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

```
from sklearn.datasets import load_files
import numpy as np
from glob import glob
import shutil
import random
import cv2
import matplotlib.pyplot as plt
```

```
import json
import os
from extract_bottleneck_features import *
from PIL import ImageFile
from tqdm import tqdm

from keras.utils import to_categorical
from tensorflow.keras.preprocessing import image
from keras.applications.resnet50 import preprocess_input,
decode_predictions
from keras.layers import Conv2D, MaxPooling2D, GlobalAveragePooling2D
from keras.layers import Dropout, Flatten, Dense
from keras.models import Sequential
from keras.optimizers import Adam
from keras.callbacks import ModelCheckpoint, ReduceLROnPlateau
from keras.applications.resnet50 import ResNet50

ImageFile.LOAD_TRUNCATED_IMAGES = True  # Allow loading of truncated
images
%matplotlib inline
random.seed(8675309)

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

Please download the file Udacity-Capstone-Data.zip containing all data that this project used and place them in their corresponding position:

- Two subfolders "dog_images" and "lfw" in the folder "Udacity-Capstone-Data/data" should be moved into "Udacity-Capstone-Project/data"
- Two files "DogResnet50Data.npz" and "DogVGG16Data.npz" in the folder "Udacity-Capstone-Data" should be moved into "Udacity-Capstone-Project/bottleneck_features"

```
# shutil.copytree('../../../data/dog_images', "data/dog_images")

# 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(f'There are {len(dog_names)} total dog categories.')
```

```
print(f'There are {len(train_files) + len(valid_files) +
len(test_files)} total dog images.')
print(f'There are {len(train_files)} training dog images.')
print(f'There are {len(valid_files)} validation dog images.')
print(f'There are {len(test_files)} test dog images.')
```

```
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`.

```
# shutil.copytree('../../../data/lfw', "data/lfw")

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

# Print statistics about the human dataset
print(f'There are {len(human_files)} total human images.')
```

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

Step 1: Detect Humans

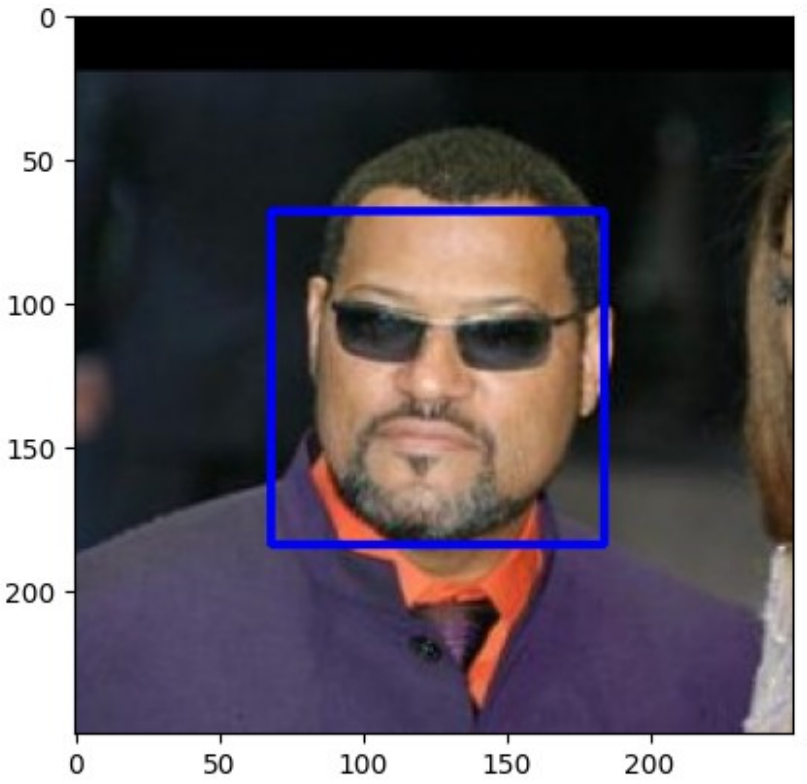
We use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images. OpenCV provides many pre-trained face detectors, stored as XML files on github. We have downloaded one of these detectors and stored it in the `haarcascades` directory.

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

```
# 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
```




```
def plot_an_image(img_path):
    """
    Display an image from a specified file path using matplotlib, with
```

```
color correction from BGR to RGB.

    This function loads an image using OpenCV, converts it from
OpenCV's default BGR color format to
    RGB, which is more suitable for displaying with matplotlib. The
function then displays the image in
    an interactive window.

    Parameters:
    img_path (str): The file path to the image that needs to be
displayed.

    Returns:
    None

    Usage:
    >>> plot_an_image('path/to/your/image.jpg')
    """
    img = cv2.imread(img_path)
    cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)  # Convert BGR image
to RGB for plotting
    plt.imshow(cv_rgb)
    plt.show()
```

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.

```
# 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
```

(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:

```
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.
is_human_human = [face_detector(img) for img in human_files_short]
perc_human_human = 100 * np.mean(is_human_human)
print("{:.0f}% of the first 100 images in human_files have a detected
human face"
          .format(perc_human_human))

is_dog_human = [face_detector(img) for img in dog_files_short]
perc_dog_human = 100 * np.mean(is_dog_human)
print("{:.0f}% of the first 100 images in dog_files have a detected
human face"
          .format(perc_dog_human))
```

```
99% of the first 100 images in human_files have a detected human face
12% of the first 100 images in dog_files have a detected human face
```

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.

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

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.

```
# 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!

