Neel Deorukhkar

Final Report

Flower Image Classification

Background

A Flower shop sells 10 different kinds of flowers such as Hibiscus, lilies and daisies. Customers buy these flowers and upload pictures to the Flower shop website. The Florist then takes these photos and manually labels each photo so that it can be added to the Customer Image Reviews database.

Problem Statement

Currently, the florist manually classifies flowers based on the image and assigns the photos a label. The business objective of the flower shop is to build an automated image classifier so that the florist does not have to spend time manually classifying customer photos. Not only will this classifier save the florist time manually classifying these images, but also help her utilize the time saved to more useful tasks such as watering the plants or decorating new flower bouquets.

Criteria for Success

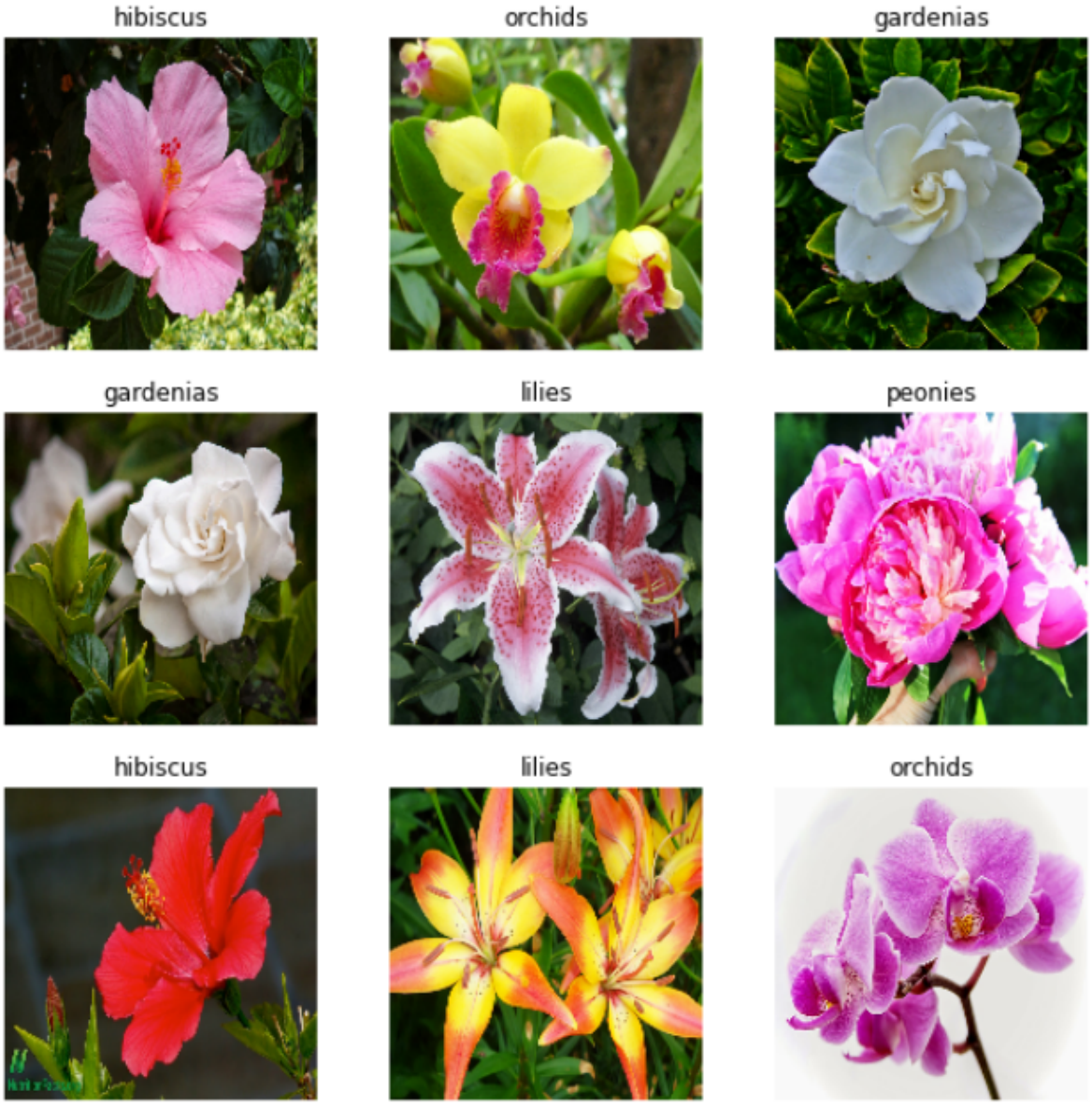
Due to high demand for the flowers and low labor shortage, the business owner has outsourced this task to a consulting firm. The Criteria of success would be to classify the flowers accurately in the least amount of time possible.

