CS4287 Neural Computing Project

Sean Fitzgibbon - 19273444

Daniel Larkin - 19257503

Adam Butler - 19244967

Table of Contents

Overview

In this project, we create a convolutional neural network based on the Xception architecture. The convolutional neural network operates on a data set containing images of different types of flowers, and classes them accordingly. We obtained an accuracy reading of **88%** overall, which we were satisfied with. Here, we discuss the details of our findings, our network structure, and issues that arose throughout the process too.

The Data Set

The dataset we used for the CNN is a dataset of flowers, obtained from https://www.tensorflow.org/tutorials/load_data/images.
This dataset contains various images of 5 different types of flowers. Below in Figure 1 is a sample output from our program, showing sample images from each category of flower (daisy, dandelion, rose, sunflower, tulip).

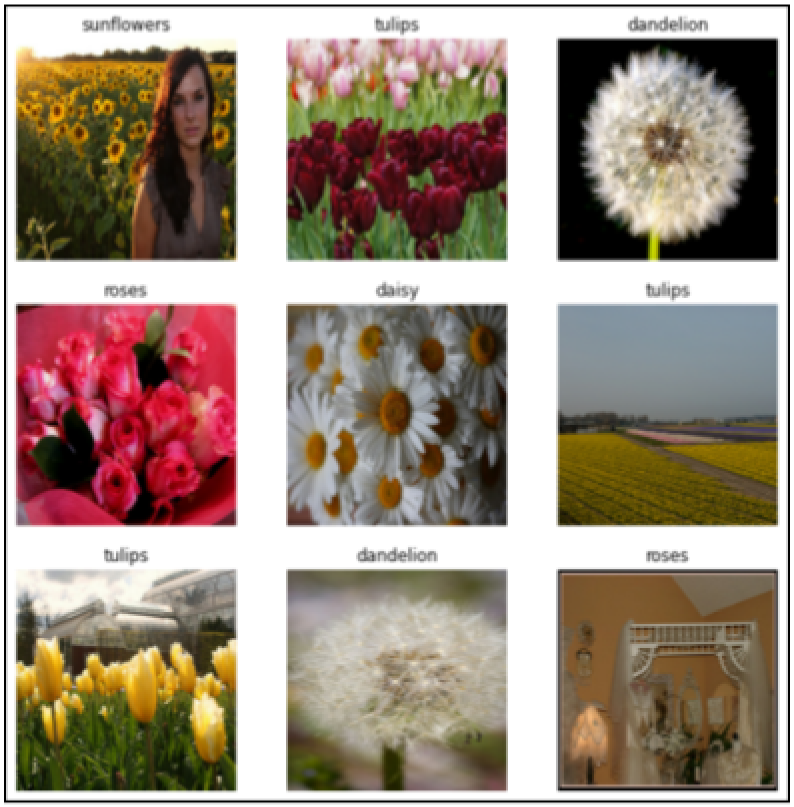
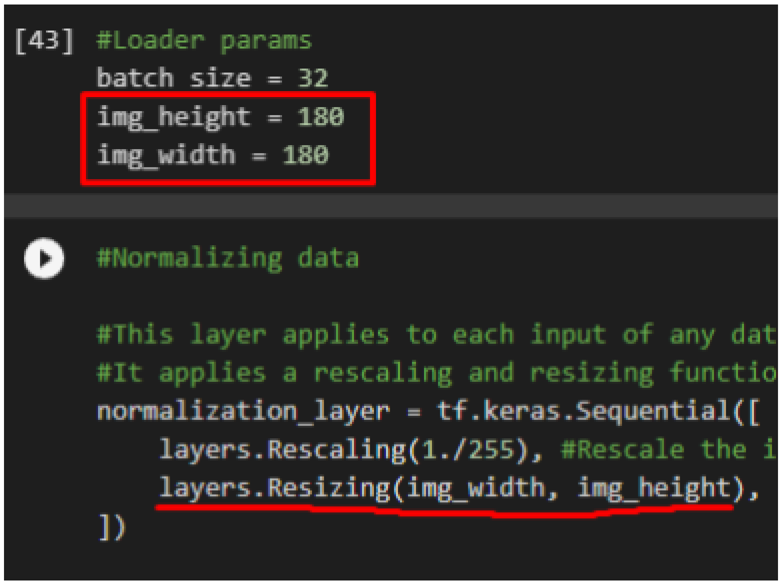


Figure 1: Dataset sample images

The data is divided into sub-directories, each labeled with their relevant label (i.e. tulip images found in ‘flowers_photos/tulips’ and so on).

We divided the dataset into a 80/10/10 split, 80% being for training, 10% for validation, and the final 10% devoted to testing of our network. This left roughly 367 images for validation, 367 for test, and 2936 images for training of the network.

The images in the dataset are all resized to 180x180 in our program, as seen in the Figure 2 code snippet below. This allowed our network to run using equal image sizes as the network uses inputs of similar sizes.



