Dog Breeds Classifier Web Application
Udacity DSND Capstone Project

Renee Shiyan Liu

2019-01-23

Abstract

This project utilizes Convolutional Neural Network — CNN — technique to train a deep learning model through an existing set of 8351 dog images, where the dogs are pre-classified into 133 unique breeds. Specifically, in this project, I used transfer learning technique where Keras' InceptionV3 model is employed as the base with a custom-defined classifier layer to build a new dog image classifier. This algorithm first uses human and dog face detectors to differentiate if the incoming image is a dog or a human face, then classifies it into a certain breed. The final product of this project is a web application deployed using FLASK to provide users an interesting interactive experience where a user can get a breed prediction with 81.46% accuracy after uploading a dog or a human image (the prediction tells you what dog breed this human face resembles the most).

Keywords: Deep Learning, Image Classification, CNN, Transfer Learning

1. Project Definition

1.1. What is the core task of this project?

The core task of this project includes three main steps as follows,

- First, use an existing CNN model as the feature extraction layer via transfer learning technique before adding a custom classifier layer at the end
- Second, use an existing set of dog images to train this newly obtained model with a reasonable accuracy on the test images
- The last step is to deploy this trained model as a web application using Flask micro framework. The end goal of this project is to allow a user

to upload her favorite image, be it a dog or a non-dog, and to predict for her what the dog breed is. If it is a human photo, the application tells her what dog breed this human mostly resemble with. (See Figure 1 below)

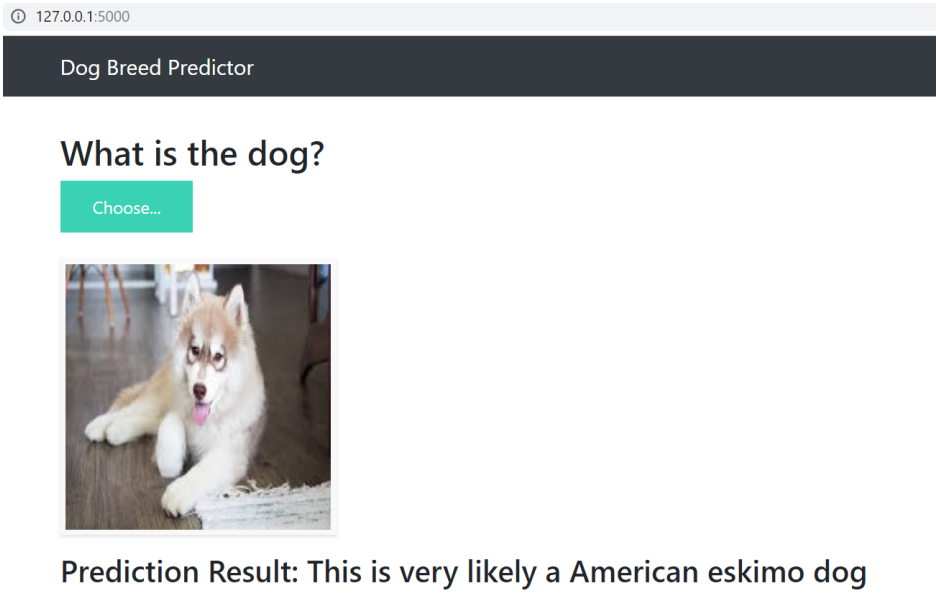
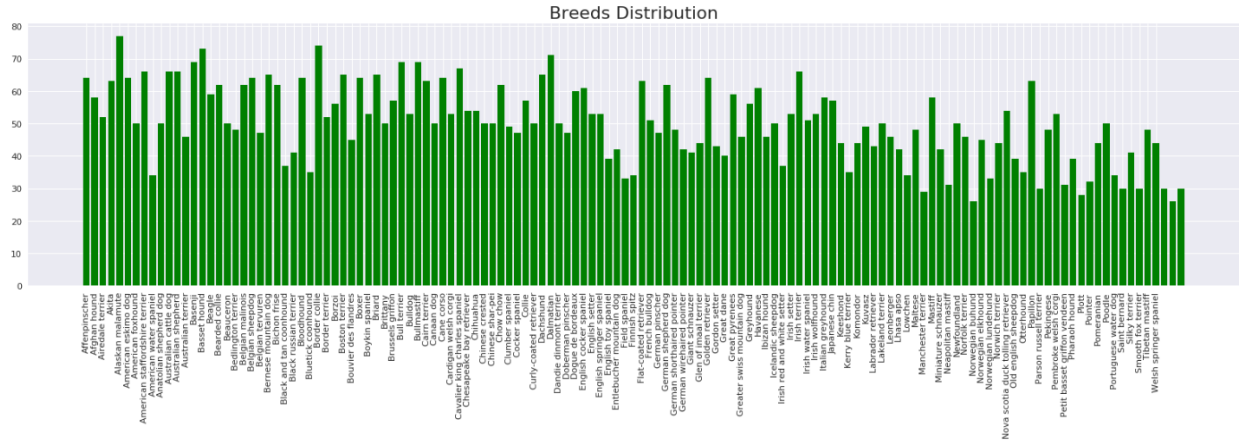


