Machine Learning Engineer Nanodegree

Capstone Project Report

By Abhilasha Kumari

Project: Write an Algorithm for a Dog Identification App

I. Definition

This project includes algorithm to detect breed of dog as per the sample input and if instead of dog, humans is present it identifies that and then tells that which breed of dog they are most similar to. In this project, a series of models are used together to perform different tasks; for instance, the algorithm that detects humans in an image will be different from the CNN that infers dog breed.

II. Different stages of Project

Step 0: Import Datasets

Import Dog Dataset

In the code cell below, we import a dataset of dog images. We populate a few variables through the use of the load_files function from the scikit-learn library:

- train_files, valid_files, test_files - numpy arrays containing file paths to images
- train_targets, valid_targets, test_targets - numpy arrays containing onehot-encoded classification labels
- dog_names - list of string-valued dog breed names for translating labels

Import Human Dataset

In the code cell below, we import a dataset of human images, where the file paths are stored in the numpy array human_files.

Step 1: Detect Humans

We use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images. OpenCV provides many pre-trained face detectors, stored as XML files on github. We have downloaded one of these detectors and stored it in the haarcascades directory. Before using any of the face detectors, it is standard procedure to convert the images to grayscale. The detectMultiScale function executes the classifier stored in face_cascade and takes the grayscale image as a parameter.

Write a Human Face Detector

We can use this procedure to write a function that returns True if a human face is detected in an image and False otherwise. This function, aptly named face_detector, takes a string-valued file path to an image as input and appears in the code block below.

Output of the algorithm:-

1. Correctly detected human faces: 98%
2. Correctly detected dog faces: 17%

Step 2: Detect Dogs

In this section, we use a pre-trained ResNet-50 model to detect dogs in images. Our first line of code downloads the ResNet-50 model, along with weights that have been trained on ImageNet, a very large, very popular dataset used for image classification and other vision tasks. ImageNet contains over 10 million URLs, each linking to an image containing an object from one of 1000 categories. Given an image, this pre-trained ResNet-50 model returns a prediction (derived from the available categories in ImageNet) for the object that is contained in the image.

Pre-process the Data

When using TensorFlow as backend, Keras CNNs require a 4D array (which we'll also refer to as a 4D tensor) as input, with shape
where nb_samples corresponds to the total number of images (or samples), and rows, columns, and channels correspond to the number of rows, columns, and channels for each image, respectively.
The path_to_tensor function below takes a string-valued file path to a color image as input and returns a 4D tensor suitable for supplying to a Keras CNN. The function first loads the image and resizes it to a square image that is 224 pixels. Next, the image is converted to an array, which is then resized to a 4D tensor. In this case, since we are working with color images, each image has three channels. Likewise, since we are processing a single image (or sample), the returned tensor will always have shaped

The paths_to_tensor function takes a numpy array of string-valued image paths as input and returns a 4D tensor with shape

Making Predictions with ResNet-50

Getting the 4D tensor ready for ResNet-50, and for any other pre-trained model in Keras, requires some additional processing. First, the RGB image is converted to BGR by reordering the channels. All pre-trained models have the additional normalization step that the mean pixel (expressed in RGB as $[103.939, 116.779, 123.68]$ and calculated from all pixels in all images in ImageNet) must be subtracted from every pixel in each image. This is implemented in the imported function preprocess_input.
Now that we have a way to format our image for supplying to ResNet-50, we are now ready to use the model to extract the predictions. This is accomplished with the predict method, which returns an array whose i-th entry is the model's predicted probability that the image belongs to the i-th ImageNet category. This is implemented in the ResNet50_predict_labels function below.
By taking the argmax of the predicted probability vector, we obtain an integer corresponding to the model's predicted object class, which we can identify with an object category through the use of this dictionary.

Write a Dog Detector

While looking at the dictionary, you will notice that the categories corresponding to dogs appear in an uninterrupted sequence and correspond to dictionary keys 151-268, inclusive, to include all categories from 'Chihuahua' to 'Mexican hairless'. Thus, in order 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).
We use these ideas to complete the dog_detector function below, which returns True if a dog is detected in an image (and False if not).

Assess the Dog Detector

Testing the performance of the dog_detector function below,

- Detected human as dog: 0%
- Correctly detected dog faces: 98%

Step 3: Create a CNN to Classify Dog Breeds (from Scratch)

Now that we have functions for detecting humans and dogs in images, we need a way to predict breed from images. In this step, you will create a CNN from scratch that classifies dog breeds. We also mention that random chance presents an exceptionally low bar: setting aside the fact that the classes are slightly imbalanced, a random guess will provide a correct answer roughly 1 in 133 times, which corresponds to an accuracy of less than 1%.

Pre-process the Data

We rescale the images by dividing every pixel in every image by 255.

