Dog Breed Classifier

Thiago Francisco Martins

Capstone Project Report

Introduction

In this project, the goal is to build a machine learning algorithm that given an image of a dog, it will identify an estimate of the canine's breed. If supplied an image of a human, the code will identify the resembling dog breed. The idea is to have a robust, fast and reliable algorithm that detects with high accuracy (metrics will be discussed later on) a dog's breed. The following paragraphs describes the main 5 stages in the developing of such algorithm. We manly used tensorflow and Keras to achieve our results.

1) Import the dataset and visualizing the data

In order to work with the data, we explicit download the dataset with the links provided using urllib.request package and zipfile.

```
[ ]  import urllib.request
     import zipfile
     import os

     # Download the file from `url` and save it locally under `file_name`:
     urllib.request.urlretrieve('https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/dogImages.zip', 'dogImages.zip')

     with zipfile.ZipFile('dogImages.zip', 'r') as zip_ref:
         zip_ref.extractall('dogImages/')

     os.remove('dogImages.zip')
```

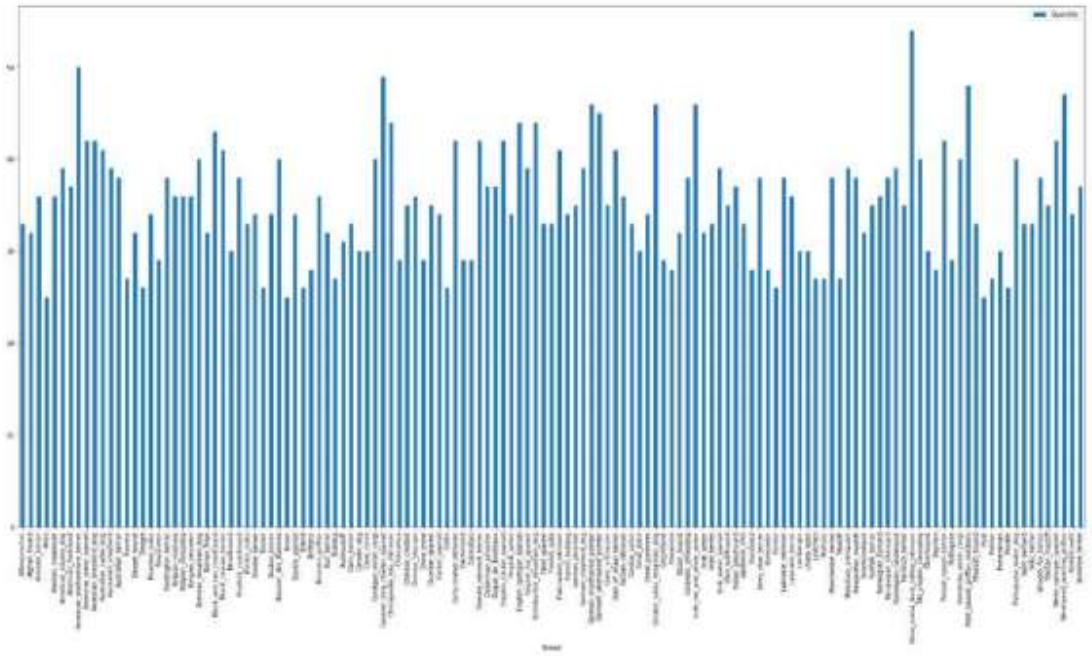
```
[ ]  # Download the file from `url` and save it locally under `file_name`:
     import shutil
     urllib.request.urlretrieve('https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/lfw.zip', 'lfw.zip')

     with zipfile.ZipFile('lfw.zip', 'r') as zip_ref:
         zip_ref.extractall('lfw/')

     os.remove('lfw.zip')
     shutil.rmtree('lfw/__MACOSX')
```

We then delete the .zip file that we just downloaded and can start working with the data.

