Implementation & Testing – Model Development Phase 1

## 1.1

As laid out in the previous chapter, the initial iteration of the model-generating program has been designed with the idea in-mind of laying a solid foundation, upon which the project can expand and evolve. Version 1.1 can be found in its entirety by following this [hyperlink](https://github.com/yungroms/y3_proj/blob/5d6a392a09a007653d99be641cfc80633c73da6e/1.1%20(3.1)/1.1.py).

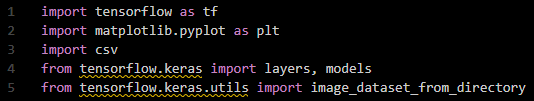


Figure Imports 1.1

Following the four-component structure proposed in the Design chapter, the program’s operation begins with the importation of the essential libraries: *TensorFlow*, *MatPlotLib*, and *CS*, as well as the specification of some packages from the *Keras* API.

A screen shot of a computer code

Description automatically generated

Figure Dataset Split 1.1

Next, some important data processing values are established such as the dataset directory, image size and batch size, before the dataset is split into training and validation sets, which are then normalised.

A screen shot of a computer program

Description automatically generated

Figure Model Architecture 1.1

The model’s architecture is then defined, compiled, and trained, and finally the training and validation results are visualised and stored for later analysis.

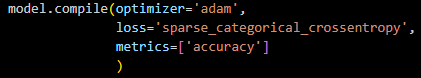


Figure Model Compile 1.1

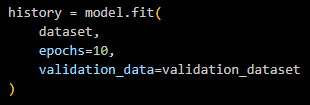


Figure Model Fit 1.1

The dataset utilised by this edition of the program, titled *LeafSnap\_15\_Lab*, can be viewed [here](https://www.kaggle.com/datasets/yungroms/leafsnap-15-lab).

## 1.1 Results

This purpose of this test is simply to verify that the initial program iteration is functioning as expected. Although the hyperparameter values are not of great concern during this test, it is worth noting for the sake of the line graphs below that the batch size was 32, while the number of epochs was 10.

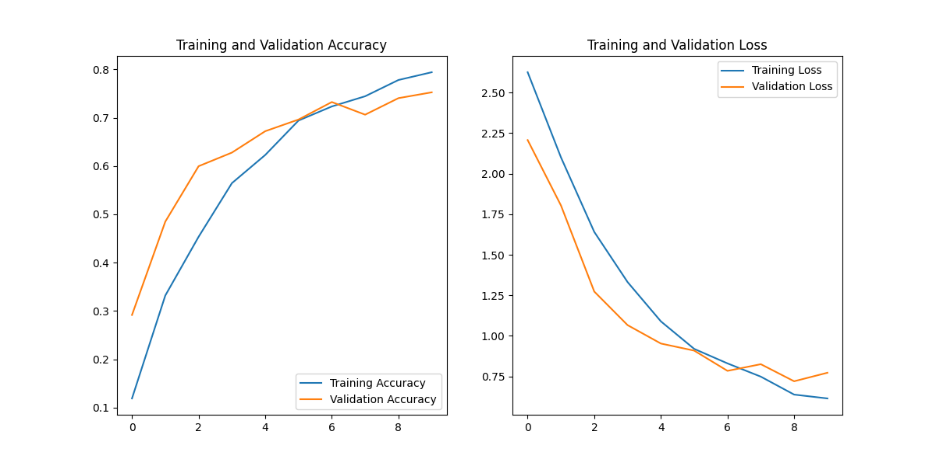


Figure Accuracy & Loss 1.1

Based upon the plots above, not only does it appear as though the code is functioning desirably, but also that the hyperparameter values utilised are promising starting points. Although there is a little turbulence in the validation plots, both the graphs show promising results, as accuracies increase, and losses decrease as the number of epochs increases. Furthermore, it is certainly true for the training results that the plots do not plateau before the test is over. This is indicative of the fact that the number of epochs should be increased, as doing so could improve the accuracy and loss results for both the training and validation sets.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Epoch | train\_accuracy | train\_loss | val\_accuracy | val\_loss |
| 1 | 0.11903566122055054 | 2.6257476806640625 | 0.2917504906654358 | 2.207411766052246 |
| 2 | 0.331993967294693 | 2.1005945205688477 | 0.48490944504737854 | 1.8041470050811768 |
| 3 | 0.4535409212112427 | 1.6409533023834229 | 0.5995975732803345 | 1.273249864578247 |
| 4 | 0.564540445804596 | 1.3317636251449585 | 0.6277666091918945 | 1.0663527250289917 |
| 5 | 0.6228026151657104 | 1.0895177125930786 | 0.6720321774482727 | 0.9528313875198364 |
| 6 | 0.6941235661506653 | 0.9198430180549622 | 0.696177065372467 | 0.9090974926948547 |
| 7 | 0.7232546210289001 | 0.8303197026252747 | 0.7323943376541138 | 0.7843056321144104 |
| 8 | 0.7443495988845825 | 0.748655378818512 | 0.7062374353408813 | 0.8254624605178833 |
| 9 | 0.7780010104179382 | 0.6380361318588257 | 0.7404426336288452 | 0.7201429605484009 |
| 10 | 0.7940733432769775 | 0.614301860332489 | 0.7525150775909424 | 0.726622223854065 |

This hypothesis is supported by the tabular data collected during the test. Both the training and validation accuracies are continuing to increase, while the training loss is also continuing to decrease. In the proceeding tests, it would be useful to manipulate the number of epochs and study its impact on these key metrics.

1.2

After a moment of contemplation following the first test, one potential improvement to the code was identified to improve record-keeping. Version 1.2 features the inclusion of the hyperparameters within the names of the output files, allowing for the improved organisation and identification within the storage files. This initial feature version is implemented into each of the output-generating functions separately, although in a later version it may be worthwhile to establish the naming convention as a global variable. Below is the hyperparameter-containing naming process of the plotting function. Version 1.2 can be found in its entirety [here](https://github.com/yungroms/y3_proj/tree/89b4a412c3cbd6ffe5f6ce891884f0032992de72/1.2%20(3.2)/1.2_results).

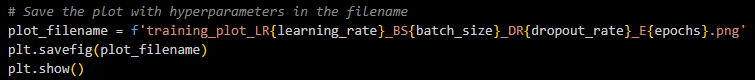


Figure Naming Convention 1.2

As stated at the conclusion of version 1.1, this next test round would specifically explore how varying the number of epochs impacts the accuracy and loss results. To gain an understanding of this impact, three broadly separated values will be used: 10, 25, and 50. Version 1.1’s test results displayed promising results when the number of epochs equalled 10, but also suggested that better results might be obtained with more epochs. To ensure that the focus of this test round is the number of epochs, all other hyperparameters will be kept consistent.

## 1.2a Results

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| No\_of\_epochs | train\_accuracy | train\_loss | val\_accuracy | val\_loss |
| 10 | 0.794575572013855 | 0.5587978363037109 | 0.7605633735656738 | 0.7212235927581787 |
| 25 | 0.9487694501876831 | 0.13891300559043884 | 0.7887324094772339 | 0.950129508972168 |
| 50 | 0.9854344725608826 | 0.04084133729338646 | 0.7444667816162109 | 1.6518267393112183 |

Based on the table above, which displays the final training and validation results from this test round’s three runs, it might appear as though increasing the number of epochs has a positive impact on the model’s accuracy. This is suggested by the fact that the training accuracy and loss results show improvement as the number of epochs progresses from 10 to 25, and again from 25 to 50. However, this is not supported by the validation accuracy and loss results, which actually worsen as the number of epochs increases. Although there is improvement in the validation accuracy between 10 and 25 epochs, the run with 50 epochs has the worst accuracy score of any run in the table, and the validation loss scores simply demonstrate a worsening performance as the number of epochs increases. While the data within this table may appear perplexing at first, there is a simple enough explanation: somewhere before the 25th epoch, overfitting starts occurring, and that by the 50th epoch, severe overfitting is certainly taking place. This analysis is supported by the plots of each of the test round’s results.



Figure Accuracy & Loss 1.2a 1

In fact, after inspecting the graph plots, it appears as though overfitting may start taking place before even the 10th epoch, as the training and validation accuracy and loss plots intersect around the 7th epoch. The plot of the test run with 50 epochs below clearly demonstrates more overfitting and worsening performance as the number of epochs increases. It could be assumed from this round of testing that the optimal number of epochs could be as low as 10, however only the number of epochs has been tested so far. Once additional hyperparameters are tested, the results may suggest a more optimal number of learning rates.



Figure Accuracy & Loss 1.2a 2

## 1.2b Results

Using the same code as above, this test round takes the experimentation a step further as it explores the performance impact of adjusting the learning rate, dropout rate and number of epochs in various combinations. Two values were tested for the learning rate (0.001, 0.005), three for the dropout rate (0.2, 0.35, 0.5), and three for number of epochs (10, 15, 20), providing insightful results. Below is a compiled table of the hyperparameter values and performance metrics of each run within this test, ranked from best to worst based primarily on validation accuracy.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Learning\_rate | batch\_size | dropout\_rate | epoch | train\_accuracy | train\_loss | val\_accuracy | val\_loss |
| 0.001 | 32 | 0.2 | 20.0 | 0.93 | 0.2 | 0.77 | 0.81 |
| 0.001 | 32 | 0.35 | 20.0 | 0.93 | 0.21 | 0.75 | 0.84 |
| 0.001 | 32 | 0.5 | 20.0 | 0.93 | 0.2 | 0.76 | 0.94 |
| 0.005 | 32 | 0.35 | 15.0 | 0.92 | 0.25 | 0.77 | 0.77 |
| 0.005 | 32 | 0.2 | 20.0 | 0.9 | 0.25 | 0.76 | 0.86 |
| 0.005 | 32 | 0.5 | 20.0 | 0.9 | 0.27 | 0.72 | 1.01 |
| 0.005 | 32 | 0.2 | 15.0 | 0.89 | 0.3 | 0.73 | 1.0 |
| 0.001 | 32 | 0.35 | 15.0 | 0.88 | 0.35 | 0.74 | 0.84 |
| 0.005 | 32 | 0.5 | 15.0 | 0.88 | 0.34 | 0.75 | 0.83 |
| 0.001 | 32 | 0.5 | 15.0 | 0.86 | 0.39 | 0.74 | 0.78 |
| 0.001 | 32 | 0.2 | 15.0 | 0.84 | 0.43 | 0.72 | 0.8 |
| 0.005 | 32 | 0.5 | 10.0 | 0.83 | 0.49 | 0.76 | 0.64 |
| 0.001 | 32 | 0.2 | 10.0 | 0.83 | 0.5 | 0.76 | 0.74 |
| 0.005 | 32 | 0.2 | 10.0 | 0.8 | 0.57 | 0.74 | 0.72 |
| 0.001 | 32 | 0.5 | 10.0 | 0.79 | 0.56 | 0.76 | 0.72 |
| 0.001 | 32 | 0.35 | 10.0 | 0.79 | 0.62 | 0.73 | 0.79 |
| 0.005 | 32 | 0.35 | 20.0 | 0.88 | 0.3 | 0.73 | 0.91 |
| 0.005 | 32 | 0.35 | 10.0 | 0.77 | 0.65 | 0.71 | 0.88 |

After analysing these results, it is clear that some hyperparameter combinations are more optimal than others. The top 3 performances, whose results are extremely close together, actually share the same values for learning rate (0.001) and number of epochs (20), supporting the inference that these are the optimal hyperparameter values for this current code iteration, with the dropout rate being the difference-maker. Dropout rates of 0.2 for 1st, 0.35 for 2nd, and 0.5 for 3rd all produced great results, and are practically inseparable, however it can be generalised that for this model architecture and dataset, a lower dropout rate leads to more optimal results. It can also be generalised that a lower learning rate and higher number of epochs produce superior results for this configuration. However, the top performers of this test run were still susceptible to overfitting, as can be seen in the accuracy and loss plots of the best performance (LR=0.001, BS=32, DR=0.20, E=20) below, suggesting that even better results could be obtained through further manipulation of the hyperparameters.

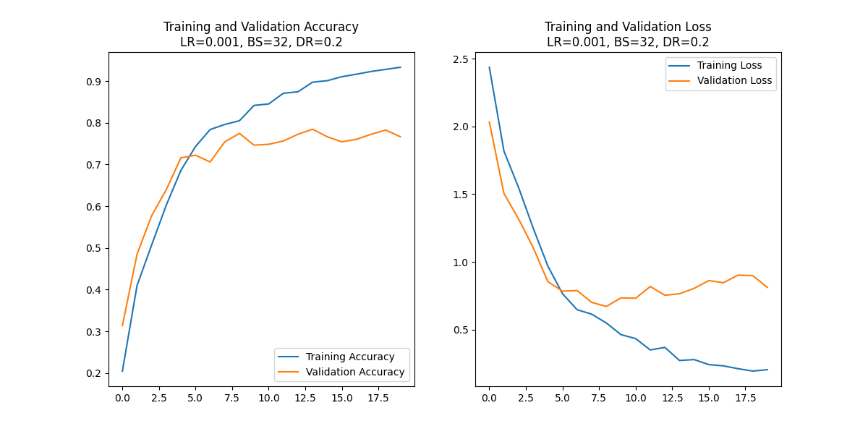


Figure Accuracy & Loss 1.2b 1

The test results of the worst run also offer some useful insight, supporting the above deductions that a smaller learning rate and higher number of epochs lead to preferrable performance metrics in this current code version. Even though its accuracy and loss plots below might appear decent, this combination of the highest learning rate (0.005), lowest number of epochs (10), and intermediate dropout rate (0.35) to produce the worst training accuracy, training loss, and validation accuracy, as well as the second worst validation loss.

A graph of a number of people

Description automatically generated with medium confidence

Figure Accuracy & Loss 1.2b 2

## 1.2c Results

For the final test round of code iteration 1.2, the dropout rate has been kept constant at 0.5, while the values of the other hyperparameters were adjusted in various combinations: The learning rate took values of 0.0005, 0.001, and 0.005; batch size took 32 and 64; and the number of epochs took 10 and 15.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| learning\_rate | batch\_size | dropout\_rate | no\_of\_epochs | tr\_accuracy | tr\_loss | val\_accuracy | val\_loss |
| 0.001 | 64 | 0.5 | 10 | 0.77 | 0.66 | 0.74 | 0.40 |
| 0.001 | 64 | 0.5 | 15 | 0.86 | 0.41 | 0.70 | 0.88 |
| 0.0005 | 32 | 0.5 | 10 | 0.76 | 0.67 | 0.74 | 0.78 |
| 0.0005 | 64 | 0.5 | 10 | 0.78 | 0.65 | 0.73 | 0.77 |
| 0.005 | 64 | 0.5 | 10 | 0.78 | 0.63 | 0.74 | 0.71 |

While the results from this round of testing are all rather close, it is still possible to identify the top performers. Three tests tie for 1st when examining what is considered the most important metric, validation accuracy, as they obtained a score of 0.74, and so the other performance metrics must be used to determine which of these hyperparameter combinations produced the best test results overall. These three frontrunners also have very close scores for training accuracy and training loss, with there being respective differences of 0.01 and 0.03 between the best and worst. Fortunately, validation loss, which is possibly the second most important metric, clarifies which combination of hyperparameter values produced the best results. Based upon the results in its column, a clear winner can be spotted, achieving a score of 0.40, compared to 0.77 and 0.71 of the other frontrunners. The best test run was obtained using a learning rate of 0.001, a batch size of 64, and 10 epochs as its hyperparameters, and its plot can be seen below.

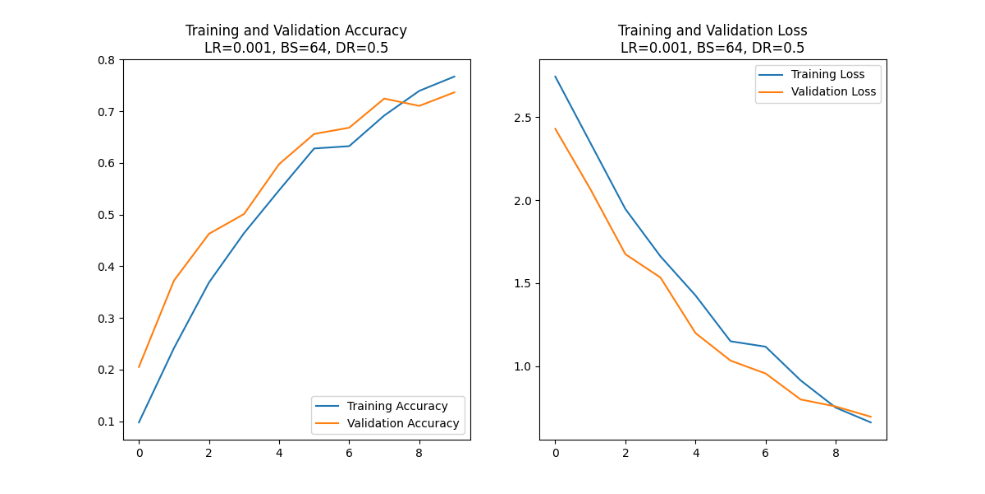


Figure Accuracy & Loss 1.2c

From this round of training, several insights have been obtained. Firstly, it can be deduced that further experimentation with batch size could lead to improved performance. This round was the first so-far to experiment with the higher batch size of 64, and every test run that utilised this value outperformed the only test with 32 as its batch size. It is interesting that even at this initial stage, with a simple code version and small dataset set, the larger batch size outperformed the test who’s only differing hyperparameter was the smaller batch size of 32. Later in the project, when larger, more complex codes and datasets are implemented, the benefit of a larger batch size should only become more pronounced, although it does come with the trade-off of being more computationally expensive.

Another deduction from this testing round is that there is a ‘sweet spot’ for the learning rate. The top three results from this round featured identical batch size, dropout rates and epochs, with the only difference being the learning rate value, which was 0.001, 0.005, and 0.0005 in order of performance from best to worst. It may be useful to experiment with a narrower range of learning rate values in a future test, to more accurately determine the aforementioned ‘sweet spot’.

Following some insightful initial tests in phase 1, specifically examining the performance impact of hyperparameter manipulation, the second phase will now begin. Hyperparameter experimentation will continue, however phase 2 will also introduce resultant variety by varying the input dataset, model architecture, and performance metrics.

# Implementation & Testing – Model Development Phase 2

## 2.1

In the first code version of phase 2, named [2.1](https://github.com/yungroms/y3_proj/blob/89b4a412c3cbd6ffe5f6ce891884f0032992de72/2.1%20(4.1)/2.1.py), three key modifications have been made in comparison to 1.2. Firstly, 2.1 introduces the functionality to further divide the input data into a third set for testing the model after training and validation have concluded. This data, which will be entirely new to the model, will provide greater insight into the model’s performance against unseen images and will generate more realistic accuracy and loss values.

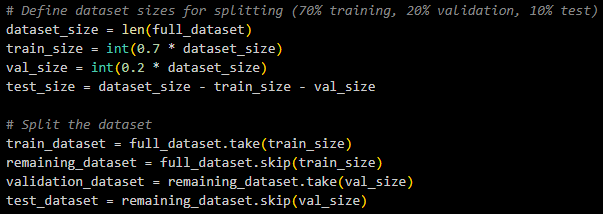


Figure Dataset Split 2.1

Another new introduction is the hard-coded, specifiable output directory. While this is not a feature that will impact performance, the project’s record-keeping ability will be improved, as this value can be updated with each new iteration to ensure that the results of the various tests can be directed to the appropriate storage destination. In order to implement this feature securely, an additional library called *OS* had to be imported, which could generate the specified output directory in the event that it hadn’t yet been created.



Figure Output Directory 2.1

The final new inclusion in 2.1 is the functionality to utilise the newly created test set to determine the model’s performance when faced with never-before-seen data. The testing is handled by the imported model.evaluate method, before the results are saved in a text file and printed to the console. As stated above, by gathering test accuracy and test loss data, a more realistic understanding of the model’s predictive capabilities can be obtained.

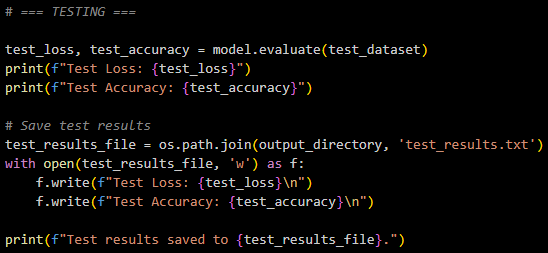


Figure Testing 2.1

## 2.1 Results

In this round of tests, 2.1, batch size and dropout rate have been kept constant at 32 and 0.5 respectively. This means that the focal hyperparameters are the learning rate and the number of epochs. The table below is a concentrated overview of the results gathered during round 2.1. Each row represents a different test, and the row’s columns contain its hyperparameters, the accuracy and loss scores of the training and validation sets from its final epoch, as well as the accuracy and loss scores from its test set.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| LR | BS | DR | E | train\_acc | train\_loss | val\_acc | val\_loss | test\_acc | test\_loss |
| 0.001 | 32 | 0.5 | 10 | 0.943 | 0.161 | 0.973 | 0.099 | 0.961 | 0.113 |
| 0.001 | 32 | 0.5 | 15 | 0.948 | 0.136 | 0.969 | 0.088 | 0.975 | 0.054 |
| 0.005 | 32 | 0.5 | 10 | 0.955 | 0.127 | 0.946 | 0.130 | 0.939 | 0.155 |
| 0.0005 | 32 | 0.5 | 10 | 0.931 | 0.2114 | 0.967 | 0.131 | 0.939 | 0.191 |
| 0.005 | 32 | 0.5 | 15 | 0.969 | 0.095 | 0.975 | 0.086 | 0.971 | 0.094 |
| 0.0005 | 32 | 0.5 | 15 | 0.957 | 0.119 | 0.983 | 0.0726 | 0.982 | 0.047 |
| 0.0005 | 32 | 0.5 | 20 | 0.977 | 0.070 | 0.962 | 0.108 | 0.950 | 0.108 |

From these results, it is clear that this was a very competitive testing round. None of the training, validation or test accuracies scored less than 0.9, and none of the losses scored greater than 0.25. Still, it is clear that some hyperparameter configurations outperformed the rest. The two most outstanding results actually ran for the same number of epochs, which was 15. The only separating factor is their learning rates: the top performer’s learning rate was 0.0005, while the close 2nd performer’s value was 0.001. Remarkably, the difference between their test accuracy scores was a mere 0.007, the same difference between their test loss scores. The configuration with the 0.0005 learning rate marginally outperforms its counterpart with a learning rate of 0.001, however both results are great. The victorious configuration’s plot can be seen below, while the 2.1’s complete results collection can be found here.

A graph of different colored lines

Description automatically generated with medium confidence

Figure 6 Accuracy & Loss 2.1

## 2.2

Now that highly accurate models have been developed, with almost certain accuracies and miniscule losses, it is time to integrate another performance metric, to verify that the results being generated are legitimate. The metric of choice is the confusion matrix, and it has been integrated into a new code version named [2.2](https://github.com/yungroms/y3_proj/blob/89b4a412c3cbd6ffe5f6ce891884f0032992de72/2.2%20(4.3)/2.2.py).

The integration of the confusion matrix is simple enough, however it does require the importation of some additional tools, this time from the *SciKit Learn* package: confusion\_matrix, and ConfusionMatrixDisplay. Additionally, the importation of NumPy is required within the matrix creation for its mathematical capabilities. Using these newly imported tools, the implementation and visualisation of the confusion matrix can be seen in the code snippet below.

A computer screen shot of text

Description automatically generated

Figure Confusion Matrix Implementation 2.2

Another implementation necessitated by the outstanding performance metrics is the ability to save the model, so that it may be used again, such as for testing, or integration into an app or website. This implementation is enabled by the model package’s model.save function.

A black screen with white text

Description automatically generated

Figure Save Model Implementation 2.2

The primary aim of the following tests is simply to verify the successful implementation of both new features covered above. However, the equally important secondary aim is to continue the manipulation of the hyperparameters, in an attempt to discover if any combinations can be discovered which might surpass those from round 2.1. The target hyperparameters will once again be the learning rate and number of epochs, as there is still much uncertainty regarding their ideal values.

## 2.2 results

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| learning\_rate | batch\_size | dropout | epochs | test\_loss | test\_accuracy |
| 0.0001 | 32 | 0.5 | 20 | 0.10644903033971786 | 0.9678571224212646 |
| 0.001 | 32 | 0.5 | 15 | 0.1099146381020546 | 0.9642857313156128 |
| 0.001 | 32 | 0.5 | 20 | 0.09333613514900208 | 0.9678571224212646 |
| 0.0005 | 32 | 0.5 | 20 | 0.08217372745275497 | 0.9678571224212646 |
| 0.005 | 32 | 0.5 | 15 | 0.0778030976653099 | 0.9714285731315613 |
| 0.005 | 32 | 0.5 | 20 | 0.104 | 0.9642857313156128 |

As was the case in the previous testing round, the results from 2.2 are similarly brilliant, as every test loss was below 0.12, and every accuracy above 0.96. On average, these results might actually surpass those obtained in 2.1. However, no individual test run manages to overtake the champion of the previous round. Here in 2.2, the hyperparameter combination used to generate the top result was 15 epochs, a learning rate of 0.005, a batch size of 32, and a dropout rate of 0.5. This combination generated a test accuracy of 97.14% as well as a test loss of 0.078, which are outstanding results, however they narrowly fail to surpass the top performer from 2.1. Interestingly, both of these results share matching values for batch size, dropout rate and the number of epochs, differing only in the learning rate, which continues to prove itself to be a vital component to a successful test. Below are the accuracy and loss plots for the test with the best results.

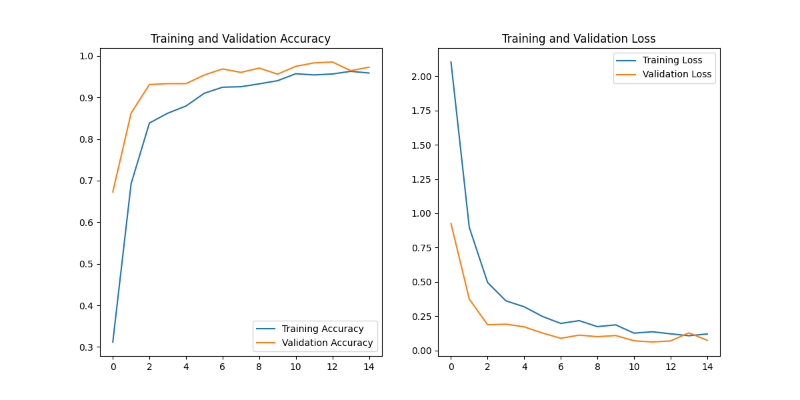


Figure Accuracy & Loss 2.2

Now let’s examine the confusion matrix below, which proves two things: firstly, it proves that the implementation of the confusion matrix within 2.2 was successful, as the matrix has been generated as intended. But more importantly, it also proves that 2.2’s top performer really is as accurate as the results above suggested. Almost every single prediction matched the true label, as indicated by the brightly-coloured diagonal line of values. There is a very minor array of incorrect predictions, which would explain the test accuracy’s missing 2-3%, but this confusion matrix certainly does support the resultant test accuracy of 97.1%.

