Deep Learning: How to build a dog detector and breed classifier using Convolutional Neural Networks

Overview:

I will walk you through how to Convolutional Neural Networks (CNN) leverage the latest state of art image classification techniques on ImageNet. This model can be used as part of a mobile or web app for the real world and user-provided images. Given an image to the model, it determines if a dog is present and returns the estimated breed. If the image is human, it will return the most resembling dog breed.

The steps that were followed to work through the project were the following:

- Step 0: Import Datasets
- Step 1: Detect Humans
- Step 2: Detect Dogs
- Step 3: Create a CNN to classify Dog Breeds (from scratch)
- Step 4: Use a CNN to classify Dog Breeds (using Transfer Learning)
- Step 5: Create a CNN to classify Dog Breeds (using Transfer Learning)
- Step 6: Write an algorithm
- Step 7: Test algorithm

The datasets were provided by Udacity.

- Dog Images — The dog images provided are available in the repository within the Images directory further organized into the train, valid and test subfolders.
- Human Faces — An exhaustive dataset of faces of celebrities have also been added to the repository in the folder
- Haarcascades — ML-based approach where a cascade function is trained from a lot of positive and negative images and used to detect objects in other images. The algorithm uses the Haar frontal face to detect humans. So the expectation is that an image with the frontal features clearly defined is required
- Test Images — A folder with certain test images have been added to be able to check the effectiveness of the algorithm.
- Pre-computed features for networks currently available in Keras (i.e. VGG19, InceptionV3, and Xception) will be made available from S3
- any other downloads to ensure the smooth running of the notebook are available in the repository.

Dog detector

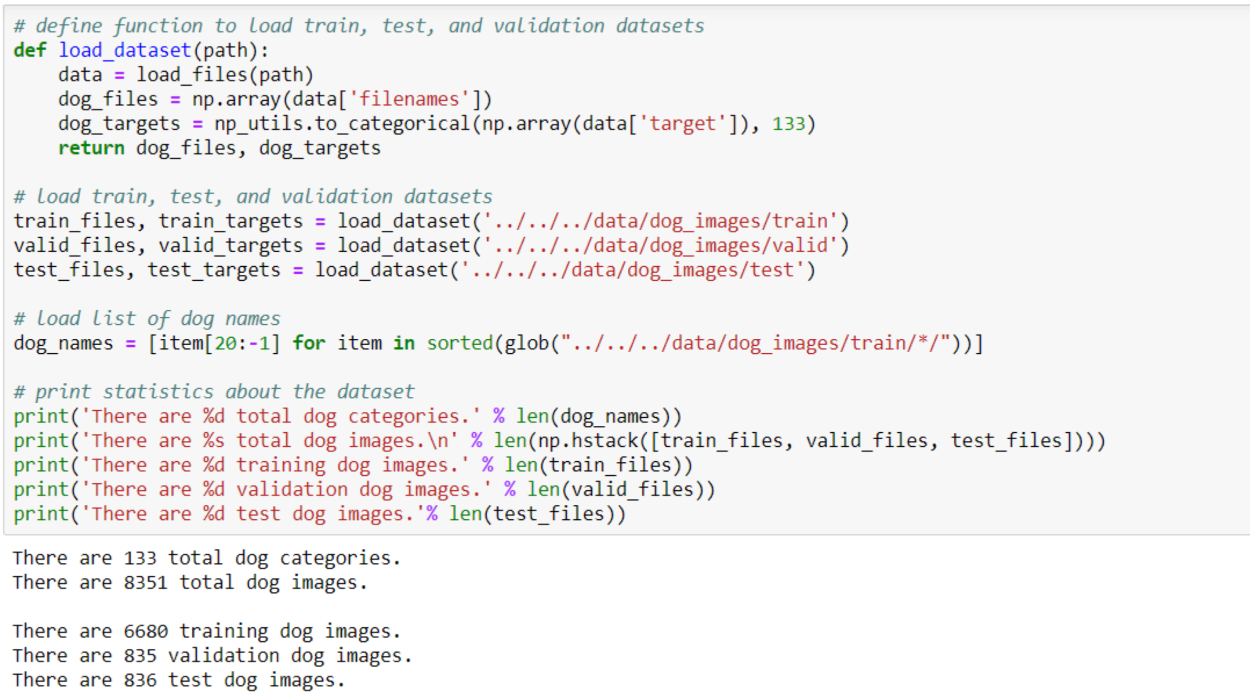
I used pre-trained ResNet50 weights on ImageNet in Keras, which is trained on over 10 million images containing 1000 labels. Predict weather mode performs by running CNN's is below image is dog.? Below is the code used to display dog image.

This data set has below stats

There are 133 total dog categories.
There are 8351 total dog images.

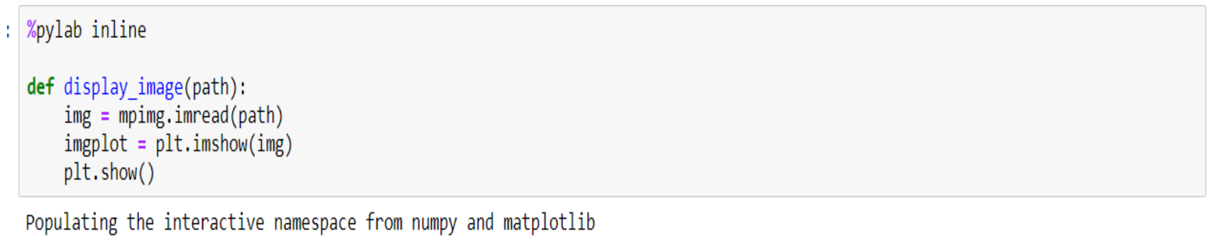
There are 6680 training dog images.
There are 835 validation dog images.
There are 836 test dog images.

