Machine Learning and Content Analytics

Deep Learning Project

Κανδυλάκης Μαρίνος

Αριθμός Μητρώου: p2822117

Επιβλέπων Καθηγητής:

Γ. Περάκης

Table of Contents

Introduction

The project's idea was to identify dog breeds by looking at images from any angle they come from but the face of the dog is required to be inside the image. Different Convolutional Neural Network architectures were used in this project in order to be able to achieve the maximum efficiency on the predictions.

The project's code was implemented through Google Colab's Python and for Storage we used Google Drive. The Tensorflow library was used to create our model architectures and the OpenCV library was used in order to read and manipulate the dog images.

In later stages of this project could evolve into an application for our mobile phones where we could take a picture of a dog and provide us what kind of breed it is and also some basic characteristics of the breed i.e. Average Age, where it comes from, etc.

Dataset Acquisition - Overview - Preprocessing

Dataset Acquisition

Our Data came from ImageNet and it was the Stanford Dog Dataset which consisted of 20 thousand images of 120 different dog breeds.

In the beginning the idea was to combine two different datasets of 30 thousand images with 190 dog breeds but the idea was abandoned due to memory limitations in the training part of the project and not having enough data to be able to correctly classify 190 classes. Thus, the project continued with the original dataset from ImageNet.

Another methodology was to scrape the images of off the internet and multiple websites. However, the time consumption and management did not allow for such methodology to be done.

Dataset Overview

The original dataset consisted of 20.580 images over 120 classes as stated above. On average each class had 171 images representing it with a standard deviation of 23.

Below we can see a few examples of the Dog Images.

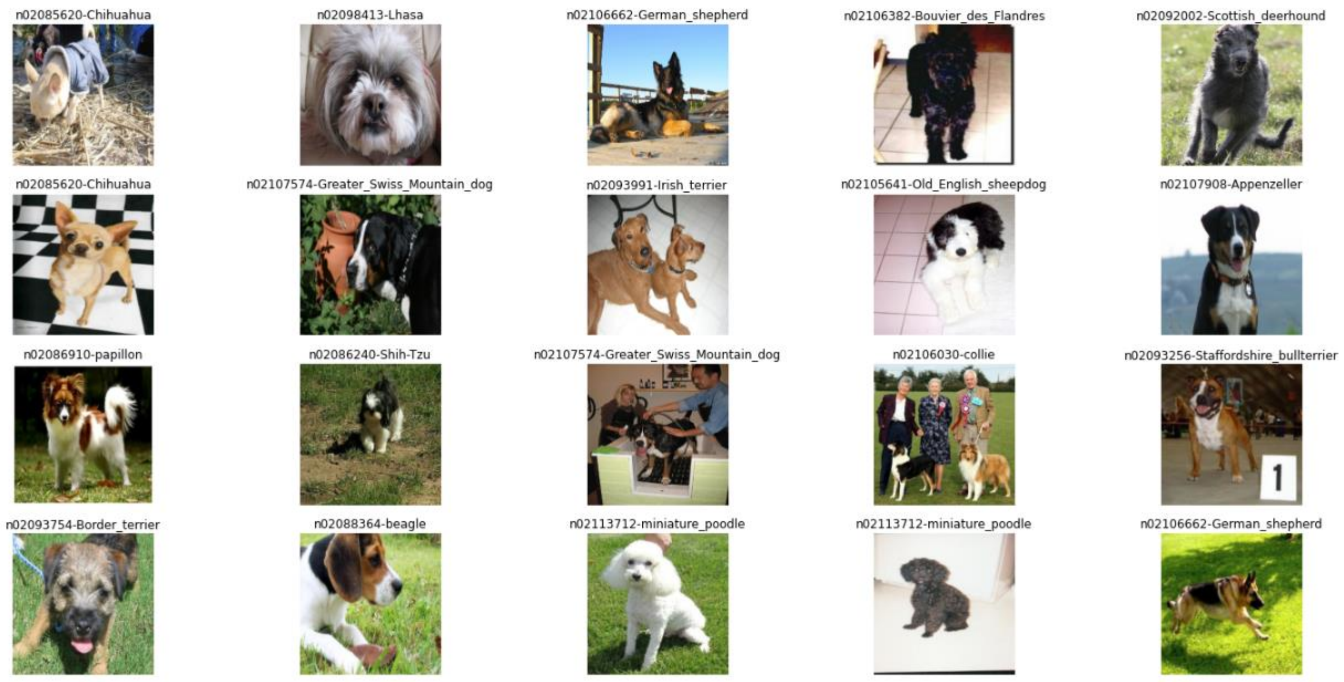

n02085620-Chihuahua
n02098413-Lhasa
n02106662-German_shepherd
n02106382-Bouvier_des_Flandres
n02092002-Scottish_deerhound
n02085620-Chihuahua
n02107574-Greater_Swiss_Mountain_dog
n02093991-Irish_terrier
n02105641-Old_English_sheepdog
n02107908-Appenzeller
n02086910-papillon
n02086240-Shih-Tzu
n02107574-Greater_Swiss_Mountain_dog
n02106030-collie
n02093256-Staffordshire_bullterrier
n02093754-Border_terrier
n02088364-beagle
n02113712-miniature_poodle
n02113712-miniature_poodle
n02106662-German_shepherd

Figure 1.1: Dog Images from different aspects and angles

From what we can understand from the above images is that the dogs are shown in many different angles along with other dogs and some with their owners. By creating bounding boxes, we could help our models identify quicker the dog in the image and try to classify the dog only driven by the bounding box and not get affected by other elements of the picture.

Below we have plotted also the distribution of the top 50 dog breeds.

