DATA9005 – Data Analytics and Visualisation

Project 2: Neural Networks

Brian Higgins – R00239570

Introduction

This project will look at exploring Convoluted Neural Networks by first explaining how Neural Networks are put together in Question 1 with three examples and a description of each Neural Network. Question 2 will explore the 2.3GB dataset of images and use CNN's to classify these images by taking a baseline CNN and showing how different layers, filters and nodes can affect our accuracy results. We will also look at more complex techniques such as Transfer Learning and Resizing the images and how this can improve the results of our Neural Networks. Question 3 will then look at feature files instead of the original images to explore how we can use these with CNN's using R and Caret's Random Forests.

The data includes 8000 images of colonoscopies that shows 8 different classes or conditions and images seen below in Figure 1, such as Ulcerative Colitis, Normal Pylorus, Normal Cecum, and Polyps for example.

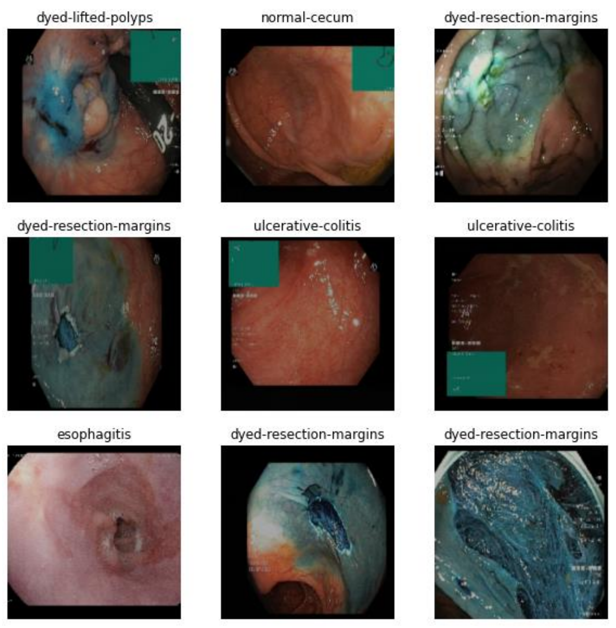


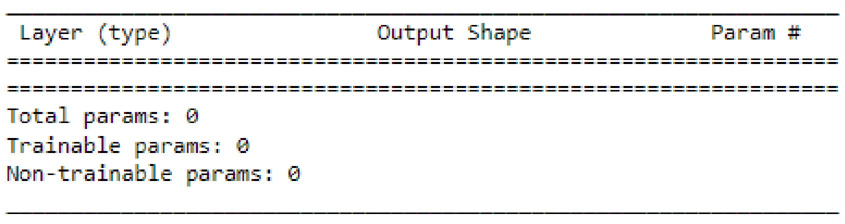
Figure 1

Question 1

2D Input with a 1D conv

Using Keras I created a 2D Input with a 1D conv which gives an output seen in Figure 2 where we can create the CNN with Keras before we use it to train any data. This created a Sequential Model with an input layer that expects a 2D input of shape (28,28,1) which is a grayscale image of 28 by 28 pixels with a single channel. The single channel indicates a greyscale colour system.

This model was set up with 32 filters in a 1D convolutional layer. In Figure 1 we can see a summary output of this model, the output shows that model has no layer and no trainable parameters at this point. This is because the model has only been defined but no layers of data have been added to it yet.



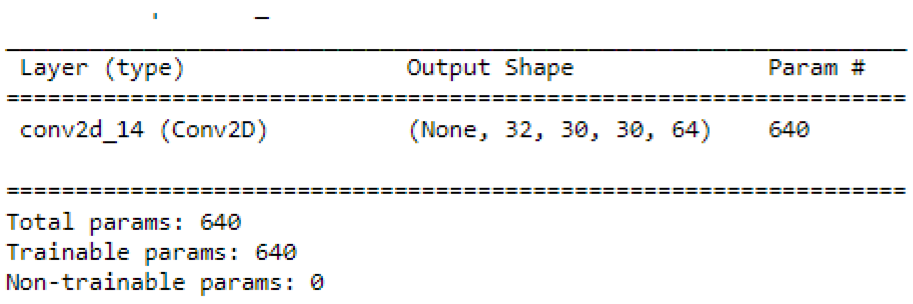
```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
=================================================================
Total params: 0
Trainable params: 0
Non-trainable params: 0

_________________________________________________________________
```

Figure 2: 2D Input with 1D conv

A Sequential Model is a linear stack of layers that we easily create more complexity by adding additional layers which can be added with the “add” method which we will see in the next models.

3D Input with a 2D conv

Here we have created a Sequential Model again with an input layer that expects a 3D input of shape (32,32,32,1) that corresponds to a grayscale volume(image) of size 32 by 32 by 32 pixels with a single channel. This model is similar to the previous model except that here we are now working with a 3D input instead of a 2D input. With image processing, this volume could be an MRI scan where each slice is a layer in the volume with the dimensions corresponding to the width, height and depth of the 3D scan. This volume example only has one colour channel (grayscale) which is common for medical scans. For example, in the images for this project, we have a value of (200,200,3) with images of 200 pixels and a 3-colour channel with Red(R), Green(G) and Blue (B).



```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_14 (Conv2D)          (None, 32, 30, 30, 64)    640

=================================================================
Total params: 640
Trainable params: 640
Non-trainable params: 0
```

Figure 3: 3D input with 2D conv

The output shape of the Con2D layer is (none, 32, 30, 30, 64) which means that the output of the layer is 3D volume and the height and width of 30 and 64 filters (depth) for each of the inputs in the batch. The total number of trainable parameters is 640, which is equal to the number of filters (64) multiplied by the size of the convolutional kernel(3x3) plus one bias term for each filter. So 3x3 = 9 +1 = 10 x 64 = 640.

2D Input with a 3D conv

For the third example, we once again use Keras to build a Sequential Model for a 3D Convolutional Neural Network we can see in Figure 4. The input shape is (28,28,1) which represents a grayscale image of size 28x28. The model adds an input layer of the same input shape that adds an additional channel dimension of 1 to the input. This is needed as 3D convolutions operate on volumes that have a depth dimension (as well as width and height).

The model then adds a 3D convolutional layer with 32 filters of size 3x3 and a ReLu activation. The kernel size is set as (3,3,1) which indicates that the convolutional operation should be applied to the width and height as well as the depth dimension (channel). The output shape of this layer is (None, 26,26,1,32) which means this layer takes a volume size of 28x28x1 and outputs to 26x26x1x32. The None is the batch size which can vary during training and evaluation. The trainable parameters are 320 calculated with 32x(3x3x1+1) =320.

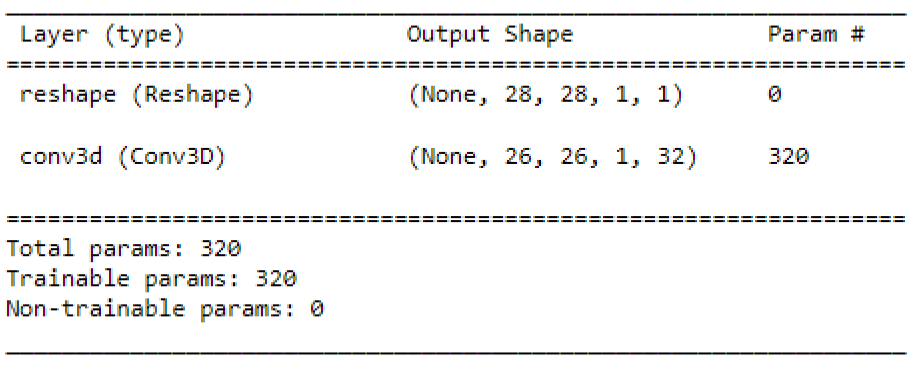

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 reshape (Reshape)           (None, 28, 28, 1, 1)      0

 conv3d (Conv3D)             (None, 26, 26, 1, 32)     320

=================================================================
Total params: 320
Trainable params: 320
Non-trainable params: 0
_________________________________________________________________
```

Figure 4: 2D Input with 3D conv

Question 2

0) General Points

You will see I have included two Jupyter Notebooks. One was ran on my home computer and has a lower number of Epochs. The original idea was to then increase this when I ran the code in the University GPUs. Since there are issues with them because of the Hack I was not able to use them (note below) so I also used Colab Pro to run my models. Code that was not used in either is commented out.

1) Split the data into Train, Validation and Test datasets

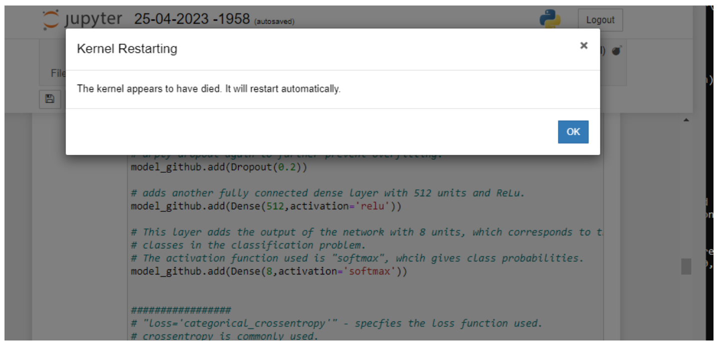
The research paper split the data into train and test. For our project, we split the data in a 75/15/15 percent split into Train, Validation and Test data (Section 6 in code). This resulted in 5600 images in the Train data and 1200 images in the Validation and Test datasets.

2) Challenges of setup configuration and environment setup

A project on Neural Networks offers additional challenges that we have not seen before. This is mainly down to the complexity of the models and the computer resources and time needed to run these models. Some of the challenges I found were:

1) **Laptop/ Desktop Computer Limitations:** I wanted to explore various filters, dense layers, nodes, etc and run models with different options but running larger epochs took many hours. So some of my examples are run with low epochs which means we did not get great results. Some models were run on larger epochs using Colab explained below.

2) **University GPU:** As discussed I created some code with low epochs on my home computer and then went to University to run the models with larger Epoch ranges overnight in lab 219 but I ran into issues. Figure 5 shows an example of the Kernel crashing repeatedly, and Figure 6 shows an issue where TensorFlow is not able to access the kernel for some reason (because of lack of internet I was not able to update the kernel myself as I did in assignment one) and this issue with the kernel matches figure 7 where you can see the GPU is not going above 1%. The lack of computer resources affected the results of models and highlights the complexity of Neural Networks and the resources that are needed to run them.



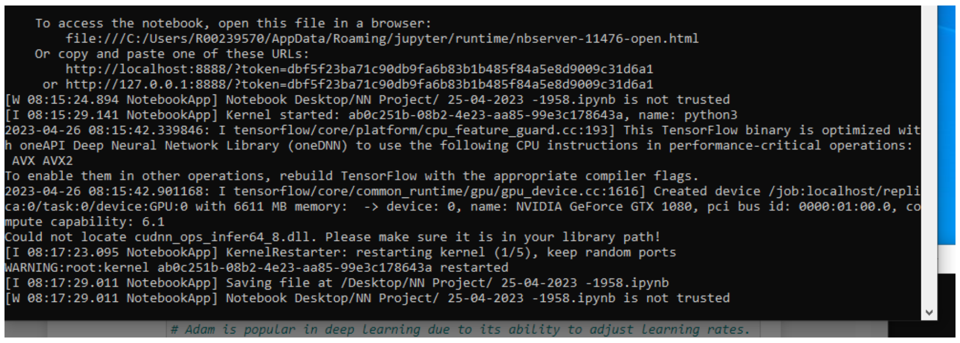
jupyter 25-04-2023 -1958 (autosaved)

Kernel Restarting

The kernel appears to have died. It will restart automatically.

