Data Scientist Nanodegree

Convolutional Neural Networks

Project: Write an Algorithm for a Dog Identification App

This notebook walks you through one of the most popular Udacity projects across machine learning and artificial intellegence nanodegree programs. The goal is to classify images of dogs according to their breed.

If you are looking for a more guided capstone project related to deep learning and convolutional neural networks, this might be just it. Notice that even if you follow the notebook to creating your classifier, you must still create a blog post or deploy an application to fulfill the requirements of the capstone project.

Also notice, you may be able to use only parts of this notebook (for example certain coding portions or the data) without completing all parts and still meet all requirements of the capstone project.

In this notebook, some template code has already been provided for you, and you will need to implement additional functionality to successfully complete this project. You will not need to modify the included code beyond what is requested. Sections that begin with **'(IMPLEMENTATION)'** in the header indicate that the following block of code will require additional functionality which you must provide. Instructions will be provided for each section, and the specifics of the implementation are marked in the code block with a 'TODO' statement. Please be sure to read the instructions carefully!

In addition to implementing code, there will be questions that you must answer which relate to the project and your implementation. Each section where you will answer a question is preceded by a **'Question X'** header. Carefully read each question and provide thorough answers in the following text boxes that begin with **'Answer:'**. Your project submission will be evaluated based on your answers to each of the questions and the implementation you provide.

> **Note:** Code and Markdown cells can be executed using the **Shift + Enter** keyboard shortcut. Markdown cells can be edited by double-clicking the cell to enter edit mode.

The rubric contains *optional* "Stand Out Suggestions" for enhancing the project beyond the minimum requirements. If you decide to pursue the "Stand Out Suggestions", you should include the code in this IPython notebook.

Why We're Here

In this notebook, you will make the first steps towards developing an algorithm that could be used as part of a mobile or web app. At the end of this project, your code will accept any user-supplied image as input. If a dog is detected in the image, it will provide an estimate of the dog's breed. If a human is detected, it will provide an estimate of the dog breed that is most resembling. The image below displays potential sample output of your finished project (... but we expect that each student's algorithm will behave differently!).

Sample Dog Output

In this real-world setting, you will need to piece together a series of models to perform different tasks; for instance, the algorithm that detects humans in an image will be different from the CNN that infers dog breed. There are many points of possible failure, and no perfect algorithm exists. Your imperfect solution will nonetheless create a fun user experience!

The Road Ahead

We break the notebook into separate steps. Feel free to use the links below to navigate the notebook.

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

In [5]:
```
from sklearn.datasets import load_files
from keras.utils import np_utils
import numpy as np
from glob import glob
import urllib.request
# 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('dog_images/train')
valid_files, valid_targets = load_dataset('dog_images/valid')
test_files, test_targets = load_dataset('dog_images/test')

# load list of dog names
dog_names = [item[20:-1] for item in sorted(glob("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, te
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.
```

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` .

In [6]:

```
import random
random.seed(8675309)

# load filenames in shuffled human dataset
human_files = np.array(glob("lfw/*/*"))
random.shuffle(human_files)

# print statistics about the dataset
print('There are %d total human images.' % len(human_files))
```

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

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.

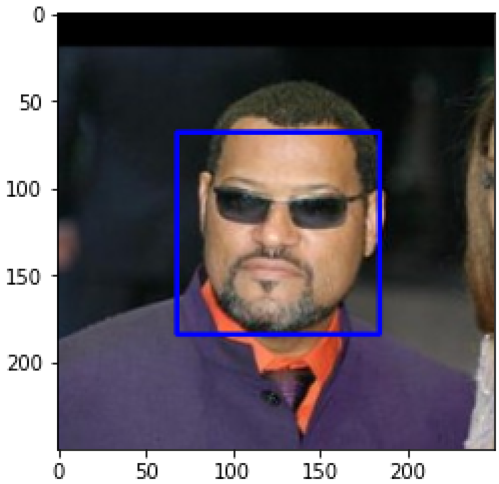
In the next code cell, we demonstrate how to use this detector to find human faces in a sample image.

In [8]:

```
import cv2
import matplotlib.pyplot as plt
%matplotlib inline
import urllib
# extract pre-trained face detector
face_cascade = cv2.CascadeClassifier('haarcascades/haarcascade_frontalface_alt.xml')
```

```
# load color (BGR) image
img = cv2.imread(human_files[3])
# 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
```



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.

In the above code, `faces` is a numpy array of detected faces, where each row corresponds to a detected face. Each detected face is a 1D array with four entries that specifies the bounding box of the detected face. The first two entries in the array (extracted in the above code as `x` and `y`) specify the horizontal and vertical positions of the top left corner of the bounding box. The last two entries in the array (extracted here as `w` and `h`) specify the width and height of the box.

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.

In [9]:

```
# returns "True" if face is detected in image stored at img_path
def face_detector(file_path):
    try:
```

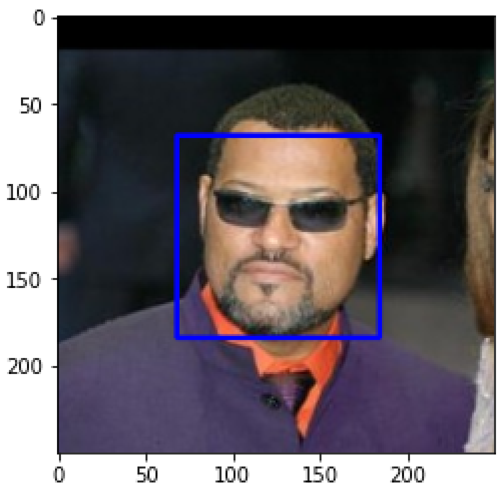
```
            req = urllib.request.urlopen(file_path)
            arr = np.asarray(bytearray(req.read()), dtype=np.uint8)
            img = cv2.imdecode(arr, -1)
        except:
            img = cv2.imread(file_path)
    # convert BGR image to grayscale

        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        faces = face_cascade.detectMultiScale(gray)
        return len(faces) > 0
```

In [18]:
```
face_detector('https://www.petihtiyac.com/labradoodle-765-blog.jpg')
```

Out[18]: False

In [20]:
```
face_cascade = cv2.CascadeClassifier('haarcascades/haarcascade_frontalface_alt.xml')


# load color (BGR) image
img = cv2.imread(human_files[3])
# 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()
face_detector(human_files[3])
```

Number of faces detected: 1



Out[20]: True

(IMPLEMENTATION) Assess the Human Face Detector

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?

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`.