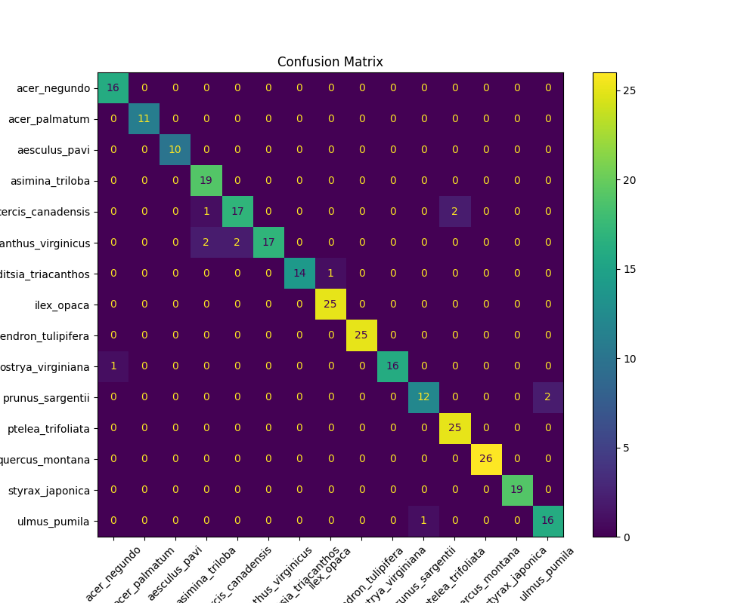


Figure Confusion Matrix 2.2

Now that near-ideal results have been obtained with this familiar and proven dataset and code combination, the next round of tests will aim to verify if these fantastic accuracies can be achieved on a more realistic dataset.

## 2.3

As mentioned at the end of 2.2, this next round of testing does not aim to improve the performance of the models. Instead, the aim of [2.3](https://github.com/yungroms/y3_proj/blob/89b4a412c3cbd6ffe5f6ce891884f0032992de72/2.3%20(4.4)/2.3.py) is to test the accuracy of these models when given realistic data to predict.

In previous sections, while the test set was genuinely new to the model, as the model had not seen any of the test images before, the data was very similar to the training and validation, having originally come from the same dataset. The desired aim of this project is to try and develop a model which can be used in real-world scenarios to classify species based on input imagery.

To test if the presently developed models are close to achieving this aim, a [new dataset](https://www.kaggle.com/datasets/yungroms/web-dataset) has been compiled manually, using images of the desired species gathered from various online sources. These images are representative of the real-world images users might try to classify, and so this test aims to determine how realistic the abilities of the current models actually are.

2.3’s operation starts with the specification of the output, model and test data directory paths:

A computer screen with text

Description automatically generated

Figure Path Specification 2.3

Next, the model is loaded using load\_model, before the test dataset is loaded using image\_dataset\_from\_directory:

A computer screen with white text

Description automatically generated

Figure Load Model 2.3

The loaded model is then tested using the model.predict function:

A computer screen with text

Description automatically generated

Figure Model Predict 2.3

Finally, the performance metrics are calculated and stored, followed by the visualisation and storage of confusion matrix:

A computer screen with text on it

Description automatically generated

Figure Performance Metrics 2.3

Every model generated during 2.2 will be tested, before being compiled into tabular form and ranked from best to worst.

## 2.3 Results

|  |  |  |
| --- | --- | --- |
| Model | Accuracy | Log\_Loss |
| model\_LR0.0001\_BS32\_DR0.5\_E20 | 0.1333 | 17.7812 |
| model\_LR0.0005\_BS32\_DR0.35\_E20 | 0.1333 | 19.0573 |
| model\_LR0.001\_BS32\_DR0.5\_E20 | 0.1200 | 17.9335 |
| model\_LR0.0005\_BS32\_DR0.5\_E20 | 0.1067 | 21.3504 |
| model\_LR0.005\_BS32\_DR0.5\_E20 | 0.0800 | 23.5620 |
| model\_LR0.005\_BS32\_DR0.5\_E15 | 0.0667 | 23.1634 |
| model\_LR0.001\_BS32\_DR0.5\_E15 | 0.0667 | 25.6092 |

Above is the table of results, populated with the accuracy and logarithmic loss of each tested model. As indicated by the poor results across the entire table, none of the models generated to this point performed at-all well when challenged with the realistic dataset. The highest accuracy obtained was a mere 13%, while this model’s accompanying log loss was almost 18.

The primary reason for this poor performance is down to the cleanliness of the input data. The original dataset *LeafSnap*, is comprised of well-taken, clear photographs, in which the contains nothing but the species sample. Therefore, the models to this point have only learned to classify and predict images such as these. The manually created dataset used within this experiment contained poor-quality, noisy images. Even though the species were identical to those within the original dataset, from the model’s perspective they were very unfamiliar. It seems as though this test was too great a challenge for the models developed to this point. If high performance is desired when input images are cluttered and noisy, then the model must be trained on similar input data. The search for a more robust and suitable dataset must now be undertaken, so that the models can more successfully handle realistic user images.

## 2.4

Before a new dataset is implemented necessarily, there is at least one remaining improvement worth attempting to improve the robustness of the current model and its generative code: data augmentation. By utilising this valuable set of tools, the original dataset can be made more challenging and diverse, as augmentations such as rotating, flipping and zooming present the model with more variety during the training process, and greatly extend the original dataset’s utility. Code version 2.4 can be found [here](https://github.com/yungroms/y3_proj/blob/89b4a412c3cbd6ffe5f6ce891884f0032992de72/2.4%20(4.5)/2.4.py).

The simplest way to implement data augmentation within the pre-existing code is to define a data augmentation layer, then include this layer at the start of the model definition. In this code version, 2.4, the three aforementioned augmentation techniques are applied.

A screen shot of a computer program

Description automatically generated

Figure Augmentation & Archiecture 2.4

The rest of the version 2.4’s code is identical to that used during the previous version, 2.2. This ensures that, aside from the varying hyperparameters, the differential feature is the newly implemented data augmentation layer.

## 2.4 Results

Below is the compiled and ranked results from 2.4’s test round.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| L.R | B.S | D.R | E | Tr. Acc. | Tr. Loss | Val. Acc. | Val. Loss | Test Acc. | Test Loss |
| 0.0005 | 32 | 0.4 | 30 | 0.7567935 | 0.6860955 | 0.8697917 | 0.4268404 | 0.9224138 | 0.3941087 |
| 0.0005 | 32 | 0.5 | 20 | 0.7858796 | 0.5902519 | 0.8916667 | 0.3401005 | 0.9000000 | 0.3303602 |
| 0.0001 | 32 | 0.5 | 20 | 0.7870370 | 0.5987527 | 0.8812500 | 0.3473099 | 0.8821428 | 0.3410837 |
| 0.0001 | 32 | 0.35 | 20 | 0.7361111 | 0.7478410 | 0.8854167 | 0.3317185 | 0.8571429 | 0.3362466 |
| 0.0050 | 32 | 0.5 | 15 | 0.7465278 | 0.7398294 | 0.8645833 | 0.3675282 | 0.8571429 | 0.3778828 |
| 0.0010 | 32 | 0.5 | 15 | 0.7430556 | 0.7220295 | 0.8354167 | 0.4161567 | 0.8321428 | 0.4124876 |

Based on this data, it appears as though the data augmentation has been implemented successfully. While the best accuracy and loss scores in the table above do not quite rival those from stages 2.1 or 2.2, the highest test accuracy of 92% is still a brilliant result, as is the second highest accuracy of 90%. This is especially true since the complexity and difficulty of the input data has been increased via the data augmentation layer.

However, only the test accuracies are comparable to the results of previous versions. The other columns, such as training accuracy and test loss, show room for much improvement, as their values are considerably poorer than the values of these columns in earlier test rounds.

That being said, no accuracy score within the table is below 70%, which is still respectable and noteworthy. If each test’s scores could be increased with further modification and enhancement while maintaining the augmentation layer, the resultant model would not only be highly accurate, but would be considerably more robust than its predecessors, and therefore should have greater success in realistic scenarios.

Below is the accuracy and loss plot of the best-performing model from version 2.4.

A graph of different colored lines

Description automatically generated with medium confidence

Figure Accuracy & Loss 2.4

Additionally, the same model’s confusion matrix can be seen below. To view the entire test results from 2.4, click [here](https://github.com/yungroms/y3_proj/tree/89b4a412c3cbd6ffe5f6ce891884f0032992de72/2.4%20(4.5)/2.4_results).

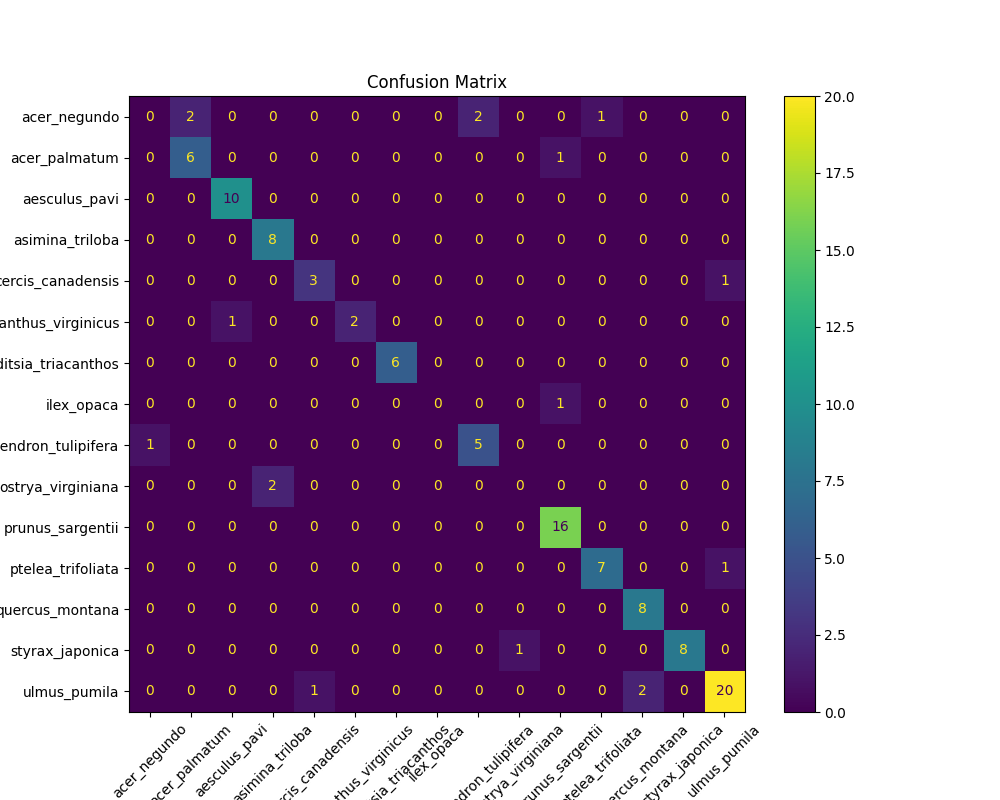


Figure Confusion Matrix 2.4

To conclude this second phase of implementation and testing, it must be stated that lots of encouraging progress has been made. While using the original dataset, extremely accurate results were obtained, while the inclusion of data augmentation enhanced the robustness and real-life capabilities of the models.

The next phase of development will focus on modifications to the model’s architecture. In particular, the integration of pre-trained models should generate interesting results and improved performance. Various pre-trained models were researched in previous areas of the project, and the understanding gained of their powerful capabilities supports the belief that they will be highly beneficial in achieving the aims of the project.

After some supplementary research into pre-trained deep-learning models, two architectural families sound particularly relevant for their efficiency and lightweight design, those being *EfficientNet* and *MobileNet*. Over the course of the next phase of implementation and analysis, both *EfficientNet* and *MobileNet* will be implemented and tested for their impact on prediction performance.

Implementation & Testing – Model Development Phase 3

## 3.1

The first of these pre-trained architectures to be implemented is *MobileNetV2*. This can be found within the keras.applications package, and must be imported prior to use.



Figure MobileNet Importation 3.1

The pre-trained architecture must then be instantiated, and can then be included within the model definition.

A computer screen with text on it

Description automatically generated

Figure Model Architecture 3.1

The initial execution of phase 3 will use the *LeafSnap\_15\_Lab* dataset utilised throughout every code version thus far. This is in order to distil the impact of the newly integrated pre-trained network, enabling easier comparison between this new code version and its predecessors. For this same reason, hyperparameter value manipulation will initially be kept minimal. 3.1 can be found in its entirety [here](https://github.com/yungroms/y3_proj/blob/89b4a412c3cbd6ffe5f6ce891884f0032992de72/3.1%20(5.1)/3.1.py).

## 3.1 Results

The table below contains the entire execution of version 3.1. The hyperparameters used within this version were an epoch number of 10, a learning rate of 0.0001, and a batch size of 32.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Epoch | train\_accuracy | train\_loss | val\_accuracy | val\_loss |
| 1 | 0.41956019401550293 | 1.9314926862716675 | 0.8812500238418579 | 0.6501874923706055 |
| 2 | 0.8043981194496155 | 0.6447209715843201 | 0.9645833373069763 | 0.29596927762031555 |
| 3 | 0.8958333134651184 | 0.3674798607826233 | 0.9833333492279053 | 0.18486452102661133 |
| 4 | 0.9241898059844971 | 0.2782028019428253 | 0.9916666746139526 | 0.13591483235359192 |
| 5 | 0.9461805820465088 | 0.20275968313217163 | 0.9916666746139526 | 0.10851286351680756 |
| 6 | 0.9479166865348816 | 0.1823740303516388 | 0.9916666746139526 | 0.09507021307945251 |
| 7 | 0.9681712985038757 | 0.13685214519500732 | 0.987500011920929 | 0.0925513207912445 |
| 8 | 0.9745370149612427 | 0.11641964316368103 | 0.9854166507720947 | 0.07577888667583466 |
| 9 | 0.9716435074806213 | 0.10396270453929901 | 0.9916666746139526 | 0.05783345550298691 |
| 10 | 0.9762731194496155 | 0.09534183889627457 | 0.9979166388511658 | 0.05179242044687271 |

As anticipated, by utilising the pre-trained *MobileNetV2* architecture, the accuracy and loss results have reached new highs. In the very first epoch, the validation accuracy reached 88%, and by the tenth and final epoch reached an almost perfect score of 99.79%. This outstanding performance is also found in the test data, where the accuracy reached 99.64%. Remarkably, the loss value of both the validation and test sets was 0.05, the lowest loss value recorded throughout the project so far.

|  |  |
| --- | --- |
| Test Loss | 0.053246717900037766 |
| Test Accuracy | 0.9964285492897034 |

This almost perfect accuracy result is also reflected in the confusion matrix, where every single prediction was true positive.

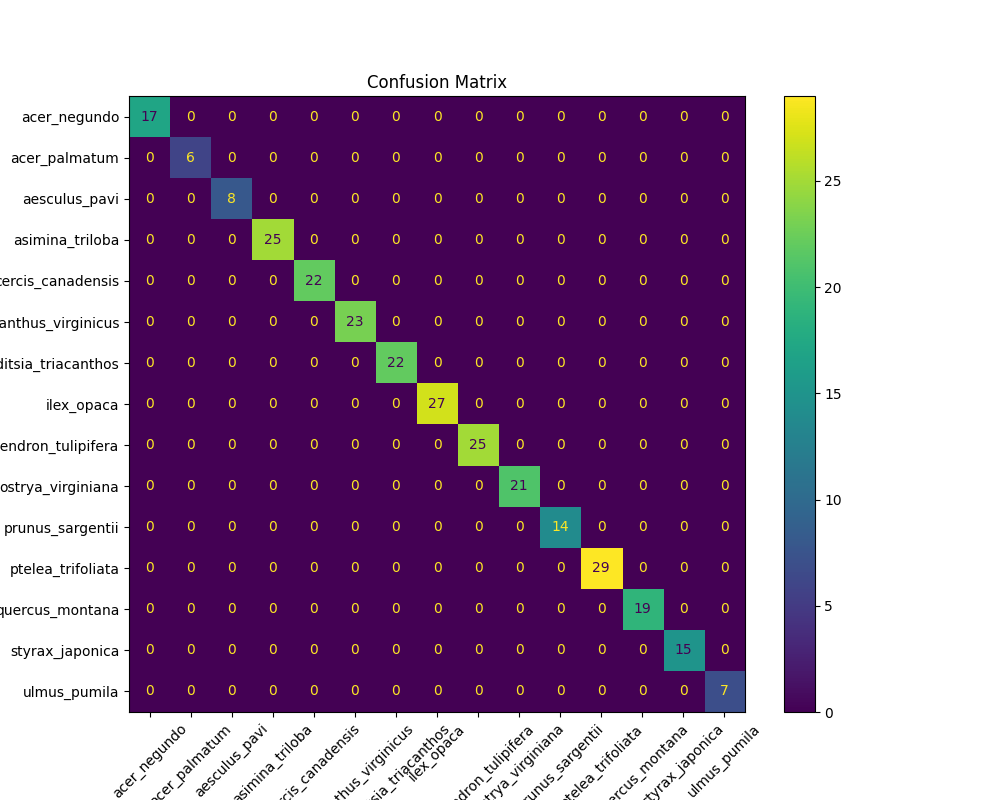


Figure Confusion Matrix 3.1

Finally, by examining the accuracy and loss plots, it can be seen that even though the accuracy was almost perfect, the loss gradients are yet to completely plateau by the end of the tenth epoch. This incomplete plateau suggests that with an increased number of epochs, even lower loss values may be obtained.

A graph of different colored lines

Description automatically generated with medium confidence

Figure Accuracy & Loss 3.1

In conclusion of this first test of phase 3, by implementing the *MobileNetV2* architecture and utilising its pre-trained network, project-wide records have been set in terms of both accuracy and loss. Even so, there may still be room for further improvement. Two methods which could be implemented to achieve further success are the aforementioned increase of the number of epochs, as well as the increase in size of the data input. That being said, the next phase of experimentation will return to the challenge of obtaining accurate predictions from an imperfect, realistic dataset. By utilizing pre-trained architectures, sufficient progress in this endeavour should be attainable.

## 3.2

As stated in the conclusion of section 3.1, this [next stage](https://github.com/yungroms/y3_proj/blob/89b4a412c3cbd6ffe5f6ce891884f0032992de72/3.2%20(5.2)/3.2.py) will attempt to address the poor performance of previous models when faced with a realistic dataset. Unfortunately, a suitable dataset of tree leaves could not be found, however a very promising mushroom dataset has been discovered, featuring thousands of realistic images of mushrooms taken in the field. Although the subject matter will of course have changed, the technological principles will remain the same.

This new dataset reignites a discussion from the project’s earlier stages, when the idea of classifying mushrooms was being heavily considered. Ultimately, tree leaves were chosen as the classification subject, as samples were easier to collect, thereby simplifying the process of manually creating a novel dataset. Now that the use of the novel dataset has been disregarded due to insufficient amounts of training data, the idea of re-focusing on the classification of mushrooms seems perfectly plausible. As mentioned above, nothing substantial must be altered to utilise this new, preferred dataset: the code can remain virtually unchanged. The model will now be trained to classify mushrooms, with the fine-tuning continuing as before to improve prediction performance.

This new dataset, initially assembled by the Mycology Society of Northern Europe, consists of nine classes varying in size from around 350 to 1350 images. Obviously, these classes are rather imbalanced, and so methods of balancing the dataset will have to be explored in an upcoming code version, however the realistic quality of the dataset makes it useful and preferred at this stage of the project. This dataset, named *shrooms\_ds*, can be found [here](https://www.kaggle.com/datasets/yungroms/shrooms-ds).

A black screen with white text

Description automatically generated

Figure dataset\_path 3.2

The only difference between code versions 3.1 and 3.2 is the implementation of this new dataset, which is done by changing the directory path of the *dataset\_path* variable, as seen above. All other parts of the code, including hyperparameter values, have been kept identical to the previous version, ensuring that the variance in performance can solely be attributed to the new dataset.

## 3.2 Results

Below are the training and validation results for test 3.2, which featured the new *shrooms\_ds* dataset, and the hyperparameter values of a learning rate of 0.0001, a batch size of 32, and an epoch number of 10. While not as brilliant as those of 3.1, these results are still promising, especially as a starting point following the implementation of the new, more challenging dataset. Both the training and validation accuracies display decent improvement between the first and last epochs, while the training and validation losses decrease simultaneously.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Epoch | train\_accuracy | train\_loss | val\_accuracy | val\_loss |
| 1 | 0.3728741407394409 | 1.8280714750289917 | 0.5892857313156128 | 1.2022218704223633 |
| 2 | 0.5599489808082581 | 1.2567644119262695 | 0.6614583134651184 | 1.0068273544311523 |
| 3 | 0.620323121547699 | 1.1051416397094727 | 0.680059552192688 | 0.9331252574920654 |
| 4 | 0.6417942047119141 | 1.0257865190505981 | 0.6822916865348816 | 0.9211941957473755 |
| 5 | 0.6626275777816772 | 0.9800623655319214 | 0.6979166865348816 | 0.8707048296928406 |
| 6 | 0.6764456033706665 | 0.9246760010719299 | 0.699404776096344 | 0.8720042109489441 |
| 7 | 0.6915391087532043 | 0.9069986939430237 | 0.703125 | 0.8379604816436768 |
| 8 | 0.6902636289596558 | 0.8786500692367554 | 0.710565447807312 | 0.8502814769744873 |
| 9 | 0.6898384094238281 | 0.8745827078819275 | 0.7209821343421936 | 0.8295154571533203 |
| 10 | 0.6972789168357849 | 0.8545891642570496 | 0.7157738208770752 | 0.8201534748077393 |

As can be seen in the small table below, containing the test loss and accuracy of 3.2, the model’s performance on these similar but new test images not only compares, but is in fact slightly better than its performance on the training and validation sets. This indicates that the model is doing well at feature extraction and generalisation.

|  |  |
| --- | --- |
| Test Loss | 0.8266926407814026 |
| Test Accuracy | 0.7203007340431213 |

Of course, adjustments will be implemented to further improve performance metrics, but these initial test results following the implementation of the new dataset offer a hopeful suggestion that excellent test scores can be obtained.

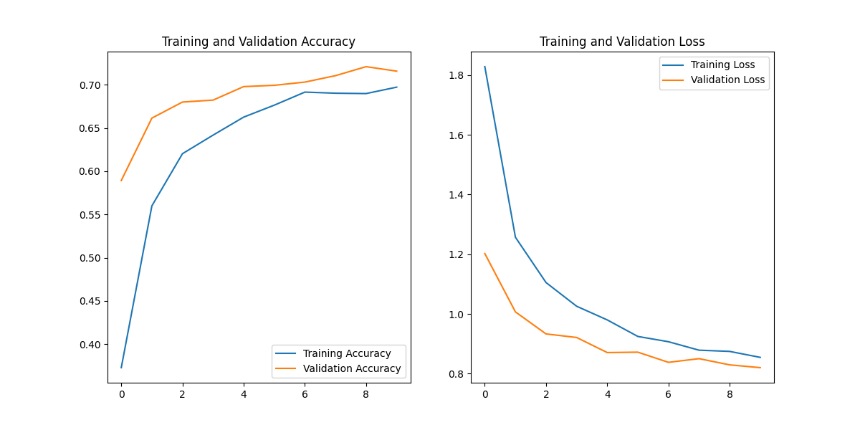


Figure Accuracy & Loss 3.2

Another promising indication within the results of version 3.2 can be seen in the plot and confusion matrix. The close proximity of the training and validation plot lines of both the accuracy and loss graphs suggest that the model is not overfitting, while the decent formation of the desirable diagonal line within the confusion matrix verifies the respectable accuracy scores.

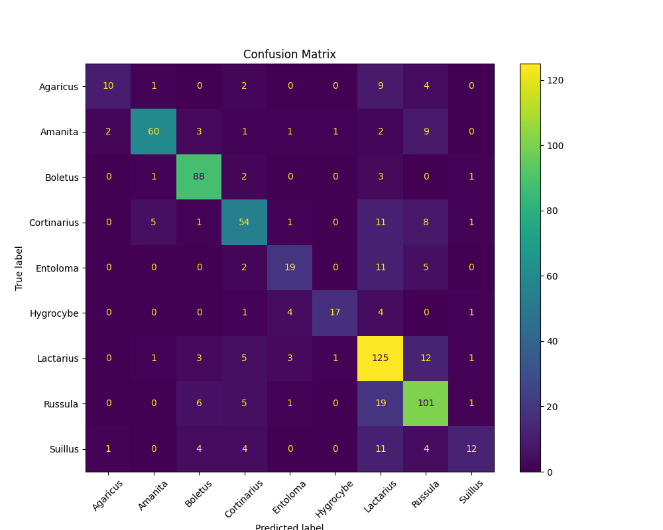


Figure Confusion Matrix 3.2

## 3.3

Following the acceptable results of 3.2, the [next test](https://github.com/yungroms/y3_proj/blob/89b4a412c3cbd6ffe5f6ce891884f0032992de72/3.3%20(5.4)/3.3.py) will explore the implementation of a balanced version of the *shrooms\_ds* dataset, and its impact on the performance metrics.

In regards to balancing the dataset, two choices seemed obvious: either reduce the dataset to match the smallest class volume, or increase the dataset to match the largest. In general, it is said that the greater the input data volume, the greater the results, and so the decision was made to increase the volume of each class within the *shrooms\_ds* to match the most populous one.

The method by which this expansion was achieved was through applying data augmentation to the images of the undersized classes, manipulating and adjusting them slightly, to produce new input images for the model. Augmentation methods included flipping horizontally and/or vertically, rotating, and zooming, however all augmentation amounts were kept minimal to avoid distortion, which could negatively affect the training process. The newly expanded mushroom dataset, named [*shrooms\_ds\_max*](https://www.kaggle.com/datasets/yungroms/shrooms-ds-max), was then implemented into the code by simply changing the value of *dataset\_path* to the location of the new dataset.

A black screen with white text

Description automatically generated

Figure dataset\_path 3.3

Test 3.3 was then executed with the same pre-trained architecture and hyperparameter values as 3.2, apart from the number of epochs, which was increased to 20 to allow more training of the larger input dataset.