We can see how many files we have on the dataset and how is the number of instances for each different dog breed we have as well as the number of photos of human faces.

```
[ ]  import numpy as np
     from glob import glob

     # load filenames for human and dog images
     human_files = np.array(glob("lfw/*/*/*"))
     dog_files = np.array(glob("dogImages/*/*/*/*"))

     # print number of images in each dataset
     print('There are %d total human images.' % len(human_files
     print('There are %d total dog images.' % len(dog_files))
```

```
There are 13233 total human images.
There are 8351 total dog images.
```



We can see that we have around 50 images per instance on average.

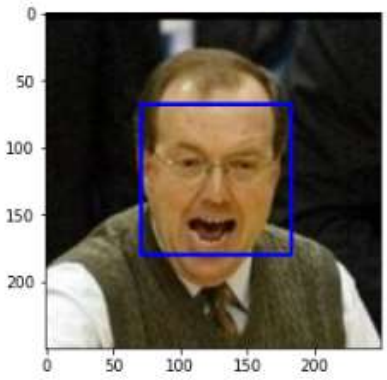
2) Human face detector and dog detector

a. Human face detector

For human detector, we used the haarcascade classifier provided by OpenCV.

```
[ ] import cv2
    import matplotlib.pyplot as plt
    %matplotlib inline

    # extract pre-trained face detector
    face_cascade = cv2.CascadeClassifier('haarcascades/haarcascade_frontalface_alt.xml')

    # load color (BGR) image
    img = cv2.imread(human_files[0])
    # convert BGR image to grayscale
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    # find faces in image
    faces = face_cascade.detectMultiScale(gray)

    # print number of faces detected in the image
    print('Number of faces detected:', len(faces))

    # get bounding box for each detected face
    for (x,y,w,h) in faces:
        # add bounding box to color image
        cv2.rectangle(img,(x,y),(x+w,y+h),(255,0,0),2)

    # convert BGR image to RGB for plotting
    cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

    # display the image, along with bounding box
    plt.imshow(cv_rgb)
    plt.show()
```

```
Number of faces detected: 1
```



We then create a Human face detector function:

```
[ ] # returns "True" if face is detected in image stored at img_path
    def face_detector(img_path):
        img = cv2.imread(img_path)
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        faces = face_cascade.detectMultiScale(gray)
        return len(faces) > 0
```

Finally, we can evaluate how this face detector classifier performs in 100 instances of human faces and 100 instances of dogs:

Question 1: Use the code cell below to test the performance of the face_detector function.

- What percentage of the first 100 images in human_files have a detected human face?
- What percentage of the first 100 images in dog_files have a detected human face?

Answer: (You can print out your results and/or write your percentages in this cell)

```
[ ] from tqdm import tqdm

    human_files_short = human_files[:100]
    dog_files_short = dog_files[:100]

    #-#-# Do NOT modify the code above this line. #-#-#

    ## TODO: Test the performance of the face_detector algorithm
    ## on the images in human_files_short and dog_files_short.

    face_human_accuracy = 0
    face_dog_accuracy=0
    for human in human_files_short:
        if face_detector(human)>0:
            face_human_accuracy +=1

    for dog in dog_files_short:
        if face_detector(dog)>0:
            face_dog_accuracy +=1

    print("Face detection on humans: "+ str(face_human_accuracy)+"%")
    print("Face detection on dogs: "+str(face_dog_accuracy)+"%")
```

```
Face detection on humans: 98%
Face detection on dogs: 15%
```

Ideally, we would like 100% of human images with a detected face and 0% of dog images with a detected face. You will see that our algorithm falls short of this goal, but still gives acceptable performance. We extract the file paths for the first 100 images from each of the datasets and store them in the numpy arrays human_files_short and dog_files_short.

b. Dog Detector

For Dog detection, we used a VGG-16 pre-trained model, which was trained to identify 1000 different classes, among dogs, flowers, people, and others. As this network was trained to identify some dogs, we use this network to our case where depending if the network thinks the image is from any dog breed it has learned to classify, we consider

the image a dog.

```
model = keras.applications.VGG16(weights="imagenet")
```

```
Using TensorFlow backend.
Downloading data from https://github.com/fchollet/deep-learning-models/releases/download/v0.1/vgg16_weights_tf_dim_ordering_tf_kernels.h5
553467904/553467096 [==============================] - 6s 0us/step
```