Data Sources

The train and test datasets are curated from the link here: [Kaggle]

There are 733 images of the 10 different categories of flowers.

Raw Data



Preprocessing Data -- Data Augmentation

Given that the dataset consists of only 733 images, we need to augment the data by randomly flipping the training images, randomly rotating the images and randomly zooming into the images

- RandomFlip
- Random Rotate
- Random Zoom

Base Model

```
import tensorflow as tf
import tensorflow.keras.layers as tfl
from tfl.experimental.preprocessing import RandomFlip, RandomRotation, RandomZoom
from tf.keras.layers import Conv2D, BatchNormalization, ReLU, MaxPool2D,
from tf.keras.layers import GlobalAveragePooling2D, Dropout, Flatten, Dense

#Image Size (160, 160)
input_shape = IMG_SIZE + (3, )
inputs = tf.keras.Input(shape=input_shape)
#Data Augmentation
data_augmentation = tf.keras.Sequential()
data_augmentation.add(RandomFlip('horizontal'))
data_augmentation.add(RandomRotation(0.2))
data_augmentation.add(RandomZoom(0.5, 0.2))
x = data_augmentation(inputs)

#Convolutional Neural Networks Architecture
#Layer 1 - CONV2D -> BatchNorm -> ReLU -> MaxPool x 64 Filters
x = Conv2D(filters = 64, kernel_size = (4, 4), strides = (1, 1), padding = 'same')(x)
x = BatchNormalization(axis = 3)(x)
x = ReLU()(x)
x = MaxPool2D(pool_size = (8, 8), strides = (8, 8), padding = 'same')(x)
#Layer 2 - CONV2D -> BatchNorm -> ReLU -> MaxPool x 128 Filters
x = Conv2D(filters = 128, kernel_size = (2, 2), strides = (1, 1), padding = 'same')(x)
x = BatchNormalization(axis = 3)(x)
x = ReLU()(x)
x = MaxPool2D(pool_size = (4, 4), strides = (4, 4), padding = 'same')(x)
#Layer 3 - CONV2D -> BatchNorm -> ReLU -> MaxPool x 256 Filters
x = Conv2D(filters = 256, kernel_size = (2, 2), strides = (1, 1), padding = 'same')(x)
x = BatchNormalization(axis = 3)(x)
x = ReLU()(x)
x = MaxPool2D(pool_size = (2, 2), strides = (2, 2), padding = 'same')(x)
x = GlobalAveragePooling2D()(x)
#Added to reduce Overfitting
x = Dropout(0.2)(x)
x = Flatten()(x)
#Dense
outputs = Dense(10, activation = 'softmax')(x)

model = tf.keras.Model(inputs = inputs, outputs = outputs)
```

Base Model Architecture

- The base model architecture consists of 3 convolutional layers, each with an increasing number of filters
- Each convolutional layer consists of a Conv2D layer followed by batch normalization that standardizes the inputs for each mini batch
- The padding is set as ‘same’ to preserve the input shape
- The batch normalization is followed by a ReLu (Rectified Linear Unit) activation
- The Relu activation is followed by a Max Pool 2D layer
- The number of filters in each layer increases from 64 to 128 to 256 followed by reducing the pool sizes from 8 to 4 to 2 in the Max Pool Layer.

Various steps have been taken to reduce overfitting such as Data augmentation and Dropout.

The Data augmentation increases the number of training examples by random flipping, randomly rotating and randomly zooming into images.