```
def path_to_tensor(img_path):
    try:
        img = image.load_img(img_path, target_size=(224, 224))
        x = image.img_to_array(img)
        # Normalize the image tensor
        return np.expand_dims(x, axis=0).astype('float32')/255
    except IOError:
        print(f"Warning: Skipping corrupted image {img_path}")
        return None

def paths_to_tensor(img_paths):
    batch_tensors = []
    for img_path in img_paths:
        tensor = path_to_tensor(img_path)
        if tensor is not None:
            batch_tensors.append(tensor[0])
    return np.array(batch_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.

```
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,verbose=0))

# Customize the ResNet50_predict_labels function
# Load the pre-trained ResNet50 model
model = ResNet50(weights='imagenet')

# Assuming you have loaded your custom labels dictionary as follows:
with open('label_id_mapping.json','r') as f:
```

```
    imagenet1000_clsidx_to_labels = json.load(f)

def ResNet50_predict_1000_labels(img_path):
    # Load and preprocess the image
    img = image.load_img(img_path, target_size=(224, 224))
    x = image.img_to_array(img)
    x = np.expand_dims(x, axis=0)
    x = preprocess_input(x)

    # Make predictions
    preds = model.predict(x,verbose=0)

    # Decode the predictions
    def decode_predictions_custom(preds, top=1,
class_list=imagenet1000_clsidx_to_labels):
        class_ids = np.argmax(preds, axis=-1)
        results = []
        for id in class_ids:
            results.append((id, class_list[str(id)]))
        return results

    # Use the custom decode function
    decoded_predictions = decode_predictions_custom(preds, top=5)

    return decoded_predictions[0][0]
```

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).

```
### 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)
    prediction = ResNet50_predict_1000_labels(img_path)
    return ((prediction <= 268) & (prediction >= 151))
```

(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:

```
### TODO: Test the performance of the dog_detector function

### on the images in human_files_short and dog_files_short.
is_human_dog = [dog_detector(img) for img in human_files_short]
perc_human_human = 100 * np.mean(is_human_dog)
print("{:.0f}% of the first 100 images in human_files have a detected
dog"
          .format(perc_human_human))

is_dog_dog = [dog_detector(img) for img in dog_files_short]
perc_dog_human = 100 * np.mean(is_dog_dog)
print("{:.0f}% of the first 100 images in dog_files have a detected
dog"
          .format(perc_dog_human))
```

```
0% of the first 100 images in human_files have a detected dog
100% of the first 100 images in dog_files have a detected dog
```

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

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.

```
def image_generator(files, targets, batch_size):
    while True:
        batch_paths = np.random.choice(a=files, size=batch_size)
        batch_input = paths_to_tensor(batch_paths)
        valid_paths = [p for p in batch_paths if path_to_tensor(p) is
not None]
        batch_indices = [np.where(files == img_path)[0][0] for
img_path in valid_paths]
        batch_output = np.array([targets[index] for index in
batch_indices])

        if len(batch_input) > 0:  # Ensure there is data to yield
            yield batch_input, batch_output

# Create generators for train, validation, and test datasets
train_generator = image_generator(train_files, train_targets,
batch_size=64)
valid_generator = image_generator(valid_files, valid_targets,
batch_size=64)
test_generator = image_generator(test_files, test_targets,
batch_size=64)
```

(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:

```
def sample_CNN_model():
    return Sequential([
    # First Convolutional Layer
    Conv2D(16, (2, 2), activation='relu', input_shape=(223, 223, 3)),
    MaxPooling2D(pool_size=(2, 2)),

    # Second Convolutional Layer
    Conv2D(32, (2, 2), activation='relu'),
    MaxPooling2D(pool_size=(2, 2)),

    # Third Convolutional Layer
    Conv2D(64, (2, 2), activation='relu'),
    MaxPooling2D(pool_size=(2, 2)),

    # Global Average Pooling Layer
    GlobalAveragePooling2D(),

    # Fully Connected Layer with Softmax Activation to classify into
133 classes
    Dense(133, activation='softmax')
])

# Define the model
model = sample_CNN_model()
# Print the model summary to check the architecture
model.summary()
```

```
c:\Users\User\anaconda3\envs\udacity3\Lib\site-packages\keras\src\
layers\convolutional\base_conv.py:107: UserWarning: Do not pass an
`input_shape`/`input_dim` argument to a layer. When using Sequential
models, prefer using an `Input(shape)` object as the first layer in
the model instead.
  super().__init__(activity_regularizer=activity_regularizer,
**kwargs)

Model: "sequential_4"
```

```
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓
┃ Layer (type)                    ┃ Output Shape           ┃       Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩
│ conv2d_3 (Conv2D)               │ (None, 222, 222, 16)   │           208 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ max_pooling2d_3 (MaxPooling2D)  │ (None, 111, 111, 16)   │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ conv2d_4 (Conv2D)               │ (None, 110, 110, 32)   │         2,080 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ max_pooling2d_4 (MaxPooling2D)  │ (None, 55, 55, 32)     │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ conv2d_5 (Conv2D)               │ (None, 54, 54, 64)     │         8,256 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ max_pooling2d_5 (MaxPooling2D)  │ (None, 27, 27, 64)     │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ global_average_pooling2d_2      │ (None, 64)             │             0 │
│ (GlobalAveragePooling2D)        │                        │               │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_2 (Dense)                 │ (None, 133)            │         8,645 │
└─────────────────────────────────┴────────────────────────┴───────────────┘

 Total params: 19,189 (74.96 KB)

 Trainable params: 19,189 (74.96 KB)

 Non-trainable params: 0 (0.00 B)
```