https://en.wikipedia.org/wiki/Residual_neural_network



```
# define function to load train, test, and validation datasets
def load_dataset(path):
    data = load_files(path)
    dog_files = np.array(data['filenames'])
    dog_targets = np_utils.to_categorical(np.array(data['target']), 133)
    return dog_files, dog_targets

# load train, test, and validation datasets
train_files, train_targets = load_dataset('../../../data/dog_images/train')
valid_files, valid_targets = load_dataset('../../../data/dog_images/valid')
test_files, test_targets = load_dataset('../../../data/dog_images/test')

# load list of dog names
dog_names = [item[20:-1] for item in sorted(glob("../../../data/dog_images/train/*/"))]

# print statistics about the dataset
print('There are %d total dog categories.' % len(dog_names))
print('There are %s total dog images.\n' % len(np.hstack([train_files, valid_files, test_files])))
print('There are %d training dog images.' % len(train_files))
print('There are %d validation dog images.' % len(valid_files))
print('There are %d test dog images.'% len(test_files))
```

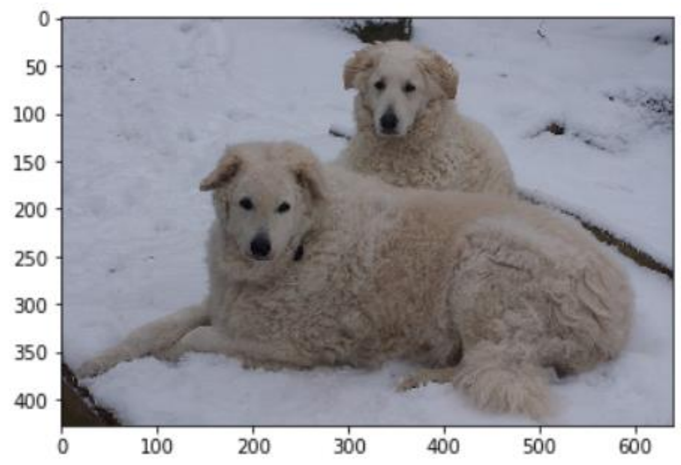
```
There are 133 total dog categories.
There are 8351 total dog images.

There are 6680 training dog images.
There are 835 validation dog images.
There are 836 test dog images.
```



```
%pylab inline

def display_image(path):
    img = mpimg.imread(path)
    imgplot = plt.imshow(img)
    plt.show()
```

```
Populating the interactive namespace from numpy and matplotlib
```

```
display_image(train_files[0])
```

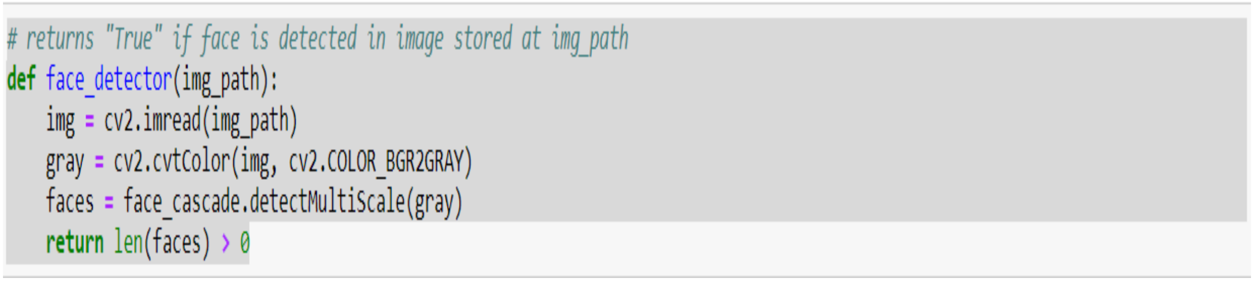


Human detector

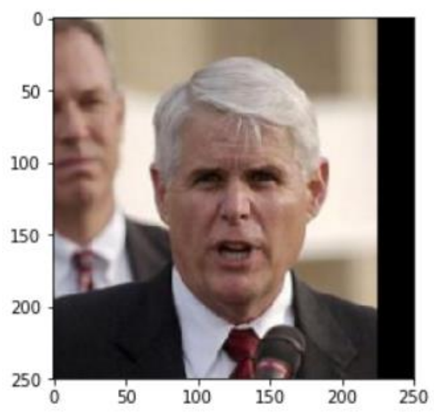
Identify in a dataset if image is human or not and find the accuracy.

I have used 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.

The following face detector function counts up how many human faces are in the photo:



```
# 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
```



Assess the Human Face Detector

- 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?

```
human_files_short = human_files[:100]
dog_files_short = train_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.

def get_percentage_of_human_faces(file_paths_arr):
    detections = [face_detector(file_path) for file_path in file_paths_arr]
    return (sum(detections)/len(detections))*100
```

```
human_files_percent = get_percentage_of_human_faces(human_files_short)
print(f"{human_files_percent}% of all the file paths from 'human_files_short' has human faces in them")
```

```
100.0% of all the file paths from 'human_files_short' has human faces in them
```