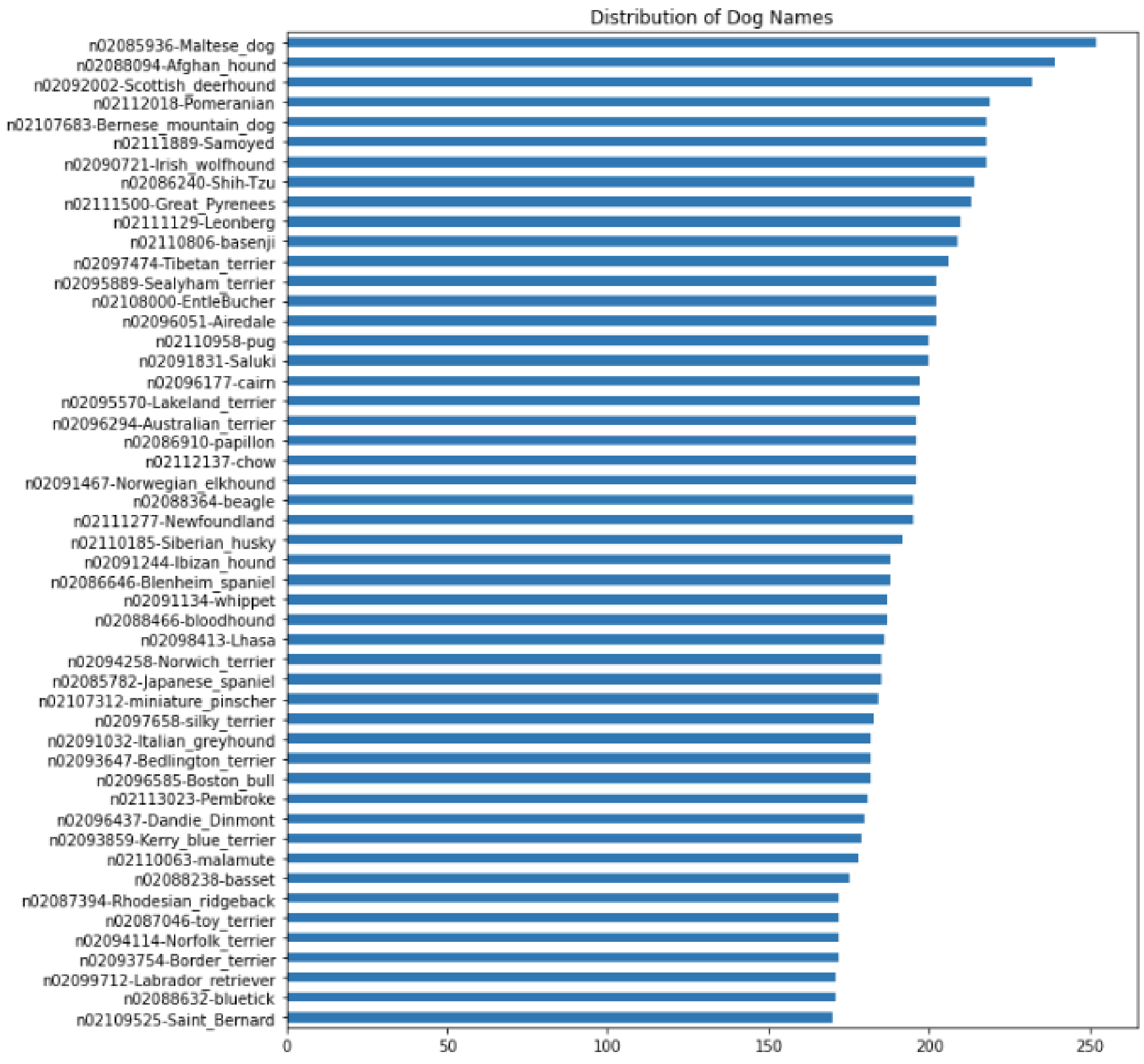


Figure 1.2: Dog Breed count and distribution

As we can see the “Maltese_dog” breed has the most images in its directory. The distribution of the dogs looks almost uniformal with the exception of some outlier breeds on top of the plot.

Dataset Pre-processing

Dataset Split

The Dataset structure was one folder with 120 subfolders representing each dog breed. Inside each subfolder you could find the images of the corresponding breed dog.

This structure although convenient in order to identify which label goes to which image it did not serve the purpose of training our models. Thus, it needed to be altered.

We split the dataset into three separate datasets:

1. The Train Dataset
2. The Validation Dataset
3. The Test Dataset

The Train Dataset was used in order to train our models. The validation dataset would help us evaluate how good the model is and adjust parameters based on metrics. Finally, the Test Dataset will be considered as an out of sample evaluation in order to see who the model behaves in unknown data.

The split was done through random probability generation of a number between 0 – 1. That being said if the probability was between [0 , 0.8] then the image would go in the Train dataset, if the probability was between (0.8 , 0.9] then the image would be copied into the validation dataset and lastly, if the probability was between (0.9 , 1] then the image would be copied into the Test dataset. This procedure was applied to all dog breeds.

In the end we ended up with the train dataset having 80% of the original dataset’s images and validation and test dataset had 10% of the original dataset’s images respectively. Of course, the same ratio was strictly kept in each class of the dogs in order to avoid having imbalanced classes between datasets.

Dataset Process

Through the OpenCV library available in Python we read each image and then converted it from a BGR format into an RGB format. Then the image would go into a resizing state of Image Height and Image Width of 224 pixels.

The reason of the resizing is that the models can only be trained with images of the same size and we need to make the image smaller than its original state else we would need a huge amount of RAM in order to store all the pixel values in higher resolutions.

The resulting Image would be a Tensor of shape (224 , 224 , 3).

- 224 pixels height
- 224 pixels width
- 3 channels of colors

All these images would go into a list according from which dataset they were coming from (Train , Test, Validation) along with their corresponding class / dog breed.

In addition, these lists were broken into two numpy arrays each. One array that would keep all the Tensor data we will call that x_array and the array would have all the labelling of each Tensor we would call that y_array.

The final arrays coming from the lists were as follow:

- X_train with shape (16411, 224, 224, 3) and y_train with shape (16411,)
- X_valid with shape (2075, 224, 224, 3) and y_valid with shape (2075,)
- X_test with shape (2094, 224, 224, 3) and y_test with shape (2094,)

We need to keep in mind that the first value in each array is the number of samples or Images.

The normalizing/rescaling of the data which needs to be done so that the model can learn and be trained correctly was done inside the model for memory optimization purposes.

Due to memory limitations another method was also implemented in order to read and process all of the images. The idea was to read and process the images in small batches (32 images per batch) and feed those into the neural network model and then read the next batch and so on. In that way we optimized our ram efficiency and never ended up in memory overflowing.

When working with many images or very large datasets the second methodology is more efficient and will not cause any issues. The first method is faster due to the fact that everything is stored in memory but it should only be used on small scale datasets.

In this project both methods were used however, the second method was more optimal for the purpose of this project.

Data Augmentation

Data augmentation was done in the images such as rotating the image into some degree and mirroring the image. However, we did not see any minor improvements in the training of our models. Thus, data augmentation technique was not implemented.

However, while reading images in batches through the second method, we used the rescaling parameter of the data augmentation technique in python so that we reduced our pixel value range from [0 , 255] into [0 , 1] although this can be done through other functions and techniques we did it from this one as it seemed to be the most memory efficient.

Neural Network Models

For the Machine learning purposes of this project we used the Convolutional Neural Network (CNN) architecture. CNNs are feed forward fully connected neural networks. The reason for using CNNs is parameter optimization. Neural network's parameters can grow exponentially with each added layer ending in a very computationally heavy training for the model. CNNs take very little time for tuning those parameters.