Answer:

In [12]:

```
human_files_short = human_files[:100]
dog_files_short = train_files[:100]
# Do NOT modify the code above this line.
result_human= []
for i in range(len(human_files_short)):
    result_human.append(face_detector(human_files_short[i]))


result_dog= []
for i in range(len(dog_files_short)):
    result_dog.append(face_detector(dog_files_short[i]))

result_dog = np.array(result_dog)

result_human = np.array(result_human)
print("Dog results: "+str(result_dog.mean()*100)+ " %\n"  "Human results: "+str(result

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

```
Dog results: 12.0 %
Human results: 99.0 %
```

Question 2: This algorithmic choice necessitates that we communicate to the user that we accept human images only when they provide a clear view of a face (otherwise, we risk having unneccessarily frustrated users!). In your opinion, is this a reasonable expectation to pose on the user? If not, can you think of a way to detect humans in images that does not necessitate an image with a clearly presented face?

Answer:

We suggest the face detector from OpenCV as a potential way to detect human images in your algorithm, but you are free to explore other approaches, especially approaches that make use of deep learning :). Please use the code cell below to design and test your own face detection algorithm. If you decide to pursue this *optional* task, report performance on each of the datasets.

In [42]:

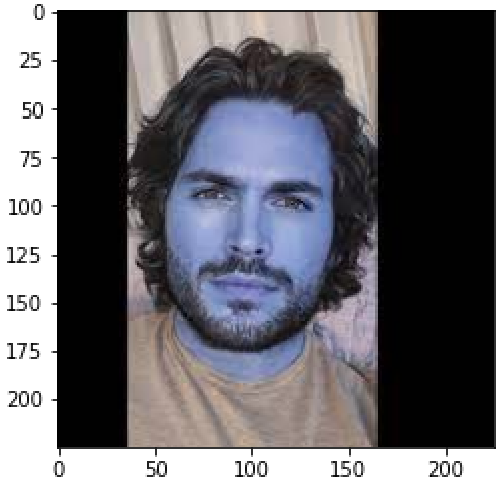
```
## (Optional) TODO: Report the performance of another
## face detection algorithm on the LFW dataset
### Feel free to use as many code cells as needed.

#human_files = np.array(glob("lfw/*/*"))
from PIL import Image

from deepface import DeepFace

def face_analyser(file_path):
#def face_analyser(file_path)
```

```
    try:
        req = urllib.request.urlopen(file_path)
        arr = np.asarray(bytearray(req.read()), dtype=np.uint8)
        img = cv2.imdecode(arr, -1)
    except:
        img = cv2.imread(file_path)

    plt.imshow(Image.fromarray(img))
    analysis = DeepFace.analyze(img_path = file_path, actions = ["age", "gender", "emo
    print(analysis)
```

In [44]:

```
face_analyser("data:image/jpeg;base64,/9j/4AAQSkZJRgABAQAAAQABAAD/2wCEAAoHCBYWFRgWFhUY
```

```
Action: age:   0%|          | 0/3 [00:00<?, ?it/s]
1/1 [==============================] - 1s 1s/step
Action: gender:  33%|███       | 1/3 [00:02<00:04,  2.16s/it]
1/1 [==============================] - 1s 1s/step
Action: emotion:  67%|███████   | 2/3 [00:03<00:01,  1.64s/it]
1/1 [==============================] - 0s 38ms/step
Action: emotion: 100%|██████████| 3/3 [00:03<00:00,  1.19s/it]
{'age': 24, 'region': {'x': 44, 'y': 52, 'w': 112, 'h': 112}, 'gender': 'Man', 'emotio
n': {'angry': 2.110096877526865, 'disgust': 0.0003673310385664207, 'fear': 0.999868594
0943473, 'happy': 4.214253406435657, 'sad': 1.8650189402722048, 'surprise': 0.04508921
0752787975, 'neutral': 90.76530823294023}, 'dominant_emotion': 'neutral'}
```



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.

In [150]:

```
from keras.applications.resnet50 import ResNet50

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

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

$$(\text{nb\_samples}, \text{rows}, \text{columns}, \text{channels}),$$

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 224×224 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` function takes a numpy array of string-valued image paths as input and returns a 4D tensor with shape

$$(\text{nb\_samples}, 224, 224, 3).$$

Here, `nb_samples` is the number of samples, or 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 your dataset!

In [151]:

```
from keras.preprocessing import image
from tqdm import tqdm

def path_to_tensor(img_path):
    try:
        with urllib.request.urlopen(img_path) as url:
            img = image.load_img(BytesIO(url.read()),target_size=(224, 224))

            img_array = np.expand_dims(image.img_to_array(img), axis=0)

        return img_array
    except:
        # loads RGB image as PIL.Image.Image type
        img = image.load_img(img_path, target_size=(224, 224))
        # convert PIL.Image.Image type to 3D tensor with shape (224, 224, 3)
        x = image.img_to_array(img)
    # convert 3D tensor to 4D tensor with shape (1, 224, 224, 3) and return 4D tensor
        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)
```

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` . If you're curious, you can check the code for `preprocess_input` here.

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.

In [152]:
```
from keras.applications.resnet50 import preprocess_input, decode_predictions

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))
```

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).

In [153]:
```
### 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))
```

In [154]:
```
face_detector(human_files[3])
```

Out[154]:
```
True
```

(IMPLEMENTATION) Assess the Dog Detector

Question 3: Use the code cell below to test the performance of your `dog_detector` function.

- What percentage of the images in `human_files_short` have a detected dog?
- What percentage of the images in `dog_files_short` have a detected dog?

Answer:

In [155]:
```
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.
result_human= []
```

```
for i in range(len(human_files_short)):
    result_human.append(dog_detector(human_files_short[i]))


result_dog= []
for i in range(len(dog_files_short)):
    result_dog.append(dog_detector(dog_files_short[i]))

result_dog = np.array(result_dog)

result_human = np.array(result_human)
print("Dog results: "+str(result_dog.mean()*100)+ " %\n"  "Human results: "+str(result
```

```
Dog results: 100.0 %
Human results: 0.0 %
```

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 that classifies dog breeds. You must create your CNN *from scratch* (so, you can't use transfer learning *yet*!), and you must attain a test accuracy of at least 1%. In Step 5 of this notebook, you will have the opportunity to use transfer learning to create a CNN that attains greatly improved accuracy.

Be careful with adding too many trainable layers! More parameters means longer training, which means you are more likely to need a GPU to accelerate the training process. Thankfully, Keras provides a handy estimate of the time that each epoch is likely to take; you can extrapolate this estimate to figure out how long it will take for your algorithm to train.

We mention that the task of assigning breed to dogs from images is considered exceptionally challenging. To see why, consider that *even a human* would have great difficulty in distinguishing between a Brittany and a Welsh Springer Spaniel.

Brittany	Welsh Springer Spaniel

It is not difficult to find other dog breed pairs with minimal inter-class variation (for instance, Curly-Coated Retrievers and American Water Spaniels).

Curly-Coated Retriever	American Water Spaniel

Likewise, recall that labradors come in yellow, chocolate, and black. Your vision-based algorithm will have to conquer this high intra-class variation to determine how to classify all of these different shades as the same breed.

Yellow Labrador	Chocolate Labrador	Black Labrador

We also mention that random chance presents an exceptionally low bar: setting aside the fact that the classes are slightly imabalanced, a random guess will provide a correct answer roughly 1 in 133 times, which corresponds to an accuracy of less than 1%.

Remember that the practice is far ahead of the theory in deep learning. Experiment with many different architectures, and trust your intuition. And, of course, have fun!

Pre-process the Data

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

In [ ]:
```
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

# pre-process the data for Keras
train_tensors = paths_to_tensor(train_files).astype('float32')/255
valid_tensors = paths_to_tensor(valid_files).astype('float32')/255
test_tensors = paths_to_tensor(test_files).astype('float32')/255
```

In [157]:
```
train_tensors.shape
```

Out[157]:
```
(6680, 224, 224, 3)
```

(IMPLEMENTATION) Model Architecture

Create a CNN to classify dog breed. At the end of your code cell block, summarize the layers of your model by executing the line:

```
model.summary()
```

We have imported some Python modules to get you started, but feel free to import as many modules as you need. If you end up getting stuck, here's a hint that specifies a model that trains relatively fast on CPU and attains >1% test accuracy in 5 epochs:

Sample CNN

Question 4: Outline the steps you took to get to your final CNN architecture and your reasoning at each step. If you chose to use the hinted architecture above, describe why you think that CNN architecture should work well for the image classification task.

