Caltech-Birds Fine-grained Image Classification

Introduction

The main aim of this project is to use the Caltech-UCSD Birds-200-2011 dataset (Wah, Branson, Welinder, Perona, & Belongie, 2011) to create and train an image classifier. This classifier is intended to classify 200 species of bird from a dataset of 11,788 images.

In support of our aim to create an image classifier we have also performed a literature review and a descriptive analysis of the dataset. Understanding the current state of research in this field enables us to use the best and most relevant techniques to solve the problem. It also helps us to avoid common pitfalls already encountered by the scientific community. Importantly, the literature review allows us to compare our results with other published results.

Performing our own descriptive analysis of the dataset helps us to understand the data, how it is represented and the challenges that must be faced to create an effective image classifier.

The first step to creating the classifier is to pre-process the data. The information gained from the dataset analysis directly leads into this step as it directly influences what needs to be done to the data to prepare it for training and classification. This step includes formatting and partitioning the data so that it can be used in training, testing, and validation.

Once the data is pre-processed, machine learning is then used to create a classifier. We train a tuned convolutional neural network using one partition of the data and validate and test it using other preselected partitions of the data.

In conclusion we present our results and then analyse the accuracies, inaccuracies and the errors encountered. We attempt to explain these errors and describe what can be done in future to improve our solution.

Description of the task & dataset

Dataset Introduction

This is a fine-grained classification data set, which is different from coarse-grained classification (such as classifying cats and dogs), this data contains birds, and a more fine-grained classification is performed. Analyzing the data set, we can find that the differences within the class are quite large. Considering Black_footed_Albatross, as shown in Figure 1, the same species of birds differ in different motion states, and it is difficult for humans to consider them as a bird. At the same time, it is difficult to distinguish distinct types of birds that are similar in the same movement state. There are fewer samples in each category, only about 60 samples, and the pictures are disturbed by background, light, etc., so classification is difficult.

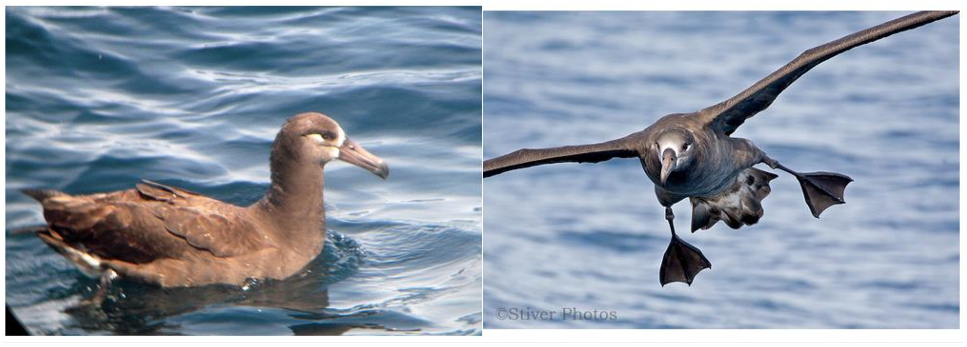
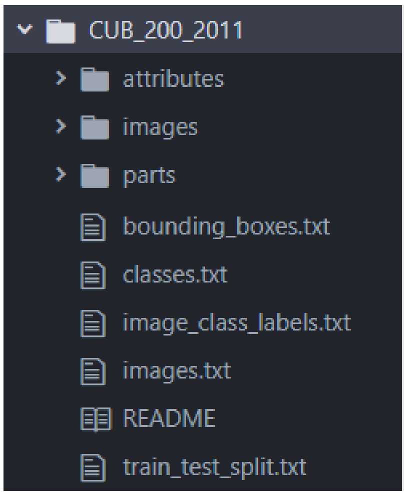


Figure 1 - Black Footed Albatross in different states.

Document introduction

The directory contains three folders (attributes, images, parts) and five txt files, as shown in Figure 2.



```
CUB_200_2011
    attributes
    images
    parts
    bounding_boxes.txt
    classes.txt
    image_class_labels.txt
    images.txt
    README
    train_test_split.txt
```

Figure 2 - Dataset Structure

The images are contained in the directory 'images/', which has 200 subdirectories (classes) that are named after the bird species.

There are 11788 images in total, and the images.txt file defines an ID for each image (one image per line: <image_id> <image_name>).

Labels are defined in the classes.txt file (one image per line: <image_id> <image_name>). According to the rules in the train_test_split.txt file (<image_id> <is_training_image>), the value of

<is_training_image> is 1 or 0, indicating that the file is in the train or test set, respectively. It is divided into a training set of 5994 and a test set of 5794.

image_class_labels.txt defines labels for each image (one image per line: <image_id> <class_id>, where <image_id> and <class_id> correspond to the IDs in images.txt and classes.txt respectively).

In addition, a bounding box is defined for each image in bounding_boxes.txt, which is used as a priori knowledge of human labels to help the model distinguish the bird in the image from the background. One image per line: <image_id> <x> <y> <width> <height> where <image_id> corresponds to the ID in images.txt, <x>, <y>, <width> and <height> are all measured in pixels.

In 'attributes/', a list of all attribute names is contained in the file attributes/attributes.txt, one attribute per line: <attribute_id> <attribute_name>.

All deterministic names (certainties used by workers to specify their attribute responses) are contained in the file attributes/certainties.txt, with each line corresponding to a certainty: <certainty_id> <certainty_name>.

The set of attribute labels for each image perception is contained in the file attributes/image_attribute_labels.txt, one image/attribute/job triple per line: <image_id> <attribute_id> <is_present> <certainty_id> <time>. Where <image_id>, <attribute_id>, <certainty_id> correspond to the IDs in images.txt, attributes/attributes.txt and attributes/certainties.txt, respectively. <is_present> is either 0 or 1 (1 means the attribute is present). <time> represents the time spent, in seconds.

The class attribute labels (attributes/class_attribute_labels_continuous.txt) contains 200 rows and 312 space-separated columns. Each row corresponds to a class (same order as classes.txt) and each column contains a real value corresponding to an attribute (same order as attributes.txt). The number is the percentage of time (between 0 and 100) that humans consider the attribute to exist in a given category.

In 'parts/', parts/parts.txt corresponds to one part per line: <part_id> <part_name>. A collection of all ground-truth part locations is contained in the file parts/part_locs.txt, with each line corresponding to a comment for a specific part in a specific image: <image_id> <part_id> <x> <y> <visible>. where <image_id> and <part_id> correspond to the IDs in images.txt and parts/parts.txt respectively. <x> and <y> represent the pixel position of the center of the widget. <visible> is 0 if the part is not visible in the image, 1 otherwise.

Multiple sets of part locations per image and part as perceived by multiple users are contained in parts/part_click_locs.txt, with each line corresponding to a different MTurk worker's annotation for a particular part in a particular image: <image_id> <part_id> < x> <y> <visible> <time>. where <image_id>, <part_id>, <x>, <y> are in the same format as defined in parts/part_locs.txt and <time> is the number of seconds spent by the worker.

The dataset contains several types of files that can be used as auxiliary input to improve the accuracy of the classification model. However, too many applications will lead to excessive model complexity. Therefore, the content should be selectively selected as input according to the actual situation.

Methodology

To pre-process the data, train the model and test the model we initially elected to use Keras. Keras is a machine learning API built on the TensorFlow 2 platform. TensorFlow is an open source machine learning platform, well known and well used by the scientific community and industry. We decided to use Python to implement the program for a multitude of reasons. Firstly, the Keras API is designed to work seamlessly with Python. Python is the strongest language of all the group members, allowing us to play to our strengths. Python is also fast becoming the language most used for machine learning tasks by the scientific community.

Choosing this API and platform gives us access to a large amount of documentation, example projects and assistance. This type of support is essential when undertaking a project like this, as it allows us to solve problems we encounter by learning from example.

Pre-processing

To pre-process the data, we got from the Caltech birds' dataset we initially used methods provided by the Keras API in Python. However, before pre-processing the data we needed to split the data into three sets: one for training, one for testing and one for validation. The training set consists of 5994 images, as predefined in the file train_test_split.txt that came with the dataset. The test set of 5794 is split in half. Half of this set is for testing and the other half is for validation.

With these partitions defined, the data can be pre-processed for use as training data. To pre-process the data at first, we used the ImageDataGenerator tool provided by the Keras API. The generator class is used to load and stream image data from a directory directly to a model to be trained. This has the advantage of not needing to preload the data into a large structure, which would make the memory overheads much larger than they already are training the model. The generator class also allows us to perform pre-processing on each image to increase the diversity of the dataset. Each image can be scaled, rotated, shifted, flipped and/or colour corrected by the ImageDataGenerator object as it is being streamed from file. Doing this increases the diversity of the dataset as each image is modified in some way to create greater variety.

We decided to also include a further pre-processing step to increase the perceived size of our dataset. We used the cv2 library to make modified copies of each image in the training set. We then placed these copies back into the data, meaning each class has more images to train with. Using ImageDataGenerator and cv2 to modify the images also makes the classifier more robust as it is learning from images with different orientations, scales, and colour pallets.