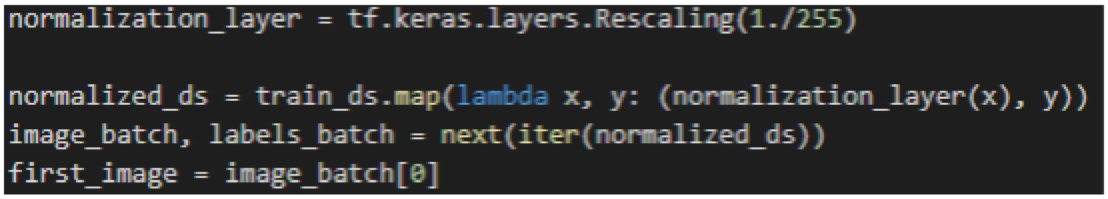
```
[43] #Loader params
     batch size = 32
     img_height = 180
     img_width = 180
```

```
#Normalizing data

#This layer applies to each input of any dat
#It applies a rescaling and resizing functio
normalization_layer = tf.keras.Sequential([
    layers.Rescaling(1./255), #Rescale the i
    layers.Resizing(img_width, img_height),
])
```

Figure 2: Resizing each image

Another pre-processing technique we utilized for our data was to normalize the RGB values of the inputs. This helps the network by reducing the range it has to operate on, and we noticed that by adding this, it made the network perform better. We found a method of doing this on the tensorflow documentation website, and is shown below in Figure 3.

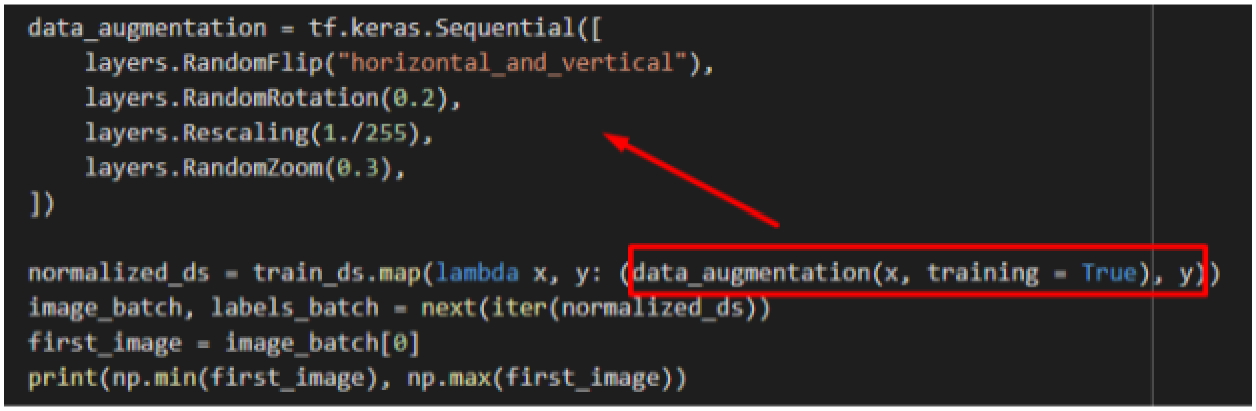


```
normalization_layer = tf.keras.layers.Rescaling(1./255)

normalized_ds = train_ds.map(lambda x, y: (normalization_layer(x), y))
image_batch, labels_batch = next(iter(normalized_ds))
first_image = image_batch[0]
```

Figure 3: Normalizing RGB values

We also found that the use of data augmentation improves performance of the network in general. We implemented data augmentation in the form of random flips, random rotations, and random zooms to the training data. This was achieved using the tensorflow.keras.layers library, using layers.RandomFlip, layers.RandomRotation, layers.Rescaling, and layers.RandomZoom. A code snippet can be seen below in Figure 4.

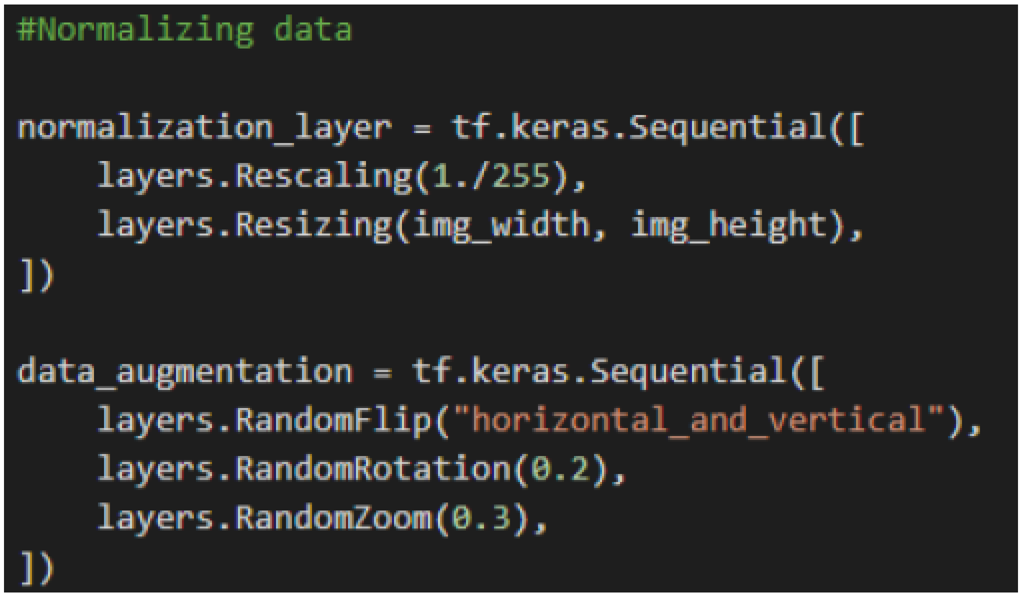


```
data_augmentation = tf.keras.Sequential([
    layers.RandomFlip("horizontal_and_vertical"),
    layers.RandomRotation(0.2),
    layers.Rescaling(1./255),
    layers.RandomZoom(0.3),
])

normalized_ds = train_ds.map(lambda x, y: (data_augmentation(x, training = True), y))
image_batch, labels_batch = next(iter(normalized_ds))
first_image = image_batch[0]
print(np.min(first_image), np.max(first_image))
```

Figure 4: Data augmentation of inputs

Later we re-structured the data augmentation following the discovery of an easier way to split our data into train, test and validation sets. This new structure can be seen below in Figure 5.

