Flower Classification – Transfer Learning

Adrian Ash, Chiran Walisundara and Don Kaluarachchi – Group 50

Introduction

The project requirement was to create a deep neural network for the task of classification. A "MobileNetV2" trained on the ImageNet, an extremely large image dataset, was used as the base network. The network was then to be trained on a flower dataset containing 5 different types of flowers. The goal of the network was to accurately predict the type of flower based on an image.

Data Pre-processing

The data pre-processing for this project consisted of:

Resizing the data from 256x256 images into 224x224 images. This was done to fit the original input sizes the "MobileNetV2" model trained the ImageNet on. If this (or one of the other predetermined sizes) was not set, then the weights loaded for the model may not fit our input as well

The data was then split into Training, Validation and Testing sets.

- The training set was 70% of the data (700 samples)
- The validation set was 15% of the data (150 samples)
- The testing set was 15% of the data (150 samples)

This split allows for a reasonable training size, while also leaving enough in the validation set to tune the hyperparameters. The test data is used at the end to determine model performance metrics. Using Sklearn's train_test_split() function we can see from the figure below (Figure 1) that the dataset classes are relatively evenly distributed. This even split is the same for both the validation and testing sets with only minor differences.

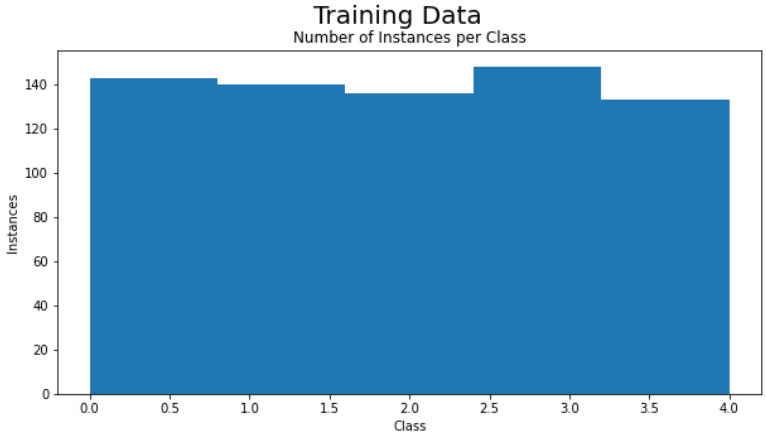

***Figure 1**: A histogram of the number of instances per class within the Training data*

This is an important aspect of our datasets as the training data needs to generally mirror the overall distribution of all the data to not overfit on larger classes.

The final pre-processing that was used was to use the "MobileNetV2" network to convert data into image embeddings. This links to the next section on computational constraints however the overall

reason was to use a simpler network to complete the final steps of the classification, thus vastly increasing training time, while still benefiting from the learnt weights from the ImageNet dataset.

Computational Constraints

As mentioned in the previous sections, there were a lot of computational considerations that were made to ensure that models could be trained in a timely manner, or, in some extreme cases help not crash the computer entirely.

The "MobileNetV2" consists of 154 different layers. This is quite a deep neural network and as such mirrors similar model architectures (like Residual Networks) by implementing "skip connections". This overcomes the problem of disappearing gradients during backpropagation that plague simple forward feeding networks. With this depth however it is computationally expensive to train or run. To overcome this, we use methods like transfer learning, freezing layer and fine-tuning which help however does not affect the forward propagation and fitting models and data in memory. This is the main issue faced by my system for this assignment as without a GPU or a high-end system it was too computationally expensive. To fix this issue I decided to use the "MobileNetV2" network to just get a 7x7x1280 embedding of our data. With the data converted to this form, a simple network (Figure 2) was used to complete the training and classification.

