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.

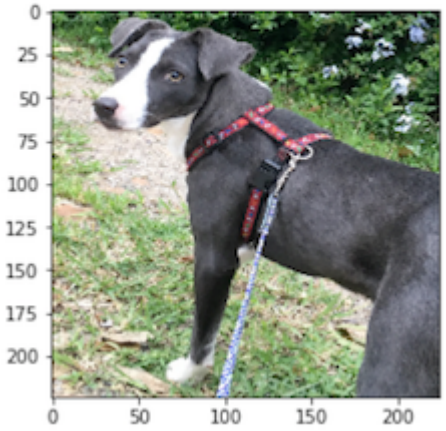
> **Note:** Code and Markdown cells can be executed using the **Shift + Enter** keyboard shortcut. Markdown cells can be edited by double-clicking the cell to enter edit mode.

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!).

```
hello, dog!
your predicted breed is ...
American Staffordshire terrier
```



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 [18]:
```
# Run the following cell only if the /workspace/home/dog-project/dog_imag
# The cell below will copy the data to your /workspace directory.
# !cp -rp /data/dog_images/ /workspace/home/dog-project

'''
NOTE
There is a known "Permission denied" issue with copying the following fil
- Ibizan_hound_05697.jpg
- American_foxhound_00503.jpg
- Basset_hound_01064.jpg
- Labrador_retriever_06476.jpg
- Manchester_terrier_06806.jpg
- Norwegian_lundehund_07217.jpg
'''
```

Out[18]:
```
'\nNOTE\nThere is a known "Permission denied" issue with copying the fol
lowing files. You can ignore them.\n- Ibizan_hound_05697.jpg\n- American
_foxhound_00503.jpg\n- Basset_hound_01064.jpg\n- Labrador_retriever_0647
6.jpg\n- Manchester_terrier_06806.jpg\n- Norwegian_lundehund_07217.jpg
\n'
```

In [19]:
```
# Install the necessary package
# Restart the kernel once after install this package

# !python3 -m pip install opencv-python-headless==4.9.0.80

#
# https://stackoverflow.com/questions/77617946/solve-conda-libmamba-solve
# conda install --solver=classic conda-forge::conda-libmamba-solver conda
#
# conda install numpy
# conda install scipy pandas
# conda install matplotlib
# conda install scikit-learn
# conda install seaborn
# conda install six
# conda install tensorflow
# conda install fastai::opencv-python-headless
#
import numpy as np
```

In [20]:
```
from sklearn.datasets import load_files
from keras.utils import to_categorical
from glob import glob

# 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

# Load train, test, and validation datasets
train_files, train_targets = load_dataset('dogImages/train')
valid_files, valid_targets = load_dataset('dogImages/valid')
test_files, test_targets = load_dataset('dogImages/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(f'There are {len(dog_names)} total dog categories.')
print(f'There are {len(train_files) + len(valid_files) + len(test_files)}
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 0 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 [21]:
```
import random
random.seed(8675309)

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

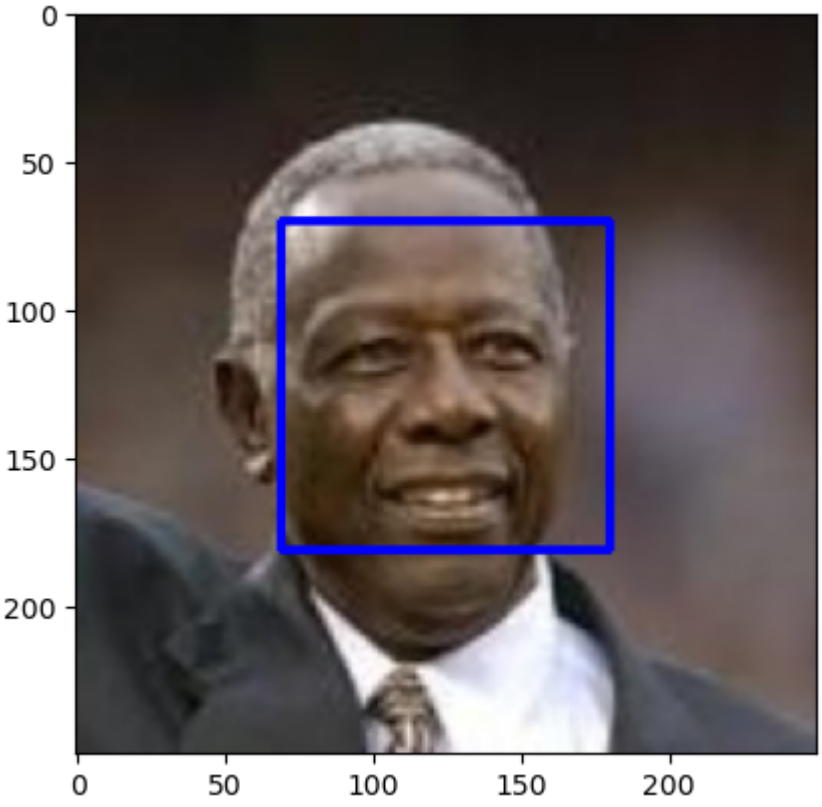
Step 1: Detect Humans

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.

In [22]:
```
import cv2
import matplotlib.pyplot as plt
%matplotlib inline

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

# 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 [23]:
```
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, faces
```

(IMPLEMENTATION) Assess the Human Face Detector

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

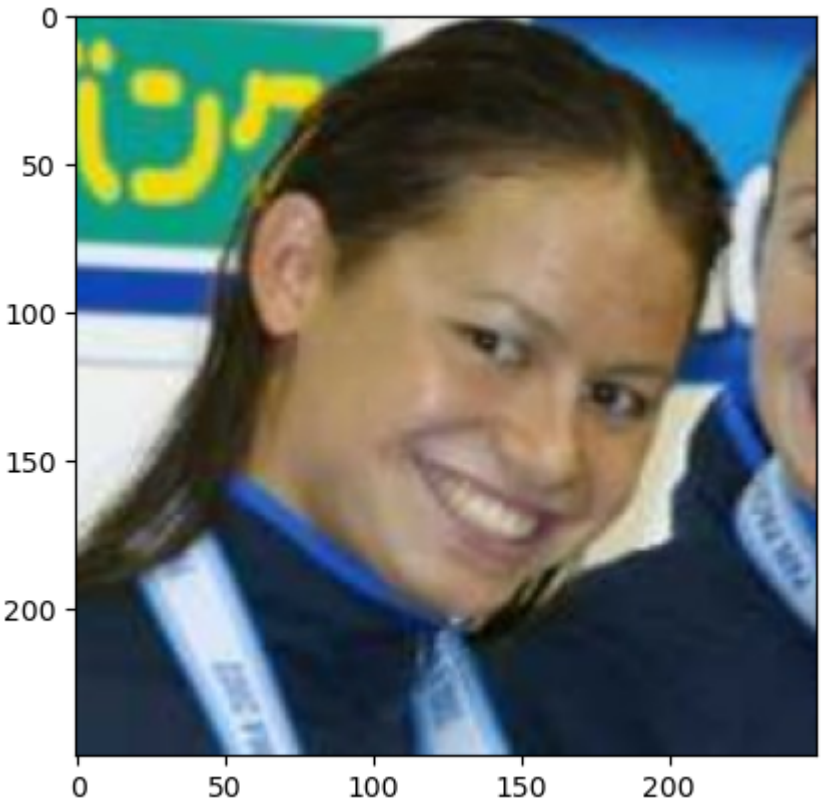
- What percentage of the first 100 images in `human_files` have a detected human face?
- What percentage of the first 100 images in `dog_files` have a detected human face?

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`.

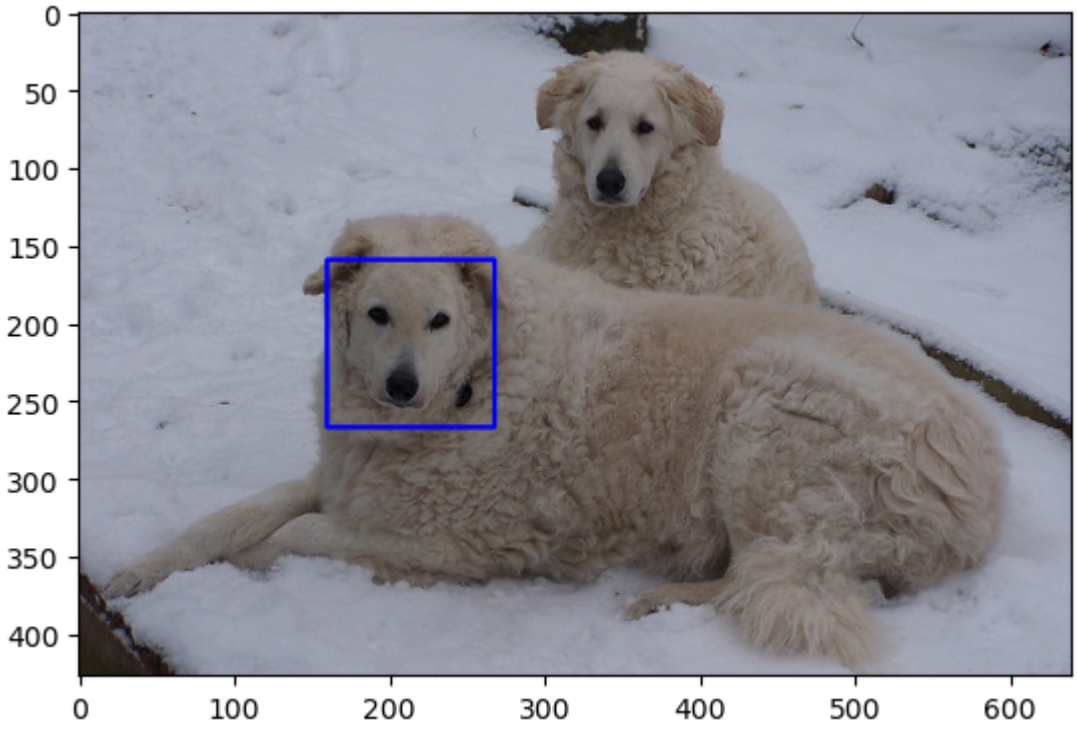
Answer:

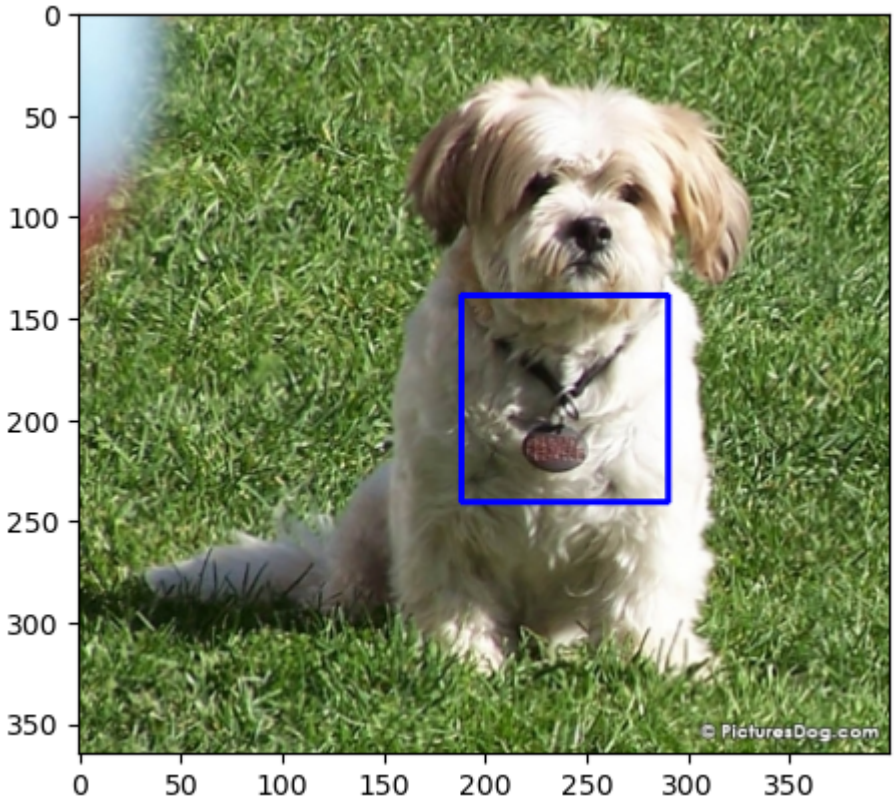
In [24]:
```
human_files_short = human_files[:100]
dog_files_short = train_files[:100]
# Do NOT modify the code above this line.