```
model_github.add(Dropout(0.2))

# adds another fully connected dense layer with 512 units and ReLu.
model_github.add(Dense(512,activation='relu'))

# This layer adds the output of the network with 8 units, which corresponds to t
# classes in the classification problem.
# The activation function used is "softmax", whcih gives class probabilities.
model_github.add(Dense(8,activation='softmax'))


#################
# "loss='categorical_crossentropy'" - specfies the loss function used.
# crossentropy is commonly used.
```

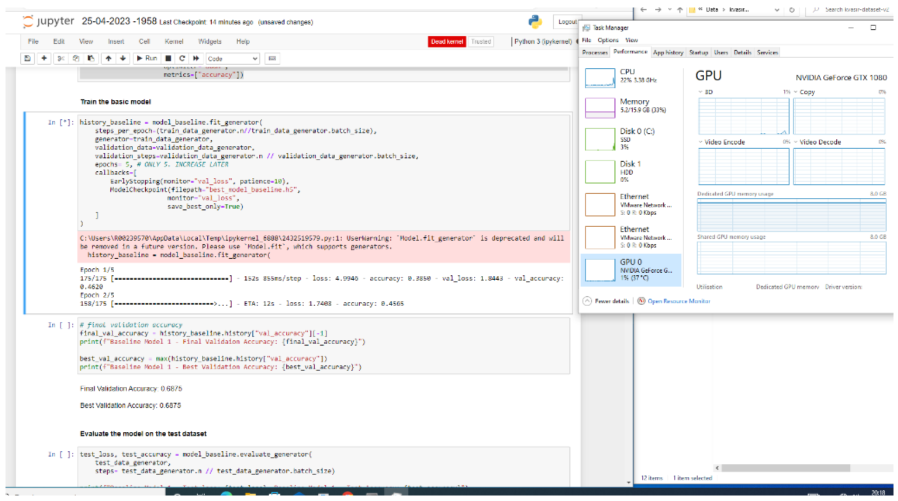
Figure 5: Dead Kernel



```
    To access the notebook, open this file in a browser:
        file:///C:/Users/R00239570/AppData/Roaming/jupyter/runtime/nbserver-11476-open.html
    Or copy and paste one of these URLs:
        http://localhost:8888/?token=dbf5f23ba71c90db9fa6b83b1b485f84a5e8d9009c31d6a1
     or http://127.0.0.1:8888/?token=dbf5f23ba71c90db9fa6b83b1b485f84a5e8d9009c31d6a1
[W 08:15:24.894 NotebookApp] Notebook Desktop/NN Project/ 25-04-2023 -1958.ipynb is not trusted
[I 08:15:29.141 NotebookApp] Kernel started: ab0c251b-08b2-4e23-aa85-99e3c178643a, name: python3
2023-04-26 08:15:42.339846: I tensorflow/core/platform/cpu_feature_guard.cc:193] This TensorFlow binary is optimized wit
h oneAPI Deep Neural Network Library (oneDNN) to use the following CPU instructions in performance-critical operations:
 AVX AVX2
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
2023-04-26 08:15:42.901168: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1616] Created device /job:localhost/repli
ca:0/task:0/device:GPU:0 with 6611 MB memory:  -> device: 0, name: NVIDIA GeForce GTX 1080, pci bus id: 0000:01:00.0, co
mpute capability: 6.1
Could not locate cudnn_ops_infer64_8.dll. Please make sure it is in your library path!
[I 08:17:23.095 NotebookApp] KernelRestarter: restarting kernel (1/5), keep random ports
WARNING:root:kernel ab0c251b-08b2-4e23-aa85-99e3c178643a restarted
[I 08:17:29.011 NotebookApp] Saving file at /Desktop/NN Project/ 25-04-2023 -1958.ipynb
[W 08:17:29.011 NotebookApp] Notebook Desktop/NN Project/ 25-04-2023 -1958.ipynb is not trusted
```

```
# Adam is popular in deep learning due to its ability to adjust learning rates.
```

Figure 6: TensorFlow issues



jupyter 25-04-2023 -1958 Last Checkpoint: 14 minutes ago (unsaved changes)

Dead kernel | Trusted | Python 3 (ipykernel)

```
               metrics=["accuracy"])
```

Train the basic model

```
In [*]: history_baseline = model_baseline.fit_generator(
            steps_per_epoch=(train_data_generator.n//train_data_generator.batch_size),
            generator=train_data_generator,
            validation_data=validation_data_generator,
            validation_steps=validation_data_generator.n // validation_data_generator.batch_size,
            epochs= 5, # ONLY 5. INCREASE LATER
            callbacks=[
                EarlyStopping(monitor="val_loss", patience=10),
                ModelCheckpoint(filepath="best_model_baseline.h5",
                                monitor="val_loss",
                                save_best_only=True)
            ]
        )
```

```
C:\Users\R00239570\AppData\Local\Temp\ipykernel_6888\2432519579.py:1: UserWarning: `Model.fit_generator` is deprecated and will
be removed in a future version. Please use `Model.fit`, which supports generators.
  history_baseline = model_baseline.fit_generator(
Epoch 1/5
175/175 [==============================] - 152s 855ms/step - loss: 4.9946 - accuracy: 0.3850 - val_loss: 1.8443 - val_accuracy:
0.4620
Epoch 2/5
158/175 [==========================>...] - ETA: 12s - loss: 1.7408 - accuracy: 0.4565
```

```
In [ ]: # final validation accuracy
        final_val_accuracy = history_baseline.history["val_accuracy"][-1]
        print(f"Baseline Model 1 - Final Validaion Accuracy: {final_val_accuracy}")

        best_val_accuracy = max(history_baseline.history["val_accuracy"])
        print(f"Baseline Model 1 - Best Validation Accuracy: {best_val_accuracy}")
```

Final Validation Accuracy: 0.6875

Best Validation Accuracy: 0.6875

Evaluate the model on the test dataset

```
In [ ]: test_loss, test_accuracy = model_baseline.evaluate_generator(
            test_data_generator,
            steps= test_data_generator.n // test_data_generator.batch_size)
```

Task Manager

GPU NVIDIA GeForce GTX 1080

CPU 22% 3.38 GHz
Memory 5.2/15.9 GB (33%)
Disk 0 (C:) SSD 3%
Disk 1 HDD 0%
Ethernet VMware Network ... S: 0 R: 0 Kbps
Ethernet VMware Network ... S: 0 R: 0 Kbps
GPU 0 NVIDIA GeForce G... 1% (37 °C)

Figure 7: GPU not working

3) **Google Colab Pro:** I did pay for a Google subscription as I did not want to miss the opportunity to explore NN and this was an opportunity to explore using Colab. A Pro subscription cost 11.38 euros per month and gave 100 units and a Colab Pro Plus cost 51.97 euros and gave you 500 units. I paid for the Pro Plus as I wanted to have access to the 500 units (which I will discuss below). Using Colab Pro Plus also added some additional challenges:
    a. **File Transfer Bottleneck:** Using Google Drive to store the 8000 images (2.3Gb) and then run Neural Network Models meant this was a bottleneck for the CNN to access the images and meant that the Epochs ran very slowly. Much slower than even on my home computer, I was initially not getting the benefit of the online GPU. I was able to work around this by loading the images into RAM and then running NN models. This greatly sped up the running of epochs and models.
    b. **Colab Credits:** A Colab Pro Plus gives you 500 credits. Running my models for 20/30 epochs used up all 500 units in less than two days. Although it was very interesting to see how the models performed over many more epochs. Note I did not have any credits left to run some later stages with for transfer learning/resizing images but we can see their potential with the limited amount of epochs I was able to run.

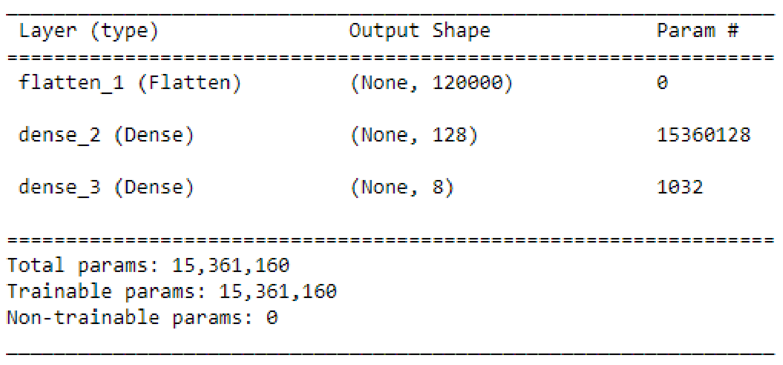
4) **Exploring Results:** Because models would take time and an expensive to run I found it a little harder to explore results. Deciding to run a model again required thought because it could take hours before a result was obtained. An interesting situation that was a good lesson for later projects.

5) **Multiple Juypter notebooks:** Because I was running models on my local machine and on Colab I had two notebooks with the different model results. The results are discussed in this report but you may notice some code that is commented out in both notebooks or results that are missing. It was confusing to have two models running at the same time but again is a good lesson on the complexity of running CNNs.

3) Create a baseline model

Baseline Model 1

I created two basic baseline models as I wanted to start with a simple model and then look at how factors such as the number of filters, dense layers and nodes would change the results. Figure 8 shows the first baseline model. For the first baseline, we will explore it in greater depth. Note for this mode I have not added in the image size, etc as I wanted to see how a basic model performed.