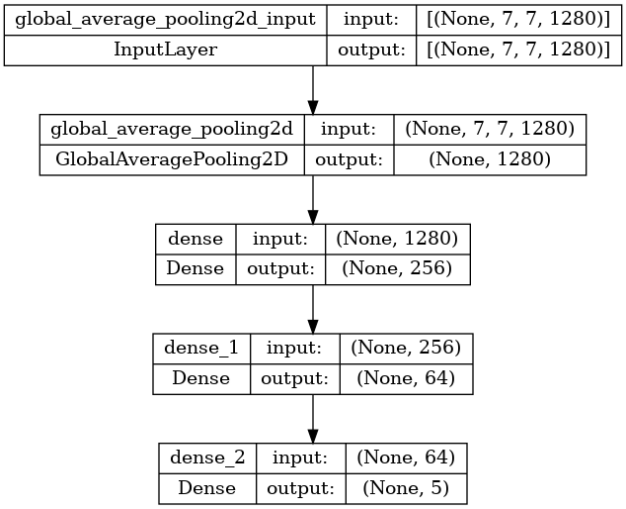

***Figure 2**: Simple model design to take in the "MobileNetV2" embedding and classify to one of five classes*

Note that the assignment required only adding a single dense layer of 5 as the output. I chose to use a slightly more complex model as compensation for not leveraging the complete "MobileNetV2" network and fine-tuning. The goal was to learn a more detailed separation of our embedded 7x7x1280 input for better classification results. After testing a single dense output vs this model, I was able to consistently achieve slightly more accuracy which was important for testing different parameters and as such chose to continue using this model.

Model Results

Before we evaluate the result from the models it is important to note that deep neural networks have a random factor to them. These two main considerations have to be made for our models:

Firstly to minimise this random factor a seed was used for both image_dataset_from_directory() and train_test_split() functions to further enhance consistency. This may however also lead to a performance decrease if the chosen seed does not split our small dataset well, for example too much noise in our testing set. That being said if the data split seriously affects our performance it is normally a sign of a poor dataset (more about this in recommendations).

The second method to minimise this random factor was using Kera's ModelCheckpoint() function as it allows us to use the model with the greatest performance from all our epochs. For this reason, we were able to train for 150 epochs. This will allow for all our models, regardless of learning rate or momentum parameters to reach a decent level of convergence and in the case of overfitting ensure we still have the best model. This is an important note when looking at our performance stats over time graphs as many will look like they are overfitting whereas our confusion matrices are based on our best models.

When evaluating the model performance, we will be judging each based on the "Weighted Average F1-score" (which can be found in the program output - classification report). This is done because "F1-score" is a combination of both precision and recall accuracy and "Weighted Average" ensures that all classes are represented evenly in the accuracy stat. This is not a major concern because as we mention before train_test_split() splits the classes in the testing set relatively evenly.

Model 1 – Learning Rate = 0.01 and Momentum = 0

This is the first model requested by the assignment (task 5) and will use the default values of our SGD optimizer. This model had an F1-Score of 55% which was a reasonable score compared to the other models. Figure 3 will be used to investigate these results. Our training confusion matrix is likely the first point of interest. We can see that we have 100% accuracy on our training data which is great however with our testing confusion matrix not showing the same results we can hypothesise that our model is overfitting to our training data.