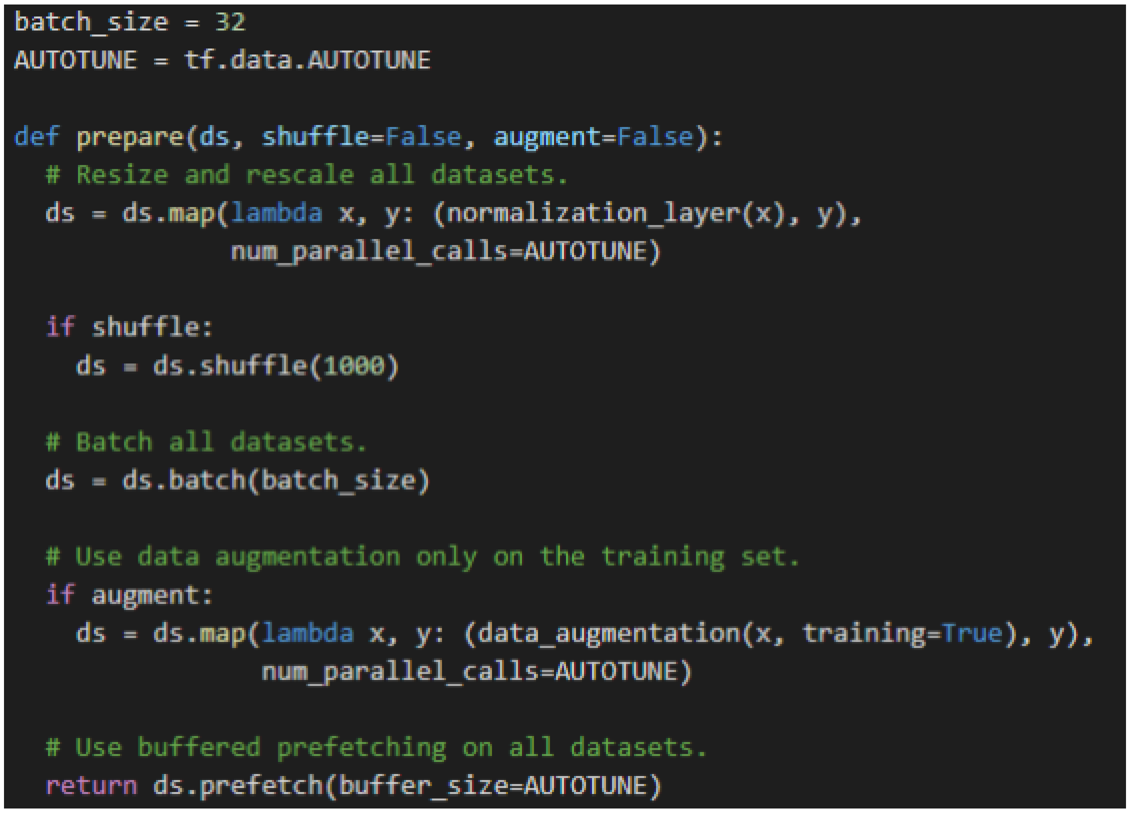


```
#Normalizing data

normalization_layer = tf.keras.Sequential([
    layers.Rescaling(1./255),
    layers.Resizing(img_width, img_height),
])

data_augmentation = tf.keras.Sequential([
    layers.RandomFlip("horizontal_and_vertical"),
    layers.RandomRotation(0.2),
    layers.RandomZoom(0.3),
])
```

Figure 5 (Continued Below)



```
batch_size = 32
AUTOTUNE = tf.data.AUTOTUNE

def prepare(ds, shuffle=False, augment=False):
  # Resize and rescale all datasets.
  ds = ds.map(lambda x, y: (normalization_layer(x), y),
              num_parallel_calls=AUTOTUNE)

  if shuffle:
    ds = ds.shuffle(1000)

  # Batch all datasets.
  ds = ds.batch(batch_size)

  # Use data augmentation only on the training set.
  if augment:
    ds = ds.map(lambda x, y: (data_augmentation(x, training=True), y),
                num_parallel_calls=AUTOTUNE)

  # Use buffered prefetching on all datasets.
  return ds.prefetch(buffer_size=AUTOTUNE)
```

Figure 5: Data Augmentation updated

Data augmentation allows us to mimic having more data than we really have by altering how images look i.e. rotating, flipping, zooming selected images and more. We opted to use resizing and rescaling for all images, called our normalization_layer. We also opted to randomFlip, randomRotation, and randomZoom our training data to efficiently train the network on different inputs which look dis-similar to the average flower photo. An example of some images which have been augmented can be seen in Figure 6 below.