## TODO: Test the performance of the face_detector algorithm
## on the images in human_files_short and dog_files_short.
human_detected = 0
for img_path in human_files_short:
    is_humam, faces = face_detector(img_path)
    if is_humam:
        human_detected += 1
```

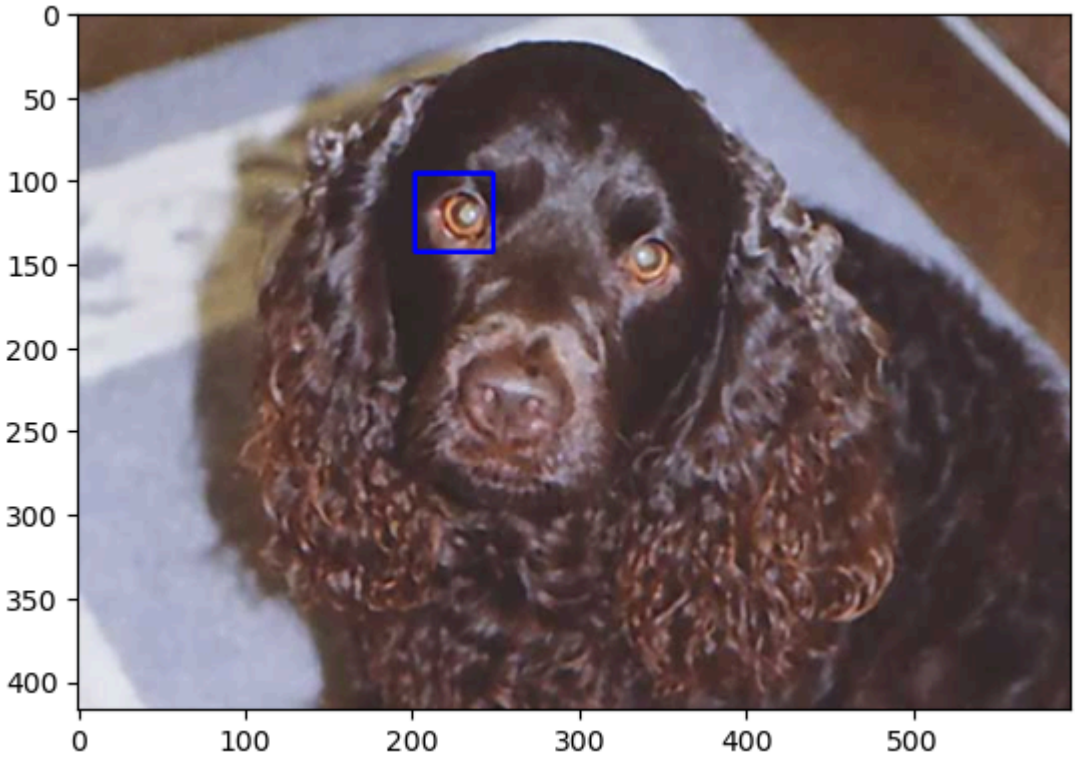
```
    else:
        img = cv2.imread(img_path)
        cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        plt.imshow(cv_rgb)
        plt.show()

human_detection_rate = human_detected / len(human_files_short) * 100
print("{:.2f}% human face detection rate".format(human_detection_rate))

dog_false_positives = 0
for img_path in dog_files_short:
    is_humam, faces = face_detector(img_path)
    if is_humam:
        dog_false_positives += 1
        img = cv2.imread(img_path)
        for (x,y,w,h) in faces:
            cv2.rectangle(img,(x,y),(x+w,y+h),(255,0,0),2)
        cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        plt.imshow(cv_rgb)
        plt.show()

dog_false_positive_detection_rate = dog_false_positives / len(dog_files_s
print("{:.2f}% dog face false-positive detection rate".format(dog_false_p
```

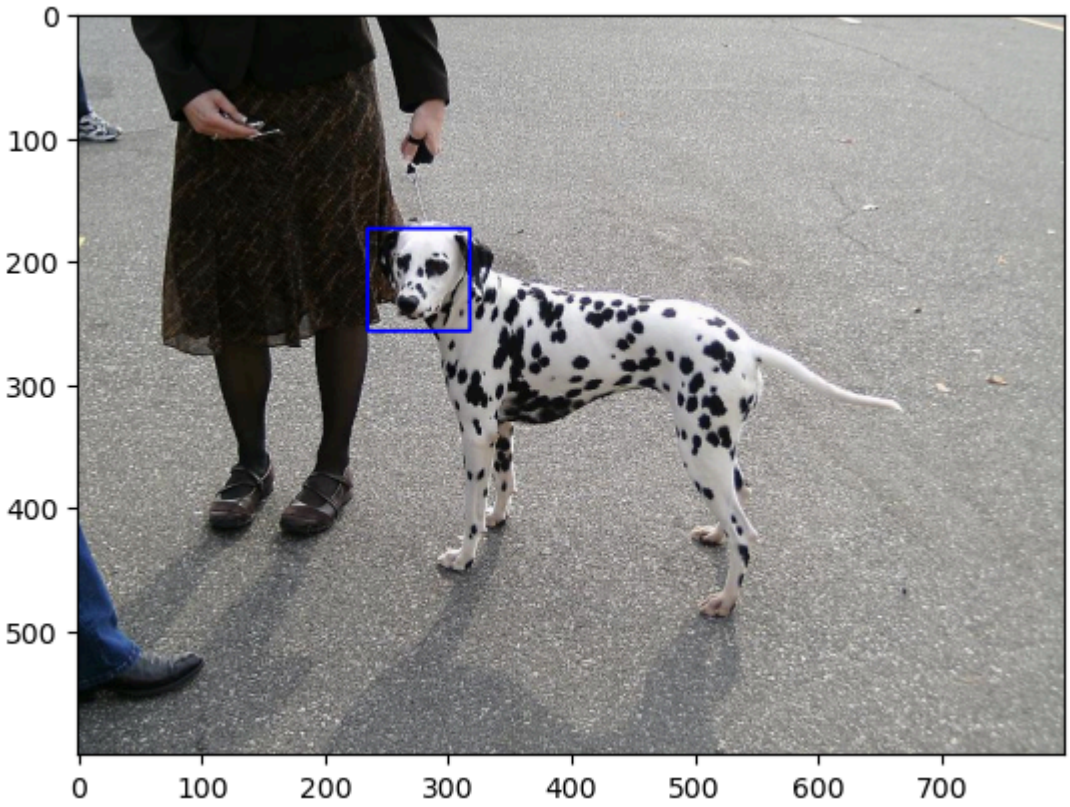


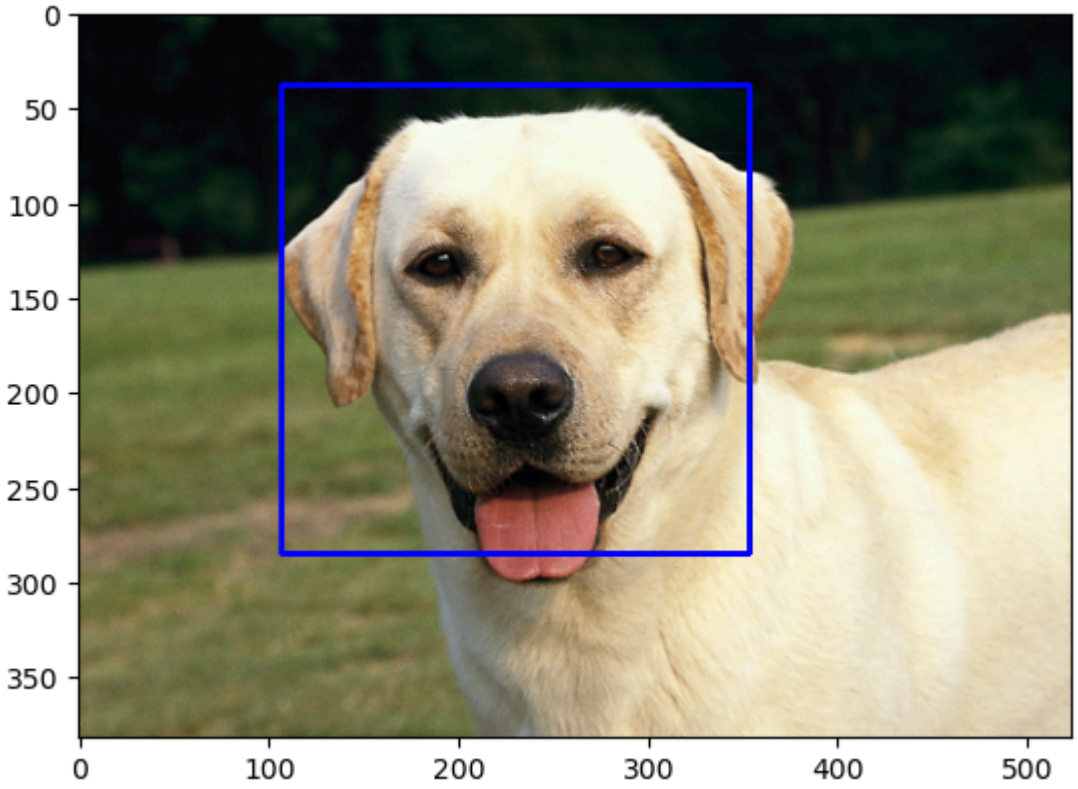