Given an image, this pre-trained VGG-16 model returns a prediction (derived from the 1000 possible categories in ImageNet) for the object that is contained in the image.

```
def VGG16_predict(img_path):
    '''
    Use pre-trained VGG-16 model to obtain index corresponding to
    predicted ImageNet class for image at specified path

    Args:
        img_path: path to an image

    Returns:
        Index corresponding to VGG-16 model's prediction
    '''

    ## TODO: Complete the function.
    ## Load and pre-process an image from the given img_path
    ## Return the *index* of the predicted class for that image
    img = image.load_img(img_path, target_size=(224, 224))
    x = image.img_to_array(img)
    x = np.expand_dims(x, axis=0)
    img_processed = preprocess_input(x)
    Y_pred = model.predict(img_processed).argmax(axis=1)
    return Y_pred # predicted class index
```

Question 2: Use the code cell below to test the performance of your dog_detector function.

Answer:

- What percentage of the images in human_files_short have a detected dog? 0%
- What percentage of the images in dog_files_short have a detected dog? 95%

3) Pre-Processing

The first step of pre-processing is to organize all the labels we have into numerical categories so we can feed our network.

```
# load filenames for human and dog images
dog_folders = "dogImages/dogImages/train/"

image_classes = len(np.array(glob("dogImages/dogImages/train/*/")))
# print number of images in each dataset
print('There are %d total dog classes.' % image_classes)

#CLASS_NAMES = sorted(np.array([item.name for item in test_dir.glob('*') if item.name != "LICENSE.txt"]))
CLASS_NAMES = []
for root, dirs, files in os.walk(dog_folders, topdown=False):
    for name in dirs:
        CLASS_NAMES.append(name)

CLASS_NAMES = sorted(CLASS_NAMES)
print("First 3 dogs:")
print(CLASS_NAMES[:3])

BATCH_SIZE = 32
IMG_HEIGHT = 299
IMG_WIDTH = 299
```

```
There are 133 total dog classes.
First 3 dogs:
['001.Affenpinscher', '002.Afghan_hound', '003.Airedale_terrier']
```

So, in this case, the index *'0'*, would be our first class, in this case named *'001.Affenpinscher'*

Then we use a class of keras named ImageDataGenerator. It is basically a class where we can set the functions we would like to apply to our images.

In the training dataset, we had 5 steps:

- Rescale the numerical values so the pixel values, which have a range from 0 to 255, fall into the range of 0 to 1, which the network will process much better
- Random horizontal flip of the image, so the network learns to distinguish a image even if its mirrored and avoid overfitting
- Randomly vertical flip of the image, so the network learns to identify the dog even if the image is upside-down
- Random rotate of the image, so the network learns to identify dogs even if they aren't in a straight position and avoid overfitting. Rotation can be 0 degrees or 45 degrees. With the vertical and horizontal flip we have possible random rotation of 0, 45, 90, 135, 180, 225, 270, 315.
- Random Brightness change, so the network learns to identify a given breed even if the image is underexposed or overexposed. It also helps avoid overfitting.

```
In [ ]: import tensorflow as tf
        import keras

        BATCH_SIZE = 32
        IMG_HEIGHT = 299
        IMG_WIDTH = 299
        batch_size = 32

        train_image_generator = keras.preprocessing.image.ImageDataGenerator(rescale=1./255,
                                                    horizontal_flip=True,
                                                    vertical_flip=True,
                                                    rotation_range=45,
                                                    brightness_range = [0.8, 1.2])
```

For the test and validation set we only apply the first step, which rescale the numerical values so the pixel values, which have a range from 0 to 255, fall into the range of 0 to 1.

```
valid_image_generator = keras.preprocessing.image.ImageDataGenerator(rescale=1./255)
test_image_generator = keras.preprocessing.image.ImageDataGenerator(rescale=1./255)
```

Then we can apply a method called Flow_from_directory, where we specify some more additional procedures so the generator can generates batches of data that we can feed our network.