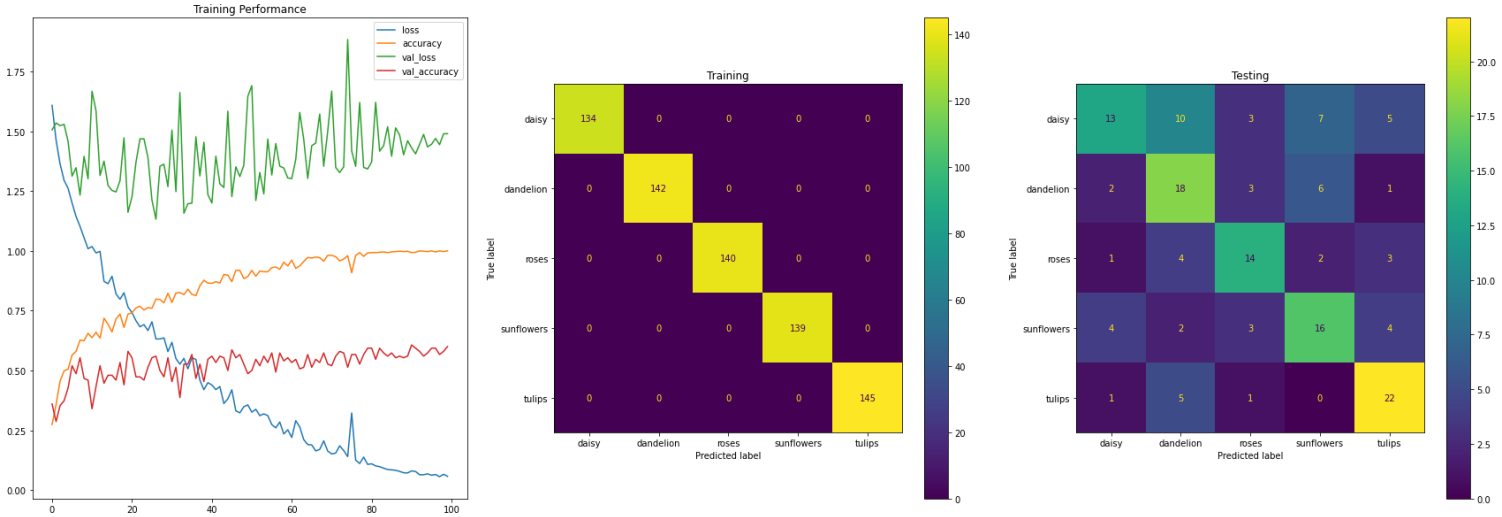

Figure 3: *Left: Loss and Accuracy over time (epochs)*
Middle: Confusion matrix on Training Data
Right: Confusion matrix on Testing

As we can see from our training accuracy and loss lines for the 100 epochs the model continues to overfit. After epoch number 60 roughly, we can see the accuracy routinely top out at 100%. The validation accuracy and loss on the other hand show a different trend. The loss hovers between 1.25

and 1.75 spiking up and down. These spikes seem to correlate with the validation accuracy however unlike the loss there seems to be an ever so slight upwards trend. Due to the ModelCheckpoint() it is likely the final model was chosen around epoch 20 which looks like the largest accuracy spike.

Key points of this model are that our validation loss seems to be violently spiking while our training loss is steadily decreasing. This is indicative of our model not generalising well and overfitting to our training data, which we will see is due to a learning rate that is too large (and a dataset that is too small).

Model 2 – Learning Rate = 0.0001 and Momentum = 0

The second task was to create 3 different models with varying Learning Rates to example the results such a change would make. This model is running on a learning rate of 0.0001 which is extremely small. As we will see in Figure 4 this leads to extremely slow learning and our 100 epochs aren't close to finding a local minimum. This is represented in our training and testing accuracy as we can see they seem to rare predict correctly.

