Dog Breed Identification App Powered by Convolutional Neural Networks (CNNs)

Table of Contents

[Definition 2](#_Toc116565029)

[Project Overview 2](#_Toc116565030)

[Problem Statement 2](#_Toc116565031)

[Evaluation 2](#_Toc116565032)

[Analysis 3](#_Toc116565033)

[Data Exploration & Visualization 3](#_Toc116565034)

[Algorithms 3](#_Toc116565035)

[Benchmark 4](#_Toc116565036)

[Methodology 4](#_Toc116565037)

[Data Processing 4](#_Toc116565038)

[Implementation 5](#_Toc116565039)

[Refinement 13](#_Toc116565040)

[Results & Conclusion 17](#_Toc116565041)

[Reflection 19](#_Toc116565042)

[Improvement and Way Forwards 19](#_Toc116565043)

# **Definition**

## Project Overview

For people, it is a not difficult task or rather trivial to see and understand the images in front of them. Computer Vision (CV) is a multi-disciplinary field that focuses on enabling computers to do so from digital images, videos and other visual input. A human can easily tell if it is either a dog or a human face, and even recognize a face/breed that they have only seen once before. How can we render the same or even more capabilities to computers?

While it remains unsolved, CV has made significant progress over these decades. Among them, deep learning methods such as Convolutional Neural Networks (CNN) are achieving promising results on challenging problems, which include image classification, object recognition, face detection, and so forth.

Taking advantage of such progress, this project aims at constructing a deep learning algorithm to classify images of dogs and human faces according to the dog's breed. The algorithm development is intended to be used as part of a web or mobile app, which accepts any user-supplied image as input and provides an estimate of the dog's breed name for either dog or human-face image.

## Problem Statement

The objective of this project is to develop an algorithm that can take an image as input and return the prediction of the dog’s breed. If a human image is detected, it will provide an estimate of the dog breed (this will be fun to know your dog cousin!). Therefore, the tasks to address this problem are as follows:

1. Develop Human-face detector
2. Develop Dog detector
3. Create a CNN to classify dog breeds
4. Develop and test an algorithm for the whole process
5. Deploy the App for dog breed identification

For the development of CNN to classify dog breeds, transfer learning will be utilized based on the pre-computed bottleneck features.

## Evaluation

The algorithm’s performance is evaluated by classification accuracy. As for the training of the model, categorical cross-entropy loss function and RMSprop optimizer are used respectively. The RMSProp algorithm accumulates only the gradients from the most recent iterations (as opposed to all the gradients since the beginning of training) to mitigate the risk of slowing down a bit too fast and never converging to the global optimum[[1]](#footnote-1).

# **Analysis**

## Data Exploration & Visualization

The dataset of dog images as well as vided by Udacity and can be downloaded from [here](https://github.com/udacity/dog-project). The dataset consists of a total of 8,351 total dog images, out of which 6,680 are for training, 835 for validation and 836 for the est are allocated, respectively.

The dog images have 133 categories. The distribution of dog categories (id\_breed) is visualized as follows (**Figure 1**).

Chart, histogram

Description automatically generated

Figure 2. Distribution and Frequency of Dog Breed Categories.

The size of the sample for each dog breed is very small (ranging from 26 to 77), indicating the necessity of transfer learning to have higher accuracy and avoid overfitting. By the same token, image augmentation should be used as well.

## Algorithms

For dog breed identification, the project will use a Convolutional Neural Network (CNN), a popular deep learning algorithm for artificial intelligence problems in computer vision. In general, CNN requires very large training datasets for the models to learn features and hierarchies of features that are general across images. Remarkably, once the CNN is trained with a dataset that is large and general enough, the pre-trained network can effectively act as a generic model of the visual world, which can be reused for new problems that may involve completely different classes (such as dog breed and human face recognition) from those of the original tasks[[2]](#footnote-2). This process is called transfer learning, and the project will utilize this practice, and compare its performance with the CNN only trained by the dog image dataset provided above.

For the development of CNN, the following parameters can be fine-tuned to improve the model performance:

* Image Data Augmentation
  + width\_shift\_range
  + height\_shift\_range
  + horizontal\_flip
* CNN Architecture
  + Type of Layers (convolutional, fully connected, or pooling)
  + Number of Layers
  + Parameters of Each Layer
* Model Fitting (Training) Parameters
  + Number of training cycles (epochs)
  + Batch size: The number of samples to work through during one training cycle
  + Type of Optimizers and Learning Rate

For the transfer learning, apart from the selection of a pre-trained model, the following options can be considered for model refinement:

* + Layers to be adopted from the pre-trained model
  + Layers to be added on top of the pre-trained model
  + Parameters to be trained or frozen

## Benchmark

The benchmark of the dog breed identification algorithm will be measured by the accuracy of prediction in the test dataset. The model returns the predicted vector where the probability for each dog breed is given by the softmax function. Thus, the dog breed with the highest probability in the predicted vector is returned as the prediction. The accuracy is measured for this prediction in the test dataset.