Compile the Model

```
# Set a smaller learning rate
adam = Adam(learning_rate=0.001)
model.compile(optimizer=adam, 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.

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

epochs = 10

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

# Add checkpoint to save the best model
checkpointer =
ModelCheckpoint(filepath='saved_models/weights.best.from_scratch.keras
',
                               verbose=1, save_best_only=True)

# ReduceLROnPlateau: This callback reduces the learning rate when a
metric has stopped improving.
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.1,
patience=5)

# Train the model
model.fit(train_generator,
                    steps_per_epoch=len(train_files) // 32,
                    validation_data=valid_generator,
                    validation_steps=len(valid_files) // 32,
                    epochs=epochs,
                    callbacks=[checkpointer, reduce_lr],
                    verbose=2)

Epoch 1/10

Epoch 1: val_loss improved from inf to 4.86801, saving model to
saved_models/weights.best.from_scratch.keras
208/208 - 408s - 2s/step - accuracy: 0.0116 - loss: 4.8749 -
val_accuracy: 0.0114 - val_loss: 4.8680 - learning_rate: 0.0010
Epoch 2/10

Epoch 2: val_loss improved from 4.86801 to 4.79703, saving model to
saved_models/weights.best.from_scratch.keras
```

```
208/208 - 436s - 2s/step - accuracy: 0.0167 - loss: 4.8166 -
val_accuracy: 0.0216 - val_loss: 4.7970 - learning_rate: 0.0010
Epoch 3/10

Epoch 3: val_loss improved from 4.79703 to 4.77886, saving model to
saved_models/weights.best.from_scratch.keras
208/208 - 373s - 2s/step - accuracy: 0.0220 - loss: 4.7489 -
val_accuracy: 0.0234 - val_loss: 4.7789 - learning_rate: 0.0010
Epoch 4/10

Epoch 4: val_loss improved from 4.77886 to 4.76772, saving model to
saved_models/weights.best.from_scratch.keras
208/208 - 309s - 1s/step - accuracy: 0.0230 - loss: 4.7092 -
val_accuracy: 0.0246 - val_loss: 4.7677 - learning_rate: 0.0010
Epoch 5/10

Epoch 5: val_loss improved from 4.76772 to 4.73736, saving model to
saved_models/weights.best.from_scratch.keras
208/208 - 295s - 1s/step - accuracy: 0.0270 - loss: 4.6909 -
val_accuracy: 0.0168 - val_loss: 4.7374 - learning_rate: 0.0010
Epoch 6/10

Epoch 6: val_loss improved from 4.73736 to 4.72983, saving model to
saved_models/weights.best.from_scratch.keras
208/208 - 286s - 1s/step - accuracy: 0.0312 - loss: 4.6593 -
val_accuracy: 0.0192 - val_loss: 4.7298 - learning_rate: 0.0010
Epoch 7/10

Epoch 7: val_loss improved from 4.72983 to 4.70698, saving model to
saved_models/weights.best.from_scratch.keras
208/208 - 296s - 1s/step - accuracy: 0.0325 - loss: 4.6461 -
val_accuracy: 0.0150 - val_loss: 4.7070 - learning_rate: 0.0010
Epoch 8/10

Epoch 8: val_loss improved from 4.70698 to 4.66300, saving model to
saved_models/weights.best.from_scratch.keras
208/208 - 295s - 1s/step - accuracy: 0.0342 - loss: 4.6206 -
val_accuracy: 0.0234 - val_loss: 4.6630 - learning_rate: 0.0010
Epoch 9/10

Epoch 9: val_loss did not improve from 4.66300
208/208 - 297s - 1s/step - accuracy: 0.0397 - loss: 4.5901 -
val_accuracy: 0.0294 - val_loss: 4.6680 - learning_rate: 0.0010
Epoch 10/10

Epoch 10: val_loss did not improve from 4.66300
208/208 - 292s - 1s/step - accuracy: 0.0454 - loss: 4.5598 -
val_accuracy: 0.0294 - val_loss: 4.7028 - learning_rate: 0.0010

<keras.src.callbacks.history.History at 0x130d9b74e90>
```

Load the Model with the Best Validation Loss

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

Test the Model

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

```
# Evaluate the model on the test data using `evaluate_generator`
test_loss, test_accuracy = model.evaluate(test_generator,
steps=len(test_files) // 32)

# Convert the accuracy to percentage
test_accuracy = test_accuracy * 100

print('Test accuracy: %.4f%%' % test_accuracy)
```

```
26/26 ━━━━━━━━━━━━━━━━━━━━ 27s 1s/step - accuracy: 0.0224 - loss:
4.7076
Test accuracy: 2.8846%
```

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

```
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.

```
VGG16_model = Sequential()
VGG16_model.add(GlobalAveragePooling2D(input_shape=train_VGG16.shape[1
:]))
```

```
VGG16_model.add(Dense(133, activation='softmax'))

VGG16_model.summary()
```

```
Model: "sequential_4"
```

```
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━┓
┃ Layer (type)                         ┃ Output Shape                ┃         Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━┩
│ global_average_pooling2d_3           │ (None, 512)                 │               0 │
│ (GlobalAveragePooling2D)             │                             │                 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ dense_3 (Dense)                      │ (None, 133)                 │          68,229 │
└──────────────────────────────────────┴─────────────────────────────┴─────────────────┘
```

```
 Total params: 68,229 (266.52 KB)

 Trainable params: 68,229 (266.52 KB)

 Non-trainable params: 0 (0.00 B)
```

Compile the Model

```
VGG16_model.compile(loss='categorical_crossentropy',
optimizer='rmsprop', metrics=['accuracy'])
```

Train the Model

```
checkpointer =
ModelCheckpoint(filepath='saved_models/weights.best.VGG16.keras',
                                verbose=1, save_best_only=True)

