dog_app

March 19, 2019

1 Convolutional Neural Networks

1.1 Project: Write an Algorithm for a Dog Identification App

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!

> **Note**: Once you have completed all of the code implementations, you need to finalize your work by exporting the Jupyter Notebook as an HTML document. Before exporting the notebook to html, all of the code cells need to have been run so that reviewers can see the final implementation and output. You can then export the notebook by using the menu above and navigating to **File -> Download as -> HTML (.html)**. Include the finished document along with this notebook as your submission.

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 Jupyter notebook.
Step 0: Import Datasets
Make sure that you've downloaded the required human and dog datasets: * Download the dog dataset. Unzip the folder and place it in this project's home directory, at the location `/dogImages`.

- Download the human dataset. Unzip the folder and place it in the home directory, at location `/lfw`.

Note: If you are using a Windows machine, you are encouraged to use 7zip to extract the folder.

In the code cell below, we save the file paths for both the human (LFW) dataset and dog dataset in the numpy arrays `human_files` and `dog_files`.

```
In [1]: import numpy as np
        from glob import glob

        # load filenames for human and dog images
        human_files = np.array(glob("/data/lfw/*/*"))
        dog_files = np.array(glob("/data/dog_images/*/*/*"))

        # print number of images in each dataset
        print('There are %d total human images.' % len(human_files))
        print('There are %d total dog images.' % len(dog_files))
        # import numpy as np
        # from glob import glob

        # # load filenames for human and dog images
        # human_files = np.array(glob("lfw/*/*"))
        # dog_files = np.array(glob("dogImages/*/*/*"))

        # # print number of images in each dataset
        # print('There are %d total human images.' % len(human_files))
        # print('There are %d total dog images.' % len(dog_files))
```

```
There are 13233 total human images.
There are 8351 total dog images.
```

Step 1: Detect Humans

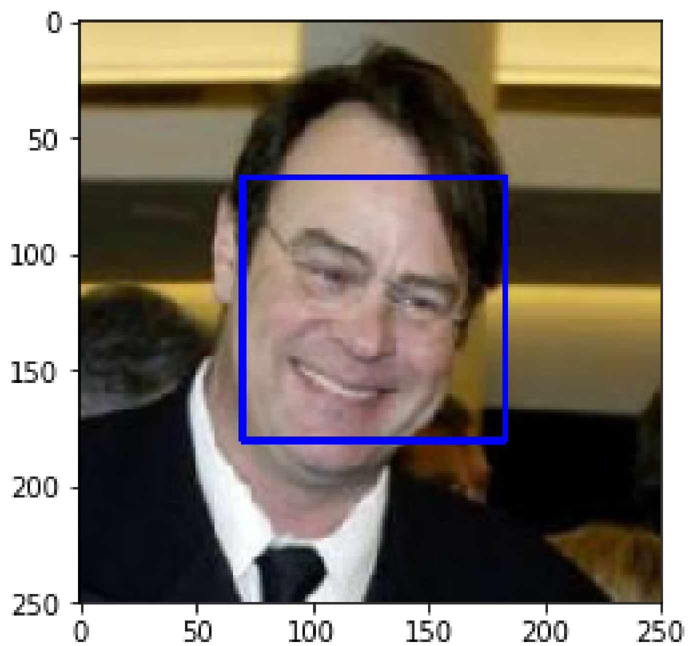
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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 [2]: import cv2
        import matplotlib.pyplot as plt
        %matplotlib inline

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

        # load color (BGR) image
        img = cv2.imread(human_files[0])
        # 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.

1.1.1 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 [3]: # 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
```

1.1.2 (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: (You can print out your results and/or write your percentages in this cell)

```
In [4]: from tqdm import tqdm

        human_files_short = human_files[:100]
        dog_files_short = dog_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.

        # Test human image with detected human faces
        human_files_test = [face_detector(human_i) for human_i in human_files_short]
        print('The percentage of first 100 human images with a detected human face is {}%.'
              .format(human_files_test.count(True)/len(human_files_short)*100))

        # Test dog image with detected human faces
        dog_files_test = [face_detector(dog_i) for dog_i in dog_files_short]
        print('The percentage of first 100 dog images with a detected human face is {}%.'
              .format(dog_files_test.count(True)/len(dog_files_short)*100))
```

```
The percentage of first 100 human images with a detected human face is 98.0%.
The percentage of first 100 dog images with a detected human face is 17.0%.
```

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 `human_files_short` and `dog_files_short`.

```
In [5]: ### (Optional)
        ### TODO: Test performance of another face detection algorithm.
        ### Feel free to use as many code cells as needed.

        ### other face detector include DNN in OpenCV
```

Step 2: Detect Dogs
In this section, we use a pre-trained model to detect dogs in images.

1.1.3 Obtain Pre-trained VGG-16 Model

The code cell below downloads the VGG-16 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.

```
In [6]: import torch
        import torchvision.models as models

        # define VGG16 model
        VGG16 = models.vgg16(pretrained=True)

        # check if CUDA is available
        use_cuda = torch.cuda.is_available()

        # move model to GPU if CUDA is available
        if use_cuda:
            VGG16 = VGG16.cuda()
```

Given an image, this pre-trained VGG-16 model returns a prediction (derived from the 1000 possible categories in ImageNet) for the object that is contained in the image.

1.1.4 (IMPLEMENTATION) Making Predictions with a Pre-trained Model

In the next code cell, you will write a function that accepts a path to an image (such as `'dogImages/train/001.Affenpinscher/Affenpinscher_00001.jpg'`) as input and returns the index corresponding to the ImageNet class that is predicted by the pre-trained VGG-16 model. The output should always be an integer between 0 and 999, inclusive.

Before writing the function, make sure that you take the time to learn how to appropriately pre-process tensors for pre-trained models in the PyTorch documentation.

```
In [7]: from PIL import Image
        import torchvision.transforms as transforms
```

```
# Set PIL to be tolerant of image files that are truncated.
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

def VGG16_predict(img_path):
    '''
    Use pre-trained VGG-16 model to obtain index corresponding to
    predicted ImageNet class for image at specified path

    Args:
        img_path: path to an image

    Returns:
        Index corresponding to VGG-16 model's prediction
    '''

    ## TODO: Complete the function.
    ## Load and pre-process an image from the given img_path
    ## Return the *index* of the predicted class for that image
    image = Image.open(img_path)

#     # train:
#     transform = transforms.Compose([
#         RandomResizedCrop(224),
#         transforms.ToTensor(),
#         transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])])

    #### eval
    image_transform = transforms.Compose([
        transforms.Resize(256),      ## from transfer_learning_exercise, vgg16 takes 224
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])]) ##

    image_tensor = image_transform(image)
    # check if use GPU and transfer tensor to cuba
    if use_cuda:
        image_tensor = image_tensor.cuda()

    # 4d tensor [batch_size, channels, height, width]
    image_tensor = image_tensor.unsqueeze(0)

    #VGG16.eval() # eval mode for testing

    output = VGG16(image_tensor)
    predicted_class = output.data.argmax(dim=1)
    return predicted_class.item() # predicted class index
```

1.1.5 (IMPLEMENTATION) 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 VGG-16 model, we need only check if the pre-trained model predicts an index between 151 and 268 (inclusive).

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 [8]: ### returns "True" if a dog is detected in the image stored at img_path
        def dog_detector(img_path):
            ## TODO: Complete the function.
            index = VGG16_predict(img_path)
            if (index>=151) & (index<=268):
                return True
            else:
                return False # true/false
```

1.1.6 (IMPLEMENTATION) Assess the Dog Detector

Question 2: 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: - The percentage of the images in `human_files_short` have a detected dog is 0.0 %. - The percentage of the images in `dog_files_short` have a detected dog is 100.0 %.

```
In [9]: ### TODO: Test the performance of the dog_detector function
        ### on the images in human_files_short and dog_files_short.
        human_test_count = 0

        for human_i in tqdm(human_files_short):
            human_test_count += dog_detector(human_i)

        print('The percentage of the images in human_files_short have a detected dog is {} %'.fo

        ### dog_files_short contains a detected dog
        dog_test_count = 0

        for dog_i in tqdm(dog_files_short):
            dog_test_count += dog_detector(dog_i)

        print('The percentage of the images in dog_files_short have a detected dog is {} %'.form
```

```
100%|| 100/100 [00:03<00:00, 29.96it/s]
  3%|          | 3/100 [00:00<00:03, 26.57it/s]

The percentage of the images in human_files_short have a detected dog is 0.0 %
```

```
100%|| 100/100 [00:04<00:00, 20.47it/s]

The percentage of the images in dog_files_short have a detected dog is 100.0 %
```

We suggest VGG-16 as a potential network to detect dog images in your algorithm, but you are free to explore other pre-trained networks (such as Inception-v3, ResNet-50, etc). Please use the code cell below to test other pre-trained PyTorch models. If you decide to pursue this *optional* task, report performance on `human_files_short` and `dog_files_short`.

```
In [10]: ### (Optional)
         ### TODO: Report the performance of another pre-trained network.
         ### Feel free to use as many code cells as needed.

         ################################################################################
         ### Use the same function (function name changed) to perform dog detection with ResNet5
         ################################################################################

         resnet50 = models.resnet50(pretrained = True)

         if use_cuda:
             resnet50 = resnet50.cuda()

         def resnet50_predict(img_path):
             '''
             Use pre-trained VGG-16 model to obtain index corresponding to
             predicted ImageNet class for image at specified path

             Args:
                 img_path: path to an image

             Returns:
                 Index corresponding to VGG-16 model's prediction
             '''
             image = Image.open(img_path)
             #### eval
             image_transform = transforms.Compose([
                 transforms.Resize(256),
                 transforms.CenterCrop(224),    ## from transfer_learning_exercise, vgg16 takes 2
                 transforms.ToTensor(),
                 transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])]) #

             image_tensor = image_transform(image)
             # check if use GPU and transfer tensor to cuba
             if use_cuda:
                 image_tensor = image_tensor.cuda()
```

```python
        # 4d tensor [batch_size, channels, height, width]
        image_tensor = image_tensor.unsqueeze(0)

        resnet50.eval() # eval mode for testing

        output = resnet50(image_tensor)
        predicted_class = output.data.argmax(dim=1)
        return predicted_class.item() # predicted class index

    ### returns "True" if a dog is detected in the image stored at img_path
    def dog_detector_resnet50(img_path):
        ## TODO: Complete the function.
        index = resnet50_predict(img_path)
        if (index>=151) & (index<=268):
            return True
        else:
            return False # true/false

    print('If use ResNet50 model architecture instead of VGG16, the accuracy is tested belo
    human_test_count_resnet50 = 0

    for human_i in tqdm(human_files_short):
        human_test_count_resnet50 += dog_detector_resnet50(human_i)

    print('The percentage of the images in human_files_short have a detected dog is {} %'
          .format(human_test_count_resnet50/len(human_files_short)*100))

    ### dog_files_short contains a detected dog
    dog_test_count_resnet50 = 0

    for dog_i in tqdm(dog_files_short):
        dog_test_count_resnet50 += dog_detector_resnet50(dog_i)

    print('The percentage of the images in dog_files_short have a detected dog is {} %'
          .format(dog_test_count_resnet50/len(dog_files_short)*100))
```

```
  5%|         | 5/100 [00:00<00:02, 43.69it/s]

If use ResNet50 model architecture instead of VGG16, the accuracy is tested below:


100%|| 100/100 [00:02<00:00, 45.35it/s]
  0%|          | 0/100 [00:00<?, ?it/s]

The percentage of the images in human_files_short have a detected dog is 0.0 %


100%|| 100/100 [00:03<00:00, 25.96it/s]
```

```
The percentage of the images in dog_files_short have a detected dog is 100.0 %
```

Optional Answer:
If I use `ResNet50` model architecture instead of `vgg16`, the performance is tested below:

- The percentage of the images in `human_files_short` have a detected dog is 0.0 %.
- The percentage of the images in `dog_files_short` have a detected dog is 100.0 %.

They have quite similiar performance, but `ResNet50` is slightly faster than `vgg16`.

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 10%. In Step 4 of this notebook, you will have the opportunity to use transfer learning to create a CNN that attains greatly improved accuracy.

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 trouble 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!

1.1.7 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

Use the code cell below to write three separate data loaders for the training, validation, and test datasets of dog images (located at `dogImages/train`, `dogImages/valid`, and `dogImages/test`, respectively). You may find this documentation on custom datasets to be a useful resource. If you are interested in augmenting your training and/or validation data, check out the wide variety of transforms!

```
In [11]: import os
         from torchvision import datasets

         ### TODO: Write data loaders for training, validation, and test sets
         data_dir = '/data/dog_images/'
         train_dir = os.path.join(data_dir, 'train/')
         valid_dir = os.path.join(data_dir, 'valid/')
         test_dir = os.path.join(data_dir, 'test/')

         # number of subprocesses to use for data loading
         num_workers = 0
         # how many samples per batch to load
         batch_size = 20

         ## Specify appropriate transforms, and batch_sizes
         train_transform = transforms.Compose([transforms.RandomResizedCrop(224),
                                               transforms.RandomHorizontalFlip(),
                                               transforms.RandomRotation(10),
                                               transforms.ToTensor(),
                                               transforms.Normalize(mean=[0.485, 0.456, 0.406],

         valid_test_transform = transforms.Compose([transforms.Resize(256),  ## from transfer_le
                                                    transforms.CenterCrop(224),
                                                    transforms.ToTensor(),
                                                    transforms.Normalize(mean=[0.485, 0.456, 0.4

         train_data = datasets.ImageFolder(train_dir, transform = train_transform)
         valid_data = datasets.ImageFolder(valid_dir, transform = valid_test_transform)
         test_data = datasets.ImageFolder(test_dir, transform = valid_test_transform)

         train_loader = torch.utils.data.DataLoader(train_data, batch_size = batch_size, num_wor
         valid_loader = torch.utils.data.DataLoader(valid_data, batch_size = batch_size, num_wor
         test_loader = torch.utils.data.DataLoader(test_data, batch_size = batch_size, num_worke


         ###
         loaders_scratch = {'train': train_loader,
                            'valid': valid_loader,
```

```
'test': test_loader}
```

Question 3: Describe your chosen procedure for preprocessing the data. - How does your code resize the images (by cropping, stretching, etc)? What size did you pick for the input tensor, and why? - Did you decide to augment the dataset? If so, how (through translations, flips, rotations, etc)? If not, why not?

Answer:

1. The RandomResizedCrop crops the given image to random size (default between 0.08 and 1.0 of the original size) and aspect ratio (default between 0.75 and 1.3333 of the original aspect ratio). The technique resizes the given image to the given size afterward. The Resize and CenterCrop for validation and test dataset resizes the given image to `256x256` and crops the center `224x224` pixels. The transforms processes return `3x224x224` tensors. The `224x224` square images make the model easier to work with. `224*224` or `227*227` (AlexNet) seems to provide enough information of the given image. `224x224` or `227x227` was chosen by the AlexNet due to the augmentation techniques (i.e. translations, reflections) that they use.

2. I use RandomRotation(10) which rotate the given image by 10 degree, and RandomHorizontalFlip which horizontally flip the given image randomly with default 0.5 probability. Technically, the dataset augmentation can help avoid overfitting of the dataset, and this should be applied only to train dataset but not to the validation dataset.

1.1.8 (IMPLEMENTATION) Model Architecture

Create a CNN to classify dog breed. Use the template in the code cell below.

```
In [22]: import torch.nn as nn
         import torch.nn.functional as F

         # define the CNN architecture
         class Net(nn.Module):
             ### TODO: choose an architecture, and complete the class
             def __init__(self):
                 super(Net, self).__init__()
                 ## Define layers of a CNN
                 ## Inspired by cifar10 exercise
                 ## Convolution lyrs
                 self.conv1 = nn.Conv2d(3, 16, 3)
                 self.conv2 = nn.Conv2d(16, 32, 3)
                 self.conv3 = nn.Conv2d(32, 64, 3)
                 self.conv4 = nn.Conv2d(64, 128, 3)
                 self.conv5 = nn.Conv2d(128, 256, 3)

                 ## Pooling lyr
                 self.pool = nn.MaxPool2d(2, 2)

                 ## Linear lyr; use
                 self.fc1 = nn.Linear(256*5*5, 500)
```

```
        ## Output lyr - 133 classes
        self.fc2 = nn.Linear(500, len(test_data.classes))

        ## dropout lyr
        # self.dropout = nn.Dropout(0.2)

    def forward(self, x):
        ## Define forward behavior
        # sequence of convolutional lyr and maxpooling lyr
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        x = self.pool(F.relu(self.conv4(x)))
        x = self.pool(F.relu(self.conv5(x)))

        # dimension -> 256, 5, 5
        x = x.view(-1, 5*5*256)
        # dropout lyr and linear lyr
        # x = self.dropout(x)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

#-#-# You do NOT have to modify the code below this line. #-#-#

# instantiate the CNN
model_scratch = Net()

# move tensors to GPU if CUDA is available
if use_cuda:
    model_scratch.cuda()

print(model_scratch)
```

```
Net(
  (conv1): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1))
  (conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1))
  (conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1))
  (conv4): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1))
  (conv5): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=6400, out_features=500, bias=True)
  (fc2): Linear(in_features=500, out_features=133, bias=True)
)
```

Question 4: Outline the steps you took to get to your final CNN architecture and your reasoning at each step.

Answer:

- We can compute the spatial size of the output volume as a function of the input volume size (W), the kernel/filter size (F), the stride with which they are applied (S), and the amount of zero padding used (P) on the border. The correct formula for calculating how many neurons define the output_W is given by (WF+2P)/S+1 (from cifar10_cnn_exercise.jpynb). In the model, S (stride) = 1, F = 3, P = 0.

Therefore, 1. The conv1 lyr loads `224x224x3` tensors and convolves to `222x222x16` tensors. 2. The pooling lyr 1 applies MaxPool to convert `222x222x16` tensors to `111x111x16` tensors. 3. The conv2 lyr convolves `111x111x16` tensors to `109x109x32` tensors. 4. The pooling lyr 2 applies MaxPool to convert `109x109x32` tensors to `54x54x32` tensors. 5. The conv3 lyr convolves `54x54x32` tensors to `52x52x64` tensors. 6. The pooling lyr 3 applies MaxPool to convert 52x52x64 tensors to `26x26x64` tensors. 7. The conv4 lyr convolves `26x26x64` tensors to `24x24x128` tensors. 8. The pooling lyr 4 applies MaxPool to convert `24x24x128` tensors to `12x12x128` tensors. 9. The conv5 lyr convolves `12x12x128` tensors to `12x12x256` tensors (padding = 1 added to make the dimension even number). 10. The pooling lyr 5 applies MaxPool to convert `12x12x256` tensors to `5x5x256` tensors. 11. The dropout lyr was commented out after a few test run. The dropout discussion explains several situations that dropout may hurt performance. It looks that my networks model is relatively small (i.e. not very deep and not a large amount of hidden nodes) which indicate regularization is unnecessary in this case. In addition, training time is limited. 12. The linear transformation lyr 1 linearly transforms `5x5x256` tensors to 500 classes. (If padding = 1, the tensor 5x5x256 should be 6x6x256). 13. The linear transformation lyr 2 linearly transforms 500 classes to 133 output classes (133 breeds of dogs).

Note: For every convolutional lyr, I use relu activation function before max pooling. For the first linear transformation lyr, I use relu activation function but I didn't use it for the second linear transformation lyr.

1.1.9 (IMPLEMENTATION) Specify Loss Function and Optimizer

Use the next code cell to specify a loss function and optimizer. Save the chosen loss function as `criterion_scratch`, and the optimizer as `optimizer_scratch` below.

```
In [23]: import torch.optim as optim

         ### TODO: select loss function
         criterion_scratch = nn.CrossEntropyLoss()

         ### TODO: select optimizer
         optimizer_scratch = optim.Adam(model_scratch.parameters(), lr = 0.001)
```

1.1.10 (IMPLEMENTATION) Train and Validate the Model

Train and validate your model in the code cell below. Save the final model parameters at filepath `'model_scratch.pt'`.

```
In [24]: # the following import is required for training to be robust to truncated images
         from PIL import ImageFile
         ImageFile.LOAD_TRUNCATED_IMAGES = True
```

```python
def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
    """returns trained model"""
    # initialize tracker for minimum validation loss
    valid_loss_min = np.Inf

    for epoch in range(1, n_epochs+1):
        # initialize variables to monitor training and validation loss
        train_loss = 0.0
        valid_loss = 0.0

        ###################
        # train the model #
        ###################
        model.train()
        for batch_idx, (data, target) in enumerate(loaders['train']):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## find the loss and update the model parameters accordingly
            # clear the grad of all optimized variables
            optimizer.zero_grad()
            # Compute output
            output = model(data)
            ## record the average training loss, using something like
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()

            train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss)
            #train_loss = train_loss/len(train_loader.dataset)
        ######################
        # validate the model #
        ######################
        model.eval()
        for batch_idx, (data, target) in enumerate(loaders['valid']):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            output = model(data)
            loss = criterion(output, target)
            ## update the average validation loss
            valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (loss.data - valid_loss)
            #valid_loss = valid_loss/len(valid_loader.dataset)

        # print training/validation statistics
        print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
            epoch,
```

```python
                train_loss,
                valid_loss
                ))

            ## TODO: save the model if validation loss has decreased
            if valid_loss <= valid_loss_min:
                print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'.fo
                    valid_loss_min, valid_loss))
                torch.save(model.state_dict(), 'model_scratch.pt')
                valid_loss_min = valid_loss

        # return trained model
        return model


    # train the model
    model_scratch = train(40, loaders_scratch, model_scratch, optimizer_scratch,
                          criterion_scratch, use_cuda, 'model_scratch.pt')

    # load the model that got the best validation accuracy
    model_scratch.load_state_dict(torch.load('model_scratch.pt'))
```

```
Epoch: 1         Training Loss: 4.841729         Validation Loss: 4.708338
Validation loss decreased (inf --> 4.708338).  Saving model ...
Epoch: 2         Training Loss: 4.660092         Validation Loss: 4.461424
Validation loss decreased (4.708338 --> 4.461424).  Saving model ...
Epoch: 3         Training Loss: 4.501321         Validation Loss: 4.443649
Validation loss decreased (4.461424 --> 4.443649).  Saving model ...
Epoch: 4         Training Loss: 4.399081         Validation Loss: 4.221007
Validation loss decreased (4.443649 --> 4.221007).  Saving model ...
Epoch: 5         Training Loss: 4.283556         Validation Loss: 4.124555
Validation loss decreased (4.221007 --> 4.124555).  Saving model ...
Epoch: 6         Training Loss: 4.188977         Validation Loss: 3.960612
Validation loss decreased (4.124555 --> 3.960612).  Saving model ...
Epoch: 7         Training Loss: 4.109920         Validation Loss: 3.870793
Validation loss decreased (3.960612 --> 3.870793).  Saving model ...
Epoch: 8         Training Loss: 4.007260         Validation Loss: 3.731004
Validation loss decreased (3.870793 --> 3.731004).  Saving model ...
Epoch: 9         Training Loss: 3.922748         Validation Loss: 3.737801
Epoch: 10         Training Loss: 3.841840         Validation Loss: 3.542965
Validation loss decreased (3.731004 --> 3.542965).  Saving model ...
Epoch: 11         Training Loss: 3.756955         Validation Loss: 3.522970
Validation loss decreased (3.542965 --> 3.522970).  Saving model ...
Epoch: 12         Training Loss: 3.677365         Validation Loss: 3.508591
Validation loss decreased (3.522970 --> 3.508591).  Saving model ...
Epoch: 13         Training Loss: 3.601952         Validation Loss: 3.405294
Validation loss decreased (3.508591 --> 3.405294).  Saving model ...
Epoch: 14         Training Loss: 3.553341         Validation Loss: 3.355959
```

```
Validation loss decreased (3.405294 --> 3.355959).  Saving model ...
Epoch: 15         Training Loss: 3.491620        Validation Loss: 3.287579
Validation loss decreased (3.355959 --> 3.287579).  Saving model ...
Epoch: 16         Training Loss: 3.425995        Validation Loss: 3.249661
Validation loss decreased (3.287579 --> 3.249661).  Saving model ...
Epoch: 17         Training Loss: 3.391093        Validation Loss: 3.405172
Epoch: 18         Training Loss: 3.360672        Validation Loss: 3.436672
Epoch: 19         Training Loss: 3.304747        Validation Loss: 3.153569
Validation loss decreased (3.249661 --> 3.153569).  Saving model ...
Epoch: 20         Training Loss: 3.249341        Validation Loss: 3.207289
Epoch: 21         Training Loss: 3.206466        Validation Loss: 3.218075
Epoch: 22         Training Loss: 3.198368        Validation Loss: 3.028088
Validation loss decreased (3.153569 --> 3.028088).  Saving model ...
Epoch: 23         Training Loss: 3.111201        Validation Loss: 2.963718
Validation loss decreased (3.028088 --> 2.963718).  Saving model ...
Epoch: 24         Training Loss: 3.086368        Validation Loss: 3.034452
Epoch: 25         Training Loss: 3.029295        Validation Loss: 2.933927
Validation loss decreased (2.963718 --> 2.933927).  Saving model ...
Epoch: 26         Training Loss: 3.002706        Validation Loss: 2.888550
Validation loss decreased (2.933927 --> 2.888550).  Saving model ...
Epoch: 27         Training Loss: 3.028734        Validation Loss: 2.906989
Epoch: 28         Training Loss: 2.970123        Validation Loss: 2.830569
Validation loss decreased (2.888550 --> 2.830569).  Saving model ...
Epoch: 29         Training Loss: 2.918545        Validation Loss: 2.977314
Epoch: 30         Training Loss: 2.876193        Validation Loss: 2.868940
Epoch: 31         Training Loss: 2.883664        Validation Loss: 2.782770
Validation loss decreased (2.830569 --> 2.782770).  Saving model ...
Epoch: 32         Training Loss: 2.843059        Validation Loss: 2.851738
Epoch: 33         Training Loss: 2.849731        Validation Loss: 2.748884
Validation loss decreased (2.782770 --> 2.748884).  Saving model ...
Epoch: 34         Training Loss: 2.817803        Validation Loss: 2.787753
Epoch: 35         Training Loss: 2.810766        Validation Loss: 2.809745
Epoch: 36         Training Loss: 2.751747        Validation Loss: 2.695361
Validation loss decreased (2.748884 --> 2.695361).  Saving model ...
Epoch: 37         Training Loss: 2.705982        Validation Loss: 2.775699
Epoch: 38         Training Loss: 2.681160        Validation Loss: 2.768860
Epoch: 39         Training Loss: 2.708738        Validation Loss: 2.806176
Epoch: 40         Training Loss: 2.661114        Validation Loss: 2.736179
```

1.1.11 (IMPLEMENTATION) Test the Model

Try out your model on the test dataset of dog images. Use the code cell below to calculate and print the test loss and accuracy. Ensure that your test accuracy is greater than 10%.

```
In [25]: def test(loaders, model, criterion, use_cuda):

             # monitor test loss and accuracy
```

```
    test_loss = 0.
    correct = 0.
    total = 0.

    model.eval()
    for batch_idx, (data, target) in enumerate(loaders['test']):
        # move to GPU
        if use_cuda:
            data, target = data.cuda(), target.cuda()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the loss
        loss = criterion(output, target)
        # update average test loss
        test_loss = test_loss + ((1 / (batch_idx + 1)) * (loss.data - test_loss))
        # convert output probabilities to predicted class
        pred = output.data.max(1, keepdim=True)[1]
        # compare predictions to true label
        correct += np.sum(np.squeeze(pred.eq(target.data.view_as(pred))).cpu().numpy())
        total += data.size(0)

    print('Test Loss: {:.6f}\n'.format(test_loss))

    print('\nTest Accuracy: %2d%% (%2d/%2d)' % (
        100. * correct / total, correct, total))

# call test function
test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 2.676996


Test Accuracy: 34% (291/836)
```

Step 4: 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.

1.1.12 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

Use the code cell below to write three separate data loaders for the training, validation, and test datasets of dog images (located at `dogImages/train`, `dogImages/valid`, and `dogImages/test`, respectively).

If you like, **you are welcome to use the same data loaders from the previous step**, when you created a CNN from scratch.

```
In [26]: ## TODO: Specify data loaders

         train_data = datasets.ImageFolder(train_dir, transform = train_transform)
         valid_data = datasets.ImageFolder(valid_dir, transform = valid_test_transform)
         test_data = datasets.ImageFolder(test_dir, transform = valid_test_transform)

         train_loader = torch.utils.data.DataLoader(train_data, batch_size = batch_size, num_wor
         valid_loader = torch.utils.data.DataLoader(valid_data, batch_size = batch_size, num_wor
         test_loader = torch.utils.data.DataLoader(test_data, batch_size = batch_size, num_worke


         ###
         loaders_transfer = {'train': train_loader,
                             'valid': valid_loader,
                             'test': test_loader}
```

1.1.13 (IMPLEMENTATION) Model Architecture

Use transfer learning to create a CNN to classify dog breed. Use the code cell below, and save your initialized model as the variable `model_transfer`.

```
In [27]: import torchvision.models as models
         import torch.nn as nn

         ## TODO: Specify model architecture
         model_transfer = models.resnet50(pretrained = True)

         # Freeze training for all "features" layers
         for param in model_transfer.parameters():
             param.requires_grad = False

         ## new layers automatically have requires_grad = True
         n_inputs = model_transfer.fc.in_features

         last_layer = nn.Linear(n_inputs, len(test_data.classes))

         model_transfer.fc = last_layer

         print(model_transfer)
         print(model_transfer.fc.out_features)

         ## use cuda if available
         if use_cuda:
             model_transfer = model_transfer.cuda()
```

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
```

```
(relu): ReLU(inplace)
(maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
(layer1): Sequential(
  (0): Bottleneck(
    (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
    (downsample): Sequential(
      (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (1): Bottleneck(
    (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
  )
  (2): Bottleneck(
    (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
  )
)
(layer2): Sequential(
  (0): Bottleneck(
    (conv1): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
    (downsample): Sequential(
      (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
```

```
    )
    (1): Bottleneck(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
    )
    (2): Bottleneck(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
    )
    (3): Bottleneck(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
    )
  )
  (layer3): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (downsample): Sequential(
        (0): Conv2d(512, 1024, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
```

```
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
    )
    (2): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
    )
    (3): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
    )
    (4): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
    )
    (5): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
    )
  )
  (layer4): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
```

```
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (downsample): Sequential(
        (0): Conv2d(1024, 2048, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
    )
    (2): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
    )
  )
  (avgpool): AvgPool2d(kernel_size=7, stride=1, padding=0)
  (fc): Linear(in_features=2048, out_features=133, bias=True)
)
133
```

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:

- The pre-trained ResNet50 model architecture handles well with the dog breed classification project. So most of the pre-trained parameters are kept to save time and effort while still retain the accuracy score. I only change the last layer of ResNet50 which is a linear transformation layer. I change the output to the total dog breeds in the dog breed classification project (`classes = 133`).
- Pre-trained models save time and effort to build models with high test accuracy. Discussions of VGG16, VGG19, InceptionV3, ResNet18, ResNet50 indicate that VGG19 and ResNet50 are the best given the limited computing power and training time I have.

1.1.14 (IMPLEMENTATION) Specify Loss Function and Optimizer

Use the next code cell to specify a loss function and optimizer. Save the chosen loss function as `criterion_transfer`, and the optimizer as `optimizer_transfer` below.

```
In [28]: criterion_transfer = nn.CrossEntropyLoss()

         # only optimize the last layer
         optimizer_transfer = optim.Adam(model_transfer.fc.parameters(), lr = 0.001)
```

1.1.15 (IMPLEMENTATION) Train and Validate the Model

Train and validate your model in the code cell below. Save the final model parameters at filepath `'model_transfer.pt'`.

```
In [29]: # train the model
         model_transfer = train(20, loaders_transfer, model_transfer, optimizer_transfer, criter

         # load the model that got the best validation accuracy (uncomment the line below)
         model_transfer.load_state_dict(torch.load('model_transfer.pt'))
```

```
Epoch: 1         Training Loss: 2.707422         Validation Loss: 0.915369
Validation loss decreased (inf --> 0.915369).  Saving model ...
Epoch: 2         Training Loss: 1.386820         Validation Loss: 0.649442
Validation loss decreased (0.915369 --> 0.649442).  Saving model ...
Epoch: 3         Training Loss: 1.242088         Validation Loss: 0.617885
Validation loss decreased (0.649442 --> 0.617885).  Saving model ...
Epoch: 4         Training Loss: 1.178380         Validation Loss: 0.636371
Epoch: 5         Training Loss: 1.110204         Validation Loss: 0.593556
Validation loss decreased (0.617885 --> 0.593556).  Saving model ...
Epoch: 6         Training Loss: 1.106920         Validation Loss: 0.553279
Validation loss decreased (0.593556 --> 0.553279).  Saving model ...
Epoch: 7         Training Loss: 1.107267         Validation Loss: 0.554281
Epoch: 8         Training Loss: 1.075400         Validation Loss: 0.577247
Epoch: 9         Training Loss: 1.064740         Validation Loss: 0.509919
Validation loss decreased (0.553279 --> 0.509919).  Saving model ...
Epoch: 10         Training Loss: 1.019020         Validation Loss: 0.543818
Epoch: 11         Training Loss: 1.037879         Validation Loss: 0.539536
Epoch: 12         Training Loss: 0.983527         Validation Loss: 0.580987
Epoch: 13         Training Loss: 1.027580         Validation Loss: 0.583776
Epoch: 14         Training Loss: 0.999924         Validation Loss: 0.559872
Epoch: 15         Training Loss: 1.001945         Validation Loss: 0.595389
Epoch: 16         Training Loss: 0.966073         Validation Loss: 0.544141
Epoch: 17         Training Loss: 0.982605         Validation Loss: 0.537484
Epoch: 18         Training Loss: 0.973335         Validation Loss: 0.512887
Epoch: 19         Training Loss: 0.974847         Validation Loss: 0.530348
Epoch: 20         Training Loss: 0.998169         Validation Loss: 0.503291
Validation loss decreased (0.509919 --> 0.503291).  Saving model ...
```

```
---------------------------------------------------------------------------

FileNotFoundError                         Traceback (most recent call last)

<ipython-input-29-48b7909b384a> in <module>()
  3
  4 # load the model that got the best validation accuracy (uncomment the line below)
----> 5 model_transfer.load_state_dict(torch.load('model_transfer.pt'))


/opt/conda/lib/python3.6/site-packages/torch/serialization.py in load(f, map_location, p
299             (sys.version_info[0] == 3 and isinstance(f, pathlib.Path)):
300         new_fd = True
--> 301         f = open(f, 'rb')
302     try:
303         return _load(f, map_location, pickle_module)


FileNotFoundError: [Errno 2] No such file or directory: 'model_transfer.pt'
```

1.1.16 (IMPLEMENTATION) Test the Model

Try out your model on the test dataset of dog images. Use the code cell below to calculate and print the test loss and accuracy. Ensure that your test accuracy is greater than 60%.

```
In [30]: test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.492309


Test Accuracy: 87% (728/836)
```

1.1.17 (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.

```
In [31]: ### TODO: Write a function that takes a path to an image as input
         ### and returns the dog breed that is predicted by the model.
         data_transfer ={'train': train_data,
                        'valid': valid_data,
                        'test': test_data}
         # list of class names by index, i.e. a name can be accessed like class_names[0]
         class_names = [item[4:].replace("_", " ") for item in data_transfer['train'].classes]

         def predict_breed_transfer(img_path):
```

```
    # load the image and return the predicted breed
    image = Image.open(img_path)
    predict_transform = transforms.Compose([transforms.Resize(256),
                                             transforms.CenterCrop(224),
                                            transforms.ToTensor(),
                                            transforms.Normalize(mean=[0.485, 0.456, 0.4
    image = predict_transform(image)
    image = image.unsqueeze(0)

    if use_cuda:
        image = image.cuda()

    model_transfer.eval()

    output = model_transfer(image)
    _, preds_tensor = torch.max(output, 1)
    # convert output probabilities to predicted class
    index = output.data.argmax(dim=1)
    dog_breed = class_names[index]
    # normalize the probability to 0-1 and sum to 1
    probability = F.softmax(output)
    breed_prob = probability.data.max(dim=1)[0]
    return dog_breed, breed_prob.item()
```

Step 5: Write your Algorithm

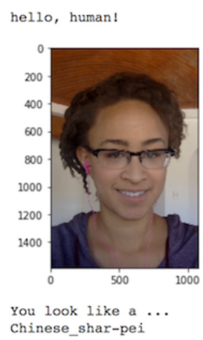
Write an algorithm that accepts a file path to an image and first determines whether the image contains a human, dog, or neither. Then, - if a **dog** is detected in the image, return the predicted breed. - if a **human** is detected in the image, return the resembling dog breed. - if **neither** is 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 4 to predict dog breed.

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

1.1.18 (IMPLEMENTATION) Write your Algorithm

In [32]:
```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.

def run_app(img_path):
    ## handle cases for a human face, dog, and neither
    img = Image.open(img_path)
    if dog_detector(img_path):
        breed, breed_prob = predict_breed_transfer(img_path)
        print('A dog is detected in this image. The breed is predicted as {} with a pro
```



Sample Human Output

```
            .format(breed, breed_prob*100))
        plt.imshow(img)
        plt.title('The predicted breed is {}.'.format(breed))
        plt.show()
    elif face_detector(img_path):
        breed, breed_prob = predict_breed_transfer(img_path)
        print('A human is detected in this image. The resembling dog breed is predicted
            .format(breed, breed_prob*100))
        plt.imshow(img)
        plt.title('Human! But resembling breed is {}.'.format(breed))
        #print('An example of resembling dog breed image is:')
        plt.show()
    else:

        plt.imshow(img)
        plt.show()
        return ValueError('Neither human nor dog is detected in this image.')
```

Step 6: 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?

1.1.19 (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: (Three possible points for improvement)

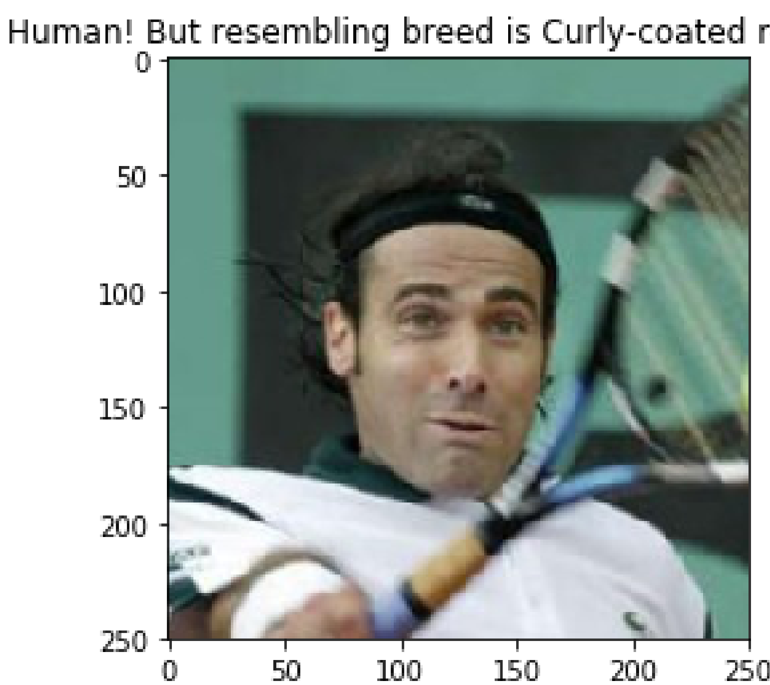
Yes. In general, the model performs well. However, for some similiar looking dog images but are of different dog breeds, the model doesn't have high probablity to confirm it is the right dog breed. Sometimes, the dog images may contain a few different dog breeds which makes the model classify incorrectly.

- Data augmentation: I can try different augmentation techniques (e.g. random cropping, translations, color scale shitts, etc.) Possibly it can help avoid overfitting and may increase test accuracy. People tried GANs which might be a good choice as well.
- The last layer of ResNet50: I can replace the last layer with a few more hidden layers and one output layer (e.g. 2048 -> 1024, 1024 -> 512, 512 -> 133). Possibly it can improve the performance of the model.
- Different model architecture or modify the model architecture of ResNet50: VGG19 model architecture may play better if tested. Removing/adding conv layers, change to different filters may help improve the performance of the model.
- There are some noises in the background of the dog images. Maybe a pre-processing technique should be applied to reduce the background noises before training the model would do better.

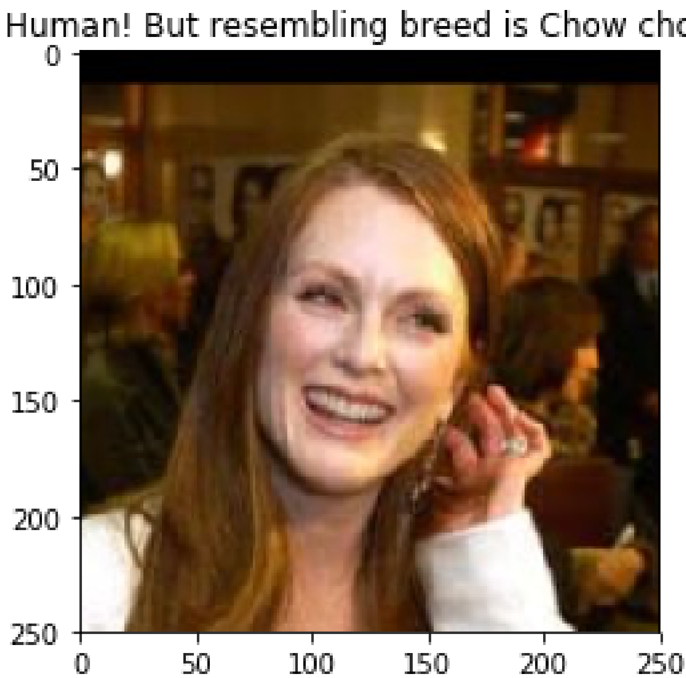
In [33]:
```
## TODO: Execute your algorithm from Step 6 on
## at least 6 images on your computer.
## Feel free to use as many code cells as needed.

## suggested code, below
for file in np.hstack((human_files[1:201:40], dog_files[1:201:40])):
    run_app(file)
```

```
/opt/conda/lib/python3.6/site-packages/ipykernel_launcher.py:30: UserWarning: Implicit dimension
```

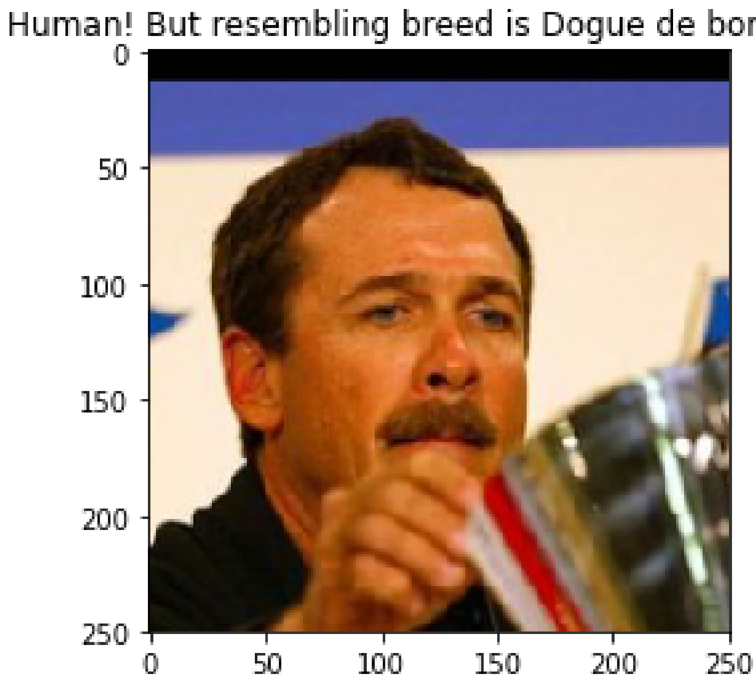
```
A human is detected in this image. The resembling dog breed is predicted as Curly-coated retriev
```



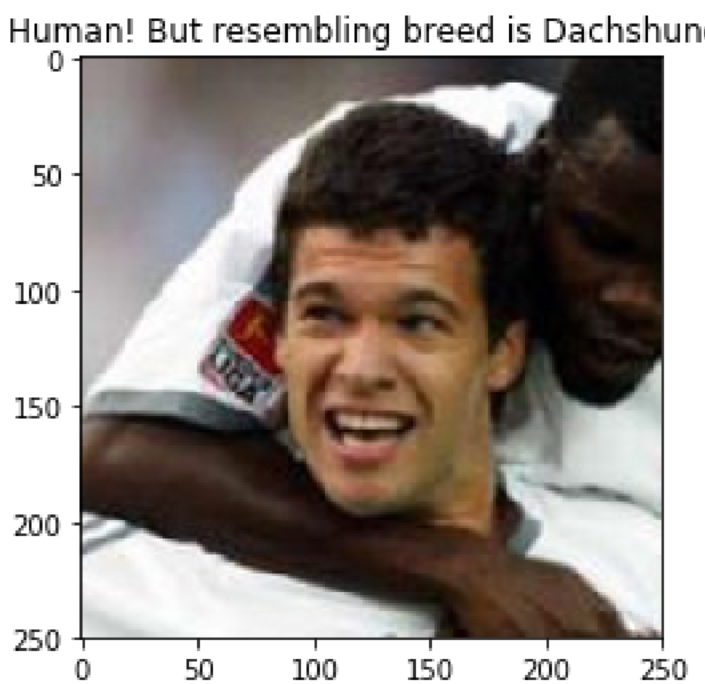
```
A human is detected in this image. The resembling dog breed is predicted as Chow chow with a pro
```



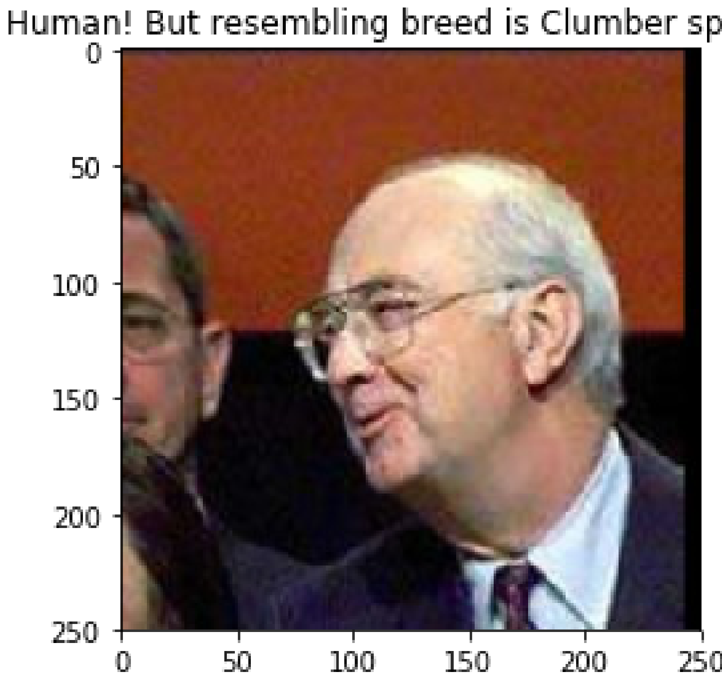

```
A human is detected in this image. The resembling dog breed is predicted as Dogue de bordeaux wi
```



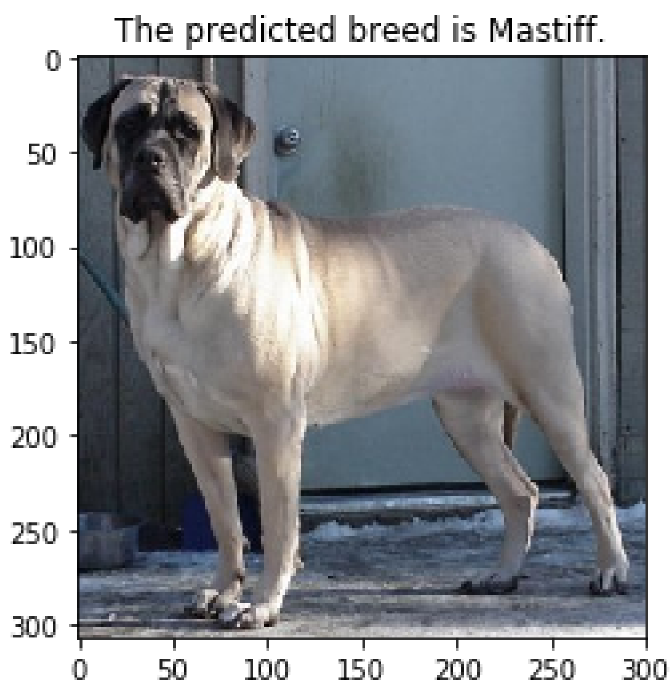

```
A human is detected in this image. The resembling dog breed is predicted as Dachshund with a pro
```



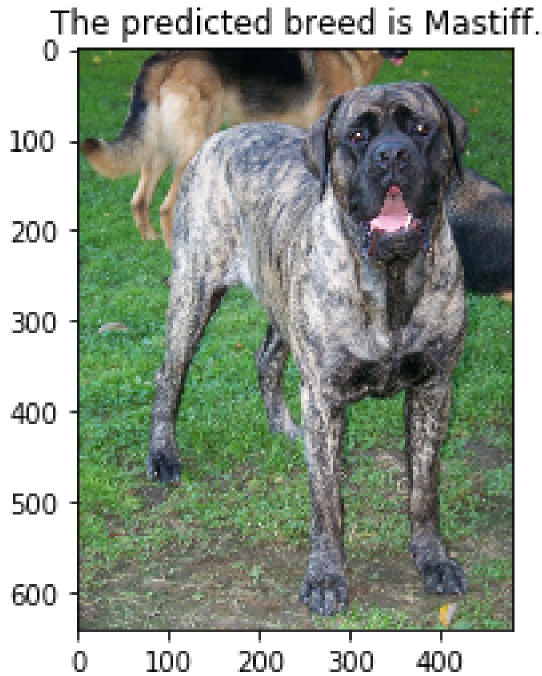
```
A human is detected in this image. The resembling dog breed is predicted as Clumber spaniel with
```



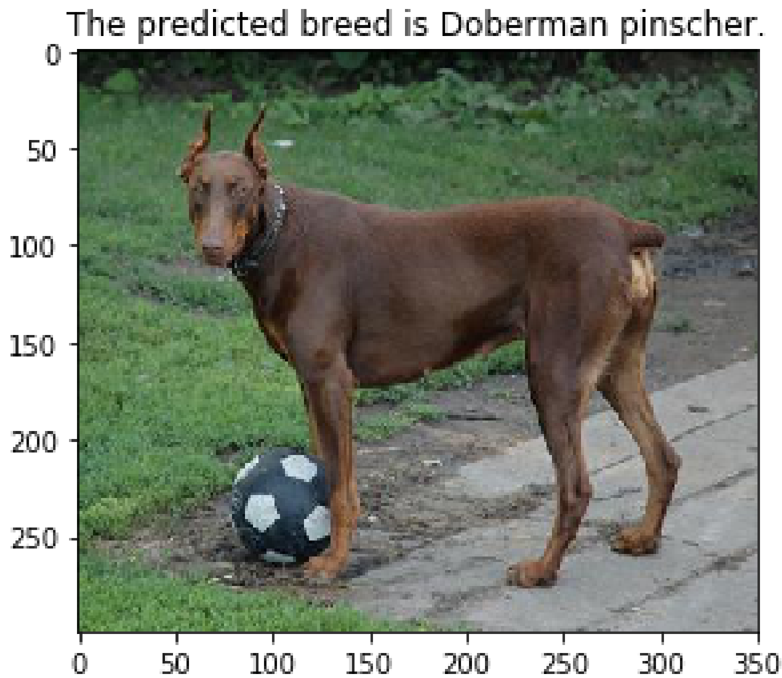

```
A dog is detected in this image. The breed is predicted as Mastiff with a probablity of 98.10 %.
```



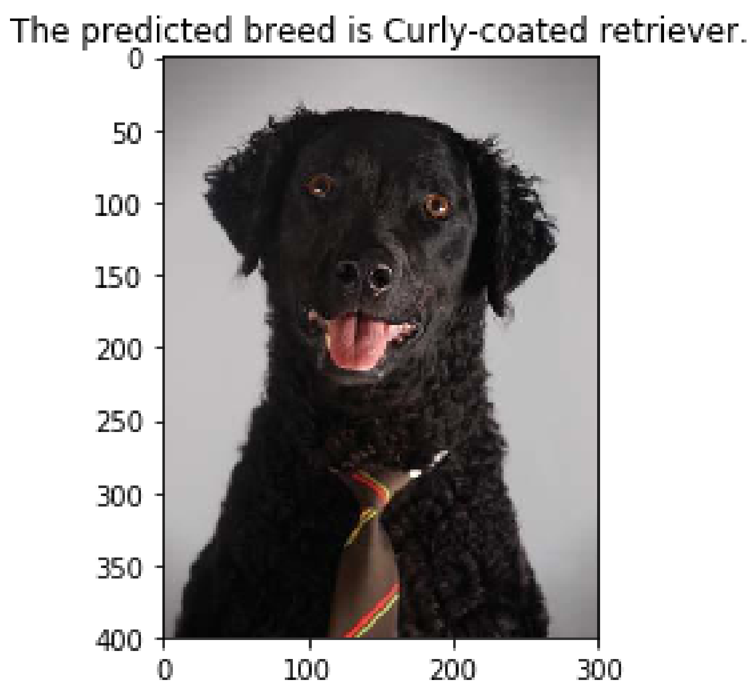
A dog is detected in this image. The breed is predicted as Mastiff with a probablity of 93.59 %.



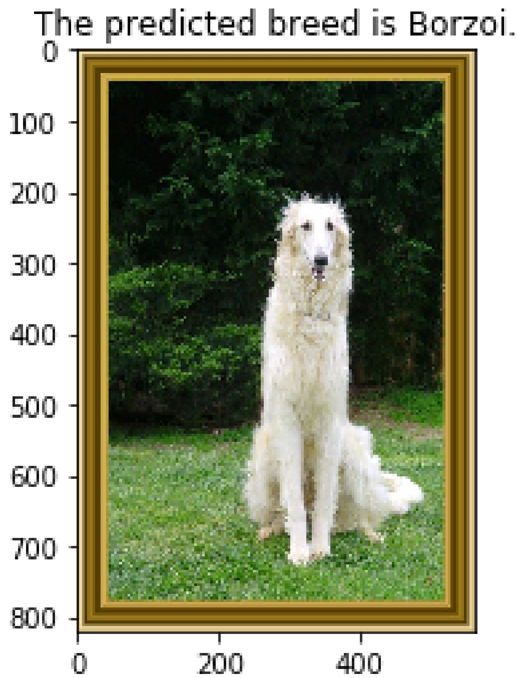
A dog is detected in this image. The breed is predicted as Doberman pinscher with a probablity o




```
A dog is detected in this image. The breed is predicted as Curly-coated retriever with a probabl
```



```
A dog is detected in this image. The breed is predicted as Borzoi with a probablity of 100.00 %.
```




```
In [ ]:
```