UDACITY

Return to Classroom

DISCUSS ON STUDENT HUB

Dog-Breed Classifier

REVIEW

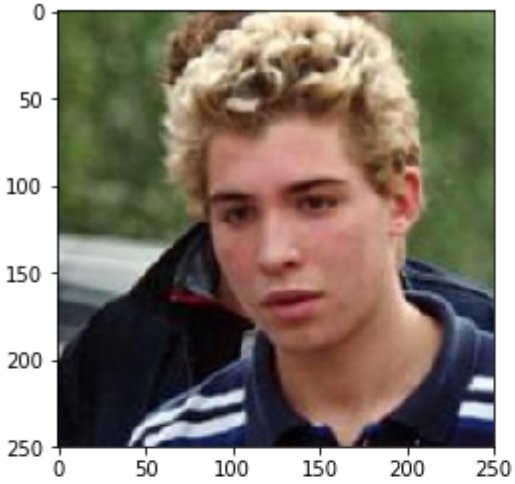
HISTORY

Meets Specifications

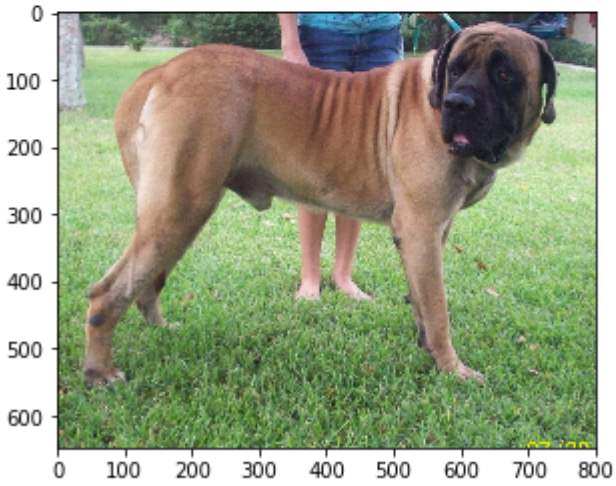
Congratulations

Great submission!

All functions were implemented correctly, the detectors easily meet the required accuracies and the final algorithm seems to work quite well.



```
its human You seems like a American foxhound
```



```
its Dog You seems like a Mastiff
```

Suggesting some further readings:

- Using Convolutional Neural Networks to Classify Dog Breeds
- How To Improve Deep Learning Performance.

Suggesting some further actions to make your project even better!

1. **Augment the Training Data**: Augmenting the training and/or validation set might help improve model performance.
2. **Turn your Algorithm into a Web App**: Turn your code into a web app using Flask or web.py!
3. **Overlay Dog Ears on Detected Human Heads**: Overlay a Snapchat-like filter with dog ears on detected human heads. You can determine where to place the ears through the use of the OpenCV face detector, which returns a bounding box for the face. If you would also like to overlay a dog nose filter, here some nice

tutorials for facial keypoints detection.

4. **Add Functionality for Dog Mutts**: Currently, if a dog appears 51% German Shephard and 49% poodle, only the German Shephard breed is returned. The algorithm is currently guaranteed to fail for every mixed breed dog. Of course, if a dog is predicted as 99.5% Labrador, it is still worthwhile to round this to 100% and return a single breed; so, you will have to find a nice balance.

5. **Experiment with Multiple Dog/Human Detectors**: Perform a systematic evaluation of various methods for detecting humans and dogs in images. Provide improved methodology for the face_detector and dog_detector functions.

About your comment:

OPTIONAL: Question for the reviewer

If you have any question about the starter code or your own implementation, please add it in the cell below.

For example, if you want to know why a piece of code is written the way it is, or its function, or alternative ways of implementing the same functionality, or if you want to get feedback on a specific part of your code or get feedback on things you tried but did not work.

Please keep your questions succinct and clear to help the reviewer answer them satisfactorily.

I just want to wish you a nice day

Thank you very much

Files Submitted

The submission includes all required, complete notebook files.

All required files were included

Your code was functional, well-documented, and organized. Well done!

Suggested reading about how to organize your code:

- Google Python Style Guide
- Python Best Practices
- Clean Code Summary

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.

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:

- After executing my code, the following is output:
- 1 - Percentage of human faces in the file: 98.0%.
- 2- The percentage of dog's faces in the file was 17.0%.

```
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.

HumanFaces = 0
DogsFaces = 0

for i in human_files_short:
    if face_detector(i):
        HumanFaces +=1
for i in dog_files_short:
    if face_detector(i):
        DogsFaces +=1

HumanPercentage = HumanFaces/len(human_files_short)*100
DogPercentage = DogsFaces/len(dog_files_short)*100

print('The Percentage of Human Faces in the File is : '+ str(HumanPerce
ntage))
print('The Percentage of Dog Faces in the File is : ' + str(DogPercenta
ge))
```

```
The Percentage of Human Faces in the File is : 98.0
The Percentage of Dog Faces in the File is : 17.0
```

Well done!

- Percentage of human images with detected human faces: **98%**
- Percentage of dog images with detected human faces: **17%**

Perfect!

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).

Well done!

You implemented the code to load the image, normalized it, resized it, cropped it and extracted the result from VGG16.

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

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 Dog's Faces Found in Human Files using Dog Detector Function is : 0.0 % .
- The Percentage of Dog's Faces Found in the Dog File using Dog Detector Function is : 100.0 % .

```
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.
HumanFaces = 0
DogsFaces = 0

for i in human_files_short:
    if dog_detector(i):
        HumanFaces += 1
for i in dog_files_short:
    if dog_detector(i):
        DogsFaces +=1

HumanPercentage = HumanFaces/len(human_files_short)*100
DogPercentage = DogsFaces/len(dog_files_short)*100

print('The Percentage of Dog Faces Found in Human Files using Dog Detec
tor Function is : '+ str(HumanPercentage) +" %")
print('The Percentage of Dog Faces Found in the Dog File using Dog Dete
ctor Function is : ' + str(DogPercentage)+ " %")
```

```
The Percentage of Dog Faces Found in Human Files using Dog Detector Fun
ction is : 2.0 %
The Percentage of Dog Faces Found in the Dog File using Dog Detector Fu
nction is : 100.0 %
```

Great!

- Percentage of first 100 images in human images with detected dog: **2.0%**
- Percentage of first 100 images in dog images with detected dog: **100%**

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

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.

Good job creating the train_loader, valid_loader and test_loader data loaders and also resizing and cropping the images to a square size of 224.

It is suggested to use PyTorch's transforms and DataLoader class

Suggested reading about the discussion of what machine learning framework should be used.

- TensorFlow vs. Pytorch.
- Tensorflow or PyTorch : The force is strong with which one?
- What is the best programming language for Machine Learning?

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

Nice work describing your preprocessing steps in question 3.

Good job with data augmentation!

The submission specifies a CNN architecture.

The submission specified a CNN architecture.

```
Net(
  (conv1): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (batchnorm1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (batchnorm2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (batchnorm3): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv4): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (batchnorm4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv5): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
```

```
  (batchnorm5): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (dropout): Dropout(p=0.25)
  (fc1): Linear(in_features=12544, out_features=1024, bias=True)
  (batchnorm_fc1): BatchNorm1d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (fc2): Linear(in_features=1024, out_features=133, bias=True)
)
```

Good job in implementing your CNN architecture!

Adding some dropout layers to reduce the risk of overfitting was a very sensible decision.

Answer describes the reasoning behind the selection of layer types.

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

Answer:

- My frist try was with 3 convolutional layers and 2 fully connected layers. My kernel size is 3x3 and padding is 1, but my accuracy was under 10% after 60 epochs so i added more Layers Becase The more convolutional layers there are, the more complex patterns in color and shape a model can detect so my architecture has 5 convolutional layers and two fully connected layer.
- after searching i found usefull information and tried it the information is that adding batch normalization after each layer is helped me to make the traning more fast and get good accurcy with less epochs. batch normalization is same MaxPool2d but as i said it'e give me better performance

Nice work describing the reasoning behind the selection of layer types in question 4.

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

```
test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 3.669544


Test Accuracy: 16% (135/836)
```

This is pretty high for a from scratch model with such limited data, well done!

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

Good job using the CrossEntropyLoss() loss function as it is suitable for classification tasks such as this.

Great choice of the SGD optimizer as well.

Step 4: Create a CNN Using Transfer Learning

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

densenet161 was a good choice.

You could try some of the other pre-trained models provided by torchvision.models to see which one can give you the highest accuracy.

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

Nice work describing the reasons architecture succeeded in question 5.

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:

- i choosed DenseNet 161 becase it is better than VGG 16 on ImageNet this arch gives us a

- I choosed DenseNet-161 because it is better than VGG-16 on ImageNet.this arch gives us a good architecture and taking all preceding feature-maps as input.and that gives me higher scores and faster performace so my accuracy 88 % after 10 epoches

Good use of transfer learning. Well done!

