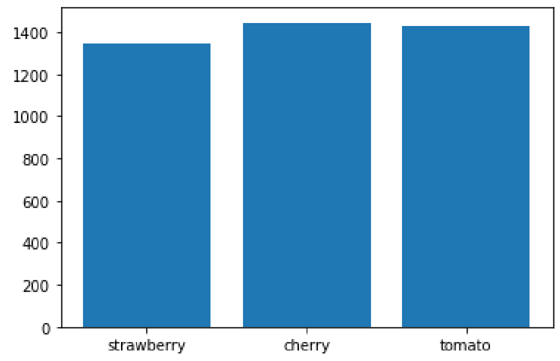
Comp 309 Project- Image Classification
Chloe Crowe - 300539785

The purpose of this report is to learn how to implement and interpret Convolutional Neural Networks. Throughout this report, I will be trying to correctly classify images. There are 6,000 images from three different classes, Strawberries, Cherries and Tomatoes 2,000 from each class. To complete the image classification, I followed a TensorFlow tutorial coding in Python. I found that following the tutorial material was a lot harder and more complicated than using TensorFlow.

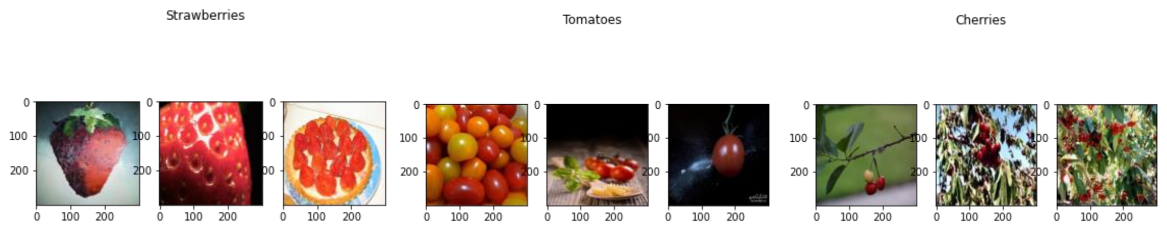
Before applying any models, first, it is vital to check the data. Because the data is images, they had to meet a set of requirements. Images must be clear / not blurry, the object must be close enough, it must be a real object, the object must be that of its type, and it must be the main object in the image. If an image does not meet these requirements, they have been removed from the data set.
From the cherries data set 57 images have been removed. From the strawberries data set 159 have been removed. 75 images have been removed from Tomatoes. These images have been removed because otherwise the model will get confused with images of t-shirts and cartoons.

This means that the data sets no longer have the same number of images. Cherries have 1,444 images, Strawberries have 1,341 images and Tomatoes have 1,425 images.
If I had not removed the images from the dataset then the classification may get confused but the occasional cartoon image.



The bar plots show that cherries and tomatoes have similar numbers of values, however, strawberries do not. This is due to the fact that strawberries had more images that did not meet my standards. Although there are only about 100 images different, so I am not too concerned about it having a negative effect on the classification.

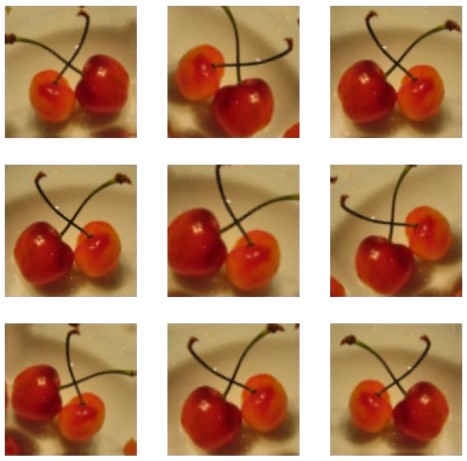
Pictured below are three example images from each of the classes. Strawberries, Tomatoes and Cherries. There are varying qualities of images. It was hard to narrow down all the images as I wanted a big enough sample to prevent overfitting. I manually removed the images from the data folder to improve the quality. This resulted in looking into thousands of images. I found that doing this myself was going to be quicker than attempting to create another program to clean the data.



To help with the model processing I have reduced the image sizes to 100 by 100. Currently, the size of the images will slow down the processing. I also did this so the neural network would not produce any errors. It is a lot easier to deal with a set of images that are the same size.