Furthermore, they can do dimensionality reduction of the images without sacrificing the information that there is. The dimensionality reduction is done through the convolution operation. This operation ensures only the important features of the classification are sent into the neural network and also keeps the computational task for training the model to the minimum.

The result of the convolution operation is referred to as a feature map. Applying filters is what leads to the feature map. The filter is usually a 3 x 3 matrix or 2 x 2 doing a multiplication of the filter, summing the values and outputs them into the feature map. This is done by sliding the convolution filter onto the input image. The sliding usually steps pixel by pixel and its called strides.

After the convolution filter is applied we use an activation function called ReLU (Rectified Linear Unit) in order to make sure non-linearity is applied. There are many activation functions to go through but ReLU is one of the most commonly used thus we chose this activation function for our models.

After the convolutional operation and the activation function there is another operation called pooling. Through pooling we take the feature map and reduce it even further. A common pooling method is called MaxPooling. In other words, it's like applying a convolutional filter again upon a feature map. The MaxPooling technique is usually a 2 x 2 or 3 x 3 matrix that slides over the feature map and gets the largest value in the matrix.

Once the convolutional layers are done, we have a procedure called Flattening. This process transforms the result of pooling into one column that goes now through our fully connected layers. This process needs to be done between the convolutional operations and the fully connected layers.

The last fully connected layer is our output layer and is as big as the classes of our classification problem, in our situation it is 120 since we have 120 classes we want to predict. There is also an activation function here called Softmax which is used for multiclass classification problems.

Dog Detector

Before proceeding with the neural network models for the image classification we will create a Dog Detector algorithm that will firstly understand if the input image has a dog inside or not. If the image has a dog inside then the image will be fed directly into the model. If not then we will request for another image.

We defined 2 functions, the first one will take the image preprocess it and return a tensor which will be fed into the dog detector. The second function is our Dog Detector algorithm which takes as input the tensor from the first function and passes it through a CNN Pretrained model called *ResNet50*. The ResNet50 will apply a prediction upon the image its fed. If the prediction number is between 151 and 268 then the picture represents a dog else the model could not classify this image having a dog inside it.

After going through the documentation of ResNet50 the predictions between 151-268 represent dog classes and that is the reason we expect a number during that margin. It is okay if the model cannot understand correctly what class the dog is that is not its purpose. The purpose is to understand that it has to do with a dog in the image and try to classify it during that range of predictions.

Feeding for prediction all of our Images the model got an accuracy of 96.70% out of the 20.580 images. Around 679 images were misclassified.

Model From Scratch

Our model consists of 274.424 Trainable parameters and 3.072 Non-Trainable parameters adding up to a total of 277.496 parameters. We used a batch size of 32 images in order to train the model in a total of 20 epochs.

We managed to get a validation accuracy of 19.18% and on the Test dataset we achieved a 20.77% accuracy.

We can see that in the later stages of the epochs our model started to over fit without increasing in the validation accuracy.

The loss of the model was computed through the "Sparse Categorical Crossentropy" and also used the optimizer RMSprop with a learning rate of 0.001 .

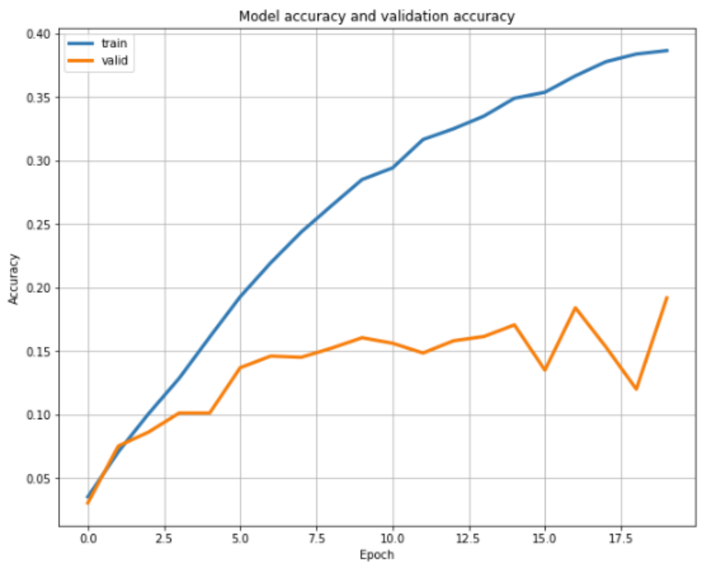
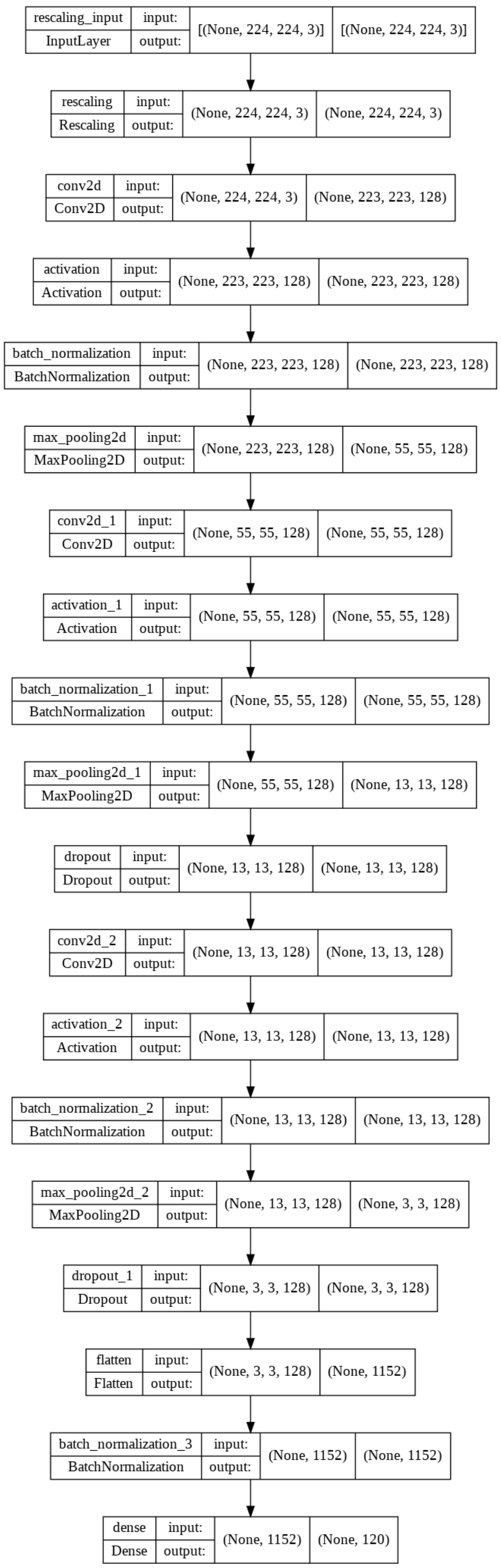