In this step we have 3 main procedures

- Produces batches of size 32
- Resize the image so it has a height and width of 299, the same size required for our network we will use transfer learning in the next steps
- Shuffle the data

```
train_dir = 'dogImages/dogImages/train/'
valid_dir = 'dogImages/dogImages/valid/'
test_dir = 'dogImages/dogImages/test/'
```

In [ ]:
```
train_ds= train_image_generator.flow_from_directory(batch_size=batch_size,
                                                    directory=train_dir,
                                                    shuffle=True,
                                                    target_size=(IMG_HEIGHT, IMG_WIDTH),
                                                    classes = list(CLASS_NAMES))
```

```
Found 6680 images belonging to 133 classes.
```

In [ ]:
```
valid_ds = valid_image_generator.flow_from_directory(batch_size=batch_size,
                                                    directory=valid_dir,
                                                    shuffle=True,
                                                    target_size=(IMG_HEIGHT, IMG_WIDTH),
                                                    classes = list(CLASS_NAMES))
```

```
Found 835 images belonging to 133 classes.
```

In [ ]:
```
test_ds = test_image_generator.flow_from_directory(batch_size=batch_size,
                                                    directory=test_dir,
                                                    shuffle=False,
                                                    target_size=(IMG_HEIGHT, IMG_WIDTH),
                                                    classes = list(CLASS_NAMES))
```

```
Found 836 images belonging to 133 classes.
```

Here's an example of some instances:

Question 3: Describe your chosen procedure for preprocessing the data.

How does your code resize the images (by cropping, stretching, etc)? What size did you pick for the input tensor, and why? Did you decide to augment the dataset? If so, how (through translations, flips, rotations, etc)? If not, why not? Answer:

Answer:

We created a dataset by reading the files from the dataset directory and decoding them from jpeg to numeric tensors.

Our code resizes the image to 299 x 299 images, the same size as the network we will use for transfer learning in he next steps. We decided to augment the data to increase data variability, make the network robust to scene conditions and avoid overfitting. Our augment has 4 steps, all of them with random generation:

- Crop the image so it gets the 85% inner part
- Flip the image so we can mirror the image.
- Rotate the image so the network is resistant to rotation
- Brighten the image so the network is robust to changes in light conditions

4) Training the data

We had 2 approaches train the data, the one where we train the network from scratch and the one where we trained the network using transfer learning.

a) Scratch model

Question 4: Outline the steps you took to get to your final CNN architecture and your reasoning at each step.

Answer:

The default convolutional 2D filter is a wrapper around the Conv2D class. The aim is to avoid having to repeat the same hyperparameters values.

The first step is to apply a convolutional layer, which has several feature maps (16) and a a smal kernel size as the image is already small (299x299). We apply small strides and paddings in order to reduce the size of the maps and hence the computational power.

Then we apply a pooling layer to reduce the spatial dimension by a factor of 2.

Then we repeat the same structure twice, and doubling the number of features so the network can combine several low level filters into greater ones.

After the convolutional and pooling layers, a dropout layer is added to increase the model's performance and accuracy.

We then apply a global average pooling to reduce and flatten the size of the output of the previous steps.

On the final layers, we add a dense network to increase the performance on classifying images using relu activation function because of its speed.

And finally, we add a softmax layers with the correspondent nodes of the total dog breeds.

```
from functools import partial

keras.backend.clear_session()

DefaultConv2D = partial(keras.layers.Conv2D,
                        kernel_size=2, activation='relu', padding="SAME")
model_scratch = keras.models.Sequential([
    DefaultConv2D(filters=16, kernel_size=2, padding= 'same', strides=1,  input_shape=[299, 299, 3]),
    keras.layers.MaxPooling2D(pool_size=2),
    DefaultConv2D(filters=32),
    #DefaultConv2D(filters=64),
    keras.layers.MaxPooling2D(pool_size=2),
    DefaultConv2D(filters=64),
    #DefaultConv2D(filters=128),
    keras.layers.MaxPooling2D(pool_size=2),
    DefaultConv2D(filters=128),
    keras.layers.MaxPooling2D(pool_size=2),
    keras.layers.Dropout(0.4),
    keras.layers.GlobalAveragePooling2D(),
    #keras.layers.Flatten(),
    keras.layers.Dense(units=500, activation='relu'),
    #keras.layers.Dense(units=512, activation='relu'),
    #keras.layers.Dropout(0.2),
    keras.layers.Dense(units=133, activation='softmax'),
])
```