```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten_1 (Flatten)         (None, 120000)            0

 dense_2 (Dense)             (None, 128)               15360128

 dense_3 (Dense)             (None, 8)                 1032

=================================================================
Total params: 15,361,160
Trainable params: 15,361,160
Non-trainable params: 0
_________________________________________________________________
```

Figure 8: Baseline model

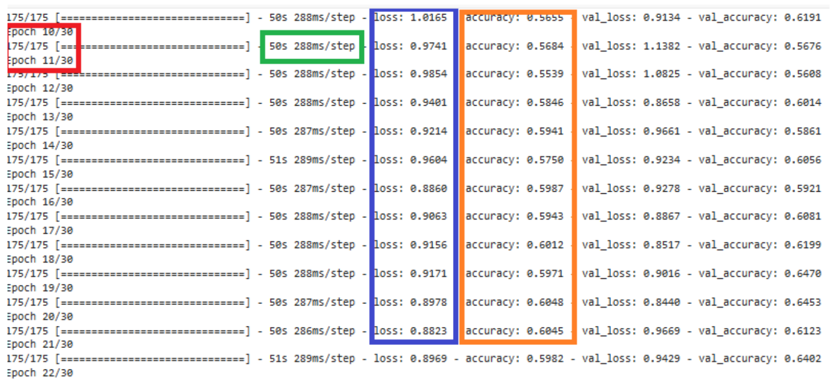
Baseline_1 has two dense layers:

The first layer is a flattened layer which flattens the input image into a 1-dimensional array. The output of this flattened layer is 120,000 with a zero batch size.

The second layer is a dense layer with 128 neurons and it has 15,360,128 parameters to train.

The third layer has another dense layer with 8 neurons which represent the number of classes in the classification task. It has 1032 parameters to train. The total number of trainable parameters we can see is 12,361,160

Using Colab with a GPU enabled you can see I ran 30 epochs to explore how well the basic model performed.



```
175/175 [==============================] - 50s 288ms/step - loss: 1.0165 - accuracy: 0.5655 - val_loss: 0.9134 - val_accuracy: 0.6191
Epoch 10/30
175/175 [==============================] - 50s 288ms/step - loss: 0.9741 - accuracy: 0.5684 - val_loss: 1.1382 - val_accuracy: 0.5676
Epoch 11/30
175/175 [==============================] - 50s 288ms/step - loss: 0.9854 - accuracy: 0.5539 - val_loss: 1.0825 - val_accuracy: 0.5608
Epoch 12/30
175/175 [==============================] - 50s 288ms/step - loss: 0.9401 - accuracy: 0.5846 - val_loss: 0.8658 - val_accuracy: 0.6014
Epoch 13/30
175/175 [==============================] - 50s 287ms/step - loss: 0.9214 - accuracy: 0.5941 - val_loss: 0.9661 - val_accuracy: 0.5861
Epoch 14/30
175/175 [==============================] - 51s 289ms/step - loss: 0.9604 - accuracy: 0.5750 - val_loss: 0.9234 - val_accuracy: 0.6056
Epoch 15/30
175/175 [==============================] - 50s 287ms/step - loss: 0.8860 - accuracy: 0.5987 - val_loss: 0.9278 - val_accuracy: 0.5921
Epoch 16/30
175/175 [==============================] - 50s 288ms/step - loss: 0.9063 - accuracy: 0.5943 - val_loss: 0.8867 - val_accuracy: 0.6081
Epoch 17/30
175/175 [==============================] - 50s 288ms/step - loss: 0.9156 - accuracy: 0.6012 - val_loss: 0.8517 - val_accuracy: 0.6199
Epoch 18/30
175/175 [==============================] - 50s 288ms/step - loss: 0.9171 - accuracy: 0.5971 - val_loss: 0.9016 - val_accuracy: 0.6470
Epoch 19/30
175/175 [==============================] - 50s 287ms/step - loss: 0.8978 - accuracy: 0.6048 - val_loss: 0.8440 - val_accuracy: 0.6453
Epoch 20/30
175/175 [==============================] - 50s 286ms/step - loss: 0.8823 - accuracy: 0.6045 - val_loss: 0.9669 - val_accuracy: 0.6123
Epoch 21/30
175/175 [==============================] - 51s 289ms/step - loss: 0.8969 - accuracy: 0.5982 - val_loss: 0.9429 - val_accuracy: 0.6402
Epoch 22/30
```

Figure 9: Baseline Model 2

Here is an example of 30 epochs run for this model.

The Red square shows the epoch number, the Green box shows the time it takes to run which can vary greatly depending on whether my home computer or Colab Pro was used.

The Blue box shows how the loss function performs. Even on this simple model, we can see it dropping which is a good result.

The Orange box shows the accuracy which is also increasing and shows a general increase as the epochs run. Note, this can go up and down as the model trains.

Baseline 1: Results	Loss	Accuracy
Final Validation Model	0.927	63.4%
Best Validation Model	0.839	64.6%
Test Data	0.8704	62.6%

Table 10

The final model in the CNN is not always the best as we can see. As the loss function changes the weights it will not always improve on the model. So we can see that the final model here is not the best. The best model got an accuracy of 65% for this basic baseline model which is not bad for a very basic model.

The model did not perform as well on the test data but still not a bad result for our very first model with an accuracy score of 63%.

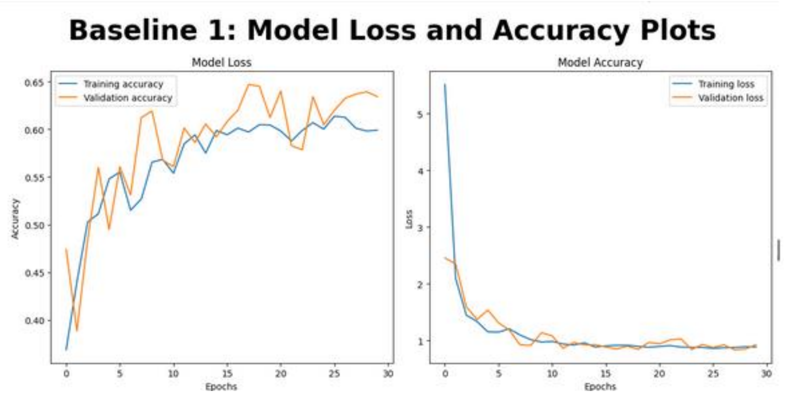


Figure 11: Plot results

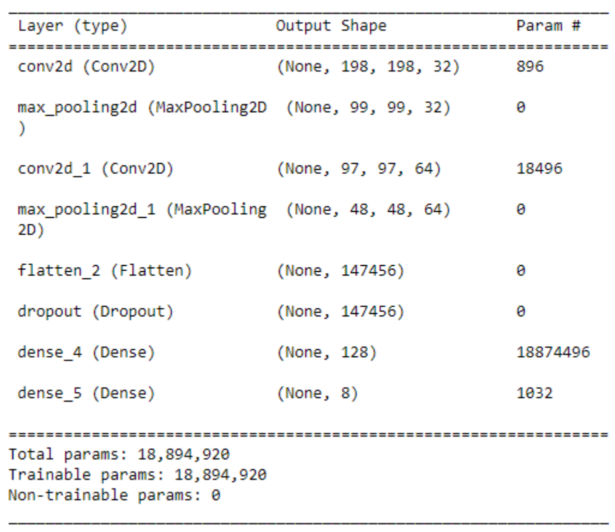
You can see in the accuracy values that they increase over time, the training accuracy does not function as much as the validation accuracy showing how the model is adjusting and learning over time.

We can see in the loss score that it drops quickly but then starts to plateau. As this is a simple model the model is only able to learn so much so the higher epochs are not making as much of a difference.

We shall see later how this is affected as our models get more complex.

Baseline Model 2

My first baseline model did well but before I moved on to explore other parameters I wanted to create a second baseline model to see how it performed and use this as the baseline for further testing, see Figure 11. This one is more like what we are expecting to see, with a filter of 3x3 and an input shape of (200,200,3). Again I ran this model on Colab Pro with 30 Epochs with an example shown in Figure 11.



```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 198, 198, 32)      896

 max_pooling2d (MaxPooling2D  (None, 99, 99, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 97, 97, 64)        18496

 max_pooling2d_1 (MaxPooling  (None, 48, 48, 64)       0
 2D)