## 3.3 Results

The table below contains the performance results of test run 3.3, featuring the training accuracy and loss as well as the validation accuracy and loss for each epoch of the test. By comparing the final rows of the results tables of 3.2 and 3.3, it can be seen that the increased input size of *shrooms\_ds\_max* has had an overall positive impact on the performance of the model. While the difference between the final training accuracy and loss values is slight, the larger volume of input data allowed for a substantially improved validation accuracy (0.71 for 3.2 versus 0.78 for 3.2) and validation loss (0.82 vs 0.67 respectively). Additionally, both validation metrics show gradual improvement as the epochs increase, which is another indication of the utility of the larger dataset.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Epoch | train\_accuracy | train\_loss | val\_accuracy | val\_loss |
| 1 | 0.44379058480262756 | 1.6309031248092651 | 0.6669034361839294 | 1.00746488571167 |
| 2 | 0.618100643157959 | 1.1240729093551636 | 0.7215909361839294 | 0.8555694818496704 |
| 3 | 0.6591923832893372 | 0.9948848485946655 | 0.734375 | 0.8018222451210022 |
| 4 | 0.6740056872367859 | 0.9369294047355652 | 0.7510653138160706 | 0.770259439945221 |
| 5 | 0.6837459206581116 | 0.9012853503227234 | 0.74609375 | 0.7653517127037048 |
| 6 | 0.6991680264472961 | 0.8773356080055237 | 0.7649147510528564 | 0.7342072129249573 |
| 7 | 0.700385570526123 | 0.8688456416130066 | 0.7670454382896423 | 0.7136929631233215 |
| 8 | 0.7046469449996948 | 0.856438159942627 | 0.7709516882896423 | 0.7181475162506104 |
| 9 | 0.7027191519737244 | 0.8587520718574524 | 0.7688210010528564 | 0.7020655870437622 |
| 10 | 0.7076907753944397 | 0.8419350981712341 | 0.7674005627632141 | 0.6976963877677917 |
| 11 | 0.7052556872367859 | 0.8510221838951111 | 0.7819602489471436 | 0.6771995425224304 |
| 12 | 0.7035308480262756 | 0.850041389465332 | 0.7688210010528564 | 0.7029702663421631 |
| 13 | 0.7068790793418884 | 0.8398709893226624 | 0.7755681872367859 | 0.6876222491264343 |
| 14 | 0.705864429473877 | 0.8405055999755859 | 0.7780539989471436 | 0.6636431217193604 |
| 15 | 0.7090097665786743 | 0.840477466583252 | 0.7819602489471436 | 0.6749394536018372 |
| 16 | 0.716010570526123 | 0.8249497413635254 | 0.7901278138160706 | 0.6604242324829102 |
| 17 | 0.7186485528945923 | 0.8304097056388855 | 0.7759233117103577 | 0.6811463236808777 |
| 18 | 0.717836856842041 | 0.8139025568962097 | 0.7784090638160706 | 0.6688222289085388 |
| 19 | 0.7167207598686218 | 0.8326066136360168 | 0.7816051244735718 | 0.6723821759223938 |
| 20 | 0.716010570526123 | 0.8319340348243713 | 0.7819602489471436 | 0.6670349836349487 |

The benefit of the larger dataset can also be confirmed by comparing the test metrics of 3.2 and 3.3: during the previous test, scores of 0.83 for test loss and 0.72 for test accuracy were recorded. During the current test, 3.3, improved metrics of 0.68 for test loss and 0.76 for test accuracy were achieved.

|  |  |
| --- | --- |
| Test Loss | 0.684188723564148 |
| Test Accuracy | 0.7634408473968506 |

Given that the only altered hyperparameter was the number of epochs, with all other details remaining identical, the 0.15 increase in test loss and 0.04 increase in test accuracy can be directly attributed to the increased size of the input dataset. The success of test run 3.3 is confirmed by the accuracy and loss plots below:

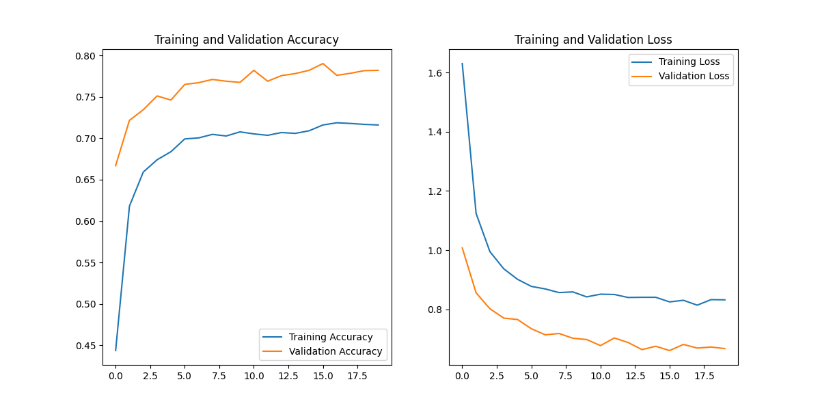


Figure Accuracy & Loss 3.3

Finally, confirmation of the improvement from 3.2 to 3.3 can be seen in the confusion matrix, in which the diagonal line of true positives is more brightly-coloured than its predecessor, while also containing higher values of successful predictions.

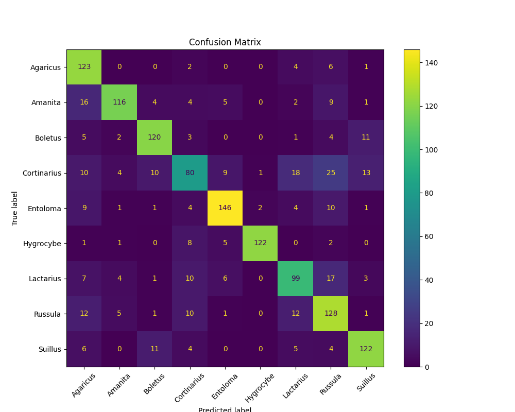


Figure Confusion Matrix 3.3

## 3.4

Following the successful implementation of *shrooms\_ds\_max* within the previous test round, it is now time to experiment with alternative pre-trained models in [3.4](https://github.com/yungroms/y3_proj/blob/89b4a412c3cbd6ffe5f6ce891884f0032992de72/3.4%20(5.5)/3.4.py). Until this point, all code versions of phase 3 have utilised *MobileNetV2* as the pre-trained architecture, and have subsequently produced solid results. However, an alternative family of pre-trained models was identified during the research stage, which may prove to be the most suitable for this relatively small-scale deep learning project, that being the *EfficientNet* family. LIL BIT BOUT EFFICIENTNET, IDEALLY WITH REFERENCE. The first architecture of the *EfficientNet* family to be implemented is *EfficientNetB0*.



Figure EfficientNetB0 Implementation 3.4

Implementing this new architecture is a simple process, requiring only the importation of the architecture from keras.applications, followed by the inclusion of *EfficientNetB0* within the establishment of the base model. For this run, all other code details will be kept identical, to allow direct comparison between this new version and the previous one.

## 3.4 Results

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Epoch | train\_accuracy | train\_loss | val\_accuracy | val\_loss |
| 1 | 0.10876623541116714 | 2.238626718521118 | 0.10830965638160706 | 2.1976335048675537 |
| 2 | 0.10805600881576538 | 2.2310092449188232 | 0.11115057021379471 | 2.1981799602508545 |
| 3 | 0.10856331139802933 | 2.232506036758423 | 0.13920454680919647 | 2.1983659267425537 |
| 4 | 0.10643263161182404 | 2.2319741249084473 | 0.12286932021379471 | 2.1977379322052 |
| 5 | 0.10947646200656891 | 2.231304883956909 | 0.10440340638160706 | 2.197862148284912 |
| 6 | 0.11333198100328445 | 2.232006549835205 | 0.1505681872367859 | 2.198192596435547 |
| 7 | 0.11282467842102051 | 2.22884202003479 | 0.1271306872367859 | 2.197988748550415 |
| 8 | 0.10541801899671555 | 2.233322858810425 | 0.11008522659540176 | 2.19805645942688 |
| 9 | 0.11059252917766571 | 2.22633695602417 | 0.11541192978620529 | 2.1983630657196045 |
| 10 | 0.11028815060853958 | 2.22463059425354 | 0.14098010957241058 | 2.197557210922241 |
| 11 | 0.11089691519737244 | 2.2263636589050293 | 0.10617897659540176 | 2.1982603073120117 |
| 12 | 0.11130276322364807 | 2.225799560546875 | 0.10724432021379471 | 2.197465658187866 |
| 13 | 0.11333198100328445 | 2.223799705505371 | 0.10617897659540176 | 2.1980137825012207 |
| 14 | 0.11678165942430496 | 2.222651958465576 | 0.14914773404598236 | 2.1977334022521973 |
| 15 | 0.11373782157897949 | 2.2249903678894043 | 0.10688920319080353 | 2.1982123851776123 |
| 16 | 0.11201298981904984 | 2.2243833541870117 | 0.10759942978620529 | 2.1981074810028076 |
| 17 | 0.10663555562496185 | 2.226109743118286 | 0.12428977340459824 | 2.197885751724243 |
| 18 | 0.11191152781248093 | 2.221920967102051 | 0.11115057021379471 | 2.1973581314086914 |
| 19 | 0.11241883039474487 | 2.22112774848938 | 0.11079545319080353 | 2.197192907333374 |
| 20 | 0.11302759498357773 | 2.220566749572754 | 0.14346590638160706 | 2.1975908279418945 |

The table above contains the results of test run 3.4. Unfortunately, it appears as though version 3.4 has produced the worst results of the entire project so far. Neither the training nor the validation accuracies ever reach 20%, and neither display any gradual improvement whatsoever as the epochs are completed, with the test accuracy only scoring 13.2%. The loss scores are just as poor, with the final training loss being 2.22, and the validation and test losses both being 2.20.

|  |  |
| --- | --- |
| Test Loss | 2.1971588134765625 |
| Test Accuracy | 0.13261649012565613 |

The poor performance of 3.4 can be seen even more clearly within the accuracy and loss visualisations. The accuracy graph to the left shows the consistently low score of the training accuracy line as it barely reaches 0.115 (11.5%), as well as the erratic validation accuracy line, which resembles an electro-cardio-graph more than a typical deep learning accuracy plot. The loss graph again reinforces this tragic performance, as even though the training loss declines minutely, neither the training nor validation losses get anywhere close to passing the already awful loss score of 2.00.

A graph of different colored lines

Description automatically generated with medium confidence

Figure Accuracy & Loss 3.4

Some explanation of the awful results of 3.4 can be found within the confusion matrix. It appears as though, for some strange reason, the model only ever gave two labels as its predictions for every single image it was given as input, those class labels being *Agaricus* and *Boletus*. After verifying that there was no issue with the dataset itself, and knowing with certainty that no adjustments were made to the code other than the replacement of the pre-trained model, it is evident that the code must be examined and adjusted to integrate the *EfficientNetB0* architecture.

A chart with different colored squares

Description automatically generated with medium confidence

Figure Confusion Matrix 3.4

While these initial *EfficientNet* results have been disappointing, this family of pre-trained architectures should be ideally-suited for this project. Additional research will be performed to ensure that the next official version and its tests will be better-adjusted to support *EfficientNetB0* and other related architectures, as successfully integrating this family of pre-trained models should reverse the outcome of 3.4 and produce some fantastic results.

Implementation & Testing – Model Development Phase 4

## 4.1

Following a brief period of additional research into the *EfficientNet* pre-trained architectures, it became apparent that two particular members of the family had the greatest potential for successful implementation. Subsequently, the decision was made to draft an entirely new program, intended to optimally implement these two most promising architectures, *EfficientNetV2S* and *EfficientNetB0*. These pre-trained models are promising due to their combinations of size, speed and accuracy [3], and the hope is that this promise will translate into highly competitive performance results.

The [new code](https://github.com/yungroms/y3_proj/blob/89b4a412c3cbd6ffe5f6ce891884f0032992de72/4.1%20(6.2)/4.1.py) version created for phase 4 shares many similarities with its predecessors, including the general layout and order of operations. The library importation section differs slightly: *OS* has been replaced with *Pathlib* to handle input and output directories; *Seaborn* has been imported to improve data visualisation; and the desired layers and models have been imported specifically to allow for cleaner implementation.

A screen shot of a computer

Description automatically generated

Figure Importation 4.1

Within the data preprocessing section, the *Keras* function image\_dataset\_from\_directory is used to handle the splitting of the dataset into training and validation sets. Unlike previous versions, codes 4.1 and onwards will utilise datasets which have already been separated into training/validation and test sets, meaning the code just needs to handle the splitting of the training/validation sets into respective training and validation sets. This has been done to ensure that all tests utilise the same set, meaning models are tested using the same images.

A screen shot of a computer program

Description automatically generated

Figure Dataset Split 4.1

The next section contains the specification of the hyperparameter values, as well as the start of the model definition, which begins will the instantiation of the pre-trained *EfficientNet* architecture.

A computer screen shot of a program code

Description automatically generated

Figure Model Definition 4.1

After the pre-trained architecture is included within the definition of the ‘full’ model, additional layers are then added to serve as final layers which flatten and transform the pre-trained model’s values into the desired output format.

A black screen with white text

Description automatically generated

Figure Output Layers 4.1

The model is then compiled, with the specification of important parameters such as the optimisation method, learning rate and loss function.

A black screen with white text

Description automatically generated

Figure Model Compilation 4.1

Next comes a new inclusion, which aims to improve the data recoding process of the program: the *CSVLoggerCallback*. The purpose of this useful feature is to automatically document the training and validation results after each epoch, in a cleaner and simpler fashion than the implementations in previous versions.

A screen shot of a computer screen

Description automatically generated

Figure Logger Callback 4.1

Model training is implemented next, including in its parameters the training and validation datasets, the number of epochs, and the newly-created logging callback. Following this, the model saving process is defined.

A screen shot of a computer program

Description automatically generated

Figure Model Training 4.1

The final sections of code 4.1 are also similar but improved in comparison to previous versions, containing the data visualisation process for plotting the accuracy and loss line graphs, followed by the definition of a new function for the testing process called *evaluate\_model*. This function incorporates the evaluation of the model using the pre-established test dataset, as well as the creation of the test results’ confusion matrix.

A computer screen shot of text

Description automatically generated

Figure Evaluate Model 4.1

For test run 4.1, the learning rate will be 0.001, the batch size will be 32, and the number of epochs will equal 10. Additionally, the chosen dataset will be a new version of the mushroom dataset named [*shrooms\_ds\_split*](https://www.kaggle.com/datasets/yungroms/shrooms-ds-split), updated to be pre-split into testing and training/validation sets.

## 4.1 Results

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| epoch | train\_loss | train\_accuracy | val\_loss | val\_accuracy |
| 1 | 1.1358641386032104 | 0.6073740124702454 | 0.8104385733604431 | 0.7142857313156128 |
| 2 | 0.7446956634521484 | 0.7466169595718384 | 0.6739667654037476 | 0.7778649926185608 |
| 3 | 0.6397733688354492 | 0.7766228914260864 | 0.6440829038619995 | 0.779434859752655 |
| 4 | 0.5722954273223877 | 0.7983918190002441 | 0.5954170227050781 | 0.7849293351173401 |
| 5 | 0.5141079425811768 | 0.8254559636116028 | 0.6367316246032715 | 0.7786499261856079 |
| 6 | 0.4740147888660431 | 0.8336929082870483 | 0.6287651062011719 | 0.784144401550293 |
| 7 | 0.45026329159736633 | 0.8450676798820496 | 0.5533455014228821 | 0.8006279468536377 |
| 8 | 0.4089750051498413 | 0.8558540940284729 | 0.5666468739509583 | 0.802982747554779 |
| 9 | 0.38659313321113586 | 0.8636987805366516 | 0.5268163681030273 | 0.8218210339546204 |
| 10 | 0.3500184118747711 | 0.8758580088615417 | 0.5276644825935364 | 0.8163265585899353 |

As can be seen in the training and validation results table above, test run 4.1 has produced very promising results. Using the *EfficientNetV2S* pre-trained architecture, the model created in 4.1 has achieved the best scores of any model trained and tested on a mushroom dataset. The previous top results were recorded during stage 3.3, which achieved accuracies of 0.72, 0.78 and 0.76 for training, validation and testing respectively, alongside loss scores of 0.83, 0.66 and 0.68. In comparison, test run 4.1 recorded accuracies of 0.87, 0.82 and 0.82 and losses of 0.35, 0.52 and 0.57. These are significant improvements which must be attributed to the *EfficientNetV2S* architecture as well as the improved code.

|  |  |
| --- | --- |
| loss | 0.5741644501686096 |
| accuracy | 0.820588231086731 |

Additionally, the model created during 3.3 was trained on *shrooms\_ds\_max*, which is substantially larger than the dataset used to train the model of 4.1. This is an indication of the utility of EfficientNetV2S within this project, providing promise that utilising the larger dataset to train and test a model with this configuration could produce even better results.

A graph of different colored lines

Description automatically generated with medium confidence

Figure Accuracy & Loss 4.1

Although the results of 4.1 are promising, room for improvement can be seen in the visualisations of its results. Firstly, in the accuracy and line graphs above, the separation between the training and validation lines is suggestive of overfitting, whereby the model is learning its training data too specifically, and may be comparatively underachieving on the validation and test sets. Also, the intersection of the training and validation lines occurs rather early on in both graphs. This suggests that the learning rate might be too high, and that potential improvement could be achieved by lowering it slightly.

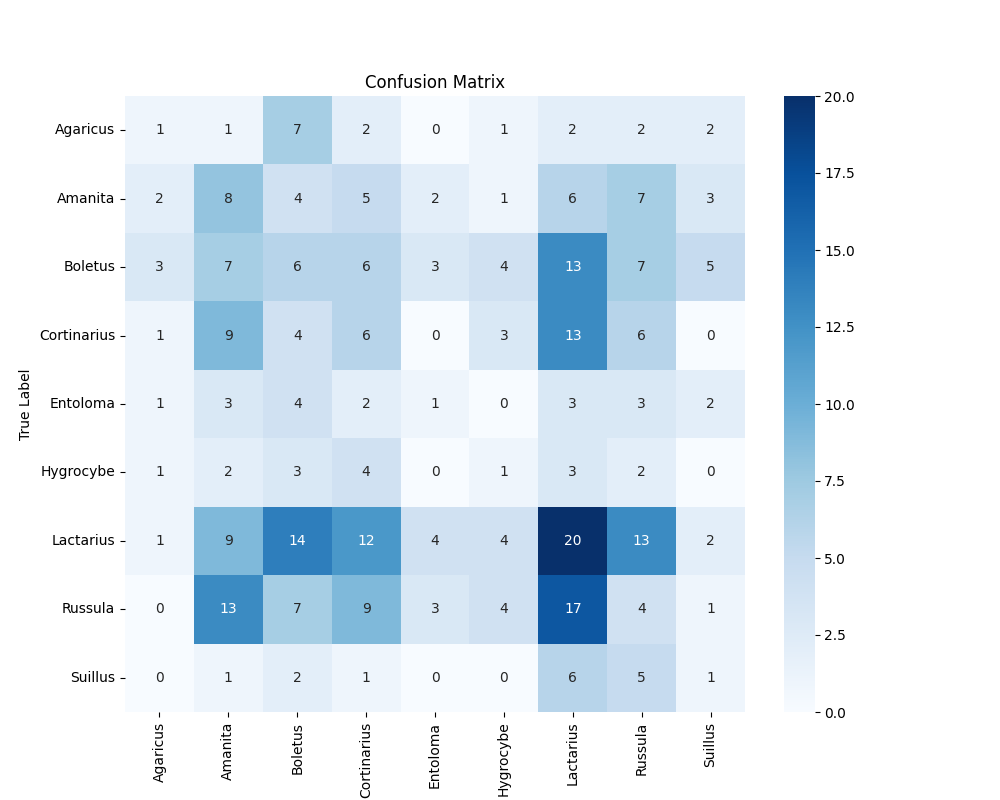


Figure Confusion Matrix 4.1

Room for further improvement can be found within the confusion matrix, whose appearance seems to conflict the success of this round of testing, as the desirable diagonal line of high values is barely present. Instead, this matrix appears to represent an inaccurate model, which performed reasonably well on certain classes such as *Lactarius* and *Amanita*, but that accuracy must be improved across entire dataset. Alterations must be made to improve the performance metrics of this model.

## 4.2

Before substantial changes are implemented, the exact same code will be executed, this time utilising *EfficientNetB7*, the other promising member of the *EfficientNet* architectural family. The hyperparameter values will be kept identical to the previous one, to isolate the architecture as the only altered variable during versions 4.1 and [4.2](https://github.com/yungroms/y3_proj/blob/89b4a412c3cbd6ffe5f6ce891884f0032992de72/4.2%20(6.4)/4.2.py).

As explained during previous versions, switching architectures is as simple as changing the specification of the importation from keras.applications.



Figure EfficientNet Importation 4.2

The only other necessary step is to adjust the definition of the model to match the choice of imported architecture.

A computer screen shot of a program code

AI-generated content may be incorrect.

Figure Model Definition 4.2

## 4.2 Results

Below are the results of version 4.2, utilising the *EfficientNetB7* pre-trained model. Once again, it appears that the new code version created for phase 4 allows the *EfficientNet* architecture to achieve better results than those achieved with its implementation in phase 3.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| epoch | train\_loss | train\_accuracy | val\_loss | val\_accuracy |
| 1 | 1.1454973220825195 | 0.5922729969024658 | 0.8210660815238953 | 0.7260596752166748 |
| 2 | 0.744627058506012 | 0.7428907752037048 | 0.709729015827179 | 0.7653061151504517 |
| 3 | 0.6187347173690796 | 0.7811335325241089 | 0.7632706761360168 | 0.7386185526847839 |
| 4 | 0.5369474291801453 | 0.8134928345680237 | 0.5996302366256714 | 0.7998430132865906 |
| 5 | 0.4531058967113495 | 0.8407530784606934 | 0.646831750869751 | 0.7880690693855286 |
| 6 | 0.408512145280838 | 0.8633065223693848 | 0.6600754261016846 | 0.784144401550293 |
| 7 | 0.3473513722419739 | 0.8772308230400085 | 0.6196209788322449 | 0.795918345451355 |
| 8 | 0.31165143847465515 | 0.8897823095321655 | 0.5546718239784241 | 0.8202511668205261 |
| 9 | 0.28371602296829224 | 0.9029221534729004 | 0.5934820175170898 | 0.8116169571876526 |
| 10 | 0.24702267348766327 | 0.9123357534408569 | 0.5574294328689575 | 0.8233909010887146 |

The results recorded are similar to 4.1, which utilised its architectural relative *EfficientNetV2S*, with the training metrics of 4.2 actually surpassing its predecessor, the validation and test accuracies being identical, and the validation and test losses being slightly worse.

|  |  |
| --- | --- |
| loss | 0.5930107831954956 |
| accuracy | 0.8176470398902893 |

The slightly larger distance between the training and validation/test results of 4.2 might be indicative of a little more overfitting, as seen in the plots below. But in general, the results of 4.1 and 4.2 are comparable, demonstrating the similar abilities of the two *EfficientNet* versions in use.

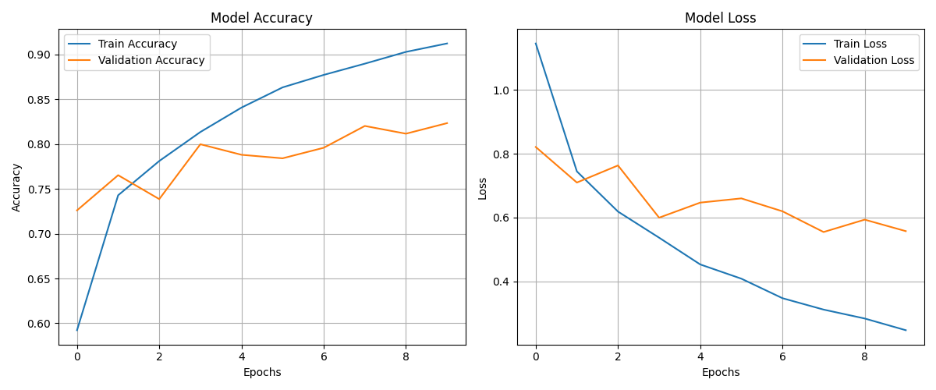


Figure Accuracy & Loss 4.2

Unfortunately, the confusion matrix again dampens the success of round 4.2, appearing almost identical to the matrix of 4.1. The desired diagonal line of true positives is absent, meaning that this model is also struggling to predict the classes with much success. This issue could be due to the fact that the *shrooms\_ds\_split* dataset is poorly balanced, meaning that the model is fed more images of certain classes. This would then cause the model to become overly familiar with some classes and not familiar enough with others.

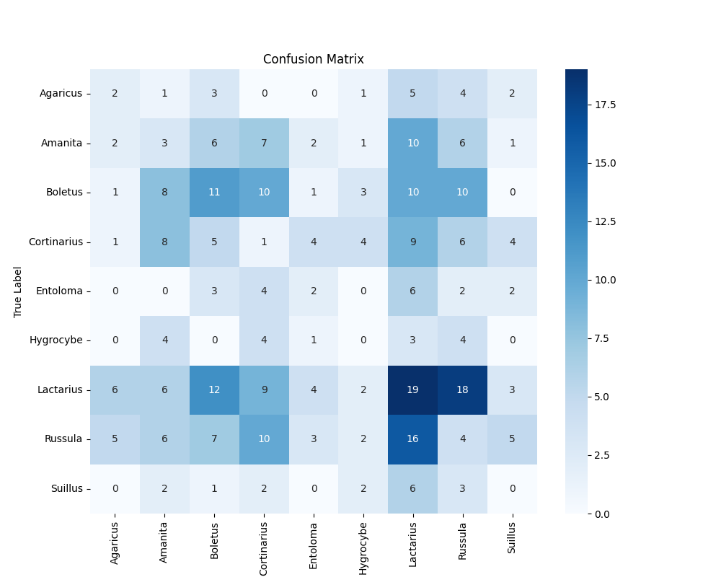


Figure Confusion Matrix 4.2

This issue can either be solved by balancing the dataset, as has been done previously with *shrooms\_ds\_max*, or by calculating and applying class weights to the model during training, enabling the under-represented classes to be given greater importance/emphasis. Both these options will be explored at a later stage of phase 4.

## 4.3

The next code version, [4.3](https://github.com/yungroms/y3_proj/blob/26c900e22d5d1cdba91a58fecca52322fa47bf12/4.3%20(6.7)/4.3.py), returns to the use of the EfficentNetB7 architecture, alongside some additional implementations and an alternative dataset.

Leaving the mushroom datasets for a moment, this version introduces a new version of the *LeafSnap* dataset. Whereas previous versions had utilised only the lab images, which were consistent and clear but limited in volume, this version of the dataset merges the lab and field images for each class, massively increasing the number of images available for use, while sacrificing a little consistency between images. In accordance with a successful integration into the code, the dataset has also been split into training/validation and test sets, with the split between training and validation occurring inside the code. Again, this simply ensures that the set of test images remains consistent. This new dataset, named *LeafSnap\_15\_merged\_split*, can be found [here](https://www.kaggle.com/datasets/yungroms/leafsnap-15-merged-split).

One new inclusion within this code is the classification report, which provides the additional useful performance metrics: precision, recall, F1-score, and support. The creation of the report is handled by classification\_report, an import from *SciKit* *Learn*’s metrics package, which calculates these performance metrics for each class and overall. These metrics are then neatly written to a CSV file, so they can be analysed and tabulated or visualised at a later stage. Explanations of these additional performance metrics can be found here, within the Methodology & Design section of the project.

A screen shot of a computer screen

Description automatically generated

Figure Classification Report 4.3

The only additional feature implemented within 4.3 is the function to store the confusion matrix in CSV format, in addition to the already existing PNG format. Doing so provides an additional copy of the confusion matrix data, enabling a tabular view of the matrix, as well as the creation of new visualisations in the event that something goes wrong with the ones created by the program.

Hyperparameter values within code 4.3 remain unchanged from the previous version.

## 4.3 Results

The table below contains the training and validation results of the test run for version 4.3. Based on the accuracies and losses of the training and validation sets, the EfficientNetB7-implemented model has learned the dataset’s classes and features very well. By the end of the first epoch the results look great, and even with fluctuations along the way, the results in the final row show the improvement of the model’s performance throughout the training process.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Epoch | train\_loss | train\_accuracy | val\_loss | val\_accuracy |
| 1 | 0.6041288375854492 | 0.819915235042572 | 0.09760291874408722 | 0.9787685871124268 |
| 2 | 0.09363394975662231 | 0.9751059412956238 | 0.06681859493255615 | 0.9766454100608826 |
| 3 | 0.05764128640294075 | 0.9867584705352783 | 0.10196834802627563 | 0.9639065861701965 |
| 4 | 0.03619294613599777 | 0.9920550584793091 | 0.058931030333042145 | 0.9787685871124268 |
| 5 | 0.08439228683710098 | 0.9692796468734741 | 0.09217304736375809 | 0.966029703617096 |
| 6 | 0.043456628918647766 | 0.9862288236618042 | 0.030875694006681442 | 0.9893842935562134 |
| 7 | 0.017142845317721367 | 0.9973517060279846 | 0.04061240330338478 | 0.9830148816108704 |
| 8 | 0.016026057302951813 | 0.9968220591545105 | 0.07946506887674332 | 0.9681528806686401 |
| 9 | 0.018259607255458832 | 0.9941737055778503 | 0.03020964190363884 | 0.9851379990577698 |
| 10 | 0.010882783681154251 | 0.9978813529014587 | 0.04407785087823868 | 0.9830148816108704 |

These excellent metrics are reinforced by the test results, which are the remarkably low loss of 0.007 and the perfect accuracy of 1.0. Based on the tables above and below, this test has been a complete success.

|  |  |
| --- | --- |
| loss | accuracy |
| 0.007476646453142166 | 1.0 |

Further confirmation of excellent performance can be seen in the graphs below. Although the fluctuation is more pronounced when visualised, particularly for the validation lines, the results still look very promising.

A graph of a graph of a graph

AI-generated content may be incorrect.