VGG16_model.fit(train_VGG16, train_targets,
          validation_data=(valid_VGG16, valid_targets),
          epochs=20, batch_size=32, callbacks=[checkpointer],
verbose=1)
```

```
Epoch 1/20
200/209 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.0901 - loss:
12.4367
Epoch 1: val_loss improved from inf to 3.84779, saving model to
saved_models/weights.best.VGG16.keras
209/209 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.0952 - loss:
```

```
12.2301 - val_accuracy: 0.4000 - val_loss: 3.8478
Epoch 2/20
204/209 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5341 - loss:
2.5558
Epoch 2: val_loss improved from 3.84779 to 2.58398, saving model to
saved_models/weights.best.VGG16.keras
209/209 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.5349 - loss:
2.5503 - val_accuracy: 0.5509 - val_loss: 2.5840
Epoch 3/20
204/209 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.7081 - loss:
1.3104
Epoch 3: val_loss improved from 2.58398 to 2.13821, saving model to
saved_models/weights.best.VGG16.keras
209/209 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.7085 - loss:
1.3102 - val_accuracy: 0.6132 - val_loss: 2.1382
Epoch 4/20
191/209 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.8170 - loss:
0.7373
Epoch 4: val_loss improved from 2.13821 to 1.91544, saving model to
saved_models/weights.best.VGG16.keras
209/209 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.8163 - loss:
0.7452 - val_accuracy: 0.6455 - val_loss: 1.9154
Epoch 5/20
206/209 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.8581 - loss:
0.5314
Epoch 5: val_loss improved from 1.91544 to 1.68450, saving model to
saved_models/weights.best.VGG16.keras
209/209 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.8581 - loss:
0.5322 - val_accuracy: 0.6970 - val_loss: 1.6845
Epoch 6/20
208/209 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.8909 - loss:
0.3791
Epoch 6: val_loss did not improve from 1.68450
209/209 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.8909 - loss:
0.3793 - val_accuracy: 0.6731 - val_loss: 1.7491
Epoch 7/20
205/209 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9276 - loss:
0.2402
Epoch 7: val_loss improved from 1.68450 to 1.67432, saving model to
saved_models/weights.best.VGG16.keras
209/209 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9274 - loss:
0.2410 - val_accuracy: 0.6862 - val_loss: 1.6743
Epoch 8/20
200/209 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9465 - loss:
0.1820
Epoch 8: val_loss did not improve from 1.67432
209/209 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.9462 - loss:
0.1829 - val_accuracy: 0.7006 - val_loss: 1.7244
Epoch 9/20
```

```
203/209 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9648 - loss:
0.1187
Epoch 9: val_loss improved from 1.67432 to 1.63412, saving model to
saved_models/weights.best.VGG16.keras
209/209 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9645 - loss:
0.1195 - val_accuracy: 0.7126 - val_loss: 1.6341
Epoch 10/20
205/209 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9729 - loss:
0.0801
Epoch 10: val_loss did not improve from 1.63412
209/209 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9727 - loss:
0.0807 - val_accuracy: 0.7246 - val_loss: 1.7114
Epoch 11/20
200/209 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9793 - loss:
0.0615
Epoch 11: val_loss improved from 1.63412 to 1.62140, saving model to
saved_models/weights.best.VGG16.keras
209/209 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9791 - loss:
0.0623 - val_accuracy: 0.7222 - val_loss: 1.6214
Epoch 12/20
203/209 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9806 - loss:
0.0591
Epoch 12: val_loss did not improve from 1.62140
209/209 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9805 - loss:
0.0592 - val_accuracy: 0.7293 - val_loss: 1.6399
Epoch 13/20
209/209 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9861 - loss:
0.0401
Epoch 13: val_loss did not improve from 1.62140
209/209 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9861 - loss:
0.0401 - val_accuracy: 0.7234 - val_loss: 1.6738
Epoch 14/20
199/209 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9908 - loss:
0.0312
Epoch 14: val_loss did not improve from 1.62140
209/209 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9907 - loss:
0.0316 - val_accuracy: 0.7281 - val_loss: 1.6873
Epoch 15/20
194/209 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9942 - loss:
0.0237
Epoch 15: val_loss improved from 1.62140 to 1.57338, saving model to
saved_models/weights.best.VGG16.keras
209/209 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9939 - loss:
0.0244 - val_accuracy: 0.7425 - val_loss: 1.5734
Epoch 16/20
198/209 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9949 - loss:
0.0198
Epoch 16: val_loss did not improve from 1.57338
209/209 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9949 - loss:
```

```
0.0199 - val_accuracy: 0.7317 - val_loss: 1.6814
Epoch 17/20
204/209 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9942 - loss:
0.0176
Epoch 17: val_loss did not improve from 1.57338
209/209 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9943 - loss:
0.0176 - val_accuracy: 0.7341 - val_loss: 1.7069
Epoch 18/20
201/209 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9966 - loss:
0.0138
Epoch 18: val_loss did not improve from 1.57338
209/209 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9966 - loss:
0.0139 - val_accuracy: 0.7521 - val_loss: 1.6384
Epoch 19/20
209/209 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9973 - loss:
0.0123
Epoch 19: val_loss did not improve from 1.57338
209/209 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9973 - loss:
0.0123 - val_accuracy: 0.7533 - val_loss: 1.6382
Epoch 20/20
208/209 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9965 - loss:
0.0102
Epoch 20: val_loss did not improve from 1.57338
209/209 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9965 - loss:
0.0102 - val_accuracy: 0.7389 - val_loss: 1.6304

<keras.src.callbacks.history.History at 0x2265dd80ed0>
```

Load the Model with the Best Validation Loss

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

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.