Figure 1.3: MyModel Accuracy over Epoch


rescaling_input InputLayer input: output: [(None, 224, 224, 3)] [(None, 224, 224, 3)]
rescaling Rescaling input: output: (None, 224, 224, 3) (None, 224, 224, 3)
conv2d Conv2D input: output: (None, 224, 224, 3) (None, 223, 223, 128)
activation Activation input: output: (None, 223, 223, 128) (None, 223, 223, 128)
batch_normalization BatchNormalization input: output: (None, 223, 223, 128) (None, 223, 223, 128)
max_pooling2d MaxPooling2D input: output: (None, 223, 223, 128) (None, 55, 55, 128)
conv2d_1 Conv2D input: output: (None, 55, 55, 128) (None, 55, 55, 128)
activation_1 Activation input: output: (None, 55, 55, 128) (None, 55, 55, 128)
batch_normalization_1 BatchNormalization input: output: (None, 55, 55, 128) (None, 55, 55, 128)
max_pooling2d_1 MaxPooling2D input: output: (None, 55, 55, 128) (None, 13, 13, 128)
dropout Dropout input: output: (None, 13, 13, 128) (None, 13, 13, 128)
conv2d_2 Conv2D input: output: (None, 13, 13, 128) (None, 13, 13, 128)
activation_2 Activation input: output: (None, 13, 13, 128) (None, 13, 13, 128)
batch_normalization_2 BatchNormalization input: output: (None, 13, 13, 128) (None, 13, 13, 128)
max_pooling2d_2 MaxPooling2D input: output: (None, 13, 13, 128) (None, 3, 3, 128)
dropout_1 Dropout input: output: (None, 3, 3, 128) (None, 3, 3, 128)
flatten Flatten input: output: (None, 3, 3, 128) (None, 1152)
batch_normalization_3 BatchNormalization input: output: (None, 1152) (None, 1152)
dense Dense input: output: (None, 1152) (None, 120)

We created a model from scratch CNN which consisted of the Rescaling layer in order to scale down our images between [0 , 1] , 3 Convolutional layers, 3 MaxPooling layers and only the Output Dense layer.

The filter size of the convolutional layer is at 126. The kernel size was 2 x 2 with a stride of 1.

After the convolutional Layer we have the “ReLU” activation function and then we added a Batch Normalisation layer in order to help our model learn better.

Proceeding with the batch Normalisation layer we go the MaxPooling layer. In this layer we have set a pool size of 2x2 over which we will take the maximum value over the pooling window.

The same parameters were put in all Convolutional and MaxPooling layers.

Lastly, we have the Flatten and the Fully connected layer. The last Dense layer has an output of 120 (as many as our classes) equipped with a “Softmax” activation function.

More than 90 different models where trained with a maximum accuracy on the validation set to be around 20% .

The model managed to predict on the out of sample test dataset an accuracy of 20.8% .

VGG16 Pretrained Model

The next model was the VGG16 pretrained model with the weights used from “imagenet”. It needs to be specified that we the layers of the model were not trained and they were left intact with their weights. The model was trained in 25 epochs The loss of the model was computed through the “Sparse Categorical Crossentropy” and also used the optimizer RMSprop with a learning rate of 0.001 .

The model managed to achieve a 50.70% accuracy on the Validation set and a 51.09% accuracy on the out of sample test dataset.

Below we can see the Accuracy plot of the model per Epoch.

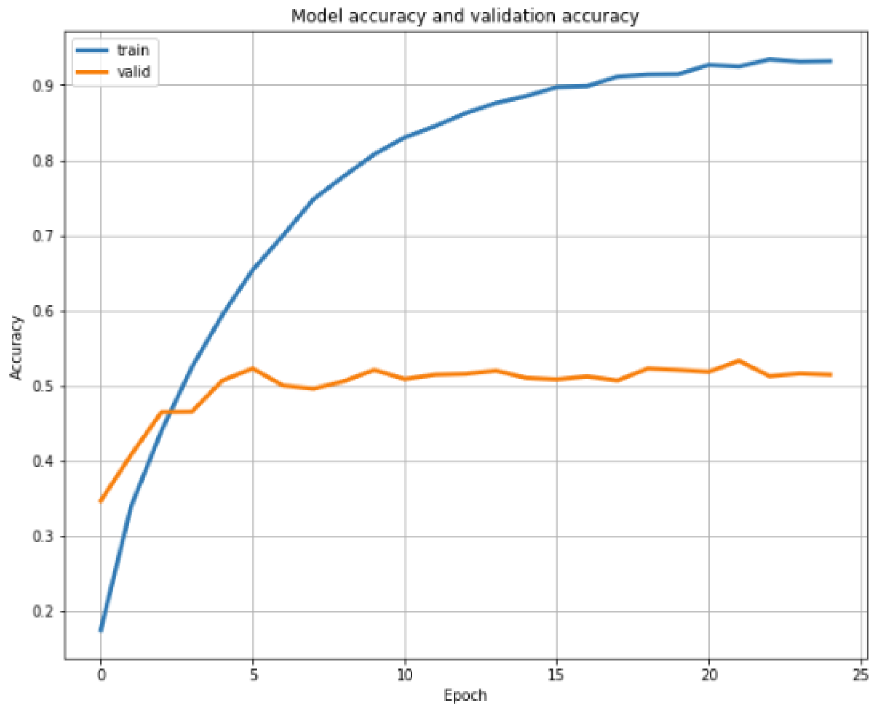
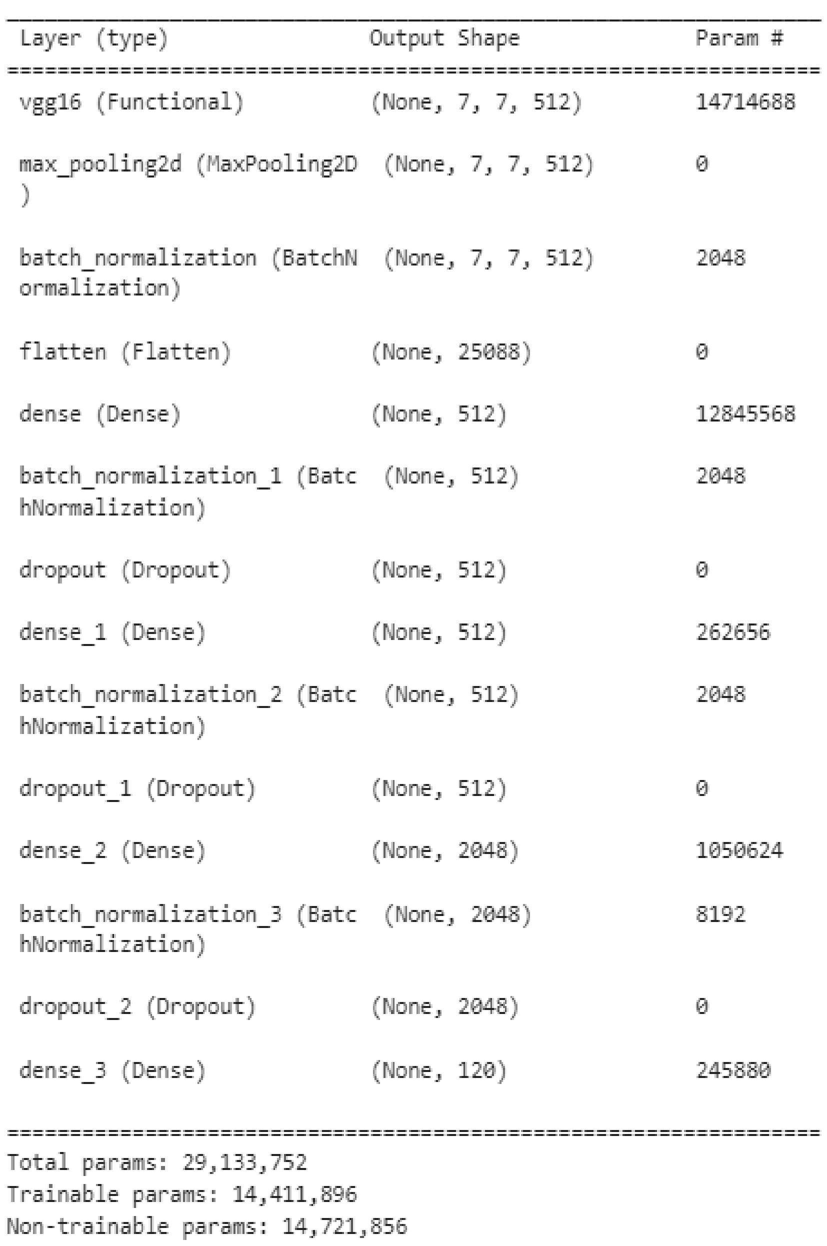