 flatten_2 (Flatten)         (None, 147456)            0

 dropout (Dropout)           (None, 147456)            0

 dense_4 (Dense)             (None, 128)               18874496

 dense_5 (Dense)             (None, 8)                 1032

=================================================================
Total params: 18,894,920
Trainable params: 18,894,920
Non-trainable params: 0
_________________________________________________________________
```

Figure 12: Baseline 2 model

Baseline_2 Model

The model has two Conv2D layers with 32 and 64 filters in each with both using a 3x3 kernel and ReLu activation function.

Each Conv2D layer is followed by a MaxPooling2D layer with a pool size of 2x2. The output is then flattened and a Dropout layer with a rate of 0.5 is applied for regularization.

Two dense layers are added with the first one containing 128 nodes and the last one containing 8 nodes with a SoftMax activation function.

The first layer parameters are 32(3x3x3+1) bias =896. Third layer is 64(32x3x3+1) bias =18,496. The fourth layer is 48x48x64=147,456. Seventh is 147,456 x128 = 18,874,368. Total is 18,894,920.

Baseline 2: Results	**Loss**	**Accuracy**
Final Validation Model	0.673	70.9%
Best Validation Model	0.581	75.5%
Test Data	0.708	65.6%

Table 13

This baseline model is much better for the Validation Data. The best Model Is 76% with a loss value of 0.58. This is a great reduction for a guess at adding in some extra items like filters and image size.

The test data does not perform as well. The accuracy drops to 65% which is a considerable drop.

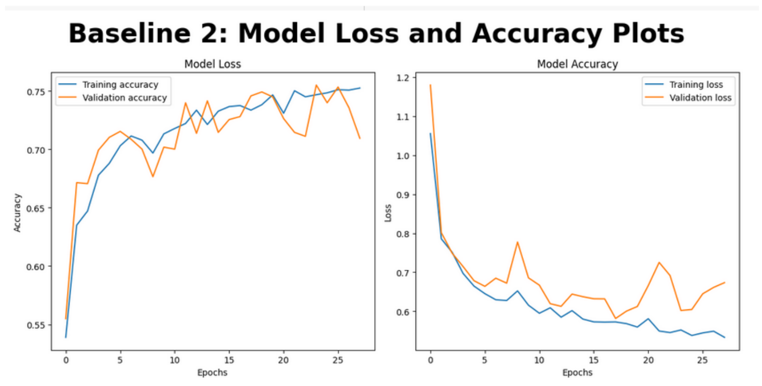
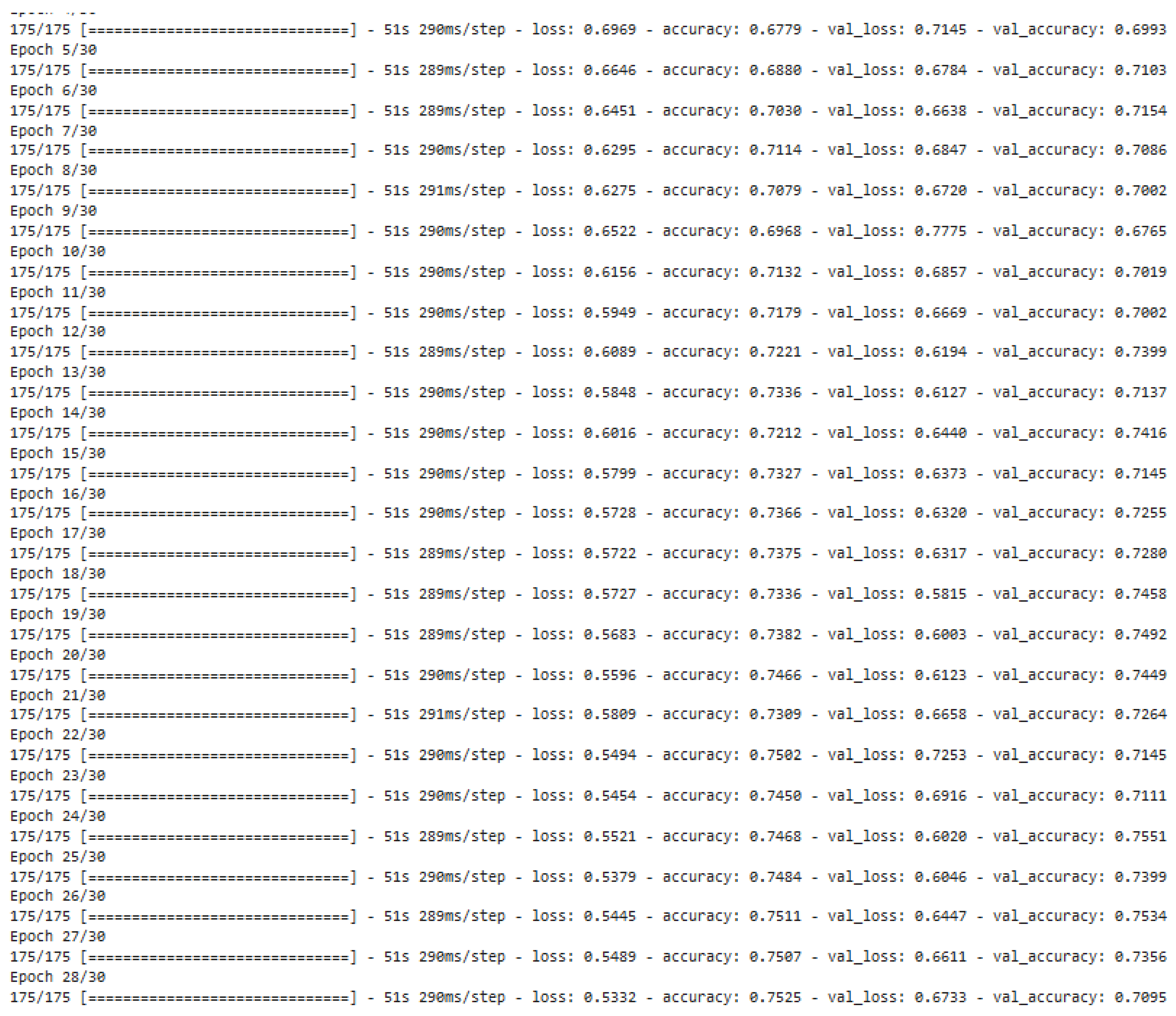


Figure 14: Plot results

You can see in the accuracy values are doing much better in this model and has a very positive upward trend. We could continue to try more epochs as it has not levelled off yet.

The training loss also is continuing to drop so again, more epochs could lead to better results. The validation has started to increase so the model maybe is on a curve to overfitting. More epochs are needed but we will explore some more parameters first.



```
175/175 [==============================] - 51s 290ms/step - loss: 0.6969 - accuracy: 0.6779 - val_loss: 0.7145 - val_accuracy: 0.6993
Epoch 5/30
175/175 [==============================] - 51s 289ms/step - loss: 0.6646 - accuracy: 0.6880 - val_loss: 0.6784 - val_accuracy: 0.7103
Epoch 6/30
175/175 [==============================] - 51s 289ms/step - loss: 0.6451 - accuracy: 0.7030 - val_loss: 0.6638 - val_accuracy: 0.7154
Epoch 7/30
175/175 [==============================] - 51s 290ms/step - loss: 0.6295 - accuracy: 0.7114 - val_loss: 0.6847 - val_accuracy: 0.7086
Epoch 8/30
175/175 [==============================] - 51s 291ms/step - loss: 0.6275 - accuracy: 0.7079 - val_loss: 0.6720 - val_accuracy: 0.7002
Epoch 9/30
175/175 [==============================] - 51s 290ms/step - loss: 0.6522 - accuracy: 0.6968 - val_loss: 0.7775 - val_accuracy: 0.6765
Epoch 10/30
175/175 [==============================] - 51s 290ms/step - loss: 0.6156 - accuracy: 0.7132 - val_loss: 0.6857 - val_accuracy: 0.7019
Epoch 11/30
175/175 [==============================] - 51s 290ms/step - loss: 0.5949 - accuracy: 0.7179 - val_loss: 0.6669 - val_accuracy: 0.7002
Epoch 12/30
175/175 [==============================] - 51s 289ms/step - loss: 0.6089 - accuracy: 0.7221 - val_loss: 0.6194 - val_accuracy: 0.7399
Epoch 13/30
175/175 [==============================] - 51s 290ms/step - loss: 0.5848 - accuracy: 0.7336 - val_loss: 0.6127 - val_accuracy: 0.7137
Epoch 14/30
175/175 [==============================] - 51s 290ms/step - loss: 0.6016 - accuracy: 0.7212 - val_loss: 0.6440 - val_accuracy: 0.7416
Epoch 15/30
175/175 [==============================] - 51s 290ms/step - loss: 0.5799 - accuracy: 0.7327 - val_loss: 0.6373 - val_accuracy: 0.7145
Epoch 16/30
175/175 [==============================] - 51s 290ms/step - loss: 0.5728 - accuracy: 0.7366 - val_loss: 0.6320 - val_accuracy: 0.7255
Epoch 17/30
175/175 [==============================] - 51s 289ms/step - loss: 0.5722 - accuracy: 0.7375 - val_loss: 0.6317 - val_accuracy: 0.7280
Epoch 18/30
175/175 [==============================] - 51s 289ms/step - loss: 0.5727 - accuracy: 0.7336 - val_loss: 0.5815 - val_accuracy: 0.7458
Epoch 19/30
175/175 [==============================] - 51s 289ms/step - loss: 0.5683 - accuracy: 0.7382 - val_loss: 0.6003 - val_accuracy: 0.7492
Epoch 20/30
175/175 [==============================] - 51s 290ms/step - loss: 0.5596 - accuracy: 0.7466 - val_loss: 0.6123 - val_accuracy: 0.7449
Epoch 21/30
175/175 [==============================] - 51s 291ms/step - loss: 0.5809 - accuracy: 0.7309 - val_loss: 0.6658 - val_accuracy: 0.7264
Epoch 22/30
175/175 [==============================] - 51s 290ms/step - loss: 0.5494 - accuracy: 0.7502 - val_loss: 0.7253 - val_accuracy: 0.7145
Epoch 23/30
175/175 [==============================] - 51s 290ms/step - loss: 0.5454 - accuracy: 0.7450 - val_loss: 0.6916 - val_accuracy: 0.7111
Epoch 24/30
175/175 [==============================] - 51s 289ms/step - loss: 0.5521 - accuracy: 0.7468 - val_loss: 0.6020 - val_accuracy: 0.7551
Epoch 25/30
175/175 [==============================] - 51s 290ms/step - loss: 0.5379 - accuracy: 0.7484 - val_loss: 0.6046 - val_accuracy: 0.7399
Epoch 26/30
175/175 [==============================] - 51s 289ms/step - loss: 0.5445 - accuracy: 0.7511 - val_loss: 0.6447 - val_accuracy: 0.7534
Epoch 27/30
175/175 [==============================] - 51s 290ms/step - loss: 0.5489 - accuracy: 0.7507 - val_loss: 0.6611 - val_accuracy: 0.7356
Epoch 28/30
175/175 [==============================] - 51s 290ms/step - loss: 0.5332 - accuracy: 0.7525 - val_loss: 0.6733 - val_accuracy: 0.7095
```

Figure 15: Example of Epochs for Baseline 2 Model

Baseline Model 2 comparison with Research Paper Results

The research paper had accuracy results for 6 layers and 3-layer models with results of 91.4% and 95.9% respectively. But they had the resources and time to run 200 epochs. This was something I did not have the resources to do as already mentioned. I built my baseline 2 models using some of their approaches with ReLu activation, 0.5 dropout, etc and the best result I got for my 30 Epochs was 65% for Test Accuracy. In the next section, I will explore different values for layers, filters, nodes, etc to see how this can affect the performance of the models and to look If I can improve on my model results even with my limitation of epochs.

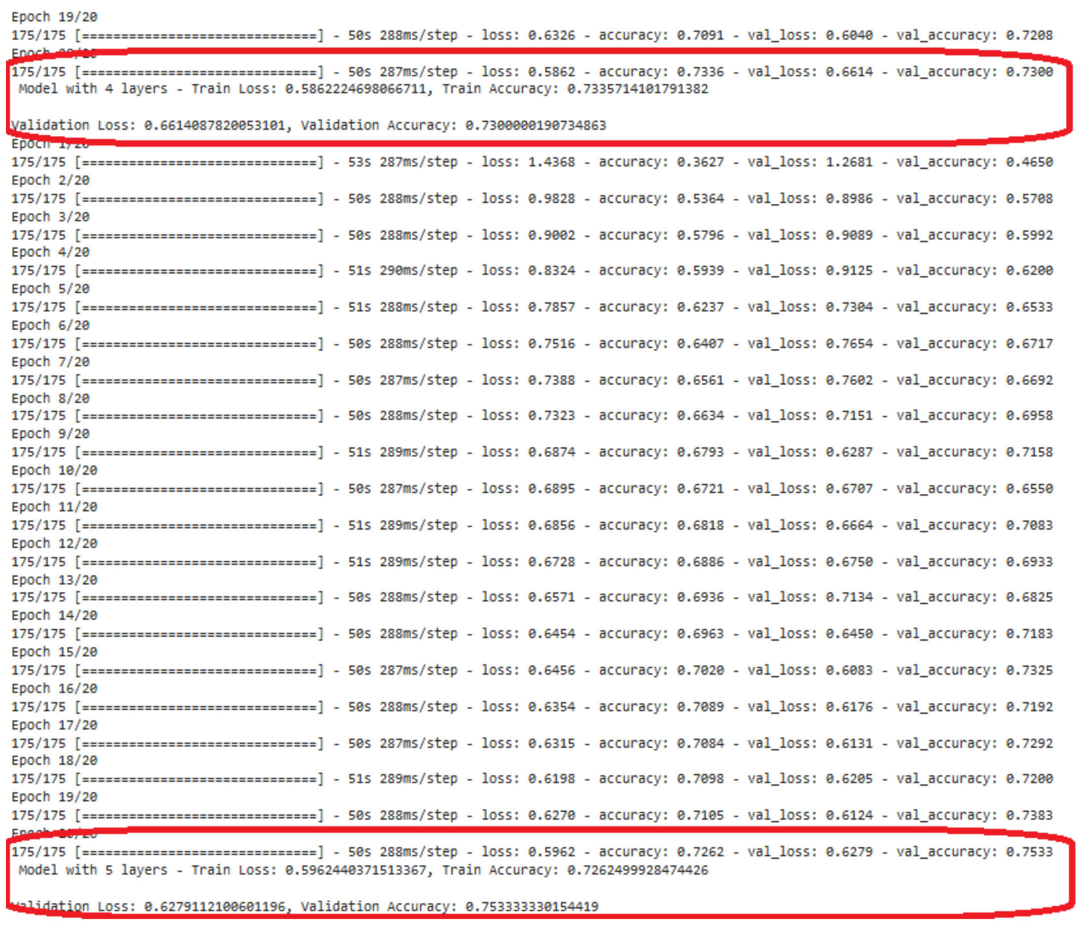
4) Change some parameters and show effects on performance

CNNs have several parameters that we can change that will affect the performance in different ways. We can look at changing the number of Layers, Filters, Dense layers and Nodes and look to see how that affects our baseline 2 accuracies. Rather than put in random values I have created a function for each of these options and run through some options to see how they can have an effect on performance.

Number of Layers Function

The number of layers in a neural network affects its ability to learn complex patterns in the input data. Usually adding more layers will increase its capacity to model complex relationships in the data. However, there is a sweet spot as having too many layers can also lead to overfitting. The number of layers will also affect the training time that models take to run so it's important to strike the right balance.

For our Layer Function, we have put in a range from 1 to 5 layers that run for 20 epochs each. This function took 84 minutes to run on Colab Pro using a GPU as it first ran with 1 layer, then 2 layers, up to 5 layers as we can see in Figure 16. In the table below we are going to evaluate the Train and Validation results.