In [ ]:
```
model_scratch.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 299, 299, 16)      208
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 149, 149, 16)      0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 149, 149, 32)      2080
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 74, 74, 32)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 74, 74, 64)        8256
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 37, 37, 64)        0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 37, 37, 128)       32896
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 18, 18, 128)       0
_________________________________________________________________
dropout_1 (Dropout)          (None, 18, 18, 128)       0
_________________________________________________________________
global_average_pooling2d_1 ( (None, 128)               0
_________________________________________________________________
dense_1 (Dense)              (None, 500)               64500
_________________________________________________________________
dense_2 (Dense)              (None, 133)               66633
=================================================================
```

We defined a function to train the model. In this function we have 2 callbacks that apply an early stopping. Meaning that if the valuation accuracy doesn't improve for 10 epochs we stop the training and restore the model so it has the weights of the best accuracy value. Our loss function is the categorical cross entropy, which is suitable to classification tasks, and our optimizer is 'Nadam', which has superior performance and yields good results.

```
### TODO: select loss function
criterion_scratch = "categorical_crossentropy"

### TODO: select optimizer
#optimizer_scratch = keras.optimizers.SGD(lr=0.02, momentum=0.9, decay=0.005)
optimizer_scratch = 'nadam'
```

```python
def train(n_epochs, train_ds, valid_ds, model, optimizer, criterion, save_path):
  """returns trained model"""
  # initialize tracker for minimum validation loss

  checkpoint_cb = keras.callbacks.ModelCheckpoint(save_path, save_best_only=True, monitor='val_accuracy', mode='max',
  early_stopping_cb = keras.callbacks.EarlyStopping(patience=10, restore_best_weights=True)

  #train_images, train_labels = train_ds

  ###################
  # train the model #
  ###################
  model.compile(loss=criterion,
              optimizer=optimizer,
              metrics=['accuracy'])


  ######################
  # validate the model #
  ######################
  # print training/validation statistics
  history = model.fit(train_ds, epochs=n_epochs,
                    validation_data=valid_ds,
                    callbacks=[checkpoint_cb, early_stopping_cb])

      ## TODO: save the model if validation loss has decreased

  # return trained model
  return model
```

History of training:

```
Epoch 29/100
209/209 [==============================] - 189s 906ms/step - loss: 3.0561 - accuracy: 0.2205 - val_loss: 5.6920 - val_accuracy: 0.0994
Epoch 30/100
209/209 [==============================] - 188s 900ms/step - loss: 3.0020 - accuracy: 0.2341 - val_loss: 3.6499 - val_accuracy: 0.0850
Epoch 31/100
209/209 [==============================] - 188s 899ms/step - loss: 2.9475 - accuracy: 0.2466 - val_loss: 3.5013 - val_accuracy: 0.0874
Epoch 32/100
209/209 [==============================] - 187s 896ms/step - loss: 2.9255 - accuracy: 0.2466 - val_loss: 4.1271 - val_accuracy: 0.1054
Epoch 33/100
209/209 [==============================] - 187s 895ms/step - loss: 2.8768 - accuracy: 0.2528 - val_loss: 3.5450 - val_accuracy: 0.1126
Epoch 34/100
209/209 [==============================] - 187s 896ms/step - loss: 2.8111 - accuracy: 0.2689 - val_loss: 6.1029 - val_accuracy: 0.1018
Epoch 35/100
209/209 [==============================] - 186s 890ms/step - loss: 2.7974 - accuracy: 0.2698 - val_loss: 6.2699 - val_accuracy: 0.0826
Epoch 36/100
209/209 [==============================] - 187s 892ms/step - loss: 2.7447 - accuracy: 0.2762 - val_loss: 7.5000 - val_accuracy: 0.1126
```

b) Training using transfer learning

Question 5: Outline the steps you took to get to your final CNN architecture and your reasoning at each step. Describe why you think the architecture is suitable for the current problem.