Figure 1.4: VGG16 Model Accuracy over Epoch

The unexpected thing with the VGG16 Model was that the model achieved a 51% validation accuracy without *Rescaling* out data. When the data were rescaled the model achieved a validation accuracy of 30%. This needs to be investigated further as to the reasons behind it.



```
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 vgg16 (Functional)          (None, 7, 7, 512)         14714688  
                                                                 
 max_pooling2d (MaxPooling2D  (None, 7, 7, 512)        0         
 )                                                               
                                                                 
 batch_normalization (BatchN  (None, 7, 7, 512)        2048      
 ormalization)                                                   
                                                                 
 flatten (Flatten)           (None, 25088)             0         
                                                                 
 dense (Dense)               (None, 512)               12845568  
                                                                 
 batch_normalization_1 (Batc  (None, 512)              2048      
 hNormalization)                                                 
                                                                 
 dropout (Dropout)           (None, 512)               0         
                                                                 
 dense_1 (Dense)             (None, 512)               262656    
                                                                 
 batch_normalization_2 (Batc  (None, 512)              2048      
 hNormalization)                                                 
                                                                 
 dropout_1 (Dropout)         (None, 512)               0         
                                                                 
 dense_2 (Dense)             (None, 2048)              1050624   
                                                                 
 batch_normalization_3 (Batc  (None, 2048)             8192      
 hNormalization)                                                 
                                                                 
 dropout_2 (Dropout)         (None, 2048)              0         
                                                                 
 dense_3 (Dense)             (None, 120)               245880    
                                                                 
=================================================================
Total params: 29,133,752
Trainable params: 14,411,896
Non-trainable params: 14,721,856
```

The model Consists of 29.133.752 parameters. On top of the VGG16 Model we added a MaxPooling layer with a pool size of 2x2 followed by a Batch Normalisation layer and a Flatten layer.

Then 3 Fully Connected Dense Layers are added. Two layers consist of 512 units and the third layer consists of 2048 units. All layers have an activation function “ReLU” and the final Output Dense layer which is the same in all our models.

Xception Pretrained Model

The next model was the Xception pretrained model with the weights used from “imagenet”. We used the Xception pretrained model for feature extraction from our Train, Validation and Test dataset and fed those features into our model.

The model was trained in 25 epochs The loss of the model was computed through the “Sparse Categorical Crossentropy” and also used the optimizer RMSprop with a learning rate of 0.001 .

The model managed to achieve a 79.76% accuracy on the Validation set and on the out of sample Test dataset we got a 78.08%

Below we can see the Accuracy plot of the model per Epoch.