```
# Test the model
test_predictions = VGG16_model.predict(test_VGG16, batch_size=20)
test_accuracy = 100 * np.mean(np.argmax(test_predictions, axis=1) ==
np.argmax(test_targets, axis=1))
print('Test accuracy: %.4f%%' % test_accuracy)

42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step
Test accuracy: 74.1627%
```

Predict Dog Breed with the Model

```
def VGG16_predict_breed(img_path):
    # extract bottleneck features
    bottleneck_feature = extract_VGG16(path_to_tensor(img_path))
    # obtain predicted vector
    predicted_vector = VGG16_model.predict(bottleneck_feature)
    # return dog breed that is predicted by the model
    return dog_names[np.argmax(predicted_vector)]
```

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']
```

```
### TODO: Obtain bottleneck features from another pre-trained CNN.
bottleneck_features =
np.load('bottleneck_features/DogResnet50Data.npz')
train_Resnet50 = bottleneck_features['train']
valid_Resnet50 = bottleneck_features['valid']
test_Resnet50 = bottleneck_features['test']
```

(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:

```
### TODO: Define your architecture.
Resnet50_model = Sequential()
Resnet50_model.add(GlobalAveragePooling2D(input_shape=train_Resnet50.s
hape[1:]))
Resnet50_model.add(Dense(133, activation='softmax'))

Resnet50_model.summary()
```

```
c:\Users\User\anaconda3\envs\udacity3\Lib\site-packages\keras\src\
layers\pooling\base_global_pooling.py:12: UserWarning: Do not pass an
`input_shape`/`input_dim` argument to a layer. When using Sequential
models, prefer using an `Input(shape)` object as the first layer in
the model instead.
  super().__init__(**kwargs)

Model: "sequential"
```

```
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━
┓
┃ Layer (type)                         ┃ Output Shape                ┃
Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━
┩
│ global_average_pooling2d             │ (None, 2048)                │
0 │
│ (GlobalAveragePooling2D)             │                             │
│
├──────────────────────────────────────┼─────────────────────────────┼─────────────────
┤
│ dense (Dense)                        │ (None, 133)                 │
```

```
272,517 │
└──────────────────────────────────┴──────────────────────────────┴──────────
──────┘

 Total params: 272,517 (1.04 MB)

 Trainable params: 272,517 (1.04 MB)

 Non-trainable params: 0 (0.00 B)
```

(IMPLEMENTATION) Compile the Model

```
### TODO: Compile the model.
Resnet50_model.compile(loss='categorical_crossentropy',
optimizer='rmsprop', 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.

```
### TODO: Train the model.
checkpointer =
ModelCheckpoint(filepath='saved_models/weights.best.Resnet50.keras',
                               verbose=1, save_best_only=True)

Resnet50_model.fit(train_Resnet50, train_targets,
          validation_data=(valid_Resnet50, valid_targets),
          epochs=20, batch_size=32, callbacks=[checkpointer],
verbose=1)

Epoch 1/20
195/209 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.3865 - loss:
2.9235
Epoch 1: val_loss improved from inf to 0.87313, saving model to
saved_models/weights.best.Resnet50.keras
209/209 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.4002 - loss:
2.8410 - val_accuracy: 0.7353 - val_loss: 0.8731
Epoch 2/20
199/209 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8548 - loss:
0.4949
Epoch 2: val_loss improved from 0.87313 to 0.74182, saving model to
saved_models/weights.best.Resnet50.keras
209/209 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.8551 - loss:
0.4931 - val_accuracy: 0.7760 - val_loss: 0.7418
Epoch 3/20
204/209 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9283 - loss:
0.2497
```

```
Epoch 3: val_loss improved from 0.74182 to 0.67383, saving model to
saved_models/weights.best.Resnet50.keras
209/209 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.9281 - loss:
0.2501 - val_accuracy: 0.7796 - val_loss: 0.6738
Epoch 4/20
197/209 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9570 - loss:
0.1572
Epoch 4: val_loss improved from 0.67383 to 0.61043, saving model to
saved_models/weights.best.Resnet50.keras
209/209 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9567 - loss:
0.1575 - val_accuracy: 0.8156 - val_loss: 0.6104
Epoch 5/20
204/209 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9756 - loss:
0.1026
Epoch 5: val_loss did not improve from 0.61043
209/209 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.9755 - loss:
0.1027 - val_accuracy: 0.8287 - val_loss: 0.6149
Epoch 6/20
201/209 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9850 - loss:
0.0623
Epoch 6: val_loss did not improve from 0.61043
209/209 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.9849 - loss:
0.0627 - val_accuracy: 0.8096 - val_loss: 0.6275
Epoch 7/20
202/209 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9911 - loss:
0.0474
Epoch 7: val_loss did not improve from 0.61043
209/209 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9910 - loss:
0.0475 - val_accuracy: 0.8156 - val_loss: 0.6228
Epoch 8/20
202/209 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9961 - loss:
0.0299
Epoch 8: val_loss did not improve from 0.61043
209/209 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.9960 - loss:
0.0301 - val_accuracy: 0.8263 - val_loss: 0.6136
Epoch 9/20
201/209 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9965 - loss:
0.0228
Epoch 9: val_loss improved from 0.61043 to 0.60809, saving model to
saved_models/weights.best.Resnet50.keras
209/209 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9964 - loss:
0.0229 - val_accuracy: 0.8311 - val_loss: 0.6081
Epoch 10/20
197/209 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9983 - loss:
0.0151
Epoch 10: val_loss did not improve from 0.60809
209/209 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9983 - loss:
0.0153 - val_accuracy: 0.8359 - val_loss: 0.6203
Epoch 11/20
```