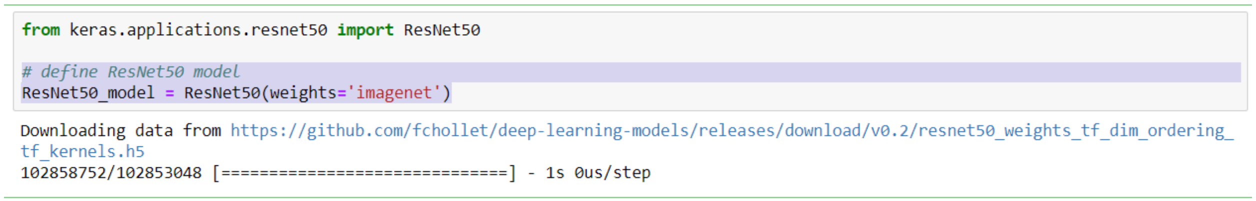
```
dog_files_percent = get_percentage_of_human_faces(dog_files_short)
print(f"{dog_files_percent}% of all the file paths from 'dog_files_short' has human faces in them")
```

```
11.0% of all the file paths from 'dog_files_short' has human faces in them
```

From above results we see data has 100% human faces in human files whereas 11% dog files has human faces in them

Detect Dogs

Here we used 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.



```
from keras.applications.resnet50 import ResNet50

# define ResNet50 model
ResNet50_model = ResNet50(weights='imagenet')
```

```
Downloading data from https://github.com/fchollet/deep-learning-models/releases/download/v0.2/resnet50_weights_tf_dim_ordering_
tf_kernels.h5
102858752/102853048 [==============================] - 1s 0us/step
```

Create tensor input from paths to images

The `path_to_tensor` the function 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 224×224 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 shape (1, 224, 224, 3).

The `paths_to_tensor` the function takes a NumPy array of string-valued image paths as input and returns a 4D tensor with shape (nbsamples, 224, 224, 3). Here, `nb_samples` is the number of samples, or a number of images, in the supplied array of image paths. It is best to think of nb_samples as the number of 3D tensors (where each 3D tensor corresponds to a different image).

In addition, ResNet-50 requires additional processing such as reordering of channels from RGB to BGR and normalization of pixels which is done using `preprocess_input`.

The model is then used to extract the predictions. The `predict` method, 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.

```
def ResNet50_predict_labels(img_path):
   # returns prediction vector for image located at img_path
   img = preprocess_input(path_to_tensor(img_path))
   return np.argmax(ResNet50_model.predict(img))
```

The categories corresponding to dogs appear in an uninterrupted sequence corresponding to keys 151–268, inclusive, to include all categories from `'Chihuahua'` to `'Mexican hairless'`. So, if the function returns any number between 151 to 268, the supplied image is that of a dog.

```
def dog_detector(img_path):
   prediction = ResNet50_predict_labels(img_path)
   return ((prediction <= 268) & (prediction >= 151))
```

The `dog_detector` function above, returns `True` if a dog is detected in an image (and `False` if not). None of the samples of human images have a detected dog as expected and all sample images of dogs have a detected dog as expected.

Create a CNN to Classify Dog Breeds

created a 4-layer CNN in Keras with Relu activation function. The model starts with an input image of 224 *224*3 color channels. This input image is big but very shallow, just R, G, and B. The convolution layers squeeze images by reducing width and height while increasing the depth layer by layer. By adding more filters, the network can learn more significant features in the photos and generalize better.

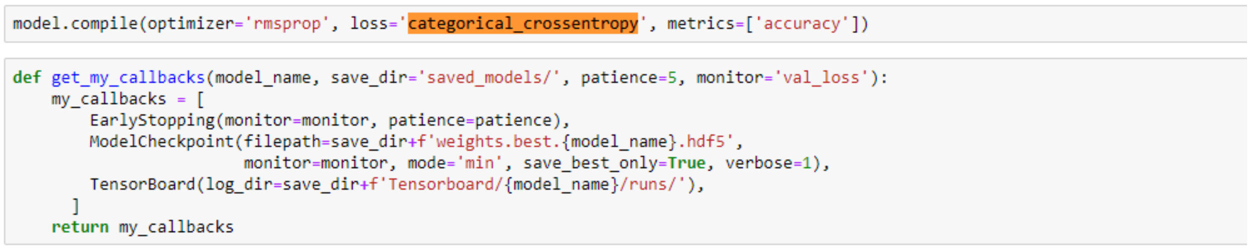
First layer produces an output with 16 feature channels that is used as an input for the next layer. The filters are the collection of 16 square matrices, output feature maps, which are weighted sums of input features and kernel. The kernel's weights are calculated during the training process by ImageNet data, and what it does is to slide across the input feature maps and produce output features. So, the shape of output features depends on the size of the kernel and input feature.

It would be ideal for input and output features to have the same size. So, I decided to use same padding to go off the edge of images with zero pads for all the layers with the stride of 2. I also used the max-pooling operation to ensure I am not losing information in the picture while lowering the chance of overfitting. Max pooling takes the maximum of pixels around a location.
After four Convolutional layers and max-pooling, followed by two fully connected layers, I trained the classifier. The Convolutional layers extract the image features, and the classifier classifies them based on the previously obtained features. The image below shows how the sequence of feature blocks and classifier on top transfer the information from the raw image and predict requested target values.