Answer:

In [158]:
```
from keras.layers import Conv2D, MaxPooling2D, GlobalAveragePooling2D
from keras.layers import Dropout, Flatten, Dense, Activation
from keras.models import Sequential

import keras

from keras.layers import Conv2D, MaxPooling2D, GlobalAveragePooling2D, BatchNormalizat
from keras.layers import Dropout, Flatten, Dense, Input
from keras.models import Sequential

model = Sequential()
```

```
model.add(Conv2D(64, 3, activation="relu", padding="same",  input_shape=(224, 224, 3))
model.add(BatchNormalization())
model.add(Conv2D(64, 3, activation="relu"))
model.add(MaxPooling2D(pool_size=2))
model.add(Dropout(0.5))


model.add(Conv2D(64, 3, activation="relu", padding="same"))
model.add(BatchNormalization())
model.add(Conv2D(64, 3, activation="relu"))
model.add(MaxPooling2D(pool_size=2))
model.add(Dropout(0.5))

model.add(Conv2D(128, 3, activation="relu", padding="same"))
model.add(BatchNormalization())
model.add(Conv2D(128, 3, activation="relu"))
model.add(MaxPooling2D(pool_size=2))
model.add(Dropout(0.5))


model.add(Conv2D(128, (3, 3), padding='same', use_bias=False))
model.add(BatchNormalization(axis=3, scale=False))
model.add(Activation("relu"))
model.add(Flatten())


model.add(Dense(133,activation='softmax'))

model.summary()

### TODO: Define your architecture.
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_8 (Conv2D)            (None, 224, 224, 64)      1792
_________________________________________________________________
batch_normalization_5 (Batch (None, 224, 224, 64)      256
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 222, 222, 64)      36928
_________________________________________________________________
max_pooling2d_12 (MaxPooling (None, 111, 111, 64)      0
_________________________________________________________________
dropout_4 (Dropout)          (None, 111, 111, 64)      0
_________________________________________________________________
conv2d_10 (Conv2D)           (None, 111, 111, 64)      36928
_________________________________________________________________
batch_normalization_6 (Batch (None, 111, 111, 64)      256
_________________________________________________________________
conv2d_11 (Conv2D)           (None, 109, 109, 64)      36928
_________________________________________________________________
max_pooling2d_13 (MaxPooling (None, 54, 54, 64)        0
_________________________________________________________________
dropout_5 (Dropout)          (None, 54, 54, 64)        0
_________________________________________________________________
conv2d_12 (Conv2D)           (None, 54, 54, 128)       73856
_________________________________________________________________
batch_normalization_7 (Batch (None, 54, 54, 128)       512
_________________________________________________________________
conv2d_13 (Conv2D)           (None, 52, 52, 128)       147584
_________________________________________________________________
max_pooling2d_14 (MaxPooling (None, 26, 26, 128)       0
_________________________________________________________________
dropout_6 (Dropout)          (None, 26, 26, 128)       0
_________________________________________________________________
conv2d_14 (Conv2D)           (None, 26, 26, 128)       147456
```

```
batch_normalization_8 (Batch (None, 26, 26, 128)        384
_________________________________________________________________
activation_394 (Activation)  (None, 26, 26, 128)        0
_________________________________________________________________
flatten_5 (Flatten)          (None, 86528)              0
_________________________________________________________________
dense_5 (Dense)              (None, 133)                11508357
=================================================================
Total params: 11,991,237
Trainable params: 11,990,469
Non-trainable params: 768
_________________________________________________________________
```

Compile the Model

In [159]:
```
model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy
```

(IMPLEMENTATION) Train the Model

Train your model in the code cell below. Use model checkpointing to save the model that attains the best validation loss.

You are welcome to augment the training data, but this is not a requirement.

In [160]:
```
from keras.callbacks import ModelCheckpoint

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

epochs = 2

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

checkpointer = ModelCheckpoint(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=2, callbacks=[checkpointer], verbose=1)
```

```
Train on 6680 samples, validate on 835 samples
Epoch 1/2
6678/6680 [============================>.] - ETA: 0s - loss: 15.9679 - acc: 0.0090Epoc
h 00001: val_loss improved from inf to 15.96367, saving model to saved_models/weights.
best.from_scratch.hdf5
6680/6680 [==============================] - 396s 59ms/step - loss: 15.9679 - acc: 0.0
090 - val_loss: 15.9637 - val_acc: 0.0096
Epoch 2/2
6678/6680 [============================>.] - ETA: 0s - loss: 15.9685 - acc: 0.0093Epoc
h 00002: val_loss did not improve
6680/6680 [==============================] - 391s 59ms/step - loss: 15.9685 - acc: 0.0
093 - val_loss: 15.9637 - val_acc: 0.0096
```

Out[160]:
```
<keras.callbacks.History at 0x7fe883f5bcf8>
```

Load the Model with the Best Validation Loss

In [161]:
```
model.load_weights('saved_models/weights.best.from_scratch.hdf5')
```

Test the Model

Try out your model on the test dataset of dog images. Ensure that your test accuracy is greater than 1%.

In [162]:

```
# 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

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

```
Test accuracy: 0.9569%
```

Step 4: Use a CNN to Classify Dog Breeds

To reduce training time without sacrificing accuracy, we show you how to train a CNN using transfer learning. In the following step, you will get a chance to use transfer learning to train your own CNN.

Obtain Bottleneck Features

In [163]:

```
bottleneck_features = np.load('bottleneck_features/DogVGG16Data.npz')
train_VGG16 = bottleneck_features['train']
valid_VGG16 = bottleneck_features['valid']
test_VGG16 = bottleneck_features['test']
```

Model Architecture

The model uses the the pre-trained VGG-16 model as a fixed feature extractor, where the last convolutional output of VGG-16 is fed as input to our model. We only add a global average pooling layer and a fully connected layer, where the latter contains one node for each dog category and is equipped with a softmax.

In [164]:

```
VGG16_model = Sequential()
VGG16_model.add(GlobalAveragePooling2D(input_shape=train_VGG16.shape[1:]))
VGG16_model.add(Dense(133, activation='softmax'))

VGG16_model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
global_average_pooling2d_4 ( (None, 512)               0
_________________________________________________________________
dense_6 (Dense)              (None, 133)               68229
=================================================================
Total params: 68,229
Trainable params: 68,229
Non-trainable params: 0
_________________________________________________________________
```

Compile the Model

In [165]:

```
VGG16_model.compile(loss='categorical_crossentropy', optimizer='rmsprop', metrics=['ac
```

Train the Model

In [166]:
```
checkpointer = ModelCheckpoint(filepath='saved_models/weights.best.VGG16.hdf5',
                               verbose=1, save_best_only=True)

VGG16_model.fit(train_VGG16, train_targets,
          validation_data=(valid_VGG16, valid_targets),
          epochs=20, batch_size=20, callbacks=[checkpointer], verbose=1)
```