Figure Accuracy & Loss 4.3

What is less promising is the first edition of the newly-implemented classification report. Ideally, each of the three key metrics, precision, recall and F1-score, will be values somewhere less than but close to 1. Even though the above results are very promising, the classification report table is full of values barely above 0.

This truly is conflicting data. It can be assured that the test data is entirely separate from the training/validation sets, meaning they are new. The results above indicate that the model performed brilliantly in the test, however according to the table below, it failed to learn or predict any class with much success. The code will have to be re-verified to eliminate the possibility of it being the source of this indiscretion, and to confirm that the difference in suggested performance is a legitimate cause of concern.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Class | Precision | Recall | F1-Score | Support |
| acer\_negundo | 0.1 | 0.1 | 0.1 | 10.0 |
| acer\_palmatum | 0.0 | 0.0 | 0.0 | 6.0 |
| aesculus\_pavi | 0.0 | 0.0 | 0.0 | 7.0 |
| asimina\_triloba | 0.0 | 0.0 | 0.0 | 10.0 |
| cercis\_canadensis | 0.0 | 0.0 | 0.0 | 9.0 |
| chionanthus\_virginicus | 0.0 | 0.0 | 0.0 | 9.0 |
| gleditsia\_triacanthos | 0.0 | 0.0 | 0.0 | 8.0 |
| ilex\_opaca | 0.091 | 0.091 | 0.091 | 11.0 |
| liriodendron\_tulipifera | 0.1 | 0.1 | 0.1 | 10.0 |
| ostrya\_virginiana | 0.2 | 0.2 | 0.2 | 10.0 |
| prunus\_sargentii | 0.0 | 0.0 | 0.0 | 6.0 |
| ptelea\_trifoliata | 0.273 | 0.273 | 0.273 | 11.0 |
| quercus\_montana | 0.1 | 0.1 | 0.1 | 10.0 |
| styrax\_japonica | 0.0 | 0.0 | 0.0 | 6.0 |
| ulmus\_pumila | 0.0 | 0.0 | 0.0 | 6.0 |
| macro avg | 0.058 | 0.058 | 0.058 | 129.0 |
| weighted avg | 0.070 | 0.070 | 0.070 | 129.0 |

The confusion matrix below supports the classification report in its assertion that the accuracy and loss scores are misleading, and that the model is far from performing optimally. Once again, the diagonal line of true positives is non-existent, with values spaced sporadically within the matrix, suggesting poor generalisation and accuracy.

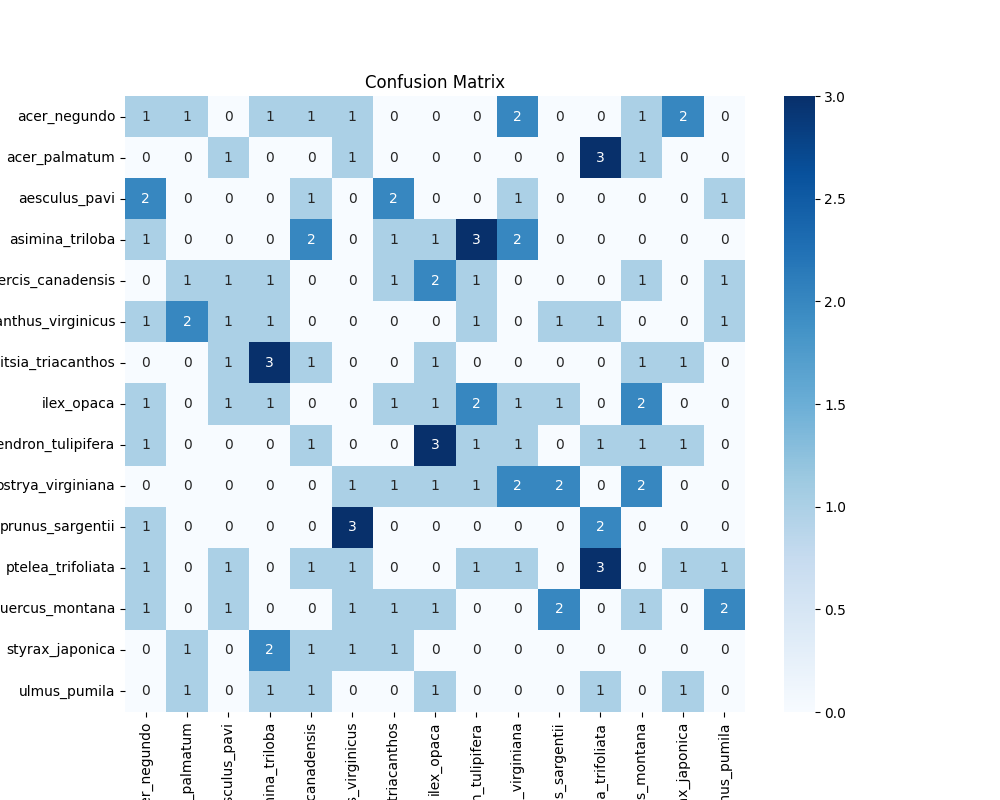


Figure Confusion Matrix 4.3

One possible source of this poor performance could be the dataset imbalance, which was pointed out during the conclusion of a previous phase 4 test run. This must be addressed through the implementation of a solution: either through manually balancing the dataset, or through integrating method of calculating class weights, to support under-represented classes and diminish the impact of larger ones. That being said, the next test will use the same code as 4.3, however *EfficientNetV2S* will be utilised as the pre-trained model. An attempt to tackle the class imbalance issue will be implemented following this next test, in versions 4.5 and 4.6.

## 4.4

Even though the results of version 4.3 were underwhelming, suggesting the need for analysis and improvement, version [4.4](https://github.com/yungroms/y3_proj/blob/26c900e22d5d1cdba91a58fecca52322fa47bf12/4.4%20(6.8)/4.4.py) will continue the phase 4 pattern of implementing both chosen *EfficientNet* models into each code version, to satisfy this phase’s main target of determining which of the two is best for the project. After implementing *EfficientNetB7* in the previous version, it is now time to test the implementation of *EfficientNetV2S* within this current code version.

The process of implementing an alternative architecture has been covered in previous sections of the project. However, if brief, all that needs to be done is to specify the desired model within the code’s importation section, then instantiate it within the complete model definition.

A computer screen shot of a program code

Description automatically generated

Figure Model Definition 4.4

## 4.4 Results

The table below contains the loss and accuracy data for both the training and validation sets. Again, it appears as though the models containing the *EfficientNetV2S* and *EfficientNetB7* architectures are highly comparable in terms of performance. The loss values for both training and validation sets can be approximated to 0.03, while the accuracy values approximate to 0.99, or 99%. Although only marginal, these results do surpass those achieved during 4.3, suggesting that *EfficientNetV2S* is the optimal pre-trained architecture, certainly within the version of code used within versions 4.3 and 4.4.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| epoch | train\_loss | train\_accuracy | val\_loss | val\_accuracy |
| 1 | 0.7184862494468689 | 0.7934321761131287 | 0.16662226617336273 | 0.9532908797264099 |
| 2 | 0.1375671923160553 | 0.9608050584793091 | 0.06190510839223862 | 0.9893842935562134 |
| 3 | 0.05877101793885231 | 0.9862288236618042 | 0.05029682442545891 | 0.9893842935562134 |
| 4 | 0.04904114082455635 | 0.991525411605835 | 0.03829913213849068 | 0.9915074110031128 |
| 5 | 0.04308179020881653 | 0.9894067645072937 | 0.03887119144201279 | 0.9915074110031128 |
| 6 | 0.0273716002702713 | 0.9947034120559692 | 0.044218435883522034 | 0.9893842935562134 |
| 7 | 0.03577205538749695 | 0.992584764957428 | 0.029522456228733063 | 0.993630588054657 |
| 8 | 0.029924970120191574 | 0.9936440587043762 | 0.03187692537903786 | 0.9915074110031128 |
| 9 | 0.047307584434747696 | 0.9841101765632629 | 0.03998721018433571 | 0.9851379990577698 |
| 10 | 0.029829248785972595 | 0.9899364113807678 | 0.029376517981290817 | 0.993630588054657 |

The excellent results above are replicated, if not exceeded, within the test set. A perfect accuracy score of 1.0 (100%) was achieved, while the loss score of the test set was even lower than those achieved during the training or validation sets, reaching the remarkably low value of 0.006.

|  |  |
| --- | --- |
| loss | 0.005861646495759487 |
| accuracy | 1.0 |

The success of 4.4 is reiterated by the visualisations of the accuracies and losses of both the training sets. In comparison with the graphs produced during 4.3, the plots of 4.4 show less fluctuation as well as less space between the training and validation lines. This indicates more consistent learning and less overfitting, both of which are indications of the superiority of the *EfficientNetV2S* architecture.

A graph of a graph of a graph

AI-generated content may be incorrect.

Figure Accuracy & Loss 4.4

Again, since no analysis has been performed nor any improvements made since 4.3, version 4.4 has produced another underwhelming classification report. Every value within the columns of each of the three important metrics is nowhere near desirable, suggesting in contrast to the results above that the model is poor at accurately classifying any of the 15 categories. As stated previously, one source of this incongruency between results could be the imbalance of the dataset, and so the priority within future versions will be to attempt to address this shortcoming.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Class | Precision | Recall | F1-Score | Support |
| acer\_negundo | 0.0 | 0.0 | 0.0 | 10.0 |
| acer\_palmatum | 0.0 | 0.0 | 0.0 | 6.0 |
| aesculus\_pavi | 0.143 | 0.143 | 0.143 | 7.0 |
| asimina\_triloba | 0.3 | 0.3 | 0.3 | 10.0 |
| cercis\_canadensis | 0.0 | 0.0 | 0.0 | 9.0 |
| chionanthus\_virginicus | 0.0 | 0.0 | 0.0 | 9.0 |
| gleditsia\_triacanthos | 0.0 | 0.0 | 0.0 | 8.0 |
| ilex\_opaca | 0.0 | 0.0 | 0.0 | 11.0 |
| liriodendron\_tulipifera | 0.1 | 0.1 | 0.1 | 10.0 |
| ostrya\_virginiana | 0.0 | 0.0 | 0.0 | 10.0 |
| prunus\_sargentii | 0.0 | 0.0 | 0.0 | 6.0 |
| ptelea\_trifoliata | 0.091 | 0.091 | 0.091 | 11.0 |
| quercus\_montana | 0.0 | 0.0 | 0.0 | 10.0 |
| styrax\_japonica | 0.167 | 0.167 | 0.167 | 6.0 |
| ulmus\_pumila | 0.0 | 0.0 | 0.0 | 6.0 |
| macro avg | 0.053 | 0.053 | 0.053 | 129.0 |
| weighted avg | 0.054 | 0.054 | 0.054 | 129.0 |

As was the case with 4.3, the confusion matrix produced in 4.4 does not reinforce the optimistic accuracy and loss data, instead supporting the poor assessment of the classification report. Effort must be made to improve the alignment of the classification and confusion matrix data with the accuracy and loss data.

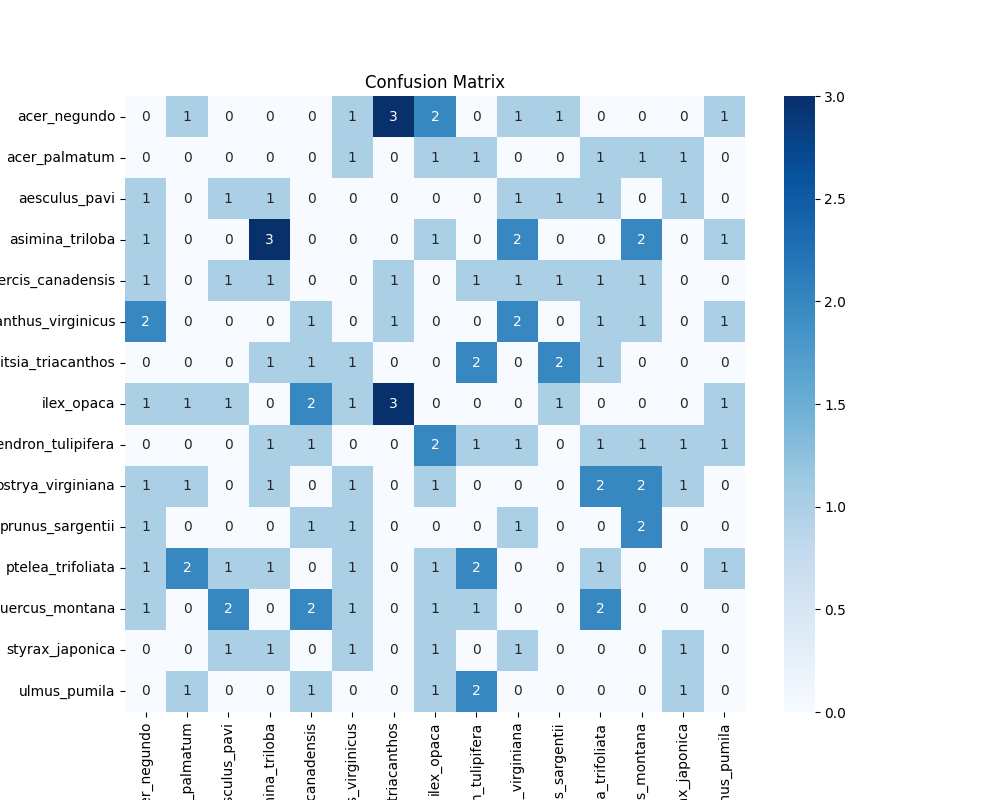


Figure Confusion Matrix 4.4

## 4.5

In an attempt to positively impact the confusion matrix and classification report, [4.5](https://github.com/yungroms/y3_proj/blob/26c900e22d5d1cdba91a58fecca52322fa47bf12/4.5%20(6.9)/4.5.py) includes the functionality to calculate and apply class weights within its code. By applying the concept of weighting, the imbalance between classes with different volumes can be reduced, ensuring under-represented classes are emphasised appropriately during the training process.

Aside from this new functionality, the other aspects of 4.5’s code are the same as 4.3-4.4, including the hyperparameter values and the input dataset. *EfficientNetB7* is the architecture version used in this test.

The calculation of the class weights is handled by the compute\_class\_weight function imported from the utils package of *Scikit-learn*:



Figure Class Weighting Importation 4.5

Once calculated, the class weights are then stored in a dictionary:

A computer screen with text on it

Description automatically generated

Figure Class Weighting Implementation 4.5

This dictionary is then included as a parameter within the fit function during model training:

A computer screen with white text

AI-generated content may be incorrect.

Figure Model Training 4.5

## 4.5 Results

The training and validation results table below contains very promising accuracy and loss data. In comparison with the previous code version used in 4.3 and 4.4, which did not contain class weighting, 4.5 displays better scores in earlier epochs, while ending with comparably outstanding values. This suggests that the inclusion of class weights has a positive impact on the predictability of the model, allowing it to comprehend the entire dataset sooner.

That being said, the results from the final epoch are actually slightly worse than those of the penultimate epochs, with peak performance occurring between epochs 7 and 9.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| epoch | train\_loss | train\_accuracy | val\_loss | val\_accuracy |
| 1 | 0.5868857502937317 | 0.8389830589294434 | 0.11270599812269211 | 0.966029703617096 |
| 2 | 0.0859081894159317 | 0.977224588394165 | 0.11162427812814713 | 0.9617834687232971 |
| 3 | 0.06531272828578949 | 0.9809321761131287 | 0.05562799051403999 | 0.9830148816108704 |
| 4 | 0.05372973904013634 | 0.9830508232116699 | 0.058138228952884674 | 0.9808917045593262 |
| 5 | 0.050654087215662 | 0.9809321761131287 | 0.05131078511476517 | 0.9893842935562134 |
| 6 | 0.03206135705113411 | 0.9888771176338196 | 0.0506136491894722 | 0.9830148816108704 |
| 7 | 0.01424815971404314 | 0.9968220591545105 | 0.044224608689546585 | 0.9830148816108704 |
| 8 | 0.011650074273347855 | 0.9973517060279846 | 0.02639131247997284 | 0.987261176109314 |
| 9 | 0.006913952063769102 | 0.9989407062530518 | 0.025180019438266754 | 0.9915074110031128 |
| 10 | 0.029858706519007683 | 0.9888771176338196 | 0.05928351357579231 | 0.9787685871124268 |

The test results do suggest that the slight dip in the final epoch is not detrimental, however, as the test loss and accuracy are closer to the values found at the training and validation peaks as opposed to the tenth and final epoch.

|  |  |
| --- | --- |
| loss | 0.014814283698797226 |
| accuracy | 0.9922480583190918 |

The training and validation plots for accuracy and loss clearly visualise the peak around the 8th epoch, as well as the slight dip towards the end. This visualisation suggests that the optimal number of epochs may be slightly less than the currently used value of 10.

A graph of a graph

Description automatically generated with medium confidence

Figure Accuracy & Loss 4.5

Unfortunately, the promising conclusions derived from the data above is not reflected in the classification report or confusion matrix below. Once again, poor values can be found throughout the classification report table, with no class achieving scores greater than 0.2, and the averages at the bottom barely exceeding 0.05. Clearly, implementing class weighting has had little positive effect on these performance metrics, which would have been the desirable outcome.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Class | Precision | Recall | F1-Score | Support |
| acer\_negundo | 0.2 | 0.2 | 0.2 | 10.0 |
| acer\_palmatum | 0.0 | 0.0 | 0.0 | 6.0 |
| aesculus\_pavi | 0.0 | 0.0 | 0.0 | 7.0 |
| asimina\_triloba | 0.0 | 0.0 | 0.0 | 10.0 |
| cercis\_canadensis | 0.0 | 0.0 | 0.0 | 9.0 |
| chionanthus\_virginicus | 0.0 | 0.0 | 0.0 | 9.0 |
| gleditsia\_triacanthos | 0.0 | 0.0 | 0.0 | 8.0 |
| ilex\_opaca | 0.0 | 0.0 | 0.0 | 11.0 |
| liriodendron\_tulipifera | 0.0 | 0.0 | 0.0 | 10.0 |
| ostrya\_virginiana | 0.1 | 0.1 | 0.1 | 10.0 |
| prunus\_sargentii | 0.167 | 0.167 | 0.167 | 6.0 |
| ptelea\_trifoliata | 0.091 | 0.091 | 0.091 | 11.0 |
| quercus\_montana | 0.2 | 0.2 | 0.2 | 10.0 |
| styrax\_japonica | 0.0 | 0.0 | 0.0 | 6.0 |
| ulmus\_pumila | 0.0 | 0.0 | 0.0 | 6.0 |
| macro avg | 0.051 | 0.051 | 0.051 | 129.0 |
| weighted avg | 0.054 | 0.054 | 0.054 | 129.0 |

As is the case for the classification report table above, the confusion matrix shows results which are equally conflicting with the accuracy and loss results. The inclusion of class weighting appears to have had negligible positive impact on the confusion matrix, as the distribution of values is no better than in the previous two rounds of testing. Clearly, more improvement is required within the model’s code to increase the alignment of the first results section with the second.

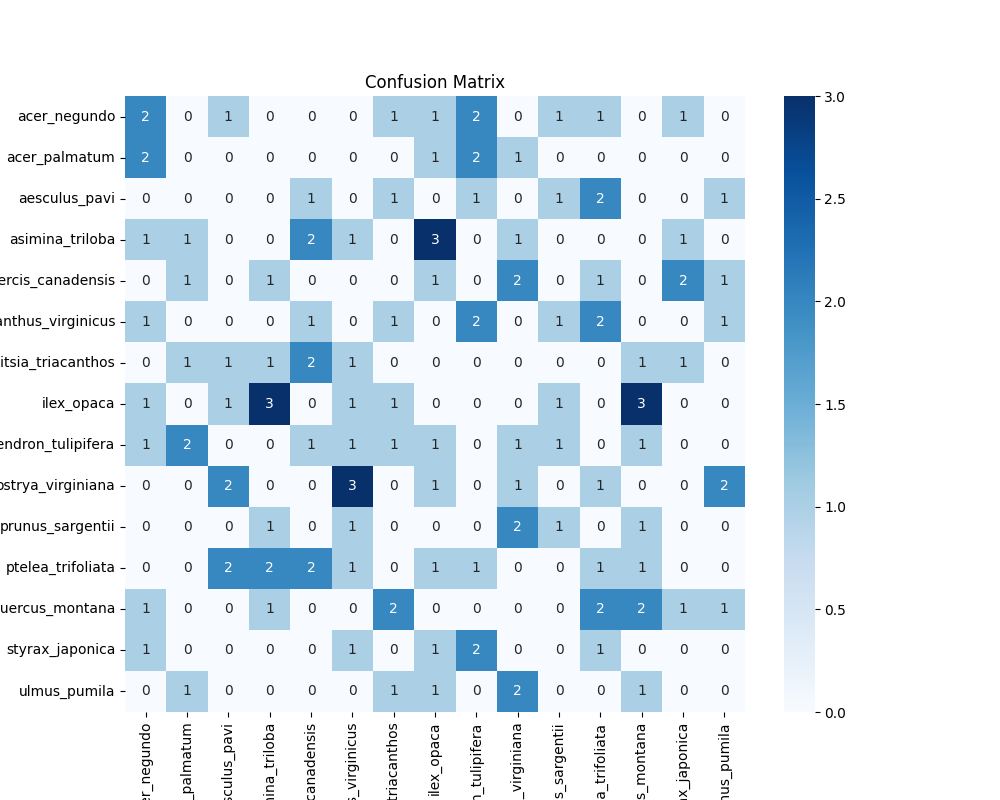


Figure Confusion Matrix 4.5

## 4.6

In continuation of the phase 4 trend of testing both *EfficientNet* versions within the code before applying updates, test [4.6](https://github.com/yungroms/y3_proj/blob/26c900e22d5d1cdba91a58fecca52322fa47bf12/4.6%20(6.10)/4.6.py) will simply include *EfficientNetV2S* as the pre-trained architecture. Based on the results of 4.5, little optimism is held regarding the outcome of the classification report and confusion matrix, but promising results are anticipated for the training and validation loss and accuracy results.

The figure below displays the simple variation required to implement *EfficientNetV2S* in place of *EfficientNetB7*:

A screen shot of a computer program

Description automatically generated

Figure Model Definition 4.6

## 

## 4.6 Results

As expected, test run 4.6 has produced excellent accuracy and loss results during training and validation. From the 2nd epoch onwards, the training and validation accuracy values around 95% or higher. From the 3rd epoch onwards, both the training and validation loss values are less than 0.1. In the 10th and final epoch, the validation results of 99.6% for accuracy and 0.017 for loss are representative of a model with fantastic predictability.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| epoch | train\_loss | train\_accuracy | val\_loss | val\_accuracy |
| 1 | 0.7032853960990906 | 0.7976694703102112 | 0.17450560629367828 | 0.9426751732826233 |
| 2 | 0.13693077862262726 | 0.9661017060279846 | 0.07915088534355164 | 0.987261176109314 |
| 3 | 0.07867415249347687 | 0.9735169410705566 | 0.04280387610197067 | 0.9957537055015564 |
| 4 | 0.048738569021224976 | 0.9872881174087524 | 0.04487476870417595 | 0.9893842935562134 |
| 5 | 0.03554752469062805 | 0.992584764957428 | 0.040064722299575806 | 0.9893842935562134 |
| 6 | 0.0530436672270298 | 0.9862288236618042 | 0.029003949835896492 | 0.993630588054657 |
| 7 | 0.030787236988544464 | 0.9931144118309021 | 0.035568684339523315 | 0.9915074110031128 |
| 8 | 0.04747902229428291 | 0.9883474707603455 | 0.030998704954981804 | 0.9915074110031128 |
| 9 | 0.04850604012608528 | 0.9894067645072937 | 0.016278285533189774 | 0.9957537055015564 |
| 10 | 0.03503787890076637 | 0.9904661178588867 | 0.016674816608428955 | 0.9957537055015564 |

The training and validation results of the final epoch are reflected in the results of the test set, in which an accuracy of 100% and a loss of 0.0039 were achieved. The model has performed equally impressively on this previously unseen data as it did with the training and validation sets.

|  |  |
| --- | --- |
| test\_loss | 0.003856297116726637 |
| test\_accuracy | 1.0 |

The training and validation plots of the accuracy and loss graphs below also reflect the brilliant results found in the above tables. In comparison to the graphs of 4.5, it appears as though 4.6 has produced even better results, as there is less separation between the training and validation plots, and no dip in performance in the last epoch. As has been the case in the last three version pairs, it seems that *EfficientNetV2S* slightly outperforms its architectural relative *EfficentNetB7*.

A graph of a graph of a graph

Description automatically generated with medium confidence

Figure Accuracy & Loss 4.6

Unfortunately, the optimistic results above do not follow through to the classification report. In fact, unlike in the accuracy and loss results, *EfficientNetV2S* actually performs slightly worse than *B7* in important metrics such as precision, recall and F1-Score. The highest value within the table below is 0.167, a little less than the highest value scored in 4.5, which was 0.2. Similarly, 4.5 achieved a weighted average of 0.054, while 4.6 only achieved a weighted average of 0.047.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Class | Precision | Recall | F1-Score | Support |
| acer\_negundo | 0.1 | 0.1 | 0.1 | 10.0 |
| acer\_palmatum | 0.167 | 0.167 | 0.167 | 6.0 |
| aesculus\_pavi | 0.0 | 0.0 | 0.0 | 7.0 |
| asimina\_triloba | 0.0 | 0.0 | 0.0 | 10.0 |
| cercis\_canadensis | 0.0 | 0.0 | 0.0 | 9.0 |
| chionanthus\_virginicus | 0.0 | 0.0 | 0.0 | 9.0 |
| gleditsia\_triacanthos | 0.125 | 0.125 | 0.125 | 8.0 |
| ilex\_opaca | 0.0 | 0.0 | 0.0 | 11.0 |
| liriodendron\_tulipifera | 0.1 | 0.1 | 0.1 | 10.0 |
| ostrya\_virginiana | 0.1 | 0.1 | 0.1 | 10.0 |
| prunus\_sargentii | 0.167 | 0.167 | 0.167 | 6.0 |
| ptelea\_trifoliata | 0.0 | 0.0 | 0.0 | 11.0 |
| quercus\_montana | 0.0 | 0.0 | 0.0 | 10.0 |
| styrax\_japonica | 0.0 | 0.0 | 0.0 | 6.0 |
| ulmus\_pumila | 0.0 | 0.0 | 0.0 | 6.0 |
| macro avg | 0.051 | 0.051 | 0.051 | 129.0 |
| weighted avg | 0.047 | 0.047 | 0.047 | 129.0 |

The confusion matrix of test 4.6 is comparable to 4.5, clearly suggesting poor predictive success across all classes. One important detail which has yet to be stated in regards to the underwhelming classification reports and confusion matrices seen so far is that the test set sizes have always been very small. With minimal input in testing, poor performance appears more obviously within these metrics. Although unlikely to drastically improve the outcomes of these metrics, the utilisation of larger test sets will at least increase the amount of testing done by the model, giving it more of a chance to prove its predictive ability.

A screenshot of a crossword puzzle

Description automatically generated

Figure Confusion Matrix 4.6

## 

## 4.7

Although substantial improvement has not yet been made regarding the classification reports or confusion matrices, highly promising accuracy and loss results have been attained during recent tests.

In order to improve the underwhelming resultant metrics of versions 4.5 and 4.6, one suggestion was made at the end of 4.6, that being to increase the input data volume. In order to do this, the code of [4.7](https://github.com/yungroms/y3_proj/blob/26c900e22d5d1cdba91a58fecca52322fa47bf12/4.7%20(6.14)/4.7.py) will implement the [*shrooms\_ds\_max\_split*](https://www.kaggle.com/datasets/yungroms/shrooms-ds-max-split) dataset, in order to test if the larger input dataset could positively impact its corresponding classification report and confusion matrix.