If we take a look at below mode code for first iteration we have 8 filter with shape of 224*224*3

model.add(Conv2D(filters=8, kernel_size=2, padding='same', activation='relu', input_shape=(224,224,3)))

Instead of implementing the code more than once created custom function as below.



```
model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])
```

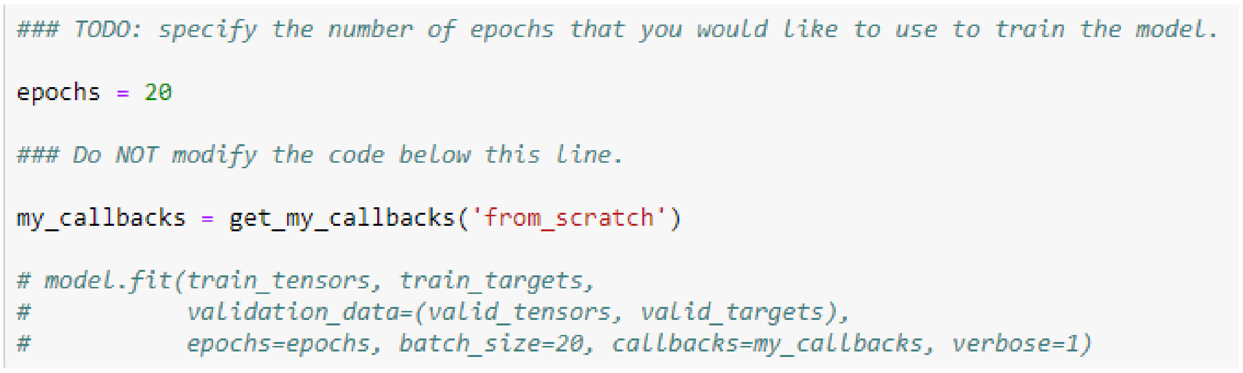
```
def get_my_callbacks(model_name, save_dir='saved_models/', patience=5, monitor='val_loss'):
    my_callbacks = [
        EarlyStopping(monitor=monitor, patience=patience),
        ModelCheckpoint(filepath=save_dir+f'weights.best.{model_name}.hdf5',
                        monitor=monitor, mode='min', save_best_only=True, verbose=1),
        TensorBoard(log_dir=save_dir+f'Tensorboard/{model_name}/runs/'),
      ]
    return my_callbacks
```

Code for dog classifier:

```
model = Sequential()

### TODO: Define your architecture.
model.add(Conv2D(filters=8, kernel_size=2, padding='same', activation='relu', input_shape=(224,224,3)))
model.add(MaxPooling2D(pool_size=2))
model.add(Conv2D(filters=16, kernel_size=2, padding='same', activation='relu'))
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(Conv2D(filters=256 , kernel_size=2, padding='same' , activation='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(2048, activation='relu'))
model.add(Dropout(0.35))
model.add(Dense(133,activation='softmax'))

model.summary()
```

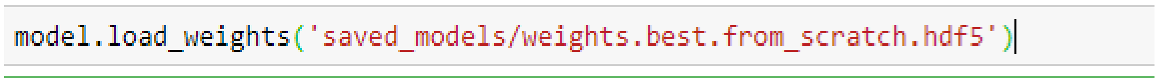
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_396 (Conv2D)          (None, 224, 224, 8)       104
_________________________________________________________________
max_pooling2d_37 (MaxPooling (None, 112, 112, 8)       0
_________________________________________________________________
conv2d_397 (Conv2D)          (None, 112, 112, 16)      528
_________________________________________________________________
max_pooling2d_38 (MaxPooling (None, 56, 56, 16)        0
_________________________________________________________________
conv2d_398 (Conv2D)          (None, 56, 56, 32)        2080
_________________________________________________________________
max_pooling2d_39 (MaxPooling (None, 28, 28, 32)        0
_________________________________________________________________
conv2d_399 (Conv2D)          (None, 28, 28, 64)        8256
_________________________________________________________________
max_pooling2d_40 (MaxPooling (None, 14, 14, 64)        0
_________________________________________________________________
conv2d_400 (Conv2D)          (None, 14, 14, 128)       32896
_________________________________________________________________
max_pooling2d_41 (MaxPooling (None, 7, 7, 128)         0
_________________________________________________________________
conv2d_401 (Conv2D)          (None, 7, 7, 256)         131328
_________________________________________________________________
max_pooling2d_42 (MaxPooling (None, 3, 3, 256)         0
_________________________________________________________________
dropout_5 (Dropout)          (None, 3, 3, 256)         0
_________________________________________________________________
flatten_5 (Flatten)          (None, 2304)              0
_________________________________________________________________
dense_8 (Dense)              (None, 2048)              4720640
```

Train Model:

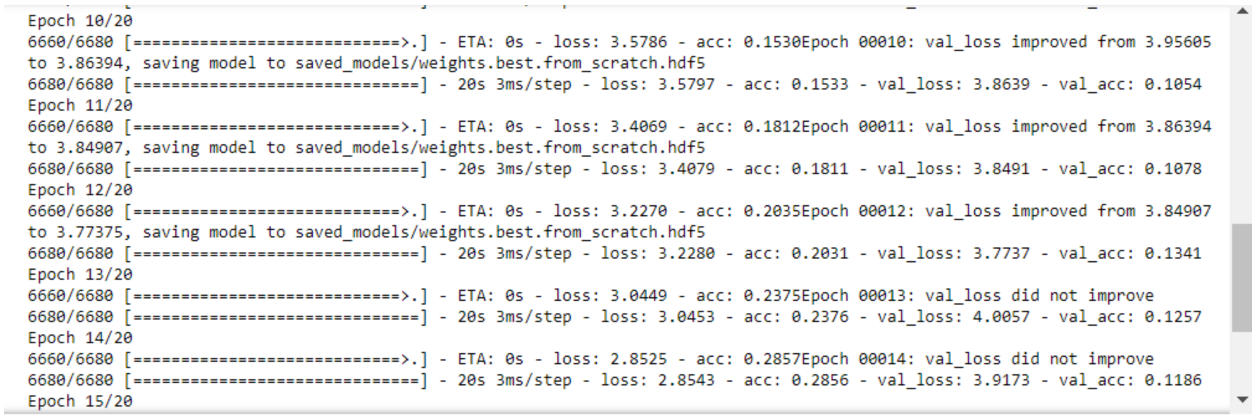