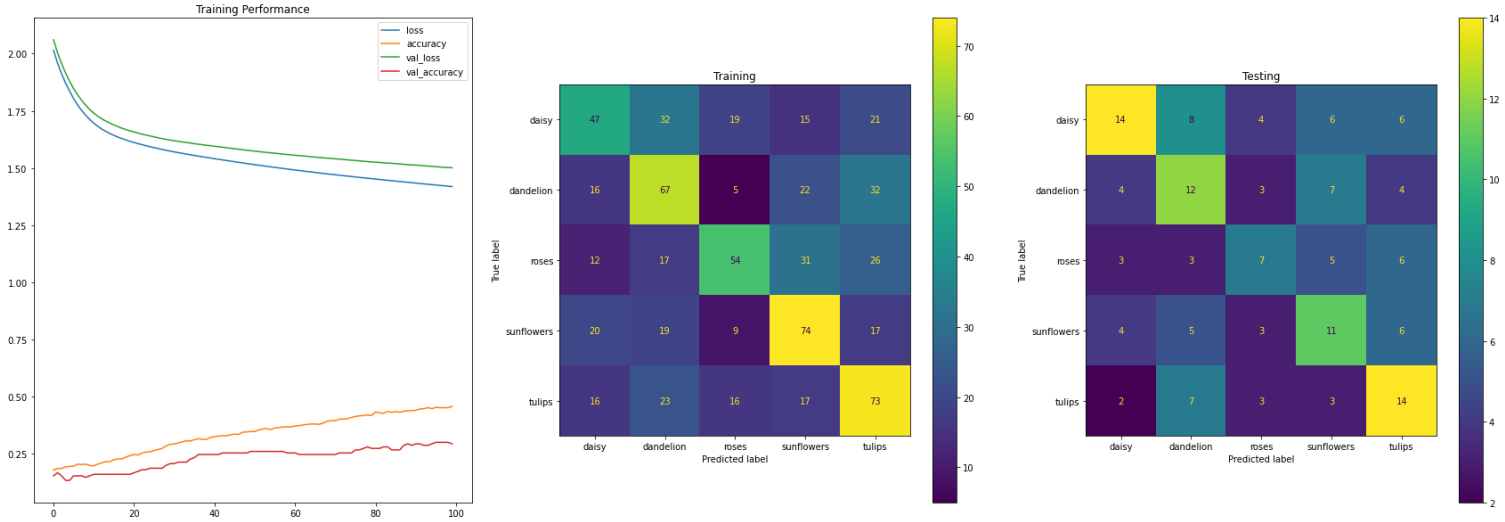

Figure 4: *Left: Loss and Accuracy over time (epochs)*
Middle: Confusion matrix on Training Data
Right: Confusion matrix on Testing

As we can see the learning rate is far too small to get any meaningful information from this diagram. This is on the extreme of small learning rates and as such would take many epochs before any convergence occurs. We can see that compared to our previous model there are almost no violent spikes in any of our metrics this is because each epoch only updates our weights and biases are smaller amounts due to our learning rate. This means that each step is very similar to our previous thus less turbulent change between them.

Final point is that this model performs the worst by far, only having an F1-Score of 39%.

Model 3 – Learning Rate = 0.001 and Momentum = 0

The second learning rate used was 0.001. This model worked the best of all our models, mostly because it ran for the 100 epochs. With an F1-Score of 58%, it was used for the models 5-7 momentum examination section. It is important to note that this is because we were able to run our model for 100 epochs, however as we will see in Figure 5, it also had the benefit of a gradual and smooth increase in our performance metrics. As we will see throughout the other models these training and testing

confusion matrices are all generally very similar with only minor differences in which classes are more accurately predicted (often at the cost of other class predictions).