```
208/209 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9979 - loss:
0.0122
Epoch 11: val_loss improved from 0.60809 to 0.58949, saving model to
saved_models/weights.best.Resnet50.keras
209/209 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9979 - loss:
0.0122 - val_accuracy: 0.8467 - val_loss: 0.5895
Epoch 12/20
197/209 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9976 - loss:
0.0132
Epoch 12: val_loss did not improve from 0.58949
209/209 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9976 - loss:
0.0131 - val_accuracy: 0.8347 - val_loss: 0.6404
Epoch 13/20
199/209 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9973 - loss:
0.0132
Epoch 13: val_loss did not improve from 0.58949
209/209 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9974 - loss:
0.0130 - val_accuracy: 0.8359 - val_loss: 0.6274
Epoch 14/20
206/209 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9985 - loss:
0.0095
Epoch 14: val_loss did not improve from 0.58949
209/209 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step - accuracy: 0.9985 - loss:
0.0095 - val_accuracy: 0.8407 - val_loss: 0.6095
Epoch 15/20
202/209 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9984 - loss:
0.0062
Epoch 15: val_loss did not improve from 0.58949
209/209 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - accuracy: 0.9984 - loss:
0.0062 - val_accuracy: 0.8419 - val_loss: 0.6253
Epoch 16/20
202/209 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9992 - loss:
0.0042
Epoch 16: val_loss did not improve from 0.58949
209/209 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9992 - loss:
0.0043 - val_accuracy: 0.8395 - val_loss: 0.6511
Epoch 17/20
204/209 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9976 - loss:
0.0085
Epoch 17: val_loss did not improve from 0.58949
209/209 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.9976 - loss:
0.0084 - val_accuracy: 0.8443 - val_loss: 0.6292
Epoch 18/20
199/209 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9992 - loss:
0.0041
Epoch 18: val_loss did not improve from 0.58949
209/209 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.9992 - loss:
0.0042 - val_accuracy: 0.8491 - val_loss: 0.6306
Epoch 19/20
```

```
208/209 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9981 - loss:
0.0056
Epoch 19: val_loss did not improve from 0.58949
209/209 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9981 - loss:
0.0055 - val_accuracy: 0.8419 - val_loss: 0.6345
Epoch 20/20
207/209 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9989 - loss:
0.0042
Epoch 20: val_loss did not improve from 0.58949
209/209 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.9989 - loss:
0.0043 - val_accuracy: 0.8539 - val_loss: 0.6303

<keras.src.callbacks.history.History at 0x2259ae4b710>
```

(IMPLEMENTATION) Load the Model with the Best Validation Loss

```
### TODO: Load the model weights with the best validation loss.
Resnet50_model.load_weights('saved_models/weights.best.Resnet50.keras'
)
```

(IMPLEMENTATION) Test the Model

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

```
### TODO: Calculate classification accuracy on the test dataset.
# Test the model
test_predictions = Resnet50_model.predict(test_Resnet50,
batch_size=20)
test_accuracy = 100 * np.mean(np.argmax(test_predictions, axis=1) ==
np.argmax(test_targets, axis=1))
print('Test accuracy: %.4f%%' % test_accuracy)

42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step
Test accuracy: 83.2536%
```

(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`.

```
### 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 Resnet50_predict_breed(img_path):
    # extract bottleneck features
    bottleneck_feature = extract_Resnet50(path_to_tensor(img_path))
    # obtain predicted vector
    predicted_vector =
Resnet50_model.predict(bottleneck_feature,verbose=0)
    # return dog breed that is predicted by the model
    return dog_names[np.argmax(predicted_vector)]
```

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

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

```
def humnan_dog_classification(img_path):
    if dog_detector(img_path):
        plot_an_image(img_path)
        print(f'This picture is a {Resnet50_predict_breed(img_path)}
dog!')
    elif face_detector(img_path):
        plot_an_image(img_path)
        print(f'This picture is a human resembling a
{Resnet50_predict_breed(img_path)} dog!')
    else:
        plot_an_image(img_path)
        print(f'This picture is a neither a human nor a dog!')
```

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.

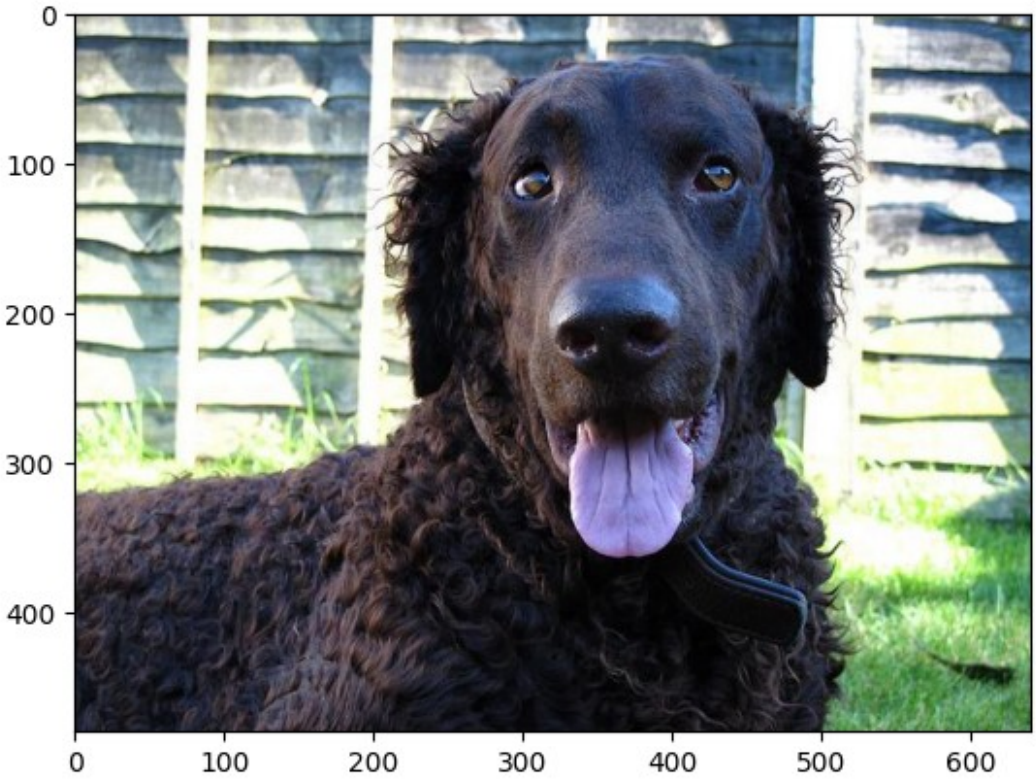
Question 6: Is the output better than you expected :) ? Or worse :(? Provide at least three possible points of improvement for your algorithm.

Answer:

```
## 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.

list_pics = os.listdir('images')[1:]
for pic in list_pics:
    humnan_dog_classification('images\\'+pic)
```


```
1/1 ━━━━━━━━━━━━━━━━━━━━ 2s 2s/step
This picture is a n\091.Japanese_chin dog!
```

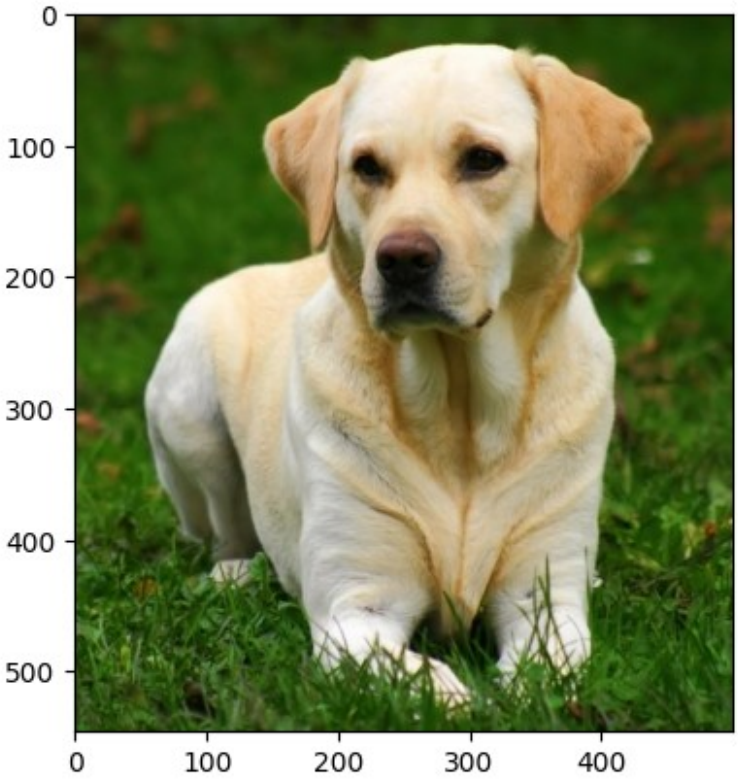

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 2s 2s/step
This picture is a n\091.Japanese_chin dog!
```



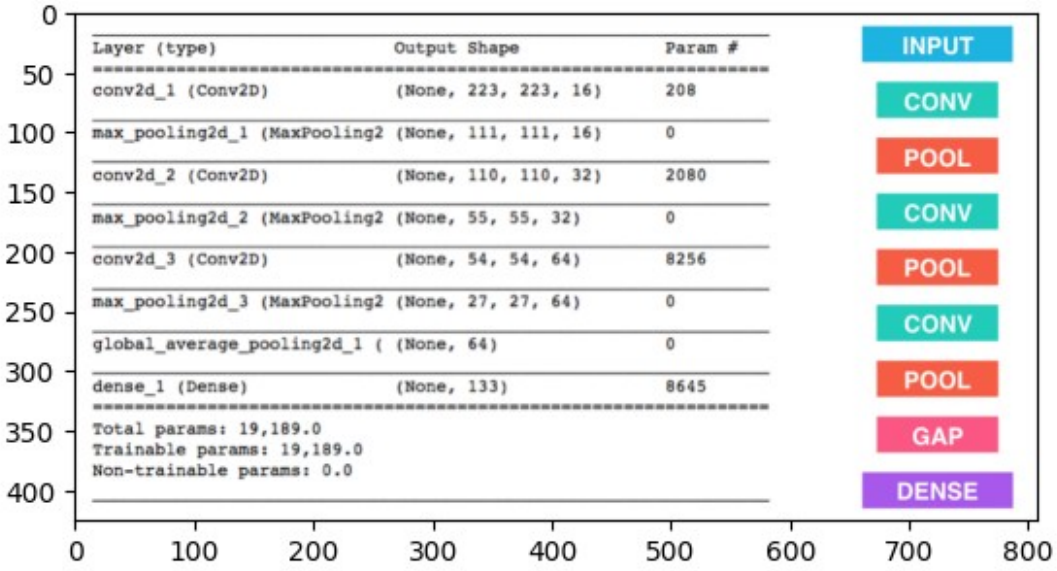

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 2s 2s/step
This picture is a n\091.Japanese_chin dog!
```