Training

To train a model to classify the data we used more tools from Keras and TensorFlow. We also used more tools from the cv2 and scikit-learn libraries. Keras provides many different pretrained and untrained models that can be used together in a model stack. To get a model ready for training it must be built out of layers. A typical structure for an image classification model is to use a pretrained model in a sequential stack followed by various manipulation layers. These layers can consist of convolutional layers, dense layers, normalisation layers or other types of neural layer provided by Keras. The pretrained models at the top of the stack are also provided by Keras and there are many different models with many different versions. We tested this while we were creating the final model by experimenting with different pretrained models.

Once the stack has been created, training the model consists of sending the ImageDataGenerator to the model's .fit() function. This will start the process of training the model on the data and tuning parameters until the data has all been used and the accuracy is acceptable.

Experimental Setting

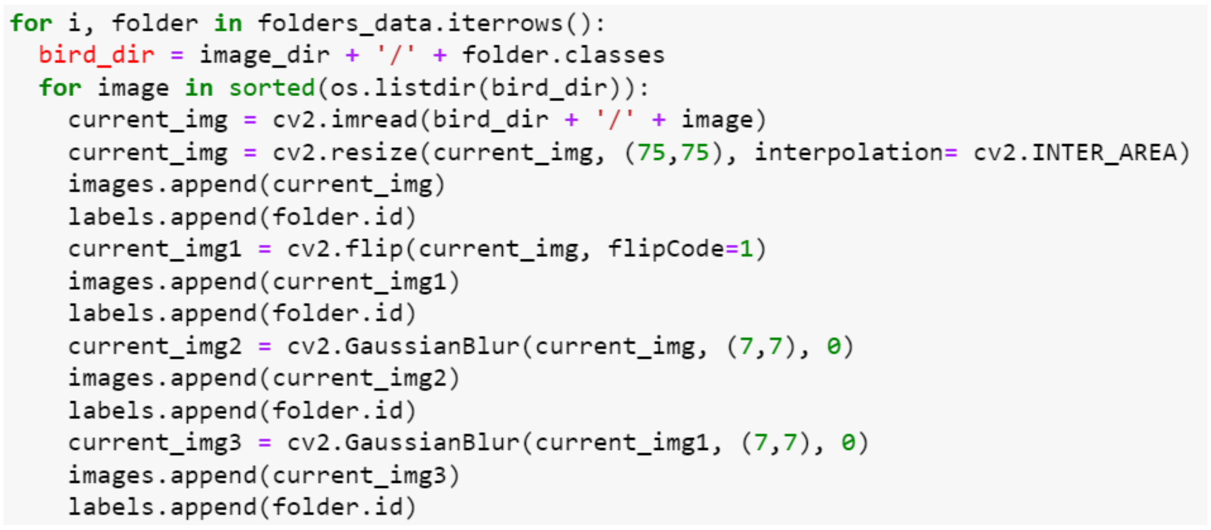
Many tweaks and modifications were made to the pre-processing and training stack during development to achieve the highest possible accuracy in training and in validation.

Pre-processing

As previously stated, the pre-processing was performed by first splitting the data, using cv2 to copy and modify the data and then using an ImageDataGenerator object to modify more of the data as it is streamed to the classifier.

To split the data, we initially took the partitions laid out in the file test_train_split.txt that came with the dataset. However, after experimentation this was not the best method for us to use. In the end we used the train_test_split function provided by the Scikit-learn module. This function splits the data using stratified sampling. This means an equal proportion of all the classes are taken for training and testing. We used 95% of the data for training in this final version.

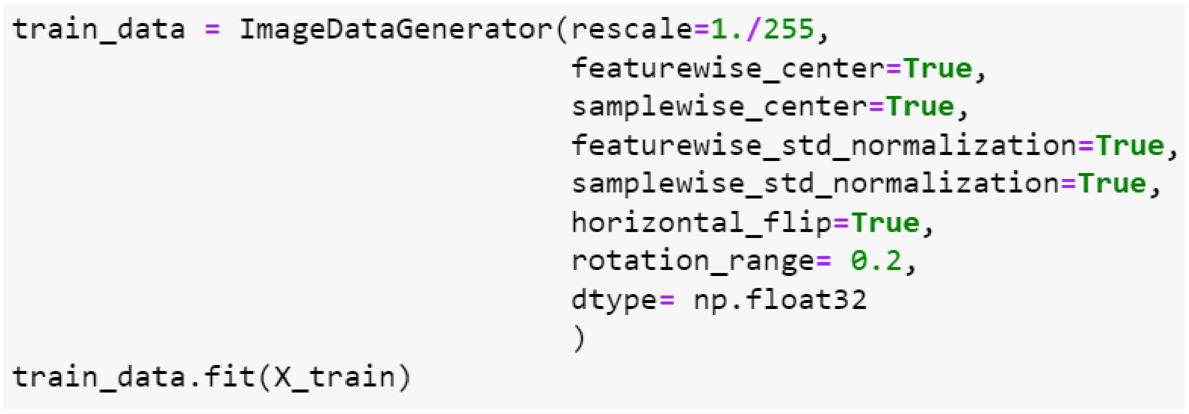
Once the images have been partitioned, some functions from the cv2 library are used to copy the images and make modifications to these copies. Figure 4 shows the modifications that are being made to each image. For every image, the original is taken and flipped horizontally. The original and the flipped version then have gaussian blur applied to them. This takes each image and creates 4 copies, effectively quadrupling the size of our training set. After this the images are all cast to NumPy arrays. NumPy arrays have a better structure for training and make the training faster.



```
for i, folder in folders_data.iterrows():
  bird_dir = image_dir + '/' + folder.classes
  for image in sorted(os.listdir(bird_dir)):
    current_img = cv2.imread(bird_dir + '/' + image)
    current_img = cv2.resize(current_img, (75,75), interpolation= cv2.INTER_AREA)
    images.append(current_img)
    labels.append(folder.id)
    current_img1 = cv2.flip(current_img, flipCode=1)
    images.append(current_img1)
    labels.append(folder.id)
    current_img2 = cv2.GaussianBlur(current_img, (7,7), 0)
    images.append(current_img2)
    labels.append(folder.id)
    current_img3 = cv2.GaussianBlur(current_img1, (7,7), 0)
    images.append(current_img3)
    labels.append(folder.id)
```

Figure 3 - Code to copy and modify images with cv2

After the modifications with cv2, an ImageDataGenerator object is initialised with different parameters. These parameters, shown in figure 5, affect the modifications that can/will be made to the images as they are loaded.



```
train_data = ImageDataGenerator(rescale=1./255,
                                featurewise_center=True,
                                samplewise_center=True,
                                featurewise_std_normalization=True,
                                samplewise_std_normalization=True,
                                horizontal_flip=True,
                                rotation_range= 0.2,
                                dtype= np.float32
                                )
train_data.fit(X_train)
```

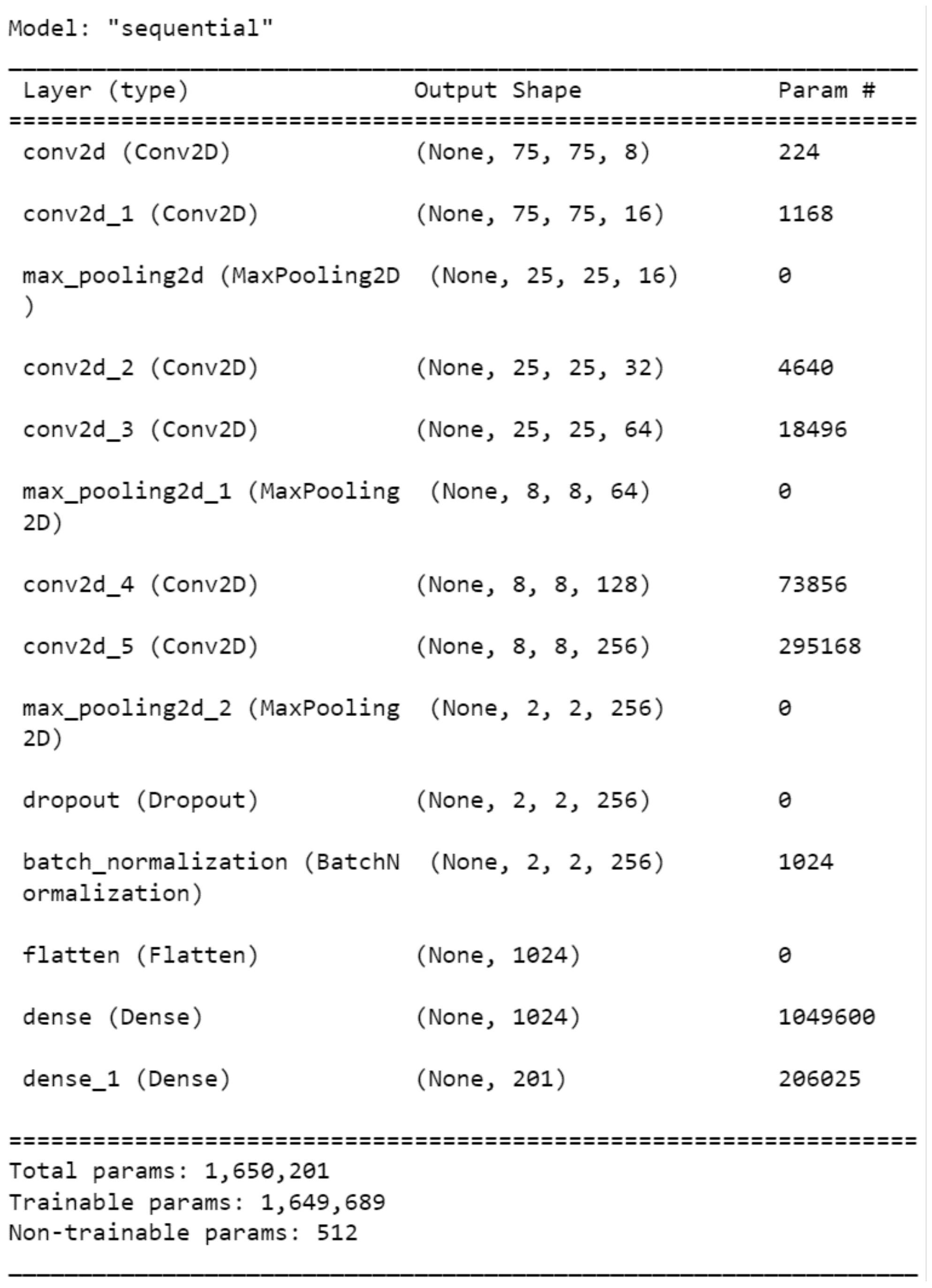
Figure 4 - Creation of ImageDataGenerator for training data.