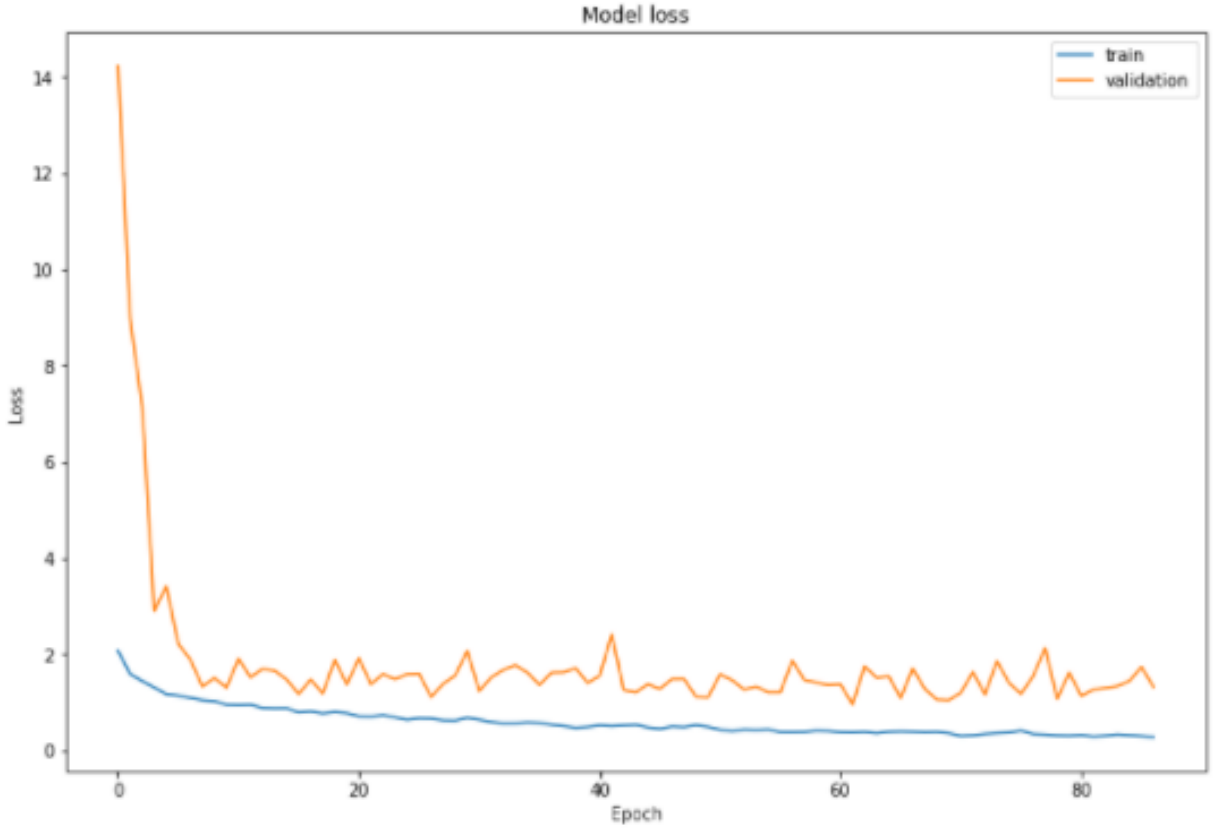
The Dropout randomly sets the weights of the hidden units to zero. As a result, I’ve introduced some bias in the model.