# **Methodology**

## Data Processing

As mentioned above, total 8,351 dog images are divided into 6,680 for training (ca.80%), 835 for validation (10%), 836 for test (10%) dataset, respectively. Data Processing consists of the following two steps:

1. Each dataset is converted into a 4D tensor.
2. Rescale the images by dividing every pixel in every image by 255
3. Image data augmentation

4D tensor is a 4D array with a shape of (number of samples, rows, columns channels), which Keras CNNs requires when using TensorFlow as the backend. To convert the datasets into 4D tensors, using [keras.preprocessing.image](https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image) module, the RGB image is loaded as PIL.Image.Image with the size (224, 224). Then PIL.Image.The image is converted to a 3D tensor array with the shape (224, 224, 3). Each 3D tensor is then converted to a 4D tensor with shape (1, 224, 224, 3) by using [NumPy.expand\_dims](https://numpy.org/doc/stable/reference/generated/numpy.expand_dims.html). Finally, the images are stacked into arrays in sequence vertically to have a 4D tensor of the images with shape (number of samples, 224, 224, 3).

*Image Data Augmentation*

To make the most of a small training dataset, image data augmentation is used to prevent overfitting and improve the model performance. This is done with [Keras.preprocessing.image.ImageDataGenerator](https://keras.io/api/preprocessing/image/) class. This class can configure random transformation on the images data during training, and instantiate generators of augmented image batches (and their labels) via .flow (data, labels). This generator is then used with the Keras model methods that accept data generators as inputs, fit\_generator, evaluate\_generator and predict\_generator.

## Implementation

The implementation of the project has the following steps:

1. Write and assess a human face detector
2. Write and assess a dog detector
3. Create and train CNNs to identify a dog breed
4. Write and test an overall algorithm for app deployment

***Step 1. Write and assess a human face detector*** *(Step 1. Detect Humans in Jupyter Notebook)*

The app should return the resembling dog breed when it receives human images. For this sake, a human face detector is needed. In this project, OpenCV’s implementation of [Harr feature-based cascade classifiers](https://docs.opencv.org/3.4/db/d28/tutorial_cascade_classifier.html) is first used and tested. OpenCV provides many pre-trained face detectors in XML format, and Haarcascades is one of them.

|  |
| --- |
| **import** cv2 *#import OpenCV libraries*  **import** numpy **as** np  **import** matplotlib.pyplot **as** plt  *# returns "True" if face is detected in image stored at img\_path*  **def** face\_detector(img\_path):  face\_cascade = cv2.CascadeClassifier('haarcascades/haarcascade\_frontalface\_alt.xml')  img = cv2.imread(img\_path)  gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)  faces = face\_cascade.detectMultiScale(gray)  return len(faces) > 0 |

Haar cascades-based human face detector was assessed with 100 images of human faces and dogs. While the face detector recognised all of the human faces successfully (100%), it also recognized 11% of dog images as human faces. The other pre-trained Cascade Classifier, LBP (Local Binary Patterns), was also tested but the performance was inferior to Haar Cascade Classifier (88% detection for a human face with LBP).

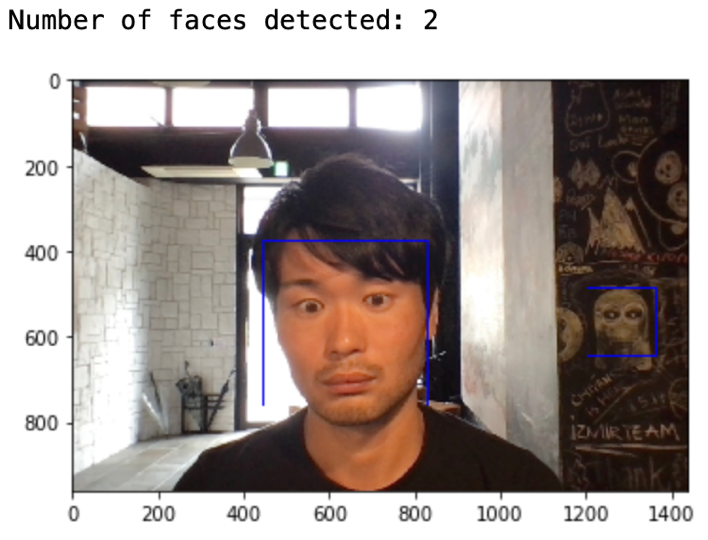
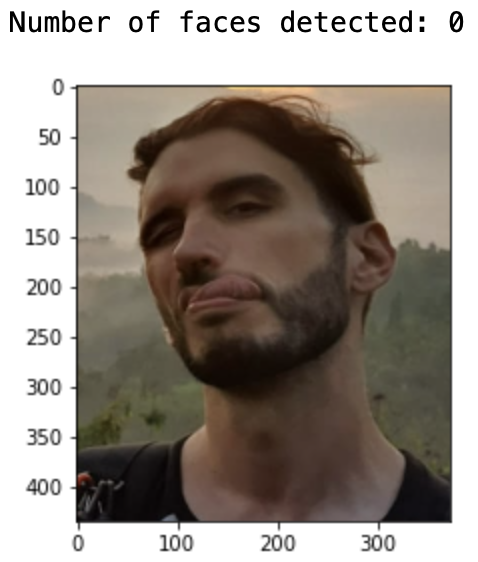
The other approach is to use deep learning. OpenCV has deep learning (dnn) module, which includes the Caffe-based face detector. This detector was also constructed and tested by following [this tutorial](https://pyimagesearch.com/2018/02/26/face-detection-with-opencv-and-deep-learning/). The trained model was downloaded from [here](https://github.com/spmallick/learnopencv/tree/master/FaceDetectionComparison). Caffe is an open-source structure for CNN and uses text documents with a predefined design for characterizing CNN’s construction. When using OpenCV’s deep neural network with Caffe models, the following two sets of files are needed:

1. **.prototxt** file – defines the model architecture
2. **.caffemodel** file – contains the weighs for the actual layers

OpenCV’s deep learning face detector is based on the Single Shot Detector (SSD) [[3]](#footnote-3) framework with a ResNet-based network. Given that the Caffe-based face detector required the latest OpenCV version, the performance of this face detector is tested in the local CPU, instead of Udacity’s workspace, which is with the older version of OpenCV.