Once the parameters have been selected and the ImageDataGenerator object has been initialised, the generator needs to be fit to the data. This allows it to make modifications when the data is used.

Training

After the pre-processing, the model is ready to be trained. To train a model on our data we first had to build one. This was a crucial decision as there exist many ways to build ML (Machine Learning) models. Keras provides many different pretrained models that we could pick from. We experimented with using pretrained models VGG19, MobileNet, ResNet50, Xception, DenseNet169, DenseNet121, DenseNet201 and vgg19 to compare their performance. However, after our experimentation with these pretrained models, we concluded that for our task a fresh model created by ourselves would fit better and produce the best performance.

Eventually we came to use the model pictured in figure 5. This is a sequential model consisting of three groups of 2d convolutional steps of increasing size with max pooling steps, which are then followed by a dropout layer, batch normalisation, flattening layer and then two dense layers with different activation functions, ReLu and softmax. Convolutional steps are used as these have shown to have the best performance for image classifying neural networks. The batch normalisation and dropout layers are used to help prevent overfitting. The dense layers are applied with an L2 kernel regulariser also to prevent overfitting.



```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 75, 75, 8)         224

 conv2d_1 (Conv2D)           (None, 75, 75, 16)        1168

 max_pooling2d (MaxPooling2D  (None, 25, 25, 16)       0
 )

 conv2d_2 (Conv2D)           (None, 25, 25, 32)        4640

 conv2d_3 (Conv2D)           (None, 25, 25, 64)        18496

 max_pooling2d_1 (MaxPooling  (None, 8, 8, 64)         0
 2D)

 conv2d_4 (Conv2D)           (None, 8, 8, 128)         73856

 conv2d_5 (Conv2D)           (None, 8, 8, 256)         295168

 max_pooling2d_2 (MaxPooling  (None, 2, 2, 256)        0
 2D)

 dropout (Dropout)           (None, 2, 2, 256)         0

 batch_normalization (BatchN  (None, 2, 2, 256)        1024
 ormalization)

 flatten (Flatten)           (None, 1024)              0

 dense (Dense)               (None, 1024)              1049600

 dense_1 (Dense)             (None, 201)               206025

=================================================================
Total params: 1,650,201
Trainable params: 1,649,689
Non-trainable params: 512
_________________________________________________________________
```

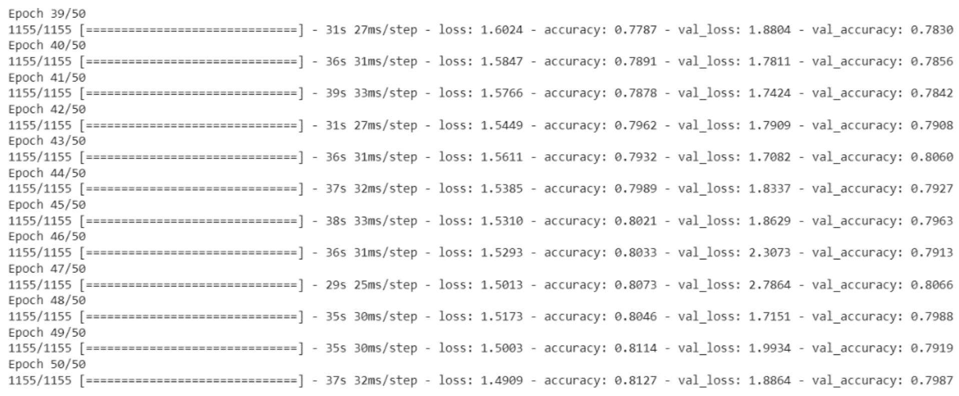
Figure 5 - Final Model stack.

Testing

To evaluate and test the models once they have been trained, we used several different tools. When training the model with Keras it will output its own accuracy statistics for both training and validation accuracy. We also used one of Scikit-learn’s tools for calculating precision, recall, accuracy and F1 score. To visualise results and to compare models, we used Matplotlib for graph creation.

Results

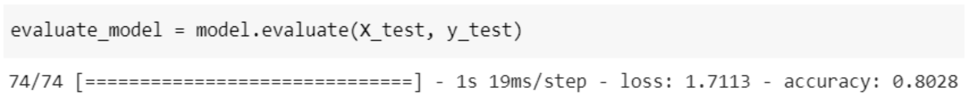
Finally, we applied the above model on the training and the validation dataset where the size of the overall training and validation dataset was 80% and 15% respectively. A decent training and test accuracy of 81% and 80% were acquired with a loss rate of 1.5 and 1.88 respectively after completing 50 epochs as shown in figure 7.



```
Epoch 39/50
1155/1155 [==============================] - 31s 27ms/step - loss: 1.6024 - accuracy: 0.7787 - val_loss: 1.8804 - val_accuracy: 0.7830
Epoch 40/50
1155/1155 [==============================] - 36s 31ms/step - loss: 1.5847 - accuracy: 0.7891 - val_loss: 1.7811 - val_accuracy: 0.7856
Epoch 41/50
1155/1155 [==============================] - 39s 33ms/step - loss: 1.5766 - accuracy: 0.7878 - val_loss: 1.7424 - val_accuracy: 0.7842
Epoch 42/50
1155/1155 [==============================] - 31s 27ms/step - loss: 1.5449 - accuracy: 0.7962 - val_loss: 1.7909 - val_accuracy: 0.7908
Epoch 43/50
1155/1155 [==============================] - 36s 31ms/step - loss: 1.5611 - accuracy: 0.7932 - val_loss: 1.7082 - val_accuracy: 0.8060
Epoch 44/50
1155/1155 [==============================] - 37s 32ms/step - loss: 1.5385 - accuracy: 0.7989 - val_loss: 1.8337 - val_accuracy: 0.7927
Epoch 45/50
1155/1155 [==============================] - 38s 33ms/step - loss: 1.5310 - accuracy: 0.8021 - val_loss: 1.8629 - val_accuracy: 0.7963
Epoch 46/50
1155/1155 [==============================] - 36s 31ms/step - loss: 1.5293 - accuracy: 0.8033 - val_loss: 2.3073 - val_accuracy: 0.7913
Epoch 47/50
1155/1155 [==============================] - 29s 25ms/step - loss: 1.5013 - accuracy: 0.8073 - val_loss: 2.7864 - val_accuracy: 0.8066
Epoch 48/50
1155/1155 [==============================] - 35s 30ms/step - loss: 1.5173 - accuracy: 0.8046 - val_loss: 1.7151 - val_accuracy: 0.7988
Epoch 49/50
1155/1155 [==============================] - 35s 30ms/step - loss: 1.5003 - accuracy: 0.8114 - val_loss: 1.9934 - val_accuracy: 0.7919
Epoch 50/50
1155/1155 [==============================] - 37s 32ms/step - loss: 1.4909 - accuracy: 0.8127 - val_loss: 1.8864 - val_accuracy: 0.7987
```

Figure 6 - Training and validation accuracy and loss

Moreover, the same accuracy and loss as the validation accuracy was achieved when the data generated by prediction was compared with the test dataset as shown in figure 8.

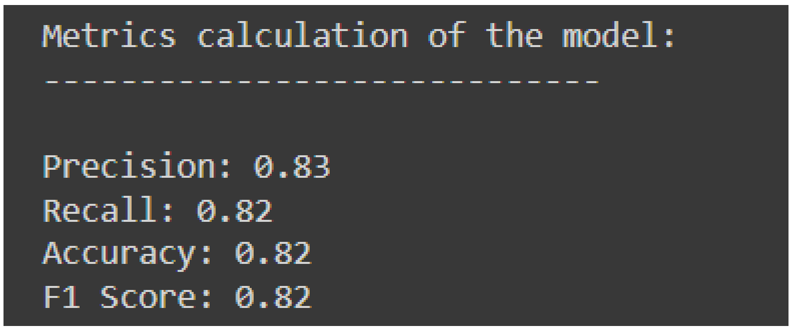


```
evaluate_model = model.evaluate(X_test, y_test)

74/74 [==============================] - 1s 19ms/step - loss: 1.7113 - accuracy: 0.8028
```

Figure 7 - comparing predicted and original values.