```
99.00% human face detection rate
```

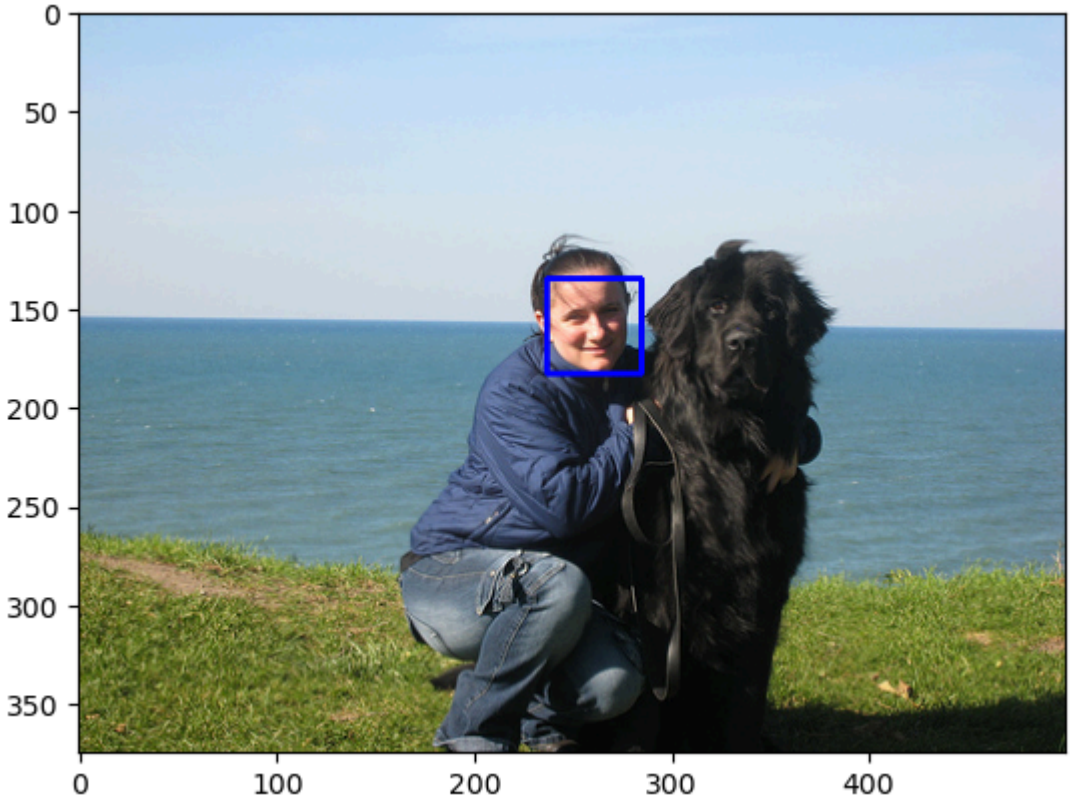

0
50
100
150
200
250
300
350
400
0
100
200
300
400
500
600

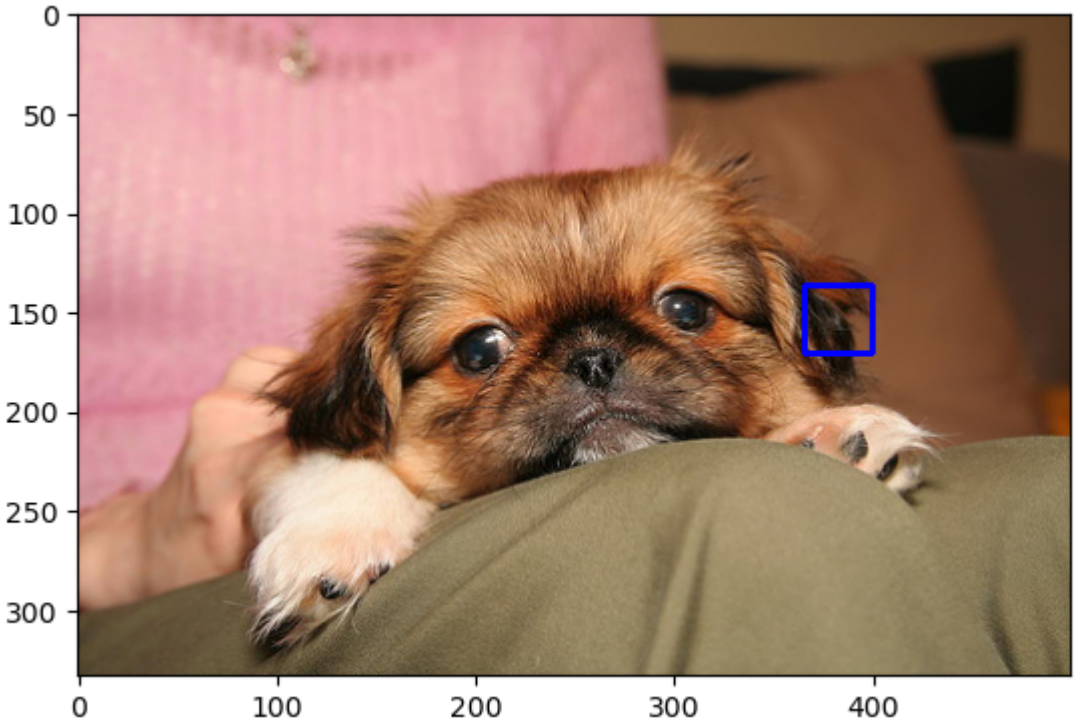

0
50
100
150
200
250
300
350
0
50
100
150
200
250
300
350
© PicturesDog.com


0
50
100
150
200
250
300
350
400
0
100
200
300
400
500


0
100
200
300
400
500
0
100
200
300
400
500
600
700

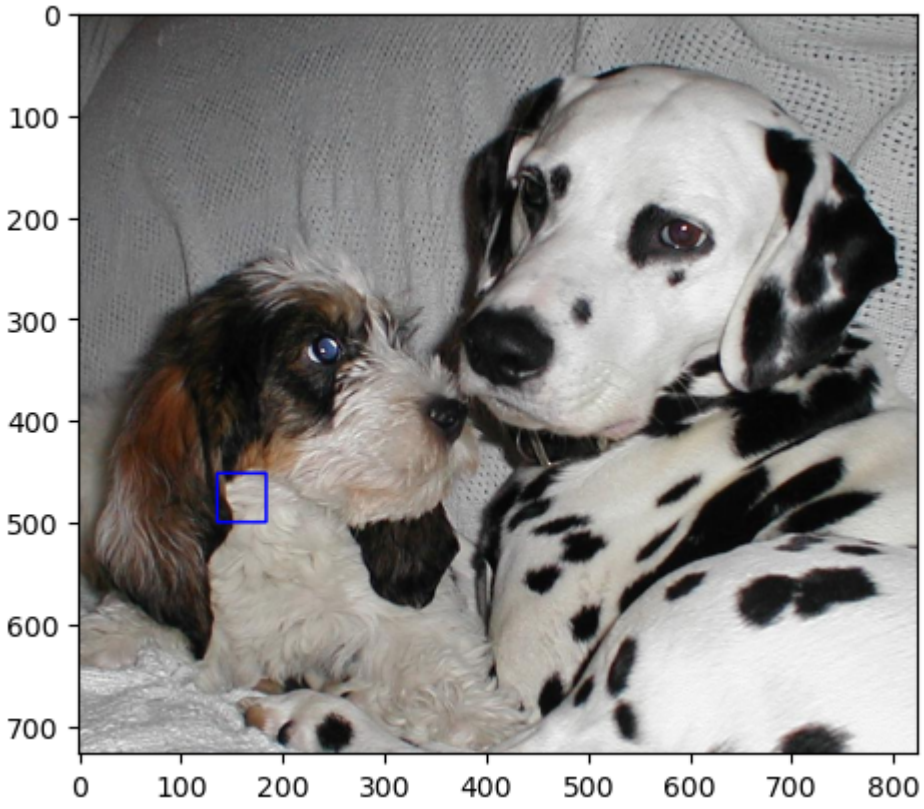

0
50
100
150
200
250
300
350
0
100
200
300
400
500

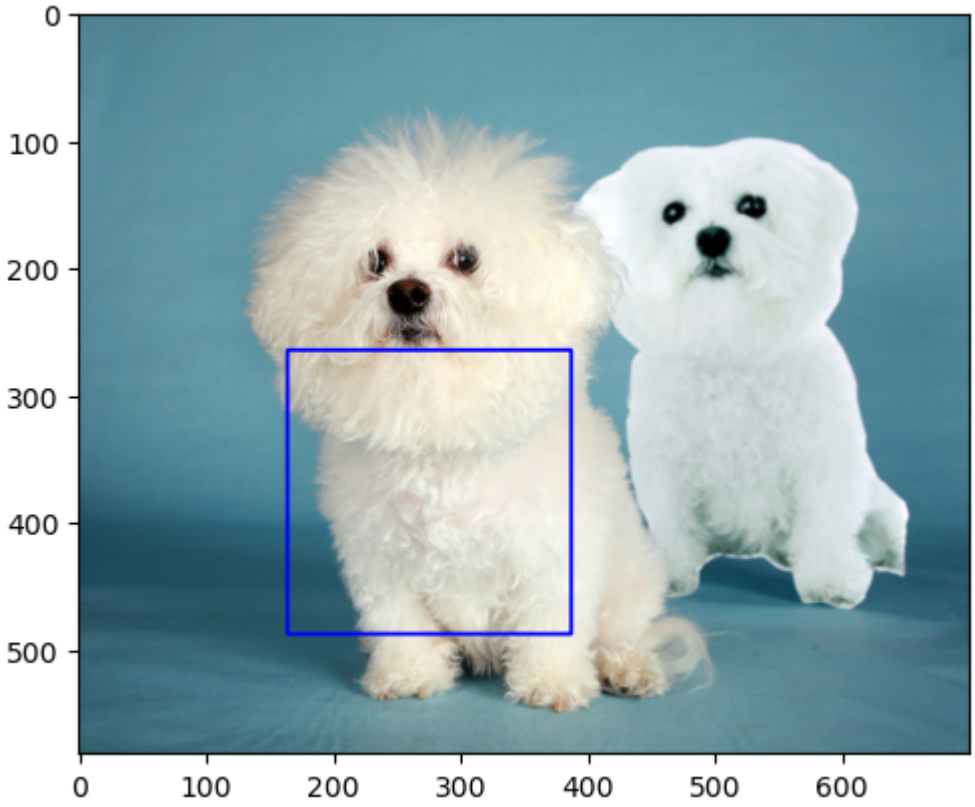

0
50
100
150
200
250
300
350
0
100
200
300
400

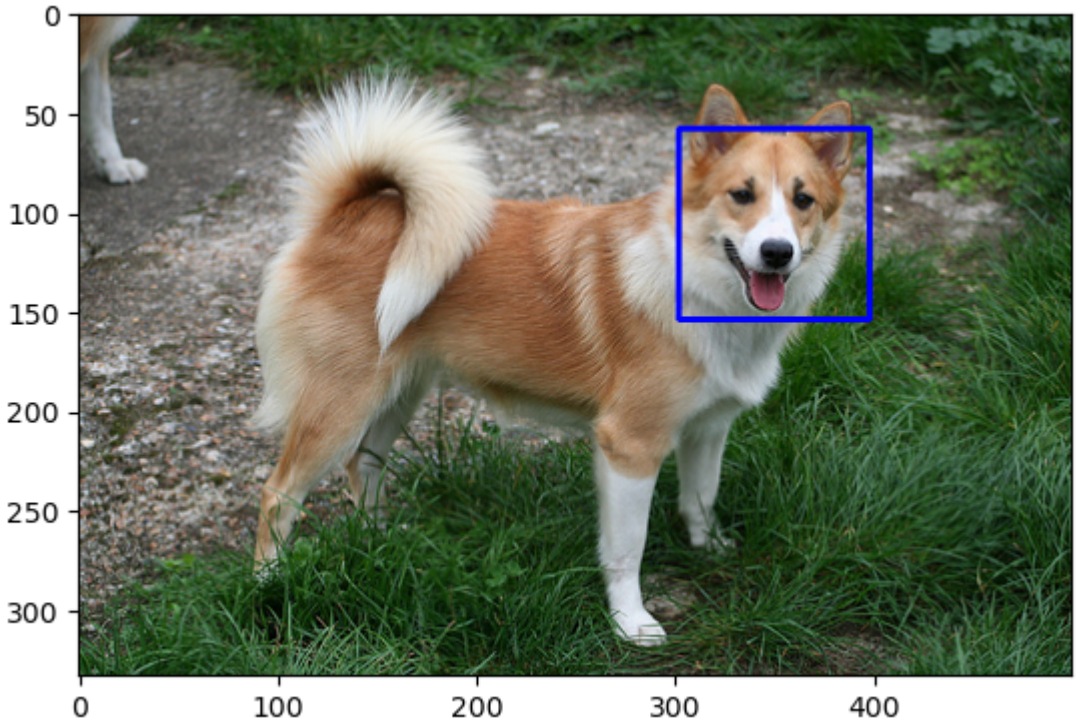

0
50
100
150
200
250
300
0
100
200
300
400


0
500
1000
1500
2000
0
500
1000
1500
2000
2500
VAVRA


0
50
100
150
200
250
300
350
0
100
200
300
400


0
100
200
300
400
500
600
700
0
100
200
300
400
500
600
700
800






```
12.00% dog face false-positive detection rate
```

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 [25]:
```
## (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.

In [26]:
```
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 [27]:
```
from keras.preprocessing import image

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.

In [28]:
```
from keras.applications.resnet50 import preprocess_input, decode_predicti
import keras

def ResNet50_predict_labels(img_path):
    img = image.load_img(img_path, target_size=(224, 224))
    x = keras.utils.img_to_array(img)
    x = np.expand_dims(x, axis=0)
```

```
    x = preprocess_input(x)
    predictions = ResNet50_model.predict(x, verbose=0)

    decoded_predictions = decode_predictions(predictions, top=1)[0]

    return decoded_predictions[0][1]
```

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 [29]:
```
### returns "True" if a dog is detected in the image stored at img_path
dog_keywords = [
    "affenpinscher",
    "afghan_hound",
    "african_hunting_dog",
    "airedale",
    "american_staffordshire_terrier",
    "australian_terrier",
    "basenji",
    "basset",
    "beagle",
    "bedlington_terrier",
    "bernese_mountain_dog",
    "black-and-tan_coonhound",
    "blenheim_spaniel",
    "bloodhound",
    "bluetick",
    "border_collie",
    "border_terrier",
    "borzoi",
    "boston_bull",
    "bouvier_des_flandres",
    "boxer",
    "briard",
    "brittany_spaniel",
    "bull_mastiff",
    "cairn",
    "cardigan",
    "chesapeake_bay_retriever",
    "chihuahua",
    "chow",
    "clumber",
    "cocker_spaniel",
    "collie",
    "curly-coated_retriever",
    "dalmatian",
    "dandie_dinmont",
```

```
    "doberman",
    "english_foxhound",
    "english_setter",
    "english_springer",
    "entlebucher",
    "eskimo_dog",
    "flat-coated_retriever",
    "french_bulldog",
    "german_shepherd",
    "german_short-haired_pointer",
    "giant_schnauzer",
    "golden_retriever",
    "gordon_setter",
    "great_dane",
    "great_pyrenees",
    "groenendael",
    "ibizan_hound",
    "irish_setter",
    "irish_terrier",
    "irish_water_spaniel",
    "irish_wolfhound",
    "italian_greyhound",
    "japanese_spaniel",
    "keeshond",
    "kelpie",
    "kerry_blue_terrier",
    "komondor",
    "kuvasz",
    "labrador_retriever",
    "lakeland_terrier",
    "leonberg",
    "lhasa",
    "malamute",
    "malinois",
    "maltese_dog",
    "mexican_hairless",
    "miniature_pinscher",
    "miniature_poodle",
    "miniature_schnauzer",
    "newfoundland",
    "norfolk_terrier",
    "norwegian_elkhound",
    "norwich_terrier",
    "old_english_sheepdog",
    "otterhound",
    "papillon",
    "pekinese",
    "pembroke",
    "pomeranian",
    "pug",
    "redbone",
    "rhodesian_ridgeback",
    "rottweiler",
    "saint_bernard",
    "saluki",
    "samoyed",
    "schipperke",
    "scotch_terrier",
    "scottish_deerhound",
    "sealyham_terrier",
```

```
    "shetland_sheepdog",
    "shih-tzu",
    "siberian_husky",
    "silky_terrier",
    "soft-coated_wheaten_terrier",
    "staffordshire_bullterrier",
    "standard_poodle",
    "standard_schnauzer",
    "sussex_spaniel",
    "tibetan_mastiff",
    "tibetan_terrier",
    "toy_poodle",
    "toy_terrier",
    "vizsla",
    "walker_hound",
    "weimaraner",
    "welsh_springer_spaniel",
    "west_highland_white_terrier",
    "whippet",
    "wire-haired_fox_terrier",
    "yorkshire_terrier",
]

def dog_detector(img_path):
    predicted_class = ResNet50_predict_labels(img_path)

    return any(keyword in predicted_class.lower() for keyword in dog_keyw
```

(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 [30]:
```
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.

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.
dog_false_positives = 0
for img_path in human_files_short:
    is_dog = dog_detector(img_path)
    if is_dog:
        dog_false_positives += 1

dog_false_positives_rate = dog_false_positives / len(human_files_short) *
print("{:.2f}% human face false-positive rate".format(dog_false_positives

dog_detected = 0
for img_path in dog_files_short:
```

```
    is_dog = dog_detector(img_path)
    if is_dog:
        dog_detected += 1

dog_detection_rate = dog_detected / len(dog_files_short) * 100
print("{:.2f}% dog face detection rate".format(dog_detection_rate))
```

```
1.00% human face false-positive rate
100.00% dog face detection rate
```

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

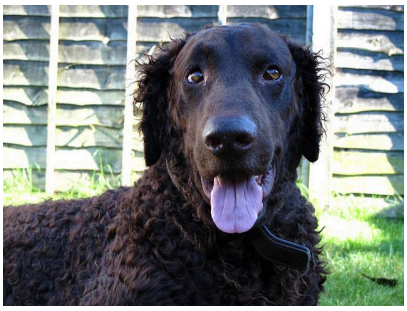
Now that we have functions for detecting humans and dogs in images, we need a way to predict breed from images. In this step, you will create a CNN that classifies dog breeds. You must create your CNN *from scratch* (so, you can't use transfer learning *yet*!), and you must attain a test accuracy of 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.