```
Train on 6680 samples, validate on 835 samples
Epoch 1/20
6560/6680 [============================>.] - ETA: 0s - loss: 12.4338 - acc: 0.1172Epoc
h 00001: val_loss improved from inf to 10.90914, saving model to saved_models/weights.
best.VGG16.hdf5
6680/6680 [==============================] - 5s 696us/step - loss: 12.4116 - acc: 0.11
86 - val_loss: 10.9091 - val_acc: 0.2084
Epoch 2/20
6580/6680 [============================>.] - ETA: 0s - loss: 10.2254 - acc: 0.2675Epoc
h 00002: val_loss improved from 10.90914 to 9.99919, saving model to saved_models/weig
hts.best.VGG16.hdf5
6680/6680 [==============================] - 2s 296us/step - loss: 10.2284 - acc: 0.26
77 - val_loss: 9.9992 - val_acc: 0.2910
Epoch 3/20
6600/6680 [============================>.] - ETA: 0s - loss: 9.6782 - acc: 0.3358Epoch
00003: val_loss improved from 9.99919 to 9.88144, saving model to saved_models/weight
s.best.VGG16.hdf5
6680/6680 [==============================] - 2s 293us/step - loss: 9.6670 - acc: 0.336
1 - val_loss: 9.8814 - val_acc: 0.3102
Epoch 4/20
6540/6680 [============================>.] - ETA: 0s - loss: 9.4724 - acc: 0.3667Epoch
00004: val_loss improved from 9.88144 to 9.76794, saving model to saved_models/weight
s.best.VGG16.hdf5
6680/6680 [==============================] - 2s 297us/step - loss: 9.4619 - acc: 0.367
1 - val_loss: 9.7679 - val_acc: 0.3174
Epoch 5/20
6560/6680 [============================>.] - ETA: 0s - loss: 9.2988 - acc: 0.3867Epoch
00005: val_loss improved from 9.76794 to 9.53296, saving model to saved_models/weight
s.best.VGG16.hdf5
6680/6680 [==============================] - 2s 296us/step - loss: 9.3106 - acc: 0.386
1 - val_loss: 9.5330 - val_acc: 0.3305
Epoch 6/20
6580/6680 [============================>.] - ETA: 0s - loss: 9.0057 - acc: 0.4076Epoch
00006: val_loss improved from 9.53296 to 9.25571, saving model to saved_models/weight
s.best.VGG16.hdf5
6680/6680 [==============================] - 2s 295us/step - loss: 9.0027 - acc: 0.407
8 - val_loss: 9.2557 - val_acc: 0.3485
Epoch 7/20
6540/6680 [============================>.] - ETA: 0s - loss: 8.8221 - acc: 0.4248Epoch
00007: val_loss improved from 9.25571 to 9.21353, saving model to saved_models/weight
s.best.VGG16.hdf5
6680/6680 [==============================] - 2s 297us/step - loss: 8.8321 - acc: 0.424
1 - val_loss: 9.2135 - val_acc: 0.3605
Epoch 8/20
6600/6680 [============================>.] - ETA: 0s - loss: 8.6217 - acc: 0.4368Epoch
00008: val_loss improved from 9.21353 to 9.07050, saving model to saved_models/weight
s.best.VGG16.hdf5
6680/6680 [==============================] - 2s 296us/step - loss: 8.6139 - acc: 0.437
4 - val_loss: 9.0705 - val_acc: 0.3689
Epoch 9/20
6520/6680 [============================>.] - ETA: 0s - loss: 8.5044 - acc: 0.4546Epoch
00009: val_loss improved from 9.07050 to 8.97947, saving model to saved_models/weight
s.best.VGG16.hdf5
6680/6680 [==============================] - 2s 300us/step - loss: 8.4997 - acc: 0.454
8 - val_loss: 8.9795 - val_acc: 0.3772
Epoch 10/20
6600/6680 [============================>.] - ETA: 0s - loss: 8.4476 - acc: 0.4630Epoch
00010: val_loss improved from 8.97947 to 8.95585, saving model to saved_models/weight
s.best.VGG16.hdf5
6680/6680 [==============================] - 2s 294us/step - loss: 8.4672 - acc: 0.462
0 - val_loss: 8.9559 - val_acc: 0.3784
Epoch 11/20
```

```
6600/6680 [============================>.] - ETA: 0s - loss: 8.4071 - acc: 0.4615Epoch
00011: val_loss did not improve
6680/6680 [==============================] - 2s 294us/step - loss: 8.3984 - acc: 0.462
1 - val_loss: 8.9600 - val_acc: 0.3701
Epoch 12/20
6600/6680 [============================>.] - ETA: 0s - loss: 8.2796 - acc: 0.4721Epoch
00012: val_loss improved from 8.95585 to 8.84979, saving model to saved_models/weight
s.best.VGG16.hdf5
6680/6680 [==============================] - 2s 293us/step - loss: 8.2804 - acc: 0.471
9 - val_loss: 8.8498 - val_acc: 0.3832
Epoch 13/20
6560/6680 [============================>.] - ETA: 0s - loss: 8.1477 - acc: 0.4832Epoch
00013: val_loss improved from 8.84979 to 8.65606, saving model to saved_models/weight
s.best.VGG16.hdf5
6680/6680 [==============================] - 2s 296us/step - loss: 8.1535 - acc: 0.482
9 - val_loss: 8.6561 - val_acc: 0.3892
Epoch 14/20
6620/6680 [============================>.] - ETA: 0s - loss: 8.0253 - acc: 0.4921Epoch
00014: val_loss improved from 8.65606 to 8.60660, saving model to saved_models/weight
s.best.VGG16.hdf5
6680/6680 [==============================] - 2s 293us/step - loss: 8.0411 - acc: 0.491
0 - val_loss: 8.6066 - val_acc: 0.4048
Epoch 15/20
6600/6680 [============================>.] - ETA: 0s - loss: 7.9590 - acc: 0.4947Epoch
00015: val_loss improved from 8.60660 to 8.51681, saving model to saved_models/weight
s.best.VGG16.hdf5
6680/6680 [==============================] - 2s 295us/step - loss: 7.9633 - acc: 0.494
0 - val_loss: 8.5168 - val_acc: 0.4024
Epoch 16/20
6580/6680 [============================>.] - ETA: 0s - loss: 7.8981 - acc: 0.5035Epoch
00016: val_loss improved from 8.51681 to 8.50448, saving model to saved_models/weight
s.best.VGG16.hdf5
6680/6680 [==============================] - 2s 295us/step - loss: 7.8998 - acc: 0.503
4 - val_loss: 8.5045 - val_acc: 0.4060
Epoch 17/20
6580/6680 [============================>.] - ETA: 0s - loss: 7.8549 - acc: 0.5055Epoch
00017: val_loss improved from 8.50448 to 8.47482, saving model to saved_models/weight
s.best.VGG16.hdf5
6680/6680 [==============================] - 2s 295us/step - loss: 7.8657 - acc: 0.504
6 - val_loss: 8.4748 - val_acc: 0.4060
Epoch 18/20
6560/6680 [============================>.] - ETA: 0s - loss: 7.7177 - acc: 0.5098Epoch
00018: val_loss improved from 8.47482 to 8.34424, saving model to saved_models/weight
s.best.VGG16.hdf5
6680/6680 [==============================] - 2s 297us/step - loss: 7.7361 - acc: 0.508
7 - val_loss: 8.3442 - val_acc: 0.4096
Epoch 19/20
6520/6680 [============================>.] - ETA: 0s - loss: 7.6389 - acc: 0.5167Epoch
00019: val_loss improved from 8.34424 to 8.19808, saving model to saved_models/weight
s.best.VGG16.hdf5
6680/6680 [==============================] - 2s 297us/step - loss: 7.6275 - acc: 0.517
7 - val_loss: 8.1981 - val_acc: 0.4347
Epoch 20/20
6560/6680 [============================>.] - ETA: 0s - loss: 7.4902 - acc: 0.5285Epoch
00020: val_loss did not improve
6680/6680 [==============================] - 2s 295us/step - loss: 7.5088 - acc: 0.527
1 - val_loss: 8.2411 - val_acc: 0.4263
```

Out[166]:

```
<keras.callbacks.History at 0x7fe85af70518>
```

Load the Model with the Best Validation Loss

In [167]:

```
VGG16_model.load_weights('saved_models/weights.best.VGG16.hdf5')
```

Test the Model

Now, we can use the CNN to test how well it identifies breed within our test dataset of dog images. We print the test accuracy below.

In [168]:
```
# 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))) f

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

```
Test accuracy: 42.2249%
```

Predict Dog Breed with the Model

In [169]:
```
from keras.preprocessing import image
from tqdm import tqdm

def path_to_tensor(img_path):
    # Loads RGB image as PIL.Image.Image type
    img = image.load_img(img_path, target_size=(224, 224))
    # convert PIL.Image.Image type to 3D tensor with shape (224, 224, 3)
    x = image.img_to_array(img)
    # convert 3D tensor to 4D tensor with shape (1, 224, 224, 3) and return 4D tensor
    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)
```

In [170]:
```
from keras.applications.vgg19 import VGG19
from keras.applications.vgg19 import preprocess_input as preprocess_input_vgg19
from keras.applications.resnet50 import ResNet50
from keras.applications.resnet50 import preprocess_input as preprocess_input_resnet50

def extract_VGG19(file_paths):
    tensors = paths_to_tensor(file_paths).astype('float32')
    preprocessed_input = preprocess_input_vgg19(tensors)
    return VGG19(weights='imagenet', include_top=False).predict(preprocessed_input, ba

def extract_Resnet50(file_paths):
    tensors = paths_to_tensor(file_paths).astype('float32')
    preprocessed_input = preprocess_input_resnet50(tensors)
    return ResNet50(weights='imagenet', include_top=False).predict(preprocessed_input,
```

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

You will now use transfer learning to create a CNN that can identify dog breed from images. Your CNN must attain at least 60% accuracy on the test set.

In Step 4, we used transfer learning to create a CNN using VGG-16 bottleneck features. In this section, you must use the bottleneck features from a different pre-trained model. To make things easier for you, we have pre-computed the features for all of the networks that are currently available in Keras:

- VGG-19 bottleneck features

- ResNet-50 bottleneck features
- Inception bottleneck features
- Xception bottleneck features

The files are encoded as such:

```
Dog{network}Data.npz
```

where `{network}`, in the above filename, can be one of `VGG19`, `Resnet50`, `InceptionV3`, or `Xception`. Pick one of the above architectures, download the corresponding bottleneck features, and store the downloaded file in the `bottleneck_features/` folder in the repository.

(IMPLEMENTATION) Obtain Bottleneck Features

In the code block below, extract the bottleneck features corresponding to the train, test, and validation sets by running the following:

```
bottleneck_features =
np.load('bottleneck_features/Dog{network}Data.npz')
train_{network} = bottleneck_features['train']
valid_{network} = bottleneck_features['valid']
test_{network} = bottleneck_features['test']
```

In [ ]:

```
train_vgg19 = extract_VGG19(train_files)
valid_vgg19 = extract_VGG19(valid_files)
test_vgg19 = extract_VGG19(test_files)
print("VGG19 shape", train_vgg19.shape[1:])

train_resnet50 = extract_Resnet50(train_files)
valid_resnet50 = extract_Resnet50(valid_files)
test_resnet50 = extract_Resnet50(test_files)
print("Resnet50 shape", train_resnet50.shape[1:])
```

(IMPLEMENTATION) Model Architecture

Create a CNN to classify dog breed. At the end of your code cell block, summarize the layers of your model by executing the line:

```
    <your model's name>.summary()
```

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:

In [172]:

```
### TODO: Define your architecture.
ResNet_50_model = Sequential()
ResNet_50_model.add(GlobalAveragePooling2D(input_shape=train_resnet50.shape[1:]))
ResNet_50_model.add(Dense(133, activation='softmax'))

ResNet_50_model.summary()
```

```
Layer (type)                 Output Shape              Param #   
=================================================================
global_average_pooling2d_5 ( (None, 2048)              0         
_________________________________________________________________
dense_7 (Dense)              (None, 133)               272517    
=================================================================
Total params: 272,517
Trainable params: 272,517
Non-trainable params: 0
_________________________________________________________________
```

In [173]:

```
### TODO: Define your architecture.
vgg19_model = Sequential()
vgg19_model.add(GlobalAveragePooling2D(input_shape=train_vgg19.shape[1:]))
vgg19_model.add(Dense(133, activation='softmax'))

vgg19_model.summary()
```

```
Layer (type)                 Output Shape              Param #   
=================================================================
global_average_pooling2d_6 ( (None, 512)               0         
_________________________________________________________________
dense_8 (Dense)              (None, 133)               68229     
=================================================================
Total params: 68,229
Trainable params: 68,229
Non-trainable params: 0
_________________________________________________________________
```

In [ ]:

(IMPLEMENTATION) Compile the Model

In [174]:

```
### TODO: Compile the model.
ResNet_50_model.compile(loss='categorical_crossentropy', optimizer='rmsprop', metrics=
```

In [175]:

```
### TODO: Compile the model.
vgg19_model.compile(loss='categorical_crossentropy', optimizer='rmsprop', metrics=['ac
```

(IMPLEMENTATION) Train the Model

Train your model in the code cell below. Use model checkpointing to save the model that attains the best validation loss.

You are welcome to augment the training data, but this is not a requirement.