```
Epoch 19/20
175/175 [==============================] - 50s 288ms/step - loss: 0.6326 - accuracy: 0.7091 - val_loss: 0.6040 - val_accuracy: 0.7208
Epoch 20/20
175/175 [==============================] - 50s 287ms/step - loss: 0.5862 - accuracy: 0.7336 - val_loss: 0.6614 - val_accuracy: 0.7300
 Model with 4 layers - Train Loss: 0.5862224698066711, Train Accuracy: 0.7335714101791382

Validation Loss: 0.6614087820053101, Validation Accuracy: 0.7300000190734863
Epoch 1/20
175/175 [==============================] - 53s 287ms/step - loss: 1.4368 - accuracy: 0.3627 - val_loss: 1.2681 - val_accuracy: 0.4650
Epoch 2/20
175/175 [==============================] - 50s 288ms/step - loss: 0.9828 - accuracy: 0.5364 - val_loss: 0.8986 - val_accuracy: 0.5708
Epoch 3/20
175/175 [==============================] - 50s 288ms/step - loss: 0.9002 - accuracy: 0.5796 - val_loss: 0.9089 - val_accuracy: 0.5992
Epoch 4/20
175/175 [==============================] - 51s 290ms/step - loss: 0.8324 - accuracy: 0.5939 - val_loss: 0.9125 - val_accuracy: 0.6200
Epoch 5/20
175/175 [==============================] - 51s 288ms/step - loss: 0.7857 - accuracy: 0.6237 - val_loss: 0.7304 - val_accuracy: 0.6533
Epoch 6/20
175/175 [==============================] - 50s 288ms/step - loss: 0.7516 - accuracy: 0.6407 - val_loss: 0.7654 - val_accuracy: 0.6717
Epoch 7/20
175/175 [==============================] - 50s 287ms/step - loss: 0.7388 - accuracy: 0.6561 - val_loss: 0.7602 - val_accuracy: 0.6692
Epoch 8/20
175/175 [==============================] - 50s 288ms/step - loss: 0.7323 - accuracy: 0.6634 - val_loss: 0.7151 - val_accuracy: 0.6958
Epoch 9/20
175/175 [==============================] - 51s 289ms/step - loss: 0.6874 - accuracy: 0.6793 - val_loss: 0.6287 - val_accuracy: 0.7158
Epoch 10/20
175/175 [==============================] - 50s 287ms/step - loss: 0.6895 - accuracy: 0.6721 - val_loss: 0.6707 - val_accuracy: 0.6550
Epoch 11/20
175/175 [==============================] - 51s 289ms/step - loss: 0.6856 - accuracy: 0.6818 - val_loss: 0.6664 - val_accuracy: 0.7083
Epoch 12/20
175/175 [==============================] - 51s 289ms/step - loss: 0.6728 - accuracy: 0.6886 - val_loss: 0.6750 - val_accuracy: 0.6933
Epoch 13/20
175/175 [==============================] - 50s 288ms/step - loss: 0.6571 - accuracy: 0.6936 - val_loss: 0.7134 - val_accuracy: 0.6825
Epoch 14/20
175/175 [==============================] - 50s 288ms/step - loss: 0.6454 - accuracy: 0.6963 - val_loss: 0.6450 - val_accuracy: 0.7183
Epoch 15/20
175/175 [==============================] - 50s 287ms/step - loss: 0.6456 - accuracy: 0.7020 - val_loss: 0.6083 - val_accuracy: 0.7325
Epoch 16/20
175/175 [==============================] - 50s 288ms/step - loss: 0.6354 - accuracy: 0.7089 - val_loss: 0.6176 - val_accuracy: 0.7192
Epoch 17/20
175/175 [==============================] - 50s 287ms/step - loss: 0.6315 - accuracy: 0.7084 - val_loss: 0.6131 - val_accuracy: 0.7292
Epoch 18/20
175/175 [==============================] - 51s 289ms/step - loss: 0.6198 - accuracy: 0.7098 - val_loss: 0.6205 - val_accuracy: 0.7200
Epoch 19/20
175/175 [==============================] - 50s 288ms/step - loss: 0.6270 - accuracy: 0.7105 - val_loss: 0.6124 - val_accuracy: 0.7383
Epoch 20/20
175/175 [==============================] - 50s 288ms/step - loss: 0.5962 - accuracy: 0.7262 - val_loss: 0.6279 - val_accuracy: 0.7533
 Model with 5 layers - Train Loss: 0.5962440371513367, Train Accuracy: 0.7262499928474426