Model Architecture

I started off with the hinted architecture with a kernel_size of (7,7) and got a validation accuracy of 1% after 15 epochs. To try a different approach to prevent overfitting, I added two hidden dropout layers with a rate of 0.5 and went back to a kernel_size of (7,7). This still had a validation accuracy of around 0.02 after 3 epochs so I changed my kernel_size to (5,5) and reduced my dropout rate to 0.3 to reduce the number of units dropped and retain more information but decrease the number of total parameters.
This architecture works because the series of convolution layers and max pooling layers learns simple to complex patterns as the network gets deeper. The convolution layers help capture spatial information while the max pooling layers help reduce the dimensions and number of parameters to be trained. The dropout layers help with overfitting by randomly dropping some of the nodes in the network during training. The global average pooling layer reduces the dimensions drastically and converts the 3D array into a vector before the final fully connected layer is used to produce the output.

The CNN Architecture for the model is given below :-

```
class Net(nn.Module):
    ### TODO: choose an architecture, and complete the class
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, 3, padding=1)
        # convolutional layer (sees 16x16x16 tensor)
        self.conv2 = nn.Conv2d(16, 32, 3, padding=1)

        # convolutional layer (sees 8x8x32 tensor)
        self.conv3 = nn.Conv2d(32, 64, 3, padding=1)

        # max pooling layer
        self.pool = nn.MaxPool2d(2, 2)
        # linear layer (64 * 28 * 28 -> 500)
        self.fc1 = nn.Linear(64 * 28 * 28, 500)
        # linear layer (500 -> 133)
        self.fc2 = nn.Linear(500, 133)
        # dropout layer (p=0.25)
        self.dropout = nn.Dropout(0.25)
        self.batch_norm = nn.BatchNorm1d(num_features=500)
    def forward(self, x):
        ## Define forward behavior
        x = self.pool(F.relu(self.conv1(x)))

        # add dropout layer
        x = self.dropout(x)

        x = self.pool(F.relu(self.conv2(x)))

        # add dropout layer
        x = self.dropout(x)

        x = self.pool(F.relu(self.conv3(x)))

        # add dropout layer
        x = self.dropout(x)

        # flatten image input
        # 64 * 28 * 28
#         x = x.view(-1, 64 * 28 * 28)
        x = x.view(x.size(0), -1)

        # add 1st hidden layer, with relu activation function
        x = F.relu(self.batch_norm(self.fc1(x)))

        # add dropout layer
        x = self.dropout(x)

        # add 2nd hidden layer, with relu activation function
        x = self.fc2(x)
        return x
```

Assessment of the architecture

I got a test accuracy of 1% (16/836) with 15 epochs.

Step 5: Create a CNN to Classify Dog Breeds (using Transfer Learning)

I used transfer learning to create a CNN using VGG-16 bottleneck features. In this section, I have used the bottleneck features from a different pre-trained model. To make things easier , I have pre-computed the features for all of the networks that are currently available in Keras. These are already in the workspace, at /data/bottleneck_features.

Model Architecture

Just like the previous example, I used a pre-trained model (Resnet50) and used the last layer as the input to a Global average pooling layer to reduce the dimenstions. I then fed this to a fully connected layer with 133 nodes - one for each category - for the output. I got 83% accuracy with 10 epochs.
This architecture is suitable because we're taking advantage of the information/patterns learned by the layers of the pre-trained model and training it further to learn what we want.

Predicting Dog Breed with the Model

I have written a function that takes an image path as input and returns the dog breed (Affenpinscher, Afghan_hound, etc) that is predicted the model.
The function has three steps:

1. Extract the bottleneck features corresponding to the chosen CNN model.
2. Supply the bottleneck features as input to the model to return the predicted vector. Note that the argmax of this prediction vector gives the index of the predicted dog breed.
3. Use the dog_names array defined in Step 0 of this notebook to return the corresponding breed.

The functions to extract the bottleneck features can be found in extract_bottleneck_features.py, and they have been imported in an earlier code cell. To obtain the bottleneck features corresponding to your chosen CNN architecture, you need to use the function

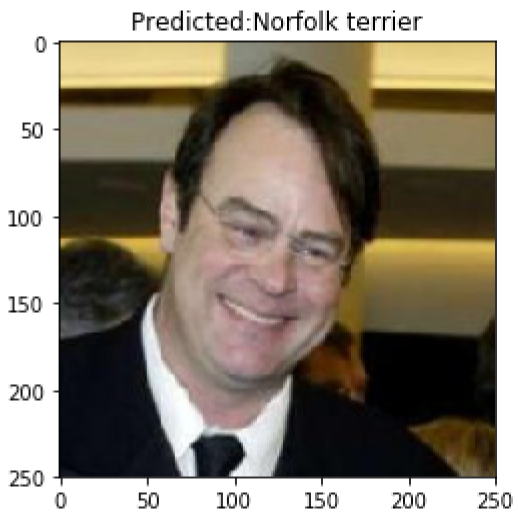
Step 6: Write final Algorithm

Write an algorithm that accepts a file path to an image and first determines whether the image contains a human, dog, or neither. Then,

- 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.