In [176]:

```
### TODO: Train the model.
checkpointer = ModelCheckpoint(filepath='saved_models/weights.best.ResNet_50_model.hdf
                               verbose=1, save_best_only=True)

ResNet_50_model.fit(train_resnet50, train_targets,
          validation_data=(valid_resnet50, valid_targets),
          epochs=20, batch_size=20, callbacks=[checkpointer], verbose=1)
```

```
Train on 6680 samples, validate on 835 samples
Epoch 1/20
6520/6680 [============================>.] - ETA: 0s - loss: 1.5929 - acc: 0.6092Epoch
00001: val_loss improved from inf to 0.75907, saving model to saved_models/weights.bes
t.ResNet_50_model.hdf5
6680/6680 [==============================] - 6s 917us/step - loss: 1.5718 - acc: 0.613
5 - val_loss: 0.7591 - val_acc: 0.7689
Epoch 2/20
```

```
6640/6680 [============================>.] - ETA: 0s - loss: 0.4002 - acc: 0.8747Epoch
00002: val_loss improved from 0.75907 to 0.66057, saving model to saved_models/weight
s.best.ResNet_50_model.hdf5
6680/6680 [==============================] - 2s 301us/step - loss: 0.4001 - acc: 0.875
0 - val_loss: 0.6606 - val_acc: 0.7916
Epoch 3/20
6620/6680 [============================>.] - ETA: 0s - loss: 0.2421 - acc: 0.9225Epoch
00003: val_loss improved from 0.66057 to 0.64891, saving model to saved_models/weight
s.best.ResNet_50_model.hdf5
6680/6680 [==============================] - 2s 302us/step - loss: 0.2420 - acc: 0.922
6 - val_loss: 0.6489 - val_acc: 0.7844
Epoch 4/20
6640/6680 [============================>.] - ETA: 0s - loss: 0.1618 - acc: 0.9503Epoch
00004: val_loss improved from 0.64891 to 0.61197, saving model to saved_models/weight
s.best.ResNet_50_model.hdf5
6680/6680 [==============================] - 2s 301us/step - loss: 0.1613 - acc: 0.950
4 - val_loss: 0.6120 - val_acc: 0.8228
Epoch 5/20
6480/6680 [============================>.] - ETA: 0s - loss: 0.1131 - acc: 0.9640Epoch
00005: val_loss did not improve
6680/6680 [==============================] - 2s 296us/step - loss: 0.1132 - acc: 0.964
1 - val_loss: 0.6303 - val_acc: 0.8216
Epoch 6/20
6560/6680 [============================>.] - ETA: 0s - loss: 0.0790 - acc: 0.9750Epoch
00006: val_loss did not improve
6680/6680 [==============================] - 2s 299us/step - loss: 0.0795 - acc: 0.975
0 - val_loss: 0.7097 - val_acc: 0.8048
Epoch 7/20
6640/6680 [============================>.] - ETA: 0s - loss: 0.0601 - acc: 0.9813Epoch
00007: val_loss did not improve
6680/6680 [==============================] - 2s 300us/step - loss: 0.0603 - acc: 0.981
3 - val_loss: 0.6928 - val_acc: 0.8144
Epoch 8/20
6520/6680 [============================>.] - ETA: 0s - loss: 0.0446 - acc: 0.9865Epoch
00008: val_loss did not improve
6680/6680 [==============================] - 2s 302us/step - loss: 0.0446 - acc: 0.986
5 - val_loss: 0.6734 - val_acc: 0.8228
Epoch 9/20
6620/6680 [============================>.] - ETA: 0s - loss: 0.0342 - acc: 0.9897Epoch
00009: val_loss did not improve
6680/6680 [==============================] - 2s 298us/step - loss: 0.0345 - acc: 0.989
4 - val_loss: 0.6907 - val_acc: 0.8228
Epoch 10/20
6640/6680 [============================>.] - ETA: 0s - loss: 0.0260 - acc: 0.9926Epoch
00010: val_loss did not improve
6680/6680 [==============================] - 2s 299us/step - loss: 0.0266 - acc: 0.992
4 - val_loss: 0.6885 - val_acc: 0.8287
Epoch 11/20
6500/6680 [============================>.] - ETA: 0s - loss: 0.0191 - acc: 0.9955Epoch
00011: val_loss did not improve
6680/6680 [==============================] - 2s 296us/step - loss: 0.0208 - acc: 0.994
6 - val_loss: 0.7148 - val_acc: 0.8120
Epoch 12/20
6500/6680 [============================>.] - ETA: 0s - loss: 0.0137 - acc: 0.9965Epoch
00012: val_loss did not improve
6680/6680 [==============================] - 2s 298us/step - loss: 0.0144 - acc: 0.996
0 - val_loss: 0.7596 - val_acc: 0.8216
Epoch 13/20
6660/6680 [============================>.] - ETA: 0s - loss: 0.0135 - acc: 0.9964Epoch
00013: val_loss did not improve
6680/6680 [==============================] - 2s 298us/step - loss: 0.0134 - acc: 0.996
4 - val_loss: 0.7877 - val_acc: 0.8216
Epoch 14/20
6520/6680 [============================>.] - ETA: 0s - loss: 0.0105 - acc: 0.9975Epoch
00014: val_loss did not improve
6680/6680 [==============================] - 2s 297us/step - loss: 0.0105 - acc: 0.997
5 - val_loss: 0.7553 - val_acc: 0.8359
Epoch 15/20
6560/6680 [============================>.] - ETA: 0s - loss: 0.0092 - acc: 0.9974Epoch
```

```
00015: val_loss did not improve
6680/6680 [==============================] - 2s 296us/step - loss: 0.0093 - acc: 0.997
5 - val_loss: 0.7927 - val_acc: 0.8240
Epoch 16/20
6600/6680 [============================>.] - ETA: 0s - loss: 0.0092 - acc: 0.9971Epoch
00016: val_loss did not improve
6680/6680 [==============================] - 2s 299us/step - loss: 0.0091 - acc: 0.997
2 - val_loss: 0.8166 - val_acc: 0.8216
Epoch 17/20
6520/6680 [============================>.] - ETA: 0s - loss: 0.0074 - acc: 0.9979Epoch
00017: val_loss did not improve
6680/6680 [==============================] - 2s 297us/step - loss: 0.0074 - acc: 0.997
9 - val_loss: 0.7734 - val_acc: 0.8395
Epoch 18/20
6520/6680 [============================>.] - ETA: 0s - loss: 0.0063 - acc: 0.9983Epoch
00018: val_loss did not improve
6680/6680 [==============================] - 2s 301us/step - loss: 0.0061 - acc: 0.998
4 - val_loss: 0.7999 - val_acc: 0.8299
Epoch 19/20
6540/6680 [============================>.] - ETA: 0s - loss: 0.0046 - acc: 0.9986Epoch
00019: val_loss did not improve
6680/6680 [==============================] - 2s 298us/step - loss: 0.0047 - acc: 0.998
5 - val_loss: 0.8367 - val_acc: 0.8335
Epoch 20/20
6500/6680 [============================>.] - ETA: 0s - loss: 0.0040 - acc: 0.9988Epoch
00020: val_loss did not improve
6680/6680 [==============================] - 2s 297us/step - loss: 0.0052 - acc: 0.998
5 - val_loss: 0.9077 - val_acc: 0.8335
```