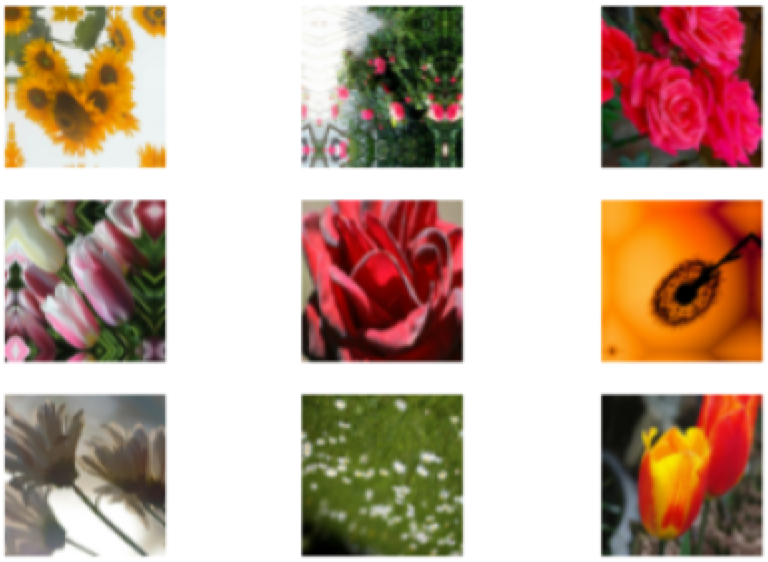


Figure 6: Augmented image examples

Notice the rotation and flipping in some images, while others are slightly zoomed in compared to their original.

The Network Structure and Hyperparameters

We based our network off of the Xception architecture, a convolutional neural network developed by Google. It is known for performing better than ResNet, Inception V3 and VGG-16, other networks with notable accuracy. It is an improved version of Inception-V3, with modified depthwise separable convolution.

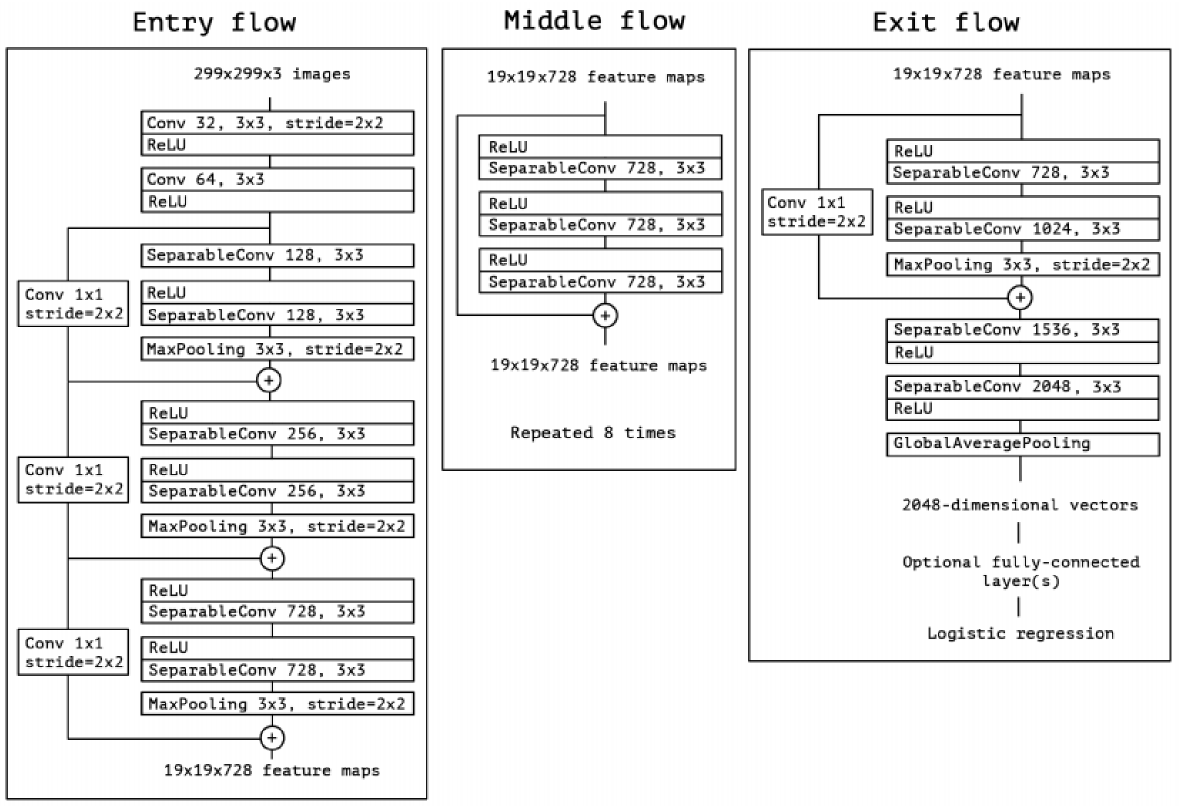


Figure 7

Depthwise Separable Convolution

Xception is an extension of the Inception architecture which replaces the standard Inception modules with depthwise separable convolutions.

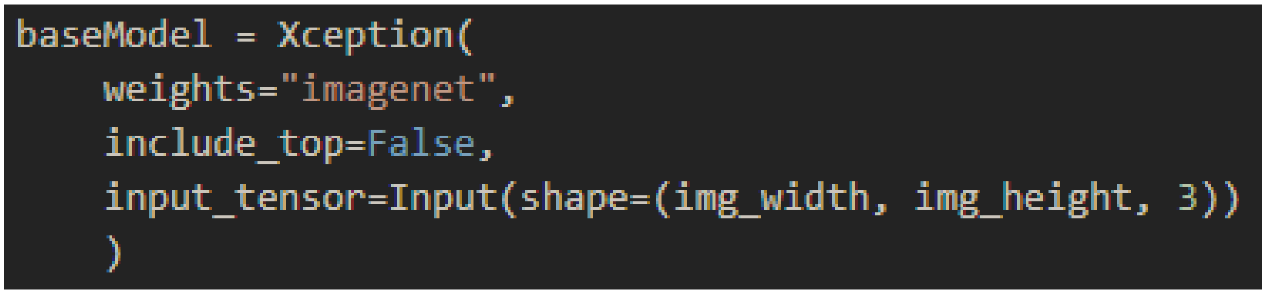
In standard convolution, the application of filters across all input channels and the combination of these values are done in a single step. Depthwise Separable Convolution on the other hand, breaks it down into two parts. The first is depthwise convolution, that is it performs the filtering stage. Then there is pointwise convolution which performs the combining stage.