```
### TODO: specify the number of epochs that you would like to use to train the model.

epochs = 20

### Do NOT modify the code below this line.

my_callbacks = get_my_callbacks('from_scratch')

# model.fit(train_tensors, train_targets,
#           validation_data=(valid_tensors, valid_targets),
#           epochs=epochs, batch_size=20, callbacks=my_callbacks, verbose=1)
```

Above model training part was commented to make sure I don't end up training the model more than once however I have created 20 epocs for iterations with batch size of 20 , also implemented EarlyStopping to make sure model stops after 5 iteration if there is no improvement in the model



```
model.load_weights('saved_models/weights.best.from_scratch.hdf5')
```



```
Epoch 10/20
6660/6680 [============================>.] - ETA: 0s - loss: 3.5786 - acc: 0.1530Epoch 00010: val_loss improved from 3.95605
to 3.86394, saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 20s 3ms/step - loss: 3.5797 - acc: 0.1533 - val_loss: 3.8639 - val_acc: 0.1054
Epoch 11/20
6660/6680 [============================>.] - ETA: 0s - loss: 3.4069 - acc: 0.1812Epoch 00011: val_loss improved from 3.86394
to 3.84907, saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 20s 3ms/step - loss: 3.4079 - acc: 0.1811 - val_loss: 3.8491 - val_acc: 0.1078
Epoch 12/20
6660/6680 [============================>.] - ETA: 0s - loss: 3.2270 - acc: 0.2035Epoch 00012: val_loss improved from 3.84907
to 3.77375, saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 20s 3ms/step - loss: 3.2280 - acc: 0.2031 - val_loss: 3.7737 - val_acc: 0.1341
Epoch 13/20
6660/6680 [============================>.] - ETA: 0s - loss: 3.0449 - acc: 0.2375Epoch 00013: val_loss did not improve
6680/6680 [==============================] - 20s 3ms/step - loss: 3.0453 - acc: 0.2376 - val_loss: 4.0057 - val_acc: 0.1257
Epoch 14/20
6660/6680 [============================>.] - ETA: 0s - loss: 2.8525 - acc: 0.2857Epoch 00014: val_loss did not improve
6680/6680 [==============================] - 20s 3ms/step - loss: 2.8543 - acc: 0.2856 - val_loss: 3.9173 - val_acc: 0.1186
Epoch 15/20
```

Test Model:

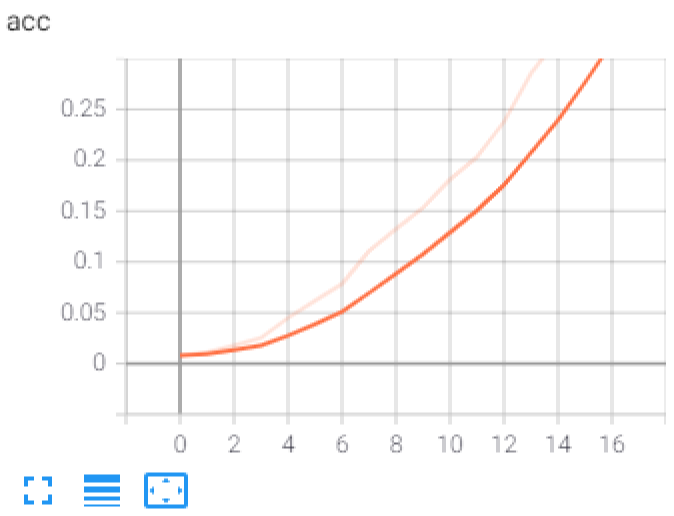
Implemented below functionality for accuracy of the mode and it was generated 12.5% accuracy.

```
# get index of predicted dog breed for each image in test set
dog_breed_predictions = [np.argmax(model.predict(np.expand_dims(tensor, axis=0))) for tensor in test_tensors]