```
1/1 ━━━━━━━━━━━━━━━━━━━━ 2s 2s/step
This picture is a n\091.Japanese_chin dog!
```

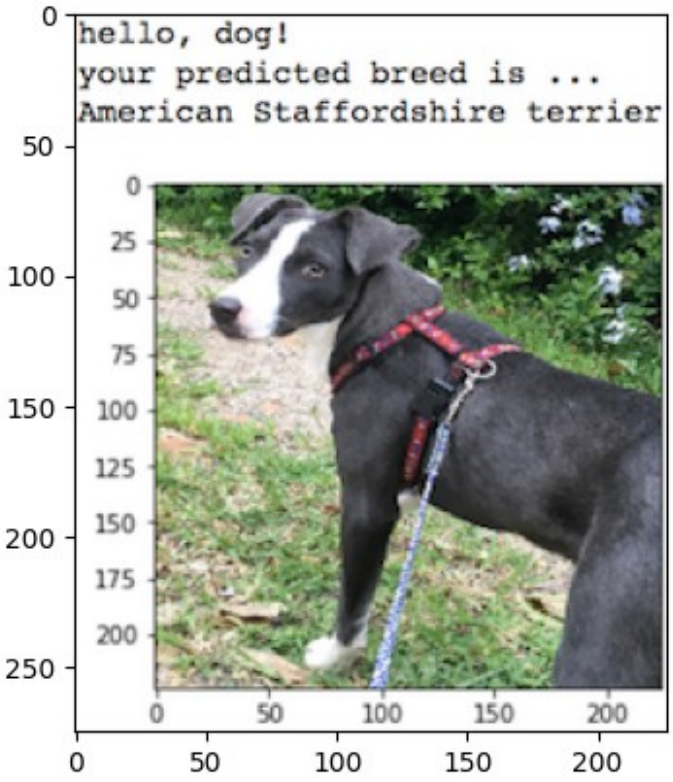
```
1/1 ━━━━━━━━━━━━━━━━━━━━ 2s 2s/step
This picture is a n\091.Japanese_chin dog!
```



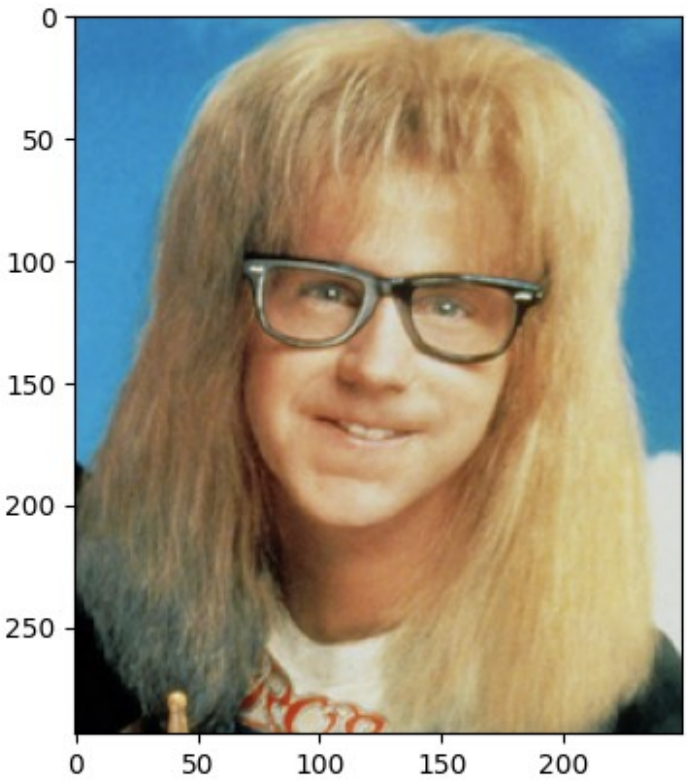
```
1/1 ━━━━━━━━━━━━━━━━━━━━ 2s 2s/step
This picture is a n\091.Japanese_chin dog!
```



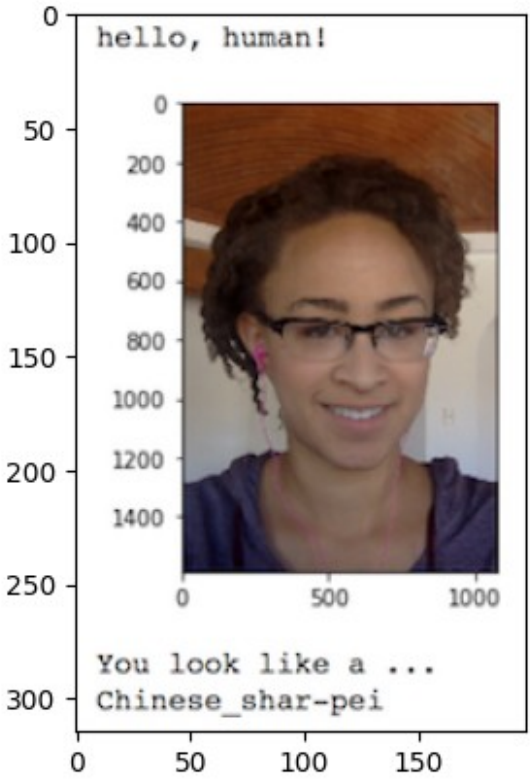
```
This picture is a neither a human nor a dog!
```




```
This picture is a neither a human nor a dog!
```




```
1/1 ━━━━━━━━━━━━━━━━━━━━ 2s 2s/step
This picture is a human resembling a n\091.Japanese_chin dog!
```




```
1/1 ━━━━━━━━━━━━━━━━━━━━ 2s 2s/step
This picture is a human resembling a n\049.Chinese_crested dog!
```


```
1/1 ━━━━━━━━━━━━━━━━━━━━ 2s 2s/step
This picture is a n\091.Japanese_chin dog!
```