To check the overall performance of the model precision, recall, accuracy and F1 accuracy were also calculated.



```
Metrics calculation of the model:
-----------------------------

Precision: 0.83
Recall: 0.82
Accuracy: 0.82
F1 Score: 0.82
```

Figure 8 - Calculation of metrics of the final model.

The overall precision, recall, accuracy and F1 score achieved by the model was 83%, 82%, 82% and 82% respectively as shown in figure 8.

Analysis

This section focusses on providing a brief description of all the results and errors which our group observed while training the model.

Initially, we were struggling to increase the accuracy where validation which was not rising above 1% and the training accuracy was around 12% as shown in figure 10. At this stage we were not aware that the pre-processing also needs to be applied on the validation and the test dataset.

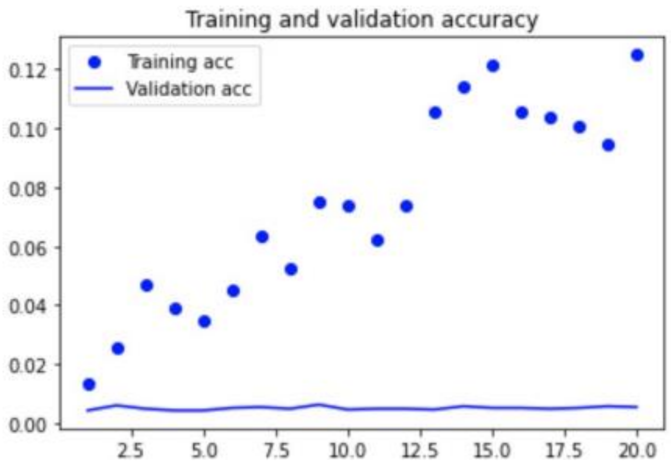
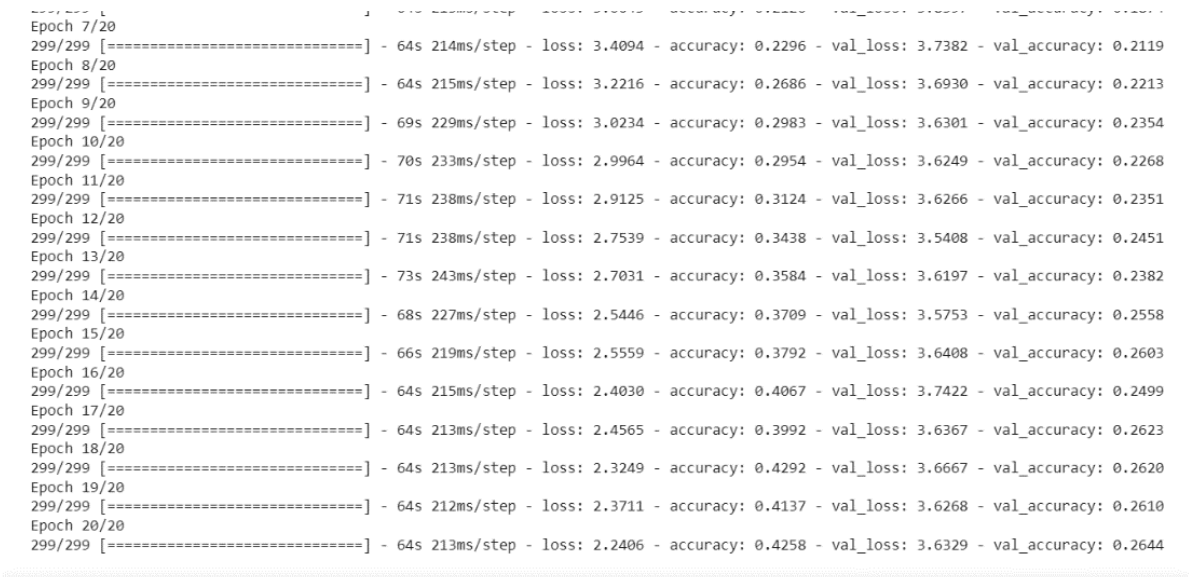


Figure 9 - Low validation and training accuracy in initial model.

Various untrained as well as pretrained model were applied, out of which one was ResNet50 and after using this model we received a validation accuracy of 26% and training accuracy of 42% as shown in figures 11 and 12.



```
Epoch 7/20
299/299 [==============================] - 64s 214ms/step - loss: 3.4094 - accuracy: 0.2296 - val_loss: 3.7382 - val_accuracy: 0.2119
Epoch 8/20
299/299 [==============================] - 64s 215ms/step - loss: 3.2216 - accuracy: 0.2686 - val_loss: 3.6930 - val_accuracy: 0.2213
Epoch 9/20
299/299 [==============================] - 69s 229ms/step - loss: 3.0234 - accuracy: 0.2983 - val_loss: 3.6301 - val_accuracy: 0.2354
Epoch 10/20
299/299 [==============================] - 70s 233ms/step - loss: 2.9964 - accuracy: 0.2954 - val_loss: 3.6249 - val_accuracy: 0.2268
Epoch 11/20
299/299 [==============================] - 71s 238ms/step - loss: 2.9125 - accuracy: 0.3124 - val_loss: 3.6266 - val_accuracy: 0.2351
Epoch 12/20
299/299 [==============================] - 71s 238ms/step - loss: 2.7539 - accuracy: 0.3438 - val_loss: 3.5408 - val_accuracy: 0.2451
Epoch 13/20
299/299 [==============================] - 73s 243ms/step - loss: 2.7031 - accuracy: 0.3584 - val_loss: 3.6197 - val_accuracy: 0.2382
Epoch 14/20
299/299 [==============================] - 68s 227ms/step - loss: 2.5446 - accuracy: 0.3709 - val_loss: 3.5753 - val_accuracy: 0.2558
Epoch 15/20
299/299 [==============================] - 66s 219ms/step - loss: 2.5559 - accuracy: 0.3792 - val_loss: 3.6408 - val_accuracy: 0.2603
Epoch 16/20
299/299 [==============================] - 64s 215ms/step - loss: 2.4030 - accuracy: 0.4067 - val_loss: 3.7422 - val_accuracy: 0.2499
Epoch 17/20
299/299 [==============================] - 64s 213ms/step - loss: 2.4565 - accuracy: 0.3992 - val_loss: 3.6367 - val_accuracy: 0.2623
Epoch 18/20
299/299 [==============================] - 64s 213ms/step - loss: 2.3249 - accuracy: 0.4292 - val_loss: 3.6667 - val_accuracy: 0.2620
Epoch 19/20
299/299 [==============================] - 64s 212ms/step - loss: 2.3711 - accuracy: 0.4137 - val_loss: 3.6268 - val_accuracy: 0.2610
Epoch 20/20
299/299 [==============================] - 64s 213ms/step - loss: 2.2406 - accuracy: 0.4258 - val_loss: 3.6329 - val_accuracy: 0.2644
```

Figure 160 - ResNet50 validation and accuracy.

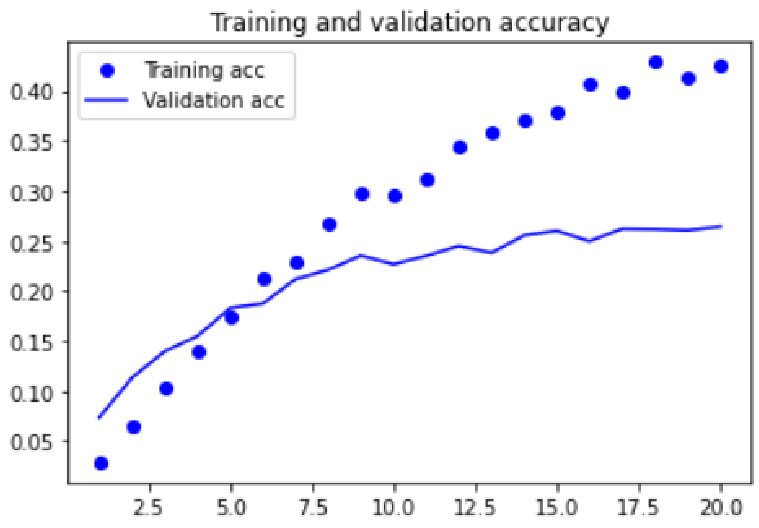
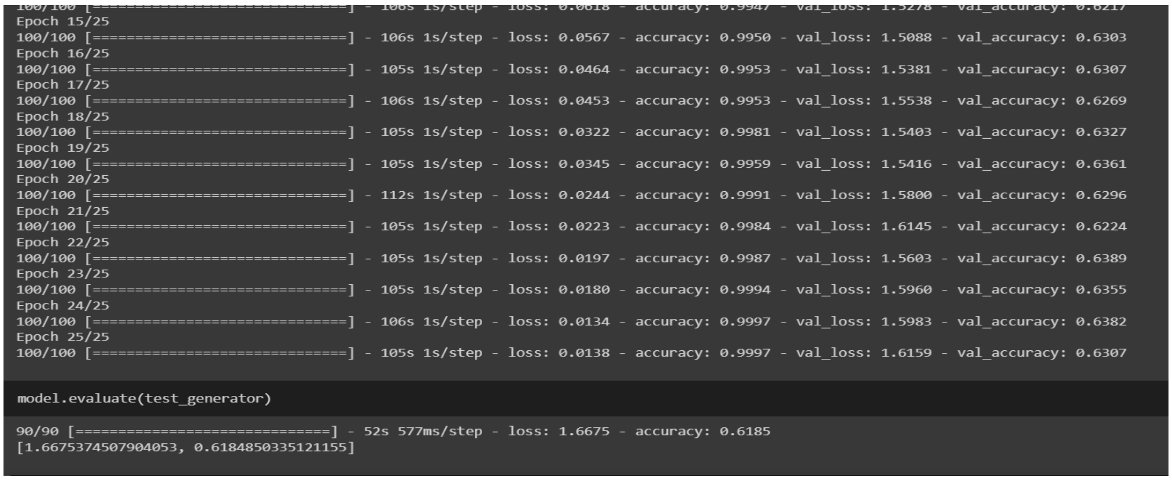


Figure 11 - ResNet50 validation and accuracy graph.