In the tests of previous versions that utilised this larger dataset, the number of epochs was increased correspondingly, to ensure the model had sufficient time to maximise its accuracy. The same approach is taken for test 4.7, which will utilise 20 epochs for training and validation, as opposed to other recent versions which utilised 10 epochs. In relation to this, a learning rate scheduler has also been implemented, to ensure that the model does not become susceptible to overfitting.

A computer screen shot of white text

Description automatically generated

Figure Learning Rate Scheduler 4.7

Based specifically on the graph plots of 4.5, which demonstrated a slight decline in performance between the penultimate and final epochs, a second improvement was suggested: to implement early stopping. The aim of this implementation is to halt the learning process if the validation loss score declines for a specified number of epochs, then restore the architecture’s weight values to those of the best-performing epoch.

The final new inclusion within version 4.7 is the re-introduction of data augmentation, which aims to perceivably expand and diversify the input dataset, hopefully leading to a more robust and learned model.

A screen shot of a computer program

Description automatically generated

Figure Data Augmentation 4.7

The last few pairs of phase 4 versions sought to compare the performance of two pre-trained architectures, *EfficientNetV2S* and *EfficientNetB7*. Based on the test results of these versions, the decision has been made to utilise only one of these architectures moving forward, that being *EfficientNetV2S*.

## 4.7 Results

Below is the training and validation results table containing the accuracy and loss values for each epoch. Although the number of epochs was initially set at 20, it appears as though the newly implemented early stopping took effect towards the end, as epoch 19 is the final epoch recorded within the table.

The results contained within this table are very strong: The values of the initial epoch are decent, but vast improvement was made within each column as the epochs progressed, producing very promising results by the 3rd and 4th epochs which continued until the end of the training process.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| epoch | train\_loss | train\_accuracy | val\_loss | val\_accuracy |
| 1 | 0.9144284725189209 | 0.6934955716133118 | 0.4609358012676239 | 0.8450018763542175 |
| 2 | 0.45768609642982483 | 0.8415535688400269 | 0.32750093936920166 | 0.8914264440536499 |
| 3 | 0.3157256841659546 | 0.8904070854187012 | 0.23558269441127777 | 0.9206289649009705 |
| 4 | 0.2528485655784607 | 0.9136171936988831 | 0.20826363563537598 | 0.9344814419746399 |
| 5 | 0.20632946491241455 | 0.9299017190933228 | 0.2016143649816513 | 0.9385997653007507 |
| 6 | 0.15220916271209717 | 0.9503041505813599 | 0.17110732197761536 | 0.9483339786529541 |
| 7 | 0.13130581378936768 | 0.957229733467102 | 0.16023558378219604 | 0.9539498090744019 |
| 8 | 0.11799157410860062 | 0.9607861638069153 | 0.1519443690776825 | 0.956570565700531 |
| 9 | 0.1122802272439003 | 0.9625643491744995 | 0.14747922122478485 | 0.9580681324005127 |
| 10 | 0.1112806424498558 | 0.9639681577682495 | 0.1466628760099411 | 0.9595656991004944 |
| 11 | 0.1100851520895958 | 0.9645296931266785 | 0.14535628259181976 | 0.9588169455528259 |
| 12 | 0.09937489032745361 | 0.9672437906265259 | 0.14668355882167816 | 0.9576937556266785 |
| 13 | 0.10316438972949982 | 0.9675245881080627 | 0.14409106969833374 | 0.9588169455528259 |
| 14 | 0.09852111339569092 | 0.9702386260032654 | 0.14611268043518066 | 0.9573193788528442 |
| 15 | 0.10268872231245041 | 0.9683668613433838 | 0.1454615294933319 | 0.9558218121528625 |
| 16 | 0.09869015216827393 | 0.9692091941833496 | 0.14352281391620636 | 0.9595656991004944 |
| 17 | 0.10429789125919342 | 0.9648104906082153 | 0.14844660460948944 | 0.9576937556266785 |
| 18 | 0.108112633228302 | 0.9653720259666443 | 0.1457192599773407 | 0.9580681324005127 |
| 19 | 0.10222280770540237 | 0.9676181674003601 | 0.1446332037448883 | 0.9573193788528442 |

In fact, the loss and accuracy values achieved during 4.7’s testing are the best results attained by any version implementing a pre-trained architecture and mushroom dataset. The loss value achieved is 0.18, while the accuracy is around 95%. While these values are not quite as impressive as those achieved while using a variation of the *LeafSnap* dataset, it is worth remembering that the mushroom datasets contain realistic photographs in natural settings, while the *LeafSnap* datasets contain cleaner, less user-realistic images. Based on this fact, the accuracy of around 95% is an amazing result, as is the test loss of 0.18, indicating great promise for future deployment within an app for users.

|  |  |
| --- | --- |
| test\_loss | 0.18072621524333954 |
| test\_accuracy | 0.945147693157196 |

Again, the accuracy and loss graphs provide a visual accompaniment to the above tables, clearly demonstrating the successful training and validation results.

A graph of a graph

AI-generated content may be incorrect.

Figure Accuracy & Loss 4.7

Unfortunately, it appears as though all implementations made within 4.7 did not have as substantial an impact in the classification report as was hoped for. Although this classification report is better than those generated during 4.5 and 4.6, containing higher values across all metric columns, these values are still far lower than what would be considered successful. The larger input dataset and the inclusion of augmentation has had the desired impact, and so the secret of aligning the classification report with the success of the accuracy and loss results has not yet been discovered.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Class | Precision | Recall | F1-Score | Support |
| Agaricus | 0.0759493670886076 | 0.0759493670886076 | 0.0759493670886076 | 79.0 |
| Amanita | 0.05128205128205128 | 0.05063291139240506 | 0.050955414012738856 | 79.0 |
| Boletus | 0.1125 | 0.11392405063291139 | 0.11320754716981132 | 79.0 |
| Cortinarius | 0.1038961038961039 | 0.10126582278481013 | 0.10256410256410256 | 79.0 |
| Entoloma | 0.14285714285714285 | 0.13924050632911392 | 0.14102564102564102 | 79.0 |
| Hygrocybe | 0.11392405063291139 | 0.11392405063291139 | 0.11392405063291139 | 79.0 |
| Lactarius | 0.1038961038961039 | 0.10126582278481013 | 0.10256410256410256 | 79.0 |
| Russula | 0.1511627906976744 | 0.16455696202531644 | 0.15757575757575756 | 79.0 |
| Suillus | 0.07692307692307693 | 0.0759493670886076 | 0.07643312101910828 | 79.0 |
| macro avg | 0.10359896525263025 | 0.1040787623066104 | 0.10379990040586456 | 711.0 |
| weighted avg | 0.10359896525263027 | 0.10407876230661041 | 0.10379990040586456 | 711.0 |

The confusion matrix below reflects the outcome of the classification report, as even though the matrix is populated with higher values than its recent predecessors, these values do not align desirably into the diagonal line of true positives. Instead, they are distributed across the entire matrix, indicating that the separation between the two different collections of results (accuracy/loss and classification/confusion matrix) is as prominent as ever.

A blue and white grid with white text

Description automatically generated

Figure Confusion Matrix 4.7

With the conclusion of phase 4, the project reaches an important point. The brilliant accuracy and loss values obtained during versions 4.6 and 4.7 suggest that it is time to move forward to the development of the application which will house the models produced by these versions. However, the comparably poorer classification reports and confusion matrices generated by these same versions indicate that further work is required, to improve these results to necessary standards.

Due to the fact that the final submission deadline is rapidly approaching, it is necessary that both tasks are carried out simultaneously. This is not a huge issue, as these tasks are mutually exclusive from each other, however emphasis must be placed on the development of the application, so that a fully-implemented version can be produced to demonstrate the abilities of the models which have been painstakingly developed throughout the project’s span.

Therefore, the next steps of the project will be to explore the development of an application prototype, while attempting to solve the discrepancies in the performance metrics.

Implementation & Testing – Model Development Phase 5

Following the inability to clarify the contradictory outcomes of the last few tests in phase 4, a comprehensive review of the project was performed, with particular focus given to phases 3 and 4.

Phase 3, which introduced pre-trained architectures to the project, produced outstanding results when utilizing the *MobileNetV2* architecture alongside the *LeafSnap\_15\_Lab* dataset in 3.1, which obtained a test accuracy of 99.6% and a test loss of 0.05, as well as a confident confusion matrix. Later tests of this phase also produced promising results as the *shrooms\_ds* dataset was implemented in 3.2, then balanced and expanded in 3.3, the latter of which obtained a test accuracy of 76.3% and a test loss of 0.68. Test 3.4 then introduced an alternative pre-trained model, *EfficientNetB0*, but unfortunately generated poor test results of 13.3% accuracy and 2.20 loss, attributed to a failure to optimize the code to the new architecture prior to training.

Although the final results of phase 3 weren’t promising, optimism was still held for the use of *EfficientNet* architectures within the project, and so phase 4 was dedicated to this completing this objective. Two variations of the architectural family were implemented, *EfficientNetB0* and *EfficientNetV2S*, and throughout phase 4 the performance of these architectures was compared to determine which was optimal. From the very first test of this phase, 4.1, great potential could be seen in the use of these pre-trained models, as the test results produced were an accuracy of 82.1% and a loss of 0.57. Furthermore, the test results of this phase only improved, as between 4.3 and 4.6 all test accuracies values were at least 99%, and the final test 4.7 produced an accuracy of 94.5% and a loss of 0.18 – truly great results.

That being said, not all the output data was promising, as difficulty was experienced in generating optimistic confusion matrices during the entirety of phase 4. Even in tests where the accuracy and loss values neared perfection, the accompanying confusion matrices told a contradictory and opposing story. Further confirmation of performance issues was found when classification reports were introduced, as they supported the confusion matrices in opposing the narrative of the great accuracy and loss results of the various tests of phase 4. Even when measures were taken to minimise the discrepancy between the performance metrics, such as balancing the datasets, applying class weights, adjusting hyperparameters and returning to a simpler dataset, no substantial progress was made. Although test 4.7 produced the brilliant accuracy and loss results mentioned above, its confusion matrix and classification report failed to align themselves with this narrative of success, remaining in contradiction weighted average values of 0.103, 0.104, and 0.103 for precision, recall and F1-Score respectively.

When broadly comparing the results of phases 3 and 4, it is clear that great accuracy and loss values can be found throughout both. However, it must be stated that phase 3 displays more immediate potential, as its confusion matrices are far more congruent with their respective accuracies and losses than those of phase 4. While substantial efforts were made to solve the issue of incongruency within the latter phase, progress was negligible.

Therefore, in conclusion of this review, the decision has been made to continue the model development process based on phase 3 instead of phase 4. Ultimately, this means abandoning the *EfficientNet* family of architectures in favour of *MobileNetV2*, and utilising the code of phase 3 as the foundation on which the project’s final outcome can be reached.

In addition to the regression to an earlier code for model development, phase 5 will utilise a new dataset, *17flowers*, which contains 17 classes of different flower species. This decision was made based upon the fact that the dataset of leaves has already been discarded due to its lab-based images being too pristine, in addition to the fact that the mushroom dataset has not produced results which combine fantastic accuracy and congruent supporting performance metrics. These issues left a window of opportunity for this new dataset, which features realistic, real-world images similar to the mushroom dataset, but will hopefully produce better and more congruent outcomes.

## 5.1

As stated above, the code responsible for model development during phase 5 is actually based upon that used during phase 3.While there is a lot of fundamental overlap between the codes used during phases 3 and 4, the biggest differences are the exclusion of the classification report, which was only introduced during phase 4, as well as the use of a different pre-trained architecture, as phase 3 utilised a variety of the *MobileNet* architecture, while phase 4 explored the *EfficientNet* family. At this current moment, the plan is to stick to *MobileNetV2*, the architecture which demonstrated great promise during the tests in which it was implemented. However, the additional performance metrics from phase 4 will be introduced during this current phase, as they are crucial tools in determining the efficacy of the models.

5.1 will consist of a series of tests, in order to allow for the optimal configuration of the hyperparameter values. The first test within this series, 5.1a, is discussed below.

## 5.1a

Since test 3.1 produced the best results of its phase, its code was chosen as the base for developing the model in test 5.1. The hyperparameter values remained identical, as the values used were a learning rate of 0.0001, a batch size of 32, and an epoch value of 10. The only real difference between 3.1 and 5.1 is the dataset used for training, as the *Leafsnap\_15\_Lab* is replaced by *flowers17*.

A black screen with white text

AI-generated content may be incorrect.

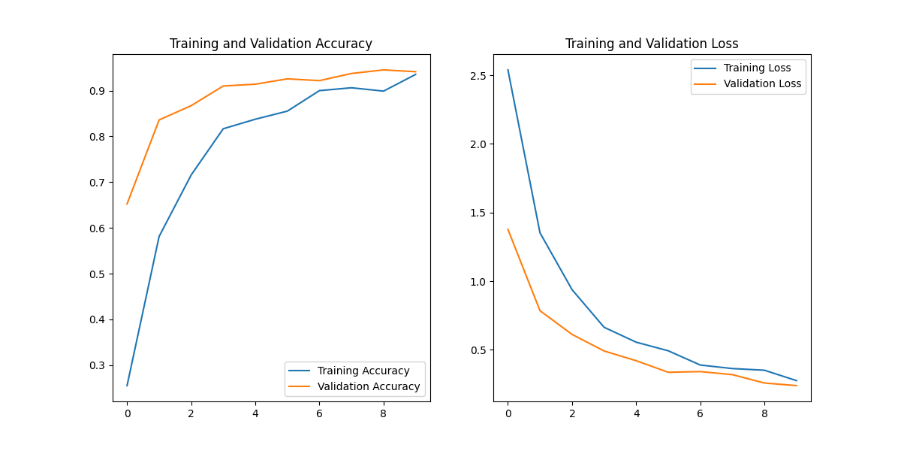
As can be seen in the training results below, this initial version of phase 5 appears to have generated highly promising results. Training accuracy rises continually, from 25.5% initially to 93.5% by the final epoch, while the training loss improves from 2.54 to 0.27. The validation results provide further confirmation of the training process with the final few epochs reaching an average accuracy of around 94% and a final validation loss value of 0.24.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Epoch | accuracy | loss | val\_accuracy | val\_loss |
| 1 | 0.2552083432674408 | 2.539233684539795 | 0.65234375 | 1.3769510984420776 |
| 2 | 0.581250011920929 | 1.352041482925415 | 0.8359375 | 0.7845145463943481 |
| 3 | 0.715624988079071 | 0.9379004836082458 | 0.8671875 | 0.6117345690727234 |
| 4 | 0.8166666626930237 | 0.6635771989822388 | 0.91015625 | 0.49095413088798523 |
| 5 | 0.8374999761581421 | 0.5551220774650574 | 0.9140625 | 0.4199945032596588 |
| 6 | 0.8552083373069763 | 0.4924551546573639 | 0.92578125 | 0.33591848611831665 |
| 7 | 0.8999999761581421 | 0.38867369294166565 | 0.921875 | 0.34080207347869873 |
| 8 | 0.90625 | 0.3629199266433716 | 0.9375 | 0.31862807273864746 |
| 9 | 0.8989583253860474 | 0.35086092352867126 | 0.9453125 | 0.25754785537719727 |
| 10 | 0.9354166388511658 | 0.2749722898006439 | 0.94140625 | 0.23866546154022217 |

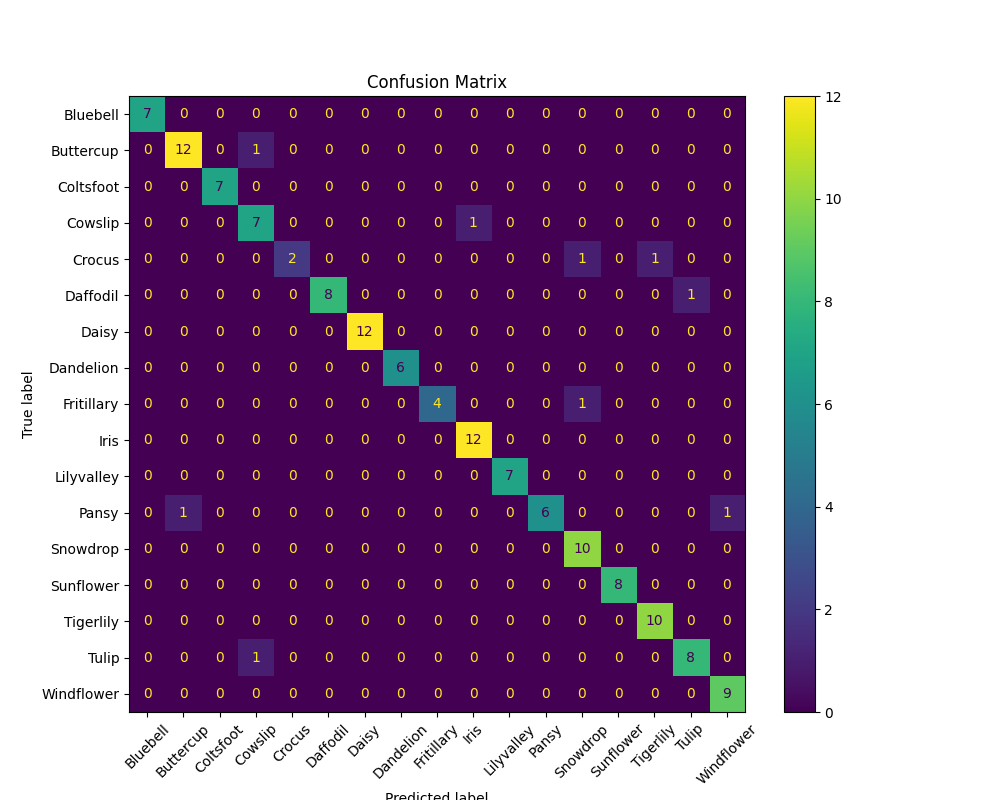
The success of the training stage is reinforced by equally fantastic test results, as an accuracy of 96.5% and a loss of 0.24 were achieved.

|  |  |
| --- | --- |
| Test Loss | 0.2416735291481018 |
| Test Accuracy | 0.9652777910232544 |

The close proximity of the training, validation and test result values suggest a confident and successful learning process, with minimal indication of overfitting. The graphs below display the accuracy and loss values for both the training and validation stages, and provide a visual demonstration of the success of these stages. One point worth noting here is that the plots of both the accuracy and loss graphs do not plateau within the allotted number of epochs. This suggests that increasing this value could allow the model to increase its performance further. This will be taken into account during the next test.



Finally, the confusion matrix for 5.1a validates its test results, as the desirable diagonal line of true positives is prominent, suggesting that only a handful of predicted labels were incorrect. Given the extreme difficulty experienced in attempting to align confusion matrices with its accompanying results during phase 4, a supportive confusion matrix such as this one for 5.1a is a welcome sight.



## 5.1b

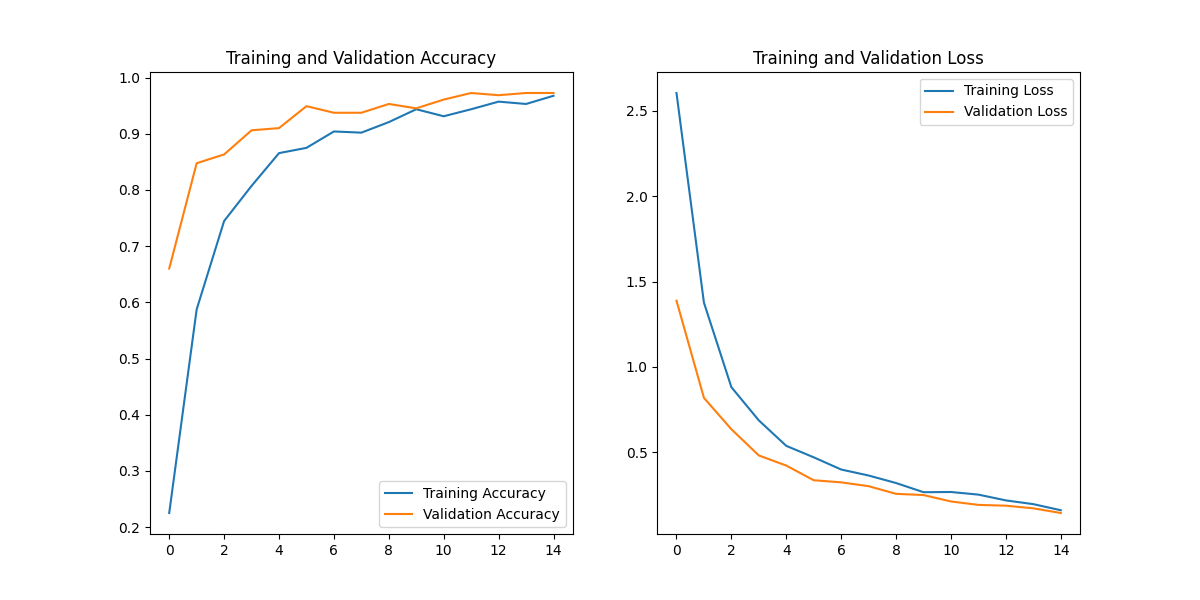
Based on the result of the 5.1a, in which the accuracy and loss plots did not yet reach a plateau, this next test aims to explore the possibility of achieving better results by simply increasing the number of epochs. In 5.1b, this value will be increased from 10 to 15, while all other hyperparameters and variables will be kept the same.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Epoch | accuracy | loss | val\_accuracy | val\_loss |
| 1 | 0.22499999403953552 | 2.605189085006714 | 0.66015625 | 1.387690782546997 |
| 2 | 0.5874999761581421 | 1.3761347532272339 | 0.84765625 | 0.8179001212120056 |
| 3 | 0.7447916865348816 | 0.8805817365646362 | 0.86328125 | 0.6340733766555786 |
| 4 | 0.8072916865348816 | 0.68577641248703 | 0.90625 | 0.4802386462688446 |
| 5 | 0.8656250238418579 | 0.536548912525177 | 0.91015625 | 0.4210643768310547 |
| 6 | 0.875 | 0.46937209367752075 | 0.94921875 | 0.33501434326171875 |
| 7 | 0.9041666388511658 | 0.39783230423927307 | 0.9375 | 0.3226008415222168 |
| 8 | 0.9020833373069763 | 0.36271336674690247 | 0.9375 | 0.30015283823013306 |
| 9 | 0.9208333492279053 | 0.31864750385284424 | 0.953125 | 0.255307674407959 |
| 10 | 0.9437500238418579 | 0.26489657163619995 | 0.9453125 | 0.2481955885887146 |
| 11 | 0.9312499761581421 | 0.2660682499408722 | 0.9609375 | 0.21062098443508148 |
| 12 | 0.9437500238418579 | 0.2507191002368927 | 0.97265625 | 0.19044290482997894 |
| 13 | 0.9572916626930237 | 0.21686074137687683 | 0.96875 | 0.1858711689710617 |
| 14 | 0.953125 | 0.19487789273262024 | 0.97265625 | 0.17034241557121277 |
| 15 | 0.9677083492279053 | 0.1592138260602951 | 0.97265625 | 0.14297834038734436 |

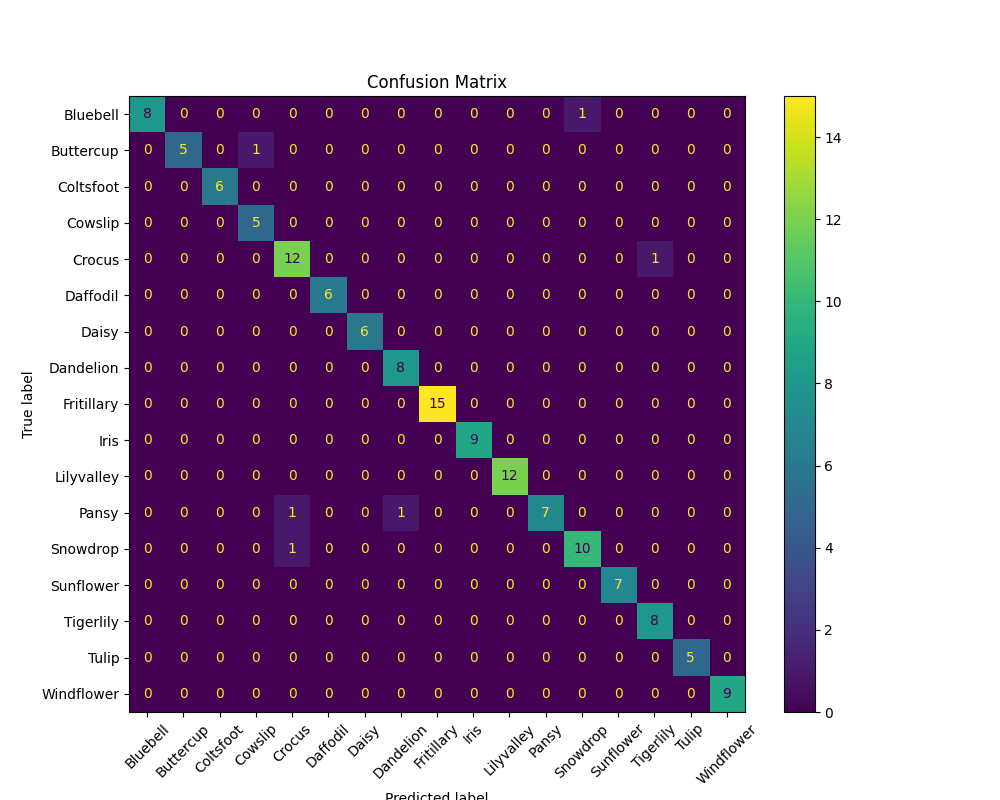
As can be seen in the training and validation results table above, it appears that the 5 additional epochs provided the model with the time to improve both its accuracy and loss values beyond those achieved during 5.1a. The training and validation accuracy values average around 97%, an improvement of around 3%, while the average value of the training and validation loss values is around 0.15, showing an improvement of at least 0.10 in comparison to 5.1a.

|  |  |
| --- | --- |
| Test Loss | 0.26249635219573975 |
| Test Accuracy | 0.9375 |

Unfortunately, this improvement did not permeate its way into the test results, as for some reason the test results of 5.1b are actually worse than those of 5.1a, even though the training and validation data showed promise for improvement. The test loss of 5.1b is 0.26, which is only 0.02 worse than the equivalent value of 5.1a, however the test accuracy of 5.1b is 93.8%, which is around 3% worse than its predecessor. Initially it was speculated that the test results of 5.1a and 5.1b had been mislabelled, but after verifying that no error was made regarding results storage, the drop in performance between training and testing is simply confusing. One possible explanation would be that the images designated to the test stage were more challenging for the model in 5.1b than in 5.1a. Aside from this potential reason, it is not clear at this point what else could cause the incongruency between the training & validation and the test results.



In more optimistic news, the accuracy and loss plots from the training and validation stages suggest less overfitting than those of 5.1a, as there is less distance between the training and validation plots while superior values are reached. Additionally, even though the number of epochs was extended from 5.1a to 5.1b, the plots fail to demonstrate plateauing, suggesting that results could again be improved further by simply increasing the number of epochs.



Interestingly, although the test results of 5.1b were inexplicably worse than 5.1a, the improvement seen in its training and validation stages is reflected in the confusion matrix which only contains 6 incorrect predictions, marginally better than the 9 incorrect labels of 5.1a’s confusion matrix.

## 5.1c

As was stated in the analysis of 5.1b, its accuracy and loss plots do not reach a point of plateau before completing the final epoch, suggesting that increasing this value further could give the model the opportunity to produce even better results. Therefore, 5.1c will include an additional 5 epochs for its training stage, making the number of epochs 20. Additionally, to ensure a more gradual learning process, the learning rate has been decreased to 0.00005.

The table below contains the accuracy and loss results of the training and validation stages for 5.1c. Interestingly, although the learning rate was halved between tests, the rate of improvement for all metrics in 5.1c is only marginally less than the 5.1b. Throughout the entire table, the values obtained for each epoch are almost identical in comparison to 5.1b.