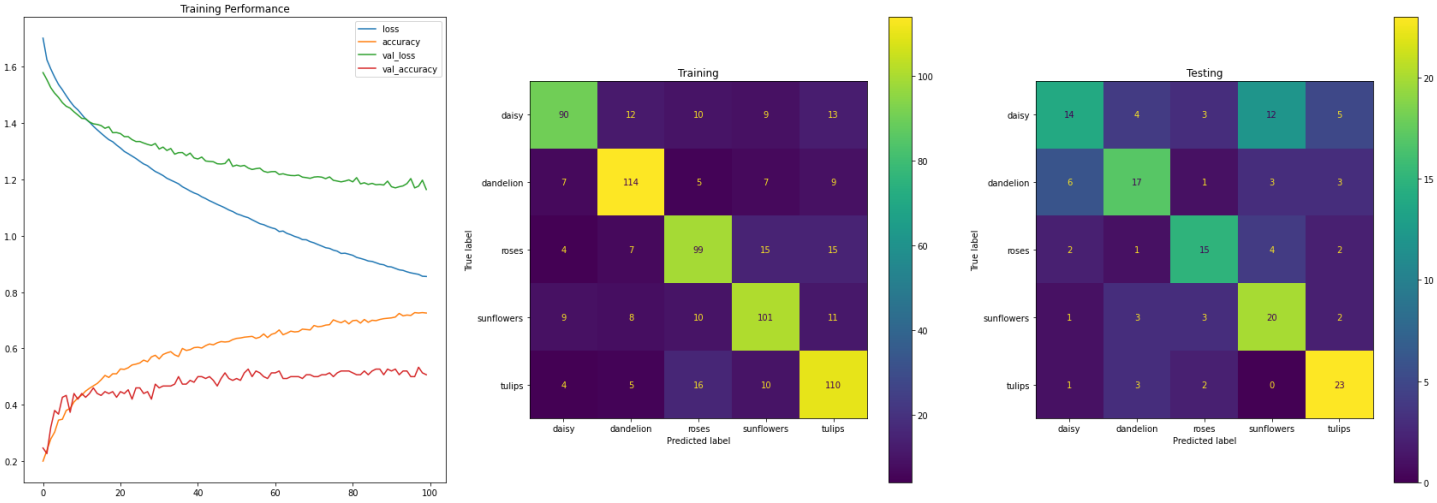

Figure 5: *Left: Loss and Accuracy over time (epochs)*
Middle: Confusion matrix on Training Data
Right: Confusion matrix on Testing

We can see from this model that both the validation and training losses continually decreased whereas both our accuracy stats showed an upwards trend. Compared to Model 1 our stats also don't spike as much due to our smaller updates. It can be said however that this model has not converged yet even though our validation stats seem to be levelling out. Note that of our learning rates both 0.01 (Model 1) and 0.001 (this Model) performed quite well, so giving more time it is likely that further investigation between these two points would provide a learning rate that converges as quicker than 0.001 but is smoother than 0.01.

Model 4 – Learning Rate = 0.1 and Momentum = 0

Our final learning rate model was 0.1. As we will see from Figure 6 this is the other end of the learning rate spectrum. By happenchance, this model found a local minima that provided an F1-Score of 55% however our training confusion matrix accuracy indicates this model might be close to overfitting.

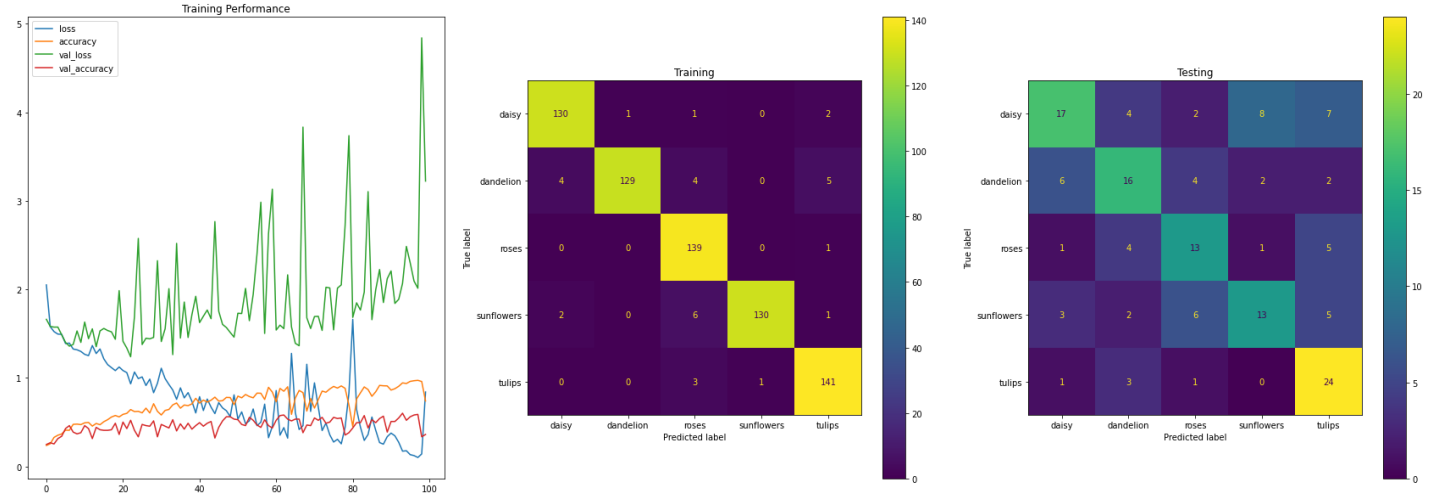

Figure 6: *Left: Loss and Accuracy over time (epochs)*
Middle: Confusion matrix on Training Data
Right: Confusion matrix on Testing

As mentioned above this model performs very poorly over our 100 epochs, especially in the latter half. Our extremely large learning rate had allowed the model to jump out of decent local minima into some very poor models. Two perfect examples of this are around models 80 and 90 where we have two large spikes. Note that these spikes lead to subsequent dips in both our validation and testing accuracy which is to be expected. This model, despite finding a configuration with decent accuracy, was by far the most untrustworthy due to its extremely large "steps" causing it never to settle to a good minimum.

Model 5 – Learning Rate = 0.001 and Momentum = 0.01

The next task was to experiment with 3 different momentum values for our model. Our first momentum was 0.01 and is the smallest of the 3. With an F1-Score of 50%, we can see that it performs worse than our original version (Model 3). We will go over the reason for this in Model 7 as it seems to be a trend.

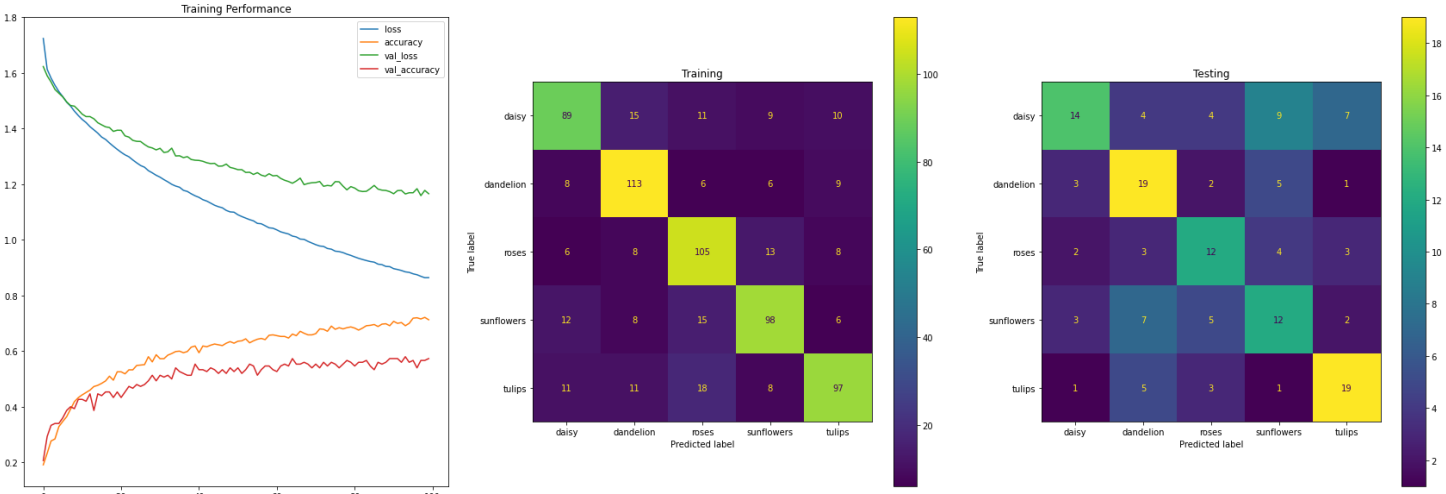

Figure 7: *Left: Loss and Accuracy over time (epochs)*
Middle: Confusion matrix on Training Data
Right: Confusion matrix on Testing

We can see in Figure 7 that our testing confusion matrix is standard. Our training performance shows that this small momentum does not seem to make notable changes to our original model. This will be the same with our next model as well however our final model will perform quite poorly. For this reason, we can hypothesise that, at least for this model, our momentum requires far higher fine-tuning. Unfortunately, due to the nature of our small data sample size, each individual run of our models provides vastly different results making it hard to fine-tune (more about this in the recommendations section). Looking at the accuracy stats, however, there is an 8% difference. Momentum allows the model to escape local minima by calculating an extra value for our optimiser which is the combination of "momentum" of the gradient descent over the previous few updates. It is likely then that this model escaped a local minimum that provided the best result on our test data.

Model 6 – Learning Rate = 0.001 and Momentum = 0.1

The second momentum experiment was taken using 0.1. This model performed extremely similar to our previous model however was 1% worst at 49%. As we will see in Figure 8 there are a few differences between the two models. Looking at the actual performance of the model, the testing confusion matrix clearly shows that this model is horrible at predicting daisies.