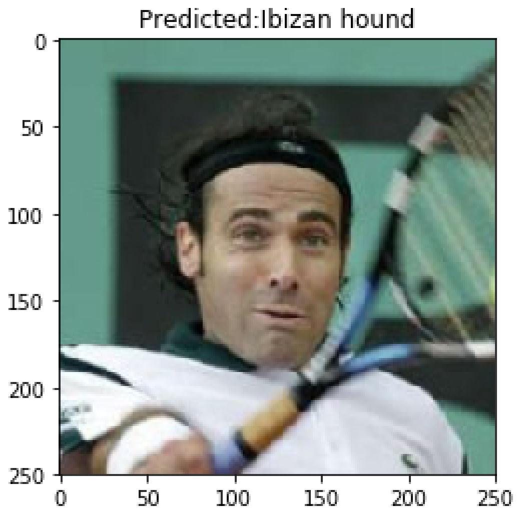
Few of the output of the algorithm is as below:-

Hello Human!



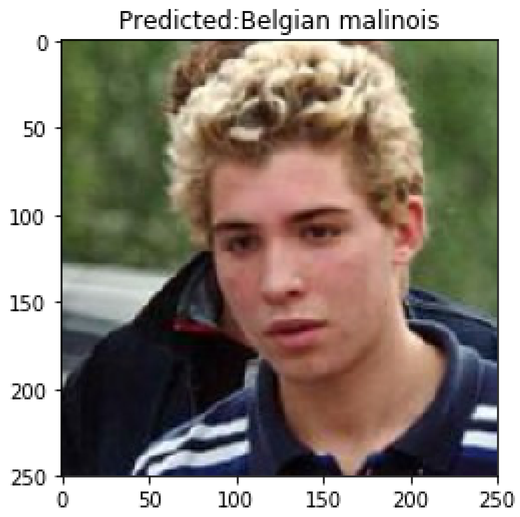
You look like a ...
NORFOLK TERRIER

Hello Human!



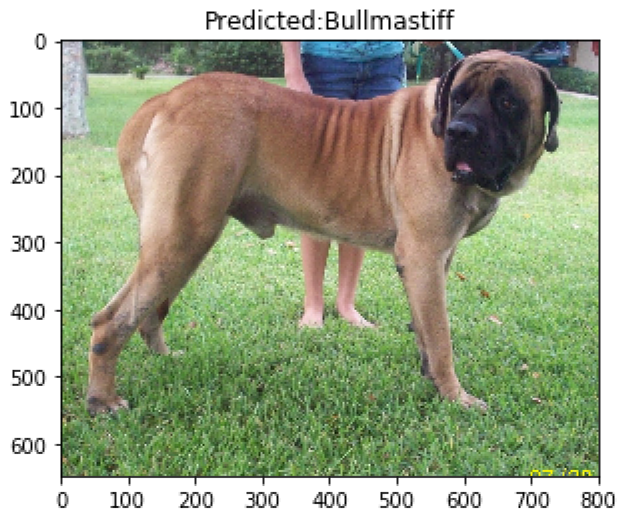
You look like a ...
IBIZAN HOUND

Hello Human!



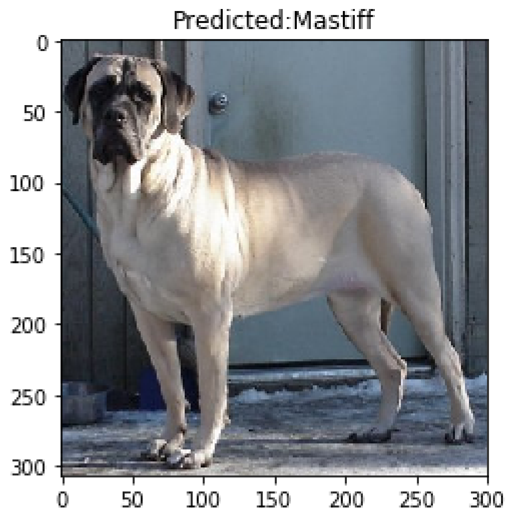
You look like a ...
BELGIAN MALINOIS

Hello Doggie!



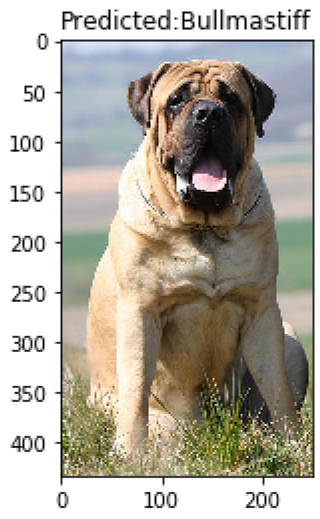
Your breed is most likley ...
BULLMASTIFF

Hello Doggie!



```
Your breed is most likley ...
MASTIFF


Hello Doggie!
```



```
Your breed is most likley ...
BULLMASTIFF
```