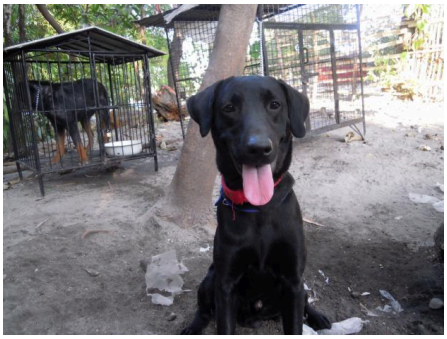
Brittany	Welsh Springer Spaniel
	

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 [31]:
```
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True  # Allow loading of truncated imag

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
        batch_indices = [np.where(files == img_path)[0][0] for img_path i
        batch_output = np.array([targets[index] for index in batch_indice

        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=
valid_generator = image_generator(valid_files, valid_targets, batch_size=
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:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 223, 223, 16)      208
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 111, 111, 16)      0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 110, 110, 32)      2080
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 55, 55, 32)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 54, 54, 64)        8256
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 27, 27, 64)        0
_________________________________________________________________
global_average_pooling2d_1 ( (None, 64)                0
_________________________________________________________________
dense_1 (Dense)              (None, 133)               8645
=================================================================
Total params: 19,189.0
Trainable params: 19,189.0
Non-trainable params: 0.0
_________________________________________________________________
```

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 [32]:
```
from keras.layers import Conv2D, MaxPooling2D, GlobalAveragePooling2D
from keras.layers import Dropout, Flatten, Dense
from keras.models import Sequential

### TODO: Define your architecture.
num_classes = 133
model = Sequential([
    Conv2D(16, (3, 3), activation='relu', input_shape=(224, 224, 3)),
```

```
    MaxPooling2D((2, 2)),
    Conv2D(32, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Flatten(),
    Dense(64, activation='relu'),
    Dense(num_classes, activation='softmax')
])
```

Compile the Model

In [33]:
```
# Set a smaller learning rate
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=

model.summary()
```

Model: "sequential_1"

Layer (type)	Output Shape	P
conv2d_2 (Conv2D)	(None, 222, 222, 16)	
max_pooling2d_2 (MaxPooling2D)	(None, 111, 111, 16)	
conv2d_3 (Conv2D)	(None, 109, 109, 32)	
max_pooling2d_3 (MaxPooling2D)	(None, 54, 54, 32)	
flatten_1 (Flatten)	(None, 93312)	
dense_2 (Dense)	(None, 64)	5,9
dense_3 (Dense)	(None, 133)	

Total params: 5,985,765 (22.83 MB)
Trainable params: 5,985,765 (22.83 MB)
Non-trainable params: 0 (0.00 B)

(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 [34]:
```
from keras.callbacks import ModelCheckpoint, ReduceLROnPlateau

### TODO: specify the number of epochs that you would like to use to trai

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",
    save_best_only=True,
```

```
    monitor="val_accuracy",
    verbose=1,
)

# ReduceLROnPlateau: This callback reduces the learning rate when a metri
reduce_lr = ReduceLROnPlateau(
    monitor="val_loss", factor=0.1, patience=5, verbose=1
)

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

```
Epoch 1/10

Epoch 1: val_accuracy improved from -inf to 0.01082, saving model to saved
_models/weights.best.from_scratch.keras
208/208 - 91s - 436ms/step - accuracy: 0.0133 - loss: 4.8662 - val_accurac
y: 0.0108 - val_loss: 4.8205 - learning_rate: 0.0010
Epoch 2/10

Epoch 2: val_accuracy improved from 0.01082 to 0.01623, saving model to sa
ved_models/weights.best.from_scratch.keras
208/208 - 89s - 428ms/step - accuracy: 0.0181 - loss: 4.7296 - val_accurac
y: 0.0162 - val_loss: 4.7686 - learning_rate: 0.0010
Epoch 3/10

Epoch 3: val_accuracy did not improve from 0.01623
208/208 - 90s - 435ms/step - accuracy: 0.0172 - loss: 4.6615 - val_accurac
y: 0.0150 - val_loss: 4.7551 - learning_rate: 0.0010
Epoch 4/10

Epoch 4: val_accuracy did not improve from 0.01623
208/208 - 89s - 430ms/step - accuracy: 0.0212 - loss: 4.6008 - val_accurac
y: 0.0156 - val_loss: 4.8141 - learning_rate: 0.0010
Epoch 5/10

Epoch 5: val_accuracy did not improve from 0.01623
208/208 - 86s - 414ms/step - accuracy: 0.0214 - loss: 4.5522 - val_accurac
y: 0.0144 - val_loss: 4.7307 - learning_rate: 0.0010
Epoch 6/10

Epoch 6: val_accuracy improved from 0.01623 to 0.01983, saving model to sa
ved_models/weights.best.from_scratch.keras
208/208 - 86s - 414ms/step - accuracy: 0.0242 - loss: 4.4987 - val_accurac
y: 0.0198 - val_loss: 4.7909 - learning_rate: 0.0010
Epoch 7/10

Epoch 7: val_accuracy did not improve from 0.01983
208/208 - 85s - 407ms/step - accuracy: 0.0222 - loss: 4.4657 - val_accurac
y: 0.0186 - val_loss: 4.7897 - learning_rate: 0.0010
Epoch 8/10

Epoch 8: val_accuracy did not improve from 0.01983
208/208 - 86s - 414ms/step - accuracy: 0.0234 - loss: 4.4360 - val_accurac
y: 0.0132 - val_loss: 4.8291 - learning_rate: 0.0010
Epoch 9/10

Epoch 9: val_accuracy improved from 0.01983 to 0.02103, saving model to sa
ved_models/weights.best.from_scratch.keras
208/208 - 84s - 402ms/step - accuracy: 0.0245 - loss: 4.4070 - val_accurac
y: 0.0210 - val_loss: 5.0316 - learning_rate: 0.0010
Epoch 10/10

Epoch 10: val_accuracy did not improve from 0.02103

Epoch 10: ReduceLROnPlateau reducing learning rate to 0.000100000004749745
13.
208/208 - 86s - 415ms/step - accuracy: 0.0262 - loss: 4.3928 - val_accurac
y: 0.0210 - val_loss: 4.8655 - learning_rate: 0.0010
```

Out[34]: <keras.src.callbacks.history.History at 0x7daf2c2d2ed0>

Load the Model with the Best Validation Loss

In [35]:
```
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%.

In [36]:
```
# Evaluate the model on the test data using `evaluate_generator`
test_loss, test_accuracy = model.evaluate(test_generator, steps=len(test_

# Convert the accuracy to percentage
test_accuracy = test_accuracy * 100

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

```
26/26 ━━━━━━━━━━━━━━━━━━━━ 6s 257ms/step - accuracy: 0.0222 - loss: 4.8688
Test accuracy: 2.0433%
```

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 [38]:
```
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 [39]:
```
VGG16_model = Sequential()
VGG16_model.add(GlobalAveragePooling2D(input_shape=train_VGG16.shape[1:]))
VGG16_model.add(Dense(133, activation='softmax'))

VGG16_model.summary()
```

```
/home/anibalsanchez/5_bin/miniconda3/lib/python3.11/site-packages/keras/sr
c/layers/pooling/base_global_pooling.py:12: UserWarning: Do not pass an `i
nput_shape`/`input_dim` argument to a layer. When using Sequential models,
prefer using an `Input(shape)` object as the first layer in the model inst
ead.
  super().__init__(**kwargs)
```

Model: "sequential_2"

Layer (type)	Output Shape	
global_average_pooling2d (GlobalAveragePooling2D)	(None, 512)	
dense_4 (Dense)	(None, 133)	

Total params: 68,229 (266.52 KB)

Trainable params: 68,229 (266.52 KB)

Non-trainable params: 0 (0.00 B)

Compile the Model

In [40]:
```
VGG16_model.compile(loss='categorical_crossentropy', optimizer='rmsprop',
```

Train the Model

In [41]:
```
checkpointer = ModelCheckpoint(filepath='saved_models/weights.best.VGG16.
                               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)
```

```
Epoch 1/20
293/334 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.1015 - loss: 12.279
3
Epoch 1: val_loss improved from inf to 3.82549, saving model to saved_mode
ls/weights.best.VGG16.keras
334/334 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.1159 - loss: 11.731
5 - val_accuracy: 0.4395 - val_loss: 3.8255
Epoch 2/20
306/334 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.5591 - loss: 2.4235
Epoch 2: val_loss improved from 3.82549 to 2.52982, saving model to saved_
models/weights.best.VGG16.keras
334/334 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.5620 - loss: 2.4064
- val_accuracy: 0.5641 - val_loss: 2.5298
Epoch 3/20
305/334 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.7278 - loss: 1.2861
Epoch 3: val_loss improved from 2.52982 to 2.20099, saving model to saved_
models/weights.best.VGG16.keras
334/334 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.7286 - loss: 1.2809
- val_accuracy: 0.6192 - val_loss: 2.2010
Epoch 4/20
328/334 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8196 - loss: 0.7666
Epoch 4: val_loss did not improve from 2.20099
334/334 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8194 - loss: 0.7676
- val_accuracy: 0.6072 - val_loss: 2.2595
Epoch 5/20
300/334 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8574 - loss: 0.5634
Epoch 5: val_loss improved from 2.20099 to 2.04345, saving model to saved_
models/weights.best.VGG16.keras
334/334 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8575 - loss: 0.5621
- val_accuracy: 0.6778 - val_loss: 2.0435
Epoch 6/20
302/334 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8939 - loss: 0.3960
Epoch 6: val_loss improved from 2.04345 to 1.88637, saving model to saved_
models/weights.best.VGG16.keras
334/334 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8938 - loss: 0.3973
- val_accuracy: 0.7006 - val_loss: 1.8864
Epoch 7/20
303/334 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9246 - loss: 0.2644
Epoch 7: val_loss did not improve from 1.88637
334/334 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9241 - loss: 0.2678
- val_accuracy: 0.6874 - val_loss: 1.9539
Epoch 8/20
291/334 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9430 - loss: 0.1921
Epoch 8: val_loss did not improve from 1.88637
334/334 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9425 - loss: 0.1965
- val_accuracy: 0.7162 - val_loss: 2.0211
Epoch 9/20
305/334 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9567 - loss: 0.1517
Epoch 9: val_loss improved from 1.88637 to 1.86545, saving model to saved_
models/weights.best.VGG16.keras
334/334 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9564 - loss: 0.1529
- val_accuracy: 0.7162 - val_loss: 1.8654
Epoch 10/20
296/334 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9648 - loss: 0.1117
Epoch 10: val_loss did not improve from 1.86545
334/334 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9644 - loss: 0.1140
- val_accuracy: 0.7066 - val_loss: 1.8807
Epoch 11/20
306/334 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9793 - loss: 0.0787
Epoch 11: val_loss did not improve from 1.86545
```

```
334/334 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9789 - loss: 0.0803
- val_accuracy: 0.7138 - val_loss: 1.8922
Epoch 12/20
307/334 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9812 - loss: 0.0643
Epoch 12: val_loss did not improve from 1.86545
334/334 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9809 - loss: 0.0655
- val_accuracy: 0.7174 - val_loss: 1.9668
Epoch 13/20
306/334 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9835 - loss: 0.0514
Epoch 13: val_loss did not improve from 1.86545
334/334 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9833 - loss: 0.0522
- val_accuracy: 0.7210 - val_loss: 1.9959
Epoch 14/20
305/334 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9887 - loss: 0.0341
Epoch 14: val_loss improved from 1.86545 to 1.84129, saving model to saved
_models/weights.best.VGG16.keras
334/334 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9883 - loss: 0.0354
- val_accuracy: 0.7305 - val_loss: 1.8413
Epoch 15/20
308/334 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9915 - loss: 0.0281
Epoch 15: val_loss did not improve from 1.84129
334/334 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9913 - loss: 0.0285
- val_accuracy: 0.7138 - val_loss: 2.0274
Epoch 16/20
292/334 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9913 - loss: 0.0256
Epoch 16: val_loss did not improve from 1.84129
334/334 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9912 - loss: 0.0259
- val_accuracy: 0.7305 - val_loss: 1.9642
Epoch 17/20
303/334 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9941 - loss: 0.0199
Epoch 17: val_loss did not improve from 1.84129
334/334 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9940 - loss: 0.0205
- val_accuracy: 0.7257 - val_loss: 1.9790
Epoch 18/20
327/334 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9960 - loss: 0.0124
Epoch 18: val_loss did not improve from 1.84129
334/334 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.9960 - loss: 0.0127
- val_accuracy: 0.7281 - val_loss: 1.9760
Epoch 19/20
305/334 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9971 - loss: 0.0112
Epoch 19: val_loss did not improve from 1.84129
334/334 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9970 - loss: 0.0117
- val_accuracy: 0.7281 - val_loss: 1.9630
Epoch 20/20
309/334 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9950 - loss: 0.0190
Epoch 20: val_loss did not improve from 1.84129
334/334 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9950 - loss: 0.0187
- val_accuracy: 0.7353 - val_loss: 1.9988
```

Out[41]: <keras.src.callbacks.history.History at 0x7daf5310e2d0>

Load the Model with the Best Validation Loss

In [42]:
```
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.

In [43]:
```
# 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.a
print('Test accuracy: %.4f%%' % test_accuracy)
```

```
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step
Test accuracy: 73.8038%
```

Predict Dog Breed with the Model

In [44]:
```
from extract_bottleneck_features import *

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']
```

In [45]:
```
### 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:

In [46]:
```
### TODO: Define your architecture.

Resnet50_model = Sequential()
Resnet50_model.add(GlobalAveragePooling2D(input_shape=train_Resnet50.shap
Resnet50_model.add(Dense(133, activation="softmax"))

Resnet50_model.summary()
```

Model: "sequential_3"

Layer (type)	Output Shape	P
global_average_pooling2d_1 (GlobalAveragePooling2D)	(None, 2048)	
dense_5 (Dense)	(None, 133)	2

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

In [47]:
```
### TODO: Compile the model.

Resnet50_model.compile(
    loss="categorical_crossentropy", optimizer="rmsprop", metrics=["accur
)
```

(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 [48]:
```
### TODO: Train the model.

checkpointer = ModelCheckpoint(
    filepath="saved_models/weights.best.Resnet50.keras", verbose=1, save_
)

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

