determine how well an algorithm is performing when modelling a dataset, minimizing these loss functions is the goal of optimization problems.
Examples of loss functions were used in previous assignments, namely MAE and MSE. For this assignment *categorical_crossentropy* was the most appropriate given the setting used for the *flow_from_directory()* function for importing class labels is *categorical* meaning the labels are encoded using 2D one-hot encoding, which *categorical_crossentropy* performs well on.

Optimization method
The job of the optimizer is to minimize an error function (loss), using the internal parameters of the model, in the case of this neural network, these are the weights and bias used during computation of output values. These values are learned and updated by the optimizer to minimise the loss during the training process.
The first used optimizer is *SGD* which was the default supplied in the template code, this optimizer performed poorly when compared to others.
RMSprop aims to optimise by dividing the gradient by a running average of its recent magnitude in an effort to avoid large oscillations in learning weights. This optimizer was a large improvement over SGD in which during the running of test.py improved my result from 60% to 73%
I also ran the training program using the Adam optimizer, which worked as well in testing as RMSprop, but built the model in less epochs. This is due to Adam essentially working as RMSprop with momentum, keeping exponentially decaying average of past gradients and squared gradients, in which it uses to compute the adaptive learning rates for internal parameters.
I chose to stick with the adam optimizer for the assignment.

Regularization strategy
Regularization is the methods used to avoid overfitting of the model, as we know that having high accuracy on the training model does not translate to having high accuracy on the testing data. To avoid overfitting in this neural network model, the implementations of Dropout and Early Stopping were used.

Dropout works by randomly setting a fraction of input units to 0 while updating during training time. In this project Dropout was used after the hidden layer with a setting of 0.2 and again after the output layer with a value of 0.3
The other method is Early Stopping, which simply monitors a set quality of the training method and will stop the training when a threshold of improvement is no longer met, in the case of this assignment, validation accuracy is used as the monitored quality.



```
cbs = [EarlyStopping(monitor="val_accuracy", min_delta=0.005, patience=4, mode="auto")]
```

Activation Function
The activation functions used in this model are needed to add non-linearity between inputs and outputs of the layers in the system. In this assignment *relu* has been used in most layers, as it generally performs well in most situations and has a lower computational cost than other functions such as *sigmoid*
For the final layer of the CNN *softmax* is used to normalize the outputs of each unit to a value between 0 and 1, this has the added benefit of being an appropriation of the probability distribution, *softmax* is used over *sigmoid* as there is more than one category, in which *sigmoid* is effect on binary classification, *softmax* works well on multiple target labels.

Hyper parameter settings
These are the parameters that determine the structure of the network created to train the data.

For the *convolutional* layers the first one used needs to know the size of the images as input, in this case 64,64,3 with 3 being the channels for RGB.
The filters that define the output space dimensions are used are set to start at 32, and increase to 64 for later layers, this helps to represent the later more detailed features that occur as layers deepen. The kernel size of all convolution layers used is set to (3,3) to represent the convolution window matrix, larger sizes inferred accuracy reductions

For the *pooling* layers, the smallest value was used to make sure not too much information is lost when reducing the dimensionality of the feature maps.

learning rate for the optimizer, which helps control the rate of learning while maintaining convergence is set to a low value, 0.0005. A value this low slows the training rate while converging in a smooth manner.
A higher value should increase the training rate, but the model may never converge.

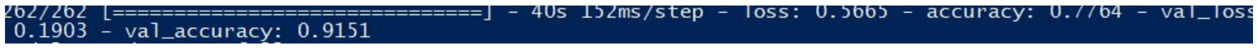
min_delta is used by Early Stopping as the value to check if insufficient improvement has been made, and the *patience* setting defines over how many epochs to check for this delta.

The *epoch* setting defines the amount of times the model will be fit with the whole training data, more epochs can be useful on certain datasets, but can easily lead to overfitting, this is where early stopping shines, to prevent the overfitting of data the training will be stopped before the *epoch* setting is reached. The setting used in this assignment is 100, early fitting usually stopped this process at around epoch 30 however.

How many more images obtained
Whilst I did look for more images, the hassle of downloading 200 or more was a time constraint, however I found this dataset of many different fruits, prepared for machine learning :

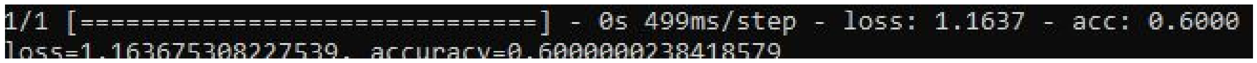
https://github.com/Horea94/Fruit-Images-Dataset

I felt this was disingenuous to the goal of the assignment by using images crafted specifically for machine learning algorithms to learn from, rather than the messy images supplied by the Template.
Just to see if they would help I added 738 cherry images, 492 strawberry images and 479 tomato images to the dataset, and whilst getting fantastic accuracy results on the validation sets:



```
262/262 [==============================] - 40s 152ms/step - loss: 0.5665 - accuracy: 0.7764 - val_loss
 0.1903 - val_accuracy: 0.9151
```

We see only a 60% accuracy of that model on the test set



```
1/1 [==============================] - 0s 499ms/step - loss: 1.1637 - acc: 0.6000
loss=1.163675308227539, accuracy=0.6000000238418579
```

Results
Comparing my results against the baseline MLP model shows an increase in accuracy of almost 50%, and the baseline model had a significantly quicker training time equivalent to one epoch of my model. This is because the MLP does not repeat weights over blocks of images in an image area, this is where the CNN uses this as a specialty function that uses much computational power.

Conclusion
The CNN model that I created has better accuracy on training, validation and testing than the baseline MLP classifier, at the cost of time, which in many situations is taken into account, but still can be detrimental. If an MLP could be written that had almost equivalent results at a fraction of the time then my CNN model would be invalidated.