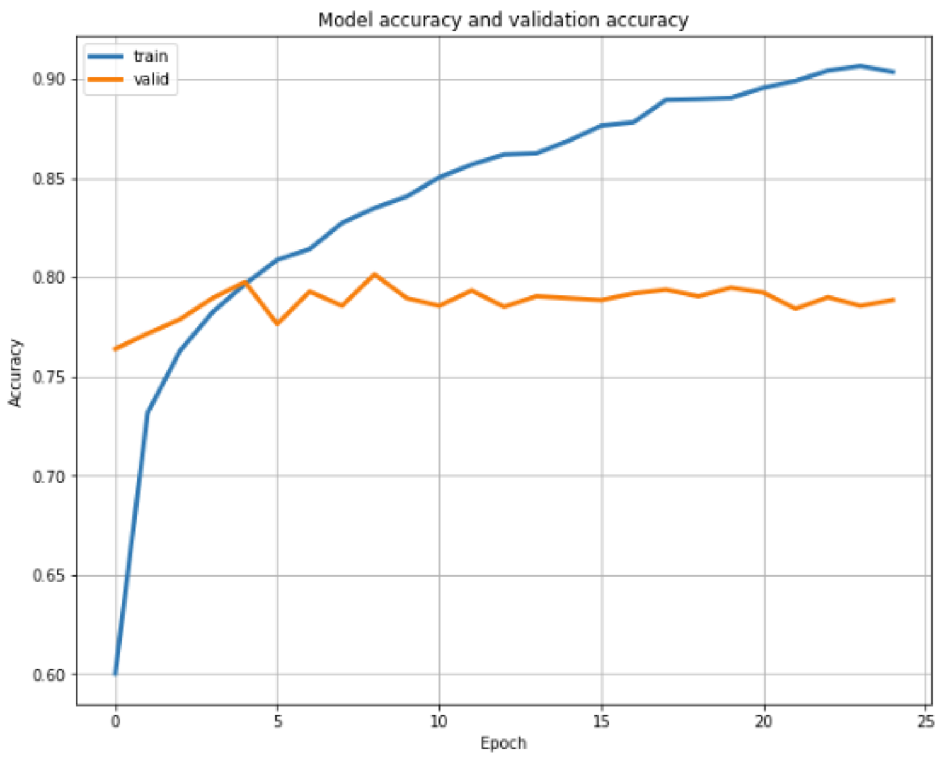
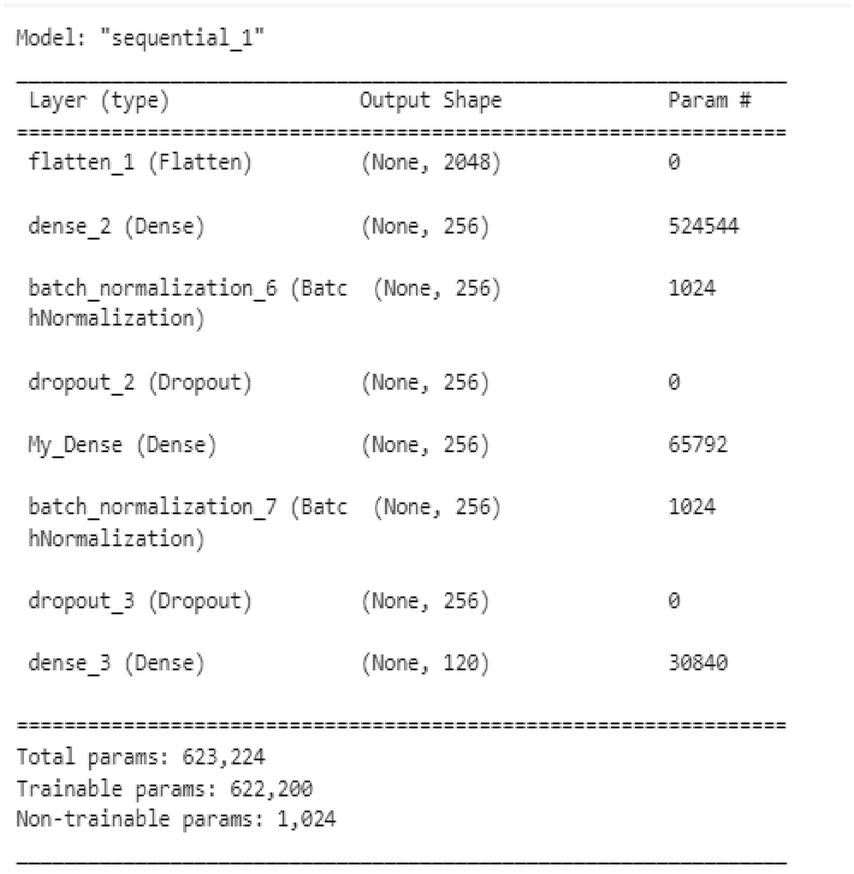


Figure 1.4: Xception Model Accuracy over Epoch



```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten_1 (Flatten)         (None, 2048)              0

 dense_2 (Dense)             (None, 256)               524544

 batch_normalization_6 (Batc  (None, 256)              1024
 hNormalization)

 dropout_2 (Dropout)         (None, 256)               0

 My_Dense (Dense)            (None, 256)               65792

 batch_normalization_7 (Batc  (None, 256)              1024
 hNormalization)

 dropout_3 (Dropout)         (None, 256)               0

 dense_3 (Dense)             (None, 120)               30840

=================================================================
Total params: 623,224
Trainable params: 622,200
Non-trainable params: 1,024
_________________________________________________________________
```

The Xception model has a Total parameters of 623.224 in addition to the 22.8 million parameters of the pretrained Xception model.

We added 2 Dense layers of 256 units and our final Dense output layer.

We also used Batch Normalisation layers and Dropout layers of 40% in order for the model to learn better and not to overfit.

InceptionV3 Pretrained Model

We also used the InceptionV3 pretrained model with the weights used from “imagenet”. We did feature extraction from our Train, Validation and Test dataset and fed those features into our final model.

The model was trained in 15 epochs. The loss of the model was computed through the “Sparse Categorical Crossentropy” and also used the optimizer RMSprop with a learning rate of 0.001 .

The model managed to achieve a 80.0% accuracy on the Validation set and on the out of sample Test dataset 79.13% accuracy.

Below we can see the Accuracy plot of the model per Epoch.

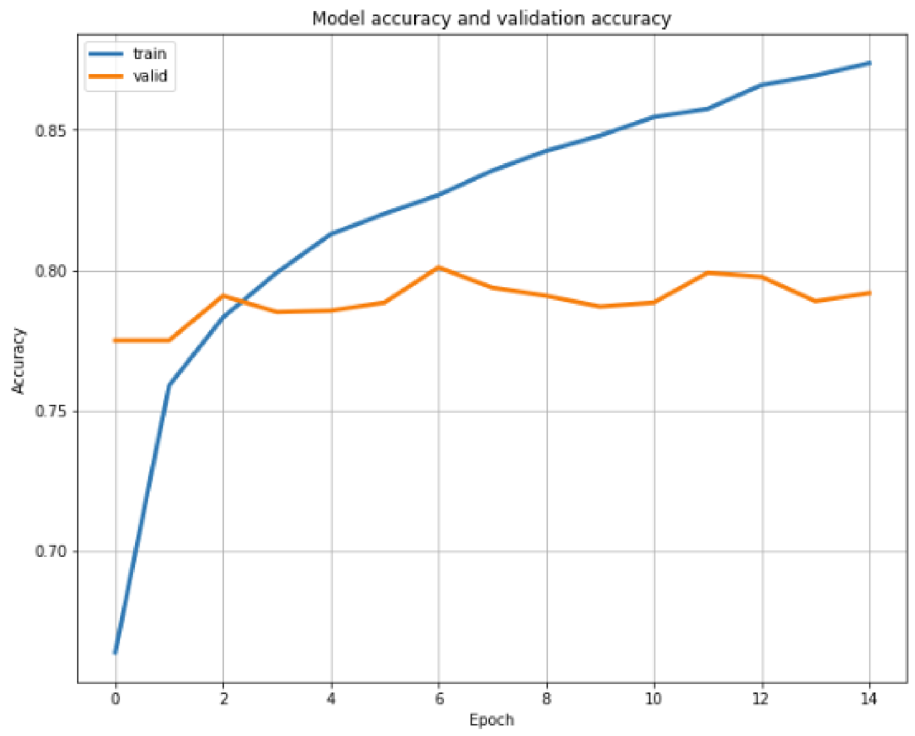
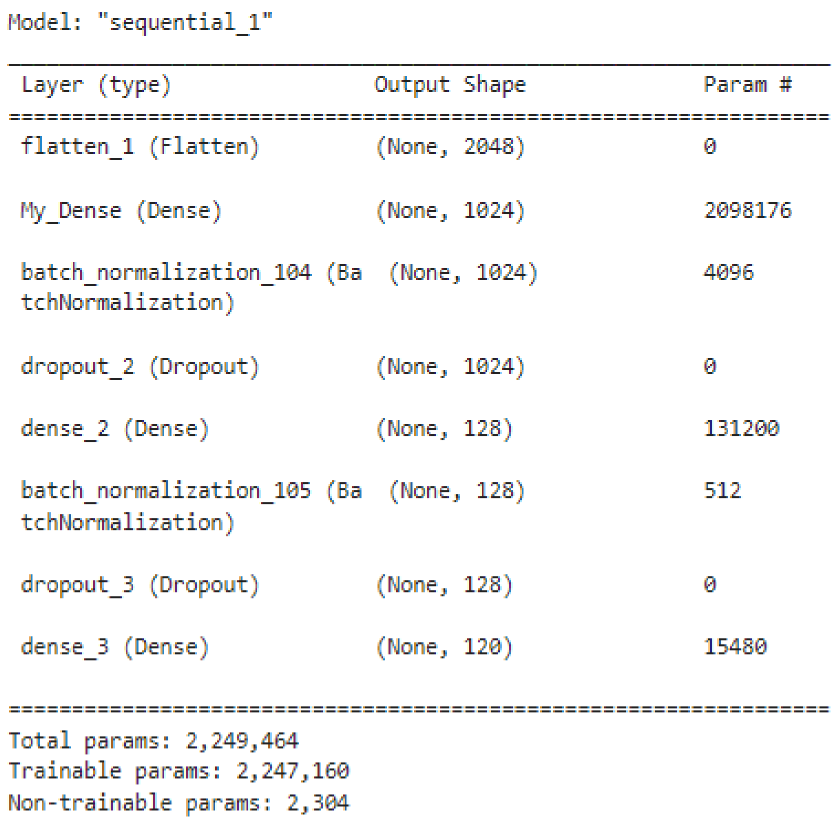


Figure 1.5: InceptionV3 Model Accuracy over Epoch



```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten_1 (Flatten)         (None, 2048)              0

 My_Dense (Dense)            (None, 1024)              2098176

 batch_normalization_104 (Ba  (None, 1024)             4096
 tchNormalization)

 dropout_2 (Dropout)         (None, 1024)              0

 dense_2 (Dense)             (None, 128)               131200

 batch_normalization_105 (Ba  (None, 128)              512
 tchNormalization)

 dropout_3 (Dropout)         (None, 128)               0

 dense_3 (Dense)             (None, 120)               15480

=================================================================
Total params: 2,249,464
Trainable params: 2,247,160
Non-trainable params: 2,304
```

The Inception V3 model has a Total parameters of 2.249.464 in addition to the 24.0 million parameters of the pretrained InceptionV3 model.

We added 2 Dense layers one of 1024 units and the second one with 128 units. Finally, we put our final Dense output layer.

We also used Batch Normalisation layers and Dropout layers of 40% in order for the model to learn better and not to overfit.

InceptionV3-XGBOOST Model

CNNXGBOOST models have shown to perform better than a typical CNN model if their hyperparameters have been tuned correctly.

To test that theory, we created an XGBoost model on top of the InceptionV3 Model. We did a feature extraction from the dense layer of InceptionV3 model named "My Dense" and fed those features into the XGBoost.

Many different parameters were tried into the parameter tuning. The tuned hyperparameters put into XGBoost were the below:

- Learning Rate : 0.10
- n_estimator: 100
- Max Depth : 7
- Reg alpha : 0
- Reg lamda : 1
- Colsample by tree : 0.5
- Objective : multi:softprob
- Tree method : gpu_hist (For GPU Utilisation)
- Everything else was set at default state

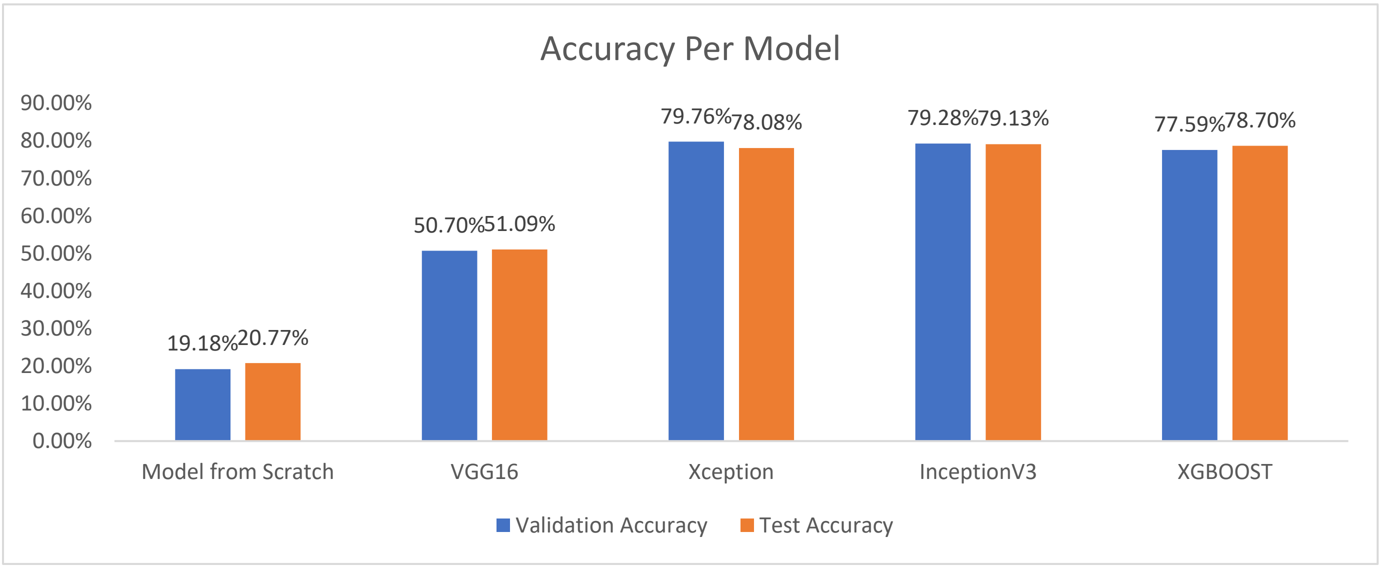
The Best accuracy achieved by the XGBOOST model was 77.59% on the validation dataset and 78.70% on the Test dataset which was below the InceptionV3 model.