Validation Loss: 0.6279112100601196, Validation Accuracy: 0.753333330154419
```

Figure 16: Layer Function Run example

Layer Number	Train Data: Loss Results	Train Data: Accuracy Results	Val Data: Loss Results	Val Data: Accuracy Results
1 Layer	0.659	69.3%	0.728	70%
2 Layers	0.586	72.8%	0.682	68.8%
3 Layers	0.595	72.4%	0.614	73.4%
4 Layers	0.586	73.3%	0.661	73%
5 Layers	0.595	72.6%	0.627	75%

At times some previous layers do perform better than later layers but it is still possible to see a general trend even for 20 Epochs (a low amount for CNNs) and we can see a general lowering of the Loss values and an increase in the Accuracy over epochs.

This is matched by the plots in Figure 17 which show a general trend but there are fluctuations which are to be expected in Neural Networks as the layers adjust.

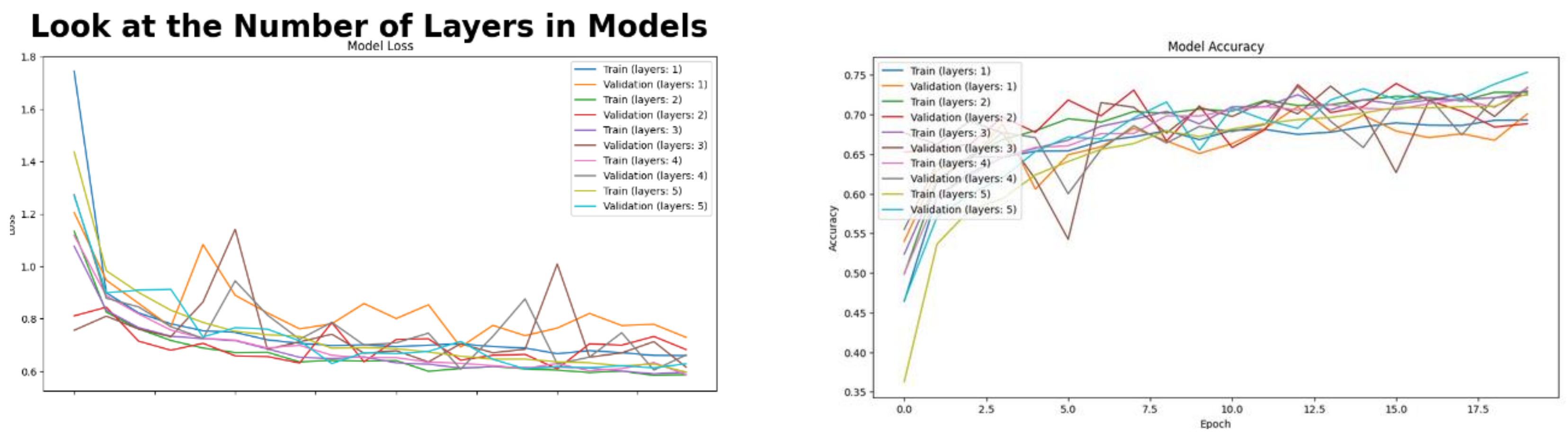


Figure 17: Layer Results

Number of filters function

The filter size is an important hyperparameter that affects the performance of CNNs. The size of the filter filters determines the field of each convolutional layer which refers to the region of the input image that affects the output of a single neuron. Larger filters allow the NN to capture more complex patterns but can also decrease the number of parameters in the model. Smaller filter sizes make the models more sensitive to small patterns and the edges of the input image.

For our filters function, we looked at filter sizes of 8, 16, 32, 64 and 128. Once again the function ran for 20 epochs for each of these values and took 111 minutes to run using Colab Pro with a GPU. The runs follow a simple pattern to Figure 16.

Filter Size	Train Data: Loss Results	Train Data: Accuracy Results	Val Data: Loss Results	Val Data: Accuracy Results
8	0.596	73.3%	0.712	69.3%
16	0.606	71.5%	0.694	67%
32	0.593	72.7%	0.618	73.4%
64	0.599	71.7%	0.602	73.8%
128	0.662	72.1%	0.662	70.6%

This time we are not looking to see what number of filters performs best we are looking at what filter size works best with the data.

We can see from the training data that there are generally similar results (with 8 doing the best) but for the validation data, we can see that 32 or 64 filter sizes do the best.

Again, we would need to run more epochs but this does give us an idea of where to start when we manually created models later.

Note: My code crashed during the night on Colab while I ran this code and so I do not have the plots but this is a good example of the challenges with running Neural Networks. I would now have to run the model again as Colab times out and does not save the workspace, unlike my home machine. Not having a GPU means I can’t do this easily.

Number of the Dense Layers function

Dense Layers are a type of layer that is commonly used in Neural Networks that are often that are used at the end of a Neural Network to perform classifications or regression based on the features. The number of dense layers is determined by the complexity of the task and the amount of data available. A simple task might need only one dense layer and more complex tasks may need multiple dense layers to capture the relationship between the input and output data.

For our function, we took a range of dense layers from 1 to 5 which allowed us to gradually increase the complexity of the model. Each dense layer can learn different types of features and by stacking dense layers we can learn to capture complex relationships. However, having too many dense layers can lead to overfitting. This function took 94 minutes to run with Colab Pro.

Dense layers	Train Data: Loss Results	Train Data: Accuracy Results	Val Data: Loss Results	Val Data: Accuracy Results
1	0.585	72.9%	0.607	73.7%
2	0.59	72.9%	0.716	67.5%
3	0.578	72.2%	0.572	74.8%
4	0.585	72.4%	0.627	73.3%
5	0.60.2	72.3%	0.73.2	68.8%

The Train accuracy is generally the same for 20 epochs. Whereas the validation accuracy gives the best results for 1,3,4 dense layers.

With more time I would have run 50 epochs for these three to see how they compare.

There is not enough information to give us an idea of how many dense layers to use.

Number of the Nodes Function

The number of nodes in a dense layer can also change the complexity of a CNN. Increasing the number of nodes in a dense layer can increase the complexity and the capacity of the network to let it learn more complex relationships in data. This can also lead to overfitting especially if the number of nodes is much larger than the number of available data points. We took values from 64 to 198 starting with smaller values to try and prevent overfitting by starting with a simpler model. This function took 38 minutes to run for 20 epochs.

Nodes	Train Data: Loss Results	Train Data: Accuracy Results	Val Data: Loss Results	Val Data: Accuracy Results
64	0.679	68.9%	0.718	29.1%
128	0.641	70.1%	0.604	73.9%
198	0.689	67.8%	0.687	74.8%

Using a range of values we can see that 128 nodes give the best results with lower and higher values for 64 and 198 nodes.

Once again, running for larger epochs would be helpful.

Function to combine all of the above

I wanted to create a function that would try different values of the above options and look to see if an approach like this would be useful with CNN. I no longer believe this given the amount of variables and the time it takes with my limited computer resources. Sure, if I had a large GPU I would let this run for a week and it would be helpful. But as I only had my home computer (as I had used the credits from Colab) this took 8 hours to run and this was only with 3 epochs!!! Three. Multiple attempts to run this led to crashes and just wasted my Colab credits.

I set the number of layers from 1 to 5, filters from 16,32,64,128, the number of dense layers from 1 to 5 and the number of nodes from 64 to 256. The best results were 72.1% but again, this was for only 3 epochs so would need to be run for a much larger number of epochs. Although this relatively "simple" example took over 8 hours to run as it went through each combination.

Run 1: Best hyperparameters:

Number of convolutional layers: 4
Initial Number of filters: 32
Number of dense layers: 1
Number of nodes in dense layers: 64
Best test accuracy: 65.1%

Run 2: Best hyperparameters:

Number of convolutional layers: 3
Initial Number of filters: 16
Number of dense layers: 1
Number of nodes in dense layers: 64
Best test accuracy: 67%

Run 3: Best hyperparameters:

Number of convolutional layers: 2
Initial Number of filters: 128
Number of dense layers: 1
Number of nodes in dense layers: 256
Best test accuracy: 72.1%

5) Change the model with the below

Transfer Learning

Transfer Learning is a technique in Neural Networks and Deep Learning where a pre-trained model is fine-tuned to solve a different but related problem. The idea is to leverage the knowledge gained from solving one task to improve the performance of another task.

In Transfer Learning, a neural network is first trained on a large dataset such as Image Net for image classification tasks. This pre-trained model has already learned many useful features and patterns already. These learned features can then be used on a new task like ours. We are taking other weights to help train our model better. Some steps used:

1) **Select a pre-trained model**. Get a NN architecture that has been trained on a large dataset.
2) **Remove the last layer**. You don't want the final classification layers as these are specific to the original task
3) **Add New layers**. Add new layers to the model to use it for the new task.
4) **Fine Tune the model**. Train the model on the new dataset.

Good for small data as it helps to prevent overfitting by using the knowledge from the pre-trained model. It also reduces training time and computational resources.

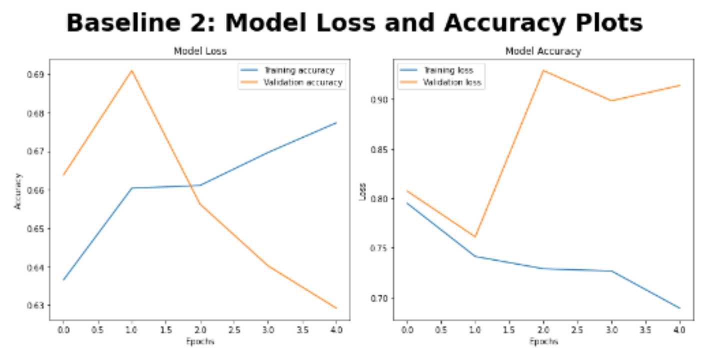
“MobileNetV2” is a CNN architecture that is designed for mobile and embedded devices. It is optimized for high accuracy with low computation and memory requirements. By using pre-trained weights from the ImageNet dataset, the model can fine-tune our smaller dataset for our classification task.

Because of a lack of Computer resources and time, I will be using my baseline 2 model as a comparison. This means only 5 Epochs were used to run the models but it still gives us an idea of the power of transfer learning.

Models	**Best Train Loss**	**Best Train Loss**		**Best Validation Loss**	**Best Accuracy Loss**		**Test Data Loss**	**Test Data Accuracy**
Baseline 2 CNN	0.990	57.3%		0.761	69%		0.801	76.1%
Baseline 2 + ImageNet weights CNN	0.399	86.2%		2.36	38.5%		2.68	19.5%

Interestingly at first while watching the neural networks running and looking at the epochs I saw the accuracy increasing and the loss dropping. Unfortunately, this worked so well for the train data but I believe it was overtraining because as we can see from the results in the table we can see the accuracy percentage for the validation data and the test data are very low. In Figure 18, with the blue lines on the right, we see very good accuracy and loss results for the train data and then bad results with the orange line for the validation data. So why has transfer learned on the baseline done so badly?

Baseline 2 CNN



Baseline 2 + ImageNet weights CNN

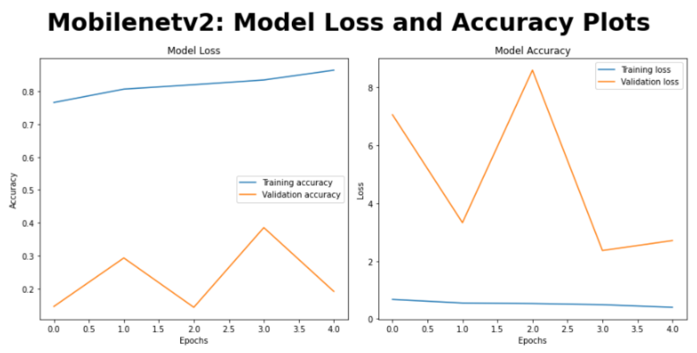


Figure 18

Potential reasons:

1) **Domain Difference:** I chose the ImageNet dataset which contains a wide variety of images from different domains. Since my images of the colon may be very different from any of these domains these pre-trained weights might not have helped.
2) **Overfitting:** Using the ImageNet weights can result in much more complex models since I only have 8000 images in total and this increases the chances of overfitting in my models.
3) **Fine Tuning Strategy:** I loaded the weights directly in and did not fine-tune and this can lead to issues.
4) **Low Epochs:** Due to time and computer resources I only used 5 Epochs which is a very low amount. Increasing the number of Epochs could lead to improvements as the models learn. These low results might just be low points.
5) **Model Architecture:** The model architecture itself could have an effect and I used only a baseline model.

While Transfer learning is a very interesting and powerful technique that sounds promising it would need a lot more work to pick the right pre-trained weights in order to leverage them with our own image classification.

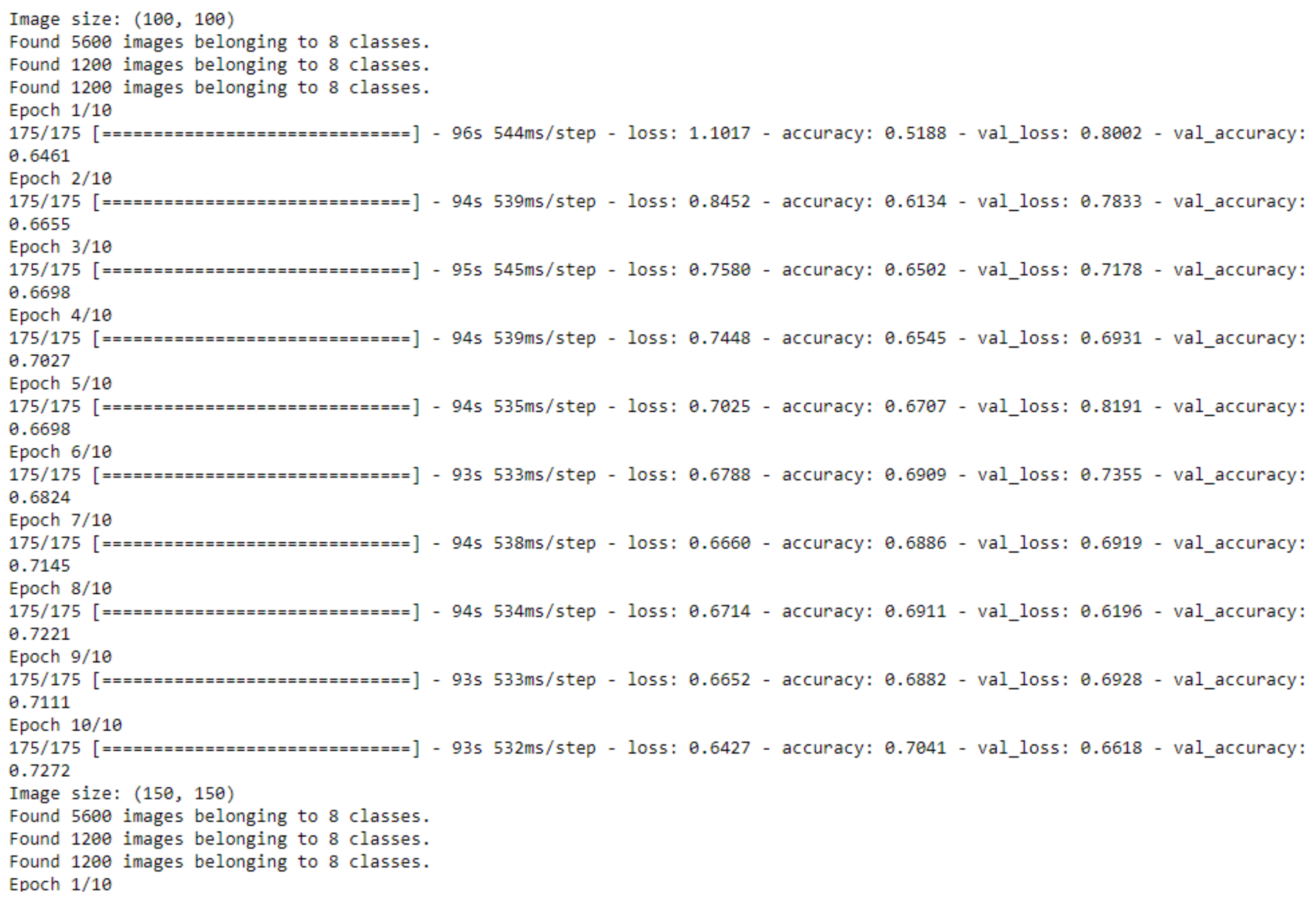
Effects of resizing the image

" Effect of resizing the image" with NN is the impact of changing the dimensions of input images and the effect it can have on the performance and characteristics of a NN. Resizing images is a common pre-processing step in computer vision tasks.

Things to consider:

1) **Computational Complexity**: Making images smaller can reduce the need for computer resources such as CPU and Memory. This can lead to faster training times and lower computer hardware resource usage. But it can also result in lower-resolution images which can reduce a model's performance.
2) **Aspect Ratio:** If you resize it's important to avoid distortion. Distorted images can reduce a model's performance.
3) **Features Preservation:** If you reduce the size of an image you can lose important features which can affect a model's performance. If you increase the size of an image, this could improve a model's performance but this would also increase computational complexity.
4) **Model Architecture:** You need to match architectures if you use pre-trained models such as VGG or ResNet.

We will look at various image sizes: 100x100, 150x150, 200x200, 250x250, and 300x300 and ran these with the baseline_2 model for 10 epochs. This took 6 hours and you can see an example of how the code was running below in Figure 19. The best image size was 100x100 and give a validation accuracy of 72.7%.



