Dog Breed Classification with Convolutional Neural Networks

Mario Rugeles Pérez

April 2019

1 Project Overview

Image classification is one of the most important applications in machine learning, more specifically, Deep Learning with Convolutional Neural Networks (CNN).

A large part of data in the world is in image format, and image classification can potentially provide solutions to very diverse problems, from entertainment purposes to applications in several fields like medicine and astronomy.

In this project we explore CNN's capacities by building a model to identify 133 dog breeds from a dataset of 8.351 images.

2 Problem Statement

Image classification is a challenging task, although CNN provides a powerful approach for solving the problem, building a good CNN can lead to a complex model that consumes significant amount of time and computational resources.

Figure 1: Despite having a common ancestor, the wolf, dogs have evolved to have very diverse characteristics. These provides a challenge for CNN's to identify dog's breeds given the big diversity of features to detect.

This resource problem can be tackled with techniques like transfer learning and data augmentation which also helps to reduce the model complexity.

We'll review these strategies that will help us to improve the performance of a CNN.

3 Metrics

For this project, we are interested on how many times the model correctly predicts the breed of a given dog's image, so we'll use accuracy for our metrics [2].

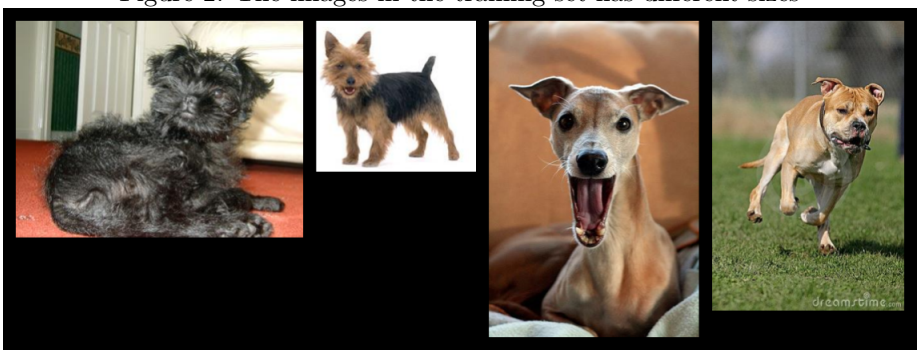
$$\texttt{accuracy}(y, \hat{y}) = \frac{1}{n_{\texttt{samples}}} \sum_{i=0}^{{n_{\texttt{samples}}}^{-1}} 1(\hat{y}_i = y_i)$$

4 Data Exploration

The dataset has 8.351 images with a training set of 6.680 images, validation set of 835 and a test set of 836 images.

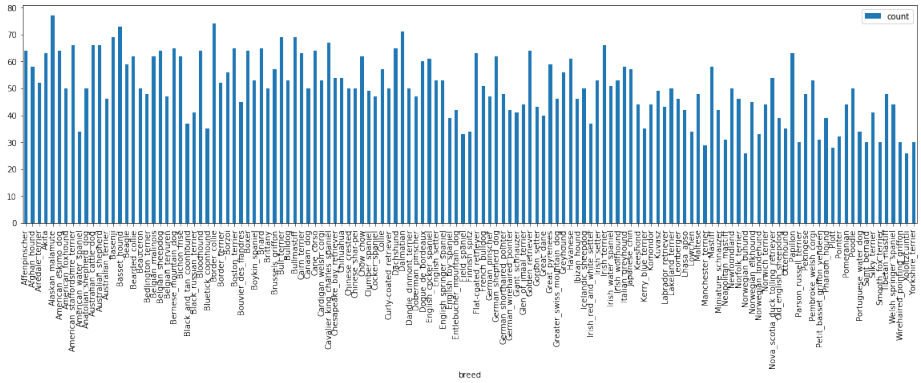
All the images have different dimensions, so all images will be resized to a common dimension of 224x224x3.