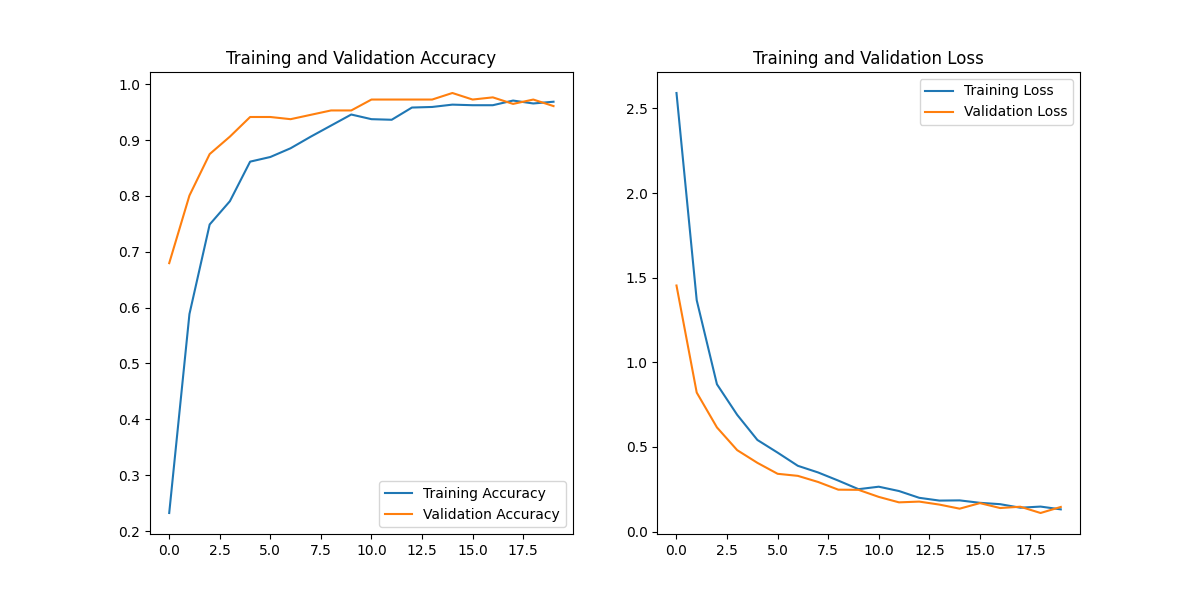
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Epoch | accuracy | loss | val\_accuracy | val\_loss |
| 1 | 0.23229166865348816 | 2.591217041015625 | 0.6796875 | 1.453842043876648 |
| 2 | 0.5885416865348816 | 1.3670449256896973 | 0.80078125 | 0.8213101029396057 |
| 3 | 0.7489583492279053 | 0.8703815340995789 | 0.875 | 0.6153343915939331 |
| 4 | 0.7906249761581421 | 0.6898549199104309 | 0.90625 | 0.4811309576034546 |
| 5 | 0.8614583611488342 | 0.5412112474441528 | 0.94140625 | 0.4060458540916443 |
| 6 | 0.8697916865348816 | 0.46641620993614197 | 0.94140625 | 0.34133467078208923 |
| 7 | 0.8854166865348816 | 0.38851770758628845 | 0.9375 | 0.3291257321834564 |
| 8 | 0.90625 | 0.34926652908325195 | 0.9453125 | 0.29308176040649414 |
| 9 | 0.9260416626930237 | 0.30114060640335083 | 0.953125 | 0.2475132793188095 |
| 10 | 0.9458333253860474 | 0.2504172623157501 | 0.953125 | 0.2465241402387619 |
| 11 | 0.9375 | 0.2651033401489258 | 0.97265625 | 0.20486091077327728 |
| 12 | 0.9364583492279053 | 0.23950694501399994 | 0.97265625 | 0.17259147763252258 |
| 13 | 0.9583333134651184 | 0.19945861399173737 | 0.97265625 | 0.1770787537097931 |
| 14 | 0.9593750238418579 | 0.18303419649600983 | 0.97265625 | 0.1593366414308548 |
| 15 | 0.9635416865348816 | 0.1842469722032547 | 0.984375 | 0.13526296615600586 |
| 16 | 0.9624999761581421 | 0.1698601096868515 | 0.97265625 | 0.16807278990745544 |
| 17 | 0.9624999761581421 | 0.16184809803962708 | 0.9765625 | 0.13906119763851166 |
| 18 | 0.9708333611488342 | 0.140947625041008 | 0.96484375 | 0.14675182104110718 |
| 19 | 0.965624988079071 | 0.14761771261692047 | 0.97265625 | 0.10969216376543045 |
| 20 | 0.96875 | 0.13121174275875092 | 0.9609375 | 0.145465686917305 |

Another interesting point is that increasing the number of epochs from 15 to 20 did not have substantial positive impact on the model’s performance during the training and validation stages: slight improvement continues to occur regarding the training accuracy and loss, however the best validation accuracy value, 98.4%, was produced during epoch 15. As for validation loss, the best value of 0.11 was produced during epoch 19.

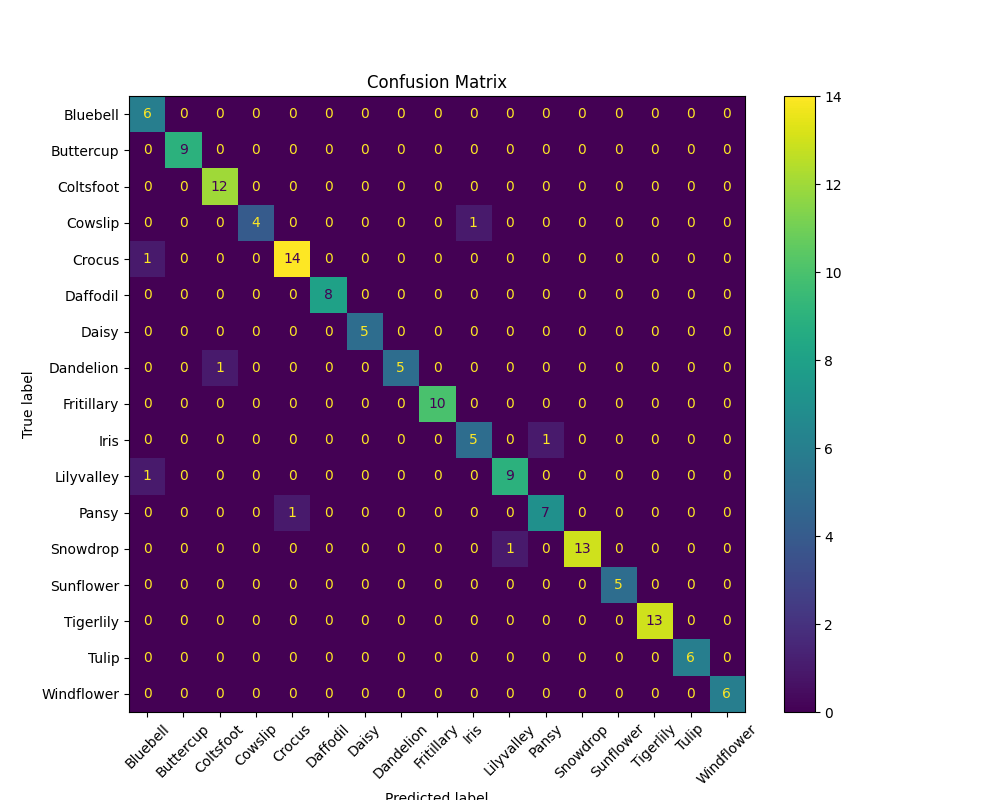
Although the training and validation data does not seem to have improved substantially, the test results of 5.1c provide confirmation that these new hyperparameter values had a positive impact on the model’s performance, as a test accuracy of 97.2% and a test loss of 0.14 were recorded. These are actually the best test results of phase 5 so far.

|  |  |
| --- | --- |
| Test Loss | 0.14287321269512177 |
| Test Accuracy | 0.9722222089767456 |

The accuracy and loss graphs visually explain the outcome of 5.1c, and the possible benefits of its hyperparameters: firstly, the increased number of epochs allows the plots to finally display plateauing, while the reduced learning rate could help explain the closer proximity of the training and validation plots, suggesting an improved learning process.



Once again, this test has produced a very promising confusion matrix, with only 7 incorrect predictions. This in line with the other tests of phase 5, and in contrast to the very confusing confusion matrices of phase 4.



## 5.1d

Now that minor adjustments to the number of epochs and learning rate have been explored, 5.1d will investigate the impact of changing the batch size, a hyperparameter which has been kept consistent to this point. The new value for the batch size will be 16, which is half the previous value of 32. The decision to reduce the batch size was made based on the fact that the dataset in use, flowers17, is relatively small, and so it seemed logical to explore the use of a smaller batch size, as opposed to a larger one. The learning rate and number of epochs are 0.00005 and 20 respectively.

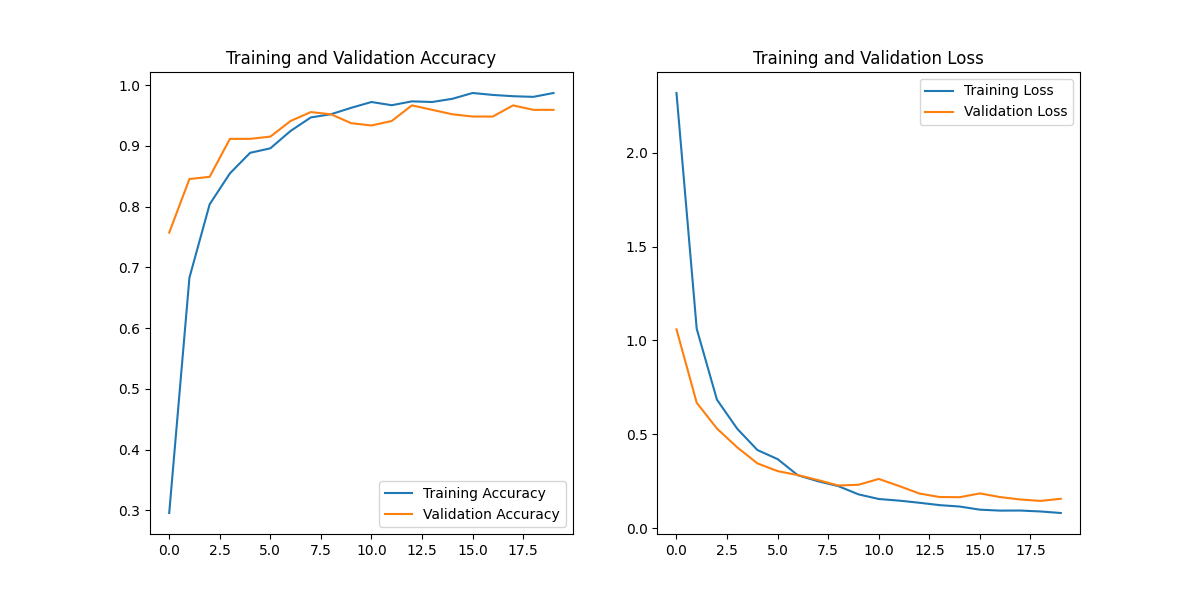
The table below contains the accuracy and loss results from the training and validation stages. It is interesting to note that this is the first time in the recent history of this project that the validation stages were outperformed by their corresponding training stages: typically, the validation accuracy and loss results are superior, however it is the training values which are better in 5.1d. From around the 10th epoch onwards the training data produces the better results, and this dynamic remains until the end, with accuracy and loss values which are respectively 3-5% and 0.06-0.07 better than their validation equivalents throughout.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Epoch | accuracy | loss | val\_accuracy | val\_loss |
| 1 | 0.2955508530139923 | 2.3196723461151123 | 0.7573529481887817 | 1.0593253374099731 |
| 2 | 0.6832627058029175 | 1.0622674226760864 | 0.845588207244873 | 0.667885959148407 |
| 3 | 0.804025411605835 | 0.6844117641448975 | 0.8492646813392639 | 0.5305262207984924 |
| 4 | 0.8548728823661804 | 0.5298187732696533 | 0.9117646813392639 | 0.4303551912307739 |
| 5 | 0.8887711763381958 | 0.41587138175964355 | 0.9117646813392639 | 0.344769686460495 |
| 6 | 0.8961864113807678 | 0.3674969971179962 | 0.9154411554336548 | 0.30353501439094543 |
| 7 | 0.9247881174087524 | 0.28137490153312683 | 0.9411764740943909 | 0.28230857849121094 |
| 8 | 0.9470338821411133 | 0.24932172894477844 | 0.9558823704719543 | 0.2555859088897705 |
| 9 | 0.9523305296897888 | 0.22340647876262665 | 0.9522058963775635 | 0.22667516767978668 |
| 10 | 0.9629237055778503 | 0.17945550382137299 | 0.9375 | 0.23072125017642975 |
| 11 | 0.9724576473236084 | 0.1548006683588028 | 0.9338235259056091 | 0.26181286573410034 |
| 12 | 0.9671609997749329 | 0.14613685011863708 | 0.9411764740943909 | 0.22436897456645966 |
| 13 | 0.9735169410705566 | 0.13485397398471832 | 0.966911792755127 | 0.1842106580734253 |
| 14 | 0.9724576473236084 | 0.12216592580080032 | 0.9595588445663452 | 0.16538400948047638 |
| 15 | 0.9777542352676392 | 0.1146000325679779 | 0.9522058963775635 | 0.16424496471881866 |
| 16 | 0.9872881174087524 | 0.09793656319379807 | 0.9485294222831726 | 0.18462693691253662 |
| 17 | 0.9841101765632629 | 0.09279216080904007 | 0.9485294222831726 | 0.16507244110107422 |
| 18 | 0.9819915294647217 | 0.09327848255634308 | 0.966911792755127 | 0.15214592218399048 |
| 19 | 0.9809321761131287 | 0.0882859155535698 | 0.9595588445663452 | 0.14459627866744995 |
| 20 | 0.9872881174087524 | 0.08013921231031418 | 0.9595588445663452 | 0.15599079430103302 |

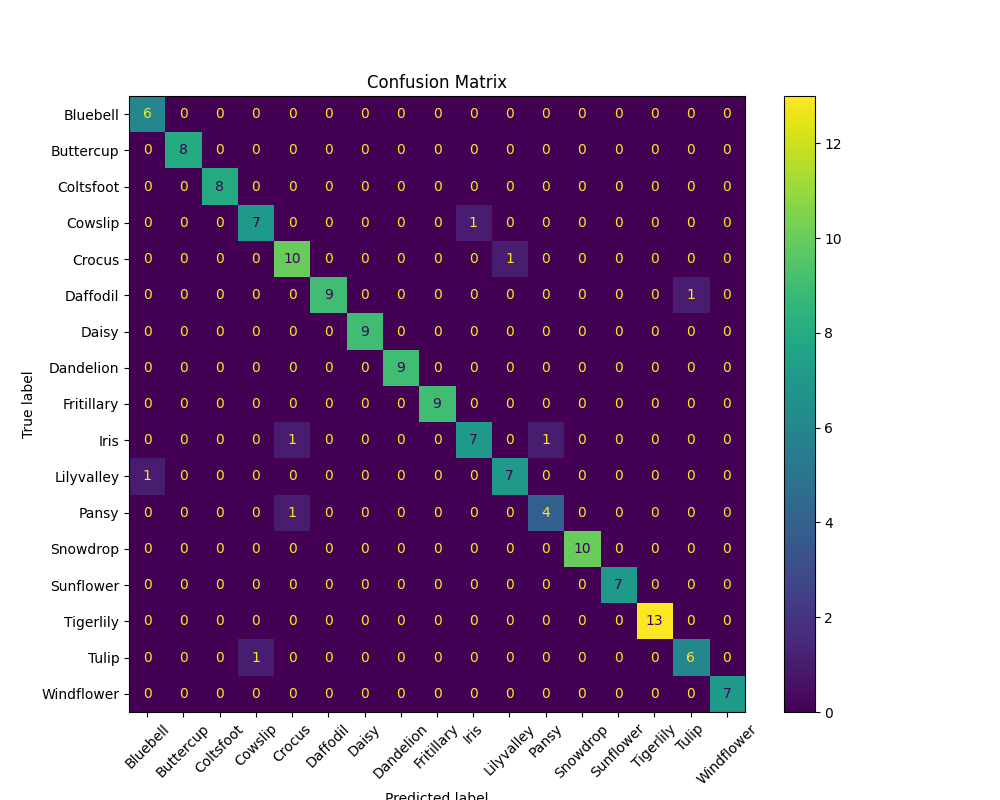
Regardless, all values obtained by the final epoch are very promising: 98.7% and 0.08 for the training accuracy and loss, and 96% and 0.15 for validation. Those training results are the best scores obtained in phase 5 so far. However, when comparing 5.1d to its predecessor, it can be seen that 5.1d’s test accuracy of 96.5% is roughly 0.7% worse than its predecessor, and its test loss of 0.22 is worse by around 0.08.

|  |  |
| --- | --- |
| Test Loss | 0.22222426533699036 |
| Test Accuracy | 0.9652777910232544 |

This interesting combination of superior training and inferior validation and accuracy results is indicative of the fact that the model has overtrained on the training set. This point is reinforced by the separation between the training and validation plots in both the accuracy and validation graphs below.



Even though this overfitting during training has occurred, these results are still very useful and informative. This conclusion is supported by the fact that the confusion matrix produced contains only 8 incorrect predictions. That being said, 5.1d did not overtake its predecessors in terms of performance, and so the batch size moving forward will return to 32.



## 5.2

Now that optimal hyperparameter values have been identified, the next experimentation stage will explore the effect of increasing the input dataset on the performance metrics.

In order to increase the volume of the dataset, a small python program was created which accepts the input dataset as well as a specified factor of multiplication, and applies TensorFlow augmentation methods to the images within each class until they have all been increased to the desired volume. For 5.2, the specified factor of multiplication was 5, and so the new dataset used here has been named *flowers17\_5x*.

INSERT IMAGE OF ORIGINAL MULTIPLIER CODE.

COVER THE HYPERPARAMETER VALUES USED

mobilenet\_LR5e-05\_BS16\_E20

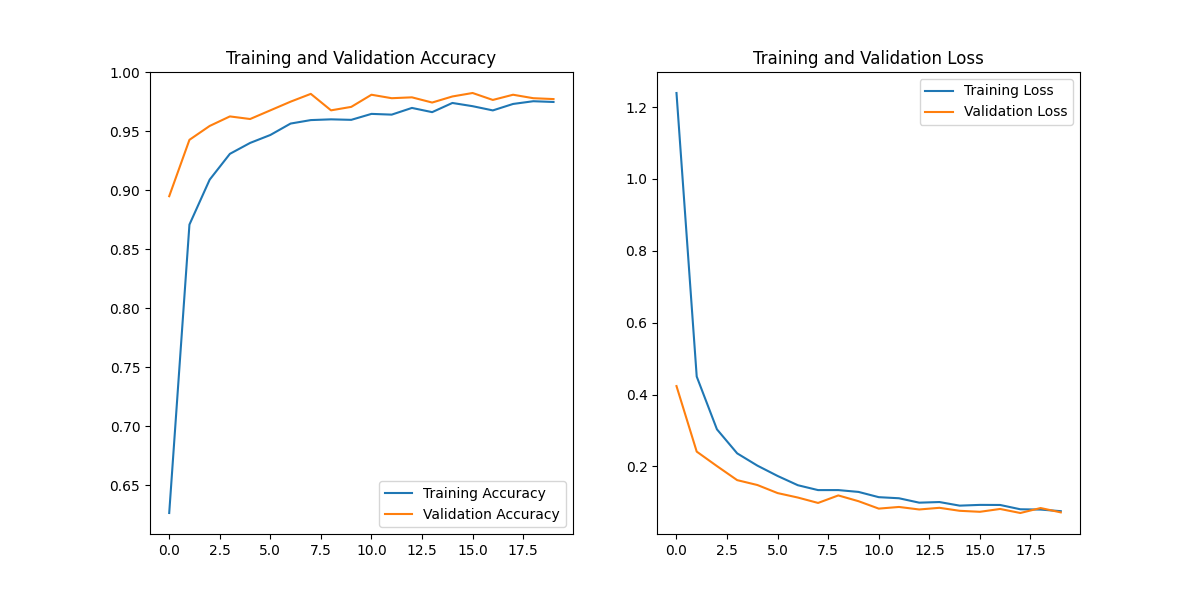
The results table for 5.2 below contains very promising data: the performance metrics obtained during epoch 20 are the best of phase 5 to this point, with an average accuracy of 97.6% and an average loss of 0.073.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Epoch | accuracy | loss | val\_accuracy | val\_loss |
| 1 | 0.6262626051902771 | 1.2390668392181396 | 0.8948529362678528 | 0.4237695038318634 |
| 2 | 0.8707912564277649 | 0.45019522309303284 | 0.9426470398902893 | 0.2414090484380722 |
| 3 | 0.9088804721832275 | 0.3033820688724518 | 0.9544117450714111 | 0.20112556219100952 |
| 4 | 0.9307659864425659 | 0.23659692704677582 | 0.9624999761581421 | 0.1620754897594452 |
| 5 | 0.940025269985199 | 0.20240691304206848 | 0.9602941274642944 | 0.14847072958946228 |
| 6 | 0.9467592835426331 | 0.1736537665128708 | 0.9676470756530762 | 0.12603363394737244 |
| 7 | 0.9564393758773804 | 0.14792826771736145 | 0.9750000238418579 | 0.11362481117248535 |
| 8 | 0.9593855142593384 | 0.13433901965618134 | 0.9816176295280457 | 0.0986107885837555 |
| 9 | 0.9600168466567993 | 0.13438545167446136 | 0.9676470756530762 | 0.11950311064720154 |
| 10 | 0.9595959782600403 | 0.12929010391235352 | 0.970588207244873 | 0.10345669090747833 |
| 11 | 0.9646464586257935 | 0.11466462165117264 | 0.9808823466300964 | 0.08283430337905884 |
| 12 | 0.9640151262283325 | 0.1116393506526947 | 0.9779411554336548 | 0.0874519795179367 |
| 13 | 0.9696969985961914 | 0.09937536716461182 | 0.9786764979362488 | 0.08043872565031052 |
| 14 | 0.9661195278167725 | 0.10111825913190842 | 0.9742646813392639 | 0.0850747600197792 |
| 15 | 0.9739057421684265 | 0.09111125767230988 | 0.979411780834198 | 0.07675392180681229 |
| 16 | 0.9711700081825256 | 0.09318546950817108 | 0.9823529124259949 | 0.07396939396858215 |
| 17 | 0.9675925970077515 | 0.0929829329252243 | 0.9764705896377563 | 0.08187142014503479 |
| 18 | 0.9730639457702637 | 0.08098549395799637 | 0.9808823466300964 | 0.0706314742565155 |
| 19 | 0.9753788113594055 | 0.08018365502357483 | 0.9779411554336548 | 0.08445794135332108 |
| 20 | 0.9747474789619446 | 0.07569650560617447 | 0.9772058725357056 | 0.07234415411949158 |

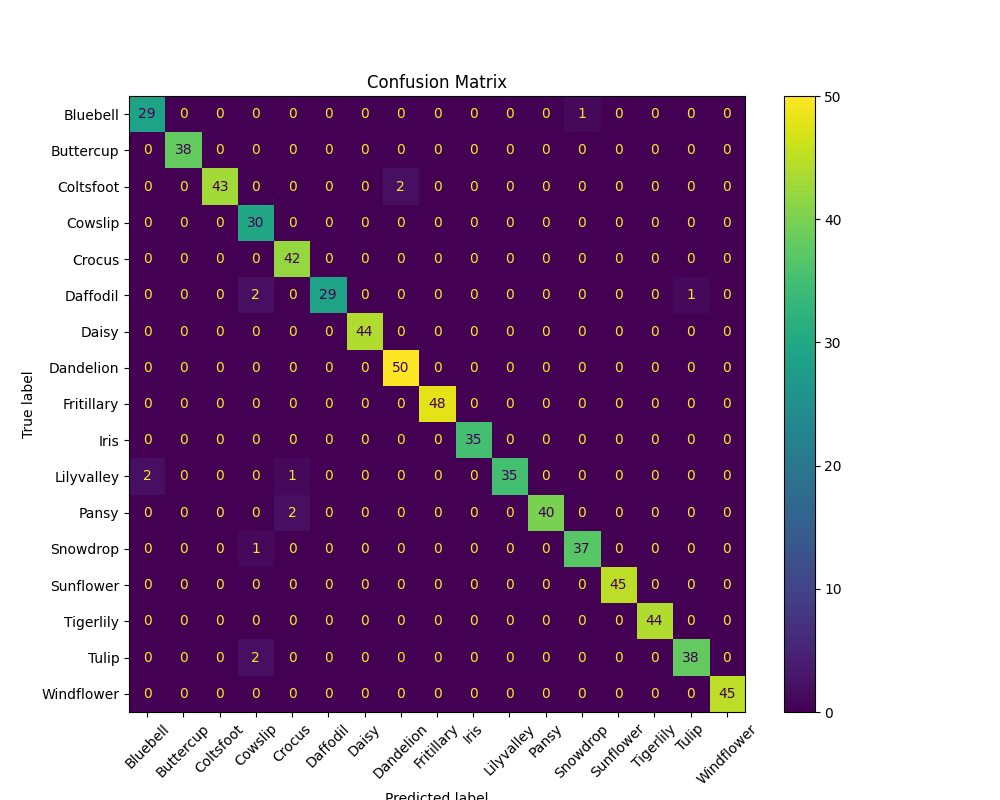
The test results echo those obtained during training and validation, and again are the best test results obtained during phase 5, with the test accuracy being 98.1% and the test loss being 0.08.

|  |  |
| --- | --- |
| Test Loss | 0.08028818666934967 |
| Test Accuracy | 0.9810495376586914 |

Validation of the tremendous results above is also found in the accuracy and loss plots from the training and validation stages. Both are great, however the loss graph is truly outstanding, as the distance between the training and validation plots is miniscule.



The confusion matrix also reinforces the success of utilising the enlarged dataset, as even though the volume of the input dataset was 5 times larger than in previous phase 5 stages, only 14 predictions were incorrect, which as a percentage of the total test images is actually far better than preceding tests.



## 5.3

In order to validate the outstanding results achieved in 5.2, this next stage will re-introduce the classification report. Ideally, the report will confirm that this current configuration is as great as is implied by the performance metrics currently in use.