Model Conclusions

Here we can see the results of all the models used in this project.

The InceptionV3 model achieved the best classification accuracy and each the one we will keep as our go to model.

Model Name	Validation Accuracy	Test Accuracy	Validation Loss
Model from Scratch	19.18%	20.77%	3.6085
VGG16	50.70%	51.09%	2.0882
Xception	79.76%	78.08%	0.7597
InceptionV3	79.28%	79.13%	0.7319
XGBOOST	77.59%	78.70%	



Lastly, with the InceptionV3 model we run a confusion matrix in order to check were are most miss classified dog breeds along with the evaluation metric scores.

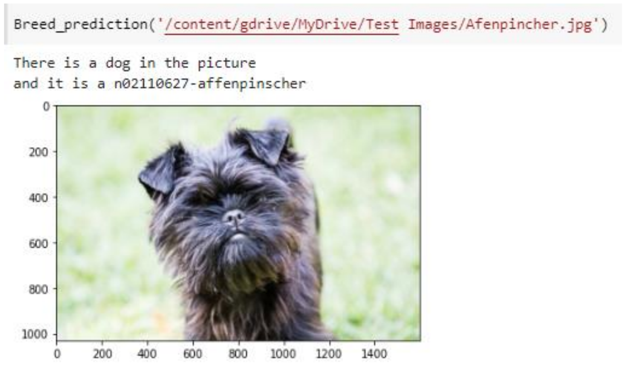
- Accuracy : 79.13%
- Precision : 79.56%
- Recall : 78.73%
- F1 Score : 78.09%

The top ten miss classified breeds were the below.

Many dog breeds are very similar. For example the Siberian Husky looks very much alike with the Eskimo dog. In that case many images of the Eskimo dog were labelled as Siberian husky or as a Malamute and vice versa.

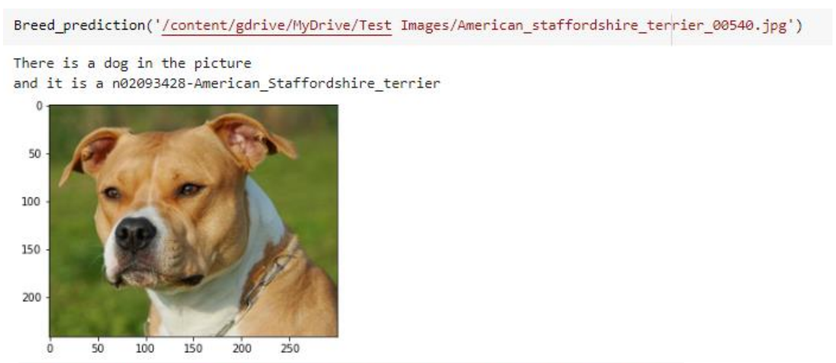
Breed Name	Miss Classification Rate
81.81%	Walker hound
76.74%	Eskimo dog
66.66%	Siberian husky
58.82%	Lhasa
57.89%	Silky terrier
57.14%	vizsla
56.25%	Miniature poodle
52.38%	Rhodesian ridgeback
50.00%	Collie
46.15%	Cardigan

Finally, we created functions that read and preprocess a new image check if the image has a dog in it and then call the InceptionV3 model to classify it.



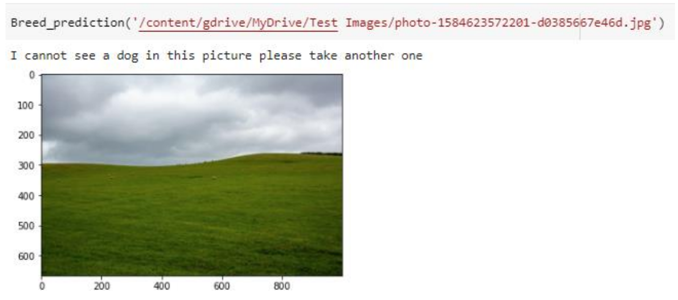
```
Breed_prediction('/content/gdrive/MyDrive/Test Images/Afenpincher.jpg')
```

```
There is a dog in the picture
and it is a n02110627-affenpinscher
```



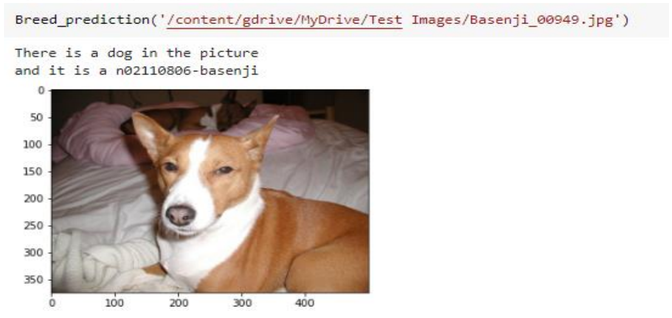
```
Breed_prediction('/content/gdrive/MyDrive/Test Images/American_staffordshire_terrier_00540.jpg')
```

```
There is a dog in the picture
and it is a n02093428-American_Staffordshire_terrier
```



```
Breed_prediction('/content/gdrive/MyDrive/Test Images/photo-1584623572201-d0385667e46d.jpg')
```

```
I cannot see a dog in this picture please take another one
```



```
Breed_prediction('/content/gdrive/MyDrive/Test Images/Basenji_00949.jpg')
```

```
There is a dog in the picture
and it is a n02110806-basenji
```

Conclusions

The model did very good considering the number of images it had been trained per class. For sure more images in training would yield better results.

The obstacles faced were memory limitations when working with large datasets, memory optimisations and python code efficiency in order for the code to run smoothly. Had to find many work arounds in order not to cause memory overflowing.

Luckily, Google Colab was used for this assignment. Training times would be ridiculously large if the code run in a local PC.

This assignment has given much knowledge of how Convolutional Neural Networks and Machine Learning work. Learnt more about libraries like Tensorflow, OpenCV, Numpy and Python code and how Python behaves in general in certain situations all of which will assist in future works.

This project after adding more breeds into the prediction and maximising the prediction accuracy to 97%-98% , we could add more animal and animal breeds into the model and then transfer the model into a mobile application where you would take a photo of an animal and get a basic description of the animal in the photo.

For what is worth, this assignment has served its purpose by teaching new and interesting things about Machine Learning!