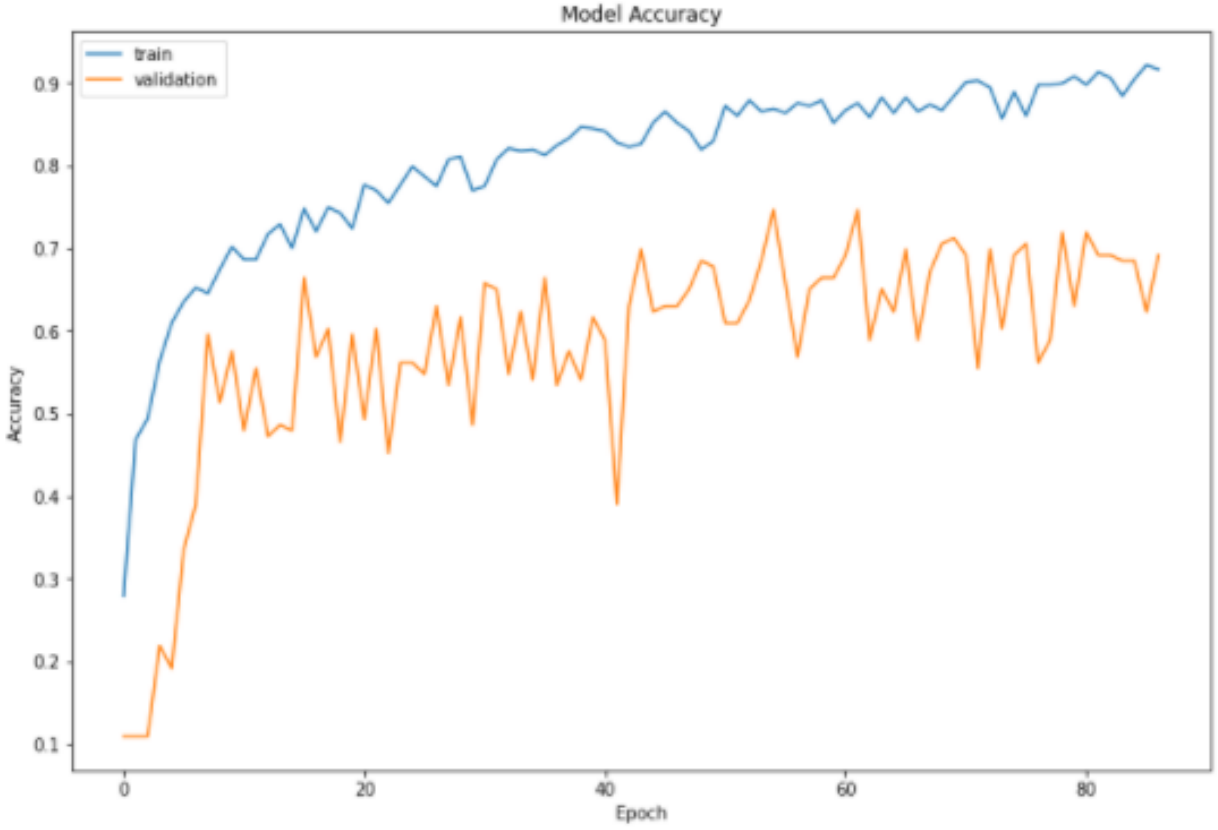
I’ve compiled the model using the **Adam** optimizer from keras with a learning rate of 0.001. The loss is set to SparseCategoricalCrossentropy for a multi class problem.

The batch size is set to 32 and the number of epochs to 70. Due to the high number of epochs, I’ve added an EarlyStopping patience of 30 epochs. The EarlyStopping monitors the validation loss and if the validation loss does not improve for 30 epochs, then the model stops training prematurely thereby saving the computational time. EarlyStopping saves the best model with the minimum validation loss.

Base Model Performance

The base model overfits the validation data set since the **training accuracy** is **98%** and the **validation accuracy** is **70%**. The **validation loss** is **1.32**.