Figure 1: Project Web Application UI

1.2. Evaluation Metrics

After training the model using the training data set (6680 out of 8351 total images) with validation data set (835 out of 8351 total images), accuracy score is calculated on the test data set (another 836 images). The accuracy score is the metrics of this algorithm. Overall, the algorithm in this project achieved a 81.46% accuracy.

I chose to use accuracy as the model metrics for two reasons. The first reason is that it is very intuitive and straightforward especially in image classification space. The score tells you directly how the model performs. The second reason is that the training data has relatively balanced classes (See the bar chart below). In other words, all 133 dog breeds are reasonably



well represented by the training images. Under relatively balanced classes, using accuracy is completely justifiable.

It is definitely not a perfect metrics because the classes are not entirely uniformly distributed but they are neither severely imbalanced.

2. Analysis

2.1. Data Exploration and Visualization

Figure 2 displays some sample images from the training dataset. However, computer sees each images as (height, width, number-of-color channels), the following shows the summary statistics of the training images

- Height: (Mean: 538, Max: 3888, Min: 120, Median: 475)
- Width: (Mean: 566, Max: 3888, Min: 129, Median: 500)
- color channels: 3. All images are colorful.

For example, a representative image size is (538, 566, 3) with 3 being the color channels.

Since raw images in the training set are of different sizes, pre-processing the images into uniform sizes and standardized shapes are necessary.

Figure 2: Sample Images from the Training set

3. Methodology

3.1. Data Preprocessing

As suggested in the previous section, I perfomred three preprocessing procedures before using any Keras pre-trained model. The three steps are as follows,

1. reordering the color channels from Red, Green, Blue (RGB) to Blue, Green, Red (BGR).
2. resizing any incoming image to 224 by 224 pixels.
3. Normalization. This step is required if we want to employ pre-trained Keras models such as ResNet-50 and InceptionV3. In this extra step, each pixel from an image must be subtracted by the mean pixels (expressed in RGB order) which are calculated from all the image.
4. What I could have done but didn't do is to augment the training images to make the models more flexible. This means that I can teach

the model to recognize the real features of the images instead of just memorizing what they look like.

3.2. Implementation

In order to build an Image classification model, we need to first refine the features of each incoming image. The is to say that we have to "pick" the features that mostly capture the "soul" of an image.

This is especially important and necessary if the predictors are image pixels because they are very large in size after being reshaped into pixels. For example, a 224 by 224 image will have 224×224 features. This idea is very similar to dimension reduction techniques such as PCA. The end product is called bottleneck features.

The best method to achieve this goal is to use Convolutional Neural Network (CNN) to do feature extraction. Why? It is because images contains lots of pixels that make dense layers very slow. CNN comes in handy because it doesn't require the model to start with looking at each single pixel at the beginning. Instead, it uses patterns to "look" at the whole image and look for features first. This largely reduces the number of inputs for the model to train efficiently.

Noted, in practice (as it is in this project), we don't train the CNN model from scratch because it takes too long and we won't have enough training images to obtain a satisfying model. Thus, I use transfer learning technique to directly obtain the infrastructure of the feature extraction layer for my model.

Specifically, I used InceptionV3 model from Keras as my feature extractor. This is done through directly downloading the corresponding bottleneck features for training, validating, and test data set prepared by Udacity. The custom-classification layer – the last layer – is build up using a GlobalAveragePooling2D layer, two dropout layers and two Dense layers.

GlobalAveragePooling2d layer is to retain the most important features among the bottleneck features that are fed in. This step is crucial because it largely reduces the complexity this model is dealing with. Then, dropout layers prevent overfitting because I want to achieve a good accuracy in the testing stage. Dense layers are to make sure that the model takes into account all the information provided by the refined bottleneck features.

This architecture is suitable to this Image Classification task because we need accuracy and we have already largely reduced the dimensionality of the input features. (see Figure 3).

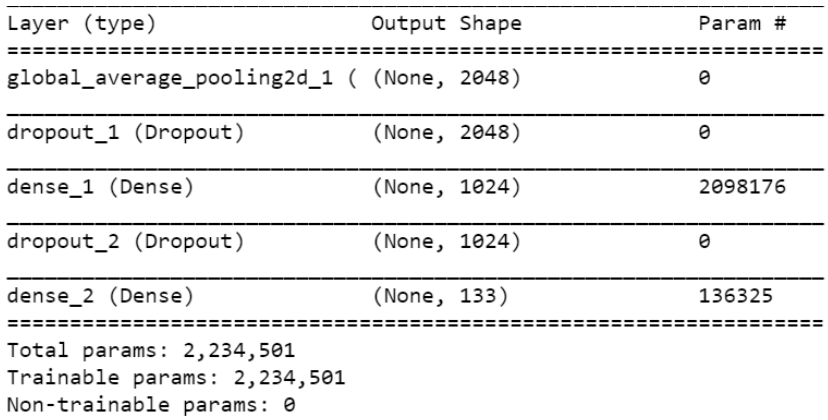


Figure 3: Model Architecture - The Classifier Layer

I then compiled the model using accuracy as the model performance metric and a RMSprop optimizer. I used 20 epochs to train the data, the validaton accuracy is around 79% on the first try. Finally, I saved the weights of the trained model for re-use in the future.

3.3. Solution Refinement

Fine tuning the trained model is crucial if we want improvement in trained models.

How did I improve the training process? I chose learning rate, batch size, and eopochs as the three hyperparameters for tuning.

The table below records the results of trying out different sets of hyperparameters. As a result of that, I chose learning rate, batch size, and epochs to be 0.002, 32, and 55 respectively, to be the hyperparameters of my final trained/saved model.

learning Rate	batch size	epoch	Test Accuracy(%)
0.05	32	15	5.38
0.001	20	20	78.65
0.001	32	35	76.67
0.001	32	55	81.22
0.002	32	25	75.7
0.002	32	35	79.43
0.002	32	55	81.46
0.002	30	55	77.75
0.002	40	55	80.02
0.002	32	65	78.46

4. Results

The trained model achieves 81.46% of accuracy when used on the test data set. Overall, a very satisfactory result. However, if more time and computational powers are allowed, I can use Grid Search to find the best possible set of hyperparameters. This includes trying out different choices of optimizers.

Moreover, I can augment the images before training the model. This can increase the capability of the model to recognize the essential features of different images than just memorizing those features from the training images.

Lastly, I can also try to use other transferred CNN model feature extractors to compare which one provides higher accuracy.

5. Conclusion

To recapture the pipeline for this project,

1. Preprocess an incoming image into a 4D array that is suitable to be fed into Keras Model.
2. Use InceptonV3 as the transfer-learning CNN layers to extract bottleneck features.
3. Build a custom classifier features using Dense layers.
4. Train and validate the model.
5. Fine Tune a set of hyperparameters.

6. Save the trained model weights for future use.
7. Reload the model with the saved weights to check its accuracy on the test data set.

What I found particularly difficult about this project is to make the correct input dimension for the image to be ready to use. Another difficult part is to make sense of the large amount of parameter values. I need to find more resources to understand how to use them than just storing them.

As this is an end-to-end project, the most challenging part is actually figuring out how to deploy this algorithm successfully to a web application. I have trouble connecting the back-end model and front-end page. I finally figured them all out by adapting some boilerplates code I found online.

How do I want to improve this project in the future? A few things I plan for the further work:

- trying out different pre-trained models.
- Visualize the relationship between Epochs and Training/Testing Errors.
- Tinker with different numbers of the hidden-layer nodes.
- I want to figure out a way to rescale the incoming image so that each image contains only the object of interest. For example, if a supplied image is a dog running in the distance, I want to be able to transform the image such that the dog image can be blown up or the irrelevant parts of the image can be removed first.

6. References

1. Classification Metrics
2. How to choose metrics