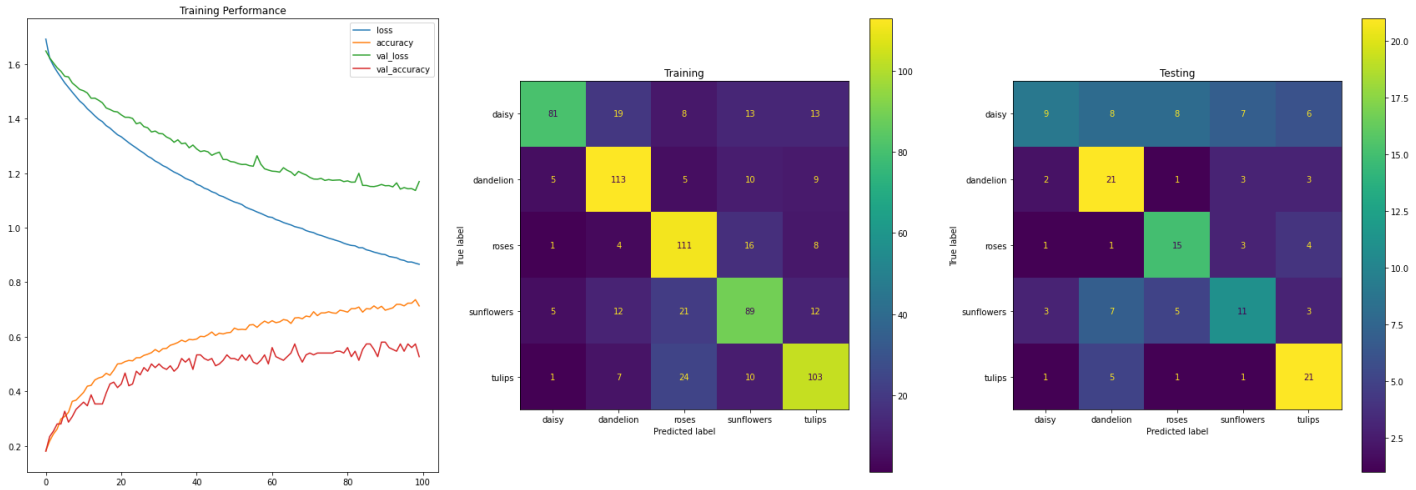

Figure 8: *Left: Loss and Accuracy over time (epochs)*
Middle: Confusion matrix on Training Data
Right: Confusion matrix on Testing

The other difference with regards to the training between this model and Model 5 is that our validation accuracy takes longer to increase early on while also showing larger spikes in the second half. Once again this is likely due to the momentum being used to escape the local minima.

Model 7 – Learning Rate = 0.01 and Momentum = 1

Our final model uses a momentum of 1. As we will see in Figure 9 this is will cause the model to have extreme changes in values before eventually setting in quite a poor performing minimum. That being said, with the help of the ModelCheckpoint() function the F1-Score was 55% which we can see allows our testing confusion matrix to perform decently. It is interesting to note that of all the models we have created this one is noticeably poorer at predicting tulips while making up for its accuracy in other classes.

