**Flower Classification – Transfer Learning**

Group 50

Adrian Ash, Chiran Walisundara and Don Kaluarachchi

**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 accuracy predict the type of flower based on an image.

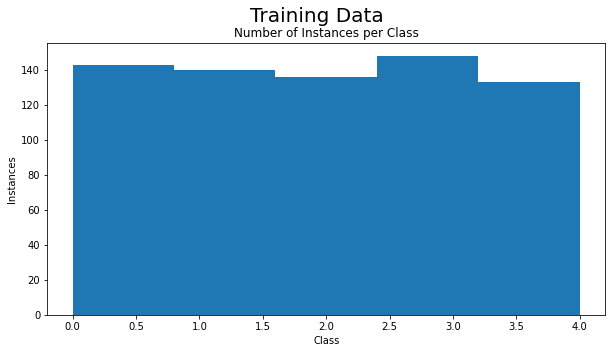
Data Pre-processing

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 hyper parameters. The test data is used at the end to determine model performance metrics. Using sklearns 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.

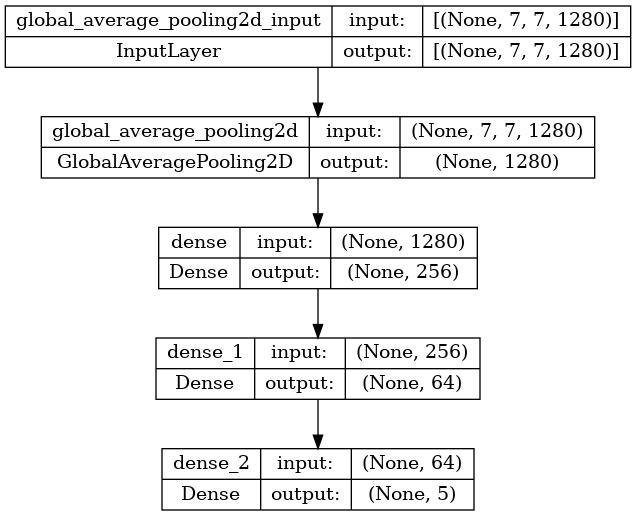
**Figure 1**: A histogram of the number of instances per class within the Training data

This is 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 use was to use the “MobileNetV2” network to convert out 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 back propagation 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 highend system to 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 the network takes the 7x7x1280 embedding in and uses a dense layer of 5 as the output, one for each class.

**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. Due to this 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 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 effects our performance it is normally a sign of a poor dataset (more about this in recommendations).

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 epoch. 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 class 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.

Chart, line chart

Description automatically generated

**Figure 3:** Train accuracy (orange), loss (blue) and Validation accuracy (red), loss (green) over 100 epochs

Text

Description automatically generated with medium confidence

As we can see from our training accuracy and loss lines over the course of the 100 epochs the model continues to overfit. After epoch roughly 60 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 is 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, we will see is due to a learning rate that is too large (and a dataset that is too small).

A picture containing text

Description automatically generated**Model 2** – Learning Rate = 0.0001 and Momentum = 0

A picture containing text

Description automatically generated**Model 3** – Learning Rate = 0.001 and Momentum = 0

**A picture containing text

Description automatically generatedModel 4** – Learning Rate = 0.1 and Momentum = 0

Text

Description automatically generated with low confidence**Model 5** – Learning Rate = 0.001 and Momentum = 0.01

A picture containing text

Description automatically generated**Model 6** – Learning Rate = 0.001 and Momentum = 0.1

**A picture containing text

Description automatically generatedModel 7** – Learning Rate = 0.01 and Momentum = 1

**Recommendations**

Recommendations

More samples

Remove noise

Data augmentation

Triple network