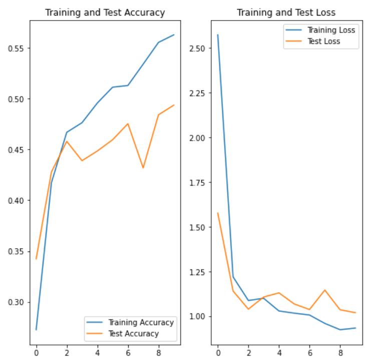
All the images within the dataset are in colour. This means that each pixel has a value from 0 to 255. To reduce the processing time, I have normalised the data. To complete this I have divided all the values by 255. Giving any given value a value between 0 and 1.

To create a model that does not over or underfits it is beneficial to have a big sample size. As the train data had issues, I needed to reduce it. So that my model can predict accurately I wanted to increase the sample size. I completed this by augmenting the data using a Keras model. The data augmentation model flips the image horizontally, rotates the image by a scale of 0.1 and zooms by a scale of 0.1. This model creates images in batches within the model in real-time. As the neural network finds patterns within the images it distinguishes the data as different images. Increasing the size of the training data. Below is an example of one image being transformed into "multiple" different images. I applied data augmentation within a Keras model, the model applies to images in batches. This means the batch size directly changes the number of extra images the model creates.



After completing an EDA there are 4,210 images across the three classes. For training, I spilt the images into a training and validation test set. Using TensorFlow Keras I applied a validation split of 0.3, this resulted in 70% of the data in the training set with the remaining 30% in a validation test set. Before using the data, I applied to autotune. I did this to help with processing throughout the application of the model. Autotune helps with processing by keeping a copy of the data within the computer's memory, it also overlaps processing to increase efficiency.

Before creating a convolutional neural network I created a basic neural network using Kera. The neural network included 4 layers. With one output later. The model uses the "Adam" optimiser (gradient descent) and a "SparseCategoricalCrossentropy" loss function.

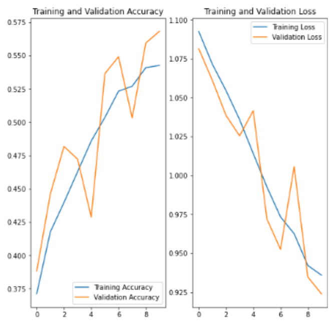


The model was not able to correctly classify a test image of a strawberry.
The model had a test accuracy of 49.37 % (2dp) this model is clearly not good enough for a final model. To increase the effectiveness of the model it will be beneficial to add more layers and create a convolutional neural network.

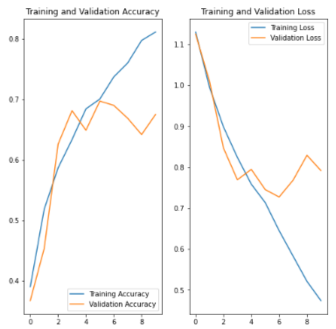
Throughout the creation of the convolutional neural network (CNN), I tested different loss functions. Keras has built-in loss functions, I trialled using two different loss functions “SparseCategoricalCrossentropy” and “MeanSquaredError”. A loss function measures how well the neural network models the training data. Throughout the training we want the loss function to minimise the loss between the prediction and the target output. A good loss function would reduce this loss.
I found that using a “SparseCategoricalCrossentropy” helped minimise this loss. This loss function is ideal for networks that have two or more classes that are represented as integers. In this case, we have three classes (Strawberry, Cherry and tomato) each class is not encoded, this means that strawberry is represented as an integer e.g cherry = 1.

An optimiser within a CNN helps reduce the overall loss and increases the model’s accuracy by modifying values through its weights.
As with the loss function, I trialled different optimisers this time 4 SGD, RMSprop, Adam and Adamax.

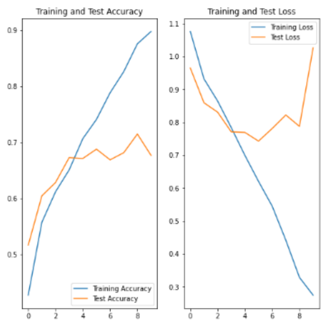
SGD is a gradient descent that updates weights throughout the training process. RMSprop is similar to SGD but it is adaptive, it changes its learning rate throughout training. Adam uses a gradient descent method to optimise updating the learning rate for each weight. Adamax is an extension of Adam but helps to stabilise the data.