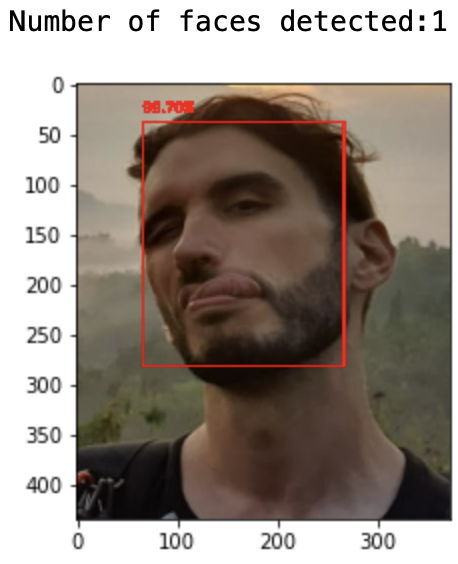
|  |
| --- |
| *# returns "True" if face is detected in image stored at img\_path*  **def** face\_detector\_cnn(img\_path, boundary=0.7):  net = cv2.dnn.readNetFromCaffe('deploy.prototxt','res10\_300x300\_ssd\_iter\_140000\_fp16.caffemodel')  *# load the model and weights*  img = cv2.imread(img\_path)  (h, w) = img.shape[:2]  blob = cv2.dnn.blobFromImage(cv2.resize(img, (300,300)), 1.0, (300, 300), (104.0, 177.0, 123.0))  *# pass the blob through the net to detect faces*  net.setInput(blob)  detections = net.forward()  *# count draw boxes around the detected faces with the probability larger than the boundary*  faces = 0  for i in range(0,detections.shape[2]):  confidence = detections[0, 0, i, 2]  if confidence > boundary:  faces += 1  *# compute the (x, y) coordinates of the bounding box for the object*  box = detections[0, 0, i, 3:7] \* np.array([w, h, w, h])  (startX, startY, endX, endY) = box.astype("int")  *# draw the bounding box of the face along with the associated probability*  text = "{:.2f}%".format(confidence \* 100)  y = startY - 10 if startY - 10 > 10 else startY + 10  cv2.rectangle(img, (startX, startY), (endX, endY),(0, 0, 255), 2)  cv2.putText(img, text, (startX, y),cv2.FONT\_HERSHEY\_SIMPLEX, 0.45, (0, 0, 255), 2)    print('Number of faces detected:{}'.format(faces))  *# show the output image*  img = cv2.cvtColor(img, cv2.COLOR\_BGR2RGB)  plt.imshow(img)  plt.show |

To compare the CNN Caffe-based face detector with Haar (ML) feature-based face detector, two pictures were tested. As in the below results, Haar faced detector could not detect fa ace when one of the eyes is not clear (left, or lateral face) and ended up identifying a simple drawing with two eyes as a human face (right).



*Figure 3. Demonstration of Haar Feature-based face detector*

On the other hand, the Caffe-based face detector successfully detected only human faces with the same picture with 70% of boundary confidence.



*Figure 4. Demonstration of CNN Caffe -based face detector*

Therefore, even though it could not be implemented in Udacity’s Workspace due to the issue of OpenCV’s version, it would be ideal to use CNN based pre-trained face detector to detect humans with not presented faces (e.g. side-view, winked etc)

***Step 2. Write and assess a dog detector*** *(Step 2. Detect Dogs in Jupyter Notebook)*

A pre-trained [ResNet-50](http://ethereon.github.io/netscope/#/gist/db945b393d40bfa26006) model is used to detect dogs in images. The pre-trained ResNet-to model returns a prediction from one of [1,000 categories](https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a) for the object in the image. The categories corresponding to dogs appear in an uninterrupted sequence and correspond to dictionary keys 151-268. To check to see if an image is predicted to contain a dog by the pre-trained ResNet-50 model, we need only check if the ResNet50\_predict\_labels function above returns a value between 151 and 268 (inclusive).

|  |
| --- |
| **from** keras.applications.resnet50 **import** ResNet50, preprocess\_input, decode\_predictions  **from** keras.preprocessing **import** image  **from** tqdm import tqdm  *# Load ResNet50 model along with the weights trained on ImageNet*  ResNet50\_model = ResNet50(weights='imagenet')  *# define data processor to convert image datasets to 4D tensor*  **def** path\_to\_tensor (img\_path):  img = image.load\_img(img\_path, target\_size=(224, 224))  x = image.img\_to\_array(img)  return np.expand\_dims(x, axis=0)  **def** paths\_to\_tensor (img\_paths):  list\_of\_tensors = [path\_to\_tensor(img\_path) for img\_path in tqdm(img\_paths)]  return np.vstack(list\_of\_tensors)  *# define the predictor that takes pre-processed 4D tensor*  **def** ResNet50\_predict\_labels (img\_path):  img = preprocess\_input(path\_to\_tensor(img\_path))  return np.argmax(ResNet50\_model.predict(img)) *# returns prediction vector for image*  *# returns "True" if a dog is detected in the image stored at img\_path*  **def** dog\_detector (img\_path):  prediction = ResNet50\_predict\_labels(img\_path)  return ((prediction <= 268) & (prediction >= 151)) |