Figure 2: The images in the training set has different sizes



By looking into the target classes in the training dataset, it shows that the target classes in the training set are highly imbalanced. This can affect the accuracy of the model as many of the classes are very under represented.

Figure 3: The target classes in the training set are highly imbalanced



5 Data Preprocessing

The only issues detected with the dataset were the variation in the image's dimensions and the uneven class distribution. These issues where addressed as part of the implementation and not as a preprocessing step given the tools provided by python and Keras for data transformations.

All images were re-sized as part of the method that loads images files as tensors, and the class balancing was addressed with scikit learn's class_weight utility

6 Implementation

Three main implementations (with sub variations that will be mentioned later) were created to define the model's construction:

- CNN from scratch
- Bottleneck Implementation
- Stacking CNNs

6.1 CNN from scratch

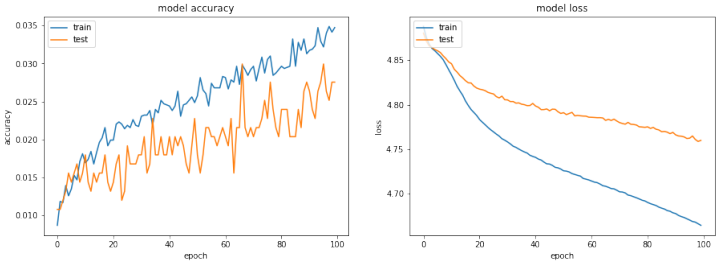
For this implementation the approach was to star with one convolutional layer and then add additional layers with the same dimensions until it got low increment with the accuracy. Next step was to add Pooling layers every two convolutional layers: it was observed that two consecutive convolutional layers had better accuracy results. As train results starting to diverge from testing (with higher accuracy for training data) dropout layers were added after every pool layer to minimize over fitting, but it was shown that only a dropout layer

after the last max pool layer was enough to push the test accuracy up to 20.57% [3].

6.1.1 Model 1

- **Accuracy**: 3.35%
- **Training time**: 13.27 minutes
- **Model**:Conv2D(16) / GlobalAveragePooling2D / Dense

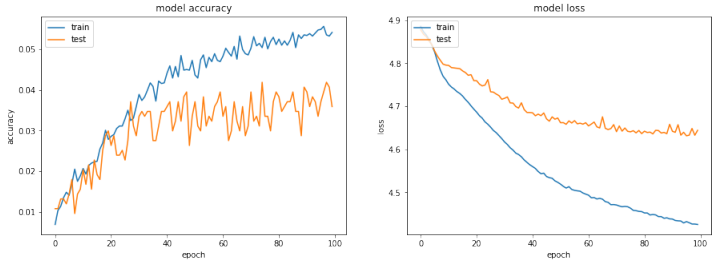
Figure 4: Accuracy: 3.35%



6.1.2 Model 2

- **Accuracy**:3.23%
- **Training time**: 27.56 minutes
- **Model**:Conv2D(16) / Conv2D(16) / GlobalAveragePooling2D / Dense

Figure 5: Accuracy:3.23%

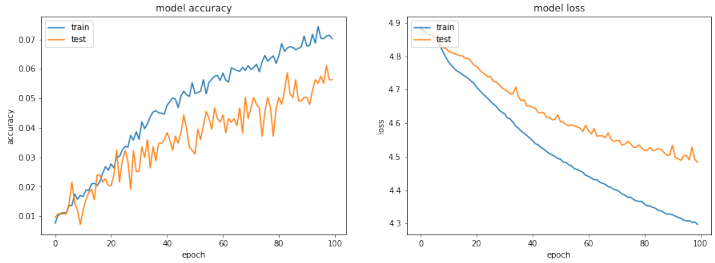


6.1.3 Model 3

- **Accuracy**: 5.14%
- **Training time**: 26.90 minutes

- **Model**: Conv2D(16) / Conv2D(16) / MaxPooling2D(2) / GlobalAverage-Pooling2D / Dense

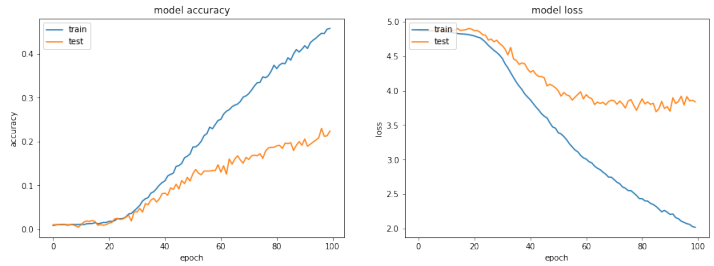
Figure 6: Accuracy: 5.14%



6.1.4 Model 4

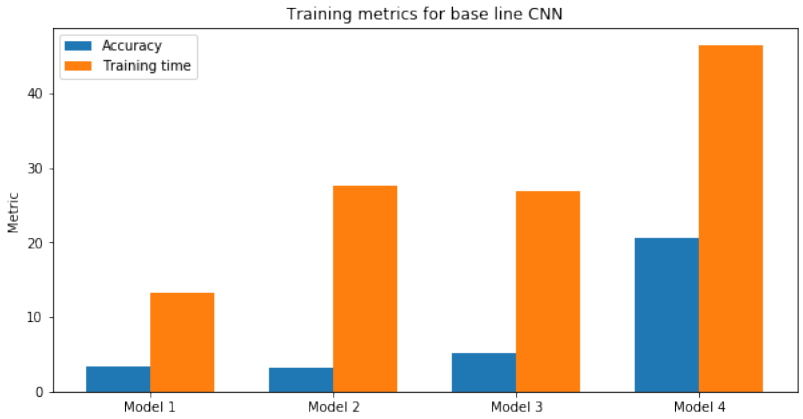
- **Accuracy**: 20.57%
- **Training time**: 46.50 minutes
- **Model**: Conv2D(16) / Conv2D(16) / MaxPooling2D(2) / Conv2D(32) / Conv2D(32) / MaxPooling2D(2) / Conv2D(64) / Conv2D(64) -¿ Max-Pooling2D(2) / GlobalAveragePooling2D / Dense

Figure 7: Accuracy: 20.57%



6.1.5 Base line metrics

Figure 8: Training metrics for base line CNN



7 Refinement

7.1 Base line CNN tuning

As it was mention in the Data Preprocessing section, the data shows imbalanced classes that might affect the model's precision. In order to addressing that issue, the strategy used was to allow the model to identify less represented classes and give them more relevance to compensate the class imbalance [4].

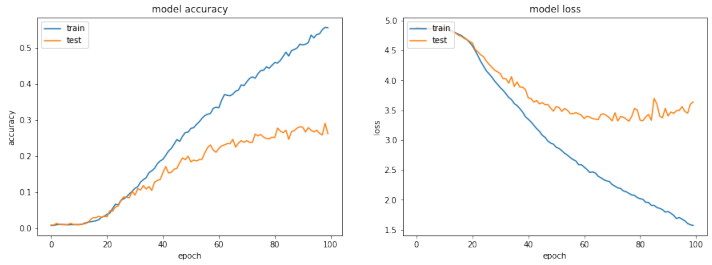
Scikit learn provides the tool **compute_class_weight** to do that. So the best model created in the **Implementation** section was trained again with the additional option **class_weight** for class balancing.

Another technique used to improve the base line model was adding data augmentation. That allows the model to detect more patterns with the same dataset by performing random modifications to the images used for training.

7.1.1 Tuning with class balancing

- **Accuracy**: 23.32%
- **Training time**: 25.46 minutes

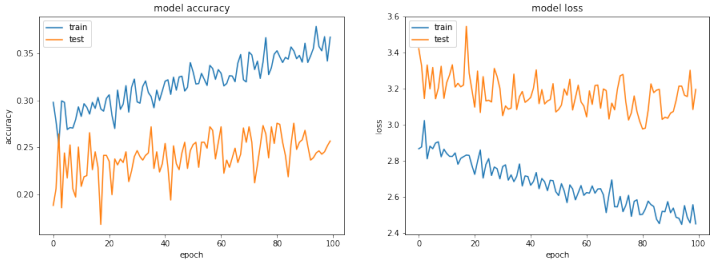
Figure 9: Accuracy: 23.32%



7.1.2 Tuning with class balancing and data augmentation

- **Accuracy**: 29.18%
- **Training time**: 66.69 minutes

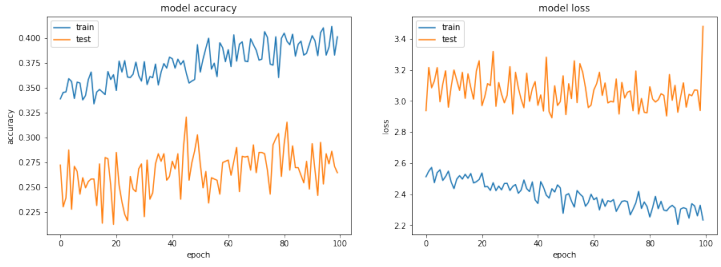
Figure 10: Accuracy: 29.18%



7.1.3 Tuning with only data augmentation (No class balancing)

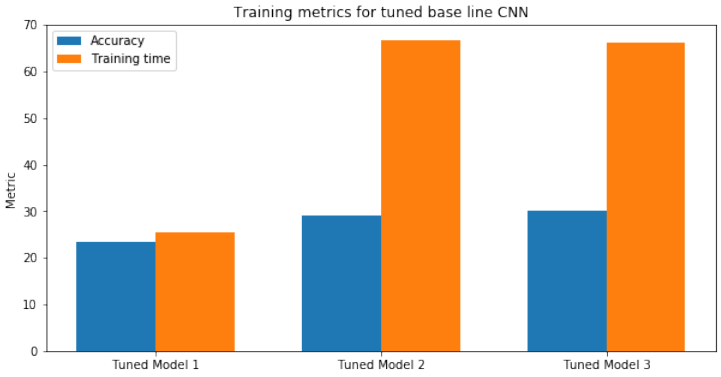
- **Accuracy**: 30.14%
- **Training time**: 66.24 minutes

Figure 11: Accuracy: 30.14%



The model with both class balancing and data augmentation (Tuned Model 2) had similar performance that the one with only data augmentation (Tuned Model 3). The difference of 1% is usually under the threshold of variability of the model, that might suggest that the model with only data augmentation can account for the class imbalance in this particular dataset with a slight improvement on both accuracy and training time.

Figure 12: Metrics for the tuned base line model

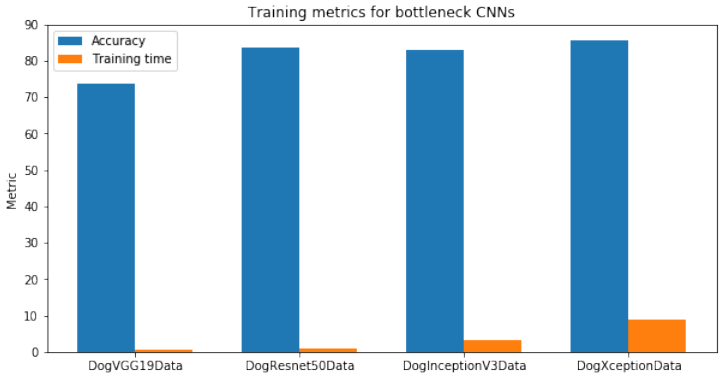


7.2 Improving accuracy with bottleneck features

Bottleneck features are features created from a dataset using a pre-trained model. This allows to create a less complex CNN with fewer layers (This translates to less computing and time resources).

For the dataset used in this project, four bottleneck features are available:DogVGG19Data.npz, DogResnet50Data.npz, DogInceptionV3Data.npz and DogXceptionData.npz [5]. Each bottleneck was tested and tuned, with the same approach used to created the model from scratch, by adding layer per layer. The four models outperform the base model by a big margin, from 73.68% to 85.64%, with training times ranging from 0.67 to 8.78 minutes, being DogXceptionData.npz the bottleneck with better performance.

Figure 13: Training metrics for bottleneck CNNs



In addition to using bottleneck features and taking advantage of the accuracy variability in the training step, the model was trained six times and the best model was preserved for use in the web application.

Figure 14: Exploiting accuracy variability for model selection

Build several models

```
bottleneck = 'DogXceptionData'

for i in range(6):
  n = i + 1
  model, history, test_features = build_bottleneck_n_model(bottleneck, n ,20)
  accuracy = model_utils.eval_model(model, 'weights.'+bottleneck+'_'+str(n)+'.hdf5', test_features, test_targets)
  print('Sub model %s %d accuracy: %.4f%%' % (bottleneck, n, accuracy))
```

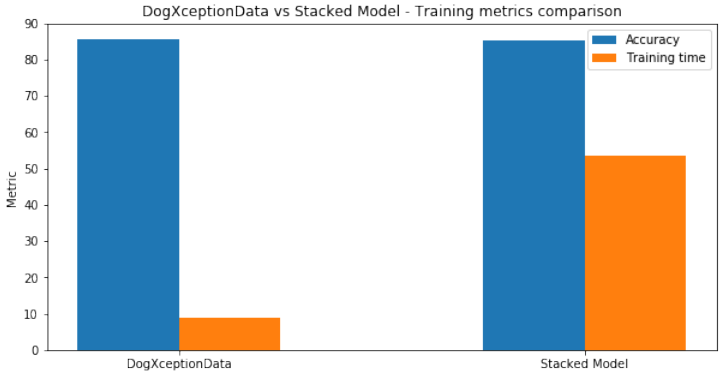
```
Sub model DogXceptionData 1 accuracy: 84.5694%
Sub model DogXceptionData 2 accuracy: 84.0909%
Sub model DogXceptionData 3 accuracy: 85.2871%
Sub model DogXceptionData 4 accuracy: 86.0048%
Sub model DogXceptionData 5 accuracy: 86.0048%
Sub model DogXceptionData 6 accuracy: 85.2871%
```

7.3 Stacking CNNs

Other approach used to improve the model accuracy was stacking 6 CNNs. This technique was borrowed from Jason Brownlee's article *How to Develop a Stacking Ensemble for Deep Learning Neural Networks in Python With Keras* [1]. The technique followed 3 steps:

- Training our best model n times (6 in this case)
- Evaluate each model and store (stack) the predictions in a new dataset.
- Use the stacked dataset to train a new model.

The new model can be any ML algorithm (RandomForestClassifier, GradientBoostingClassifier, etc.). For this project a Neural Network was used. The stacked model gave a Test accuracy of 85.16% with a Training time of 53.39 minutes. This is pretty much the same accuray's performance of the Bottleneck model, but with roughly n times the training time [6].

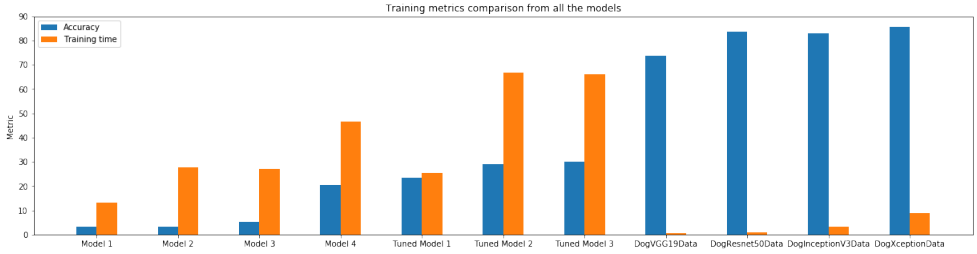


8 Model Evaluation and Validation

12 models were implemented for this project, and all show evidence to converge to similar accuracy across the techniques used to create them, so all base line models converge to the same values, same for bottleneck models, suggesting robustness in the final model.

The model with best performance was the bottleneck model from DogXceptionData. The accuracy variation by training the model 6 times range from 84.56% to 86% with a standard deviation of 0.76 which makes the model's predictions consistent with the average accuracy.

Figure 15: Training metrics comparison from all the models



9 Justification

The created model does fairly good predictions, in a small test of 5 dogs images it predicted correctly all the dogs breeds. The dog from *Figure 20* predicted the breed correctly, but it was marked as *Incorrect* because a threshold of 50% was defined to decide if a given image is from a dog or not: If the probability is under 50% it will consider is not a dog picture.

A curious observation was that for images that where not dogs, Poodle and Japanese chin breeds appeared quite frequently.

Figure 16: Correct prediction

Figure 17: Correct prediction

Figure 18: Correct prediction

Figure 19: Correct prediction

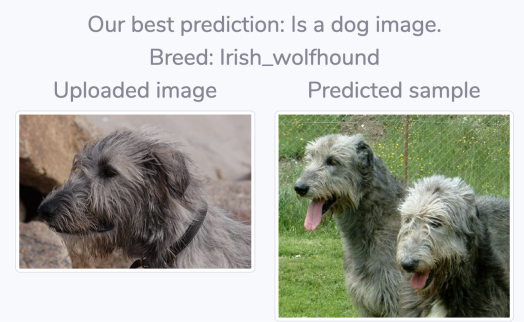


Figure 20: Incorrect prediction

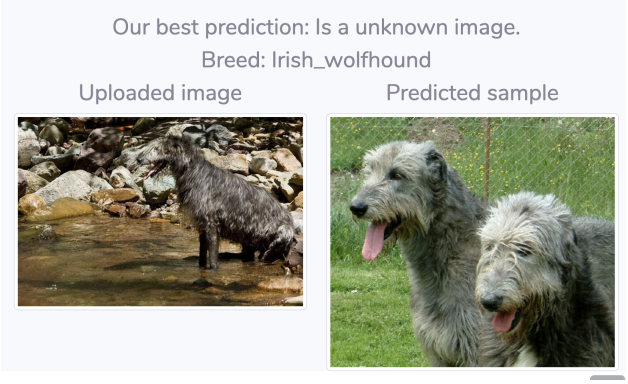


Figure 21: Correct prediction

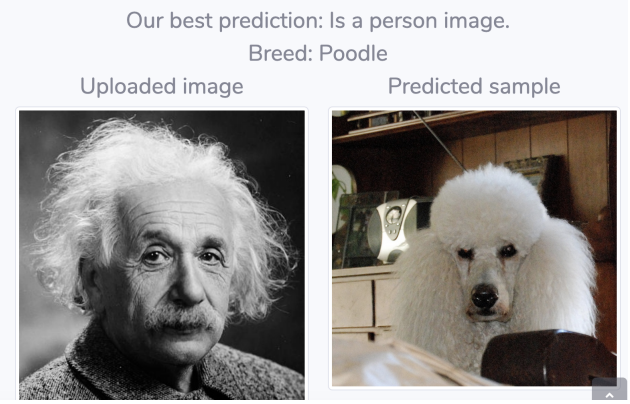


Figure 22: Incorrect prediction

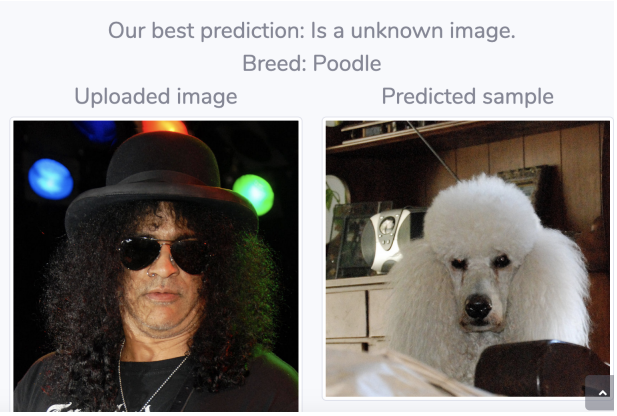


Figure 23: Correct prediction

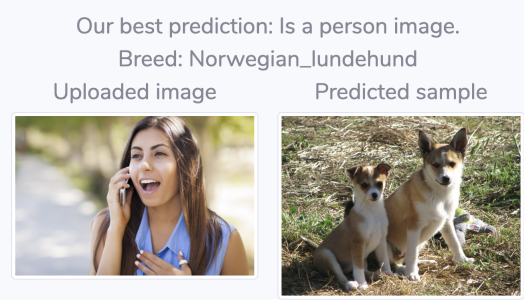


Figure 24: Correct prediction

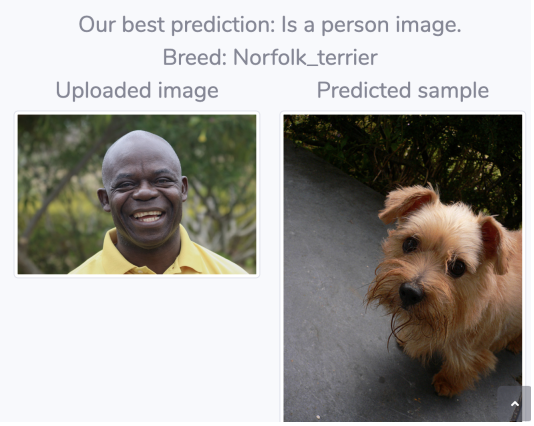


Figure 25: Correct prediction

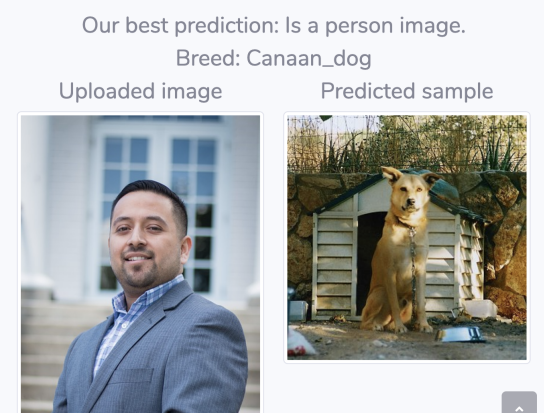


Figure 26: Correct prediction

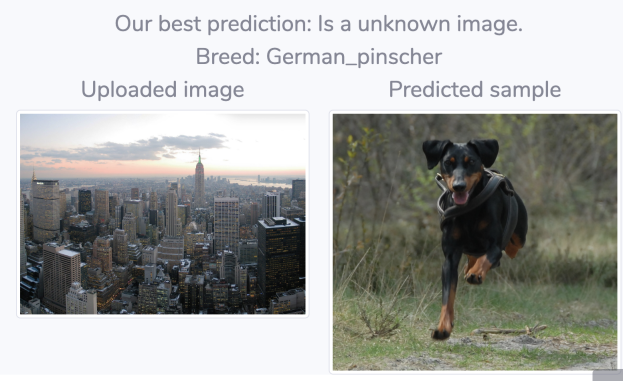


Figure 27: Correct prediction

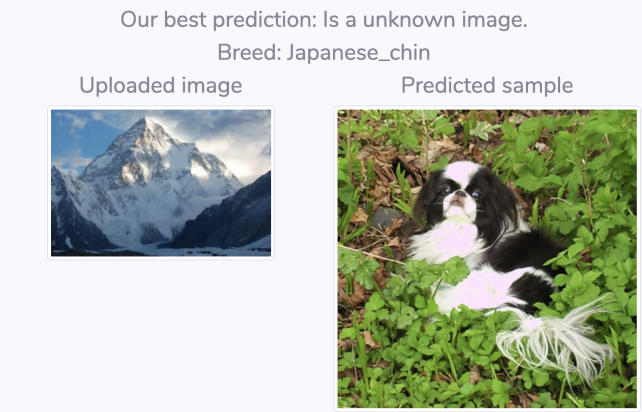


Figure 28: Incorrect prediction

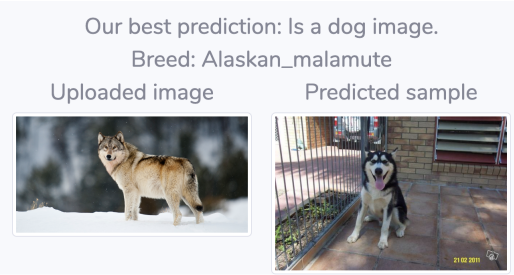


Image	**Predicted**	Breed	**Correct prediction**
Afghan hound	Dog	Afghan hound	Yes
Airedale terrier	Dog	Airedale terrier	Yes
Dandie dinmont terrier	Dog	Dandie dinmont terrier	Yes
Irish wolfhound	Dog	Irish wolfhound	Yes
Irish wolfhound	Unknown	Irish wolfhound	Yes
Albert Einstein	Person	Poodle	Yes
Slash	Unknown	Poodle	No
Woman on the phone	Person	Norwegian lundehund	Yes
Adult smiling	Person	Norkflolk terrier	Yes
Man picture	Person	Canaan dog	Yes
NYC	Unknown	German pischer	Yes
Mountain	Unknown	Japanese chin	Yes
Wolf	Dog	Alascan malamute	No

10 Reflection

For the model creation, three strategies were used: From scratch, bottleneck features and stacking, choosing the model with best accuracy. The final model was the one created from Xception bottleneck features. Six Xception models where created, and the one with best score in the validation was used in the web application.

The stacking method gave pretty much the same results as the Xception model, but it took more training time.

11 Improvement

It was particularly challenging to achieve more that 85% of accuracy, despite testing several architectures and different set of optimizers and other parameters. Although the model with no class balancing did slightly better than the model with class balancing, one approach that can be use to increment the score would be to have both a more balanced and bigger dataset, taking into account this might require more computational resources.

For this project, it was used bottlenecks already created by a third party, so creating bottlenecks from other pre-trained networks (Like GoogleLeNet) could potentially give better statistics for choosing other architectures.

A more recently innovation in Deep Learning are the Generative Adversarial Networks (GANs) that consists on two neural networks, one trying to beat the other: The first one trying to cheat results and the second one trying to get better at identifying these cheats. That would be an interesting case to apply for this project.

References

[1] Jason Brownlee. *How to Develop a Stacking Ensemble for Deep Learning Neural Networks in Python With Keras*. URL: https://machinelearningmastery.com/stacking-ensemble-for-deep-learning-neural-networks/.

[2] Scikit Learn. *Accuracy score*. URL: https://scikit-learn.org/stable/modules/model_evaluation.html#accuracy-score.

[3] Mario Rugeles. *Notebook 1) Base line CNN implementation*. URL: https://github.com/mrugeles/dog_breeds/blob/master/1)_Base_line_CNN_implementation.ipynb.

[4] Mario Rugeles. *Notebook 2) Base line CNN tuning*. URL: https://github.com/mrugeles/dog_breeds/blob/master/2)_Base_line_CNN_tuning.ipynb.

[5] Mario Rugeles. *Notebook 3) Bottleneck Models*. URL: https://github.com/mrugeles/dog_breeds/blob/master/3)_Bottleneck_Models.ipynb.

[6] Mario Rugeles. *Notebook 4) Stack Models*. URL: https://github.com/mrugeles/dog_breeds/blob/master/4)_Stack_Models.ipynb.