In order to remain consistent and comparable with the previous stage, 5.3 will include no changes apart from the re-introduction of the classification report.

mobilenet\_LR5e-05\_BS16\_E20

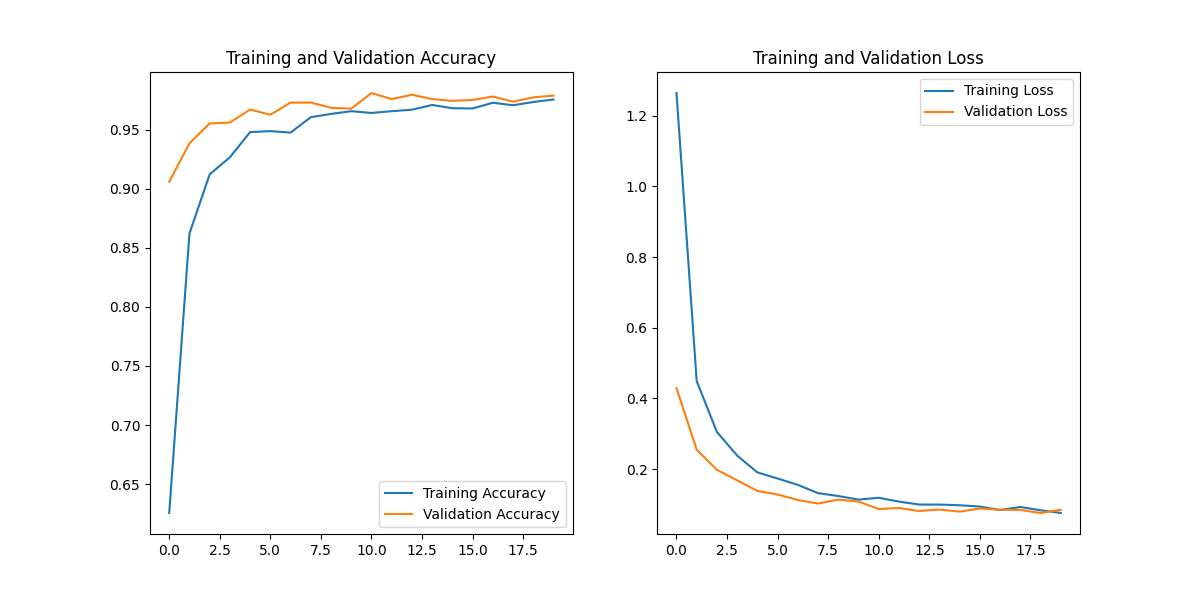
The table below contains the accuracy and loss data from the training and validation stages of 5.3. As seen in 5.2, this current configuration of dataset and hyperparameters has again produced amazing results which are almost identical to those of the previous stage. This was of course expected, since no model-related details were changed.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Epoch | accuracy | loss | val\_accuracy | val\_loss |
| 1 | 0.6256313323974609 | 1.2641345262527466 | 0.9058823585510254 | 0.42930635809898376 |
| 2 | 0.8619528412818909 | 0.44874125719070435 | 0.9382352828979492 | 0.2549041211605072 |
| 3 | 0.9120370149612427 | 0.30498579144477844 | 0.9551470875740051 | 0.1979990154504776 |
| 4 | 0.9265572428703308 | 0.23819653689861298 | 0.9558823704719543 | 0.16820833086967468 |
| 5 | 0.9478114247322083 | 0.19067534804344177 | 0.966911792755127 | 0.1382799744606018 |
| 6 | 0.9486532211303711 | 0.17341232299804688 | 0.9624999761581421 | 0.12811554968357086 |
| 7 | 0.9473905563354492 | 0.15539051592350006 | 0.9727941155433655 | 0.11250313371419907 |
| 8 | 0.9604377150535583 | 0.1319790780544281 | 0.9727941155433655 | 0.1027107685804367 |
| 9 | 0.9631733894348145 | 0.12396258115768433 | 0.9683823585510254 | 0.11364690959453583 |
| 10 | 0.9654881954193115 | 0.11370489001274109 | 0.9676470756530762 | 0.10800516605377197 |
| 11 | 0.9640151262283325 | 0.11889924854040146 | 0.9808823466300964 | 0.08710624277591705 |
| 12 | 0.9654881954193115 | 0.10811132192611694 | 0.9757353067398071 | 0.09001830965280533 |
| 13 | 0.9667508602142334 | 0.09977269172668457 | 0.979411780834198 | 0.08148282021284103 |
| 14 | 0.9707491397857666 | 0.09966247528791428 | 0.9757353067398071 | 0.08560279756784439 |
| 15 | 0.9680134654045105 | 0.09781938046216965 | 0.9742646813392639 | 0.07973917573690414 |
| 16 | 0.9678030014038086 | 0.09409473836421967 | 0.9750000238418579 | 0.08852533251047134 |
| 17 | 0.9726430773735046 | 0.08419913798570633 | 0.9779411554336548 | 0.08499638736248016 |
| 18 | 0.9705387353897095 | 0.09282692521810532 | 0.9735293984413147 | 0.08458393067121506 |
| 19 | 0.9732744097709656 | 0.08357356488704681 | 0.9772058725357056 | 0.0758606418967247 |
| 20 | 0.9753788113594055 | 0.07606006413698196 | 0.9786764979362488 | 0.08495640009641647 |

The test results achieved are also almost identical, with the exact same test accuracy of 98.1%, and a fractionally better test loss of 0.07.

|  |  |
| --- | --- |
| Test Loss | 0.07193555682897568 |
| Test Accuracy | 0.9810495376586914 |

The accuracy and loss plots demonstrate the success of the training stage, with the close proximity of training and validation plots suggesting that overfitting is not occurring.



The confusion matrix of 5.3 is equally as brilliant as 5.2, as they both achieved the exact same percentage of true positives, and only mislabelled 14 test images.



Now for the true purpose of 5.3: examining if the outstanding results above are reflected in the classification report. Given the brilliance of the confusion matrix, it was expected that the classification report would support the optimistic evaluation of the other metrics, and fortunately that is the case. All values within the report are north of 0.90, and the macro and weighted averages at the bottom of the report are over 0.97, meaning that the performance of the model is evidenced in the precision, recall and F1-score values. This classification report reassuringly reflects not only the confusion matrix, but also the overall success of the model development process.

In stark contrast to phase 4, all performance metrics are aligned and unified in providing an optimistic analysis of the models developed during phase 5. Given that near perfect scores have been recorded already, the room for improvement is small, however there are a couple more adjustments to test before concluding the model development process.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Class | Precision | Recall | F1-Score | Support |
| Bluebell | 0.90625 | 0.9666666666666667 | 0.9354838709677419 | 30.0 |
| Buttercup | 0.9743589743589743 | 1.0 | 0.987012987012987 | 38.0 |
| Coltsfoot | 1.0 | 0.9555555555555556 | 0.9772727272727273 | 45.0 |
| Cowslip | 0.9666666666666667 | 0.9666666666666667 | 0.9666666666666667 | 30.0 |
| Crocus | 0.9333333333333333 | 1.0 | 0.9655172413793104 | 42.0 |
| Daffodil | 1.0 | 0.90625 | 0.9508196721311475 | 32.0 |
| Daisy | 1.0 | 1.0 | 1.0 | 44.0 |
| Dandelion | 0.9615384615384616 | 1.0 | 0.9803921568627451 | 50.0 |
| Fritillary | 1.0 | 1.0 | 1.0 | 48.0 |
| Iris | 1.0 | 1.0 | 1.0 | 35.0 |
| Lilyvalley | 1.0 | 0.9210526315789473 | 0.958904109589041 | 38.0 |
| Pansy | 1.0 | 0.9523809523809523 | 0.975609756097561 | 42.0 |
| Snowdrop | 0.9487179487179487 | 0.9736842105263158 | 0.961038961038961 | 38.0 |
| Sunflower | 1.0 | 1.0 | 1.0 | 45.0 |
| Tigerlily | 1.0 | 1.0 | 1.0 | 44.0 |
| Tulip | 0.9512195121951219 | 0.975 | 0.9629629629629629 | 40.0 |
| Windflower | 1.0 | 1.0 | 1.0 | 45.0 |
| macro avg | 0.9789461704006182 | 0.9774856872573591 | 0.9777459477636382 | 686.0 |
| weighted avg | 0.9804520796525525 | 0.9795918367346939 | 0.9795886425809673 | 686.0 |

## 5.4

As the model development process nears its conclusion, there are firstly a handful of adjustments to make in hopes of further increasing model performance. One such adjustment, explored in 5.4, is to reduce the degree of augmentation applied to the dataset during expansion.

LINE OR TWO ABOUT THE DIFFERENCE IN AUGMENTATION BETWEEN 5.4 AND 5.3 (5.6 AND 5.5)

SCREENSHOT OF NEW VALUES

mobilenet\_LR5e-05\_BS16\_E20 CHANGE

The augmentation values applied during previous stages were rather substantial, meaning that the images produced may push the recognition ability of the model too far, causing confusion between classes instead of simply generating additional data for each class. By reducing the degrees of augmentation, the images produced for the enlarged dataset will be of more utility to the model, allowing it to gain a deeper understanding of each class, while hopefully avoiding the confusion caused by acute augmentation.

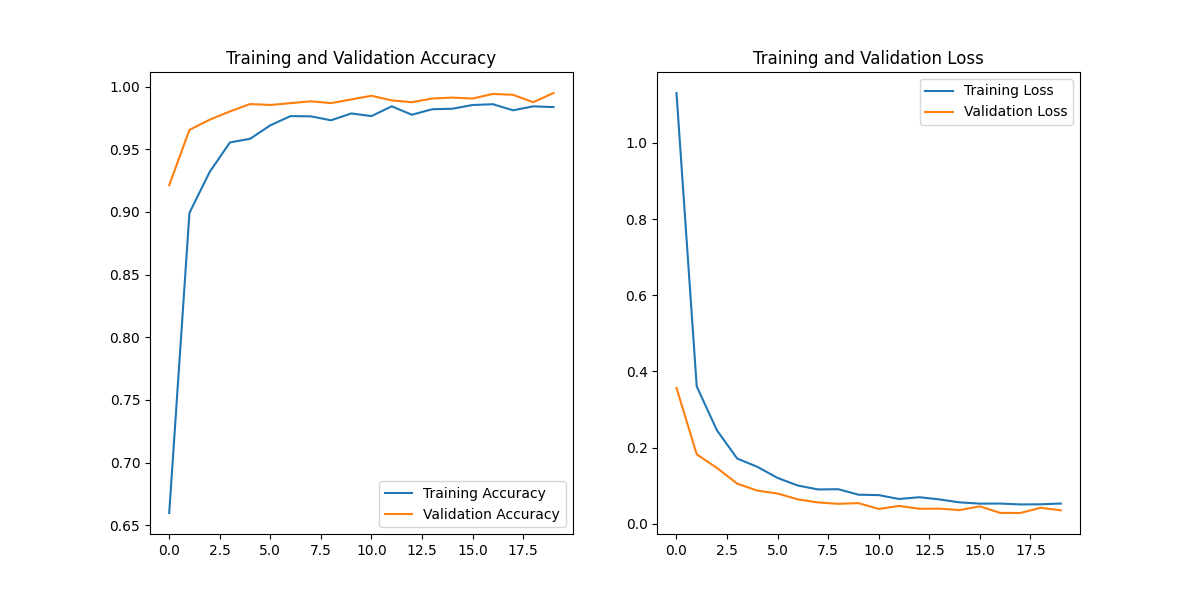
By examining the results table of 5.4 below, it appears as though reducing the augmentation has had the desired effect, as new high scores were obtained across all columns: 98.4%, 0.054, 99.5% and 0.036 for training accuracy, training loss, validation accuracy and validation loss respectively.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Epoch | accuracy | loss | val\_accuracy | val\_loss |
| 1 | 0.6597222089767456 | 1.130545735359192 | 0.9213235378265381 | 0.35696738958358765 |
| 2 | 0.8992003202438354 | 0.36091750860214233 | 0.9654411673545837 | 0.18275301158428192 |
| 3 | 0.9318181872367859 | 0.24568496644496918 | 0.9735293984413147 | 0.14691409468650818 |
| 4 | 0.9553872346878052 | 0.17170915007591248 | 0.9801470637321472 | 0.10592234879732132 |
| 5 | 0.9583333134651184 | 0.1498570442199707 | 0.9860293865203857 | 0.0874500498175621 |
| 6 | 0.9690656661987305 | 0.1207321509718895 | 0.9852941036224365 | 0.07984044402837753 |
| 7 | 0.9764309525489807 | 0.10084906220436096 | 0.9867647290229797 | 0.06459816545248032 |
| 8 | 0.9762205481529236 | 0.09062345325946808 | 0.9882352948188782 | 0.056459397077560425 |
| 9 | 0.9730639457702637 | 0.09119988232851028 | 0.9867647290229797 | 0.05287778377532959 |
| 10 | 0.9785353541374207 | 0.07676930725574493 | 0.9897058606147766 | 0.05467388406395912 |
| 11 | 0.9764309525489807 | 0.07573401927947998 | 0.9926470518112183 | 0.03934750705957413 |
| 12 | 0.9842171669006348 | 0.06555674225091934 | 0.9889705777168274 | 0.04741937294602394 |
| 13 | 0.9774831533432007 | 0.07023613899946213 | 0.987500011920929 | 0.03992653638124466 |
| 14 | 0.9819023609161377 | 0.06439781188964844 | 0.9904412031173706 | 0.04027123004198074 |
| 15 | 0.9823232293128967 | 0.05667911842465401 | 0.9911764860153198 | 0.03640379011631012 |
| 16 | 0.9852693676948547 | 0.05336226522922516 | 0.9904412031173706 | 0.04627865552902222 |
| 17 | 0.9859007000923157 | 0.05362839251756668 | 0.9941176176071167 | 0.029119273647665977 |
| 18 | 0.9810606241226196 | 0.051260173320770264 | 0.9933823347091675 | 0.028798824176192284 |
| 19 | 0.9842171669006348 | 0.05163523182272911 | 0.987500011920929 | 0.04245174303650856 |
| 20 | 0.9835858345031738 | 0.053631141781806946 | 0.9948529601097107 | 0.035872623324394226 |

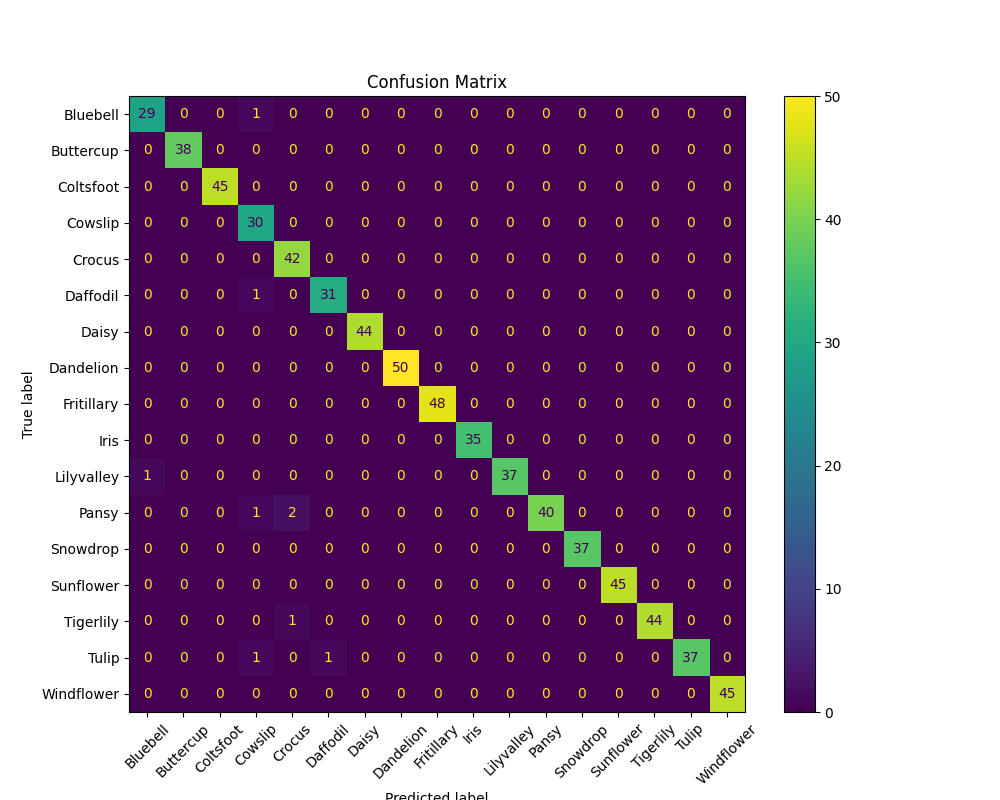
Although minutely less than the training and validation results, the test results of 5.4 still surpassed 5.3, with an improvement of around 0.01 for loss and 0.4% for accuracy. While these improvements are miniscule, they are improvements nevertheless, providing confirmation that slightly reducing the augmentation is a step in the right direction.

|  |  |
| --- | --- |
| Test Loss | 0.06092633306980133 |
| Test Accuracy | 0.9854227304458618 |

The accuracy and loss graphs provide a visual representation of the outstanding performance of the model during the training and validation stages of 5.4:



The confusion matrix provides another visual representation of the brilliance of the model produced during 5.4, as it only mislabelled 9 images. This is 5 images less than 5.3, and another high score in terms of true positive percentage.



Additionally, the minute increase in performance is reflected in the classification report. Although it appears identical to its predecessor, the averages for precision, recall and F1-score are marginally higher than those of 5.3.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Class | Precision | Recall | F1-Score | Support |
| Bluebell | 0.9666666666666667 | 0.9666666666666667 | 0.9666666666666667 | 30.0 |
| Buttercup | 1.0 | 1.0 | 1.0 | 38.0 |
| Coltsfoot | 1.0 | 1.0 | 1.0 | 45.0 |
| Cowslip | 0.8823529411764706 | 1.0 | 0.9375 | 30.0 |
| Crocus | 0.9333333333333333 | 1.0 | 0.9655172413793104 | 42.0 |
| Daffodil | 0.96875 | 0.96875 | 0.96875 | 32.0 |
| Daisy | 1.0 | 1.0 | 1.0 | 44.0 |
| Dandelion | 1.0 | 1.0 | 1.0 | 50.0 |
| Fritillary | 1.0 | 1.0 | 1.0 | 48.0 |
| Iris | 1.0 | 1.0 | 1.0 | 35.0 |
| Lilyvalley | 1.0 | 0.9736842105263158 | 0.9866666666666667 | 38.0 |
| Pansy | 1.0 | 0.9302325581395349 | 0.963855421686747 | 43.0 |
| Snowdrop | 1.0 | 1.0 | 1.0 | 37.0 |
| Sunflower | 1.0 | 1.0 | 1.0 | 45.0 |
| Tigerlily | 1.0 | 0.9777777777777777 | 0.9887640449438202 | 45.0 |
| Tulip | 1.0 | 0.9487179487179487 | 0.9736842105263158 | 39.0 |
| Windflower | 1.0 | 1.0 | 1.0 | 45.0 |
| macro avg | 0.9853589965397923 | 0.9862252448134261 | 0.9853767206982075 | 686.0 |
| weighted avg | 0.9878580003429944 | 0.9868804664723032 | 0.9870027796454705 | 686.0 |

By implementing less intense augmentation in 5.4, improvements were made across all metrics, implying that the adjustment made was a success. Performance data as edged even closer to perfection, however there is still a little room for improvement.

## 5.5

Following the success of increasing the volume of the input dataset from 1x to 5x, stage 5.5 will take this exploration further, by increasing the factor of multiplication to 10. Since the results of 5.4 were marginally better than 5.3, and the lesser degree of augmentation was superior, the degree of augmentation in 5.5 will be the same as its predecessor, the only difference being the aforementioned increased multiplication factor. It is predicted that by providing more high-quality data for training, the model produced will achieve even better results.

mobilenet\_LR1e-05\_BS32\_E30 – HYPERPARAMETER CHAT

In order to compensate for increasing the input dataset, the learning rate has been reduced to 0.00005, and the number of epochs has been increased to 30. This will hopefully allow for a more gradual, thorough and consistent learning process.

After analysing the training results below, it appears that the prediction made above was correct: by providing twice the volume in comparison to the previous stage, the model was able to achieve better results. In the 2nd epoch, a training accuracy of 92.0% was achieved, while from the 4th epoch onwards, the validation accuracy remained north of 99%. The loss data is equally as impressive: from the 6th epoch onwards, the training loss remains below 0.1, while from the 13th epoch onwards, the validation loss was less than 0.02. Remarkably, in the final 6 epochs, the validation loss actually remained below 0.01. Once again, with a new stage, new high scores have been achieved.

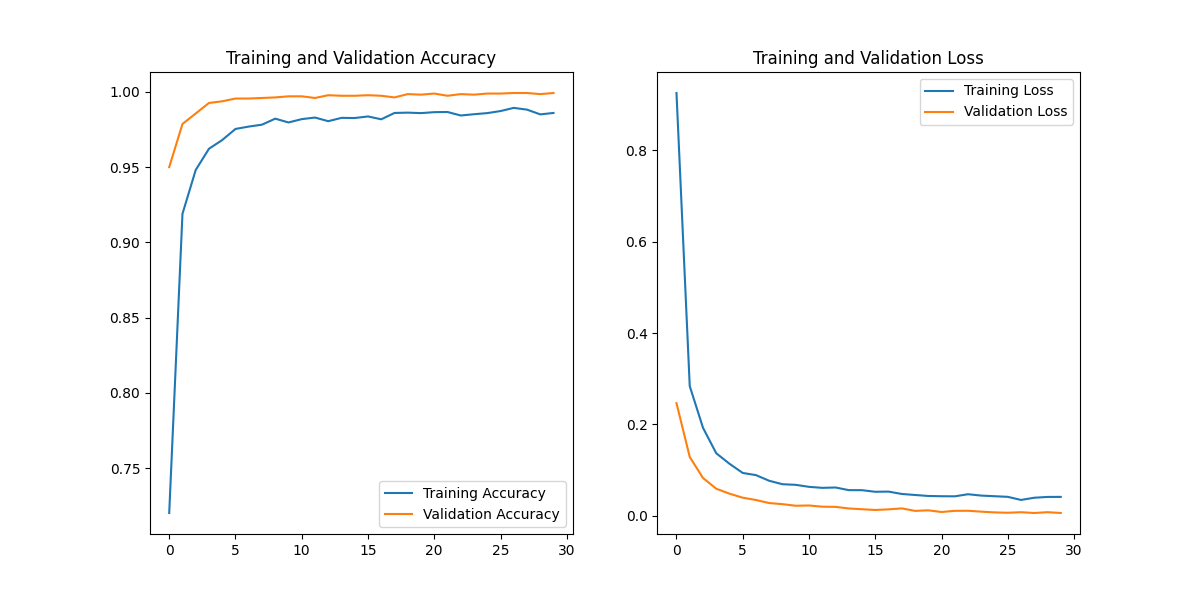
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Epoch | accuracy | loss | val\_accuracy | val\_loss |
| 1 | 0.720223069190979 | 0.9252597689628601 | 0.949999988079071 | 0.24670648574829102 |
| 2 | 0.9189814925193787 | 0.28397542238235474 | 0.9786764979362488 | 0.12890176475048065 |
| 3 | 0.9481270909309387 | 0.19273102283477783 | 0.9856617450714111 | 0.08291604369878769 |
| 4 | 0.9622264504432678 | 0.13700279593467712 | 0.9926470518112183 | 0.059462714940309525 |
| 5 | 0.9680134654045105 | 0.11401835083961487 | 0.9937499761581421 | 0.048569709062576294 |
| 6 | 0.9753788113594055 | 0.09390944242477417 | 0.9955882430076599 | 0.039764150977134705 |
| 7 | 0.9769570827484131 | 0.08915504068136215 | 0.9955882430076599 | 0.03469324856996536 |
| 8 | 0.9782196879386902 | 0.07676397264003754 | 0.9959558844566345 | 0.02802014909684658 |
| 9 | 0.9822180271148682 | 0.06924855709075928 | 0.9963235259056091 | 0.025619052350521088 |
| 10 | 0.9796927571296692 | 0.06791752576828003 | 0.9970588088035583 | 0.022172439843416214 |
| 11 | 0.9819023609161377 | 0.06351611018180847 | 0.9970588088035583 | 0.022720884531736374 |
| 12 | 0.9829545617103577 | 0.06131436303257942 | 0.9959558844566345 | 0.02010742388665676 |
| 13 | 0.9805344939231873 | 0.0621052160859108 | 0.9977940917015076 | 0.019726401194930077 |
| 14 | 0.9827440977096558 | 0.0563134029507637 | 0.997426450252533 | 0.01621229015290737 |
| 15 | 0.9826388955116272 | 0.05621267110109329 | 0.997426450252533 | 0.014518540352582932 |
| 16 | 0.9836910963058472 | 0.05271307751536369 | 0.9977940917015076 | 0.012878702953457832 |
| 17 | 0.9817971587181091 | 0.05311378464102745 | 0.997426450252533 | 0.014269452542066574 |
| 18 | 0.9860059022903442 | 0.04794914647936821 | 0.9963235259056091 | 0.016370059922337532 |
| 19 | 0.9862163066864014 | 0.04570940509438515 | 0.9985294342041016 | 0.010918369516730309 |
| 20 | 0.9859007000923157 | 0.04356583580374718 | 0.998161792755127 | 0.012143265455961227 |
| 21 | 0.9865319728851318 | 0.04298747330904007 | 0.9988970756530762 | 0.008486246690154076 |
| 22 | 0.9866372346878052 | 0.04276638478040695 | 0.997426450252533 | 0.011019202880561352 |
| 23 | 0.9843223690986633 | 0.04735783115029335 | 0.9985294342041016 | 0.011176807805895805 |
| 24 | 0.9851641654968262 | 0.04437693953514099 | 0.998161792755127 | 0.009232908487319946 |
| 25 | 0.9859007000923157 | 0.043020255863666534 | 0.9988970756530762 | 0.007596189621835947 |
| 26 | 0.9872685074806213 | 0.04164636507630348 | 0.9988970756530762 | 0.00683439988642931 |
| 27 | 0.9893729090690613 | 0.034871093928813934 | 0.9992647171020508 | 0.00801680888980627 |
| 28 | 0.9882155060768127 | 0.03970054164528847 | 0.9992647171020508 | 0.0062927426770329475 |
| 29 | 0.9850589036941528 | 0.04143829643726349 | 0.9985294342041016 | 0.008067913353443146 |
| 30 | 0.9860059022903442 | 0.04153968393802643 | 0.9992647171020508 | 0.00635097362101078 |

New high scores were also obtained during testing, as can be seen in the table below: 0.01 is the lowest loss score ever achieved, and the accuracy of 99.85% means that room for improvement has become even smaller.

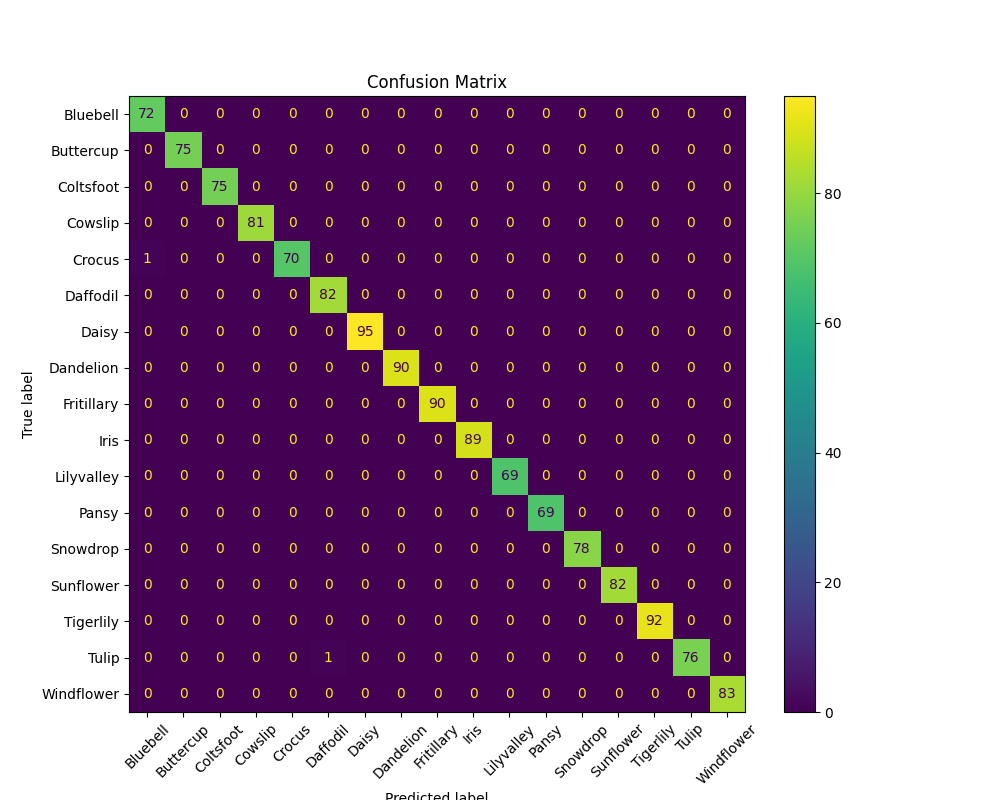
|  |  |
| --- | --- |
| Test Loss | 0.010740709491074085 |
| Test Accuracy | 0.9985401630401611 |

Although the training and validation plots of the accuracy and loss graphs appear to have a slight gap between them, this is not being classed as overfitting, as statistically the distance between the plots is miniscule, and is exaggerated by the scale of the graphs. The visible plateaus are also a welcome sight, as they confirm that the model has been trained thoroughly, while its performance remains consistent.