Transfer Learning

❖ Mobile Net v2 Architecture

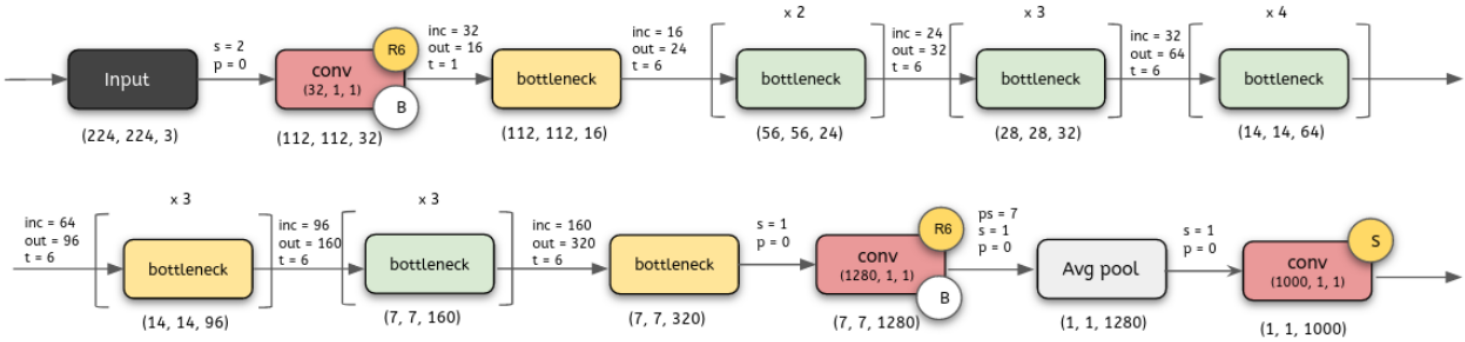
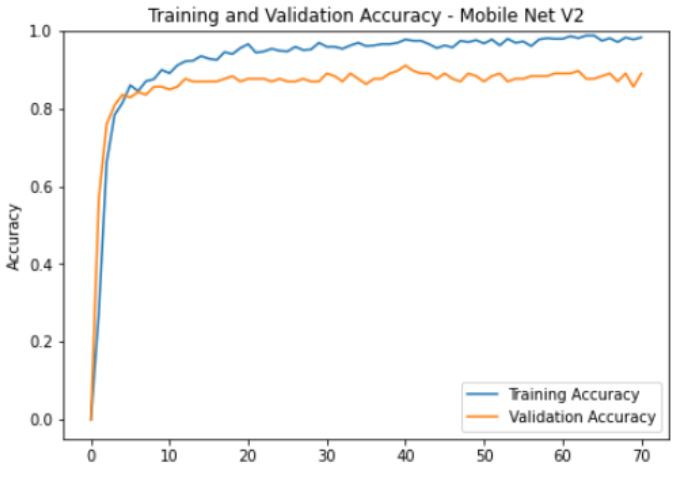


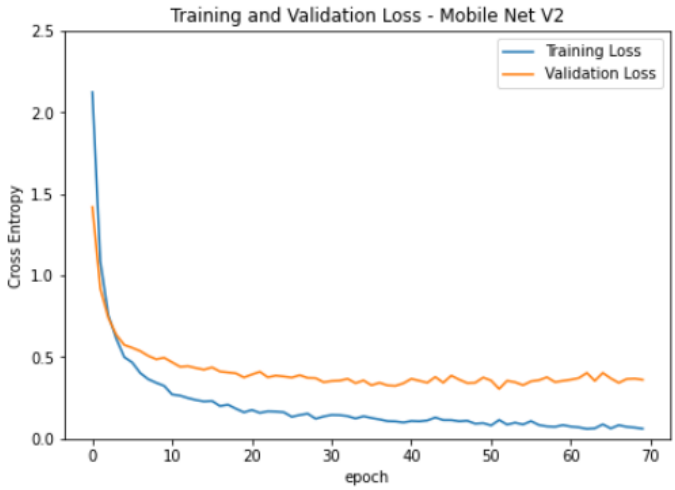
***Image adapted from : here*

The MobileNetV2 is a pretrained model that uses pretrained weights from ImageNet. The original model predicts 1000 classes. However, the dataset that we are working with consists of only 10 classes. As a result, we can drop the top layer, and include a Dense layer of 10 classes with a 'softmax' activation function.

The Mobile Net V2 model does not overfit the validation data set since the **training accuracy is 98%** and the **validation accuracy is 89%**. The validation **loss** too has improved to **0.35**

Furthermore, the time per epoch is only 300 ms.





❖ VGG - 16 Architecture

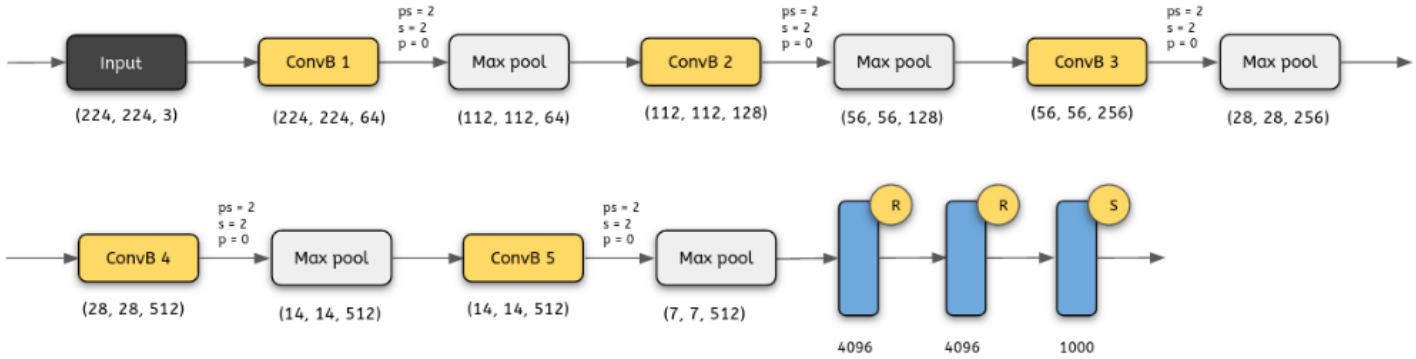
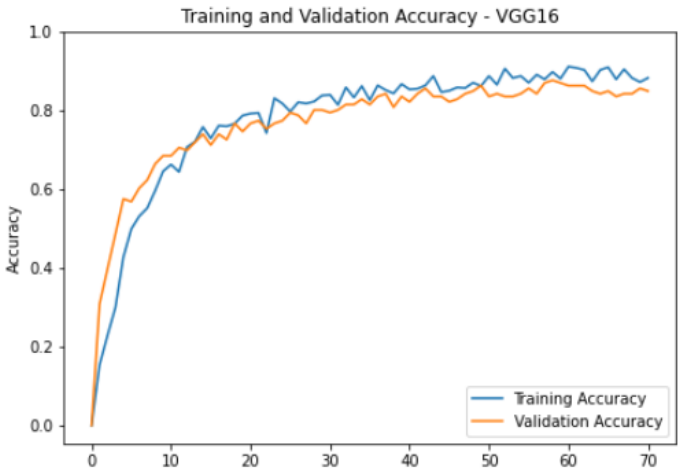


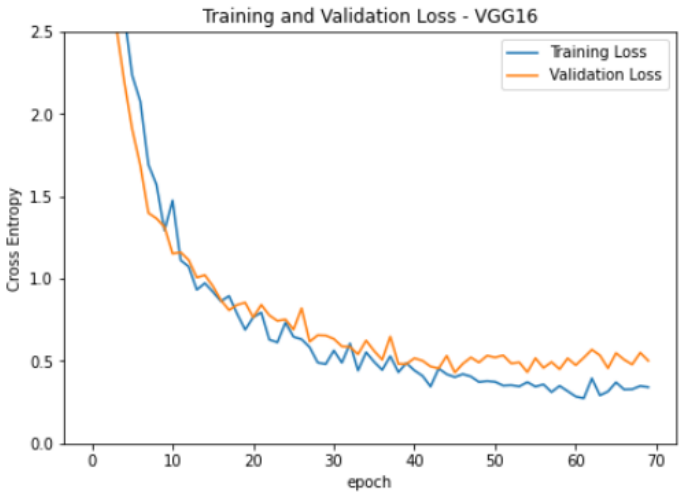
***Image adapted from : here*

The VGG16 model is much slower than the MobileNetV2 model because of the size of the network and number of parameters.

Secondly, the model predicts better than the base model but much more poorly than the MobileNetV2 model.

The VGG-16 model does not overfit the data but has a high bias since the **training accuracy is 88%** and the **validation accuracy is 85%**. The validation **loss** too has improved to **0.50**





This gives a sense that not all pretrained models work in the similar way. We can try out the ResNet50 model next to see the performance

❖ ResNet50 Architecture

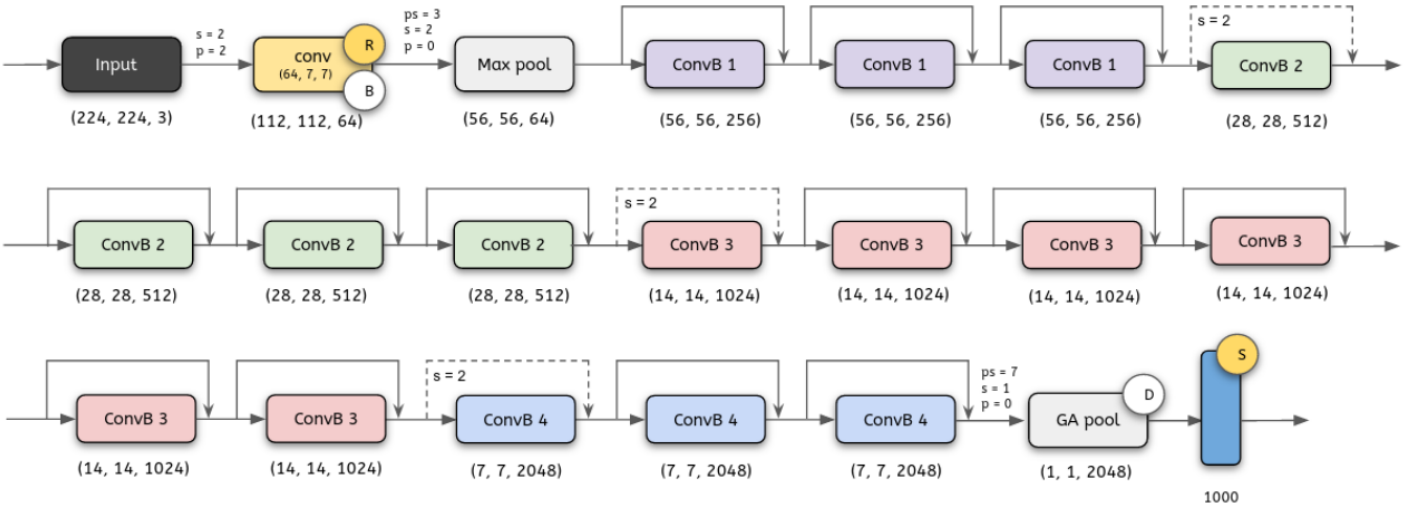
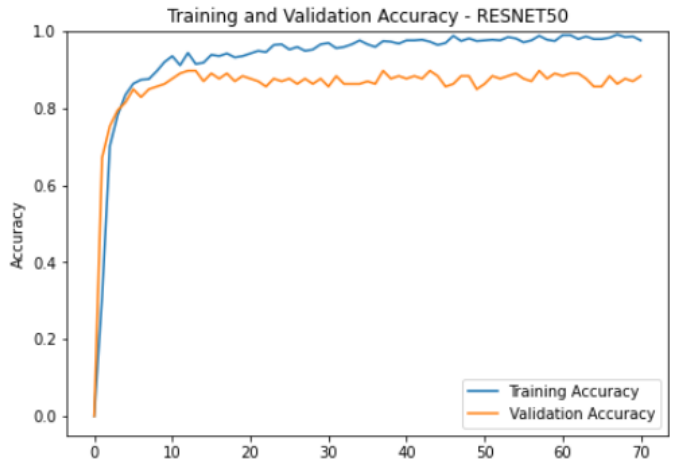
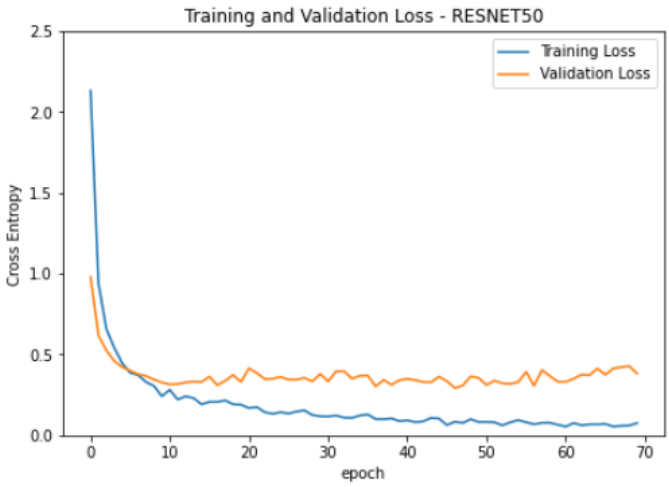


**Image adapted from : here*





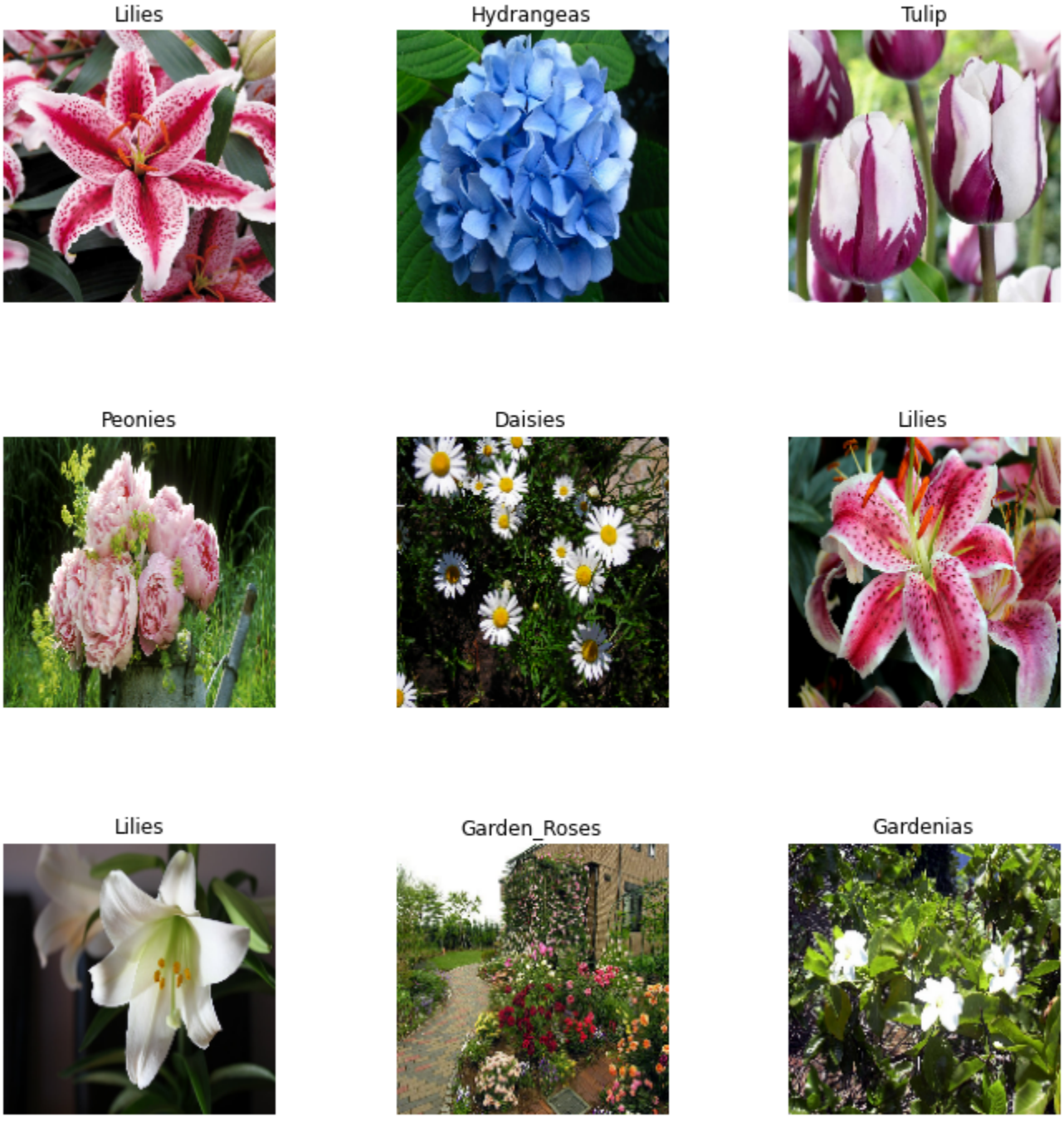
The Resnet50 model does not overfit the data since the **training accuracy is 97%** and the **validation accuracy is 88%**. The validation **loss** too has improved to **0.38**

The Resnet50 model does better than the base model and VGG16 model.

This is pretty good, but the Mobile V2 Model is the best in terms of accuracy and lowest validation loss.

Modeling Predictions

Now that we've finalized the Mobile Net v2 model, we can apply the model to predict the images from the validation dataset.



Modeling Summary

To wrap up, we've tried and tested different CNN modeling architectures and built a deep learning model to predict 89% of the labels correctly.

The model diagnostics can be viewed below:

Model	Validation_Accuracy	Validation_loss	Time per Epoch
base_model	70%	1.32	500ms
MobileNetV2	**89%**	**0.35**	**350ms**
VGG16	85%	0.50	34s
ResNet50	88%	0.38	786ms