As was done for the human face detector, the dog detector was tested with 100 images of both human faces and dogs respectively. The ResNet-50-based dog detector is very accurate, detecting dogs with 100% of accuracy while no human face was recognized as a dog. Thus, it is reasonable for the app to have the dog detector first, and then put the human face detector to check if an image contains human faces.

***Step 3. Create and train CNNs to identify dog breeds*** *(Steps 3-5 in Jupyter Notebook)*

This step will test and compare two different approaches to the development of CNN that classifies dog breeds. First, a CNN will be created from scratch without using transfer learning. Given the very small size of samples as well as the imbalanced distribution among breeds, the performance target of this approach is to exceed a random guess (1 in 133 times), which corresponds to an accuracy of less than 1%. In the second approach, transfer learning will be used to develop more accurate CNNs. In both implementations, Keras deep learning API will be used.

* 1. *Create a CNN from Scratch (Step 3 in Jupyter Notebook)*

The baseline architecture of the CNN is defined as follows:

|  |
| --- |
| **from** keras.layers **import** Conv2D, MaxPooling2D, GlobalAveragePooling2D  **from** keras.layers **import** Dropout, Flatten, Dense  **from** keras.models **import** Sequential  model = Sequential ()  model.add(Conv2D(filters=16, kernel\_size=7, strides = (2,2), padding='same', activation='relu', input\_shape=(224, 224, 3)))  model.add(MaxPooling2D(pool\_size=2))  model.add(Conv2D(filters=32, kernel\_size=2, padding='same', activation='relu'))  model.add(MaxPooling2D(pool\_size=2))  model.add(Conv2D(filters=64, kernel\_size=2, padding='same', activation='relu'))  model.add(MaxPooling2D(pool\_size=2))  model.add(Conv2D(filters=128, kernel\_size=2, padding='same', activation='relu'))  model.add(MaxPooling2D(pool\_size=2))  model.add(GlobalAveragePooling2D())  model.add(Dense(133, activation='softmax'))  model.summary() *# return the summary of the model with the number of parameters*  *# compile the model*  model.compile(optimizer='rmsprop', loss='categorical\_crossentropy', metrics=['accuracy']) |

The baseline model structure follows the general practice of CNN[[4]](#footnote-4). Typical CNN architectures stack a few convolutional layers (each one with ReLU activation), then a pooling layer, then another few convolutional layers (+ReLU), then another pooling layer, and so on. At the top of the stack, a regular feedforward neural network is added, composed of a few fully connected layers (+ReLUs), and the final layer outputs the prediction (e.g., a softmax layer that outputs estimated class probabilities).

The number of filters in the convolutional layer grows as it goes down deeper toward the output layer. This is because the number of low-level features is often quite low (e.g., circles, squares, lines etc.). On the other hand, there are many ways to combine them into higher-level or more complex features. Therefore, it is a common practice to double the number of filters after each pooling layer.

Max pooling layers (with 2x2 window and stride 2) are added after convolutional layers. Since a pooling layer divides each spatial dimension by two, it will lower the risk of exploding the number of parameters and overfitting. Max pooling also introduces some level of rotational and scale invariance. Such invariance albeit limited is believed to be useful for classification tasks.

The inputs for the output Dense Layer must be flattened since a dense network expects a 1D array for each instance, the number of which corresponds to the number of breed types, 133. The global average pooling layer outputs the mean of each feature map, dropping any remaining spatial information. Given that the challenge is a classification task, it does not matter where the object is. Thanks to the global average pooling layer, there is no need to include flattening and several fully connected layers before the output layer.

Once the CNN model is defined, the training will be implemented with the following code lines. Model refinement will be discussed in the later section.

|  |
| --- |
| **from** keras.callbacks **import** ModelCheckpoint  epochs = 30 *# specify the number of epochs*  checkpointer = ModelCheckpoint(  *# specify the path where the best model to be saved*  filepath='saved\_models/weights.best.from\_scratch.hdf5',  verbose=1,  save\_best\_only=True)  model.fit(train\_tensors, train\_targets,  validation\_data=(valid\_tensors, valid\_targets),  epochs=epochs, batch\_size=20, callbacks=[checkpointer], verbose=1) |

* 1. *Create a CNN using Transfer Learning*

As discussed in the following section (**Refinement**), the CNN from scratch can identify dog breeds with low accuracy (c.a. ~10%) even after the model optimization. To have a more accurate prediction algorithm, it is more practical to use transfer learning, where a pre-trained neural network model on a similar problem to the one we are trying to tackle (in this project, dog breed identification from image). This will considerably save training time and require significantly less training data.

While the pre-computed bottleneck features are provided by Udacity for several models, transfer learning without using them has been also implemented to familiarize me with a more real-world use case. In the below, the transfer learning with Xception is explained.

