# Capstone project
# Dog Breed Classifier

**Udacity machine learning nanodegree**

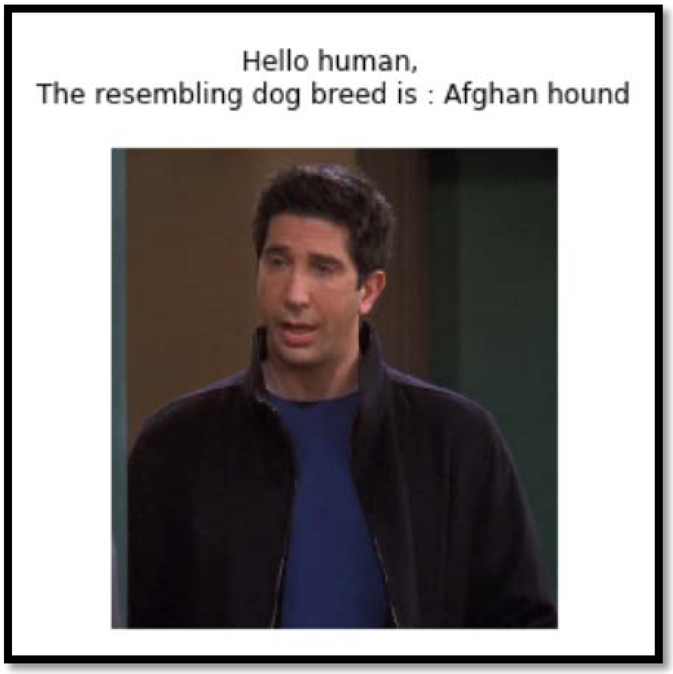
*August, 2021*

## Table of Contents

## Definition

### Project Overview

The project is about using convolutional neural networks in order to detect breeds of dogs. The main purpose of the project is to create an API that can be easily used for inference. The application main purpose is to detect humans or dogs and then return the corresponding dog breed. If the image sent to the API is a human, then an estimate of the resembling dog breed is sent back to the user. However, if the input image was neither a human nor a dog, the API returns an error message. Here is an example of the program behavior from the implemented model and API:


```
Hello human,
The resembling dog breed is : Afghan hound
```

### Problem statement

In order to implement the project, we used 2 methods of learning for CNN in order to do the previous tasks described in the overview:

1. Transfer learning for a pre-trained model
2. Implementing a CNN model from scratch and train it

In this project, we used a pre-trained model for human face detection where an OpenCV pre-trained model is used, however the accuracy wasn't great but it was sufficient for the required task. We also used a pre-trained model, VGG-16 model, and it was used in order to detect dogs in images.

The final layer for this model had 1000 classes as the pre-trained model was trained on image-net dataset, so we needed to check a specific range in these 1000 classes for the dogs. The 3rd pre-trained model was also VGG-16 model, however we used transfer learning to be able to retrain the model on the given dataset which contained the breeds. The project also tested the results of the pre-trained model with a CNN implemented from scratch.

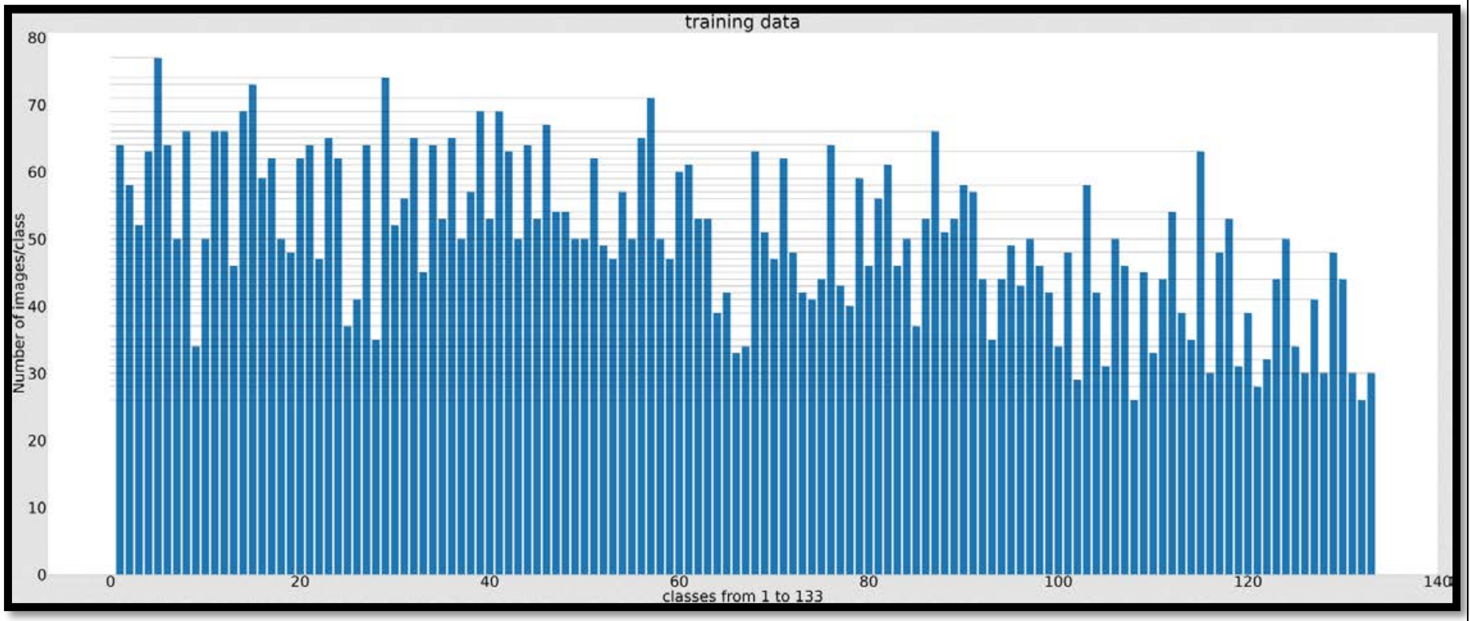
### Metrics

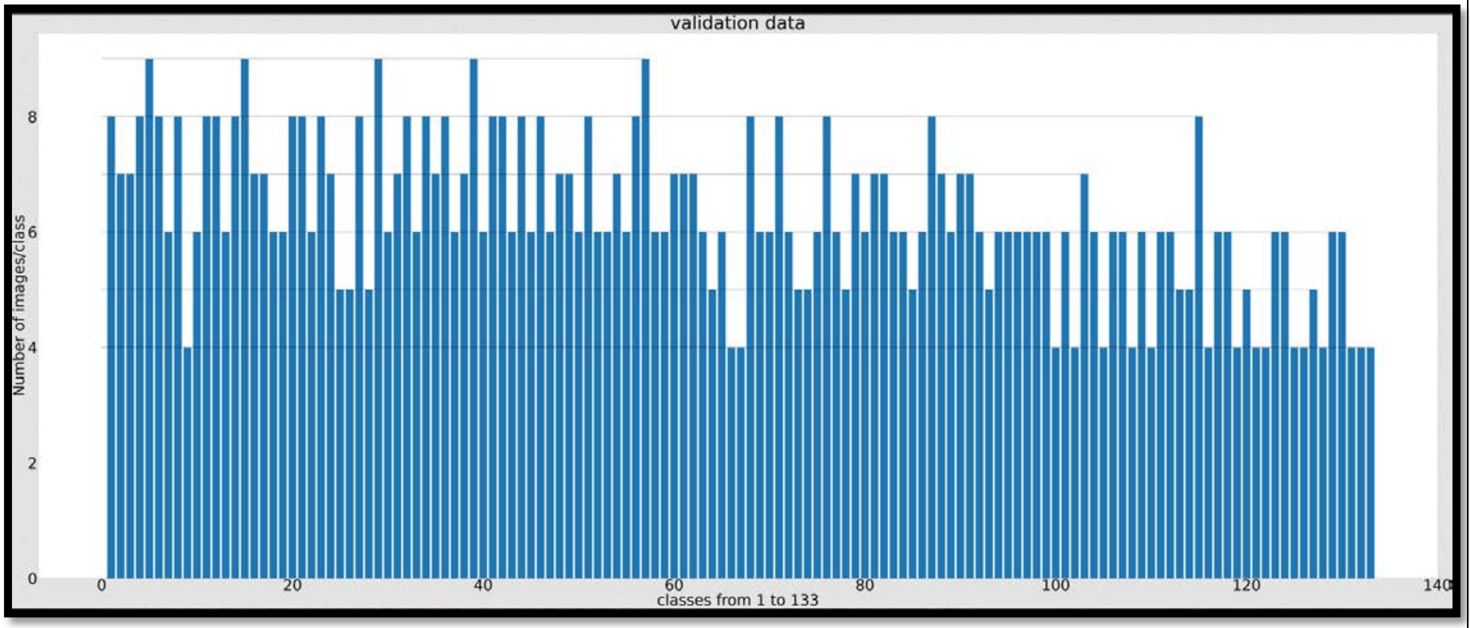
In order to measure the performance in this model, we used recall, precision, F1 score and testing, validation and training accuracy. These metrics can identify if there was a problem as data imbalance exists. Later in this report we will discuss the results and how we calculated them.

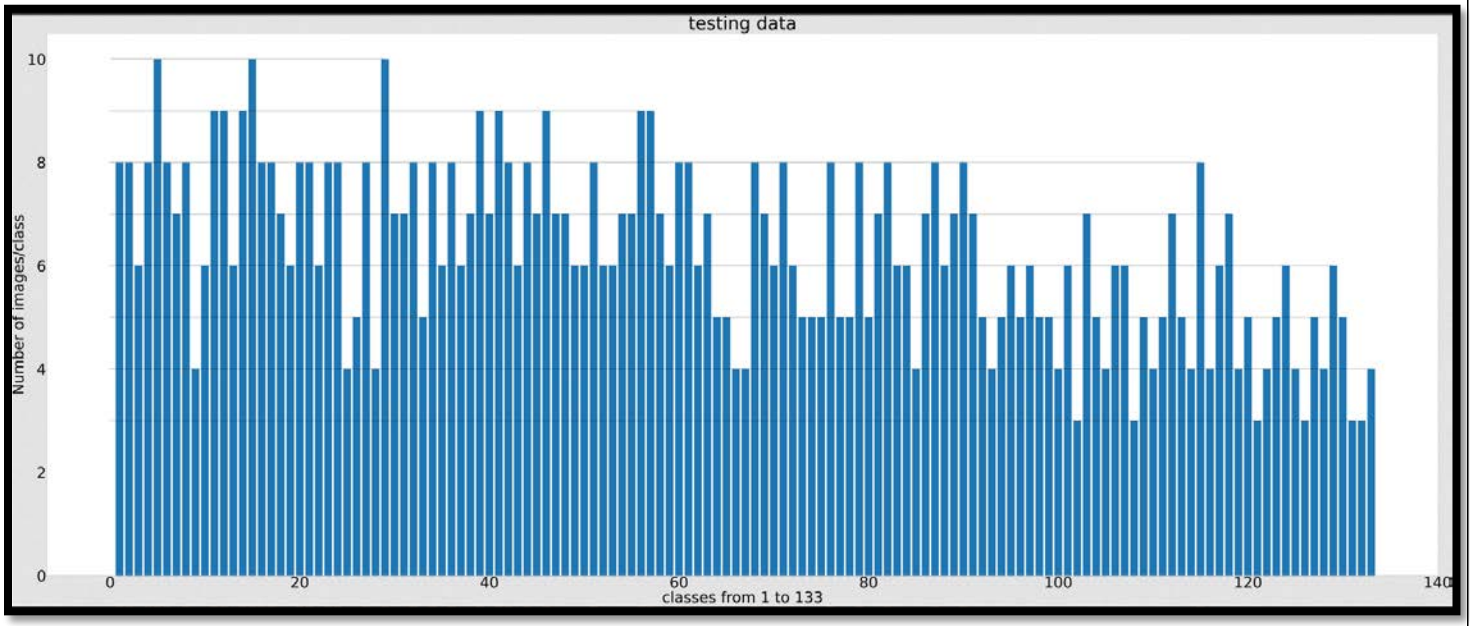
## Analysis

### Data exploration and Exploratory Visualization:

In this project we used the dataset given by Udacity in [1] and [2]. Starting with dog's dataset, it was divided into train, validation and testing folder. The next 3 figures shows the full distribution for training, validation and testing data respectively.



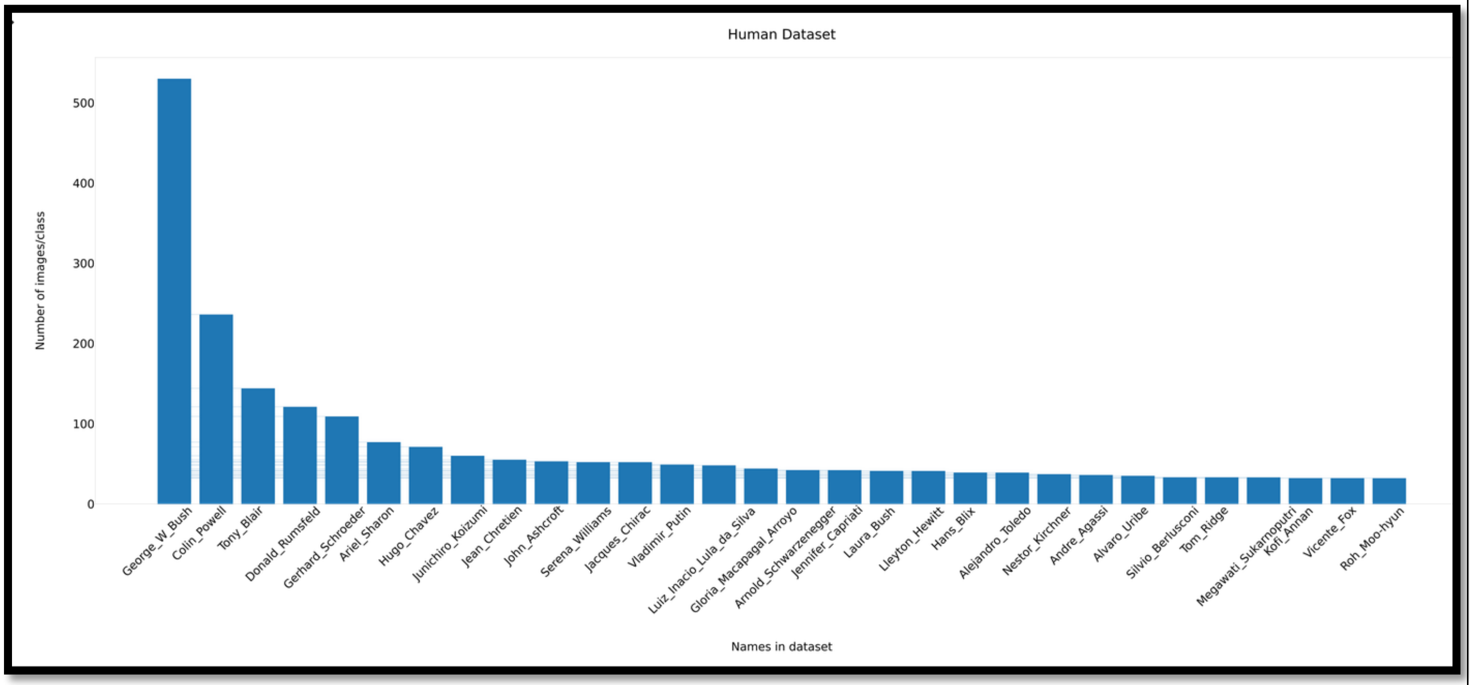




From the data distribution, there are certain classes that have fewer number of images compared to the other classes. This raises a problem called image imbalance, where data isn't distributed evenly between classes and this may cause a problem and make the model a little bit biased towards some classes and causes overfitting to those classes. This issue won't cause a problem as the dataset is small and the imbalance doesn't have high variations among classes. The dog's dataset contained 8351 images divided into 3 parts;

| Total images | Training | Validation | Testing |
|---|---|---|---|
| 8351 | 6680 | 835 | 836 |

Now for human dataset, the figure shows the names in the dataset along with how many images per name, this figure is sorted from the highest number of images to the lowest and shows the top 30 names in the dataset. Number of folders in the dataset was **5749.** Each folder represents a name, and the folder contains images for that name.



**However out of 5749 folders, 4069 folder have 1 image only**. This may cause imbalance and bias the model towards the top 5 images.

## Algorithms and Techniques

The project depends on transfer learning, pre-trained models and implementing CNN from scratch in order to get the final API to detect the breed.

Transfer learning is basically using a pre-trained model and then modify the final classification layer and add a new dense layer that has number of output classes equal to the new amount of classes needed. Then while freezing all the model parameters, except for the new dense layer, we retrain the model. In this case the model only trains only the new layer. This results in low number of epochs needed and less effort to optimize the whole model. The model also used a pre-trained model to detect human faces, the model was obtained from OpenCV and a model called VGG-16.

In order to implement a CNN from scratch, the implemented model was inspired by the classic AlexNet [3]. However, i didn't use the architecture as it is. I modified the number of layers needed and some other parameters i.e. stride, kernel size and layer type.

### Benchmark

- The benchmark for this project was to compare the results between the implemented CNN model, while it's trained from scratch, with a pre-trained model. In this case, i compared the CNN model from scratch with a VGG-16 pre-trained model.
- In case for benchmarking the CNN architecture itself, i used trial and error method in order to find parameters for the model and tested many combinations of layers in order to find a mode that's able to converge while training.
- Recall, precision and F1 score were calculated for both models

## Methodology

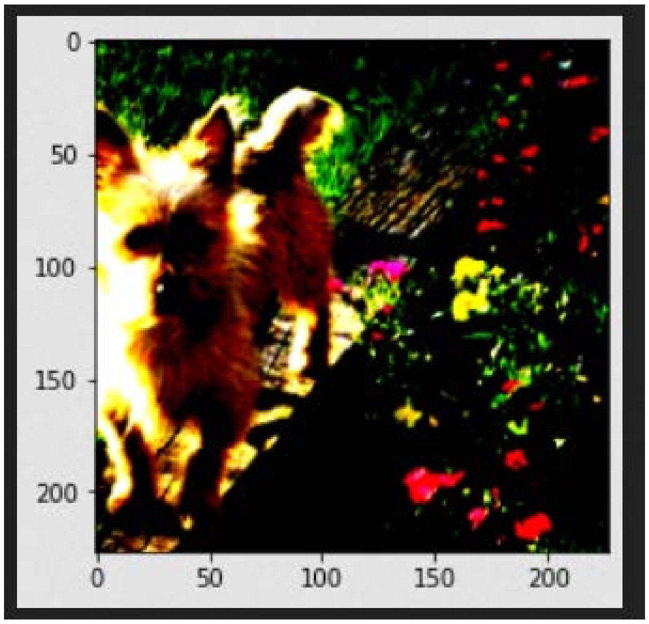
### Data preprocessing

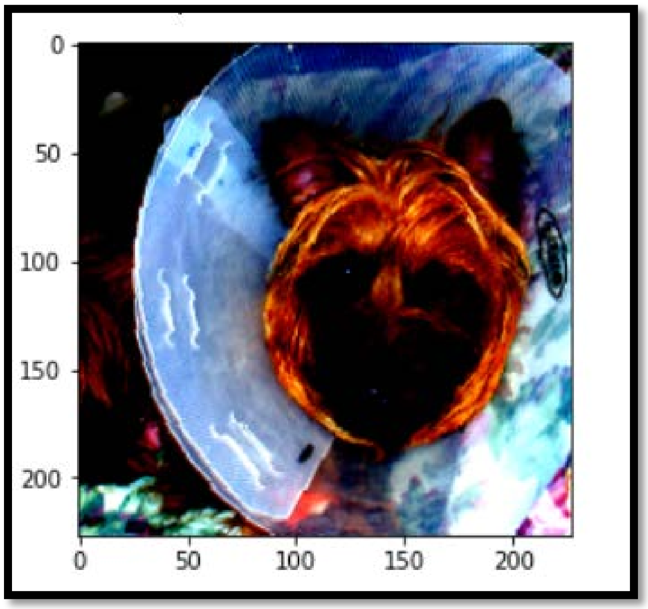
In order to improve the model results, i used data augmentation by applying transformers to the training data only as in code snippet [?]

```
transform = transforms.Compose([transforms.Resize(256),
                               transforms.RandomHorizontalFlip(),
                                transforms.RandomRotation(25),
                                transforms.CenterCrop(227),
                                transforms.ToTensor(),
                                transforms.Normalize([0.485, 0.456, 0.406],
                                                     [0.229, 0.224, 0.225]),])
```

The augmentation used in this case was to randomly flipping the image horizontally and random rotation with 25 degree specified. No more pre-processing is done for data. The augmentation however is happening when training data, so the data content itself isn't changed, just a copy of the data is returned when training the model so the dataset itself doesn’t change at all.

Here's an example for augmenting data while training:





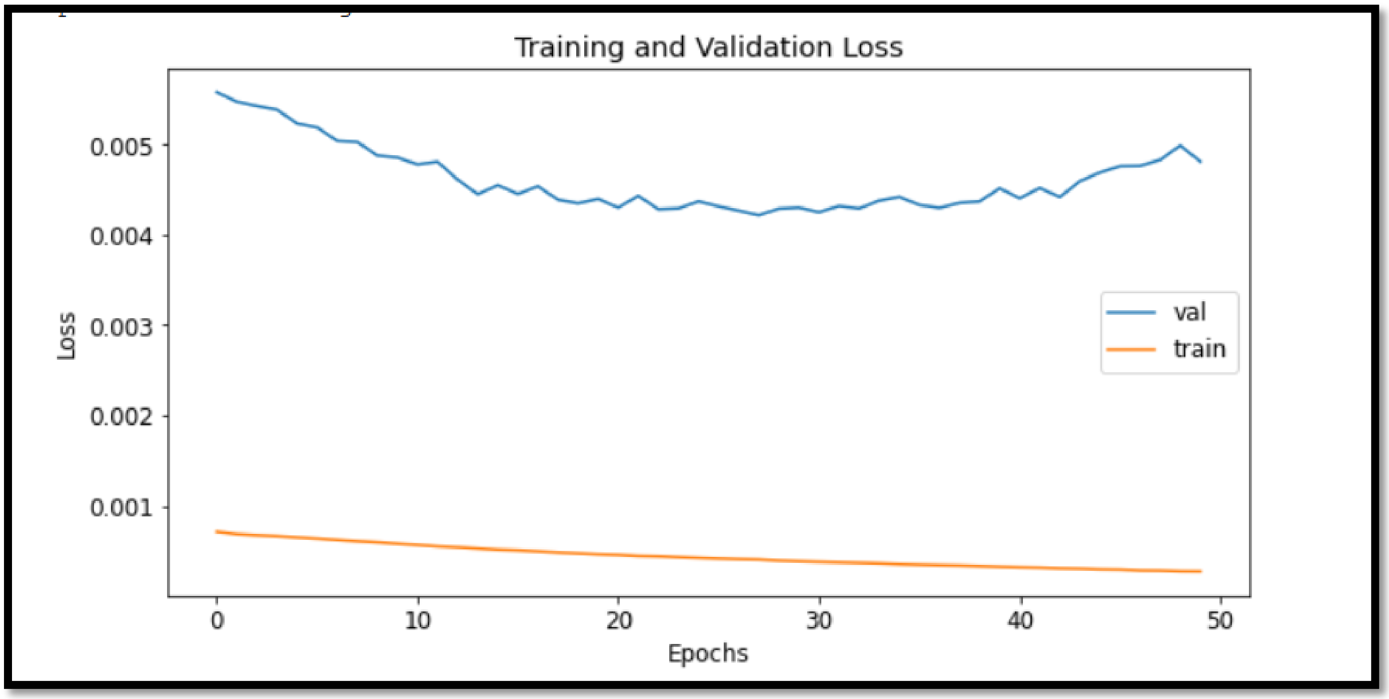
## Implementation and refinement

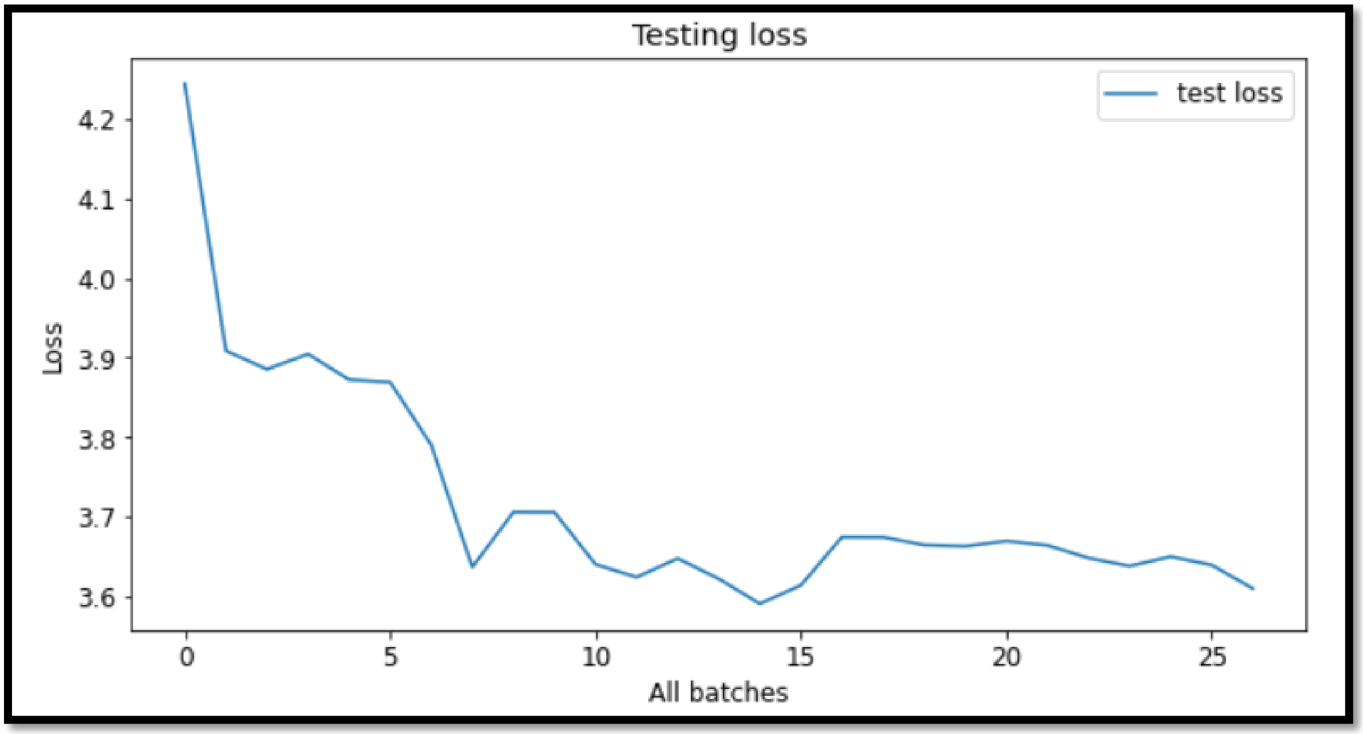
Implementing the CNN from scratch had some challenges in order to find a sequence of layers that end up detecting 133 breeds. The model was based on AlexNet architecture [3], however the network is huge and having so many parameters to learn. When training the same network as AlexNet, it didn't converge easily while training and hence i removed certain layers to make the network smaller and changed kernel sizes and strides.

In this case, the reduced network which gave see the following code snippet:

```
(conv1):    Conv2d (3, 96, kernel_size = (3, 3), stride = (2, 2))
(maxpool1): MaxPool2d (kernel_size=3, stride=2, padding=0)
(conv2):    Conv2d (96, 256, kernel_size = (3, 3), stride = (1, 1), padding = (1, 1))
(maxpool2): MaxPool2d (kernel_size=3, stride=2, padding=0)
(conv3):    Conv2d (256, 384, kernel_size = (3, 3), stride = (1, 1), padding = (1, 1))
(Avgpool):  AdaptiveAvgPool2d (output_size=1)
(FC2):      Linear (in_features=384, out_features=133, bias=True)
```

The trial and error in this part was the refinement, as i had to try the whole network of AlexNet, then check the validation and training accuracy. Then looping again till I have enough training losses and validation losses to be minimum. However, the model was trained for 50 epochs finally and did meet the requirement in the Notebook with testing accuracy 23%. Metrics like recall, precision and F1 score were used directly from sklearn library. When training the model, it saves at epoch 28 and then saturates:



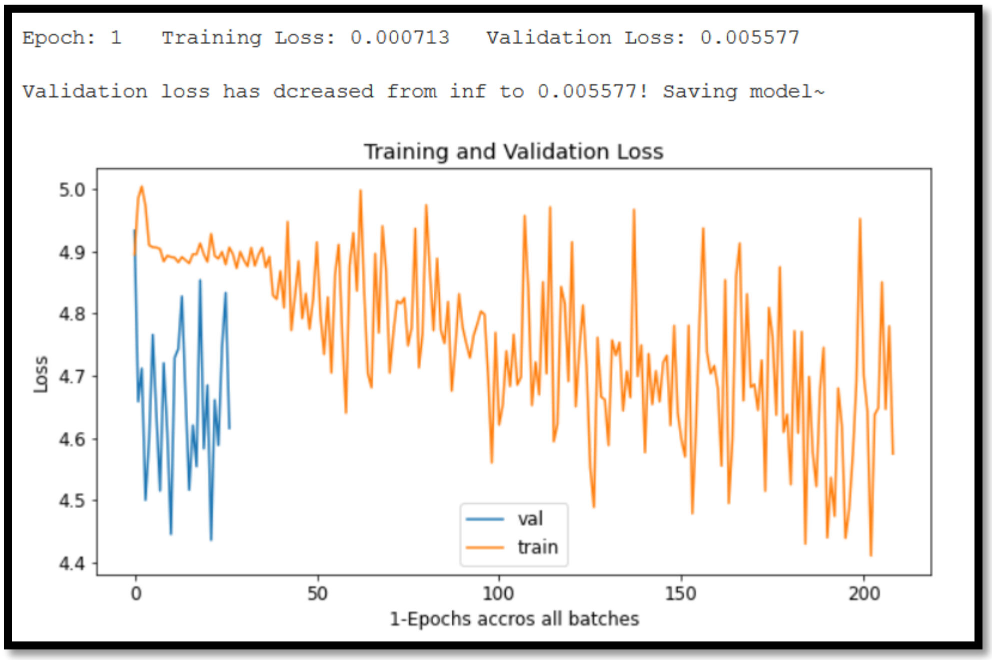


The model now has 2 models to work with, the CNN from scratch and VGG-16. As for the VGG-16 pre-trained model, i used transfer learning in order to get the most out of the pre-trained model, as it's already trained on image-net dataset and has remarkable performance. The transfer learning in this case replaces only the classifier and just retrain this layer only instead of the whole model again.

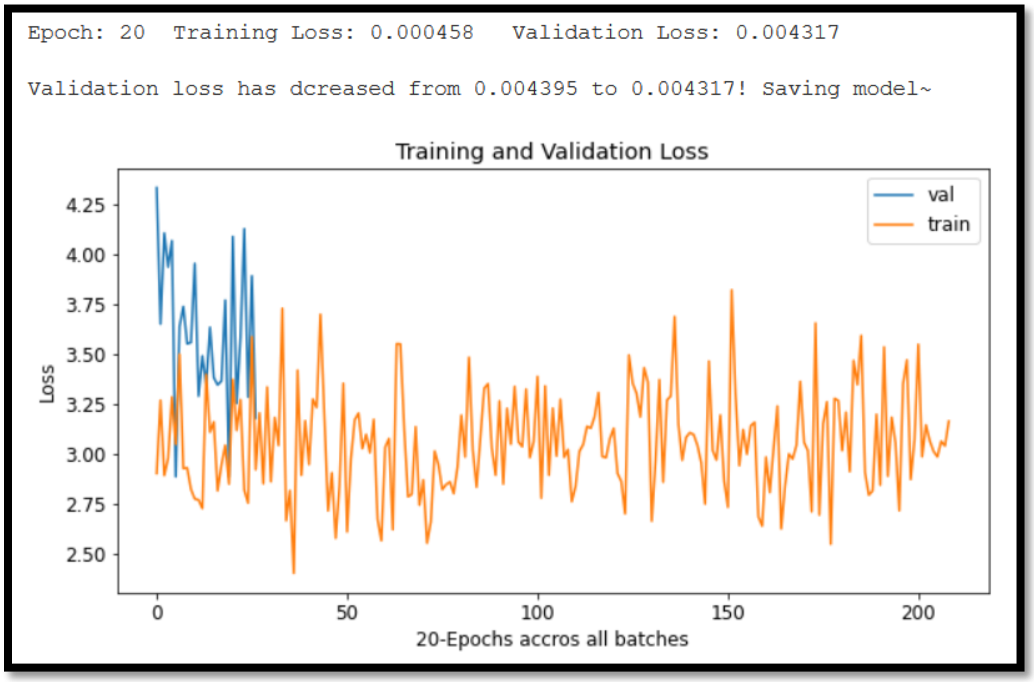
## Results

### Model Evaluation and Validation

For more details as the number of epochs increases, the following are 2 figures which represent training and validation accuracy at the beginning of the training across all batches in 1 epoch only, and after 20 epochs.

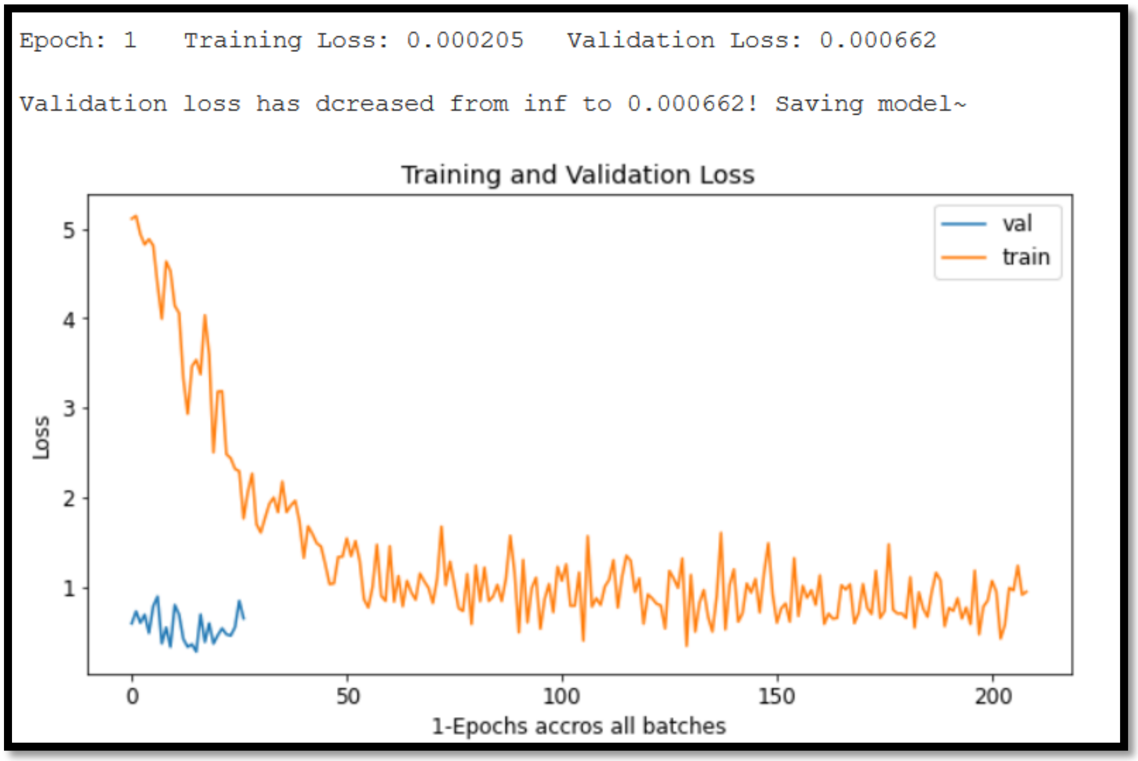


```
Epoch: 1    Training Loss: 0.000713    Validation Loss: 0.005577

Validation loss has dcreased from inf to 0.005577! Saving model~
```

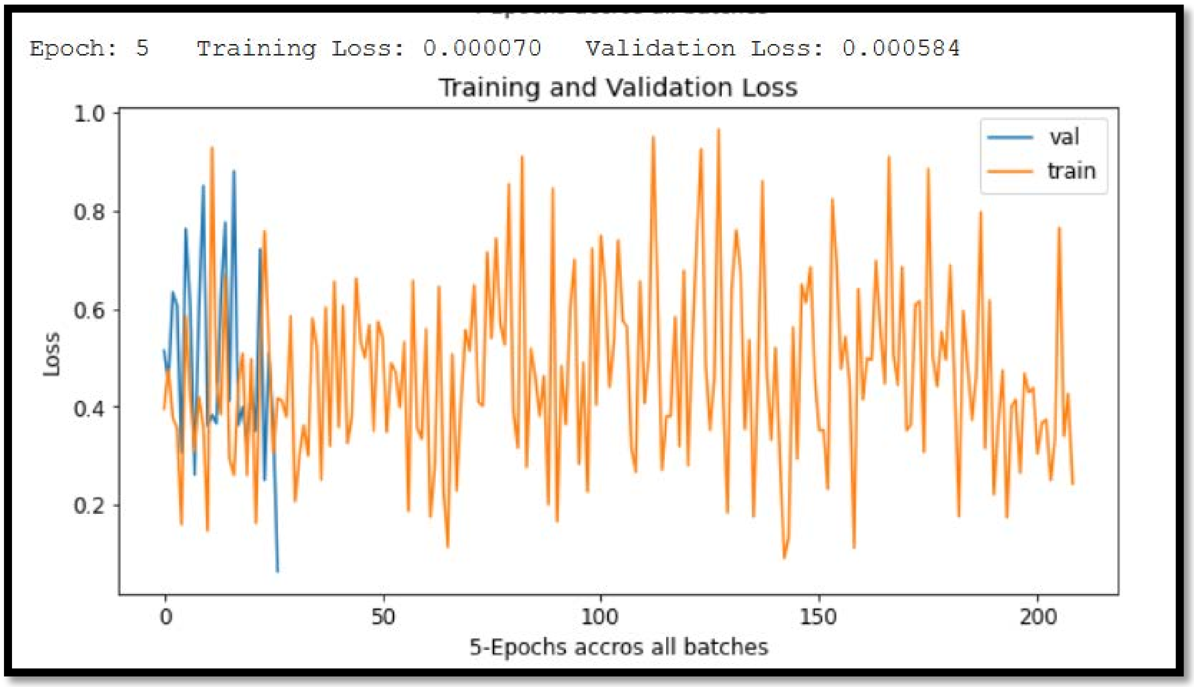


```
Epoch: 20   Training Loss: 0.000458    Validation Loss: 0.004317

Validation loss has dcreased from 0.004395 to 0.004317! Saving model~
```

When comparing with VGG-16 pre-trained model, the performance is way better in 5 epochs only. That's due to the fact that VGG-16 is trained already on **image net dataset** which is pretty much the largest dataset and i only changed the final classifier to have 133 classes instead of 1000 classes and retrained the model.



```
Epoch: 1    Training Loss: 0.000205    Validation Loss: 0.000662

Validation loss has dcreased from inf to 0.000662! Saving model~
```



```
Epoch: 5    Training Loss: 0.000070    Validation Loss: 0.000584
```

## Justification

The performance for both models is summarized in the following table

| Metric | VGG-16 pre-trained | CNN from scratch |
|---|---|---|
| **Recall** | 82.10914 % | 22 % |
| **Precision** | 83.649247 % | 20 % |
| **F1 score** | 80.7280541 % | 19 % |
| **Test accuracy** | 83% | 23% |

The rest of the application moves forward with the VGG-16 network as it outperforms the CNN implemented from scratch and will be able to give the user a pretty good experience when using the API.

## References

[1] [Online]. Available: https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/lfw.zip.

[2] [Online]. Available: https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/dogImages.zip.

[3] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," 2014.