Depthwise convolution applies convolution to a single input channel at a time, which is different from the standard convolution that applies it to all channels.

Transfer Learning

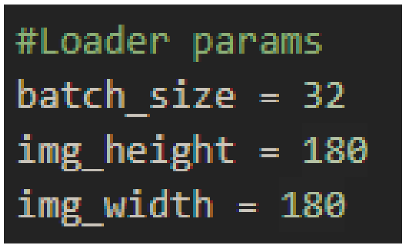
This involves taking a pretrained model and then retraining it to solve a different, but related, problem. As mentioned before we opted for a pretrained model Imagenet. By exchanging the weights that the model learned in the first task to the new task, This is a technique for manipulating a model's knowledge from one task in order to enhance generalization in another task.

It allows us to avoid the need for a lot of new data and focus on furthering our model. Using a model that has already been trained helps expedite the learning process for a new task while also improving the model's overall accuracy and performance.



```
baseModel = Xception(
    weights="imagenet",
    include_top=False,
    input_tensor=Input(shape=(img_width, img_height, 3))
    )
```

Figure 8



```
#Loader params
batch_size = 32
img_height = 180
img_width = 180
```

FIgure 9
Here we initialize our weights, using imagenet weights as we are working with images. It is beneficial as it helps the model converge in less epochs.A the base of our model was imported the weights and parameters of the model are pre-set.

Here we can also see *include_top=False,* which is where we tell the model to omit the fully-connected later at the top of the network.
We then have the *input_tensor=Input(shape=(img_width, img_height, 3))* line where we are setting the input type to match the input dimensions and channels.

We also decided to use a Dropout layer which offers a very computationally cheap and remarkably effective regularization method to reduce overfitting and improve generalization error. This works by randomly setting input units to 0 with a frequency of our choosing at each step during training time. We decided to set it to a 20% rate.

We added a Dense layer to our model which implements the operation: *output = activation(dot(input, kernel) + bias.* It is used to classify images based on output from convolutional layers.

Batch Normalisation

We utilized the prepackaged batch normalization that is implemented in Xception. We tested implementing our own batch normalization but the results were not good.

Hyperparameters

Dropout: As mentioned before the dropout rate allows us to reduce overfitting and improves generalization.
Normalization layer: this was used to rescale the image from RGB 0-255 to RGB 0-1 and also resize the image to specified image width and image height
Pooling: this reduces the dimensions of the feature maps, thus reducing the number of parameters to learn which then makes the network less computation heavy.
Epochs: this is set to how many times we want to cycle the training with all the training data
Activation Function: Responsible for activating the node or output for the input by converting the node's weighted input total into an output.

The Cost/Loss/Error/Objective Function

The function we used was the *Sparse Categorical Cross Entropy Loss Function.* We decided on this as it is mainly used for classification problems, where the output can belong to one of a discrete set of classes.

We chose this over the Mean Squared Error Loss Function, as we learned that it is more appropriate for regression problems, where the output is a numerical value. MSE doesn't punish misclassifications enough but is the right loss for regression, where the distance between two values that can be predicted is small.

The Optimiser

Initially we started out using the Stochastic Gradient Descent optimiser algorithm for our model, in an attempt to achieve the most optimal parameter settings that give the best fit between our predicted and actual outputs. The library we have referenced, Tensorflow, conveniently had a built-in implementation of a gradient descent algorithm. However, in the end we decided not to use the standard gradient descent algorithm, in the hopes that another may prove to be more efficient.

We switched to using the Adam optimiser to reduce the model's runtime and to somewhat increase the accuracy of the model. Its ease of implementation also made it an attractive switch. The Adam optimiser computes individual learning rates for the various parameters of the model, making it an "adaptive learning rate method" optimiser. The Adam optimiser is a very memory efficient algorithm, which may have allowed it to have a better use of the system's memory which caused it to run notably faster.

Cross Fold Validation

Initially we were only able to split our data across an 80/20 split, with eighty percent of the data being for training and twenty per cent being for validation. Our goal was to have a test set of ten per cent of the data and a validation set of ten per cent of the data. We used this 3-fold validation as we wanted to have a validation set in place to make sure the model was functioning during training and so that we could compare its accuracy with our test accuracy results. After we had gotten the 3-fold validation working and fixed our initial overfitting issue, we considered implementing k-fold cross validation, specifically 5-fold cross validation, where there would be an even split among four training sets and a fifth set for testing. We opted not to continue with this method as it would not have included the validation set, which we wanted to use, to be more confident of our model's accuracy.

Results

The plot below in FIGURE 10 shows the plotted history of the accuracy of both training and validation data sets, labeled as 'train' and 'test' as seen in the upper left corner. This plot is from a previous version of our program, in which we loaded the data differently and augmented images in a different way. We were also initially using vanilla SGD (stochastic gradient descent), which produced satisfactory results, but we found the next result to be sufficient while also running faster. From this plot, we can see that our model is producing accurate results, leveling off sufficiently once a few epochs have been completed.