Out[176]: `<keras.callbacks.History at 0x7fe8d528c550>`

In [177]:

```
checkpointer = ModelCheckpoint(filepath='saved_models/weights.best.vgg19_model.hdf5',
                               verbose=1, save_best_only=True)

vgg19_model.fit(train_vgg19, train_targets,
          validation_data=(valid_vgg19, valid_targets),
          epochs=20, batch_size=20, callbacks=[checkpointer], verbose=1)
```

```
Train on 6680 samples, validate on 835 samples
Epoch 1/20
6500/6680 [============================>.] - ETA: 0s - loss: 10.0654 - acc: 0.1780Epoc
h 00001: val_loss improved from inf to 7.98468, saving model to saved_models/weights.b
est.vgg19_model.hdf5
6680/6680 [==============================] - 6s 892us/step - loss: 10.0295 - acc: 0.18
11 - val_loss: 7.9847 - val_acc: 0.3150
Epoch 2/20
6500/6680 [============================>.] - ETA: 0s - loss: 7.2220 - acc: 0.4188Epoch
00002: val_loss improved from 7.98468 to 7.40583, saving model to saved_models/weight
s.best.vgg19_model.hdf5
6680/6680 [==============================] - 2s 324us/step - loss: 7.2089 - acc: 0.419
8 - val_loss: 7.4058 - val_acc: 0.3964
Epoch 3/20
6500/6680 [============================>.] - ETA: 0s - loss: 6.7745 - acc: 0.5009Epoch
00003: val_loss improved from 7.40583 to 7.17786, saving model to saved_models/weight
s.best.vgg19_model.hdf5
6680/6680 [==============================] - 2s 323us/step - loss: 6.7625 - acc: 0.502
4 - val_loss: 7.1779 - val_acc: 0.4371
Epoch 4/20
6500/6680 [============================>.] - ETA: 0s - loss: 6.5654 - acc: 0.5351Epoch
00004: val_loss improved from 7.17786 to 7.01447, saving model to saved_models/weight
s.best.vgg19_model.hdf5
6680/6680 [==============================] - 2s 320us/step - loss: 6.5682 - acc: 0.535
2 - val_loss: 7.0145 - val_acc: 0.4467
Epoch 5/20
6500/6680 [============================>.] - ETA: 0s - loss: 6.3692 - acc: 0.5655Epoch
00005: val_loss improved from 7.01447 to 7.01282, saving model to saved_models/weight
s.best.vgg19_model.hdf5
6680/6680 [==============================] - 2s 324us/step - loss: 6.3598 - acc: 0.566
3 - val_loss: 7.0128 - val_acc: 0.4647
```

```
Epoch 6/20
6500/6680 [============================>.] - ETA: 0s - loss: 6.2804 - acc: 0.5846Epoch
00006: val_loss improved from 7.01282 to 6.92007, saving model to saved_models/weight
s.best.vgg19_model.hdf5
6680/6680 [==============================] - 2s 324us/step - loss: 6.2852 - acc: 0.584
0 - val_loss: 6.9201 - val_acc: 0.4778
Epoch 7/20
6500/6680 [============================>.] - ETA: 0s - loss: 6.1430 - acc: 0.5880Epoch
00007: val_loss improved from 6.92007 to 6.77286, saving model to saved_models/weight
s.best.vgg19_model.hdf5
6680/6680 [==============================] - 2s 324us/step - loss: 6.1470 - acc: 0.587
4 - val_loss: 6.7729 - val_acc: 0.4778
Epoch 8/20
6500/6680 [============================>.] - ETA: 0s - loss: 5.8976 - acc: 0.6085Epoch
00008: val_loss improved from 6.77286 to 6.63936, saving model to saved_models/weight
s.best.vgg19_model.hdf5
6680/6680 [==============================] - 2s 323us/step - loss: 5.8990 - acc: 0.608
2 - val_loss: 6.6394 - val_acc: 0.4886
Epoch 9/20
6500/6680 [============================>.] - ETA: 0s - loss: 5.8189 - acc: 0.6183Epoch
00009: val_loss improved from 6.63936 to 6.46272, saving model to saved_models/weight
s.best.vgg19_model.hdf5
6680/6680 [==============================] - 2s 323us/step - loss: 5.8166 - acc: 0.618
4 - val_loss: 6.4627 - val_acc: 0.5042
Epoch 10/20
6500/6680 [============================>.] - ETA: 0s - loss: 5.7478 - acc: 0.6297Epoch
00010: val_loss improved from 6.46272 to 6.43372, saving model to saved_models/weight
s.best.vgg19_model.hdf5
6680/6680 [==============================] - 2s 324us/step - loss: 5.7321 - acc: 0.630
4 - val_loss: 6.4337 - val_acc: 0.5138
Epoch 11/20
6500/6680 [============================>.] - ETA: 0s - loss: 5.7331 - acc: 0.6352Epoch
00011: val_loss did not improve
6680/6680 [==============================] - 2s 323us/step - loss: 5.7099 - acc: 0.636
5 - val_loss: 6.4939 - val_acc: 0.5102
Epoch 12/20
6500/6680 [============================>.] - ETA: 0s - loss: 5.6743 - acc: 0.6397Epoch
00012: val_loss improved from 6.43372 to 6.40624, saving model to saved_models/weight
s.best.vgg19_model.hdf5
6680/6680 [==============================] - 2s 322us/step - loss: 5.6908 - acc: 0.638
8 - val_loss: 6.4062 - val_acc: 0.5078
Epoch 13/20
6640/6680 [============================>.] - ETA: 0s - loss: 5.6280 - acc: 0.6435Epoch
00013: val_loss did not improve
6680/6680 [==============================] - 2s 320us/step - loss: 5.6346 - acc: 0.643
0 - val_loss: 6.4135 - val_acc: 0.5186
Epoch 14/20
6500/6680 [============================>.] - ETA: 0s - loss: 5.5821 - acc: 0.6452Epoch
00014: val_loss improved from 6.40624 to 6.38774, saving model to saved_models/weight
s.best.vgg19_model.hdf5
6680/6680 [==============================] - 2s 323us/step - loss: 5.5955 - acc: 0.644
2 - val_loss: 6.3877 - val_acc: 0.5066
Epoch 15/20
6500/6680 [============================>.] - ETA: 0s - loss: 5.4991 - acc: 0.6449Epoch
00015: val_loss improved from 6.38774 to 6.28030, saving model to saved_models/weight
s.best.vgg19_model.hdf5
6680/6680 [==============================] - 2s 322us/step - loss: 5.4786 - acc: 0.646
3 - val_loss: 6.2803 - val_acc: 0.5162
Epoch 16/20
6660/6680 [============================>.] - ETA: 0s - loss: 5.4178 - acc: 0.6547Epoch
00016: val_loss improved from 6.28030 to 6.22983, saving model to saved_models/weight
s.best.vgg19_model.hdf5
6680/6680 [==============================] - 2s 323us/step - loss: 5.4209 - acc: 0.654
5 - val_loss: 6.2298 - val_acc: 0.5293
Epoch 17/20
6640/6680 [============================>.] - ETA: 0s - loss: 5.4120 - acc: 0.6583Epoch
00017: val_loss improved from 6.22983 to 6.22134, saving model to saved_models/weight
s.best.vgg19_model.hdf5
6680/6680 [==============================] - 2s 325us/step - loss: 5.4044 - acc: 0.658
```

```
7 - val_loss: 6.2213 - val_acc: 0.5246
Epoch 18/20
6620/6680 [============================>.] - ETA: 0s - loss: 5.3893 - acc: 0.6601Epoch
00018: val_loss did not improve
6680/6680 [==============================] - 2s 324us/step - loss: 5.3892 - acc: 0.660
2 - val_loss: 6.3145 - val_acc: 0.5198
Epoch 19/20
6500/6680 [============================>.] - ETA: 0s - loss: 5.3793 - acc: 0.6623Epoch
00019: val_loss did not improve
6680/6680 [==============================] - 2s 320us/step - loss: 5.3746 - acc: 0.662
6 - val_loss: 6.2942 - val_acc: 0.5090
Epoch 20/20
6500/6680 [============================>.] - ETA: 0s - loss: 5.2712 - acc: 0.6588Epoch
00020: val_loss improved from 6.22134 to 6.17698, saving model to saved_models/weight
s.best.vgg19_model.hdf5
6680/6680 [==============================] - 2s 325us/step - loss: 5.2661 - acc: 0.659
3 - val_loss: 6.1770 - val_acc: 0.5257
```

Out[177]:
```
<keras.callbacks.History at 0x7fe8d528c2e8>
```

(IMPLEMENTATION) Load the Model with the Best Validation Loss

In [178]:
```
### TODO: Load the model weights with the best validation loss.
ResNet_50_model.load_weights('saved_models/weights.best.ResNet_50_model.hdf5')
```

In [179]:
```
### TODO: Load the model weights with the best validation loss.
vgg19_model.load_weights('saved_models/weights.best.vgg19_model.hdf5')
```

(IMPLEMENTATION) Test the Model

Try out your model on the test dataset of dog images. Ensure that your test accuracy is greater than 60%.