Another pre-trained model was DenseNet169 in which we almost achieved training accuracy of 100% but the validation accuracy was just 63% as shown in figure 13.



```
100/100 [==============================] - 106s 1s/step - loss: 0.0618 - accuracy: 0.9947 - val_loss: 1.5278 - val_accuracy: 0.6217
Epoch 15/25
100/100 [==============================] - 106s 1s/step - loss: 0.0567 - accuracy: 0.9950 - val_loss: 1.5088 - val_accuracy: 0.6303
Epoch 16/25
100/100 [==============================] - 105s 1s/step - loss: 0.0464 - accuracy: 0.9953 - val_loss: 1.5381 - val_accuracy: 0.6307
Epoch 17/25
100/100 [==============================] - 106s 1s/step - loss: 0.0453 - accuracy: 0.9953 - val_loss: 1.5538 - val_accuracy: 0.6269
Epoch 18/25
100/100 [==============================] - 105s 1s/step - loss: 0.0322 - accuracy: 0.9981 - val_loss: 1.5403 - val_accuracy: 0.6327
Epoch 19/25
100/100 [==============================] - 105s 1s/step - loss: 0.0345 - accuracy: 0.9959 - val_loss: 1.5416 - val_accuracy: 0.6361
Epoch 20/25
100/100 [==============================] - 112s 1s/step - loss: 0.0244 - accuracy: 0.9991 - val_loss: 1.5800 - val_accuracy: 0.6296
Epoch 21/25
100/100 [==============================] - 105s 1s/step - loss: 0.0223 - accuracy: 0.9984 - val_loss: 1.6145 - val_accuracy: 0.6224
Epoch 22/25
100/100 [==============================] - 105s 1s/step - loss: 0.0197 - accuracy: 0.9987 - val_loss: 1.5603 - val_accuracy: 0.6389
Epoch 23/25
100/100 [==============================] - 105s 1s/step - loss: 0.0180 - accuracy: 0.9994 - val_loss: 1.5960 - val_accuracy: 0.6355
Epoch 24/25
100/100 [==============================] - 106s 1s/step - loss: 0.0134 - accuracy: 0.9997 - val_loss: 1.5983 - val_accuracy: 0.6382
Epoch 25/25
100/100 [==============================] - 105s 1s/step - loss: 0.0138 - accuracy: 0.9997 - val_loss: 1.6159 - val_accuracy: 0.6307

model.evaluate(test_generator)

90/90 [==============================] - 52s 577ms/step - loss: 1.6675 - accuracy: 0.6185
[1.6675374507904053, 0.6184850335121155]
```

Figure 12- DenseNet169 Training and validation accuracy.

The graphs shown in figure 14, shows the validation accuracy and loss remains constant after reaching a certain point. We had an assumption that it might be because we have not converted an array.

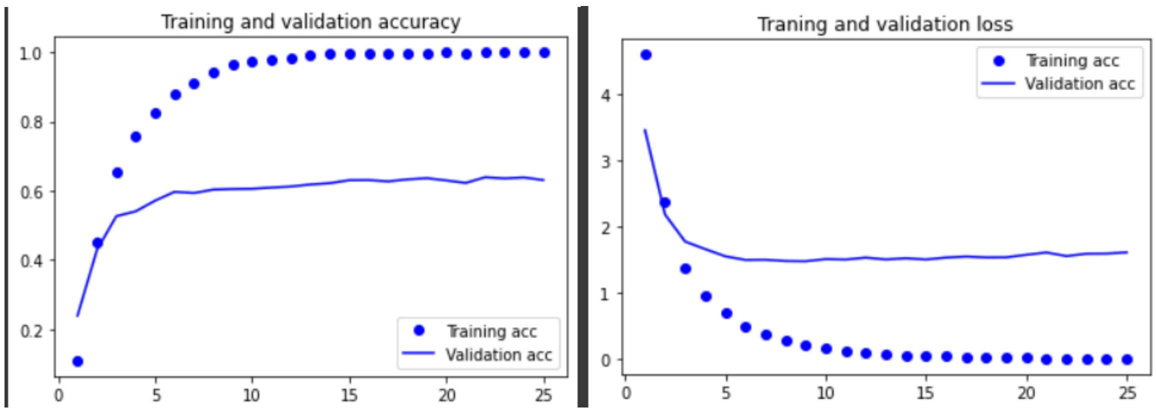
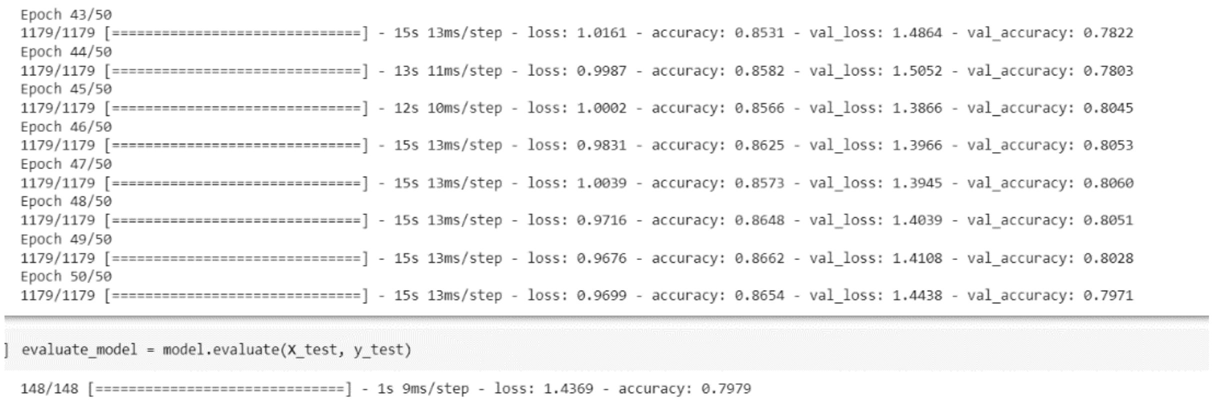


Figure 13 - DenseNet169 training and validation accuracy graphs

We tried converting the images of 256X256 into a NumPy array, but the RAM used to always crash in Google Colab. We tried reducing the frame to 75X75 as this was compatible with various models and, we were not facing the issue of RAM crashes. Moreover, there are approximately twelve thousand images in the data set and for classifying each bird we just have 30 images for training and 30 for test and validation. There were a few changes made to the pre-processing part we introduced the OpenCV library through which the replicas of images were made as discussed earlier in the pre-processing which helped us increase the size of the dataset by 4 times. Conv2D from Keras was applied on the model and we received training, validation accuracy and test accuracy of 86% and 80% and 80% respectively as shown in figures 15 and 16.



```
Epoch 43/50
1179/1179 [==============================] - 15s 13ms/step - loss: 1.0161 - accuracy: 0.8531 - val_loss: 1.4864 - val_accuracy: 0.7822
Epoch 44/50
1179/1179 [==============================] - 13s 11ms/step - loss: 0.9987 - accuracy: 0.8582 - val_loss: 1.5052 - val_accuracy: 0.7803
Epoch 45/50
1179/1179 [==============================] - 12s 10ms/step - loss: 1.0002 - accuracy: 0.8566 - val_loss: 1.3866 - val_accuracy: 0.8045
Epoch 46/50
1179/1179 [==============================] - 15s 13ms/step - loss: 0.9831 - accuracy: 0.8625 - val_loss: 1.3966 - val_accuracy: 0.8053
Epoch 47/50
1179/1179 [==============================] - 15s 13ms/step - loss: 1.0039 - accuracy: 0.8573 - val_loss: 1.3945 - val_accuracy: 0.8060
Epoch 48/50
1179/1179 [==============================] - 15s 13ms/step - loss: 0.9716 - accuracy: 0.8648 - val_loss: 1.4039 - val_accuracy: 0.8051
Epoch 49/50
1179/1179 [==============================] - 15s 13ms/step - loss: 0.9676 - accuracy: 0.8662 - val_loss: 1.4108 - val_accuracy: 0.8028
Epoch 50/50
1179/1179 [==============================] - 15s 13ms/step - loss: 0.9699 - accuracy: 0.8654 - val_loss: 1.4438 - val_accuracy: 0.7971

] evaluate_model = model.evaluate(X_test, y_test)

148/148 [==============================] - 1s 9ms/step - loss: 1.4369 - accuracy: 0.7979
```

Figure 74 - Model performance

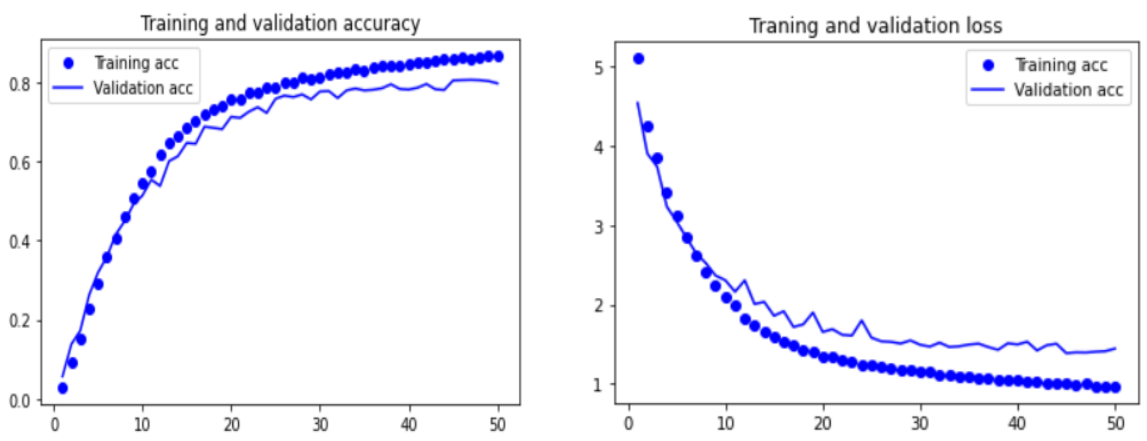


Figure 15 - Model performance graphs.

Here, the model achieved a decent accuracy but there was still room for improvement, so we implemented various other models like VGG19, MobileNet, ResNet50, ResNet101, ResNet152, Xception and many other in search for better accuracy. However, none of the model's performance was good which could be seen in the below figures 17.

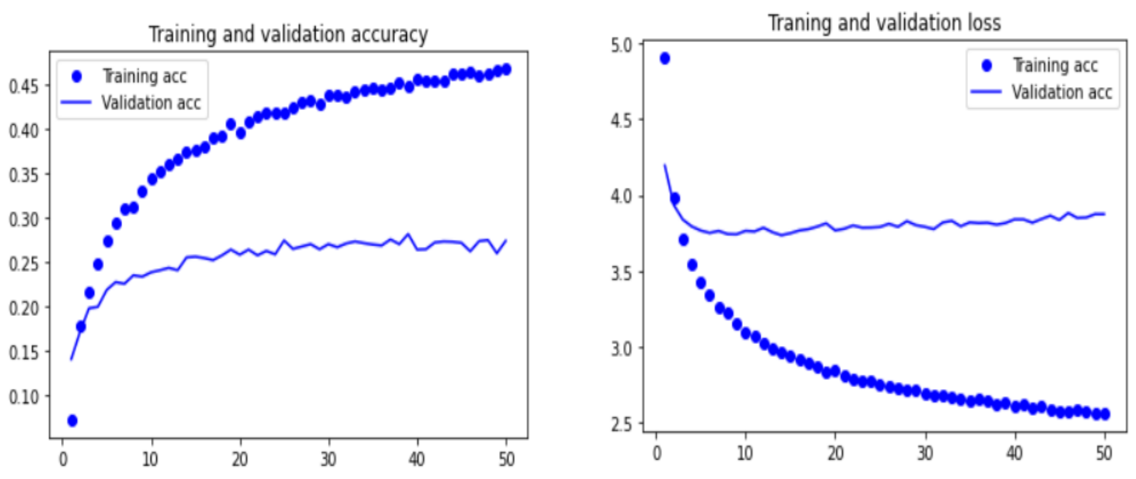


Figure 15 a - BGG19 accuracy and loss

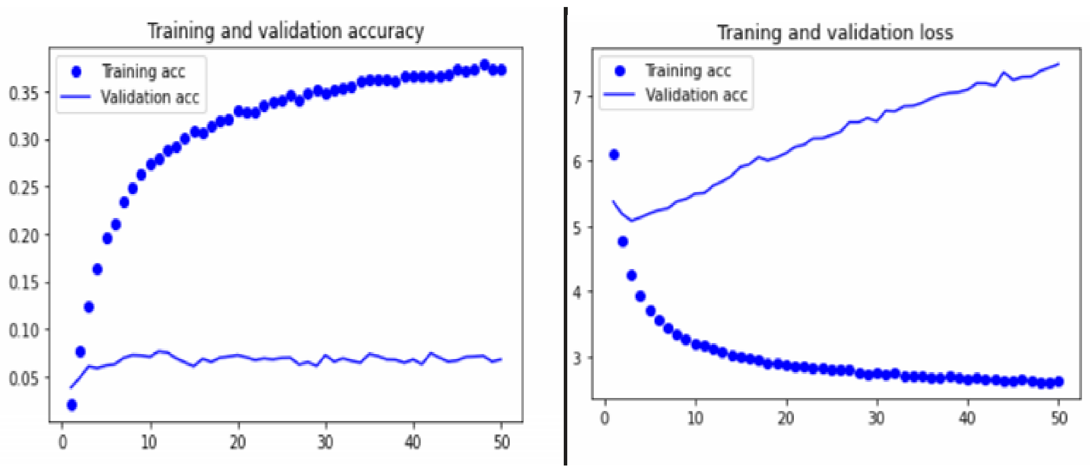


Figure 15 b - MobileNet accuracy and loss

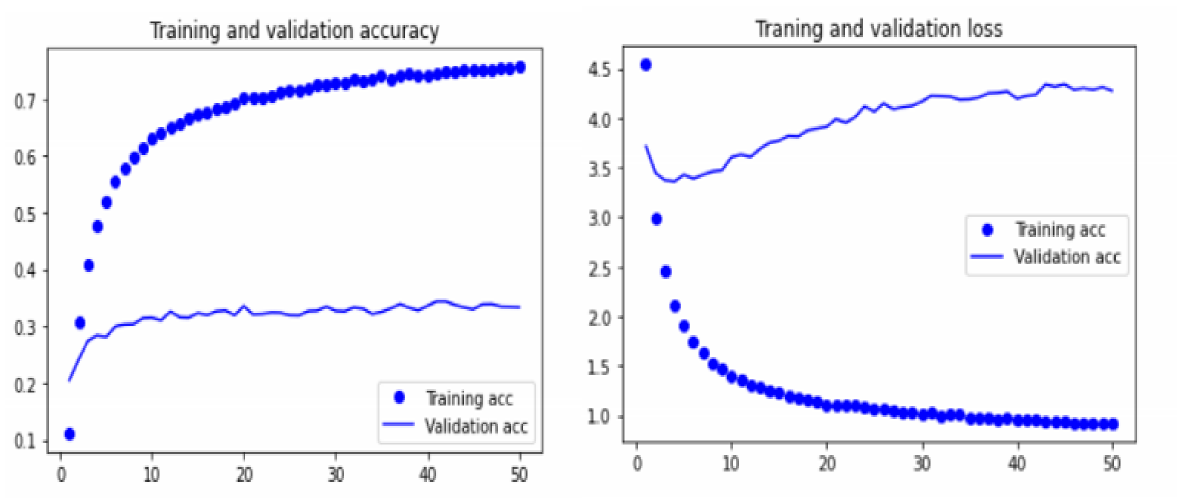


Figure 15 c - ResNet50 accuracy and loss

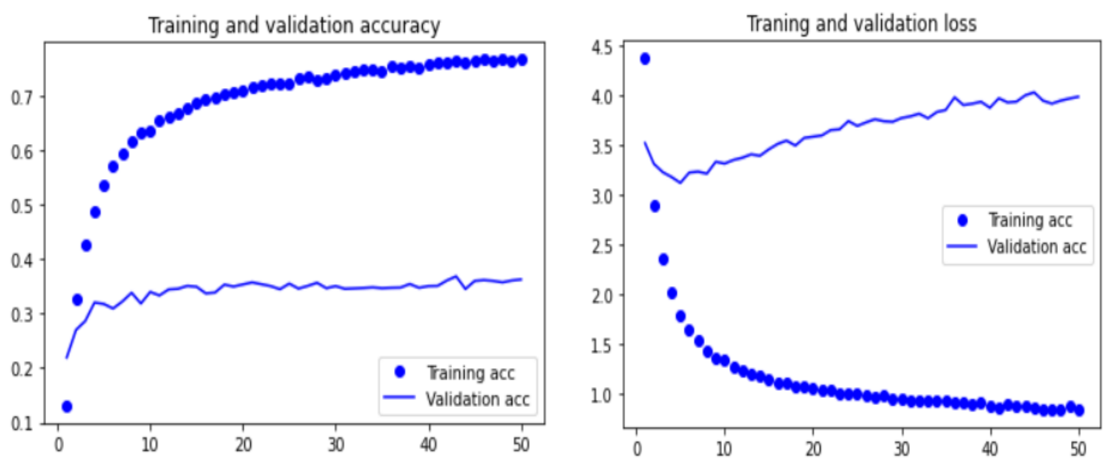


Figure 15d - ResNet101 accuracy and loss

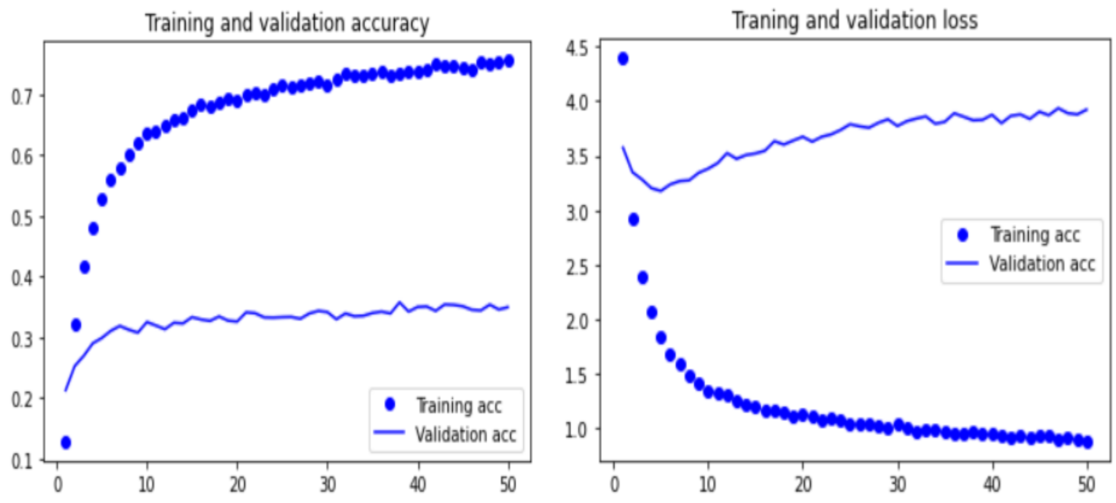


Figure 15 e - ResNet152 accuracy and loss

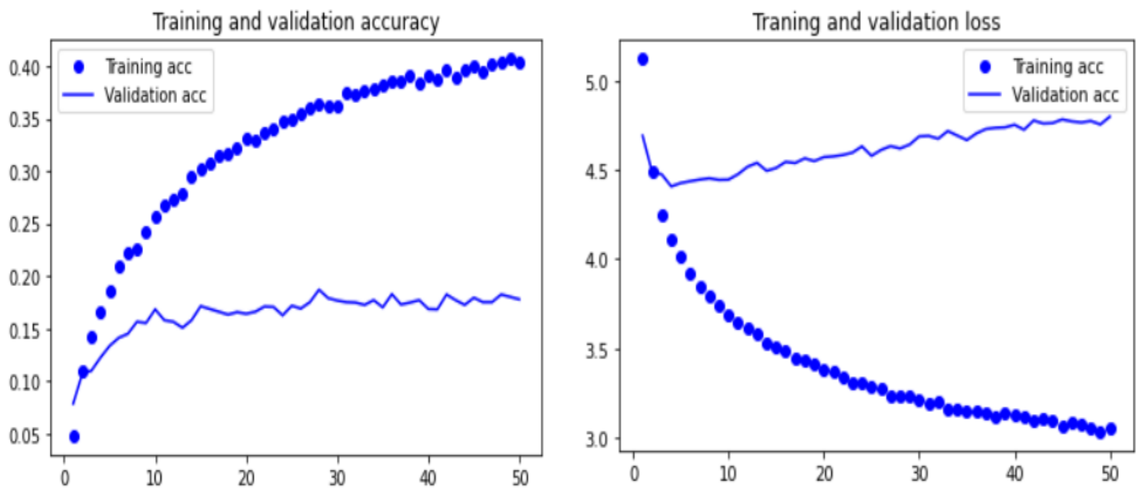
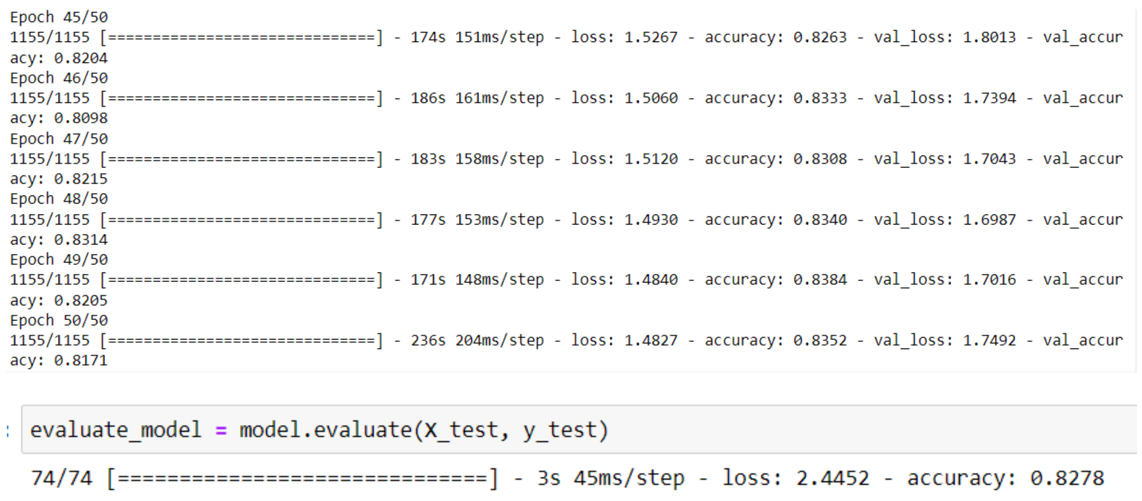


Figure 15 f - Xception accuracy and loss

Models	Training accuracy	Validation accuracy	Test accuracy
Customized CNN	83.5%	81.7%	82.8%
MobileNet	37%	6.8%	6.7%
ResNet50	75%	33%	34.18%
ResNet101	76%	36%	37.50%
ResNet152	75%	35%	36%
Xception	40%	18%	18.89%
VGG19	46%	27%	28%

The results of the customized Conv2D model which we used for training as shown in the figures 17.



```
Epoch 45/50
1155/1155 [==============================] - 174s 151ms/step - loss: 1.5267 - accuracy: 0.8263 - val_loss: 1.8013 - val_accur
acy: 0.8204
Epoch 46/50
1155/1155 [==============================] - 186s 161ms/step - loss: 1.5060 - accuracy: 0.8333 - val_loss: 1.7394 - val_accur
acy: 0.8098
Epoch 47/50
1155/1155 [==============================] - 183s 158ms/step - loss: 1.5120 - accuracy: 0.8308 - val_loss: 1.7043 - val_accur
acy: 0.8215
Epoch 48/50
1155/1155 [==============================] - 177s 153ms/step - loss: 1.4930 - accuracy: 0.8340 - val_loss: 1.6987 - val_accur
acy: 0.8314
Epoch 49/50
1155/1155 [==============================] - 171s 148ms/step - loss: 1.4840 - accuracy: 0.8384 - val_loss: 1.7016 - val_accur
acy: 0.8205
Epoch 50/50
1155/1155 [==============================] - 236s 204ms/step - loss: 1.4827 - accuracy: 0.8352 - val_loss: 1.7492 - val_accur
acy: 0.8171
```

```
evaluate_model = model.evaluate(X_test, y_test)

74/74 [==============================] - 3s 45ms/step - loss: 2.4452 - accuracy: 0.8278
```

Figure 16-a - Training, validation, and test accuracy.

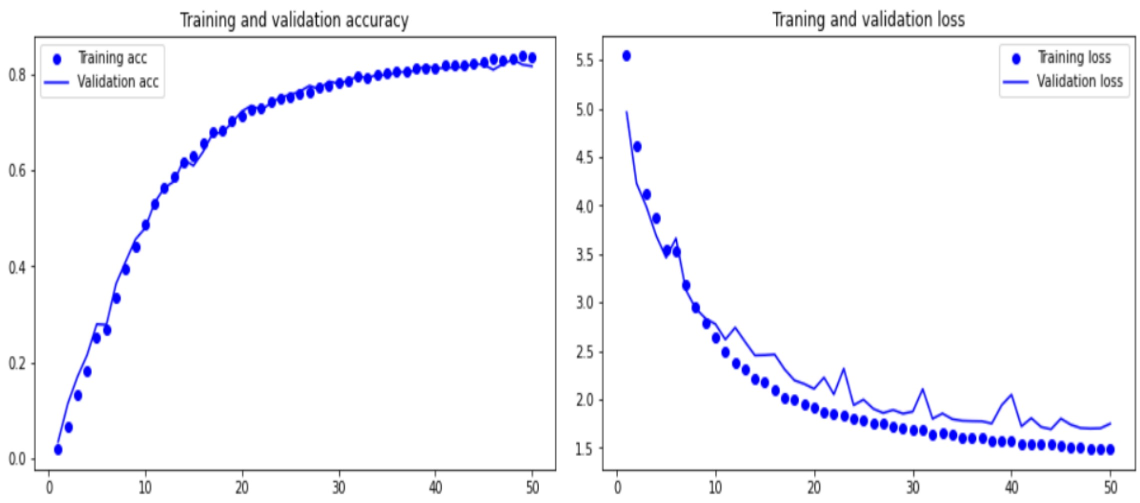


Figure 16-b - Accuracy and Loss graph.

We also tried training the model with different train, test and validation split and we observed that if the validation data set is small then there is more loss.

There was dramatic fluctuation observed when the validation split was 10% and the validation accuracy was more than training accuracy as shown in the figure 18. In both the cases only the train, validation and test split were changed and nothing else.

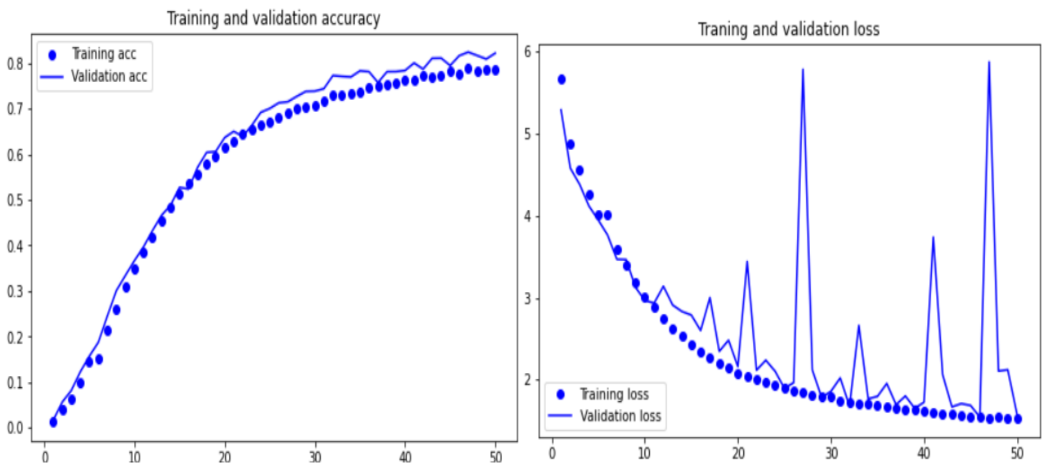


Figure 17- More fluctuation when using small validation dataset.

Finally, we decided to keep the same model and the train, test and validation split was 80%, 5% and 15% respectively.

Literature Review & Related Work

As an emerging but also challenging research topic, visual fine-grained categorization aims at distinguishing between different breeds based on the subtle differences in some characteristics (Welinder 2010). The reason why we are interested in it is that people without specific training can do better in the basic-level recognition, however, fine-grained recognition tends to share the same parts in the basic-level recognition (Yao 2012), and discriminated by the subtle differences, people perform bad without specific expert guidance on this (Reed 2016).

The common assumption in fine-grained recognition is that there are shared semantic parts for all objects (Farrell 2011). Does that mean that all the rich knowledge in fine-grained recognition will never be useful for other areas? Are fine-grained and generic classification so different? How could we free ourselves from the tedious and subjective manual annotations for fine-grained recognition (Krause 2015), which we refer to automatic part discovery? Most previous works rely on object / part level annotations to build part-based representation, which is demanding in practice (Branson 2014). Then from the point of view of practical application, what new methods should we consider proposing?

Zhang et al. (2014) propose a method which outperforms state-of-the-art fine-grained categorization methods in an end-to-end evaluation without requiring a bounding box at test time. It learns detectors and part models and enforces learned geometric constraints between parts and with the object frame. Besides that, in the research of Simon and Rodner (2015) presents an unsupervised approach for the selection of generic parts for fine-grained and generic image classification. Given a CNN pre-trained for classification, a part constellation model is estimated by analyzing the predicted part locations for all training images (Khosla 2011). The resulting model contains a selection of useful part proposals as well as their spatial relationship in different views of the object of interest. Also, in Zhang (2016) research propose an automatic fine-grained recognition approach which is free of any object / part annotation at both training and testing stages, where a unified framework based on two steps of deep filter response pickings. In the research of Lin (2015) presents the bilinear models, a recognition architecture that consists of two feature extractors whose outputs are multiplied using outer product at each location of the image and pooled to obtain an image descriptor.

After understanding some authors' views on fine-grained recognition, we can't help but think about some questions: one of the motivations for the bilinear model was the modular separation of factors that affect the overall appearance (Lin 2015), but do the networks specialize into roles of localization (“where”) and appearance modelling (“what”) when initialized asymmetrically and fine-tuned? Also, there are two main challenges for applying fine-grained classification approaches to other tasks. First, the semantic part detectors need to be replaced by more abstract interest point detectors (Gehler 2019). Second, the selection or training of useful interest point detectors needs to consider that each object class has its own unique shape and set of semantic parts. In the future, we can improve accuracy by studying how to solve the above problems.

Conclusions

During this project, we mostly experimented with the use of different pretrained models to try and improve the accuracy. Some of the pretrained models we worked with did not produce any meaningful results or accuracy, like the ResNet50 model which only returned around 42% accuracy. Some of the pretrained models we experimented with made a far greater impact and did produce some acceptable accuracies. The model DenseNet169 was one of these models, as it achieved a training accuracy of 99.97%. This, however, did not indicate that it was a successful model. It suffered from overfitting and only gave 63% accuracy on the validation set. Eventually we found that we were better off not using a pretrained model at all. The final model we used, which was created from untrained layers, provided the best performance. Using multiple convolutional layers to make a convolutional neural network meant that its performance on images was good. And the dropout and normalisation layers helped to prevent overfitting.

This shows that while pretrained models may be good for some tasks, they are not always the best choice when making an image classifying network from scratch. For this, it is better to create and tune a model for the task at hand.

To experiment this further we could try many different layers or parameters. Our best model consisted of six convolutional layers, in future we could improve this by including more of these layers or maybe by changing the parameters of these layers. We could also potentially try modifying the dropout and normalisation layers of the model to see if this results in a better fit for the data.

Another potential improvement which was mentioned in the literature review is using attribute parts in class detection. Currently these attributes need to be gathered by humans, but if a method for automatically detecting and identifying parts of birds could be developed, this could be incorporated into a bird species classifier with potentially far greater accuracy and range then the one we have created.

References

[1] Branson, S., Van Horn, G., Belongie, S., Perona, P., 2014. Bird species categorization using pose normalized deep convolutional nets. pp. 1406–2952.

[2] Farrell, R., Oza, O., Zhang, N., Morariu, V.I., Darrell, T., Davis, L.S., 2011. Birdlets: Subordinate categorization using volumetric primitives and pose-normalized appearance, in: 2011 International Conference on Computer Vision. IEEE, pp. 161–168.

[3] Gehler, P., Nowozin, S., 2009. On feature combination for multiclass object classification, in: 2009 IEEE 12th International Conference on Computer Vision. IEEE, pp. 221–228.

[4] Khosla, A., Jayadevaprakash, N., Yao, B., Li, F.-F., 2011. Novel dataset for fine-grained image categorization: Stanford dogs, in: Proc. CVPR Workshop on Fine-Grained Visual Categorization (FGVC). Citeseer.

[5] Krause, J., Jin, H., Yang, J., Fei-Fei, L., 2015. Fine-grained recognition without part annotations, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. pp. 5546–5555.

[6] Lin, T.-Y., RoyChowdhury, A., Maji, S., 2015. Bilinear CNN Models for Fine-Grained Visual Recognition. Presented at the Proceedings of the IEEE International Conference on Computer Vision, pp. 1449–1457.

[7] Reed, S., Akata, Z., Lee, H., Schiele, B., 2016. Learning deep representations of fine-grained visual descriptions, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. pp. 49–58.

[8] Simon, M., Rodner, E., 2015. Neural activation constellations: Unsupervised part model discovery with convolutional networks, in: Proceedings of the IEEE International Conference on Computer Vision. pp. 1143–1151.

[9] Welinder, P., Branson, S., Mita, T., Wah, C., Schroff, F., Belongie, S., Perona, P., 2010. Caltech-UCSD Birds 200 [WWW Document]. URL https://resolver.caltech.edu/CaltechAUTHORS:20111026-155425465 (accessed 5.8.22).

[10] Yao, B., Bradski, G., Fei-Fei, L., 2012. A codebook-free and annotation-free approach for fine-grained image categorization, in: 2012 IEEE Conference on Computer Vision and Pattern Recognition. IEEE, pp. 3466–3473.

[11] Zhang, N., Donahue, J., Girshick, R., Darrell, T., 2014. Part-based R-CNNs for fine-grained category detection, in: European Conference on Computer Vision. Springer, pp. 834–849.

[12] Zhang, X., Xiong, H., Zhou, W., Lin, W., Tian, Q., 2016. Picking deep filter responses for fine-grained image recognition, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. pp. 1134–1142.