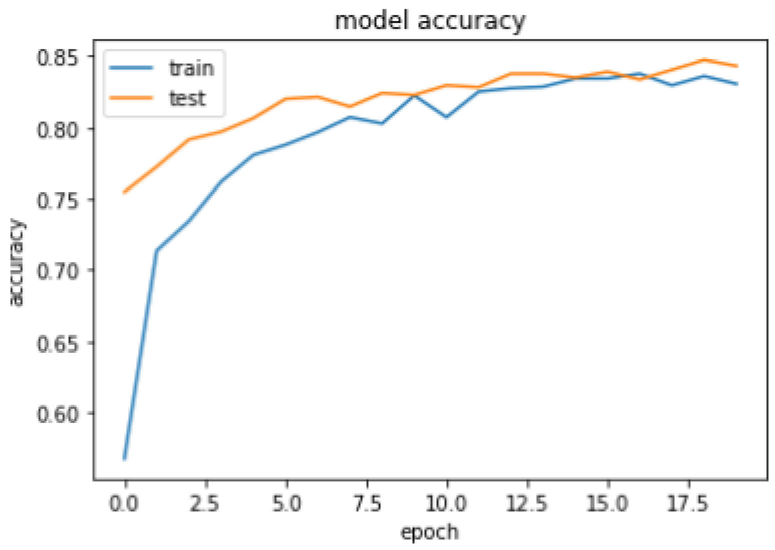


Figure 10

In the next plot seen in FIGURE 11, our final model's results are visualized for train and validation accuracy.

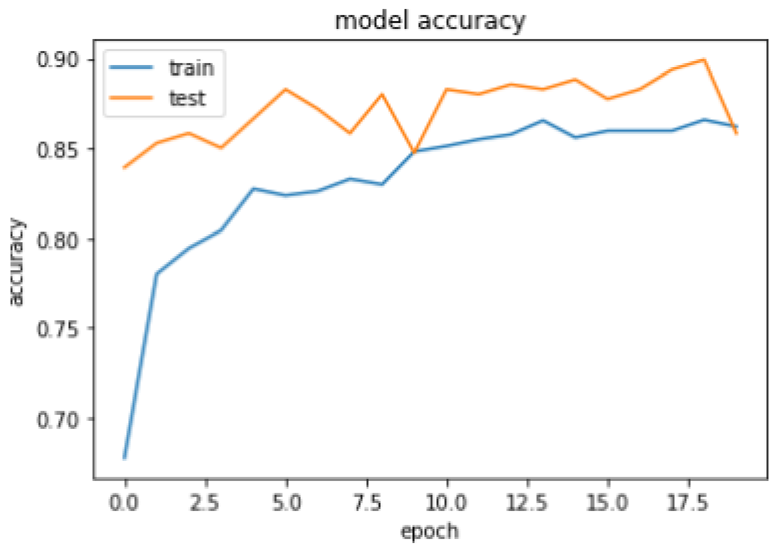


Figure 11: Model Accuracy

In this plot seen in Figure 11, we see our validation data maintaining a solid accuracy level, albeit fluctuating throughout the network's iteration. It eventually somewhat levels off with the training accuracy, which is the result we were looking for. This result differs from Figure 10 because we decided to load our images from the dataset in a different way, allowing us to add a test set which we initially did not have (originally used 80/20 split for train/validation sets). We also switched to using the ‘Adam’ optimiser in the end, which resulted in a slightly more accurate model with a faster runtime.

The plot below in Figure 12 shows the original model loss of our network, using the SGD function and loading data with only train/val splits. We noticed that the validation loss is ultimately smaller than the train loss, which is a good result.

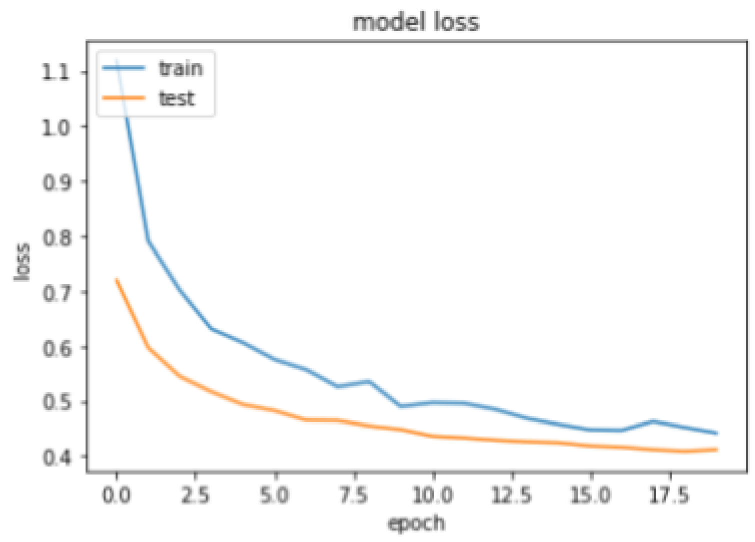


Figure 12

The next plot in Figure 13 shows our final model loss of the network, using the Adam optimiser and loading data with the train/val/test split of 80/10/10. We also augmented the images differently.

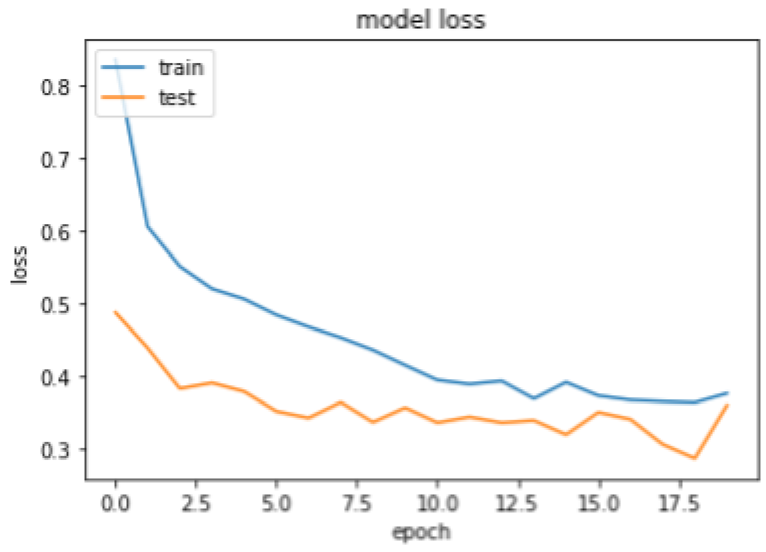
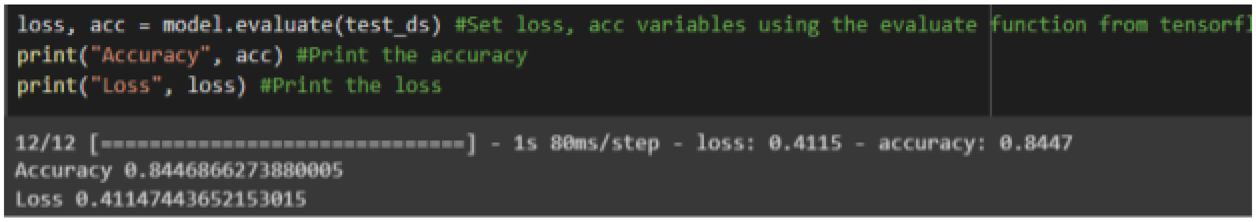


Figure 13: Model Loss

We noticed the validation set loss fluctuated quite a bit, similar to how it fluctuates for the final model accuracy in Figure 11. The loss remains low, however, on the validation set and only slightly higher on average for the training set.

Finally, we ran a model.evaluate() function to evaluate the accuracy and loss of our model based on our test set, which contained images that the network has not yet seen. We saw a final accuracy of around 0.84 -> 84%, with a loss of around 0.41, as seen in Figure 14 below.



```
loss, acc = model.evaluate(test_ds) #Set loss, acc variables using the evaluate function from tensorfl
print("Accuracy", acc) #Print the accuracy
print("Loss", loss) #Print the loss
```

```
12/12 [==============================] - 1s 80ms/step - loss: 0.4115 - accuracy: 0.8447
Accuracy 0.8446866273880005
Loss 0.41147443652153015
```

Figure 14: Model Evaluation

Evaluation of the Results

As we can see from the plots above, there is some element of overfitting present in our final model accuracy (train and valid using Adam optimizer) in Figure 11. The validation plot fluctuates as the network progresses, which should not happen as heavily as it is in our network. We attempted to fix this in the next section with dropout and more, but we could not find a solution in time.

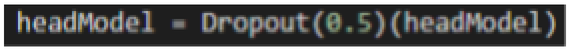
The cause of this is that the network generalizes well with the training data but fails to generalize properly for the validation set, which causes some reductions in accuracy as the network runs through the data/epochs.

We can also see that the loss differs between the use of SGD and Adam, with Figure Z1 and Z2 respectively showing SGD and Adam optimizers (above). The loss in Figure Z2 using Adam was also fluctuating more than when we used SGD, and spikes abnormally towards the last few epochs. This could be the result of a large learning rate, or even something like too small of a batch size.

Impact of Varying Hyperparameters

Varying Dropout

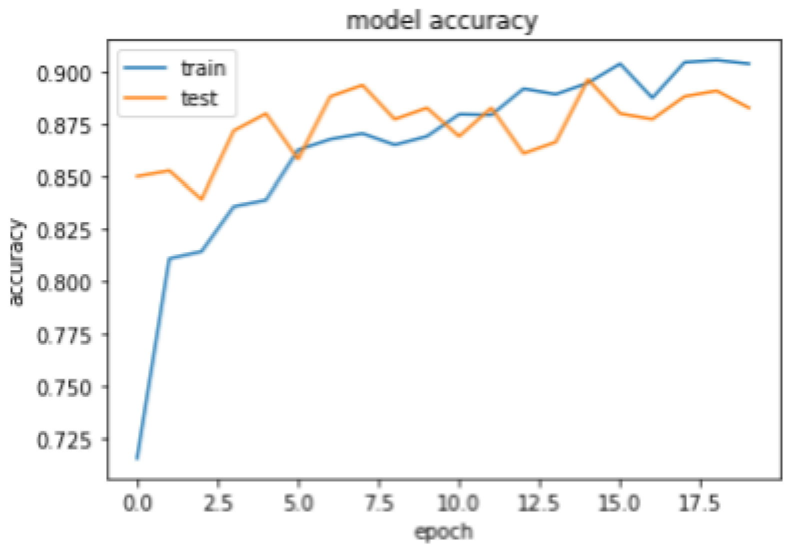
Dropout is included in our model definition as seen below in Figure 15. This layer helps to improve overfitting and generalization, and can improve performance in a network. Our initial figure was set to 0.2. We investigated the results of changing it to 0.4.



```
headModel = Dropout(0.5)(headModel)
```