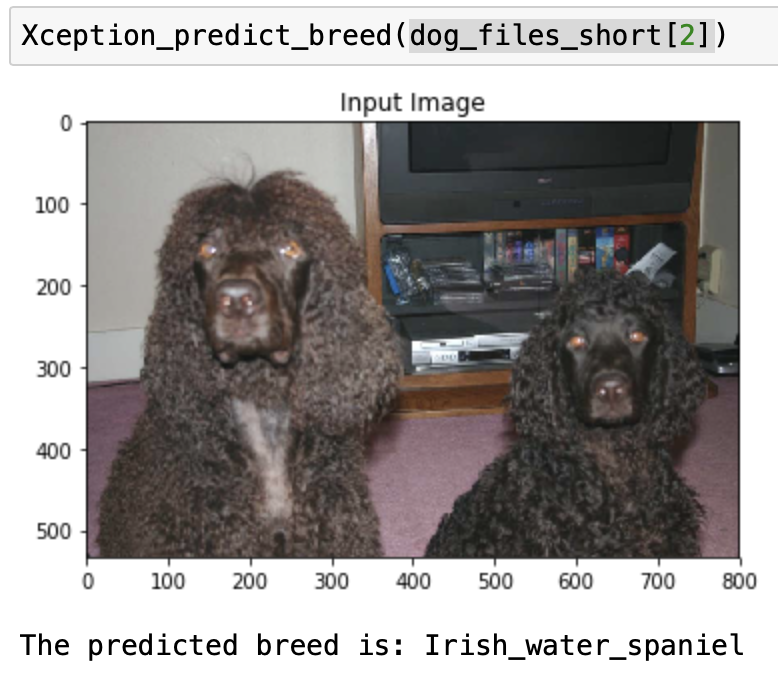
|  |
| --- |
| **import** keras.applications.xception **as** xception  *# Fuction for preprocessing images through Xception's proprocess\_input() function.*  *# After preprocssing images, store them as 4D tensor*  **def** preprocess\_xception(img\_paths):  list\_of\_tensors = []  for img\_path in tqdm(img\_paths):  final\_image = xception.preprocess\_input(path\_to\_tensor(img\_path))  list\_of\_tensors.append(final\_image)  return np.vstack(list\_of\_tensors)  *# preprocess the images for xception model*  train\_tensors\_xception = preprocess\_xception(train\_files)  valid\_tensors\_xception = preprocess\_xception(valid\_files)  test\_tensors\_xception = preprocess\_xception(test\_files)  *### define xception model*  *# exclude the top of the network and add fully connected layer preceeded by global average pooling*  base\_model = xception.Xception(weights='imagenet',include\_top=False)  avg = GlobalAveragePooling2D()(base\_model.output)  output = Dense(133, activation="softmax")(avg)  xception\_model = Model(input=base\_model.input, outputs=output)  *# freeze the weights of the pretrained layers*  for layer in base\_model.layers:  layer.trainable = False  xception\_model.summary() *# show model structure and trainable/un-trainable parameters*  *# compile the model*  xception\_model.compile(loss='categorical\_crossentropy',optimizer='rmsprop', metrics=['accuracy'])  *# train the model*  checkpointer = ModelCheckpoint(filepath='saved\_models/weights.best.xception.hdf5',  verbose=1, save\_best\_only=True)  xception\_history = xception\_model.fit(train\_tensors\_xception, train\_targets,  validation\_data=(valid\_tensors\_xception, valid\_targets),  epochs=10, batch\_size=20, callbacks=[checkpointer], verbose=1) |

Once the model is trained and refined to have met the benchmark (at least 60%), the next step is to create a function that takes an image path as input and returns the dog breed name predicted by the model. When using the bottleneck features provided by Udacity, the image first must be pre-processed for the model, passed to the prediction function of the pre-trained model without the top layer, and then the bottleneck features are extracted from the predicted vector. The code for this step can be found in the jupyter notebook.

Again, to familiarize me for a more general case, a prediction function without relying on pre-computed bottleneck features provided by the audacity is developed as follows.

|  |
| --- |
| **import** keras.applications.xception **as** xception  **def** Xception\_predict\_breed(img\_path):  *# Preprocess 4D tensor of img\_path for exception model*  xception\_input = xception.preprocess\_input(path\_to\_tensor(img\_path))  *# obtain predicted vector*  predicted\_vector = xception\_model.predict(xception\_input)  *# read the image*  img\_input = cv2.imread(img\_path)  img\_input = cv2.cvtColor(img\_input, cv2.COLOR\_BGR2RGB)  *# return the dog breed that is predicted by the model*  predicted\_breed = dog\_names[np.argmax(predicted\_vector)].split('.')[1]  plt.imshow(img\_input)  plt.title('Input Image')  plt.show()  print('The predicted breed is: {}'.format(predicted\_breed))  **return** predicted\_breed |

This function returns the predicted dog breed and the input image as in the below example.