One note worth making from both the table above and the graph below is that peak values are reached within the first 10 epochs. While it is good that they remained at their peaks for the remainder of the training period, 30 epochs may be excessively high, as no benefit is gained during the latter epochs.



Impact of the increased dataset is brilliantly reflected in the confusion matrix. Even though the input data was increased 10x compared to the original, and 5x compared to the previous stage, this confusion matrix contains only 2 incorrect predictions. Clearly, the additional input data allowed the model to gain even deeper understanding of the differentiating details of each class, enabling it to achieve near-perfect results.



Once again, even though the difference is minute, the classification report for 5.5 reflects yet another improvement in model performance. 1.0 is the most populous value within the table, reiterating the almost perfect analysis of the confusion matrix, while the remaining values such as the averages at the bottom of the table edge even closer to total accuracy.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Class | Precision | Recall | F1-Score | Support |
| Bluebell | 0.9863013698630136 | 1.0 | 0.993103448275862 | 72.0 |
| Buttercup | 1.0 | 1.0 | 1.0 | 75.0 |
| Coltsfoot | 1.0 | 1.0 | 1.0 | 75.0 |
| Cowslip | 1.0 | 1.0 | 1.0 | 81.0 |
| Crocus | 1.0 | 0.9859154929577465 | 0.9929078014184397 | 71.0 |
| Daffodil | 0.9879518072289156 | 1.0 | 0.9939393939393939 | 82.0 |
| Daisy | 1.0 | 1.0 | 1.0 | 95.0 |
| Dandelion | 1.0 | 1.0 | 1.0 | 90.0 |
| Fritillary | 1.0 | 1.0 | 1.0 | 90.0 |
| Iris | 1.0 | 1.0 | 1.0 | 89.0 |
| Lilyvalley | 1.0 | 1.0 | 1.0 | 69.0 |
| Pansy | 1.0 | 1.0 | 1.0 | 69.0 |
| Snowdrop | 1.0 | 1.0 | 1.0 | 78.0 |
| Sunflower | 1.0 | 1.0 | 1.0 | 82.0 |
| Tigerlily | 1.0 | 1.0 | 1.0 | 92.0 |
| Tulip | 1.0 | 0.987012987012987 | 0.9934640522875817 | 77.0 |
| Windflower | 1.0 | 1.0 | 1.0 | 83.0 |
| macro avg | 0.9984854810054077 | 0.9984075576453373 | 0.9984361585836046 | 1370.0 |
| weighted avg | 0.9985589392867943 | 0.9985401459854014 | 0.9985399010990842 | 1370.0 |

## 5.6

Although the results of 5.5 were close to perfection, there is still a small amount of room for improvement. Stage 5.6 will hopefully conclude the model development process, as the final point of interest is explored, that being the fine-tuning of the base model. Throughout all previous stages of phase 5, the pre-trained *MobileNetV2* architecture had been frozen, meaning that the values within its layers were not manipulated, it was only the additional layers of the network whose values could be adjusted to improve performance. 5.6 is different, as the final 20 layers of the *MobileNetV2*’s architecture have been unfrozen, allowing them to be fine-tuned and optimized to produce a model with even better performance overall.

Due to the excessively long training process in 5.5, the number of epochs has been reduced to 15 for 5.6. To balance this and ensure efficient convergence, the learning rate has been increased to 0.00005.

mobilenet\_finetune\_LR5e-05\_BS32\_E15

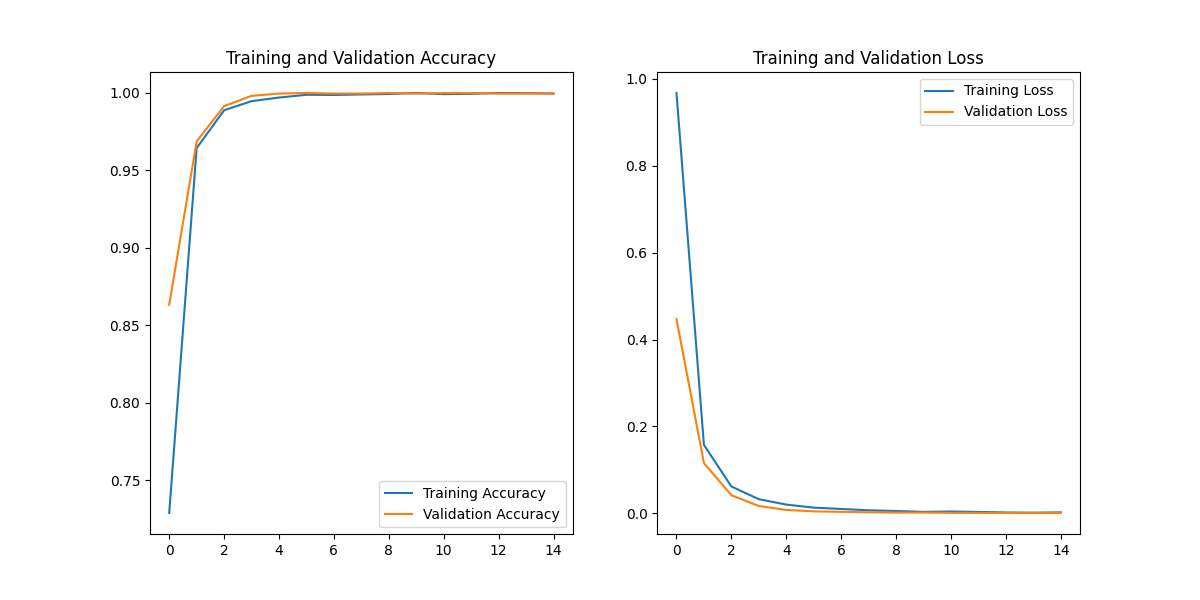
The table below contains the performance results of the training process of 5.6. It appears that implementing the fine-tuning was a success, as from around the 3rd and 4th epochs onwards, both the training and validation accuracy is effectively 100%. Additionally, from around a similar point onwards, the training and validation loss values are negligibly minute, with the final training accuracy being 0.002 and the final validation accuracy being 0.0009. This training process has clearly produced the best results of any stage or test throughout the entire project.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Epoch | accuracy | loss | val\_accuracy | val\_loss |
| 1 | 0.7287458181381226 | 0.9673189520835876 | 0.8632352948188782 | 0.4471583068370819 |
| 2 | 0.9644360542297363 | 0.15753474831581116 | 0.96875 | 0.11574357002973557 |
| 3 | 0.9888467788696289 | 0.06195348873734474 | 0.9915441274642944 | 0.04180223494768143 |
| 4 | 0.9947390556335449 | 0.032655052840709686 | 0.998161792755127 | 0.016980042681097984 |
| 5 | 0.997053861618042 | 0.020272744819521904 | 0.9996323585510254 | 0.007962165400385857 |
| 6 | 0.9988425970077515 | 0.013228721916675568 | 1.0 | 0.004651791416108608 |
| 7 | 0.9987373948097229 | 0.010243386961519718 | 0.9996323585510254 | 0.003601826261729002 |
| 8 | 0.9990530014038086 | 0.007150378543883562 | 0.9996323585510254 | 0.0026229710783809423 |
| 9 | 0.9993686676025391 | 0.005513632670044899 | 1.0 | 0.001840853481553495 |
| 10 | 1.0 | 0.0035256489645689726 | 0.9996323585510254 | 0.0019930799026042223 |
| 11 | 0.9992634654045105 | 0.004370170179754496 | 1.0 | 0.001312418025918305 |
| 12 | 0.9994739294052124 | 0.0033531379885971546 | 1.0 | 0.0010911185527220368 |
| 13 | 1.0 | 0.0021787055302411318 | 0.9996323585510254 | 0.0010156809585168958 |
| 14 | 0.9998947978019714 | 0.0016533697489649057 | 0.9996323585510254 | 0.0009210658608935773 |
| 15 | 0.9996843338012695 | 0.0023378911428153515 | 0.9996323585510254 | 0.0009553879499435425 |

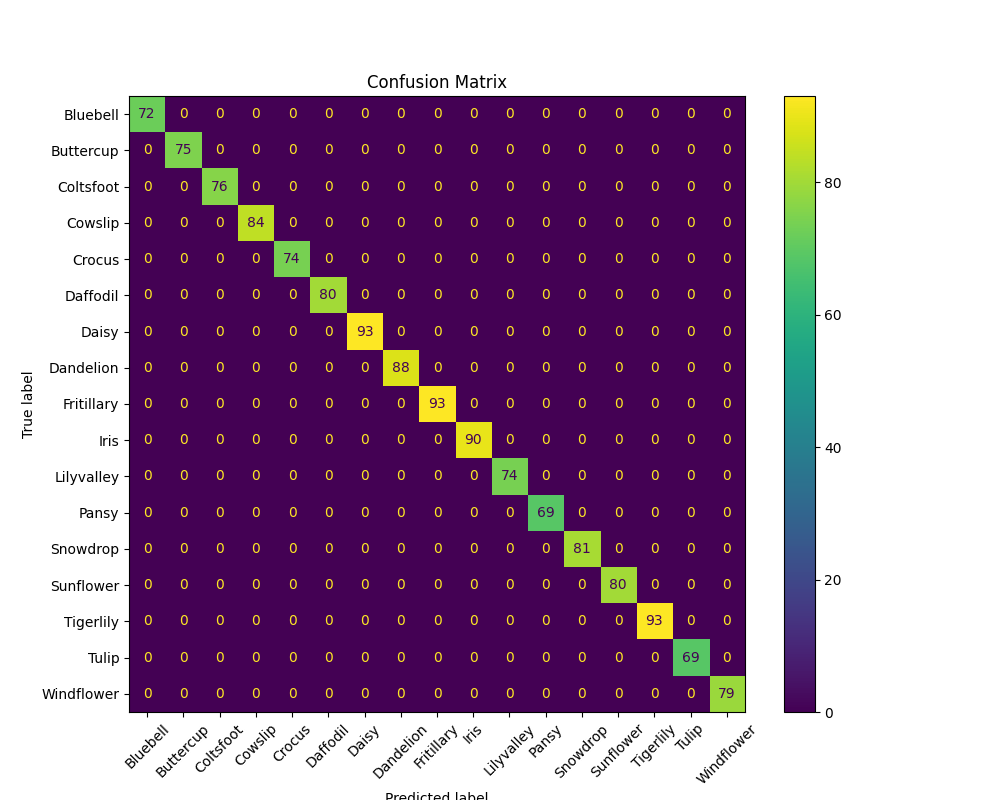
The outstanding training process is reflected in equally outstanding test results, as the test accuracy is 100%, while the test loss is 0.0003.

|  |  |
| --- | --- |
| Test Loss | 0.000334896583808586 |
| Test Accuracy | 1.0 |

The graphs accompanying the training process display the perfect outcome of the training process, with rapid convergence and negligible fluctuation.



Further confirmation of the success of this training stage comes in the form of the perfect confusion matrix below, which contains not a single incorrect prediction.



As expected, the accompanying classification report echoes the rest of the performance metrics, with the precision, recall and F1-Score of every single class achieving a score of 1.0, or perfection.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Class | Precision | Recall | F1-Score | Support |
| Bluebell | 1.0 | 1.0 | 1.0 | 72.0 |
| Buttercup | 1.0 | 1.0 | 1.0 | 75.0 |
| Coltsfoot | 1.0 | 1.0 | 1.0 | 76.0 |
| Cowslip | 1.0 | 1.0 | 1.0 | 84.0 |
| Crocus | 1.0 | 1.0 | 1.0 | 74.0 |
| Daffodil | 1.0 | 1.0 | 1.0 | 80.0 |
| Daisy | 1.0 | 1.0 | 1.0 | 93.0 |
| Dandelion | 1.0 | 1.0 | 1.0 | 88.0 |
| Fritillary | 1.0 | 1.0 | 1.0 | 93.0 |
| Iris | 1.0 | 1.0 | 1.0 | 90.0 |
| Lilyvalley | 1.0 | 1.0 | 1.0 | 74.0 |
| Pansy | 1.0 | 1.0 | 1.0 | 69.0 |
| Snowdrop | 1.0 | 1.0 | 1.0 | 81.0 |
| Sunflower | 1.0 | 1.0 | 1.0 | 80.0 |
| Tigerlily | 1.0 | 1.0 | 1.0 | 93.0 |
| Tulip | 1.0 | 1.0 | 1.0 | 69.0 |
| Windflower | 1.0 | 1.0 | 1.0 | 79.0 |
| macro avg | 1.0 | 1.0 | 1.0 | 1370.0 |
| weighted avg | 1.0 | 1.0 | 1.0 | 1370.0 |

In conclusion, based upon all the data gathered during the training and testing stages, the model created in 5.6 has mastered the input dataset, and has become highly confident and accurate in predicting the 17 species of flowers contained within. Of course, this perfect accuracy is unlikely to remain intact once the model is given new images from external sources such as user photographs, however it has clearly understood the nuances which differentiate the various classes of the dataset, and should retain a solid capability to classify the species it knows with confidence.

Now that perfect results have been achieved by the model of 5.6, the model development stage can be relievingly concluded. The next stage of the project will be to create some kind of application to house the model, so that users will be able to interact with it, providing it with images of flowers and receiving predictions on their species. Once the app is created, both the app and model can be tested in real-world scenarios.

Implementation & Testing – App Development Phase

Now that the substantial model development process has been completed, the next task is to create an app which can house the model and allow it to be deployed and used in real-world scenarios. Various approaches could be taken to create this. For one, a full mobile application could be created, using either native technologies such as Swift or Kotlin, or cross-platform frameworks such as Flutter or React Native. Alternatively, a desktop application could be created, like a Windows *WinForm* application using C# and *.NET*, or a MacOS application using Swift. Or finally, a web-based application could be created using any of the many available technologies, such as HTML and CSS for a simple front-end design, or React.js or Vue.js for more dynamism, and then for the back-end, Node.js, Django or Flask would be suitable tools.

Initial experimentation was performed with Kotlin and Android studio to create a native mobile application, as well as with Dart and Flutter to explore the possibility of creating a cross-platform application, with TensorFlow integrated seeming highly plausible due to both these frameworks being supported by Google, just like TensorFlow itself. Various tutorials were completed, primarily from the Google and Android Developer services, however the complexity and learning process were daunting, and deemed excessive for a simple application such as this. Therefore, the decision was made to explore the creation of a web-based application, as this approach would be relatively simple and efficient yet completely appropriate for the task at hand. The technologies targeted for use are HTML and CSS for the front-end due to their simplicity and familiarity, as well as Flask for the back-end due to its light weight, flexibility, and ease of TensorFlow integration.

Additional important technologies will be a hosting service such as Google Cloud Run, as well as a containerisation service such as Docker. The former will facilitate a smooth deployment process and provide scalability and reliability, while the latter will package the entire application into a light-weight container, including the front-end, back-end, model and all required tools and libraries, ensuring consistency between devices.

## Wireframe

The figure below is a wireframe depicting the desired design of the web application. It features a text box containing an appropriate title; two buttons, one for uploading an image and another for requesting a prediction; a section to display the most recently uploaded image; and a text box at the bottom containing the model’s predicted label alongside its confidence percentage.

A screenshot of a phone

AI-generated content may be incorrect.

Although rather basic in its design, this wireframe contains all essential components, while maintaining a clean and elegant aesthetic.

## Front-End

As previously stated, the front-end design is intentionally kept minimal and clean while ensuring all necessary functionality is incorporated, using HTML and CSS. The purpose of the front-end code is to provide the user with an interactive, comprehensible interface through which they can obtain predictions on uploaded flower imagery.

During operation, the front-end accepts the user’s image input and sends it to the back-end, where it is classified by the trained TensorFlow model. Once processed, the model’s prediction and confidence percentage are returned and displayed via the front-end. Additionally, the uploaded image is shown to allow the user to verify that the correct image has been classified.

The first key section of the front-end code is the snippet shown in the figure below, which specifies the document type and language used. The <title> tag at the bottom of this section sets the text that appears on the browser tab when the web application is running.

A black screen with white text

AI-generated content may be incorrect.

Following this, the next important section is the CSS styling block, responsible for the visual appearance of all UI elements. The upper body styling area defines the background colour, font, text alignment, and padding, ensuring consistency across the application. The .container section specifies details of the central content box, which houses all core UI features, including the header (h2), the upload form, buttons, and the image container. This section is primarily responsible for the clean and user-friendly appearance of the interface.

A screen shot of a computer program

AI-generated content may be incorrect.

Next is the header section, containing the visible title displayed at the top of the web application. This header provides clear instructions, ensuring that users understand the purpose of the application at first glance.

A black background with white text

AI-generated content may be incorrect.

The following section of vital importance is the image upload form, which handles the receipt of the user’s uploaded image. In the opening <form> tag, the action="/" attribute sends the data to the back-end’s root route, the method="post" attribute specifies that the form will use the HTTP POST method to transmit the file securely, and enctype="multipart/form-data" ensures that the binary file data is properly encoded. The next line contains <input type="file" name="file" required>, where type="file" creates a file picker button in the UI, allowing the user to select an image from their device. The name="file" attribute is essential because it sets the key name used by the back-end to retrieve the uploaded file. Including the required attribute prevents form submission without a file, thereby improving usability by enforcing correct input. The final line within the form is the <button type="submit">Predict</button> element, which creates the Predict button. When pressed, this button triggers the submission of the form and the selected image, sending a POST request to the back-end as defined earlier. The back-end processes this request, passing the image to the TensorFlow model to generate a classification result.

A black screen with white text

AI-generated content may be incorrect.

The penultimate section of significance is responsible for displaying the classification result once the model has processed the uploaded image. This block is conditionally displayed only after a prediction has been made by the back-end. It dynamically shows the uploaded image, the predicted class label, and the confidence percentage. These features ensure that the user not only sees the classification result but can also verify the image they uploaded and view the model’s confidence in its prediction.

A computer code with text on it

AI-generated content may be incorrect.

Lastly, the final section, shown in the figure below, handles the display of any errors encountered throughout the process, such as the submission of an invalid file type. This section plays an important role in enhancing the user experience by clearly informing users of any issues that may arise during operation, allowing them to correct errors promptly.

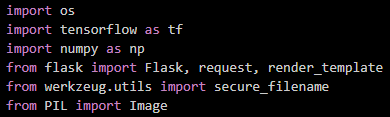
A black background with white text

AI-generated content may be incorrect.

## Back-End

The back-end of the application is implemented using Flask, a lightweight Python-based web framework, and is responsible for handling the core functionality of the system. Specifically, it manages image uploads from the user, processes the uploaded image using the trained TensorFlow model, and returns the resultant prediction and confidence score to the front-end. The back-end ensures smooth interaction between the user interface and the machine learning model, allowing the system to provide accurate, real-time classifications.

The first important section of the back-end code is the list of imported dependencies, as shown in Figure 4.1 below. This section imports several essential libraries, including Flask and its *render\_template* function, which allows the application to render dynamic HTML pages. TensorFlow and NumPy are imported to load the pre-trained model and process numerical data, while PIL (Python Imaging Library) is imported for image processing tasks. Additionally, *secure\_filename* from the werkzeug.utils library is imported to safely handle file names when saving uploaded images.

**

*Insert Figure 4.1 – Imported Dependencies and Libraries*

The next key section of the back-end, shown in Figure 4.2, involves the initialization and configuration of the Flask application. Here, the Flask app is initialized, and the UPLOAD\_FOLDER variable is defined to specify the directory where uploaded images will be stored. The folder is created if it does not already exist, ensuring the application can store incoming files. The ALLOWED\_EXTENSIONS set is also defined, listing the supported image formats (PNG, JPG, JPEG) that the system will accept for classification.

*A computer screen shot of a program code

AI-generated content may be incorrect.*

*Insert Figure 4.2 – Flask Initialization and Upload Configuration*

Following this, the TensorFlow model is loaded, as illustrated in Figure 4.3. The trained model is loaded from the specified file path using TensorFlow’s load\_model function. Alongside this, a list of class names is defined, corresponding to the output indices of the model. This allows the system to map the model’s numeric output to a human-readable class label, which will later be displayed to the user.

*A screen shot of a computer code

AI-generated content may be incorrect.*

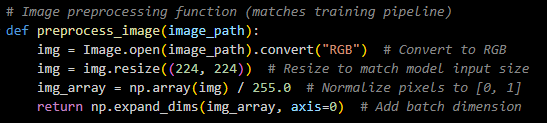
*Insert Figure 4.3 – Model Loading and Class Name Definitions*

The next section of significance is the file validation function, shown in Figure 4.4. The allowed\_file() function checks the uploaded file’s extension to ensure it matches one of the supported image formats defined earlier. This step is crucial in preventing invalid files from being processed and improves system reliability and security.

**

*Insert Figure 4.4 – Allowed File Type Check Function*

The image preprocessing function, presented in Figure 4.5, is responsible for preparing the uploaded image to match the input requirements of the TensorFlow model. This function converts the image to RGB format, resizes it to 224x224 pixels, normalizes the pixel values, and adds a batch dimension. Proper preprocessing ensures that the model receives input in the same format as it was trained on, which is essential for achieving accurate predictions.

**

*Insert Figure 4.5 – Image Preprocessing Function*

A simple health check route is also included in the back-end, as seen in Figure 4.6. This route returns an HTTP 200 response when accessed, indicating that the server is running correctly. While not directly related to user-facing features, it plays an important role in deployment, particularly on platforms like Google Cloud, where health monitoring is essential.

**

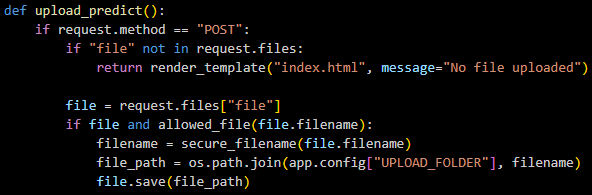
*Insert Figure 4.6 – Health Check Endpoint*

The main route definition of the application is shown in Figure 4.7. This route handles both GET and POST requests sent to the root URL. When the user accesses the application initially, a GET request is made, prompting the back-end to render the front-end interface. When the user submits an image, a POST request is sent, triggering the subsequent processing steps.

**

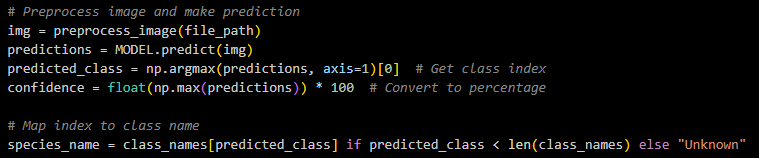
*Insert Figure 4.7 – Main Route Definition*

The next section, displayed in Figure 4.8, handles the file upload process. The system first checks if a file has been submitted and validates its format using the previously defined allowed\_file() function. If the file is valid, it is safely saved to the designated upload folder using a secure filename. If no file is submitted, or if the file is invalid, an appropriate error message is rendered back to the front-end.

**

*Insert Figure 4.8 – File Upload Handling and Validation*

Following this, the model prediction process is conducted, as demonstrated in Figure 4.9. The uploaded image is preprocessed, passed through the TensorFlow model, and the resulting prediction is obtained. The model’s output is processed to extract the predicted class index and the associated confidence score, which is converted into a percentage for display purposes. The class index is then mapped to the corresponding species name from the predefined list.

**

*Insert Figure 4.9 – Model Prediction and Result Extraction*

Once the prediction has been completed, the system dynamically renders the front-end HTML page, passing the prediction result, confidence score, and filename of the uploaded image back to the user interface. This is achieved through Flask’s render\_template function, as shown in Figure 4.10. This step ensures that the front-end can display the results to the user clearly and accurately.

**

*Insert Figure 4.10 – Rendering Prediction Results to Front-End*

In the case of a GET request, or if no valid image has been uploaded, the system simply renders the front-end page without any prediction data, as illustrated in Figure 4.11. This allows the user to view the interface and upload an image when desired.



*Insert Figure 4.11 – Rendering Initial Front-End Page*

The final section of the back-end, shown in Figure 4.12, involves starting the Flask web server. The application is configured to listen on a dynamic port, which is particularly useful for cloud-based deployments, and runs on all available IP addresses to ensure accessibility.



*Insert Figure 4.12 – Starting the Flask Web Server*

In summary, the back-end handles all essential functionality behind the scenes, including validating user input, preprocessing images, executing the TensorFlow model, and returning dynamic results to the front-end interface. Its clean and efficient design allows for seamless interaction between the user and the classification model, ensuring a smooth user experience throughout the application.

## Containerisation

To ensure the application can be deployed and run consistently across different environments, containerisation is utilised through Docker. Docker allows the entire application, including its dependencies and runtime environment, to be packaged into a single, lightweight container image. This eliminates potential issues arising from differing system configurations and simplifies deployment.

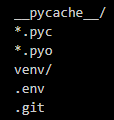
The container is built based on the configuration defined in the Dockerfile, shown in *Figure Y.1*. The Dockerfile begins by specifying an official lightweight Python 3.9 base image. It then sets the working directory within the container to /app and copies all necessary files from the current directory into this directory. Following this, it installs the application's dependencies using pip and the requirements.txt file. Port 8080 is exposed to allow communication with the Flask web server. Finally, the application is configured to run using Gunicorn, a production-ready WSGI HTTP server, which serves the Flask app on all available network interfaces at port 8080.

*A screenshot of a computer program

AI-generated content may be incorrect.*

*Insert Figure Y.1 – Dockerfile Configuration*

To further optimise the containerisation process, a .dockerignore file is included, shown in *Figure Y.2*. This file specifies files and directories that should be excluded from the Docker image, such as Python cache files, virtual environments, environment files, and version control metadata. By excluding unnecessary files, the build context is kept lightweight, resulting in a smaller, more efficient container image.

**

*Insert Figure Y.2 – .dockerignore Configuration*

Through the use of Docker, the application can be reliably built, tested, and deployed in a reproducible manner, ensuring consistency across different environments.

## Hosting

For hosting the containerised application, Google Cloud Run is employed. Google Cloud Run is a fully managed serverless platform that enables the deployment of containerised applications without the need to manage infrastructure.

The container image, built locally using Docker, is pushed to Google Container Registry or Artifact Registry. From there, Google Cloud Run is used to deploy the container, allowing the application to scale automatically in response to incoming traffic. The platform handles all aspects of provisioning, managing, and scaling the infrastructure, ensuring high availability and minimal operational overhead.

Additionally, Cloud Run supports dynamic port assignment, which is accommodated within the Flask back-end code by retrieving the port from the environment variable PORT, defaulting to 8080 if not specified. This seamless integration ensures that the application runs correctly within the Cloud Run environment.

By leveraging Google Cloud Run, the application benefits from scalability, security, and reliability, while minimising the complexity involved in managing servers or infrastructure manually.

## Minor Adjustments

Include project title – Flower Power – at top of application

Include name and student number at bottom of application

Implement webp format acceptance

Improved error handling

[https://chatgpt.com/c/67d8581f-9a00-8009-b6bf-7072d2c3efdb - brave - roms11497](https://chatgpt.com/c/67d8581f-9a00-8009-b6bf-7072d2c3efdb%20-%20brave%20-%20roms11497)

[https://chatgpt.com/c/67d6cd69-b544-8000-be0f-fd46f871a3fd - chrome - rms11497](https://chatgpt.com/c/67d6cd69-b544-8000-be0f-fd46f871a3fd%20-%20chrome%20-%20rms11497)