Answer: We downloaded a pre-trained Xception model which is suitable to the problem we want to tackle and performs well regarding training speed and resources.

We got rid of the last layers since we want to classify 133 classes of dogs while Imagenet dataset has over 1000 classes from different kind of things (plants, objects, people..)

After that we only applied a global average pooling layer to reduce the computational power in a way that information from feature maps are somewhat preserved. Global average pooling layer cut off a lot of the spatial information from previous layers but it is necessary to have a good trade-off between training time and accuracy.

The last step is to add a dense layer with softmax activation function, suitable to classification tasks.

```
keras.backend.clear_session()
base_model = keras.applications.xception.Xception(weights="imagenet", include_top=False)
avg = keras.layers.GlobalAveragePooling2D()(base_model.output)
output = keras.layers.Dense(133, activation="softmax")(avg)
model_transfer = keras.models.Model(inputs=base_model.input, outputs=output)

for layer in base_model.layers:
    layer.trainable = False
```

```
In [ ]: criterion_transfer = 'categorical_crossentropy'
        optimizer_transfer = 'nadam'
```

Our training method consists of training the data with all except the last layer frozen so that they don't get updated by the training and only the last layer (our softmax function) learns the best patterns. Then after achieving good accuracy, we unfreeze the bottom layers and train the network again, this time with all layers free to update its weights.

```
model_transfer = train(100, train_ds, valid_ds, model_transfer, optimizer_transfer,
                       criterion_transfer, 'model_transfer.h5')


model_transfer = keras.models.load_model('model_transfer.h5')

for layer in base_model.layers:
    layer.trainable = True

model_transfer = train(100, train_ds, valid_ds, model_transfer, optimizer_transfer,
                       criterion_transfer, 'model_transfer.h5')

# load the model that got the best validation accuracy
model_transfer = keras.models.load_model('model_transfer.h5')
```

History of training

```
Epoch 2/100
209/209 [==============================] - 194s 927ms/step - loss: 0.3726 - accuracy: 0.8855 - val_loss: 0.7127 - val_accuracy: 0.8754
Epoch 3/100
209/209 [==============================] - 195s 935ms/step - loss: 0.3475 - accuracy: 0.8922 - val_loss: 0.5999 - val_accuracy: 0.8635
Epoch 4/100
209/209 [==============================] - 198s 948ms/step - loss: 0.3446 - accuracy: 0.8901 - val_loss: 3.3300e-04 - val_accuracy: 0.8790
Epoch 5/100
209/209 [==============================] - 199s 952ms/step - loss: 0.3327 - accuracy: 0.8967 - val_loss: 0.0045 - val_accuracy: 0.8838
Epoch 6/100
209/209 [==============================] - 199s 950ms/step - loss: 0.3374 - accuracy: 0.8963 - val_loss: 0.0854 - val_accuracy: 0.8707
Epoch 7/100
209/209 [==============================] - 202s 969ms/step - loss: 0.3482 - accuracy: 0.8874 - val_loss: 1.2451 - val_accuracy: 0.8695
Epoch 8/100
209/209 [==============================] - 202s 967ms/step - loss: 0.3449 - accuracy: 0.8907 - val_loss: 1.7773 - val_accuracy: 0.8766
Epoch 9/100
209/209 [==============================] - 203s 969ms/step - loss: 0.3212 - accuracy: 0.8993 - val_loss: 1.0491 - val_accuracy: 0.8647
Epoch 10/100
209/209 [==============================] - 202s 966ms/step - loss: 0.3174 - accuracy: 0.8960 - val_loss: 0.0152 - val_accuracy: 0.8611
Epoch 11/100
209/209 [==============================] - 202s 968ms/step - loss: 0.3151 - accuracy: 0.9010 - val_loss: 8.1267e-04 - val_accuracy: 0.8719
Epoch 12/100
209/209 [==============================] - 202s 968ms/step - loss: 0.3116 - accuracy: 0.9003 - val_loss: 0.0723 - val_accuracy: 0.8826
Epoch 13/100
209/209 [==============================] - 200s 956ms/step - loss: 0.3169 - accuracy: 0.8960 - val_loss: 1.0553 - val_accuracy: 0.8766
Epoch 14/100
209/209 [==============================] - 197s 941ms/step - loss: 0.2999 - accuracy: 0.8999 - val_loss: 0.3503 - val_accuracy: 0.8778
```

5) Test the model and results

We evaluate both methods (from scratch and using transfer learning) under 2 metrics.