# report test accuracy
test_accuracy = 100*np.sum(np.array(dog_breed_predictions)==np.argmax(test_targets, axis=1))/len(dog_breed_predictions)
print('Test accuracy: %.4f%%' % test_accuracy)
```

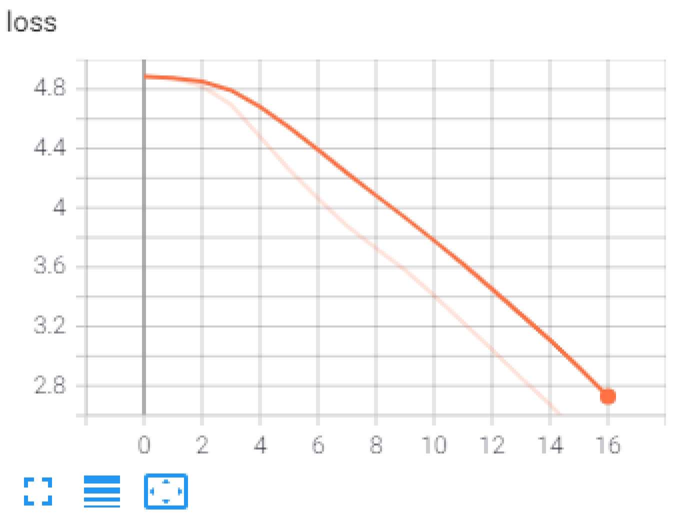
```
Test accuracy: 12.7990%
```

For reference I have capture tensor board loss accuracy

acc



oss



Use a CNN to Classify Dog Breeds

I have implemented VGG16 to demonstrate of transfer learning Bottleneck features is the concept of taking a pre-trained model and chopping off the top classifying layer, and then providing this “chopped” VGG16 as the first layer into our model.

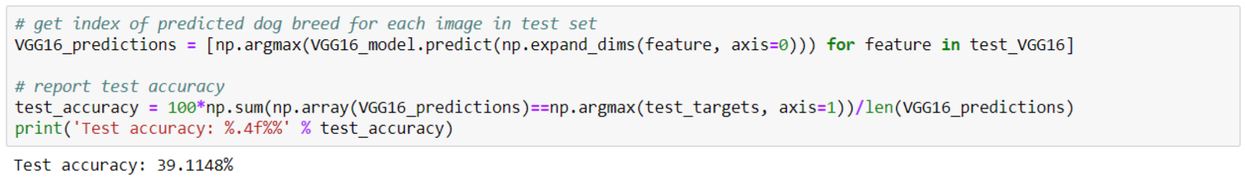
The pre-trained VGG-16 model was then used as a fixed feature extractor, where the last convolutional output of VGG-16 is fed as input to our model. VGG16 pre-trained has samples of 6680 unning this model for 20 epochs resulted in an increase in the accuracy of 39%

bottleneck_features = np.load('bottleneck_features/DogVGG16Data.npz')

train_VGG16 = bottleneck_features['train']

valid_VGG16 = bottleneck_features['valid']

test_VGG16 = bottleneck_features['test']



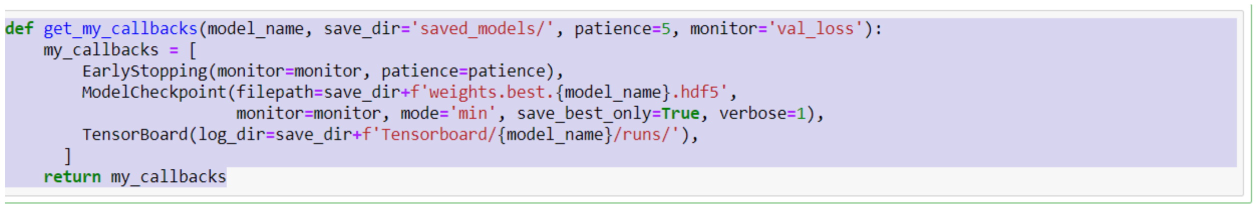
```
# get index of predicted dog breed for each image in test set
VGG16_predictions = [np.argmax(VGG16_model.predict(np.expand_dims(feature, axis=0))) for feature in test_VGG16]

# report test accuracy
test_accuracy = 100*np.sum(np.array(VGG16_predictions)==np.argmax(test_targets, axis=1))/len(VGG16_predictions)
print('Test accuracy: %.4f%%' % test_accuracy)
```

```
Test accuracy: 39.1148%
```

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

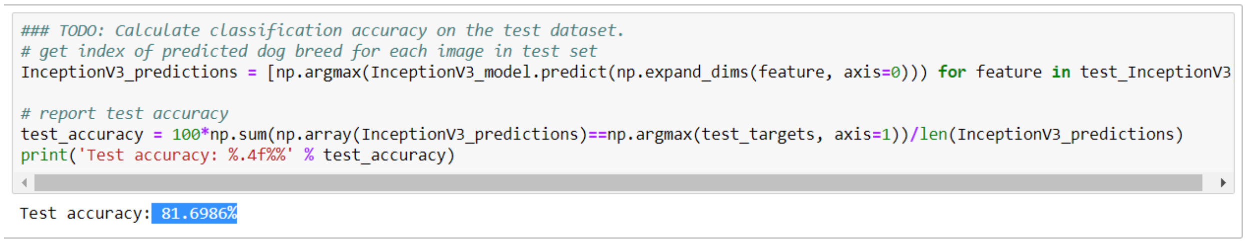
use transfer learning to create a CNN that can identify dog breed from images , in this I have used inception model to perform to identify dog breed

my_callbacks = get_my_callbacks('InceptionV3') using this custom function I am training the model on 6680 samples along with validation on 835 samples with early stopping in place



```
def get_my_callbacks(model_name, save_dir='saved_models/', patience=5, monitor='val_loss'):
    my_callbacks = [
        EarlyStopping(monitor=monitor, patience=patience),
        ModelCheckpoint(filepath=save_dir+f'weights.best.{model_name}.hdf5',
                        monitor=monitor, mode='min', save_best_only=True, verbose=1),
        TensorBoard(log_dir=save_dir+f'Tensorboard/{model_name}/runs/'),
      ]
    return my_callbacks
```

Once training is completed I have tested the model on target samples as show below and see accuracy of 81.6986%



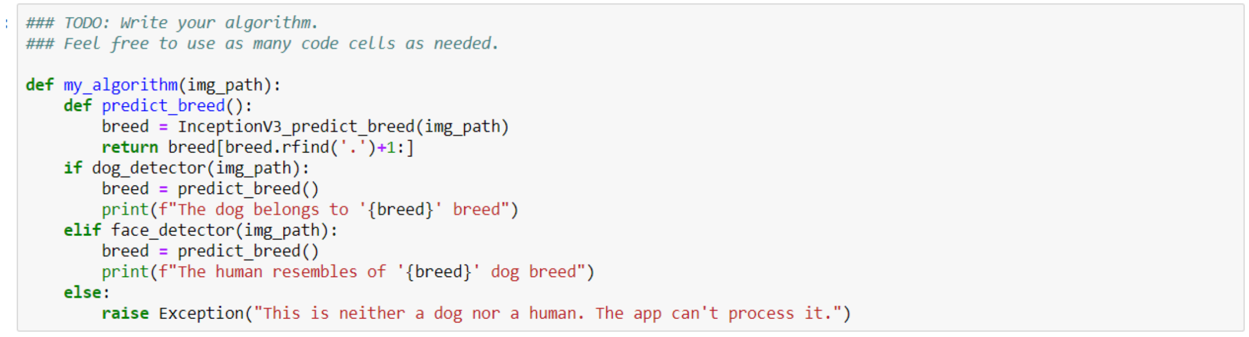
```
### TODO: Calculate classification accuracy on the test dataset.
# get index of predicted dog breed for each image in test set
InceptionV3_predictions = [np.argmax(InceptionV3_model.predict(np.expand_dims(feature, axis=0))) for feature in test_InceptionV3

# report test accuracy
test_accuracy = 100*np.sum(np.array(InceptionV3_predictions)==np.argmax(test_targets, axis=1))/len(InceptionV3_predictions)
print('Test accuracy: %.4f%%' % test_accuracy)
```

```
Test accuracy: 81.6986%
```

Write your Algorithm

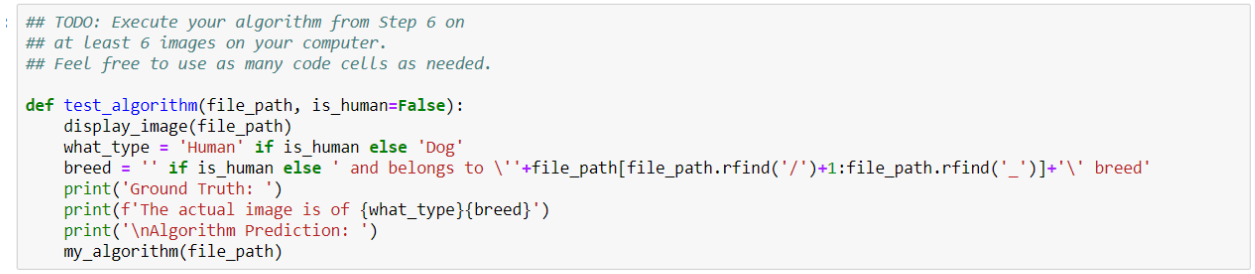
Write an algorithm that accepts a file path to an image and first determines whether the image contains a human, dog, or neither. Then,

- 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.



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

def my_algorithm(img_path):
    def predict_breed():
        breed = InceptionV3_predict_breed(img_path)
        return breed[breed.rfind('.')+1:]
    if dog_detector(img_path):
        breed = predict_breed()
        print(f"The dog belongs to '{breed}' breed")
    elif face_detector(img_path):
        breed = predict_breed()
        print(f"The human resembles of '{breed}' dog breed")
    else:
        raise Exception("This is neither a dog nor a human. The app can't process it.")
```

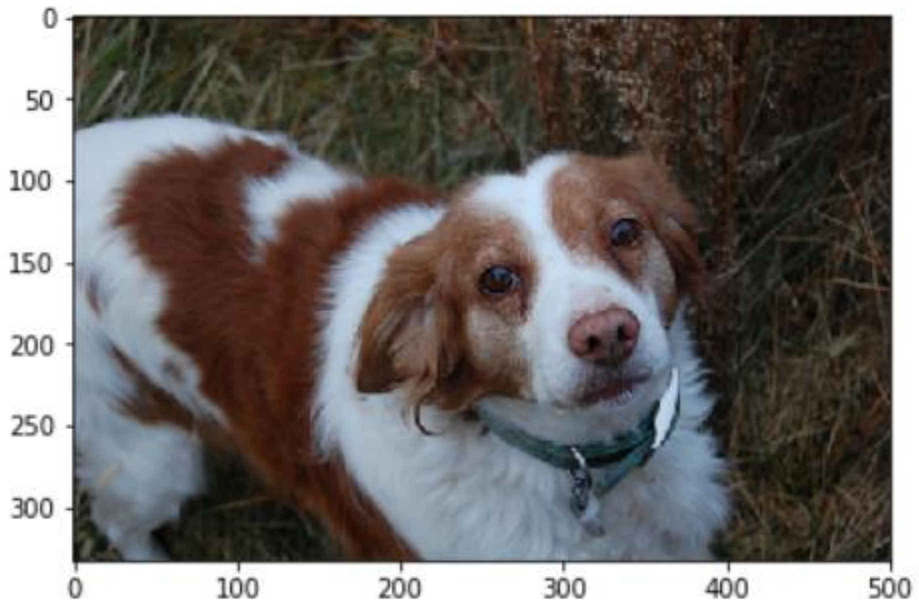
Test Algorithm



```
## TODO: Execute your algorithm from Step 6 on
## at least 6 images on your computer.
## Feel free to use as many code cells as needed.

def test_algorithm(file_path, is_human=False):
    display_image(file_path)
    what_type = 'Human' if is_human else 'Dog'
    breed = '' if is_human else ' and belongs to \''+file_path[file_path.rfind('/')+1:file_path.rfind('_')]+'\' breed'
    print('Ground Truth: ')
    print(f'The actual image is of {what_type}{breed}')
    print('\nAlgorithm Prediction: ')
    my_algorithm(file_path)
```