*Figure 5. Demonstration of the Xception Prediction Function*

***Step 4. Write and test an overall algorithm for app deployment*** *(Steps 6-7 in Jupyter Notebook)*

Finally, the above functions are put together to develop an overall algorithm that accepts a file path to an image and computes as follows:

1. If a dog is detected in the image, return the predicted breed.
2. If a human is detected in the image, return the resembling dog breed.
3. If neither is detected in the image, provide an output that indicates an error.

|  |
| --- |
| **def** dog\_app\_xception(img\_path):  ''' The function of dog detection from an image file. The algorithm uses the  convolutional neural network. The function first deternmines whether the image  contains a human, dog, or neither. Then, the function will return as follows:    - if a dog is detected in the image, return the predicted breed.  - if a human is detected in the image, return the resembling dog breed.  - if neither is detected in the image, provide output that indicates an error.    Args:  img\_path: an image to be analyzed    Returns:  str: predicted breed if dog or human image is detected. If neither is dectected, error  message will be returend.  '''  **if** dog\_detector(img\_path):  Xception\_predict\_breed(img\_path)    **elif** face\_detector(img\_path):  print('human image is detected. The resembling dog breed is:')  Xception\_predict\_breed(img\_path)    **else**:  print('error: neither dog nor human image was detected. Please input a dog or human image.')  img\_input = cv2.imread(img\_path)  img\_input = cv2.cvtColor(img\_input, cv2.COLOR\_BGR2RGB)  plt.imshow(img\_input)  plt.title('Neither human nor dog')  plt.show() |

Below are the examples for each case.

|  |  |
| --- | --- |
| Graphical user interface, text, application  Description automatically generated | Graphical user interface, text, application  Description automatically generated |
| Graphical user interface, text, application, email  Description automatically generated | |

*Figure 6. Demonstration of the Dog Breed Identification Algorithm with Xception Model*

## Refinement

As described in the **Algorithms** section, many parameters can be modified to optimize the identification model.

**Image Augmentation**

Before using transfer learning, the CNN was constructed from the scratch (**3.1**). The learning curves of the CNN constructed from scratch without image augmentation are as follows (**Figure 7**).

Chart, line chart

Description automatically generated

*Figure 7. Learning curves of the CNN constructed from scratch.*

The decrease of validation loss (val\_loss) and training loss (loss) starts decoupling at the very early stage of the epoch (after c.a. 10 epochs), indicating the overfitting of the model. Image augmentation was also tested to address the small sample size and thus overfitting. The coding for image augmentation is as follows:

|  |
| --- |
| *### model definition and compilation follow the same*  *# create and confiture augmented image generator*  datagen\_train = ImageDataGenerator(  rotation\_range=40,  width\_shift\_range=0.1,  height\_shift\_range=0.1,  shear\_range=0.1,  zoom\_range=0.1,  horizontal\_flip=True)  *# fit augmented image generator on data*  batch\_size = 16  datagen\_train.fit(train\_tensors)  *# train the model*  checkpointercheckpoint = ModelCheckpoint(filepath='saved\_models/aug\_weights.best.from\_scratch.hdf5',  verbose=1, save\_best\_only=True)  aug\_history = aug\_model.fit\_generator(datagen\_train.flow(train\_tensors, train\_targets,batch\_size=batch\_size),  steps\_per\_epoch=train\_tensors.shape[0]//batch\_size,  epochs=epochs, verbose=1, callbacks=[checkpointer],  validation\_data=(valid\_tensors, valid\_targets),  validation\_steps=valid\_tensors.shape[0]//batch\_size) |

The learning curve with the image augmentation is as follows (**Figure 8**).

Chart, line chart

Description automatically generated

*Figure 8. Learning curves of the CNN constructed from scratch with image augmentation*

It seems that image augmentation indeed mitigates overfitting. The validation loss keeps steadily decreasing with the same degree as the training loss even after 10 epochs till 15-20 cycles. Moreover, the improvement of accuracy was observed in the training with the image augmentation negligible ( vs 11.9617%, with 30 epochs). Thus, it was implied that image augmentation can be utilized to improve the model performance by mitigating the risk of overfitting even with a very small sample size like this project. Having said that, it is clear that we should utilize transfer learning to achieve higher accuracy with the limited sample size.

***Transfer Learning and Model Selection***

According to [Keras Application](https://keras.io/api/applications/), the available pre-trained models have different accuracy, parameter, depth and required computing time. Notably, Xception outperforms VGG19, ResNet50 and InceptionV3 with fewer parameters and depth.

Xception (*‘Extreme Inception’*) by Google is a variant of Inception architecture[[5]](#footnote-5), and it significantly outperformed InceptionV3 on a larger image classification dataset comprising 350 million images and 17,000 classes. The architect of Xception, François Chollet (the author of Keras) suggests that the performance gains are not due to increased capacity but rather to more efficient use of model parameters because Xception architecture has the same number of parameters as Inception V3. In short, Xception merges the ideas of GoogLeNet and ResNet like Inception but replaces the inception modules with a special type of layer called a depthwise separable convolution layer. While a regular convolutional layer uses filters that try to simultaneously capture spatial patterns and cross-channel patterns, a separable convolutional layer makes the strong assumption that spatial patterns and cross-channel patterns can be modelled separately[[6]](#footnote-6). In addition, Xception also has residual (skip) connections, originally proposed by ResNet to have very deep CNN[[7]](#footnote-7). In practice, it is believed that separable convolutional layers generally perform better.

Taking into account this argument, Xception and ResNet50 were tested in this project hoping for the best performance. First, the pooling layer and fully connected layer are added to the pre-trained models without the top layers, and training was done while freezing the weights of the pre-trained layers. The learning curves of Xception and ResNet50 are as follows:

|  |  |
| --- | --- |
| Chart, line chart  Description automatically generated |  |

*Figure 9. Learning curves of the CNN with transfer learning. Left – Xception, Right – ResNet50*

Overfitting is the major obstacle for transfer learning with the milestone models. By this norm, the sign of overfitting is obvious in the above learning curve. For Xception-based CNN (Figure 9, left), the validation loss stopped decreasing steadily only after 2 epochs and decoupled with the training loss. As for that ResNet50, the learning curve of the ResNet50 pre-trained model shows that the steady decrease of validation loss stopped only after 1 epoch. On the other hand, the training loss keeps decreasing lower than the Xception model. This observation implies that ResNet50 might be more prone to overfitting with the provided dataset than Xception. The accuracy of the Xception pre-trained model was slightly better than that of ResNet50 after 30 epochs (79.6651% vs 78.3493%). Given the small size of training data as well as the similarity of data with ImageNet, unfreezing the weights of lower layers will not make sense. Thus, to improve the model further, the image augmentation was then tested with Xception pre-trained model.

The learning curve of the Xception pre-trained model combined with image augmentation is as follows.

**Chart, line chart

Description automatically generated**

*Figure 10. Learning curves of the pre-trained Xception with image augmentation*

Interestingly, the validation loss kept decreasing till epochs 6-8 albeit at a slow pace. However, the accuracy did not improve compared to that without image augmentation (79.4258% vs 79.6651).

The small size of training data as well as the similarity of data with ImageNet implies that unfreezing the weights of lower layers will not make sense. Thus, as the final attempt, only the weights of the second top convolutional layer (last separable convolutional layers) of pre-trained Xception were unfrozen and tested with the image augmentation to see if there is any performance improvement. This assumes that the last top layer might be responsible for more complex pattern recognition such as subtle differences in dog breeds. With this setting, the number of trainable parameters increases from 272,517 to 3,439,237 out of a total of 21,133,997 parameters.

Chart, line chart

Description automatically generated

*Figure 11. Learning curves of the Xception with unfrozen parameters combined with image augmentation*

Unfortunately, this approach did not work and only increased the validation loss. Thus, it seems that we need a larger sample size to further improve the model performance while mitigating the risk of overfitting.

Thus, the CNN based on pre-trained Xception trained with image augmentation was deployed for the dog identification algorithm given the lower risk of overfitting.

# **Results**

## Model Evaluation and Validation

The developed algorithm for a dog identification app was tested for several images to test its function.

Labradors are tricky because they come in yellow, chocolate and black yet have to be all classified as the same breed. The algorithm seems to be able to identify such cases correctly.

|  |  |
| --- | --- |
|  |  |

*Figure 12. Algorithm’s output for yellow and black Labradors pictures.*

Black (Toy/Miniature) Poodle, Portuguese Water Dog, and Irish Water Spaniel all look similar to each other. The algorithm could not identify black toy poodle images as in the below example. Given that it is difficult to even for a human to distinguish them and their hairstyle varies pretty wide range (largely depending on their host’s treatment), successful identification may be a difficult task.

|  |  |
| --- | --- |
| Graphical user interface, text, application  Description automatically generated | A picture containing text, mammal  Description automatically generated |

*Figure 13. Algorithm’s output for two images of Black Toy Poodle*

Regarding the human images, the dog breed estimation seems to rely on some characteristic spatial and cross-channel patterns. Hairstyles and colours seem to have a bigger influence than other characteristics (based on my examples' results).

|  |  |
| --- | --- |
| A picture containing grass, dog, outdoor, tree  Description automatically generated Graphical user interface, text, application  Description automatically generated | A dog lying on a bed  Description automatically generated with low confidence Graphical user interface, application  Description automatically generated |
| Graphical user interface, application  Description automatically generated | Graphical user interface, text, application  Description automatically generated |

*Figure 14. Examples of the Algorithm’s output for human face images and corresponding dog breed images.*

In some cases (e.g., right top and right bottom of figure 14), the characteristic patterns and similarity can be inferred from the images. However, it is not always the case and inference might be based on more complex patterns that humans cannot easily tell.

# **Conclusion**

## Reflection

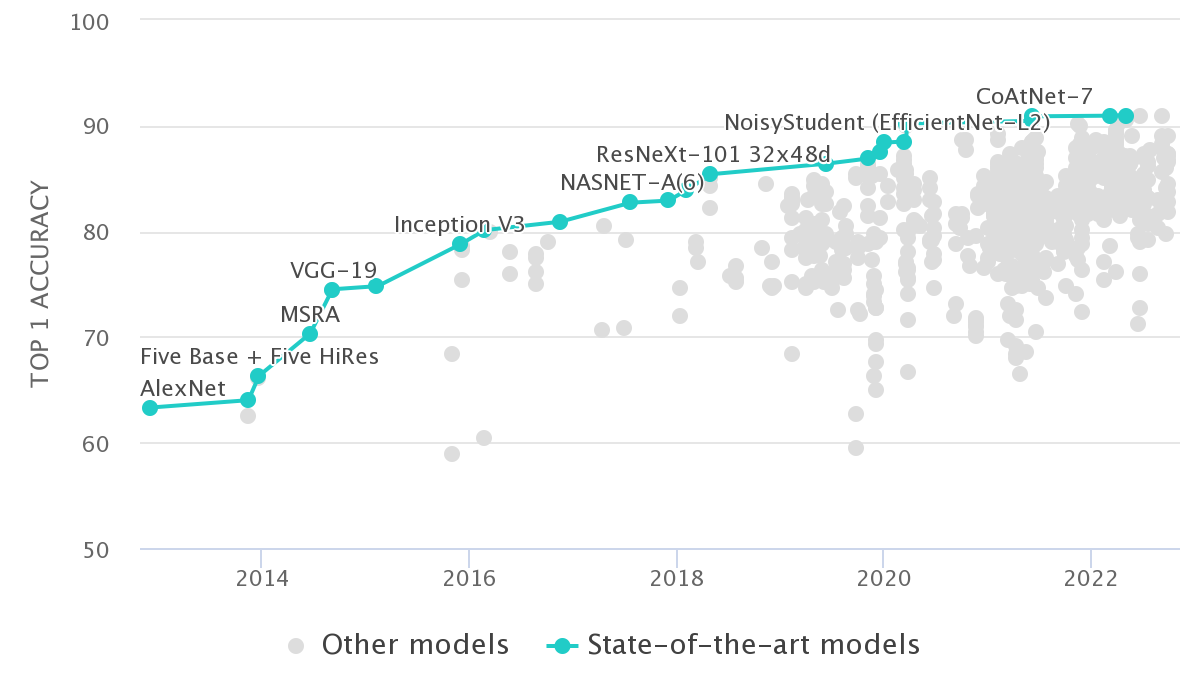
Even though the most of coding was simple or hints provided by Udacity, it took me a very long time to understand and digest the concepts of deep learning and use them to refine the model performance. The challenge for me was to design and how to select the parameters for the CNN model among so many variants. Try-and-error approaches while reading the relevant articles were helpful.

Overall, this project was a great example which fascinates me with how powerful and versatile deep learning is these days. Through the project, I had to do much research and reading, and this process gave me glimpses into the daily life of data scientists/machine learning engineers. I would like to seek the path of a data scientist/ML engineer.

## Improvement

The ImageNet is a dataset of over 15 million labelled high-resolution images belonging to roughly 22,000 categories. Since 2010, an annual competition called the [ImagneNet Large-Scale Visual Recognition Challenge (ILSVRC)](https://www.image-net.org/challenges/LSVRC/) has been held. Image classification is one of the challenging tasks and many key milestones such as Inception (GoogLeNet, ILSVRC-2014) and ResNet (ILSVRC-2015). Therefore, if the concerned image classification is similar to ImageNet’s categories, the model’s accuracy can be as high as the pre-trained milestone model used by transfer learning.

The current state-of-the-art models and their Top-1 accuracy of image classification on ImageNet are much higher than those of the model used in this project (**Figure 15**).



*Figure 15. Top 1 Accuracy of image classification on ImageNet by state-of-the-art models [[8]](#footnote-8).*

Thus, one approach to improve the model performance for dog identification is to simply use the models with higher Top-1 accuracy that are available in Keras Applications. For example, [EfficientNetV2](https://keras.io/api/applications/efficientnet_v2/#efficientnetv2l-function) has the highest Top-1 and Top5 accuracy (86.7% and 97.5%, respectively) among available pre-trained models in Keras Applications. As of now, Contrastive Captioner (CoCa) model obtains the highest Top-1 accuracy (91%) on ImageNet with a finetuned encoder[[9]](#footnote-9). However, since this model is implemented in Pytorch, the algorithm must be developed in Pytorch as well to utilize this state-of-the-art model.

The other issue is the classification of similar dog breeds. As demonstrated above, the similarity between some dog breeds is very high and almost impossible to distinguish them only by one image with high confidence (i.e., DNA test, inspection by a vet). Hence, the practical approach could be to return a few probable candidates, instead of giving only the highest probable one, while informing the user as such. Both Xception and ResNet50 have above 90% of Top-5 accuracy[[10]](#footnote-10). Thus, a possible algorithm might analyze the probability across the prediction vector and give top-5 candidates if the confidence of top-1 prediction is not so high.

1. Géron, Aurélien. Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (p. 355). [↑](#footnote-ref-1)
2. Francois Chollet, Deep Learning with Python (2017). [↑](#footnote-ref-2)
3. Liu, W et al (2015) SSD: Single Shot Multi-Box Detector. ECCV 2016. [arXiv:1512.02325v5](https://arxiv.org/abs/1512.02325v5) [↑](#footnote-ref-3)
4. Géron, Aurélien. Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow. O'Reilly Media. [↑](#footnote-ref-4)
5. François Chollet, “Xception: Deep Learning with Depthwise Separable Convolutions,” arXiv preprint arXiv:1610.02357 (2016). [↑](#footnote-ref-5)
6. Géron, Aurélien. Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow [↑](#footnote-ref-6)
7. Kaiming He et al., “Deep Residual Learning for Image Recognition,” arXiv preprint arXiv:1512:03385 (2015). [↑](#footnote-ref-7)
8. https://paperswithcode.com/sota/image-classification-on-imagenet [↑](#footnote-ref-8)
9. Yu, Jiahui et al. “CoCa: Contrastive Captioners are Image-Text Foundation Models.” *ArXiv*abs/2205.01917 (2022): n. page. [↑](#footnote-ref-9)
10. https://keras.io/api/applications/ [↑](#footnote-ref-10)