REPORT WRITING

ON

IMAGE RECOGNITION TO IDENTIFY SPECIES OF FLOWERS

Artificial Intelligence and Data sciences

Introduction

The journey of Artificial Intelligence has upgraded over time with the Indepth improvement in the world of image recognition systems. One breakthrough is the identification of flowers using deep learning convolutional neural networks as a means solving such a vital problem in the world of technology. As we explore the large-scale ecosystem of botany, the traditional way of identifying flowers will be eradicated using a well-tested technique to identify each flower.

In this journey, this gives a huge pathway to revolutionize the floral kingdom using the power of computer vision and advanced machine learning algorithms to detect patterns, shape, colors available in a flower image.

AIM

The objective of this project is to apply usage of images to recognize varied species of image.

The Data Process

Data collection is a method of accumulating and measuring information based on interest in a well-defined manner. This helps to answer research questions, measure outcomes, and test hypotheses.

Type of Data collection

- Qualitative Data: These are not numerical data; there are descriptive data, mostly words.
- Quantitive Data: These are numerical data, there are data inform of numbers. E.g. 1,2,3
- Mixed Data: These are data that consist of both numerical data and data inform of words.

Types of Data

- Primary Data: These is a type of data acquired directly from the source. The real reason for this collection is to get feedback that is peculiar and unique to your research. Examples are listed as follows; Observations, Questionnaires, Interviews, Surveys, and Case-studies.
- Secondary Data: This is a type of data acquired and used for a purpose.

Data collection

Based on the breakdown of data above, the type of data used in this project is **Secondary Data.** Because the data is gotten from the car sales database source and the type of data collection is a **mixed method** because it contains both the quantitative and qualitative methods.

Tools/Libraries used in this project

Data provided for an image classification project, which means it cannot be explored using spreadsheets.

Python programming language was used for the data science process. This language has libraries that perform this process.

The libraries used are as follows;

- NumPy
- Matplotlib
- Keras

NumPy: is a fundamental library that is used for numerical computation. It has an N-dimensional array.

Matplotlib: is a particularly good library that is used for creating amazing visualizations. It is a plotting library for the Python programming language.

Keras: is a deep learning library that is used for deep learning procedures. This library has all the algorithms needed for your deep learning processes.

TensorFlow: is a deep learning built as a backend as subsidiary of the keras library.

Data Description

The dataset contains images of flowers from the TensorFlow data flower image directory.

Data Acquisition

This is a method of acquiring data into a system which will have a different format e.g. csv, Json, xlsx

Data Acquisition

This is the stage where you bring in data in your desired format.

```
In [23]: #downloading and extraction of the flower data from this url
         dataset_url = "https://storage.googleapis.com/download.tensorflow.org/example_images/flower_photos.tgz"
         archive = tf.keras.utils.get_file(origin=dataset_url, extract=True)
         data_dir = pathlib.Path(archive).with_suffix('')
```

```
In [24]: dataset_url
Out[24]: 'https://storage.googleapis.com/download.tensorflow.org/example_images/flower_photos.tgz'
```

```
In [25]: #checking the lenght of the data
         image_count = len(list(data_dir.glob('*/*.jpg')))
         print(image_count)

         3670
```

Viewing the Images

```
In [26]: #Checking out the first image in the data
         roses = list(data_dir.glob('roses/*'))
         PIL.Image.open(str(roses[0]))
Out[26]:
```

Loading the data

There are 3670 thousand images in the in the downloaded data

specifying the width and the height of the image

```
In [27]: batch_size = 32
         img_height = 180
         img_width = 180
```

extracting my training data

```
In [28]: train_ds = tf.keras.preprocessing.image_dataset_from_directory(
           data_dir,
           validation_split=0.2,
           subset="training",
           seed=123,
           image_size=(img_height, img_width),
           batch_size=batch_size)

         Found 3670 files belonging to 5 classes.
         Using 2936 files for training.
```

extracting my validation data

```
In [29]: val_ds = tf.keras.preprocessing.image_dataset_from_directory(
           data_dir,
           validation_split=0.2,
           subset="validation",
           seed=123,
           image_size=(img_height, img_width),
           batch_size=batch_size)

         Found 3670 files belonging to 5 classes.
         Using 734 files for validation.
```

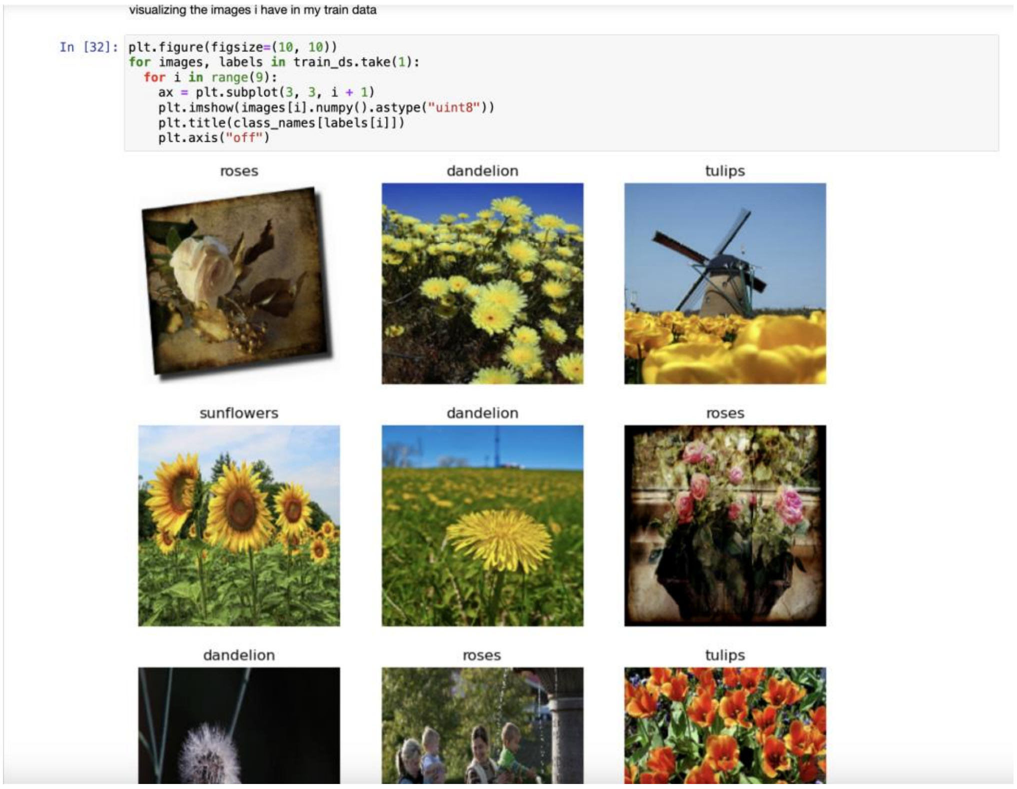
check to verify my class names

```
In [30]: #checking your classnames
         class_names = train_ds.class_names
         print(class_names)

         ['daisy', 'dandelion', 'roses', 'sunflowers', 'tulips']
```

checking the length of my class names

```
In [31]: num_classes = len(train_ds.class_names)
         num_classes

Out[31]: 5
```

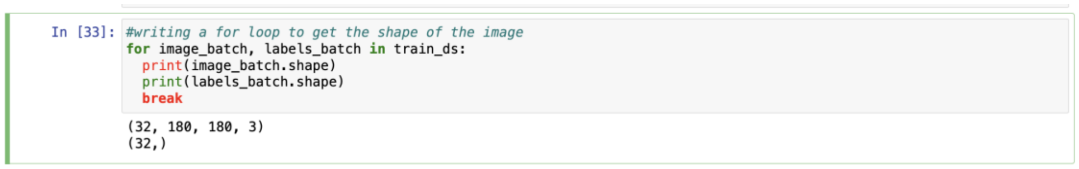
visualizing the images i have in my train data



visualizing the images i have in my train data

```
In [32]: plt.figure(figsize=(10, 10))
         for images, labels in train_ds.take(1):
           for i in range(9):
             ax = plt.subplot(3, 3, i + 1)
             plt.imshow(images[i].numpy().astype("uint8"))
             plt.title(class_names[labels[i]])
             plt.axis("off")
```

Getting the shape of the image



```
In [33]: #writing a for loop to get the shape of the image
         for image_batch, labels_batch in train_ds:
           print(image_batch.shape)
           print(labels_batch.shape)
           break

         (32, 180, 180, 3)
         (32,)
```

Building a CNN Model

Data Preprocessing and Data Augumentation

Constructor Stage

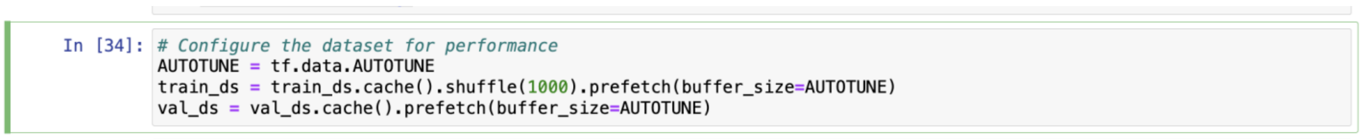
Compilation Stage

Training Stage

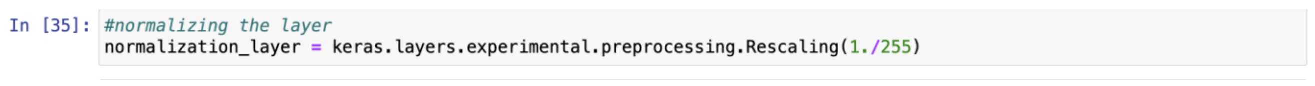
Visualization Stage

Evaluation stage

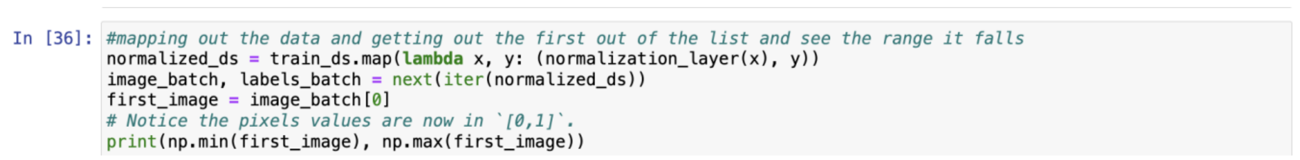
Data Preprocessing



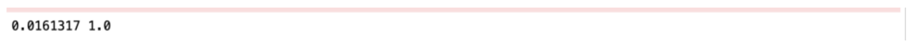
```
In [34]: # Configure the dataset for performance
         AUTOTUNE = tf.data.AUTOTUNE
         train_ds = train_ds.cache().shuffle(1000).prefetch(buffer_size=AUTOTUNE)
         val_ds = val_ds.cache().prefetch(buffer_size=AUTOTUNE)
```



```
In [35]: #normalizing the layer
         normalization_layer = keras.layers.experimental.preprocessing.Rescaling(1./255)
```

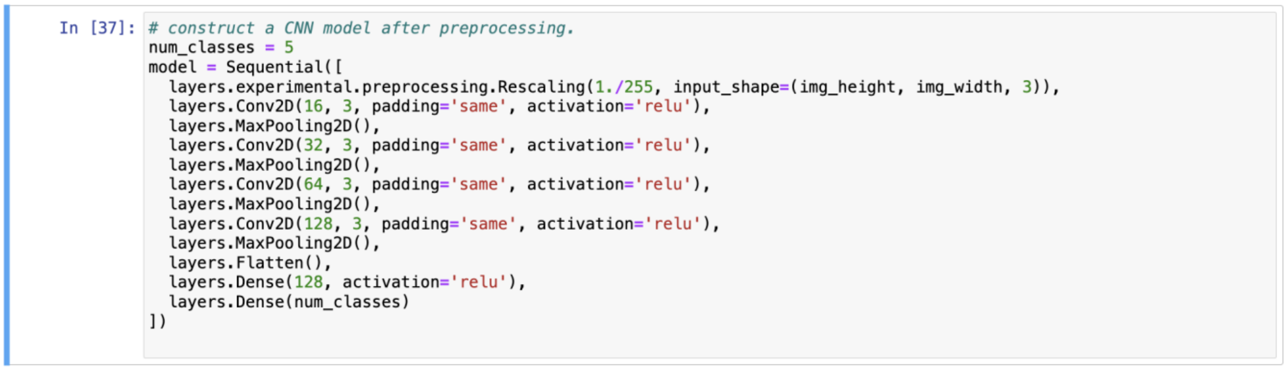


```
In [36]: #mapping out the data and getting out the first out of the list and see the range it falls
         normalized_ds = train_ds.map(lambda x, y: (normalization_layer(x), y))
         image_batch, labels_batch = next(iter(normalized_ds))
         first_image = image_batch[0]
         # Notice the pixels values are now in `[0,1]`.
         print(np.min(first_image), np.max(first_image))
```



```
0.0161317 1.0
```

Constructor Stage


```
In [37]: # construct a CNN model after preprocessing.
         num_classes = 5
         model = Sequential([
           layers.experimental.preprocessing.Rescaling(1./255, input_shape=(img_height, img_width, 3)),
           layers.Conv2D(16, 3, padding='same', activation='relu'),
           layers.MaxPooling2D(),
           layers.Conv2D(32, 3, padding='same', activation='relu'),
           layers.MaxPooling2D(),
           layers.Conv2D(64, 3, padding='same', activation='relu'),
           layers.MaxPooling2D(),
           layers.Conv2D(128, 3, padding='same', activation='relu'),
           layers.MaxPooling2D(),
           layers.Flatten(),
           layers.Dense(128, activation='relu'),
           layers.Dense(num_classes)
         ])
```

Describe the architecture of the CNN model that you used (for example, the number and types of layers that you used, the activation functions that you used etc.), and discuss your justifications for the choices that you made.

Based on the model given above is based on a series of convolutional layers with max pooling for feature extraction, followed by two fully connected layers for classification. The output layer has as many neurons as there are classes, and the network is designed for a classification task with 5 classes. The ReLU activation function is used throughout, except for the output layer. The Rescaling layer at the beginning is used to normalize pixel values in the input images.

Rescaling layers: rescaling the layer is normal in the preprocessing stage ranging from 0 to 1 is a normal preprocessing step;

Convolutional blocks: The convolutional blocks consist of Conv2D layers followed by MaxPooling2D layers. This architecture is a common pattern in CNNs for feature extraction. The Conv2D layers with ReLU activation learn spatial hierarchies of features, while the MaxPooling2D layers reduce spatial dimensions, capturing the most valuable information and making the model more computationally efficient.

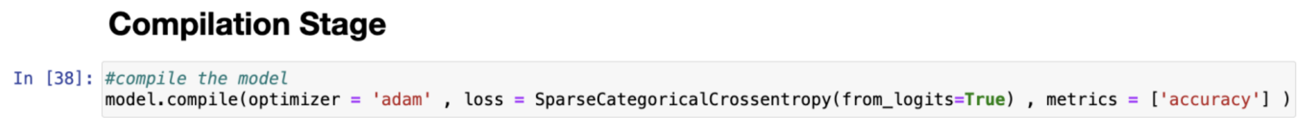
Flatten Layer: After the convolutional and pooling layers, a Flatten layer is used to transform the high-dimensional feature maps into a one-dimensional vector. This is necessary before transitioning to the fully connected layers, as they require a flat input.

Fully Connected (Dense) Layers: The two dense layers at the end are responsible for combining the learned features and making final predictions. The first dense layer with 128 neurons and ReLU activation introduces non-linearity and helps in learning more complex representations. The output layer has as many neurons as there are classes in the classification task (5 in this case) and does not have an explicit activation function, which is suitable for a multi-class classification problem.

Choice of Activation Function (ReLU): Rectified Linear Unit (ReLU) activation is used in the convolutional and dense layers (except for the output layer). ReLU is a common choice for introducing non-linearity in neural networks, allowing the model to learn complex patterns. It has been widely adopted due to its simplicity and effectiveness in training deep networks.

Output Layer: The output layer has a few neurons equal to the number of classes in the classification task (5 in this case). The lack of an explicit activation function in the output layer implies a linear activation, suitable for regression or multi-class classification tasks.

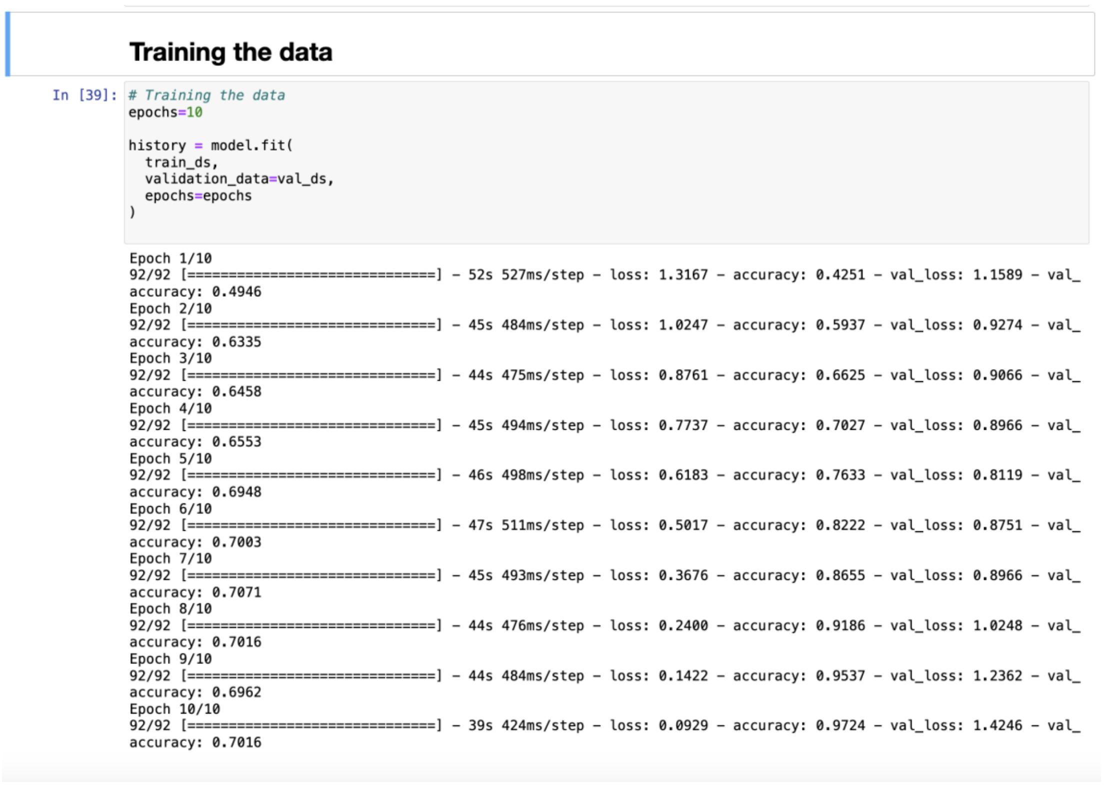
Compilation Stage



Compilation Stage

```
In [38]: #compile the model
         model.compile(optimizer = 'adam' , loss = SparseCategoricalCrossentropy(from_logits=True) , metrics = ['accuracy'] )
```

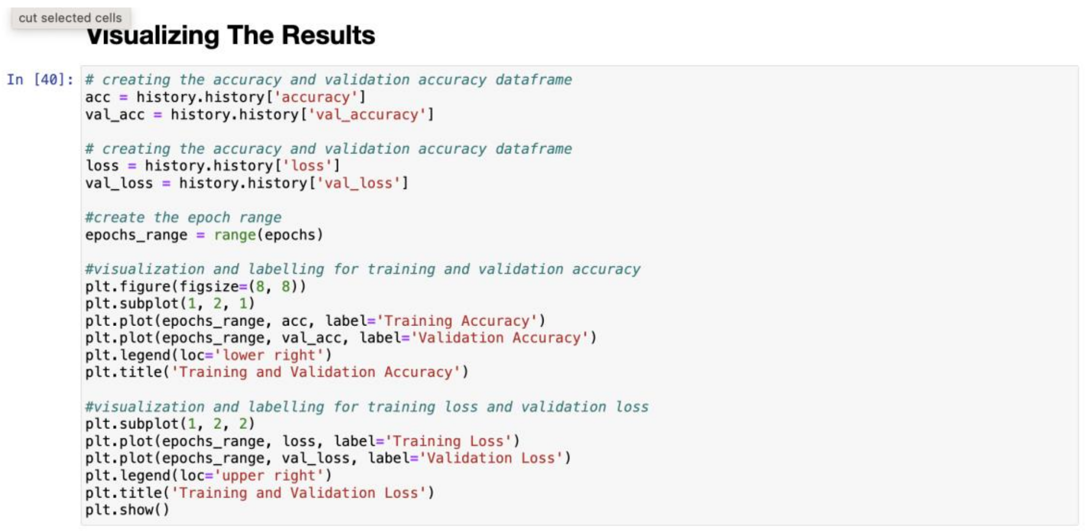
Training Stage



Training the data

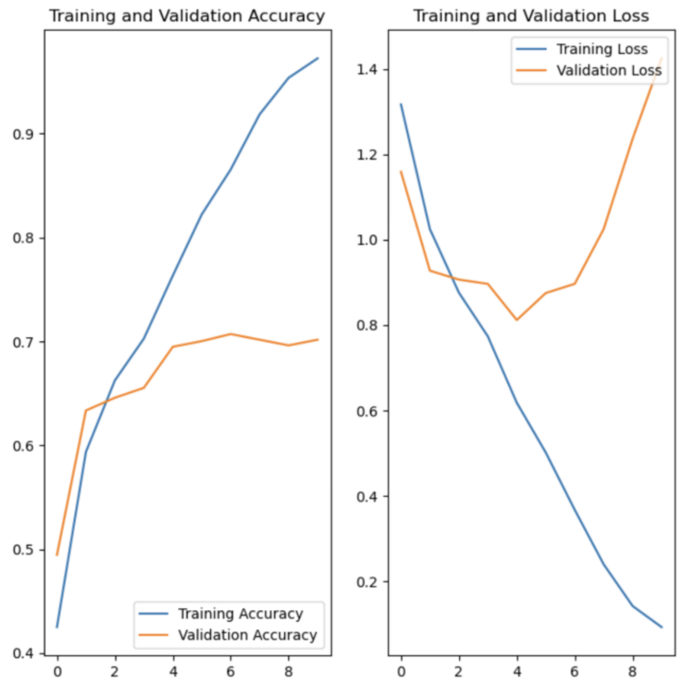
```
In [39]: # Training the data
         epochs=10

         history = model.fit(
           train_ds,
           validation_data=val_ds,
           epochs=epochs
         )
```

```
Epoch 1/10
92/92 [==============================] - 52s 527ms/step - loss: 1.3167 - accuracy: 0.4251 - val_loss: 1.1589 - val_
accuracy: 0.4946
Epoch 2/10
92/92 [==============================] - 45s 484ms/step - loss: 1.0247 - accuracy: 0.5937 - val_loss: 0.9274 - val_
accuracy: 0.6335
Epoch 3/10
92/92 [==============================] - 44s 475ms/step - loss: 0.8761 - accuracy: 0.6625 - val_loss: 0.9066 - val_
accuracy: 0.6458
Epoch 4/10
92/92 [==============================] - 45s 494ms/step - loss: 0.7737 - accuracy: 0.7027 - val_loss: 0.8966 - val_
accuracy: 0.6553
Epoch 5/10
92/92 [==============================] - 46s 498ms/step - loss: 0.6183 - accuracy: 0.7633 - val_loss: 0.8119 - val_
accuracy: 0.6948
Epoch 6/10
92/92 [==============================] - 47s 511ms/step - loss: 0.5017 - accuracy: 0.8222 - val_loss: 0.8751 - val_
accuracy: 0.7003
Epoch 7/10
92/92 [==============================] - 45s 493ms/step - loss: 0.3676 - accuracy: 0.8655 - val_loss: 0.8966 - val_
accuracy: 0.7071
Epoch 8/10
92/92 [==============================] - 44s 476ms/step - loss: 0.2400 - accuracy: 0.9186 - val_loss: 1.0248 - val_
accuracy: 0.7016
Epoch 9/10
92/92 [==============================] - 44s 484ms/step - loss: 0.1422 - accuracy: 0.9537 - val_loss: 1.2362 - val_
accuracy: 0.6962
Epoch 10/10
92/92 [==============================] - 39s 424ms/step - loss: 0.0929 - accuracy: 0.9724 - val_loss: 1.4246 - val_
accuracy: 0.7016
```



Visualizing The Results

```
In [40]: # creating the accuracy and validation accuracy dataframe
         acc = history.history['accuracy']
         val_acc = history.history['val_accuracy']

         # creating the accuracy and validation accuracy dataframe
         loss = history.history['loss']
         val_loss = history.history['val_loss']

         #create the epoch range
         epochs_range = range(epochs)

         #visualization and labelling for training and validation accuracy
         plt.figure(figsize=(8, 8))
         plt.subplot(1, 2, 1)
         plt.plot(epochs_range, acc, label='Training Accuracy')
         plt.plot(epochs_range, val_acc, label='Validation Accuracy')
         plt.legend(loc='lower right')
         plt.title('Training and Validation Accuracy')

         #visualization and labelling for training loss and validation loss
         plt.subplot(1, 2, 2)
         plt.plot(epochs_range, loss, label='Training Loss')
         plt.plot(epochs_range, val_loss, label='Validation Loss')
         plt.legend(loc='upper right')
         plt.title('Training and Validation Loss')
         plt.show()
```

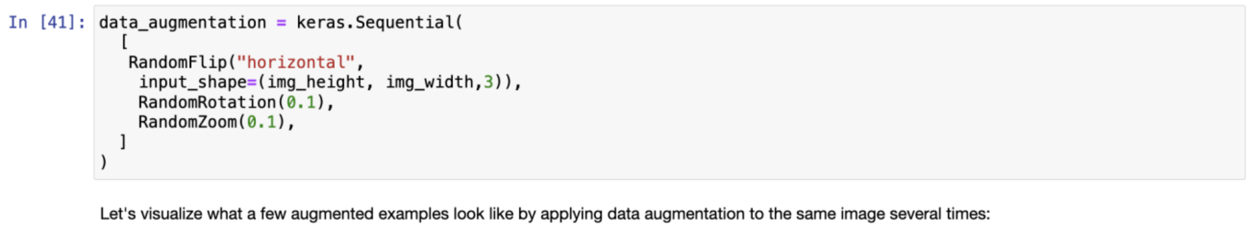


Overfitting:

The training accuracy steadily increases throughout the depicted plots, while the validation accuracy plateaus at around 60%. Notably, there is a distinct gap between the two measures, indicating potential overfitting. When there is a limited amount of training data, it is possible for the model to learn from irrelevant or extraneous information, hindering its ability to perform well on new data. This phenomenon, referred to as overfitting, presents a challenge for the model to effectively generalize to new datasets.

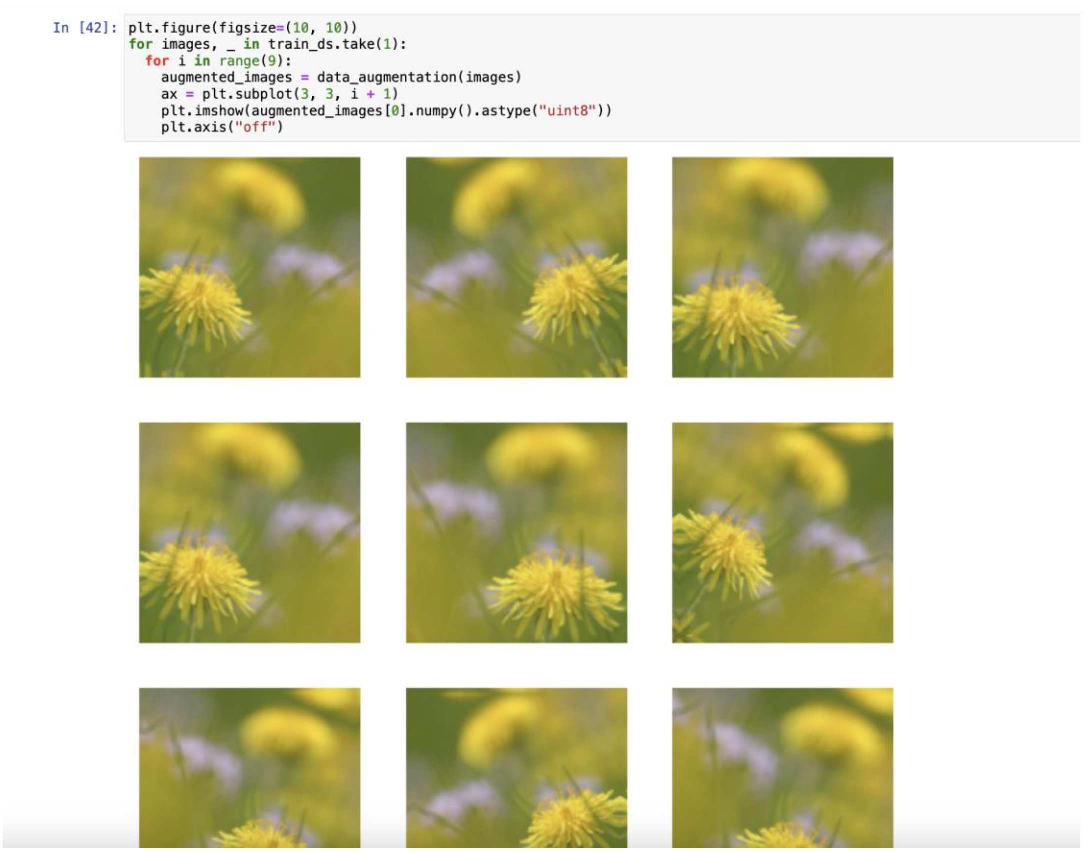
Data Augumentation

The occurrence of overfitting is closely tied to a limited number of training examples. To combat this issue, data augmentation employs a clever tactic of expanding your current examples by applying random transformations that produce realistic images. By doing so, the model is exposed to a wider range of data and can better generalize.



```
In [41]: data_augmentation = keras.Sequential(
           [
            RandomFlip("horizontal",
             input_shape=(img_height, img_width,3)),
             RandomRotation(0.1),
             RandomZoom(0.1),
           ]
         )
```

Let's visualize what a few augmented examples look like by applying data augmentation to the same image several times:



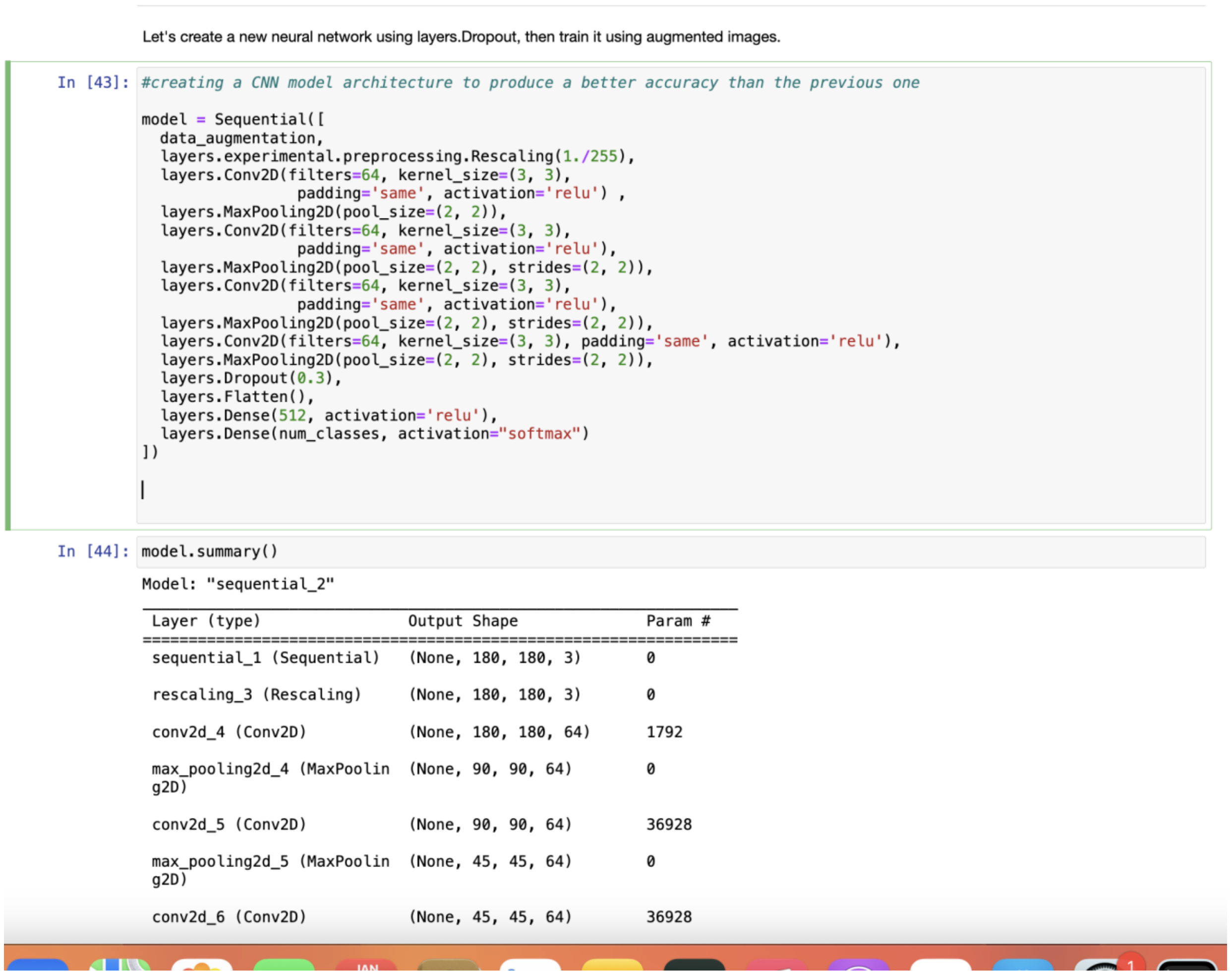
```
In [42]: plt.figure(figsize=(10, 10))
         for images, _ in train_ds.take(1):
           for i in range(9):
             augmented_images = data_augmentation(images)
             ax = plt.subplot(3, 3, i + 1)
             plt.imshow(augmented_images[0].numpy().astype("uint8"))
             plt.axis("off")
```

Describe the regularization methods that you used in your CNN model. How do they affect the accuracy of your results?

Dropout:

In addition to being a form of regularization, this technique involves deactivating random neurons during training to prevent overfitting. Another approach to mitigating overfitting is through the implementation of Dropout, which is a form of regularization. When Dropout is applied to a layer, a certain fraction of its output units is randomly dropped out during training by setting their activation to zero. These Dropout values typically take the form of decimal numbers such as 0.1, 0.2, 0.4, and so on.

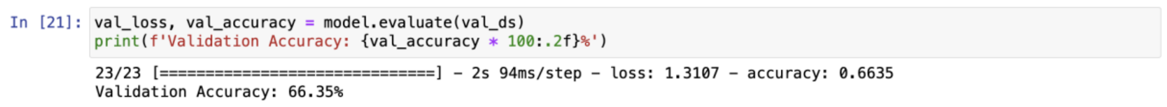
After we add the dropout to the data and other optimizer, we will have rerun the model again.



Let's create a new neural network using layers.Dropout, then train it using augmented images.

```
In [43]: #creating a CNN model architecture to produce a better accuracy than the previous one

         model = Sequential([
           data_augmentation,
           layers.experimental.preprocessing.Rescaling(1./255),
           layers.Conv2D(filters=64, kernel_size=(3, 3),
                         padding='same', activation='relu') ,
           layers.MaxPooling2D(pool_size=(2, 2)),
           layers.Conv2D(filters=64, kernel_size=(3, 3),
                         padding='same', activation='relu'),
           layers.MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),
           layers.Conv2D(filters=64, kernel_size=(3, 3),
                         padding='same', activation='relu'),
           layers.MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),
           layers.Conv2D(filters=64, kernel_size=(3, 3), padding='same', activation='relu'),
           layers.MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),
           layers.Dropout(0.3),
           layers.Flatten(),
           layers.Dense(512, activation='relu'),
           layers.Dense(num_classes, activation="softmax")
         ])
```

```
In [44]: model.summary()

         Model: "sequential_2"
         _________________________________________________________________
          Layer (type)                Output Shape              Param #
         =================================================================
          sequential_1 (Sequential)   (None, 180, 180, 3)       0

          rescaling_3 (Rescaling)     (None, 180, 180, 3)       0

          conv2d_4 (Conv2D)           (None, 180, 180, 64)      1792

          max_pooling2d_4 (MaxPoolin  (None, 90, 90, 64)        0
          g2D)

          conv2d_5 (Conv2D)           (None, 90, 90, 64)        36928

          max_pooling2d_5 (MaxPoolin  (None, 45, 45, 64)        0
          g2D)

          conv2d_6 (Conv2D)           (None, 45, 45, 64)        36928
```



```
In [21]: val_loss, val_accuracy = model.evaluate(val_ds)
         print(f'Validation Accuracy: {val_accuracy * 100:.2f}%')

         23/23 [==============================] - 2s 94ms/step - loss: 1.3107 - accuracy: 0.6635
         Validation Accuracy: 66.35%
```

Discuss any other hyperparameter tuning that you undertook to optimize your model. Which hyperparameters have the strongest effect on the performance of your model? Use suitable figures to visualize the accuracy and performance of your final model.

There were tons of optimizers used to adjust the accuracy of my model like using the adam optimizer, increasing the number of epochs, introducing the dropout, data augmentation and so on.

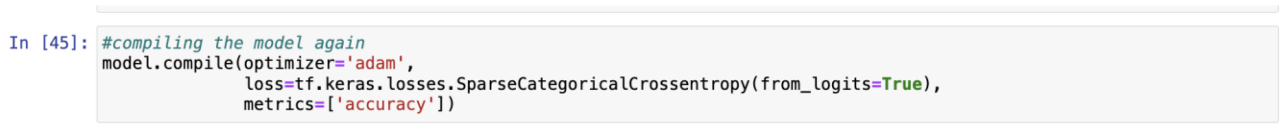
Learning Rate: This is an important hyperparameter optimizer that determines the step size during the optimizing process.

Batch size: adjusting the batch size can add speed to the model.

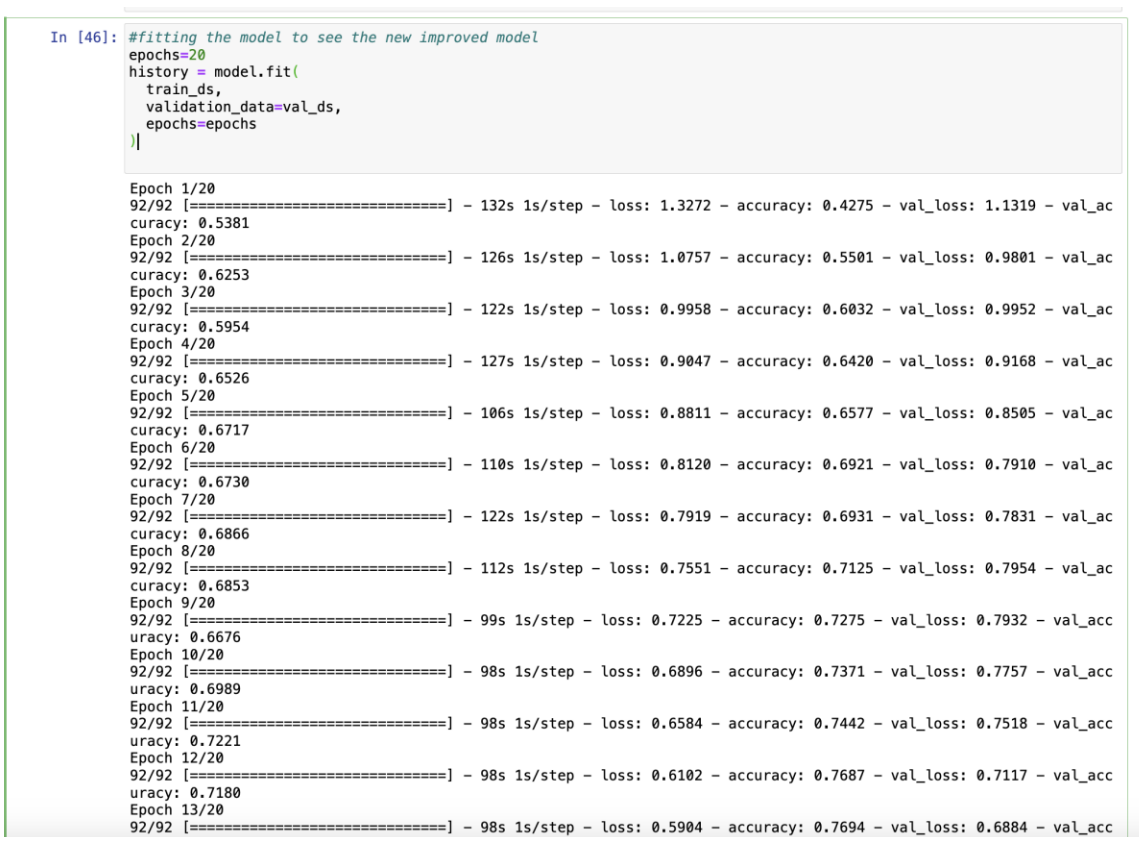
Number of Dense Layer: Increase in the Number of dense layers can also affect the performance of the model.

Dropout Rate: Dropout Rate introduced is always used to prevent overfitting, this can be used to optimize the model.

Kernel size and Filters: This accepts receptive fields and capacity of the model to extract hierarchical features.

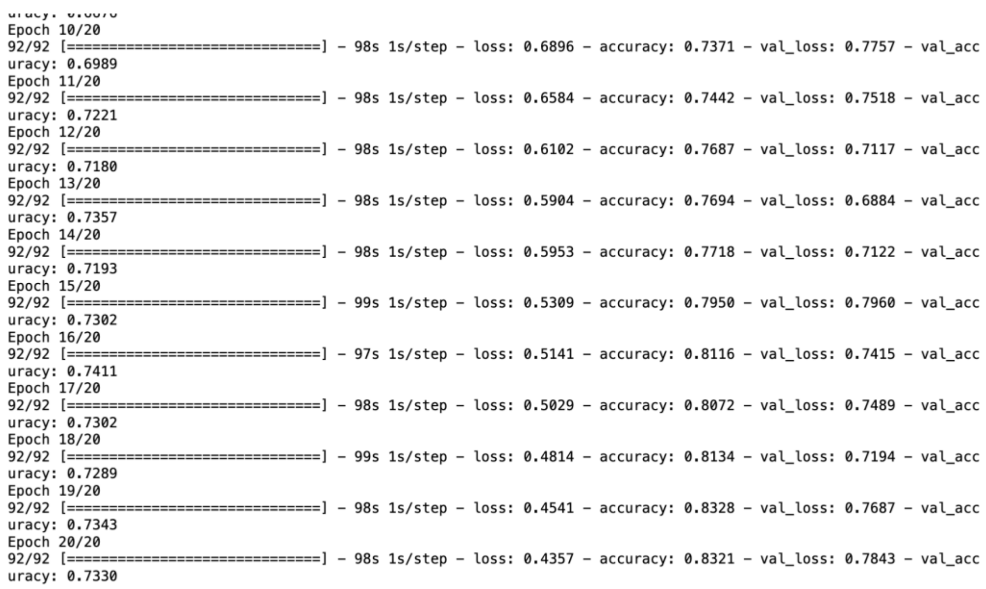


```
In [45]: #compiling the model again
         model.compile(optimizer='adam',
                       loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
                       metrics=['accuracy'])
```



```
In [46]: #fitting the model to see the new improved model
         epochs=20
         history = model.fit(
           train_ds,
           validation_data=val_ds,
           epochs=epochs
         )
```

```
Epoch 1/20
92/92 [==============================] - 132s 1s/step - loss: 1.3272 - accuracy: 0.4275 - val_loss: 1.1319 - val_ac
curacy: 0.5381
Epoch 2/20
92/92 [==============================] - 126s 1s/step - loss: 1.0757 - accuracy: 0.5501 - val_loss: 0.9801 - val_ac
curacy: 0.6253
Epoch 3/20
92/92 [==============================] - 122s 1s/step - loss: 0.9958 - accuracy: 0.6032 - val_loss: 0.9952 - val_ac
curacy: 0.5954
Epoch 4/20
92/92 [==============================] - 127s 1s/step - loss: 0.9047 - accuracy: 0.6420 - val_loss: 0.9168 - val_ac
curacy: 0.6526
Epoch 5/20
92/92 [==============================] - 106s 1s/step - loss: 0.8811 - accuracy: 0.6577 - val_loss: 0.8505 - val_ac
curacy: 0.6717
Epoch 6/20
92/92 [==============================] - 110s 1s/step - loss: 0.8120 - accuracy: 0.6921 - val_loss: 0.7910 - val_ac
curacy: 0.6730
Epoch 7/20
92/92 [==============================] - 122s 1s/step - loss: 0.7919 - accuracy: 0.6931 - val_loss: 0.7831 - val_ac
curacy: 0.6866
Epoch 8/20
92/92 [==============================] - 112s 1s/step - loss: 0.7551 - accuracy: 0.7125 - val_loss: 0.7954 - val_ac
curacy: 0.6853
Epoch 9/20
92/92 [==============================] - 99s 1s/step - loss: 0.7225 - accuracy: 0.7275 - val_loss: 0.7932 - val_acc
uracy: 0.6676
Epoch 10/20
92/92 [==============================] - 98s 1s/step - loss: 0.6896 - accuracy: 0.7371 - val_loss: 0.7757 - val_acc
uracy: 0.6989
Epoch 11/20
92/92 [==============================] - 98s 1s/step - loss: 0.6584 - accuracy: 0.7442 - val_loss: 0.7518 - val_acc
uracy: 0.7221
Epoch 12/20
92/92 [==============================] - 98s 1s/step - loss: 0.6102 - accuracy: 0.7687 - val_loss: 0.7117 - val_acc
uracy: 0.7180
Epoch 13/20
92/92 [==============================] - 98s 1s/step - loss: 0.5904 - accuracy: 0.7694 - val_loss: 0.6884 - val_acc
```



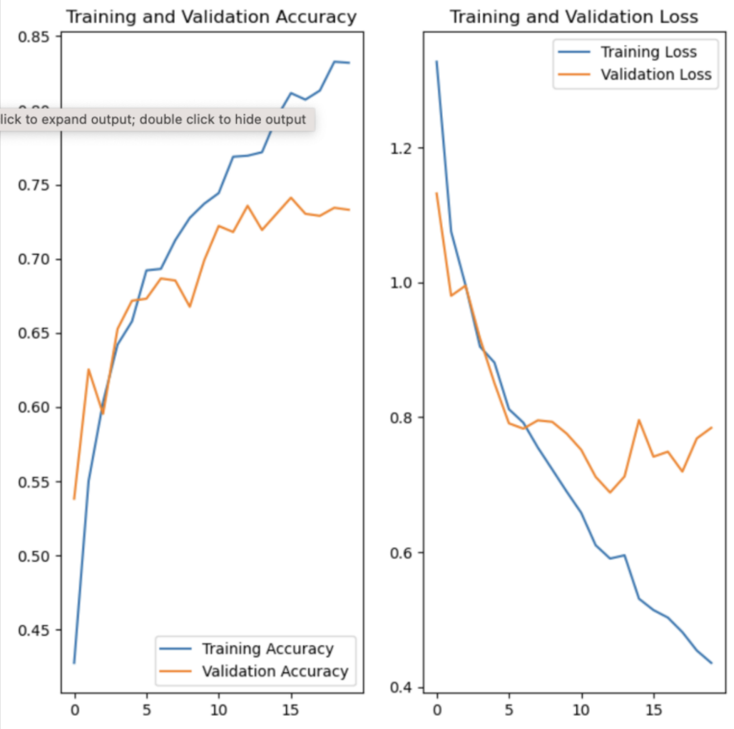
```
Epoch 10/20
92/92 [==============================] - 98s 1s/step - loss: 0.6896 - accuracy: 0.7371 - val_loss: 0.7757 - val_acc
uracy: 0.6989
Epoch 11/20
92/92 [==============================] - 98s 1s/step - loss: 0.6584 - accuracy: 0.7442 - val_loss: 0.7518 - val_acc
uracy: 0.7221
Epoch 12/20
92/92 [==============================] - 98s 1s/step - loss: 0.6102 - accuracy: 0.7687 - val_loss: 0.7117 - val_acc
uracy: 0.7180
Epoch 13/20
92/92 [==============================] - 98s 1s/step - loss: 0.5904 - accuracy: 0.7694 - val_loss: 0.6884 - val_acc
```

```
uracy: 0.7357
Epoch 14/20
92/92 [==============================] - 98s 1s/step - loss: 0.5953 - accuracy: 0.7718 - val_loss: 0.7122 - val_acc
uracy: 0.7193
Epoch 15/20
92/92 [==============================] - 99s 1s/step - loss: 0.5309 - accuracy: 0.7950 - val_loss: 0.7960 - val_acc
uracy: 0.7302
Epoch 16/20
92/92 [==============================] - 97s 1s/step - loss: 0.5141 - accuracy: 0.8116 - val_loss: 0.7415 - val_acc
uracy: 0.7411
Epoch 17/20
92/92 [==============================] - 98s 1s/step - loss: 0.5029 - accuracy: 0.8072 - val_loss: 0.7489 - val_acc
uracy: 0.7302
Epoch 18/20
92/92 [==============================] - 99s 1s/step - loss: 0.4814 - accuracy: 0.8134 - val_loss: 0.7194 - val_acc
uracy: 0.7289
Epoch 19/20
92/92 [==============================] - 98s 1s/step - loss: 0.4541 - accuracy: 0.8328 - val_loss: 0.7687 - val_acc
uracy: 0.7343
Epoch 20/20
92/92 [==============================] - 98s 1s/step - loss: 0.4357 - accuracy: 0.8321 - val_loss: 0.7843 - val_acc
uracy: 0.7330
```

Training Results after Hyperparameter tuning

After applying data augmentation and Dropout, there is less overfitting than before, and training and validation accuracy are closer aligned.

In [47]:
```
# creating the accuracy and validation accuracy dataframe
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

# creating the accuracy and validation accuracy dataframe
loss = history.history['loss']
val_loss = history.history['val_loss']

#create the epoch range
epochs_range = range(epochs)

#visualization and labelling for training and validation accuracy
plt.figure(figsize=(8, 8))
plt.subplot(1, 2, 1)
plt.plot(epochs_range, acc, label='Training Accuracy')
plt.plot(epochs_range, val_acc, label='Validation Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')

#visualization and labelling for training loss and validation loss
plt.subplot(1, 2, 2)
plt.plot(epochs_range, loss, label='Training Loss')
plt.plot(epochs_range, val_loss, label='Validation Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.show()
```



In [29]:
```
val_loss, val_accuracy = model.evaluate(val_ds)
print(f'Validation Accuracy: {val_accuracy * 100:.2f}%')
```

```
23/23 [==============================] - 5s 227ms/step - loss: 0.7668 - accuracy: 0.7330
Validation Accuracy: 73.30%
```

Conclusion

In conclusion, the evolution of Artificial intelligence, particularly in the realm of image recognition, has ushered in a transformative era for solving complex issues, notably the identification of flowers. The advent of deep learning convolutional neural networks marks a significant breakthrough, offering a sophisticated solution to a crucial problem in the technological landscape. This progression signifies a monumental shift in the traditional methods of flower identification, paving the way for a more efficient, accurate and technologically advanced approach.

References

Syed Muhammad Sajjad Kabir

Method of Data Collection

https://www.researchgate.net/publication/325846997_METHODS_OF_DATA_COLLECTION

Malcolm Chisholm

Defining data acquisition and why it matters

https://www.firstsanfranciscopartners.com/blog/defining-data-acquisition-importance/

Nikita Duggal

Top 10 Python Libraries in 2022

https://www.simplilearn.com/top-python-libraries-for-data-science-article