- Accuracy, which is the number of correct predicted instances divided by the total of instances
- Multi Log Loss, which accurately represents not only if the classes are labeled correct but also how confident our model is when it predicted classes are equal to the target

$$logloss = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij}\log(p_{ij})$$

Where,

N	*No of Rows in Test set*
M	*No of Fault Delivery Classes*
Y_{ij}	*1 if observation belongs to Class j; else 0*
P_{ij}	*Predicted Probability that observation belong to Class j*

a) Model from scratch

Accuracy: 10.765%

```
[ ]  def test(test_ds, model):

       #y_test = test_ds.labels
       score= model.evaluate(test_ds)

       print('Test Loss: {:.6f}'.format(score[0]))

       print('\nTest Accuracy: %2f %%' % (100*score[1]))

     # call test function
     test(test_ds, model_scratch)
```

```
27/27 [==============================] - 6s 229ms/step
Test Loss: 6.479346

Test Accuracy: 10.765550 %
```

Multi Log Loss: 4.222

```
[ ]  multiclass_loss_loss = multiclass_log_loss(y_true, y_pred)
     print('Multiclass Log Loss: %4f' % (multiclass_loss_loss))
```

```
Multiclass Log Loss: 4.222705
```

b) Transfer Learning

Accuracy: 86.722%

```
[ ] test(test_ds, model_transfer)
```

```
27/27 [==============================] - 7s 263ms/step
Test Loss: 1.102527

Test Accuracy: 86.722487 %
```

Multi Log Loss: 0.411

```
[ ] y_pred = model_transfer.predict(test_ds)

    y_true = test_ds.labels

    multiclass_loss = multiclass_log_loss(y_true, y_pred)
    print('Multiclass Log Loss: %4f' % (multiclass_loss))
```

```
Multiclass Log Loss: 0.410422
```

After testing, we can finally build our own algorithm that first detects if a given image is from a dog, or human, and then predicts which dog breed is the most close to it.