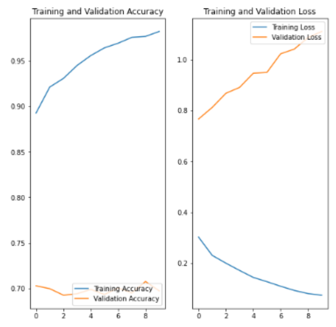
SGD



RMSprop



Adam

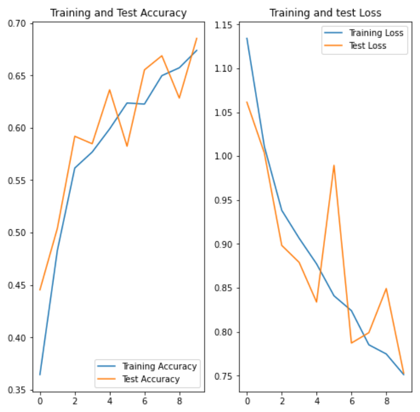


Adamax

Using an epoch of size 10, the Adam optimiser has worked best for this network. It gave a test accuracy of 0.68 which is the best it has been so far. It produced the best test accuracy score with some overfitting. Because of the overfitting, there are more values that I should investigate tuning.

This includes the activation function. Within the original model, I included a dropout of 0.2. Dropout() applies to the input of the network, it sets the value to 0. The dropout activation function helps reduce overfitting.

The first thing I investigated was removing this function completely. Obviously, this really did not work in my favour and made the model terrible.

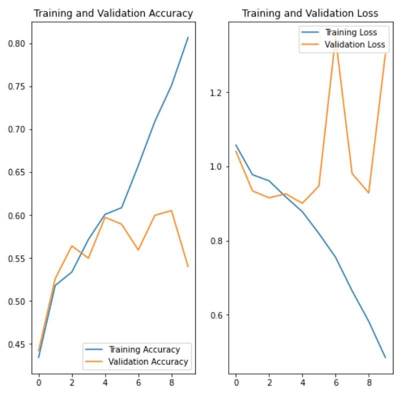


My next step was to increase the dropout function to 0.5. This gave me a model with a lot better fitting that performed a lot better. The graph on the left shows the test vs training accuracy and loss. With more epochs, we can see that the test and training rise at relatively the same rate, with the loss decreasing together as well.

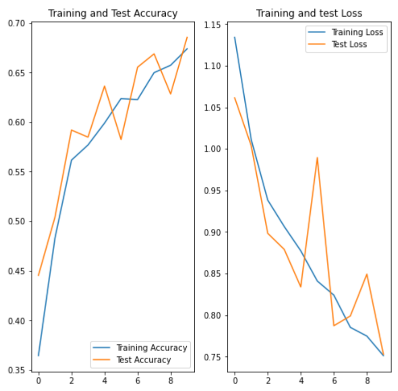
This network gave me a test accuracy score of 0.69. Performing a lot better than a model with no or a smaller dropout value.

Although this model still does not perform as well as I hoped it would. To try to increase the model accuracy I investigated changing the batch size. The model above is using a random batch size of 34. To create comparisons, I ran the model with a batch size of 1 and a batch size of 50.

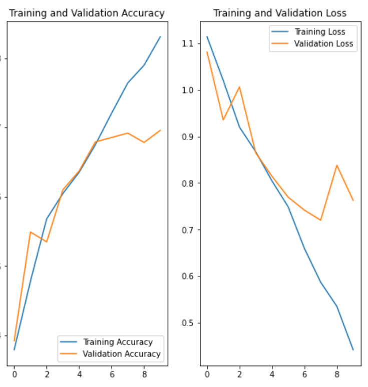
1



34



50

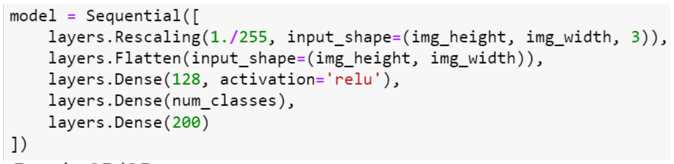


Using just one batch meant the model took a lot longer to run and it ended up not benefiting the model at all. The increased batch size has made the model even more accurate. It predicts correctly with 99.38% confidence. This is greater than the original model!
Using a bigger batch size also meant the model ran a lot faster too. Although there is some potential overfitting as the training and test accuracy scores aren't too similar. The network with a batch size of 50 has given a test accuracy score of 0.71.