```
Epoch 1/20
315/334 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.4077 - loss: 2.7815
Epoch 1: val_loss improved from inf to 0.75829, saving model to saved_mode
ls/weights.best.Resnet50.keras
334/334 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.4190 - loss: 2.7128
- val_accuracy: 0.7617 - val_loss: 0.7583
Epoch 2/20
314/334 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8742 - loss: 0.4239
Epoch 2: val_loss improved from 0.75829 to 0.70030, saving model to saved_
models/weights.best.Resnet50.keras
334/334 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8738 - loss: 0.4248
- val_accuracy: 0.7952 - val_loss: 0.7003
Epoch 3/20
320/334 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9182 - loss: 0.2615
Epoch 3: val_loss improved from 0.70030 to 0.69773, saving model to saved_
models/weights.best.Resnet50.keras
334/334 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9183 - loss: 0.2613
- val_accuracy: 0.7820 - val_loss: 0.6977
Epoch 4/20
321/334 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9517 - loss: 0.1557
Epoch 4: val_loss improved from 0.69773 to 0.63475, saving model to saved_
models/weights.best.Resnet50.keras
334/334 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9516 - loss: 0.1564
- val_accuracy: 0.8084 - val_loss: 0.6347
Epoch 5/20
319/334 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9712 - loss: 0.0969
Epoch 5: val_loss did not improve from 0.63475
334/334 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9710 - loss: 0.0977
- val_accuracy: 0.8156 - val_loss: 0.6556
Epoch 6/20
319/334 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9806 - loss: 0.0725
Epoch 6: val_loss did not improve from 0.63475
334/334 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9806 - loss: 0.0727
- val_accuracy: 0.8180 - val_loss: 0.6529
Epoch 7/20
316/334 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9880 - loss: 0.0496
Epoch 7: val_loss improved from 0.63475 to 0.60547, saving model to saved_
models/weights.best.Resnet50.keras
334/334 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9878 - loss: 0.0501
- val_accuracy: 0.8287 - val_loss: 0.6055
Epoch 8/20
320/334 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9914 - loss: 0.0373
Epoch 8: val_loss did not improve from 0.60547
334/334 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9914 - loss: 0.0375
- val_accuracy: 0.8335 - val_loss: 0.6428
Epoch 9/20
323/334 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9955 - loss: 0.0225
Epoch 9: val_loss did not improve from 0.60547
334/334 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9955 - loss: 0.0226
- val_accuracy: 0.8311 - val_loss: 0.6394
Epoch 10/20
319/334 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9961 - loss: 0.0189
Epoch 10: val_loss did not improve from 0.60547
334/334 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9961 - loss: 0.0190
- val_accuracy: 0.8287 - val_loss: 0.6265
Epoch 11/20
322/334 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9986 - loss: 0.0119
Epoch 11: val_loss did not improve from 0.60547
334/334 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9986 - loss: 0.0121
- val_accuracy: 0.8287 - val_loss: 0.6516
```

```
Epoch 12/20
308/334 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9977 - loss: 0.0119
Epoch 12: val_loss did not improve from 0.60547
334/334 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9976 - loss: 0.0119
- val_accuracy: 0.8311 - val_loss: 0.6541
Epoch 13/20
320/334 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9984 - loss: 0.0074
Epoch 13: val_loss did not improve from 0.60547
334/334 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9984 - loss: 0.0074
- val_accuracy: 0.8371 - val_loss: 0.6363
Epoch 14/20
316/334 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9993 - loss: 0.0049
Epoch 14: val_loss did not improve from 0.60547
334/334 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9992 - loss: 0.0050
- val_accuracy: 0.8180 - val_loss: 0.6773
Epoch 15/20
323/334 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9979 - loss: 0.0084
Epoch 15: val_loss did not improve from 0.60547
334/334 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9980 - loss: 0.0084
- val_accuracy: 0.8407 - val_loss: 0.6265
Epoch 16/20
310/334 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9988 - loss: 0.0055
Epoch 16: val_loss did not improve from 0.60547
334/334 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9987 - loss: 0.0056
- val_accuracy: 0.8371 - val_loss: 0.6370
Epoch 17/20
320/334 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9994 - loss: 0.0041
Epoch 17: val_loss did not improve from 0.60547
334/334 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9994 - loss: 0.0042
- val_accuracy: 0.8419 - val_loss: 0.6569
Epoch 18/20
318/334 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9994 - loss: 0.0041
Epoch 18: val_loss did not improve from 0.60547
334/334 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9994 - loss: 0.0042
- val_accuracy: 0.8419 - val_loss: 0.6545
Epoch 19/20
322/334 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9985 - loss: 0.0097
Epoch 19: val_loss did not improve from 0.60547
334/334 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9985 - loss: 0.0095
- val_accuracy: 0.8515 - val_loss: 0.6472
Epoch 20/20
317/334 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9987 - loss: 0.0038
Epoch 20: val_loss did not improve from 0.60547
334/334 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9987 - loss: 0.0038
- val_accuracy: 0.8431 - val_loss: 0.6680
```

Out[48]: `<keras.src.callbacks.history.History at 0x7daf423ea550>`

(IMPLEMENTATION) Load the Model with the Best Validation Loss

In [49]:
```
### 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%.

In [50]:
```
### TODO: Calculate classification accuracy on the test dataset.
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 1ms/step
Test accuracy: 83.0144%
```

(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 [51]:
```
### 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 BestResnet50_predict_labels(img_path):
    img = image.load_img(img_path, target_size=(224, 224))
    img_array = keras.utils.img_to_array(img)
    img_array = np.expand_dims(img_array, axis=0)
    img_array = preprocess_input(img_array)
    predictions = ResNet50_model.predict(img_array, verbose=0)

    decoded_predictions = decode_predictions(predictions, top=1)[0]

    return decoded_predictions[0][1]

def BestResnet50_dog_detector(img_path):
    predicted_class = ResNet50_predict_labels(img_path)

    return any(keyword in predicted_class.lower() for keyword in dog_keyw
```

```
print(
    "Label Beagle_01155.jpg:",
    BestResnet50_predict_labels("dogImages/test/016.Beagle/Beagle_01155.j
)

print(
    "Face Detector Beagle_01155.jpg: ",
    face_detector("dogImages/test/016.Beagle/Beagle_01155.jpg")[0],
)

print("Face Detector John_Travolta_0006.jpg: ", face_detector("lfw/John_T

print(
    "BestResnet50_dog_detector Beagle_01155.jpg:",
    BestResnet50_dog_detector("dogImages/test/016.Beagle/Beagle_01155.jpg
)
print(
    "BestResnet50_dog_detector John_Travolta_0006.jpg:",
    BestResnet50_dog_detector("lfw/John_Travolta/John_Travolta_0006.jpg")
)
```

```
Label Beagle_01155.jpg: beagle
Face Detector Beagle_01155.jpg:  False
Face Detector John_Travolta_0006.jpg:  True
BestResnet50_dog_detector Beagle_01155.jpg: True
BestResnet50_dog_detector John_Travolta_0006.jpg: False
```

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!

This photo looks like an Afghan Hound.

(IMPLEMENTATION) Write your Algorithm

In [52]:
```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.

def my_algorithm(img_path):
    if BestResnet50_dog_detector(img_path):
        return BestResnet50_predict_labels(img_path)
    if face_detector(img_path)[0]:
        return 'human'
    return 'neither'

print(
    "my_algorithm Beagle_01155.jpg:",
    my_algorithm("dogImages/test/016.Beagle/Beagle_01155.jpg"),
)
print(
    "my_algorithm John_Travolta_0006.jpg:",
    my_algorithm("lfw/John_Travolta/John_Travolta_0006.jpg"),
)
```

```
my_algorithm Beagle_01155.jpg: beagle
my_algorithm John_Travolta_0006.jpg: human
```

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!

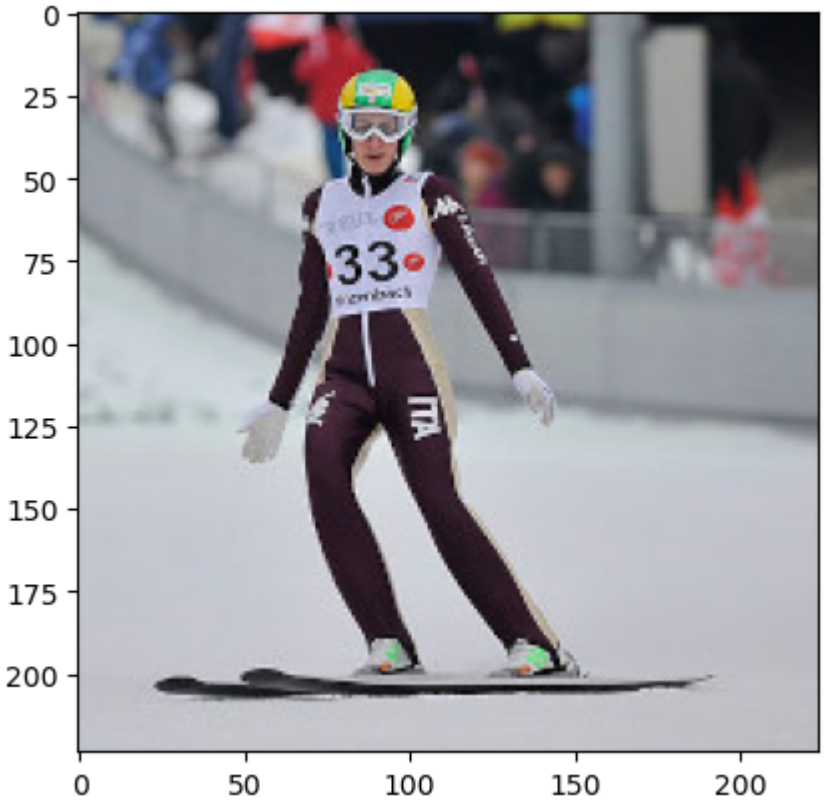
Test your algorithm at least six images on your computer. Feel free to use any images you like. Use at least two human and two dog images.

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

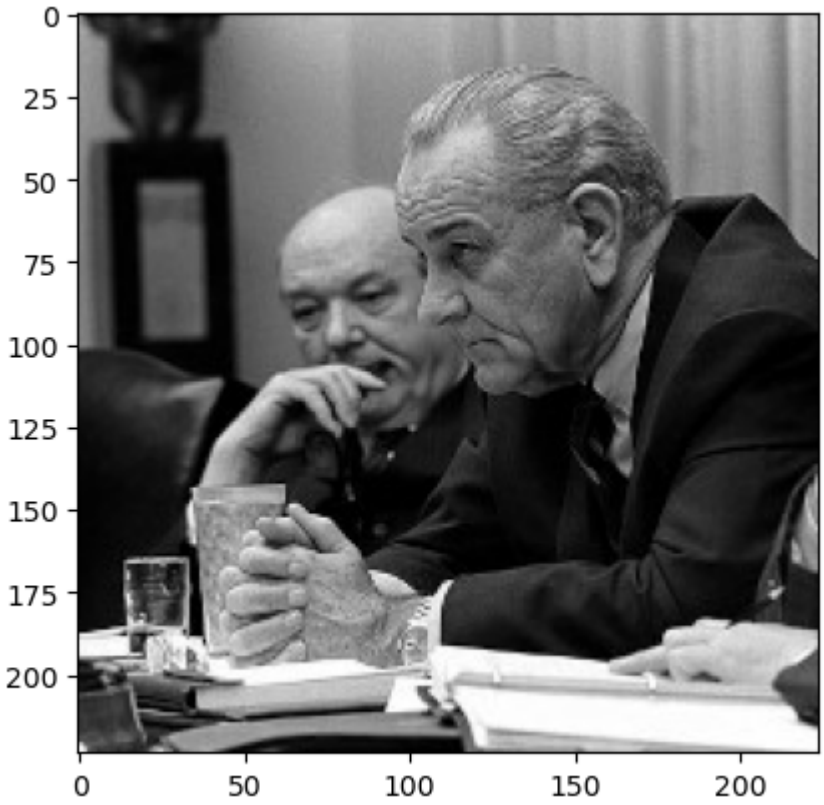
Answer:

In [53]:
```
## TODO: Execute your algorithm from Step 6 on
## at least 6 images on your computer.
## Feel free to use as many code cells as needed.

def test_my_algorithm(img_file):
    img_path = "sampleImages/" + img_file
    print(img_path, " => ", my_algorithm(img_path))
    img = cv2.imread(img_path)
    img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    plt.imshow(img_rgb)
    plt.show()

image_files = [
    "20140202_Hinzenbach_Evelyn_Insam_2389.jpg",
    "640px-Dean_Rusk,_Lyndon_B._Johnson_and_Robert_McNamara_in_Cabinet_Ro
    "640px-Football_player_before_throw.jpg",
    "640px-Riding_the_Plasma_Wave_-_Flickr_-_NASA_Goddard_Photo_and_Video
    "640px-Sally_Ride_(1984).jpg",
    "640px-Vitoria_-_Parque_de_Olárizu_-_Niebla_y_cencellada_-BT-_01.jpg"
    "Aminah_Cendrakasih,_c._1959,_by_Tati_Photo_Studio.jpg",
    "Campamento_de_ganado_de_la_tribu_Mundari,_Terekeka,_Sudán_del_Sur,_2
    "Canis_lupus_familiaris.002_-_Monfero.jpg",
    "Canis_lupus_familiaris,_Neuss_(DE)_--_2024_--_0085.jpg",
    "Flickr_cc_runner_wisconsin_u.jpg",
    "Greenland_467_(35130903436)_(cropped).jpg",
    "Liver_yellow_dog_in_the_water_looking_at_viewer_at_golden_hour_in_Do
    "Mexican_Sunflower_Tithonia_rotundifolia_Flower_2163px.jpg",
    "MotorCycle.jpg",
    "RobotDragon.jpg",
    "Sikh_man,_Agra_10.jpg",
    "Vitruvian.jpg",
    "Woman_with_photo,_Afghanistan.jpg",
    "York_Minster_Chapter_House_Ceiling.jpg",
]

for image_file in image_files:
    test_my_algorithm(image_file)
```

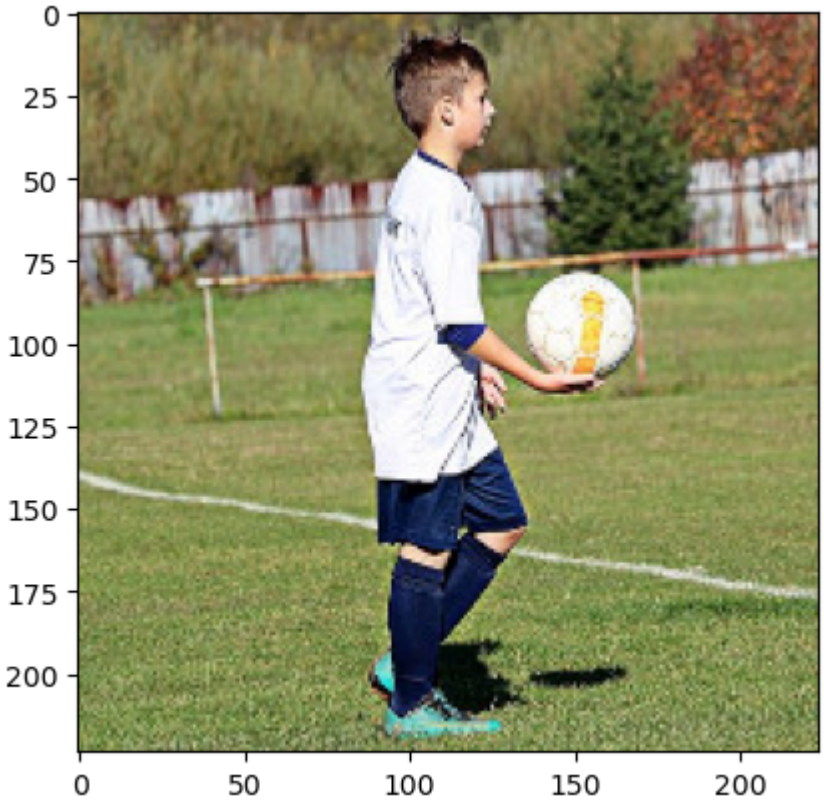
```
sampleImages/20140202_Hinzenbach_Evelyn_Insam_2389.jpg  =>  neither
```



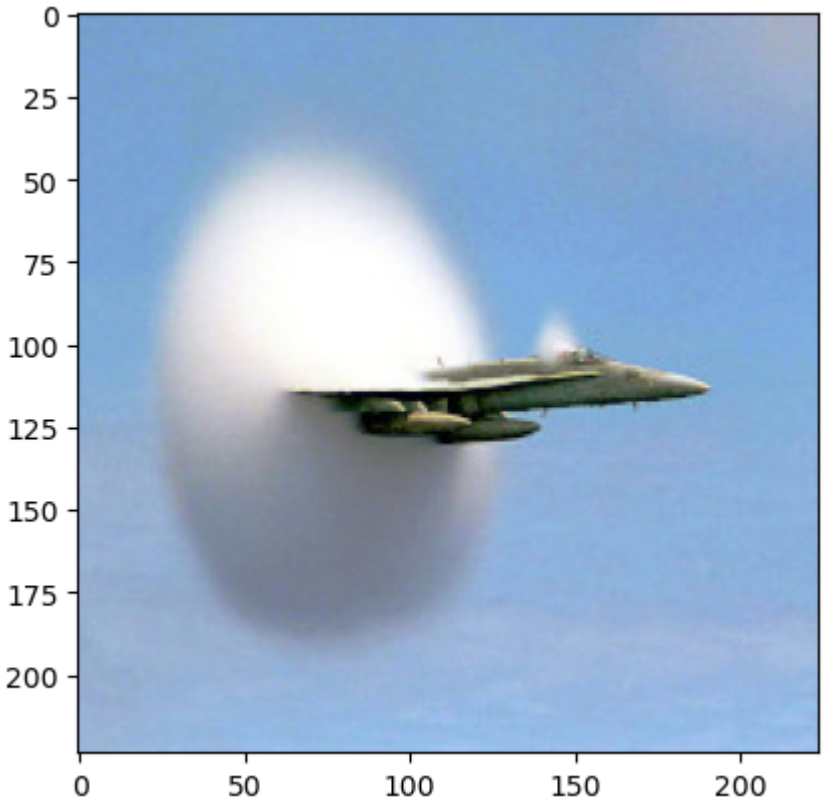

```
sampleImages/640px-Dean_Rusk,_Lyndon_B._Johnson_and_Robert_McNamara_in_Cab
inet_Room_meeting_February_1968.jpg  =>  human
```



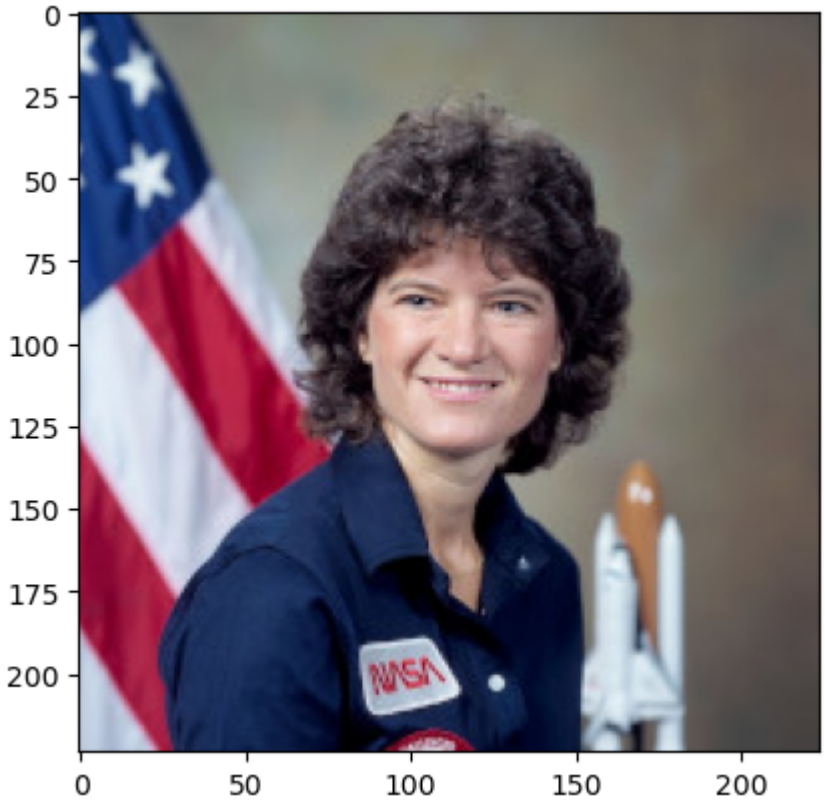

```
sampleImages/640px-Football_player_before_throw.jpg  =>  neither
```



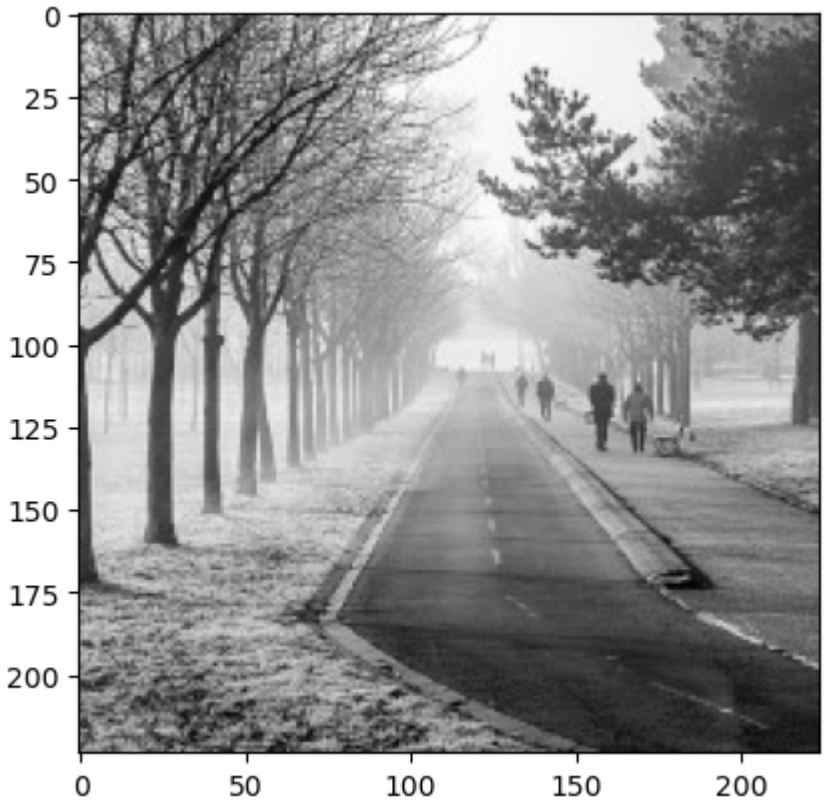

```
sampleImages/640px-Riding_the_Plasma_Wave_-_Flickr_-_NASA_Goddard_Photo_an
d_Video.jpg  =>  neither
```



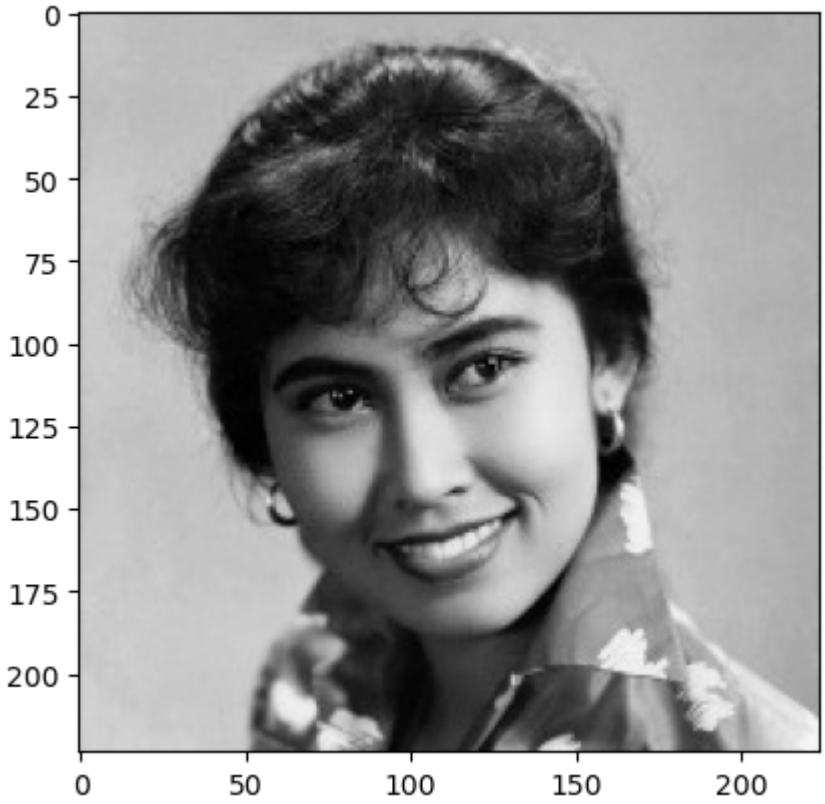

```
sampleImages/640px-Sally_Ride_(1984).jpg  =>  human
```



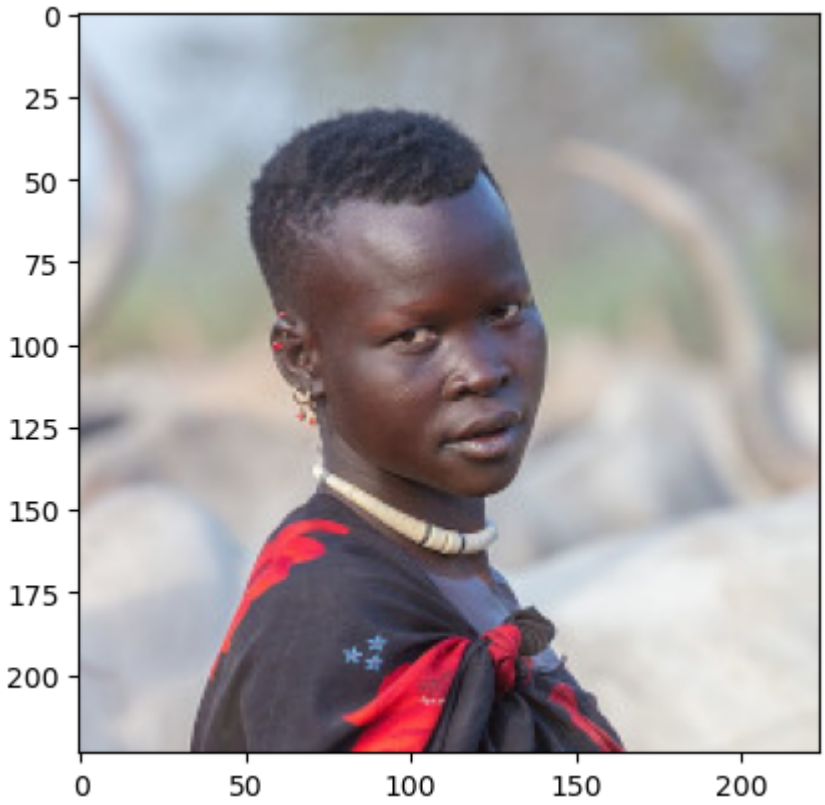

```
sampleImages/640px-Vitoria_-_Parque_de_Olárizu_-_Niebla_y_cencellada_-BT-_
01.jpg  =>  neither
```



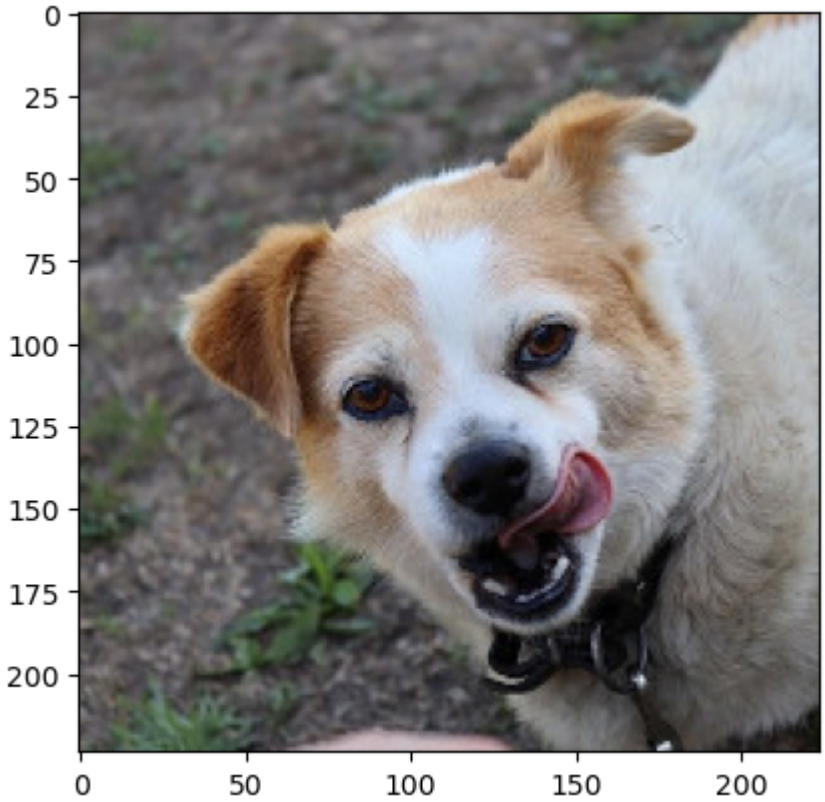

```
sampleImages/Aminah_Cendrakasih,_c._1959,_by_Tati_Photo_Studio.jpg  =>  hu
man
```



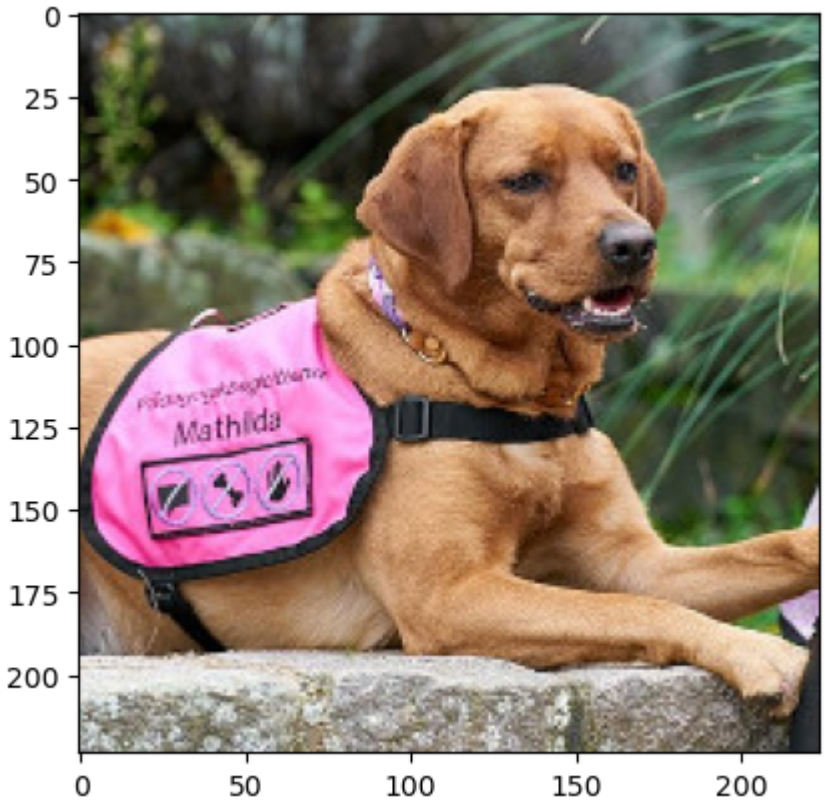

```
sampleImages/Campamento_de_ganado_de_la_tribu_Mundari,_Terekeka,_Sudán_del
_Sur,_2024-01-28,_DD_157.jpg  =>  neither
```



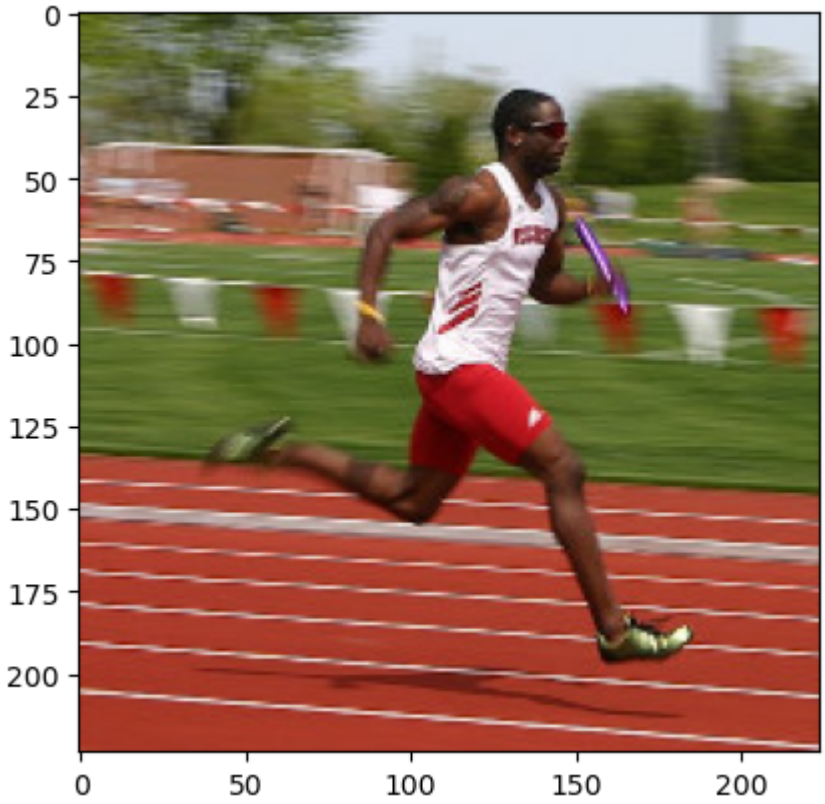

```
sampleImages/Canis_lupus_familiaris.002_-_Monfero.jpg  =>  Chihuahua
```



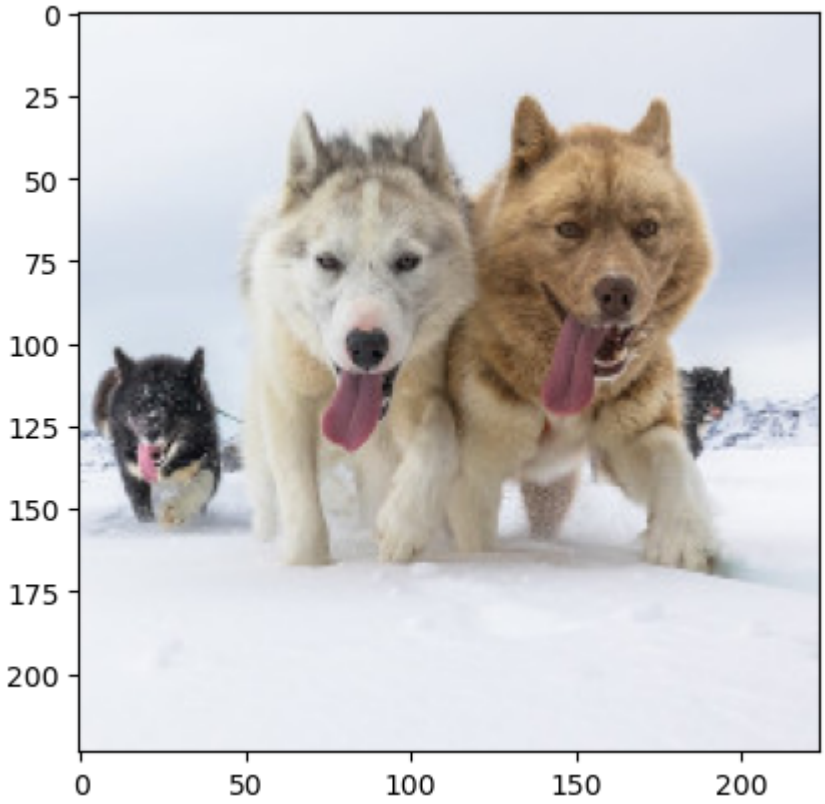

```
sampleImages/Canis_lupus_familiaris,_Neuss_(DE)_--_2024_--_0085.jpg  =>  v
izsla
```



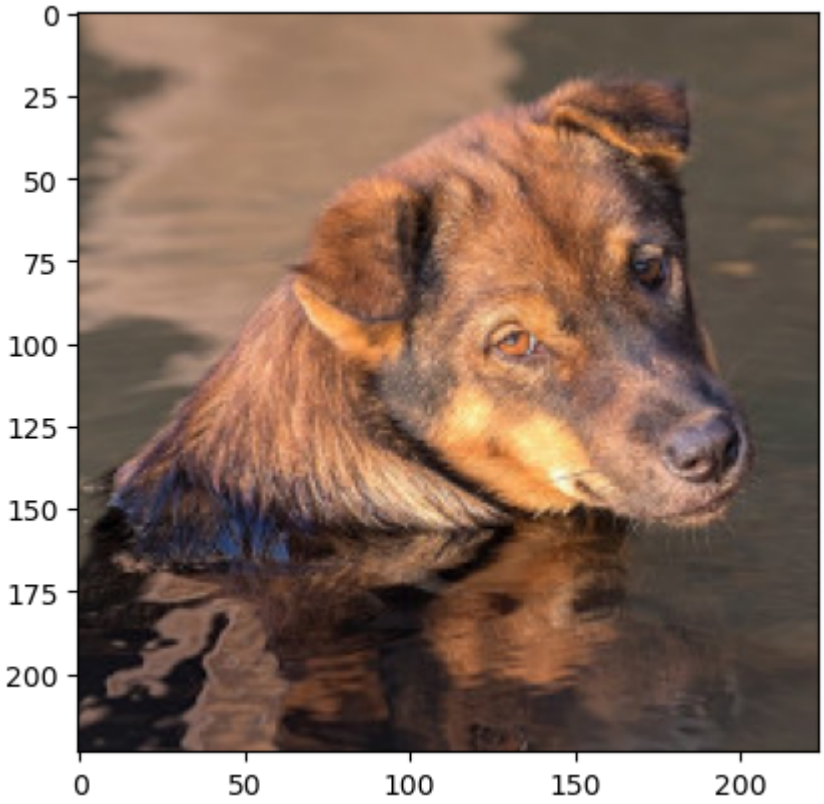

```
sampleImages/Flickr_cc_runner_wisconsin_u.jpg  =>  neither
```



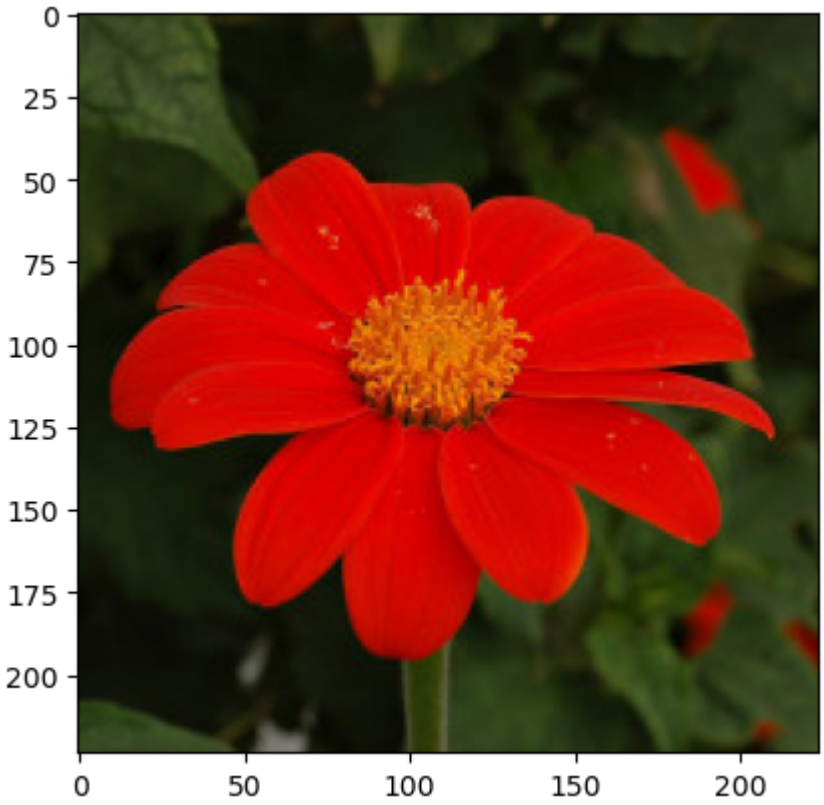

```
sampleImages/Greenland_467_(35130903436)_(cropped).jpg  =>  neither
```



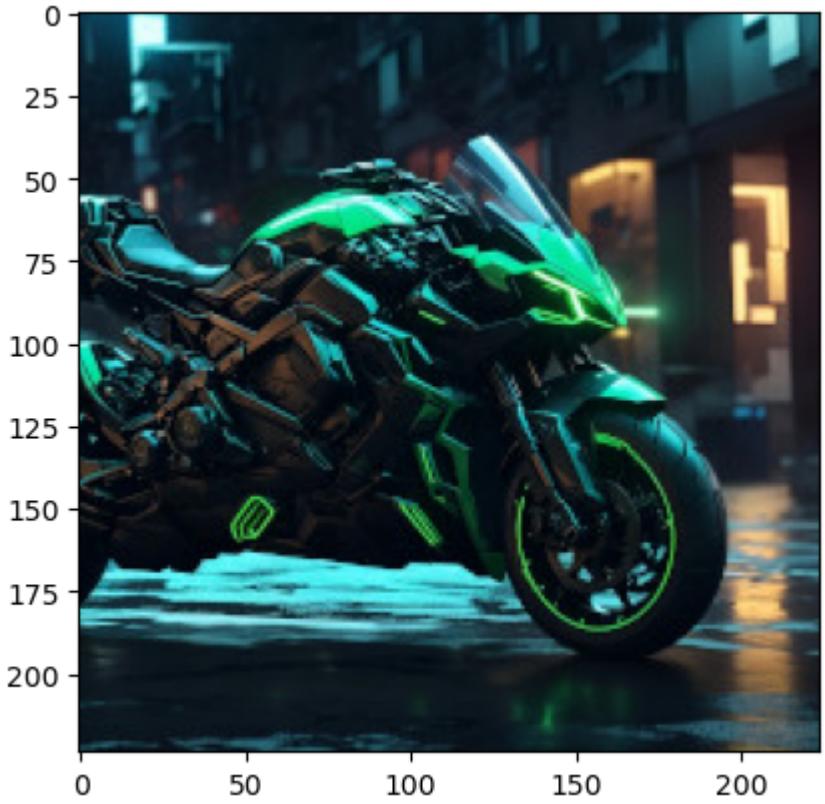

```
sampleImages/Liver_yellow_dog_in_the_water_looking_at_viewer_at_golden_hou
r_in_Don_Det_Laos.jpg  =>  kelpie
```



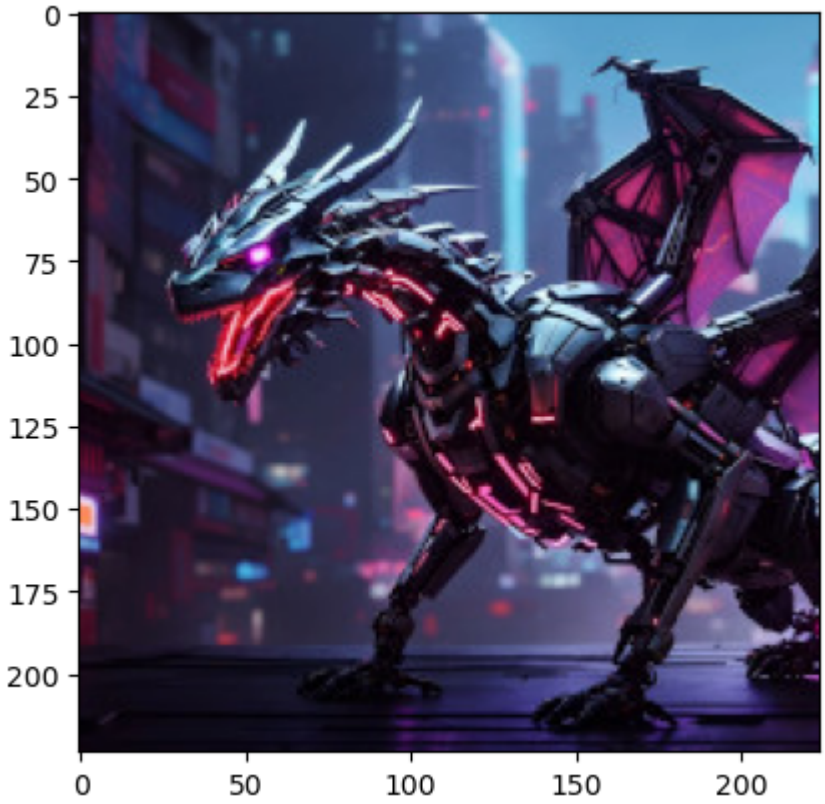

```
sampleImages/Mexican_Sunflower_Tithonia_rotundifolia_Flower_2163px.jpg  =>
neither
```



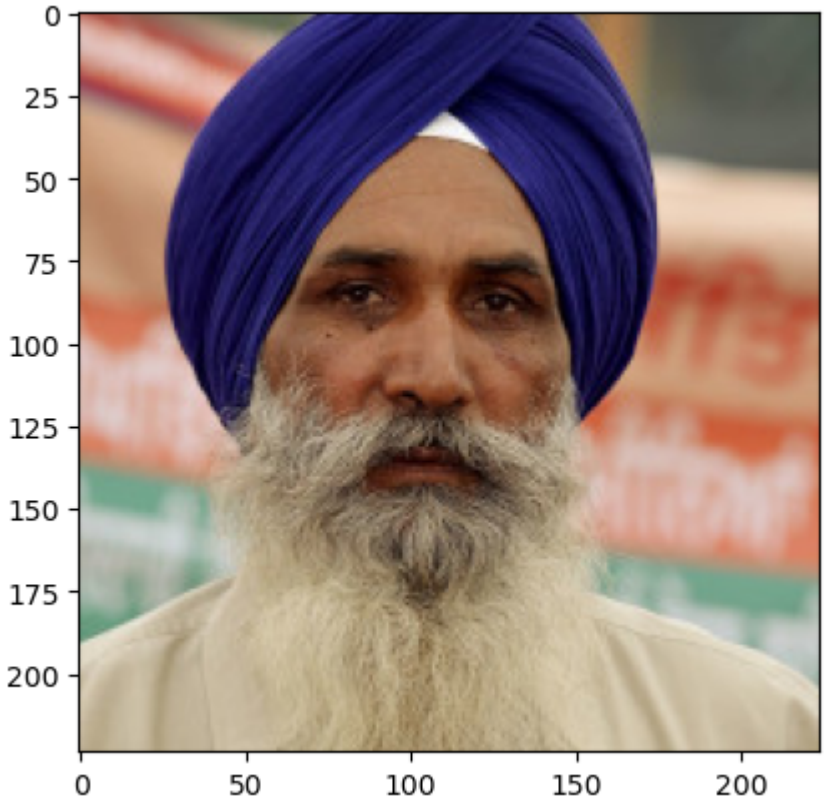

```
sampleImages/MotorCycle.jpg  =>  neither
```



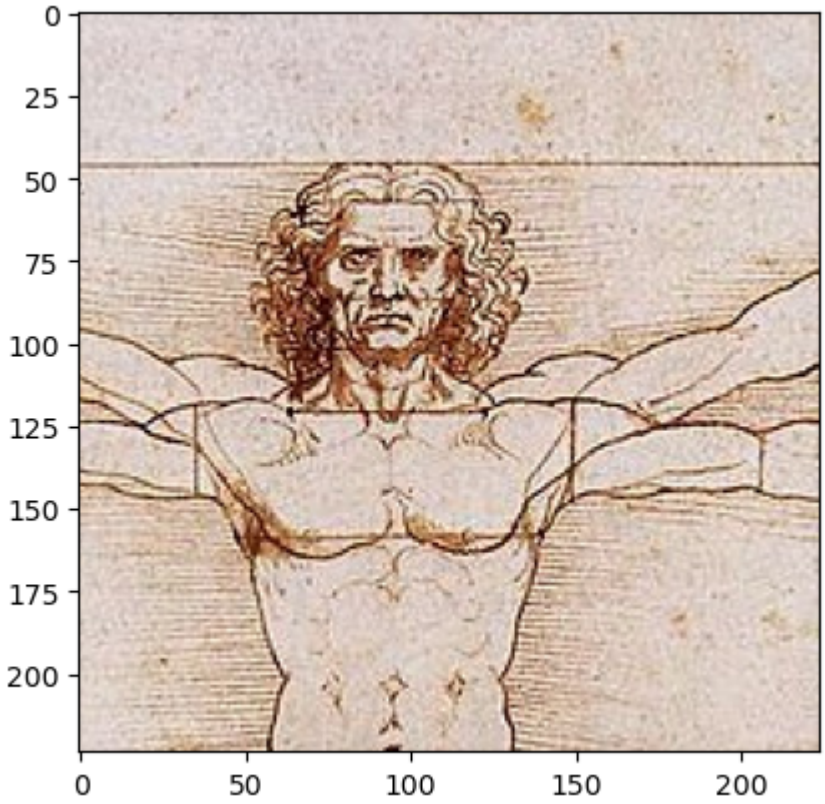

```
sampleImages/RobotDragon.jpg  =>  neither
```



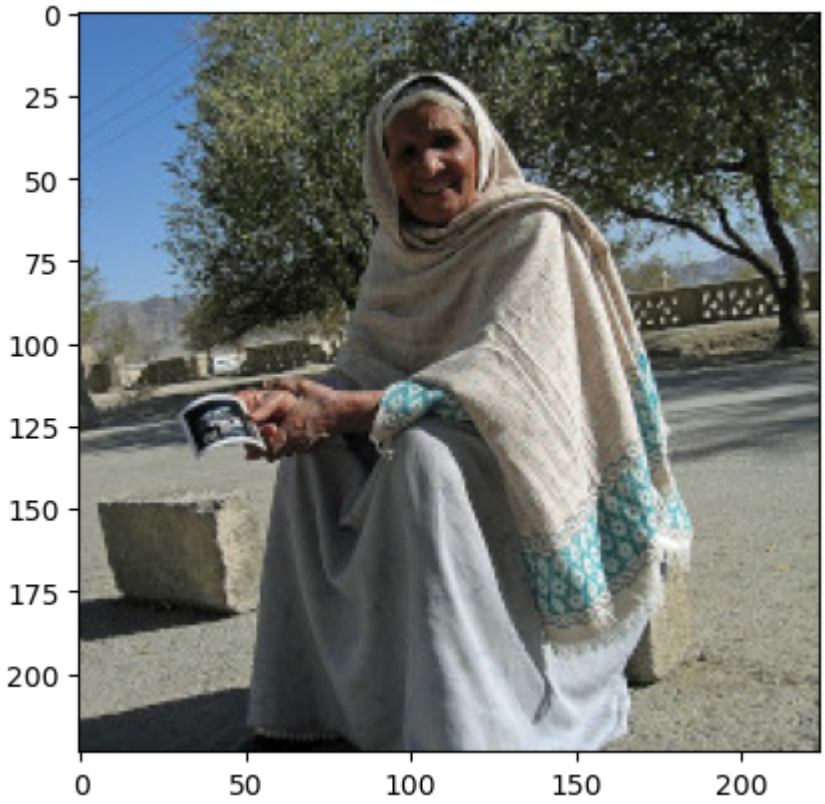

```
sampleImages/Sikh_man,_Agra_10.jpg  =>  human
```



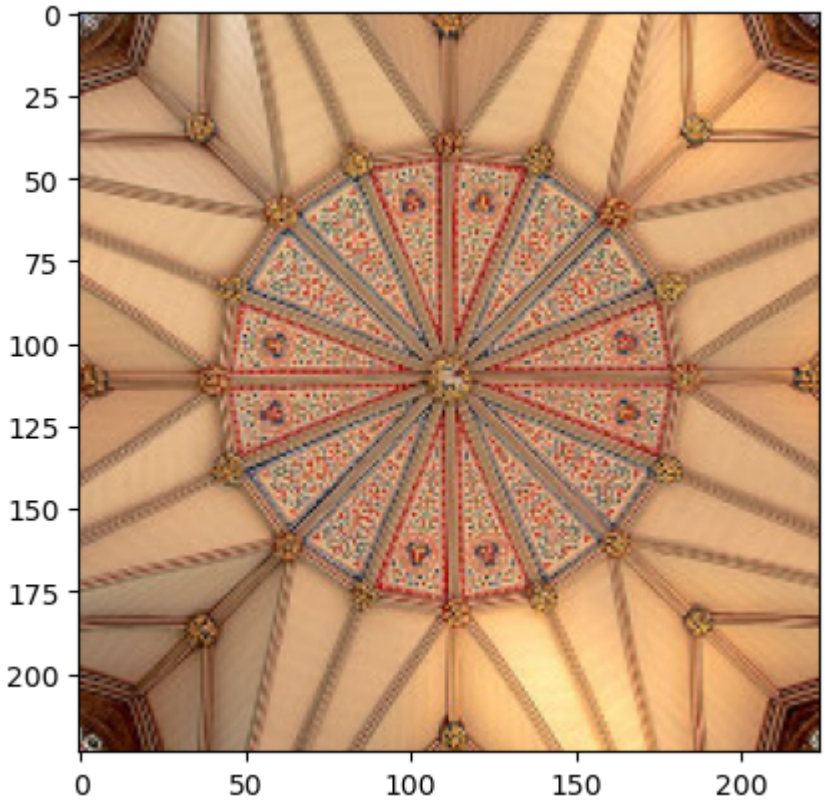

```
sampleImages/Vitruvian.jpg  =>  human
```




```
sampleImages/Woman_with_photo,_Afghanistan.jpg  =>  neither
```




```
sampleImages/York_Minster_Chapter_House_Ceiling.jpg  =>  neither
```



Analysis of the Human/Dog/Neither Detector

The Human/Dog/Neither detector demonstrates promising results, exceeding initial expectations. The algorithm employs a Haar feature-based cascade classifier (frontal face) for human detection and an enhanced ResNet50 model for dog identification, establishing a robust framework for multi-class image classification.

Dog Detection Accuracy

The detector performs well in identifying dog images. All canine images in the sample set were correctly classified, highlighting the improved ResNet50 model's effectiveness in this task. While the algorithm did miss one image containing two dogs, overall, its high-level accuracy is excellent.

Human Detection Performance

Human detection relies on the Haar feature-based cascade classifier (frontal face). Given that the category of human images is broader than the frontal face images used to train the detector, more errors are expected. The "Neither" classification shows promise but is impacted by errors in human misdetection.

Areas for Improvement

Human Detection Accuracy

The mixed results in human image detection suggest a need for improvement. Using a specific human detector rather than a frontal face detector could enhance accuracy.

Handling Ambiguous Cases

Improving human detection will subsequently enhance the accuracy of "Neither" image classification.

Training Data Diversity

The frontal face detector missed images featuring clear frontal human faces, likely due to training data bias. Overrepresenting certain ethnicities, ages, or facial features in the training set can skew results. Haar cascades, which rely on contrast differences, may underperform on darker skin tones. While Haar cascade classifiers are valued for their speed and efficiency, modern deep learning-based approaches offer better performance and can be trained to mitigate various biases.