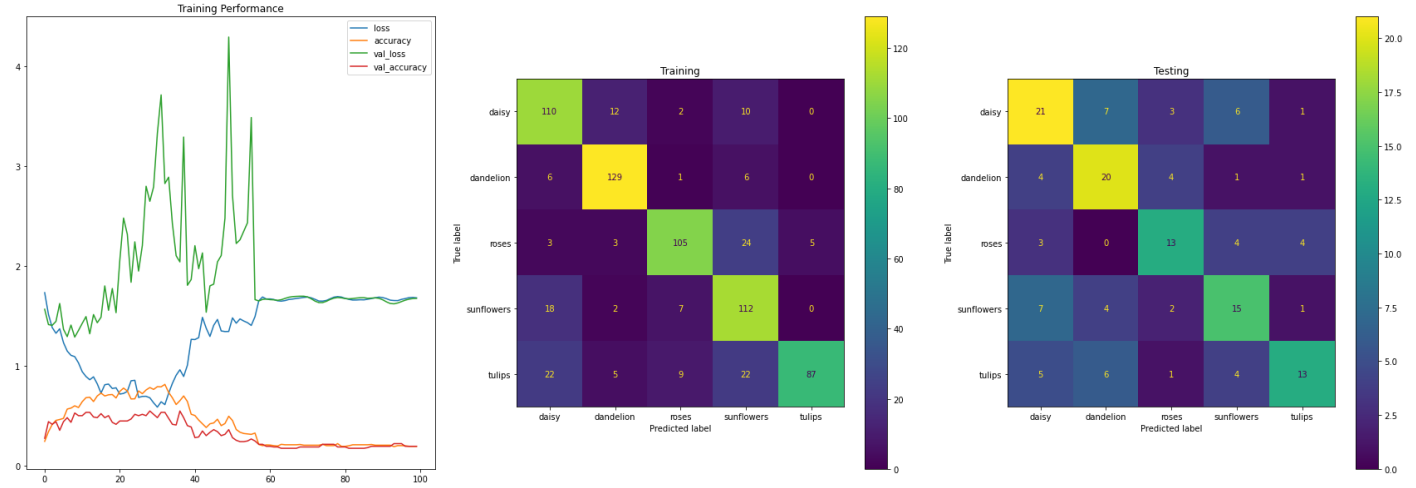

Figure 9: *Left: Loss and Accuracy over time (epochs)*
Middle: Confusion matrix on Training Data
Right: Confusion matrix on Testing

Initially, there is a downward trend for 20ish epochs however after that the model seems to use the high momentum to escape the local minima that the other models found resulting in horrible performance. Our loss stats become large while our accuracy stats seem to nosedive down to 20%. Remember that with 5 classes, the random chance is 20% which is what we are seeing here. After epoch 60 we see that the model is not able to learn anything and as such the model has failed to fit.

Note that the program also generates a classification report, where the F1-Scores were taken from, however, they were not able to fit in the report. A lot of information can be gleaned from these stats. For example, all models predict tulips with more accuracy than others with some reaching as high as 72% prediction rate. See program output for these performance stats.

Note that all these models were trained with 100 epochs despite many not reaching convergence. When experimenting with 1000 epochs it was discovered that all models (apart from model 2) reached peak accuracy within 100 epochs while also making many plots unreadable. For that reason, our Figures are using 100 epochs for the sake of clarity in our early epochs where most of our learning is done.

Recommendations

Based on the above results, the ideal parameters would be a learning rate of 0.001, and momentum of 0 with 1000 epochs (to reach convergence). Using model checkpoints the best mode would be selected. To further improve performance here are three of the most important recommendations:

The first recommendation and generally useful for all machine learning models are increasing the number of samples. Our model is trained on 1000 samples split evenly among 5 classes (200 each). This is a decent amount however most networks nowadays are training on 1000+ samples per class. This is likely the best way to increase performance and it leads nicely into the second recommendation.

The second recommendation is data augmentation. This works by taking our original corpus of data and slightly augmenting it in different ways to produce more samples. It could be a slight rotation, zoom, colour correction, image shift etc. This can lead to far more samples and there are many functions to perform this task out there like for example, Keras has “experimental.preprocessing” layers or an ImageDataGenerator function however there is much more.

The third recommendation is the data itself. Visually inspecting the data, we can clear examples where there is no flower in sight. Looking at the images below we can see these are clear examples of noise.

***Figure 10**: Some images taken from the “rose” folder that do not depict roses*

Figure 10 shows just a few images present in the roses folder, but it is prevalent in all classes. This is an issue when we already have such small sample sizes. Examples like these will have to be removed to help the model increase performance.

The final set of recommendations is a combination of more advanced neural network techniques and some that require more computing power.

- Using learning rate and momentum decay to train the model quickly at the beginning and slow down learning as the model trains more.
- Using Siamese/Triplet networks to learn an embedding to help with classification by pulling similar classes together while moving different classes apart.
- Fine-tuning more of the model to better fit our specific task.