My model is performing a lot better I investigated the number of epochs. To get a good idea of how the model was performing I used 10 epochs. However, as I have found the best-fitting model, I increased this value to 25. It takes a lot longer to run but it has produced the final model with a test accuracy of 0.72. This model also correctly identifies an image of a strawberry with 99.02 percent confidence.

MLP vs CNN

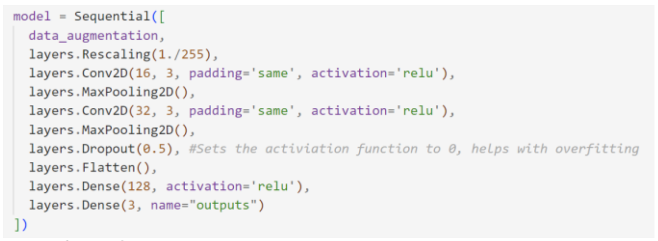
MLP



```
model = Sequential([
    layers.Rescaling(1./255, input_shape=(img_height, img_width, 3)),
    layers.Flatten(input_shape=(img_height, img_width)),
    layers.Dense(128, activation='relu'),
    layers.Dense(num_classes),
    layers.Dense(200)
])
```

```
Epoch 25/25
87/87 [==============================] - 1s 15ms/step - loss: 0.6773 - accuracy: 0.7197 - val_los
s: 1.1191 - val_accuracy: 0.4921
```

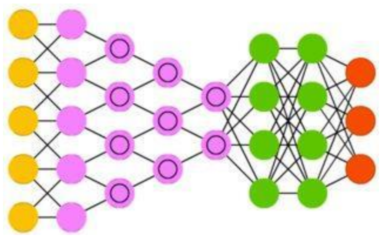
CNN



```
model = Sequential([
  data_augmentation,
  layers.Rescaling(1./255),
  layers.Conv2D(16, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(32, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Dropout(0.5), #Sets the activiation function to 0, helps with overfitting
  layers.Flatten(),
  layers.Dense(128, activation='relu'),
  layers.Dense(3, name="outputs")
])
```

```
Epoch 25/25
87/87 [==============================] - 13s 144ms/step - loss: 0.3812 - accuracy: 0.8422 - val_lo
ss: 0.8270 - val_accuracy: 0.7203
```

The basic neural network ran 25 epochs a lot faster than the CNN but it definitely compromised on accuracy the basic network had an accuracy of only 49% whereas the CNN had an accuracy of 72%. This is because the CNN has more layers, including pooling and dense layers. The CNN looks at more than one pixel at a time, learning different patterns. The CNN receives input from other surrounding layers whereas a neural network passes information from one layer to another.
Pictured below is an example of a CNN. The orange layers cross over and pool together before moving on through the network.



https://i.stack.imgur.com/p2qz1m.png

CNNs are better for images because like a human it takes in more than one pixel, and it relates pixels to other neighbouring pixels. Taking in more information to create patterns (images).

The final CNN model has a test accuracy score of 72%, this score is acceptable in my opinion for this context. The model will predict some images wrong but for this project, it is not going to have any devastating effects. If there was more risk involved, then I would expect a model with higher accuracy. Throughout this project, I learnt how to implement neural networks to correctly classify images. I found the TensorFlow tutorial extremely helpful as I didn't understand how to implement the tutorials provided to us in lectures.

If I was to do this project again, I would get better quality and quantities of images. I found that manually going through the training data took a long period of time. It also makes you think about what defines the object, is a cake topped with a strawberry classified as a strawberry? It would have been beneficial for the dataset to contain more images of the true object in the front and centre.

One specific negative about my model is that the data does not contain much noise. Will a bigger dataset it would be beneficial to add some noise to the data. It might increase the accuracy of the model.

Overall, I am proud of what I have completed. Although the model is defiantly not perfect it can predict images correctly most of the time.

TensorFlow image classification tutorial: https://www.tensorflow.org/tutorials/images/classification