Testing algorithm with dog image to predict dog breed and I see its doing great job on prediction

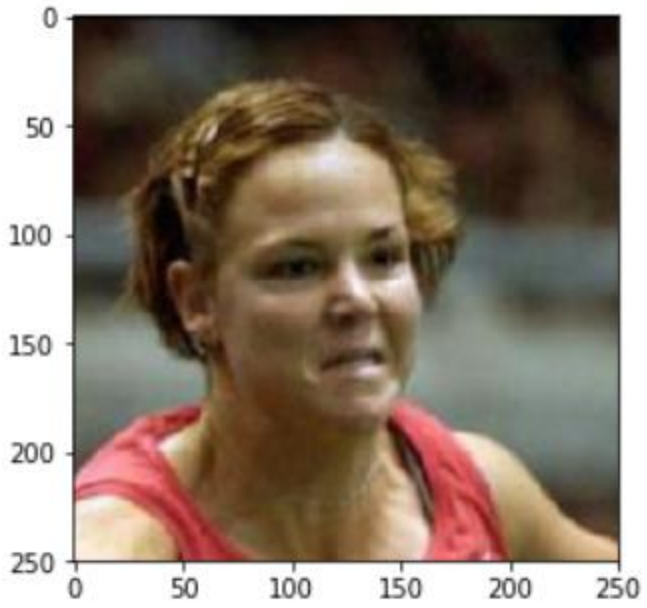
```
test_algorithm(test_files[89])
```



```
Ground Truth:
The actual image is of Dog and is of 'Brittany' breed

Algorithm Prediction:
The dog belongs to 'Brittany' breed
```

Test algorithm to predict human face resemblance

```
test_algorithm(human_files[89], is_human=True)
```

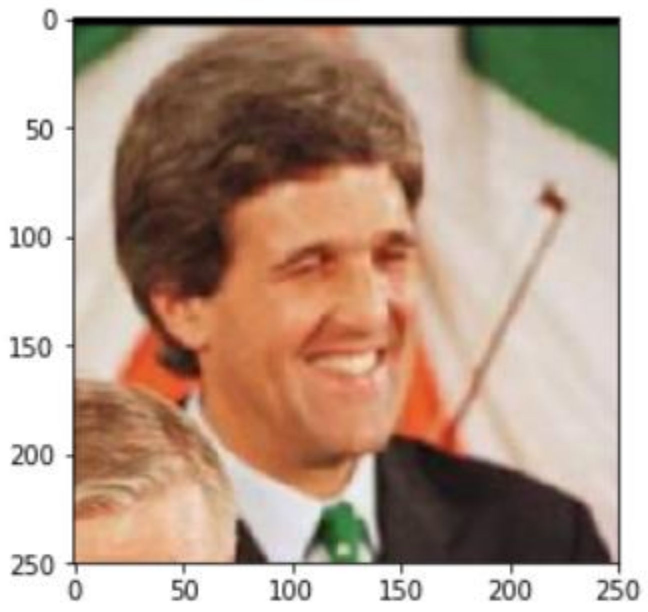


```
Ground Truth:
The actual image is of Human

Algorithm Prediction:
The human resembles of 'Canaan_dog' dog breed
```

```
test_algorithm(human_files[3], is_human=True)
```

```
Ground Truth:
The actual image is of Human

Algorithm Prediction:
The human resembles of 'Dachshund' dog breed
```

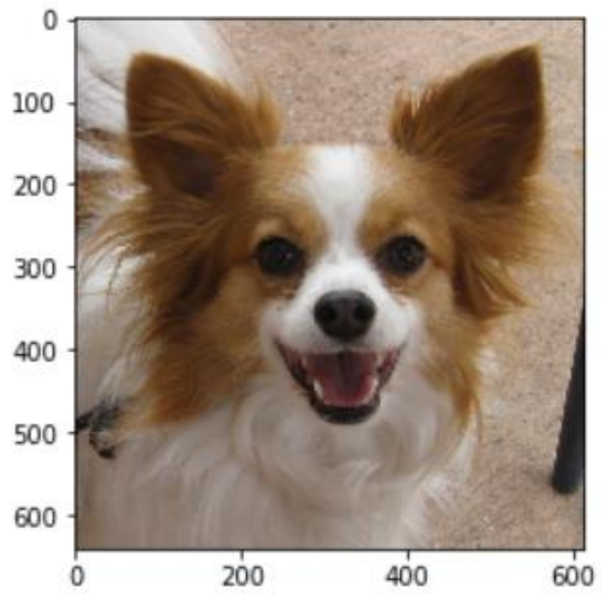
```
test_algorithm(human_files[50], is_human=True)
```



```
Ground Truth:
The actual image is of Human

Algorithm Prediction:
The human resembles of 'Dachshund' dog breed
```

Test on dog breed:

```
: test_algorithm(test_files[2])
```



```
Ground Truth:
The actual image is of Dog and belongs to 'Papillon' breed

Algorithm Prediction:
The dog belongs to 'Papillon' breed
```

Reflection

At the start, my objective was to create a CNN with **90%** testing accuracy. Our final model obtained **81%** testing accuracy.

There are a few breeds that are virtually identical and are sub-breeds. There's also a possibility of some images being either blurred or having too much noise. There's also a possibility of enhancing the quality by additional image manipulation.

Following the above areas, I’m sure we could increase the testing accuracy of the model to above 90%.

A simple web application in Flask could be built to leverage the model to predict breeds through user-input images.