```
Image size: (100, 100)
Found 5600 images belonging to 8 classes.
Found 1200 images belonging to 8 classes.
Found 1200 images belonging to 8 classes.
Epoch 1/10
175/175 [==============================] - 96s 544ms/step - loss: 1.1017 - accuracy: 0.5188 - val_loss: 0.8002 - val_accuracy:
0.6461
Epoch 2/10
175/175 [==============================] - 94s 539ms/step - loss: 0.8452 - accuracy: 0.6134 - val_loss: 0.7833 - val_accuracy:
0.6655
Epoch 3/10
175/175 [==============================] - 95s 545ms/step - loss: 0.7580 - accuracy: 0.6502 - val_loss: 0.7178 - val_accuracy:
0.6698
Epoch 4/10
175/175 [==============================] - 94s 539ms/step - loss: 0.7448 - accuracy: 0.6545 - val_loss: 0.6931 - val_accuracy:
0.7027
Epoch 5/10
175/175 [==============================] - 94s 535ms/step - loss: 0.7025 - accuracy: 0.6707 - val_loss: 0.8191 - val_accuracy:
0.6698
Epoch 6/10
175/175 [==============================] - 93s 533ms/step - loss: 0.6788 - accuracy: 0.6909 - val_loss: 0.7355 - val_accuracy:
0.6824
Epoch 7/10
175/175 [==============================] - 94s 538ms/step - loss: 0.6660 - accuracy: 0.6886 - val_loss: 0.6919 - val_accuracy:
0.7145
Epoch 8/10
175/175 [==============================] - 94s 534ms/step - loss: 0.6714 - accuracy: 0.6911 - val_loss: 0.6196 - val_accuracy:
0.7221
Epoch 9/10
175/175 [==============================] - 93s 533ms/step - loss: 0.6652 - accuracy: 0.6882 - val_loss: 0.6928 - val_accuracy:
0.7111
Epoch 10/10
175/175 [==============================] - 93s 532ms/step - loss: 0.6427 - accuracy: 0.7041 - val_loss: 0.6618 - val_accuracy:
0.7272
Image size: (150, 150)
Found 5600 images belonging to 8 classes.
Found 1200 images belonging to 8 classes.
Found 1200 images belonging to 8 classes.
Epoch 1/10
```

Figure 19

Mixup

The research paper also looked at other techniques, two I looked into were Mixup. This is a data augmentation method that can be used with Neural Networks to improve a model's performance. I did try to use Mixup but was not able to get the code to work (see Home Computer Juypter Notebook) so here I give a description and a describe with some Pseudocode on how I would use these two techniques with more time.

Mixup

Mixup is a method of data augmentation that creates new training samples by interpolating random pairs of input samples with their labels. If you have two samples (x1, y1) and (x2, y2) and you have a mixing parameter "λ". Mixup generates a new training sample with:

- Mixed Input = $x\_mix = \lambda * x1 + (1- \lambda) * x2$
- Mixed Input = $y\_mix = \lambda * y1 + (1- \lambda) * y2$

Mixup helps the model to behave with a more linearly with in-between training samples which can lead to better generalization. Below is pseudocode on how I would use mixup with my code (this was the basis of writing my code that I could not get to work)

```
1) Load data ( train, validation, test images)
2) Create baseline 2 model
    a. Start Sequential model
    b. Add Conv2D, Maxpooling, Flatten, Dropout, Dense layers
    c. Compile model
3) Create a mixup data function (images, labels, alpha value)
    a. Generate random mixing value
    b. Randomly pair images and labels
    c. Create new mixing images and labels
    d. Return mixed images and labels.
4) Train the model with a mix
5) Evaluate the model with test data
```

6) As this dataset is not very large it is important to analyse the variability in the results. Explain?

8000 images is not a large dataset for our project, the training dataset is 5600 images, 1200 for the validation and 12000 for the test dataset. So analysing the variability of the dataset is crucial to tune a model's performance to make sure the models perform well. There are different techniques you can use to help.

1) **Train multiple models:** You can train different NN models with different parameters to find the best model by comparing the performance of these different models you can get an idea of the variability within each model.
2) **Take the average:** You can also run different models and take the average of their performance to reduce the variability. This is known as ensemble learning which we have seen in our Applied Machine Learning model. There are several different ways to take the averages with: Arithmetic mean, weighted average and Soft voting for example.
3) **K Fold Cross-Validation**: Using K (equally sized folds) folds to train the model with different combinations of the folds which you can then use to take the average metric. Similar to number two which uses ensemble learning.
4) **Bootstrap Resampling**: Create multiple datasets using resampling and training these to analyse the variability in performance.

Question 2 Final Words

At first, I had the misinformed opinion that if we have a problem we should just throw a Neural Network at it and this would get the best results. But given the challenges mentioned and the lack of a supercomputer to run my models, I can see now why an organisation might not want to have the complexity of a NN or use the resources it would take. The most epochs I ever ran (and did not record as Colab timed out) was 30 epochs and the research paper used 200 for their models! You really have to think through your plans as you can waste a lot of time rerunning models if you made a mistake or you want to try different options. Nothing is worse than seeing that you got a value but labelled it "test accuracy" instead of " test loss" and knowing you will have to run the model for many hours to change that mistake.

Although saying that, the highest validation results I had was 75% and this was with just 20 epochs so I can see how powerful CNNs can be. The ideas in section 5 are also very interesting, especially Transfer Learning, and while this did not do well for my project and gave a test accuracy of 19% so it massively overfitting, I can see its potential.

Question 3

General

Question 3 uses the “Kvasir A Multi-Class Image Dataset for Computer Aided Gastrointestinal Disease Detection” by Pogorelov et al paper but instead of the original dataset that is 2.3Gb of images, this dataset contains the feature files of the images and this manages to reduce the size of the original dataset to 9.3MB.

For question 3 we will be using R for our project on the research paper and run our own Random Forest models to see how they compare to the Models in Question 2 that were run on the original images with Convolutional Neural Network with Python. Given that this is just feature files that contain 6 global features, it is unlikely that these models will perform as well as models on the original data. But we will see.

1) Explain how these feature dataset is constructed and why in this manner.

This time for each of the images we have a feature file which contains 6 global features; JCD, Tamura, ColorLayout, EdgeHistogram, AutoColorCorrelogram and PHOG where each feature has a different amount of numeric values. As long as each feature file contains the same order of features and the same amount of values then we can use this to create a single vector for each of the images. This was tested in my code and all the features have the right order of global features and the right amount of elements. This was important to check, because if a feature file had a different order or number of values they could no longer be related.

For example for the first feature file in Dyed-Lifted-Polyps in Table 1. We can see that we have 6 Global Features, which all have different amounts of values and whole or float numbers. Notably, Tamura has values up to 6 decimal points and PHOG has 630 elements. When you add up the number of elements for each you are missing one numeric value to give 1185 elements.

Global Feature	**Number of values**	**Example (First few values)**
JCD	168	3.0 3.5 6.0 0.0 0.0….
Tamura	18	3.608456 6.681779……..
ColorLayout	33	11 25 9 4 16…….
EdgeHistogram	80	0 0 4 0 1 1 4…….
AutoColorCorreogram	256	13 13 12 12 13 13…….
PHOG	630	12 10 9 10 10 10……
Total	**1185**	**(missing one element of 1186)**

Table 1: A feature file layout example

Interpretation of the 1186 single vector

The research paper talks about a single vector of 1186. The interpretation of this had me thinking a lot. We can see that the 1185 elements are found in the example above. There is an extra element needed. This was either a unique value or a value to represent each image class for each of the 8 folders/classes. This was the decision I went with. A unique number for each of the 8000 feature files does not add any additional information whereas adding a value for the 8 classes is useful to add more information to the data and can also be used as a target variable later for the Random Forest models.

After I created the 1886 single vector and went to train the models I created a data frame of 1186 elements and used the 1st column (this is classes) as the target variable for the Random Forest models. As discussed in the lab I found this confusing as I would have thought that a single vector meant one item.

2) Construct a single 1186 feature vector

I read in each feature file to a character object, I then used a for loop to separate out each of the global features and create a list of lists. Each list was then a global feature that had their own elements and I was able to create Table 1 and explore how many elements each global feature had with print() and Length(), etc.

Now, that I had a list of lists it was now very easy to use the unlist() function to unlist the 6 lists and create a single vector with 1185 elements. I performed a length() check to check that there were 1185 elements in this vector.

As mentioned already I would then add an extra element to the beginning for each of the 8 classes based on the folders. This is added later when we create the dataset as at this point I had not decided to use an index value of 1 to 8000 or a class value of 1 to 8. But this is easily added in later.

3) Construct your own 1186 feature vector dataset

The next step now is to create a data frame with all 8000 feature files as the rows and the columns as each value in the 1186 vector. To do this I just created a list of 8000 feature files with 1185 elements. I perform some checks to make sure the Global Feature names are in the right order and that each Global feature has the right number of elements.

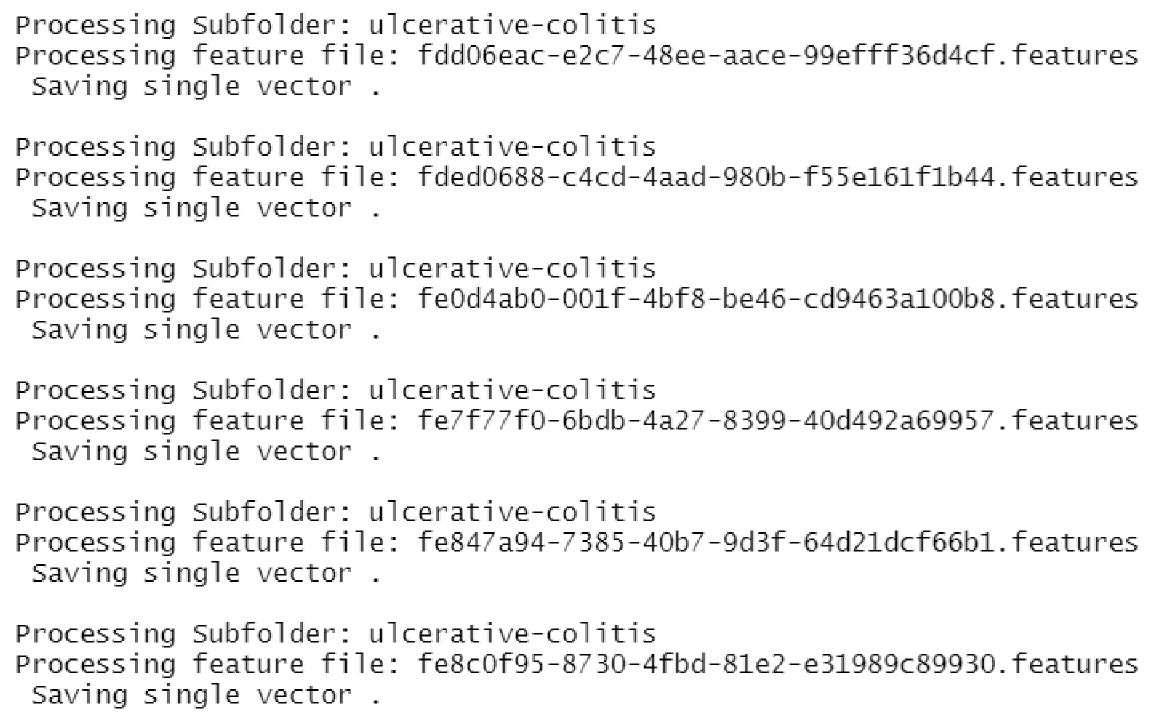
NOTE: This check is important. We are able to use the 1185 elements as a single vector to represent each feature file as long as the order and number of Global Features are the same for each file.

I then add the values from 1 to 8 for each of the classes from the folders and then separate the list out into a data frame. The reason for using lists and separating them out into a data frame at this late stage is that lists are faster to work with.

For loop to iterate through each feature file, extract the global feature and perform checks and balances

First, I took the global features' names and the number of elements in each of the global features; 168, 18, 33, 80, 256, 630. Then I created a for loop that would check that 1) each feature file had the right order of global features and 2) it had the right amount of values we saw above. I then created a new list with a single vector of 1185 elements for each of the feature files. This is based on the same code I used in part 2.

Figure 2 shows an output as the for loop iterated through each folder and feature file. This code ran very fast as I was using lists. I attempted using data frames at this point but it was much slower. It was much easier and faster to use lists at this point.