In [180]:
```
### TODO: Calculate classification accuracy on the test dataset.
# get index of predicted dog breed for each image in test set
ResNet_50_predictions = [np.argmax(ResNet_50_model.predict(np.expand_dims(feature, axi

# report test accuracy
test_accuracy = 100*np.sum(np.array(ResNet_50_predictions)==np.argmax(test_targets, ax
print('Test accuracy: %.4f%%' % test_accuracy)
```

```
Test accuracy: 80.9809%
```

In [181]:
```
### TODO: Calculate classification accuracy on the test dataset.
# get index of predicted dog breed for each image in test set
vgg19_predictions = [np.argmax(vgg19_model.predict(np.expand_dims(feature, axis=0))) f

# report test accuracy
test_accuracy = 100*np.sum(np.array(vgg19_predictions)==np.argmax(test_targets, axis=1
print('Test accuracy: %.4f%%' % test_accuracy)
```

```
Test accuracy: 55.5024%
```

(IMPLEMENTATION) Predict Dog Breed with the Model

Write a function that takes an image path as input and returns the dog breed (`Affenpinscher` , `Afghan_hound` , etc) that is predicted by your model.

Similar to the analogous function in Step 5, your function should have 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

```
extract_{network}
```

where `{network}` , in the above filename, should be one of `VGG19` , `Resnet50` , `InceptionV3` , or `Xception` .

In [224]:
```
def dog_detecter(file_path):
    def loadImage(file_path):
            with urllib.request.urlopen(file_path) as url:
                img = image.load_img(BytesIO(url.read()),target_size=(224, 224))
                img_array = np.expand_dims(image.img_to_array(img), axis=0)
            return img_array
    def ResNet50_predict_labels(file_path):
        # returns prediction vector for image located at img_path
        img = preprocess_input(loadImage(file_path))
        return np.argmax(ResNet50_model.predict(img))
    ### returns "True" if a dog is detected in the image stored at img_path
    def dog_detector(file_path):
        prediction = ResNet50_predict_labels(file_path)
        return ((prediction <= 268) & (prediction >= 151))
    return dog_detector(file_path)
```

In [233]:
```
### TODO: Write a function that takes a path to an image as input
### and returns the dog breed that is predicted by the model.
def dog_expert(file_path):
    def loadImage(file_path):
        with urllib.request.urlopen(file_path) as url:
            img = image.load_img(BytesIO(url.read()),target_size=(224, 224))

            img_array = np.expand_dims(image.img_to_array(img), axis=0)
        print (img_array.shape)
        return img_array


    def extract_Resnet50(file_path):
        tensors = loadImage(file_path).astype('float32')
        preprocessed_input = preprocess_input_resnet50(tensors)
        return ResNet50(weights='imagenet', include_top=False).predict(preprocessed_ir

    file_resnet50 = extract_Resnet50(file_path)

    ResNet_50_predictions = [np.argmax(ResNet_50_model.predict(np.expand_dims(feature,



    return dog_names[ResNet_50_predictions[0]].split(".")[1]
```

In [234]:
```
dog_detecter("https://petihtiyac.com/Data/EditorFiles/blog/kopeklerinsevmedigikokular1
```

Out[234]: True

Step 6: 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.

You are welcome to write your own functions for detecting humans and dogs in images, but feel free to use the `face_detector` and `dog_detector` functions developed above. You are **required** to use your CNN from Step 5 to predict dog breed.

A sample image and output for our algorithm is provided below, but feel free to design your own user experience!

Sample Human Output

This photo looks like an Afghan Hound.

(IMPLEMENTATION) Write your Algorithm

In [235]:
```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.
def dog_human_distinguisher(file_path):

    if face_detector(file_path):
        with urllib.request.urlopen(file_path) as url:
            img = image.load_img(BytesIO(url.read()),target_size=(224, 224))
            plt.figure(figsize=(10,10))

            plt.imshow(img)
            plt.xticks([])
            plt.yticks([])

            plt.show()
        print("This is a human image")

    elif dog_detecter(file_path):
        with urllib.request.urlopen(file_path) as url:
            img = image.load_img(BytesIO(url.read()),target_size=(224, 224))
            plt.figure(figsize=(10,10))

            plt.imshow(img)
            plt.xticks([])
            plt.yticks([])

            plt.show()

        print("This is a "+dog_expert(file_path)+" image")

    else:
        with urllib.request.urlopen(file_path) as url:
            img = image.load_img(BytesIO(url.read()),target_size=(224, 224))
```

```
            plt.figure(figsize=(10,10))

            plt.imshow(img)
            plt.xticks([])
            plt.yticks([])

            plt.show()
        print("This is something else")
```

In [236]:
```
dog_human_distinguisher("https://arkeofili.com/wp-content/uploads/2022/07/jwel1.jpg")
```

```
This is something else
```

Step 7: Test Your Algorithm

In this section, you will take your new algorithm for a spin! What kind of dog does the algorithm think that **you** look like? If you have a dog, does it predict your dog's breed accurately? If you have a cat, does it mistakenly think that your cat is a dog?

(IMPLEMENTATION) Test Your Algorithm on Sample Images!

Test your algorithm at least six images on your computer. Feel free to use any images you like. Use at least two human and two dog images.

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

Answer:

In [237]:
```
## 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.
dog_human_distinguisher("https://arkeofili.com/wp-content/uploads/2022/07/jwel1.jpg")
```

```
This is something else
```

In [238]:
```
dog_human_distinguisher("https://www.petihtiyac.com/labradoodle-765-blog.jpg")
```

```
(1, 224, 224, 3)
This is a Portuguese_water_dog image
```

In [239]:
```
dog_human_distinguisher("https://petihtiyac.com/Data/EditorFiles/blog/kopeklerinsevmec
```

```
(1, 224, 224, 3)
This is a Golden_retriever image
```

In [240]:
```
dog_human_distinguisher("https://arkeofili.com/wp-content/uploads/2017/05/naled3.jpg")
```

```
This is a human image
```

In [241]:
```
dog_human_distinguisher("https://arkeofili.com/wp-content/uploads/2022/07/jwel7.jpg")
```

```
This is something else
```

In [242]:
```
dog_human_distinguisher("https://www.petihtiyac.com/Data/Blog/3/336.jpg")
```

```
This is a human image
```

In [243]:

```
dog_human_distinguisher("https://www.petihtiyac.com/Data/Blog/17.jpg")
```

```
(1, 224, 224, 3)
This is a Beagle image
```

In [244]:
```
dog_human_distinguisher("https://www.petihtiyac.com/Data/Blog/15.png")
```

```
(1, 224, 224, 3)
This is a German_shepherd_dog image
```

In [245]:
```
dog_human_distinguisher("https://www.petihtiyac.com/Data/Blog/34.jpg")
```

```
(1, 224, 224, 3)
This is a American_staffordshire_terrier image
```

In [ ]: