Dog Breed Classifier

REVIEW HISTORY

Meets Specifications

Great job with the assignment.
Do look at how to deploy these models on the edge like a browser (using tj.js) or on iot/mobile devices using tflite.
Also, visualize the features learnt by CNN layers for complex architectures to understand how layers learn and based on that experience you can always decide on number of layers for a complex/simple image classification task.
Happy Learning :)

Files Submitted

The submission includes all required, complete notebook files.

All files present.

Step 1: Detect Humans

The submission returns the percentage of the first 100 images in the dog and human face datasets that include a detected, human face.

Good use of only using counter variable in list comprehension as we are testing on 100 images so total detection count would give us percentage.
Also you can look at using MTCNN instead of haarcascades.
This blog is has very good insights at implementing face detection with multiple algorithms ->
https://www.pyimagesearch.com/2018/09/24/opencv-face-recognition/

Step 2: Detect Dogs

Use a pre-trained VGG16 Net to find the predicted class for a given image. Use this to complete a `dog_detector` function below that returns True if a dog is detected in an image (and False if not).

Tip : You can use torch.no_grad() after using model.eval() here to avoid backpropogation as we are only using it for inference and forward prop. The preprocessing is done well.

The submission returns the percentage of the first 100 images in the dog and human face datasets that include a detected dog.

```
What percentage of the images in human_files_short have a detected dog? 2%
What percentage of the images in dog_files_short have a detected dog? 100%
```

Near perfect accuracy! Done well!

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

Write three separate data loaders for the training, validation, and test datasets of dog images. These images should be pre-processed to be of the correct size.

```
train_loader = torch.utils.data.DataLoader(dataset=train_data,batch_size=batch_size,shuffle
=True)

valid_loader = torch.utils.data.DataLoader(dataset=valid_data,batch_size=batch_size,shuffle
=False)
test_loader = torch.utils.data.DataLoader(dataset=test_data,batch_size=batch_size,shuffle=F
alse)
```

Good job avoiding shuffling and augmenting validation and test set as everytime we would train again, the validation and test accuracies would differ as the batches would differ alongwith randomly new generated images so to benchmark things it is preferable to not shuffle and augment them.
Also pytorch's transform resize behaves differently when you pass only integer so keep that in mind
https://pytorch.org/docs/stable/torchvision/transforms.html#torchvision.transforms.Resize

Answer describes how the images were pre-processed and/or augmented.

A tip while augmenting data - You should also look at the output generated images to see that the augmentation is not making the image, not even human classification worthy because that would be bad for the network to learn.

The submission specifies a CNN architecture.

```
Net(
  (conv1): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=50176, out_features=500, bias=True)
  (fc2): Linear(in_features=500, out_features=133, bias=True)
  (dropout): Dropout(p=0.2)
)
```

3conv 2fc is s decent architecture to play around.

Answer describes the reasoning behind the selection of layer types.

Tips:

- You can also try and imitate cifar_10/Lenet like structure for training scratch models as well. Good job on adding dropout to avoid overfitting.
- You can visualize each layer and filters with what shape they are learning at each layer to make it more intuitive to builds CNNs from scratch.
  https://cs231n.github.io/understanding-cnn/

Choose appropriate loss and optimization functions for this classification task. Train the model for a number of epochs and save the "best" result.

Tip:Plotting the losses against epochs is a good way to see if the training set goes into overfitting/underfitting mode or not.
Also, we can implement early stopping as well if you feel some epsilon delta (1e^-04) change is not happening consistently for n epochs in validation loss.

The trained model attains at least 10% accuracy on the test set.

```
Test Loss: 3.800414


Test Accuracy: 13% (111/836)
```

Try to introduce batchnorm with more layers here as there is class imbalance in the dataset and definitely you can train for more epochs to push accuracy > 20%

Step 4: Create a CNN Using Transfer Learning

The submission specifies a model architecture that uses part of a pre-trained model.

```
model_transfer = models.vgg16(pretrained=True)
```

Good choice of sticking with VGG16

The submission details why the chosen architecture is suitable for this classification task.

You can also look at Resnet, inception and Xception networks which work well on imagenet in general as imagenet also has some dog breeds.
Also, everytime you try transfer learning the first dirty implementation should just be cutting off the last output layer and forward training it for 5-10 epochs to see if this model is making sense for our dataset.

Train your model for a number of epochs and save the result wth the lowest validation loss.

10 epochs is okay.
Ideally, you should train for 2-10 epochs max in a pretrained model because the possibility of overfitting is more. So you can use high learning rate and close it early.

Accuracy on the test set is 60% or greater.

```
Test Loss: 0.445822


Test Accuracy: 86% (726/836)
```

This is good accuracy! With more denser models like resnet151 you can push this to 89-90 as well.

The submission includes a function that takes a file path to an image as input and returns the dog breed that is predicted by the CNN.

```
    image_transofrmer = transforms.Compose([transforms.Resize((224,224)),
                                            transforms.ToTensor(),
                                            transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                                 std=[0.229, 0.224, 0.225])])
```

Good job implementing the preprocessing steps and using model.eval(). Also you can add torch.no_grad() to avoid backpropogation

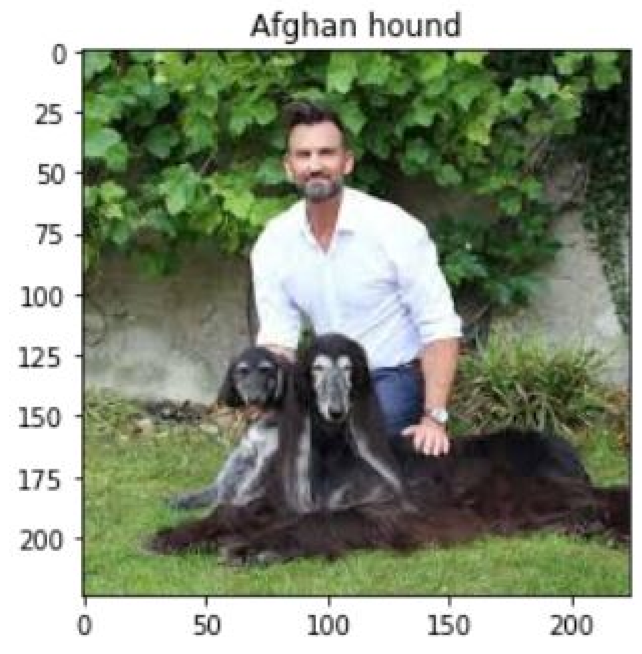
Step 5: Write Your Algorithm

The submission uses the CNN from the previous step to detect dog breed. The submission has different output for each detected image type (dog, human, other) and provides either predicted actual (or resembling) dog breed.

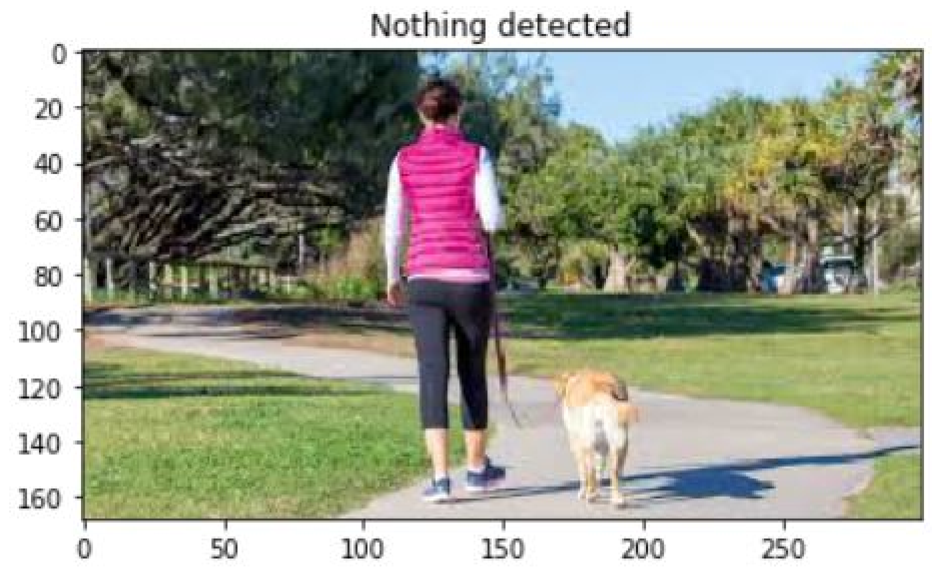
Done well. Glad that you added image plotting which comes handy to see the input image.

Step 6: Test Your Algorithm

The submission tests at least 6 images, including at least two human and two dog images.



```
--------------------------
Sorry :( we can't detect dogs or human
```



Good testing of images outside the dataset and some interesting cases like dogs and human faces not visible and multiple dogs in an image

Submission provides at least three possible points of improvement for the classification algorithm.

Some more points to think about

- You can also think in terms of class imbalance and lack of data for some classes causing the model to be biased to some breeds.
- You can also discuss the possibility of detecting multiple faces and accuracy of face detector as we have cnn based face detectors performing well now.
- For CNN architecture, you can discuss how you can improve your accuracy by introducing weights initialization and more sophisticated network in your scratch model to improve its accuracy.

DOWNLOAD PROJECT

RETURN TO PATH