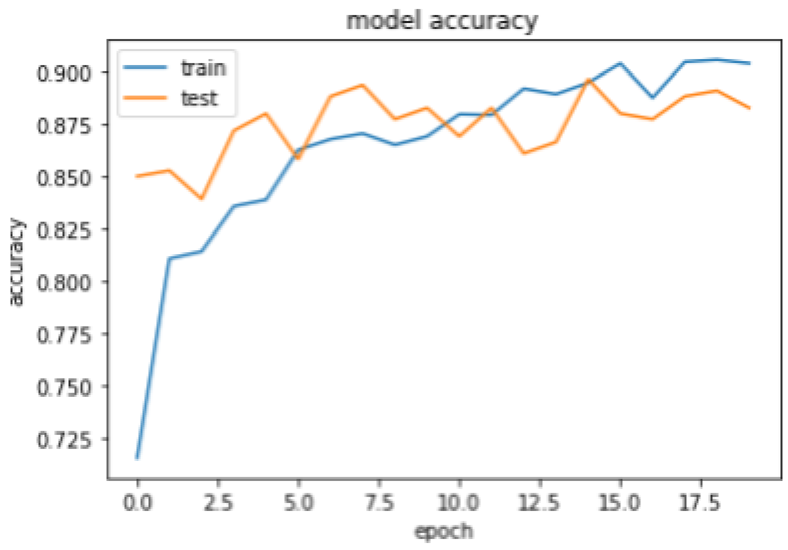
Figure 15: Dropout

When we altered the value to 0.4, we ran the model and analyzed the results. Below in Figure 16 we can see the value of accuracy, loss from our original figure of 0.2 for dropout.



Dropout(0.2)

Figure 16



Dropout(0.4)

Figure 17

<u>Evaluation of model accuracy/loss with 0.2 dropout</u>

```
12/12 [==============================] - 1s 89ms/step - loss: 0.4070 -
accuracy: 0.8447
Accuracy 0.8446866273880005
Loss 0.40701988339424133
```

<u>Evaluation of model accuracy/loss with 0.4 dropout</u>

```
12/12 [==============================] - 1s 85ms/step - loss: 0.3804 -
accuracy: 0.8828
Accuracy 0.8828337788581848
Loss 0.38035067915916443
```

We can see that with 0.4 dropout the accuracy was slightly higher with a lower loss.

Varying <u>Epochs</u>

Next we tried different amounts of epochs, the amount of times our network iterates. Our original value of 20 works well from what we saw, so we began by reducing it to 10 to analyze the impact on accuracy and loss overall.

20 epochs
Accuracy 0.8828337788581848
Loss 0.38035067915916443

10 epochs
Accuracy 0.8692098259925842
Loss 0.36360156536102295

We can see a slight drop in accuracy, but also a slight reduction in the loss figure when we reduce the number of epochs.

Now we try increasing the epochs to 25.

25 epochs

Accuracy 0.8692098259925842
Loss 0.4331775903701782

We noticed the accuracy remained the same, but the loss had increased by a notable amount. We opted to keep the program at around 20 epochs for the best performance we could get.

Varying Loss Function
We opted to sparse categorical cross entropy as our loss function, as we found our results were a lot better than using MSE with our network.

Sparse Categorical Cross-Entropy

Accuracy 0.8828337788581848

Loss 0.38035067915916443

MSE (Mean Squared Error)

Accuracy 0.20435968041419983
Loss 5.438910484313965

We can see that Mean Squared Error does not suit our data/network structure, as the accuracy is very bad compared to sparse-categorical cross entropy. The loss is also extremely high, which is not acceptable. Upon discovering Sparse Categorical Cross Entropy, we tested it with our network and ultimately remained with this loss function.

References

Primo.ai. (2020). *Objective vs. Cost vs. Loss vs. Error Function - PRIMO.ai*. [online] Available at: http://primo.ai/index.php?title=Objective_vs._Cost_vs._Loss_vs._Error_Function [Accessed 5 Nov. 2022].

Rosebrock, A. (2016). *ImageNet classification with Python and Keras - PyImageSearch*. [online] PyImageSearch. Available at: https://pyimagesearch.com/2016/08/10/imagenet-classification-with-python-and-keras/ [Accessed 5 Nov. 2022].

Shacklett, M.E. (2021). *dropout*. [online] SearchEnterpriseAI. Available at: https://www.techtarget.com/searchenterpriseai/definition/dropout [Accessed 5 Nov. 2022].

Ajinkya Jawale (2019). *Recognize Flowers using Transfer Learning - Ajinkya Jawale - Medium*. [online] Medium. Available at: https://medium.com/@ajinkyajawale/recognize-flowers-using-transfer-learning-26c2188c50ba [Accessed 5 Nov. 2022].

Deepanshi (2021). *Convolutional Neural Network with Implementation in Python*. [online] Analytics Vidhya. Available at: https://www.analyticsvidhya.com/blog/2021/08/beginners-guide-to-convolutional-neural-network-with-implementation-in-python/ [Accessed 5 Nov. 2022].

and, L. (2022). *Load and preprocess images | TensorFlow Core*. [online] TensorFlow. Available at: https://www.tensorflow.org/tutorials/load_data/images [Accessed 5 Nov. 2022].

Data augmentation (2022). *Data augmentation | TensorFlow Core*. [online] TensorFlow. Available at: https://www.tensorflow.org/tutorials/images/data_augmentation [Accessed 5 Nov. 2022].

Cross (2018). *Cross Entropy vs. Sparse Cross Entropy: When to use one over the other*. [online] Cross Validated. Available at: https://stats.stackexchange.com/questions/326065/cross-entropy-vs-sparse-cross-entropy-when-to-use-one-over-the-other [Accessed 5 Nov. 2022].

Park, S. (2021). *A 2021 Guide to improving CNNs-Optimizers: Adam vs SGD*. [online] Medium. Available at: https://medium.com/geekculture/a-2021-guide-to-improving-cnns-optimizers-adam-vs-sgd-495848ac6008 [Accessed 5 Nov. 2022].