```
Processing Subfolder: ulcerative-colitis
Processing feature file: fdd06eac-e2c7-48ee-aace-99efff36d4cf.features
 Saving single vector .

Processing Subfolder: ulcerative-colitis
Processing feature file: fded0688-c4cd-4aad-980b-f55e161f1b44.features
 Saving single vector .

Processing Subfolder: ulcerative-colitis
Processing feature file: fe0d4ab0-001f-4bf8-be46-cd9463a100b8.features
 Saving single vector .

Processing Subfolder: ulcerative-colitis
Processing feature file: fe7f77f0-6bdb-4a27-8399-40d492a69957.features
 Saving single vector .

Processing Subfolder: ulcerative-colitis
Processing feature file: fe847a94-7385-40b7-9d3f-64d21dcf66b1.features
 Saving single vector .

Processing Subfolder: ulcerative-colitis
Processing feature file: fe8c0f95-8730-4fbd-81e2-e31989c89930.features
 Saving single vector .
```

Figure 2

Adding the extra element to go from 1185 to 1186 and creating a data frame.

I then added a class value for each folder from 1 to 8 to add an element to the beginning of the list to make up the 1186 vector. After this, I turned the list into a data frame of 1186 columns. For the Random Forest models, we need a target variable and we were able to use the first column then as a target and the rest of the 1185 columns as features to train the Random Forest models on with Caret.

Alternative: Previously I mentioned we could have used an index value from 1 to 8000 and then another column with the class value of 1 to 8. But this would have made 1187 columns. I wanted to try and stick to the 1186 number as much as possible since the research paper did not give enough details. Also, I did not believe adding an index from 1 to 8000 would have added anything to the models.

We now have a data frame with 8000 rows and 1186 columns which may take a very long time to train with Random Forest models on my computer.

4) Reproduce the "6 GF Random Forest"

Looking to reproduce the "6 GF Random Forest" we will use the full 1186 elements we have seen above. 1185 elements come from the combination of the 6 global features we have seen above with the first column acting as the target variable which is a value from 1 to 8 for each of the different classes (folders).

The research paper used a 50:50 split with a CV value of 2 for a further 50:50 split so as a first step I have tried to replicate these values. The research paper did not offer any other values they used to tune the Random Forest so it was hard to reproduce these results. As a starting point, I also used "createDataPartition" to make sure that each 50:50 split had an equal split of the classes to make sure I did not have an imbalance in the classes, again the paper did not give enough information on their approach taken.

In the table, you can the top result where the research paper had an accuracy value of 93.3% which is a very good result. Running several models and looking at different numbers of ntrees from 100 to 500 and mtry from 2 to 1185 I was only able to get a max accuracy of 77.33%. For each increase in the number of ntree, you can see that the accuracy only increases a very small amount, less the 0.25% for an increase from 100 to 500 ntrees. The research paper does not offer enough details of the values for hyperparameters used. We will look to improve this with our own models next.

	Ntree	**mtry**	**Accuracy**	**PREC**	**REC**	**SPEC**	**F1**
Research Paper Results	N/a	N/a	93.3%	0.732	0.732	0.958	0.727
Research Model 1	100	48	77.08%	0.77	0.771	0.967	0.769
Research Model 2	250	48	77.17%	0.771	0.772	0.967	0.77
Research Model 3	500	48	77.33%	0.772	0.773	0.968	0.772

Table 2

5) Try to improve your results

Instead of using a 50:50 split with tuning parameters, we did not know, I wanted to try a 70:15:15 split that we have seen in the second part of the project when we used the full images. We will also use a more common CV value of 5 and 10 vs the 2 that the research paper used. In the below table, you can see a general improvement of the models with 500 ntrees giving the best Test Accuracy of 79.33% for Model 3 which used a ntree value of 250 for mtry 48 with 10-fold cross-validation. This is an improvement on the 77.33% we were able to get above but did not come close to the research paper's accuracy.

	Ntree	**mtry**	**CV**	**Test Accuracy**	**PREC**	**REC**	**SPEC**	**F1**
RF Model 1	10	1185	5	73.33%	0.736	0.738	0.963	0.737
RF Model 2	100	48	10	77.92%	0.774	0.775	0.968	0.773
RF Model 3	250	48	10	79.33%	0.791	0.793	0.97	0.792
RF Model 4	500	48	10	78.5%	0.785	0.786	0.969	0.784
Grid Search Model	250	10	10	76.25%	0.761	0.762	0.966	0.759

Table 3

Looking at the metric results in more detail:

Test Accuracy:

The accuracy is the overall score on the Test data and we can see again, that model 3 had the highest results. What is interesting is the RF model with 500 ntrees did not perform as well. I was not able to run this often as it took a long time. I would liked to have run again for more trees which expanded the parameters in the Grid Search. Still, almost 80% is not bad but does not match the results from the research paper.

Precision and Recall:

Precision is a measure of the proportion of correctly predicted positive instances out of all instances of correctly predicted positive results. Once again model 3 performed the best. Recall measures the proportion of the correctly predicated positive instances out of all actual positive instances. Model 3 has once again performed the best.

Specificity:

Specificity measures the proportion of correctly measured negative results. The results are even better here with values in the high 90s. Once again model 3 performs the best with a value of 97%.

F1:

F1 score combines precision and recall into one metric which offers a balance between the two. Model 3 yet again performs the best.

Model Overview

As you can see in Table 3, I ran many combinations of models as well as attempting to use grid search in an attempt to find the best accuracy results. Some of these models took many hours to run with the grid search models running overnight. I would have liked to have ran higher ntrees and more combination with grid search if I had more time and computer resources. This is an issue with the research, I am trying to guess their models. So the results are hard to reproduce. Without more information, you cannot reproduce their findings and this is a critical issue with research papers and a common complaint as well.

Question 3 Final words

I was able to reproduce their 1186 vector but because they do not provide enough detail in its creation I cannot be sure it has been created in the same way.

Model three has performed the best in my testing. I was not able to reach the 93.3% of the research paper. An issue here is something that is common in quite a lot of research papers. While they have provided some code, they have not provided enough details to be able to reproduce their results. This is a flaw in their research as any results should be easily replicated by any other.

Project 3 Conclusion

A very interesting project to explore CNN's. I was impressed with how well a simple CNN can perform and up to this point I was definitely under the impression it would be easier to use NN for all problems. But given the increased use of resources, I now see why a manager in a company might prefer the quicker and less expensive results from other Statistical Models or Machine Learning Models.

This was the biggest project we ran so far with the largest amount of data and I now have a better understanding that you need to plan more before running models. Nothing annoyed me more than running a model overnight and noticing a small spelling mistake.

I was not able to reproduce the accuracy results from the research papers models and this is also a good lesson that I need to include details of the models and hyperparameters I used or at least include full notebooks so that my results can be reproduced.