Transfer learning is a machine learning method where a model developed for a task is reused as the starting point for a model on a second task.

Here are some good documents you can check out for more insight on transfer learning:

- CS231n Convolutional Neural Networks for Visual Recognition
- Transfer Learning Introduction

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

Well done training the model with Adagrad optimizer and CrossEntropyLoss() function.

This is a nice article that would help enrich your knowledge about them: Loss Functions and Optimization Algorithms. Demystified. may help you understand the uses of these functions and algorithms.

Accuracy on the test set is 60% or greater.

```
test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.377528


Test Accuracy: 88% (742/836)
```

The test accuracy exceeds the required 60%, well done!

Compared to training from scratch (step 3), transfer learning results in a pretty impressive accuracy.

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.

A function predict_breed_transfer() was correctly defined to take a file path as input and return the breed predicted by the CNN.

```
def predict_breed_transfer(img_path):

    # load the image and return the predicted breed
    model_transfer.eval()
    transform = transforms.Compose([transforms.Resize(224),
                                    transforms.CenterCrop(224),
                                    transforms.ToTensor()])
    img = Image.open(img_path)
    img = transform(img).float()
    img = img.unsqueeze(0)

    if use_cuda:
        img = img.cuda()

    predictions = model_transfer(img)_, index = predictions.max(1)
    return class_names[index.item()]
```

Great job!

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.

The implemented algorithm in run_app() uses the CNN (predict_breed_transfer() function) to predict dog breeds.

```
def run_app(img_path):
    img = cv2.imread(img_path)
    cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    plt.imshow(cv_rgb)
    plt.show()
    if dog_detector(img_path):
        print(' its Dog You seems like a ' + predict_breed_transfer(img_path))
    elif face_detector(img_path):
        print('its human You seems like a ' + predict_breed_transfer(img_path))
```

```
        print('its human You seems like a ' + predict_breed_transfer(img_path))
    else:
        print('Can not Detect anything')
```

Your algorithm gives the correct output!

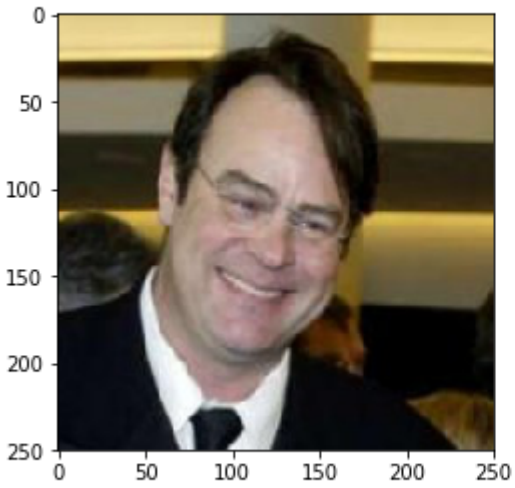
To give some extra information to the user you could think about adding the predicted probability of a dog breed and showing an (example) image of the predicted dog breed.

Step 6: Test Your Algorithm

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

Nice job testing the algorithm on several human and dog images.

```
for file in np.hstack((human_files[:3], dog_files[:3])):
    run_app(file)
```




```
its human You seems like a Kerry blue terrier
```

The predicted dog breeds for the images are excellent. Impressive results!

The algorithm was sufficiently tested

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

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)

- display images where the model fails to understand why it fails
- Some codes (e.g transforms.Normalize) are repeated, may be we can create a function or something and make it more usable .
- Doing more data augmentations techniques
- Providing more images to train
- Maybe Trying other transfer learning models
- Increaseing the number of epochs as validation loss is still decreasing

Nice work describing possible points of improvement for the classification algorithm in question 6.

Extra Feedback from Reviewer

UNGRADED (Always pass). No requirements for learners. Reviewers will use this rubric item to provide additional feedback.

Overall you did a fine job implementing your 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
        self.conv1 = nn.Conv2d(3, 16, 3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.batchnorm1 = nn.BatchNorm2d(16)

        self.conv2 = nn.Conv2d(16, 32, 3, padding=1)
        self.batchnorm2 = nn.BatchNorm2d(32)

        self.conv3 = nn.Conv2d(32, 64, 3, padding=1)
        self.batchnorm3 = nn.BatchNorm2d(64)

        self.conv4 = nn.Conv2d(64, 128, 3, padding=1)
        self.batchnorm4 = nn.BatchNorm2d(128)

        self.conv5 = nn.Conv2d(128, 256, 3, padding=1)
        self.batchnorm5 = nn.BatchNorm2d(256)

        self.dropout = nn.Dropout(0.25)

        self.fc1 = nn.Linear(256*7*7, 1024)
        self.batchnorm_fc1 = nn.BatchNorm1d(1024)

        self.fc2 = nn.Linear(1024, DogBreedClasess)

    def forward(self, x):
        ## Define forward behavior
        x = self.pool(F.relu(self.conv1(x)))
        x = self.batchnorm1(x)

        x = self.pool(F.relu(self.conv2(x)))
        x = self.batchnorm2(x)

        x = self.pool(F.relu(self.conv3(x)))
        x = self.batchnorm3(x)

        x = self.pool(F.relu(self.conv4(x)))
        x = self.batchnorm4(x)

        x = self.pool(F.relu(self.conv5(x)))
        x = self.batchnorm5(x)

        #Flatten Data
        x = self.dropout(x.view(-1, 256*7*7))

        x = self.dropout(F.relu(self.fc1(x)))
        x = self.batchnorm_fc1(x)
```

```
        x = self.fc2(x)
        return x
```

Comments about you code:

- You chose to use maxpooling layers to decrease the spatial size.
- Adding some dropout layers to reduce the risk of overfitting was a very sensible decision.
- You used ReLU activations between convlayers to introduce non-linearity and to allow gradients to flow backwards through the layer unimpeded.
- You have used Batch normalization to transform the input to zero mean/unit variance distributions.

Batch normalization is becoming very popular to further improve the performance of the model.

Batch normalization layers avoid covariate shift and accelerate the training process

Look at these suggested readings:

- Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift
- Understanding the backward pass through Batch Normalization Layer

When you have any time look at this discussion on how do you decide the parameters of a Convolutional Neural Network for image classification and another one, about how can you decide the kernel size, output maps and layers of CNN.

UNGRADED (Always pass). No requirements for learners. Reviewers will use this rubric item to provide additional feedback.

So, computer vision is sometimes seen as a part of the artificial intelligence field or the computer science field in general.

Considering deep learning methods

- They can achieve state-of-the-art results on challenging computer vision problems such as image classification, object detection, and face recognition.
- And one of the major advantages of neural nets is their ability to generalize.

This means that a trained net could classify data from the same class as the

learning data that it has never seen before.

The training set is used to train a neural net and the error of this dataset is minimized during training.

UNGRADED (Always pass). If you have a question for the reviewer specifically related to your implementation, please add the question in the cell provided at the bottom of the notebook. This rubric item will then be used by the reviewer to answer your question.

As you did not ask any specific questions I will comment on the training

Due to the size of the training set, this training usually lasts from an hour and a half to two hours and this variation depends on the complexity of the neural network.

There are factors actually increases the training time:

1. The large input size.
2. The data structure (shifted mean, and unnormalized).
3. The large network (depth and/or width).
4. Low, high, symmetric weights initialization strategy.
5. Slow weights update rule.
6. Very small or high learning rate.
7. Large number of epochs.
8. The batch size not compatible with physical available memory.
9. No data normalization within network.
10. No or very high regularization factor.

The training is dependent on the error, accuracy evaluation, and precision goal factors.

As soon as we get to the lowest error, highest accuracy and precision, we can take down the training session.

The above factors can lead to slow convergence to the target goals factor, and thus increases the overall time.

Convolutional Neural Networks generally can take a long time to train, even

performing adequate transfer learning on a pre-trained model such as VGG16 or ResNet.

At first simple checks and optimisations can lead to huge reductions in training times for convolutional neural networks.

Such as:

- Confirming correct GPU driver installation,
- Using a fast SSD to store training data, and
- Optimising your model to take advantage of multicore CPUs

For speedier convergence, take a look at these factors:

1. Reduce the input size to the appropriate dimensions.
2. Always preprocess the input to make it zero mean, and normalized it.
3. Keep the network depth and width that is not to high or low.
4. Use gradient decent weight update like Adam.
5. Learning rate should be determine by trying multiple, and using that which gives the best reduction in error.
6. If you are not able to make any further improvements, there is no need to take more epochs.
7. The batch size is based on the available memory, and number of CPUs/GPUs.
8. Use batch normalization.
9. Use regularization if need to reduce the risk of overfitting. But high use of regularization . leads to slow convergence.

DOWNLOAD PROJECT