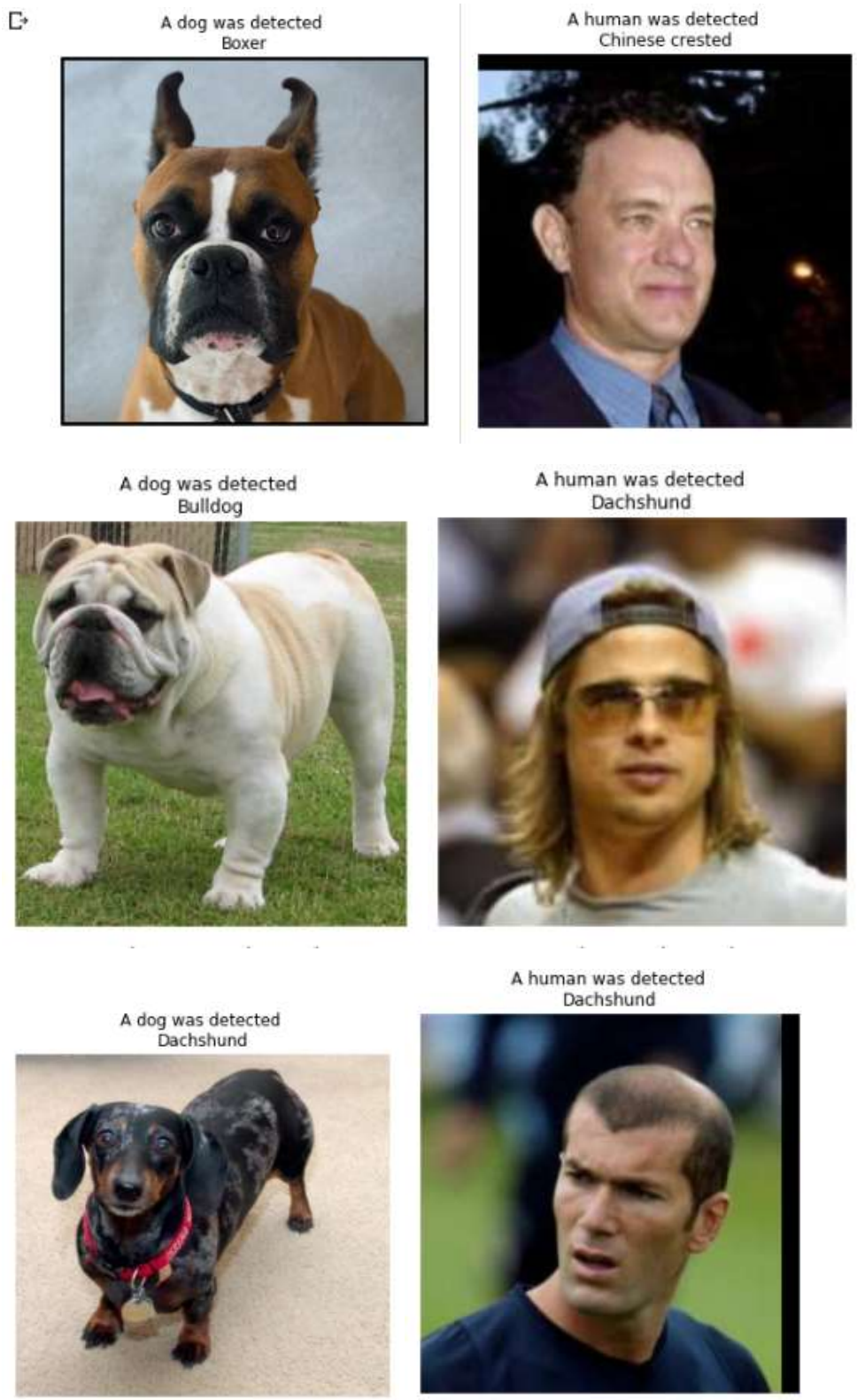
```
[ ] ### TODO: Write your algorithm.
    ### Feel free to use as many code cells as needed.
    import cv2

    def run_app(img_path):
      ## handle cases for a human face, dog, and neither
      img = cv2.imread(img_path)
      img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
      plt.figure(figsize=(5,5))
      plt.axis('off')
      if dog_detector(img_path):
          plt.title('A dog was detected \n'+ predict_breed_transfer(img_path))
          plt.imshow(img_rgb)
      elif face_detector(img_path):
          plt.title('A human was detected \n'+ predict_breed_transfer(img_path))
          plt.imshow(img_rgb)
      else:
          pass
          #raise ValueError("Dog or Face not detected")
```

Question 6: Is the output better than you expected :) ? Or worse :(? Provide at least three possible points of improvement for your algorithm.

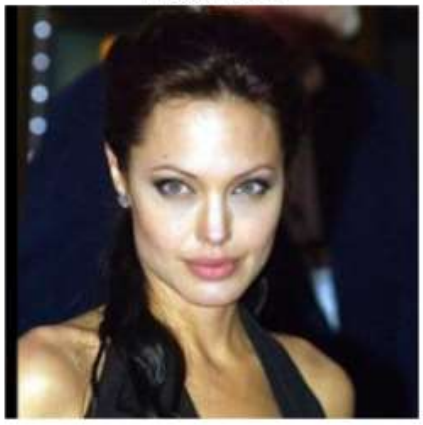
Answer: The output is better than expected, we were expecting results in the range of the 60%, but got around 86%. We could improve our algorithm by feeding more

training instances, by creating a detector that crops only the portion of the image that belongs to a dog, or even trying other networks like Resnet or GoogleNet.

Results



A human was detected
Pharaoh hound



A human was detected
Cardigan welsh corgi

A dog was detected
Dalmatian

A dog was detected
Labrador retriever

A dog was detected
Golden retriever

A dog was detected
Border collie