Dog breed classifier

Project Overview

In this project, I built a deep learning image classification with open source deep learning framework Pytorch in order to distinguish different breeds of dogs. The model I have used in the project is Convolutional Neural Network (CNN), which is the state-of-the-art model for image classification task. Recent studies even show that CNN is having better overall performances than Recurrent Neural Network in sequence processing and prediction tasks such as Natural Language Processing (NLP). In this project, I have both

1. Implemented a CNN with Pytorch with minimal starter code provided by Udacity
2. Applied Transfer Learning with a pretrained CNN architecture called VGG16 by taking advantage of the features learned by it. I only needed to implement one fully connected layer to get the desired dog class prediction

Problem Statement

The objective is to build an image classifier such that when a new image (of a dog) is provided, the classifier can return a breed of the dog, with the help of 8,351 dog images from 133 different breeds. Of course, we would like our classifier to have high accuracy for

prediction. The strategy to solve the problem is to build a Convolutional Neural Network model with the help of transfer learning. In this specific project, I opted for VGG16 since It is a very good architecture for benchmarking on an image classification task. Also, pre-trained networks for VGG are available freely on the internet, so it is commonly used out of the box for various applications.

Metrics

Accuracy on the test set

This is a standard metric for evaluation. For this dataset, this is a strict performance metric because of the fact that we have 133 different classes. The objective of final classifier is 60% test accuracy for the transfer learning model and 10% for the model from scratch. I have been able to achieve both of them.

Categorical Cross Entropy

This is the metric we are minimizing during training. It acts as a surrogate for the accuracy but since it is differentiable with respect to the weights but accuracy is not, hence it is used to update weights of the neural network.

Data Exploration & Exploratory Visualization

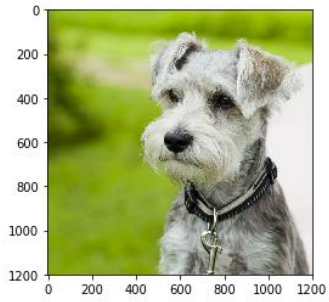
Before I implemented the solution, I explored some random images with OpenCV's implementation of Haar feature-based cascade

classifiers to detect human faces in images and applied VGG16 to tell if an image is a dog. During this part of the exploration, I found that the face detector has a pretty high false positive rate, since 17% of the dog images have been detected as human. It might be an overestimation since some of the dog images do contain human beings. However, it shows that some classifiers might not perform so well even on binary classification on computer vision problems. We might not have a high testing accuracy since we have 133 total breeds of dogs. Assuming the dogs are evenly distributed among the classes, by random guess the accuracy rate would be less than 1%.

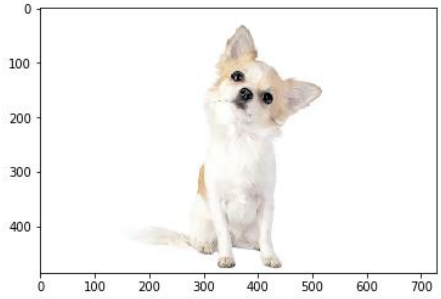
Visual exploration is quite literal for this task. At the end of the project, after a classifier has been built, I have shown several of the images I found on the Internet and printed its most likely dog breed. For example

```
In [119]: own_files = np.array(glob("/home/workspace/dog_project/own_images/*"))

In [120]: for file in own_files:
              run_app(file)
```



```
Breed-Hero-Miniature-Schnauzer-1-1200x1200.jpg
Dogs Detected!
It looks like a...
 Miniature schnauzer
```



```
Chihuahua-longhaired-sitting-its-head-tilted.jpg
Neither human nor dog!
```

Figure 1 – Example of exploratory visualization

With this tool, I am building up more intuition for my dataset and it is quite a fun part in this project!

Algorithms and Techniques

The most exciting part of this project is to apply Transfer Learning by using a pretrained Deep Learning architecture called VGG16. The architecture of VGG16 is quite complicated as can be seen in the following figure

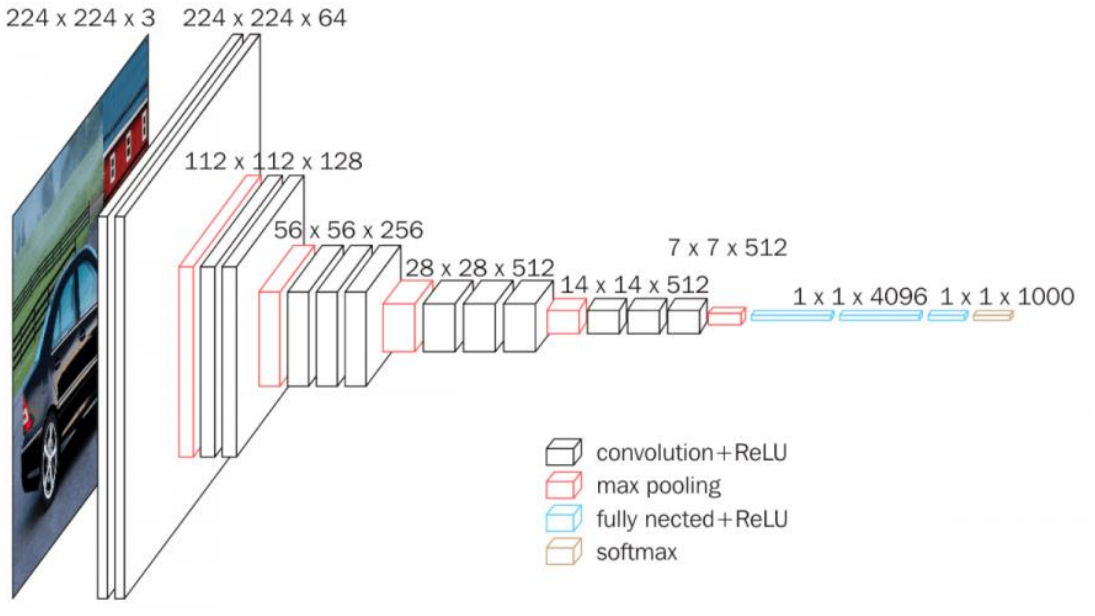


Figure 2 – structure of VGG16

It has various convolutional layers followed by ReLU (Rectified Linear Unit) activation function, max pooling layers, fully connected layers followed by ReLU, and softmax layer which basically transform output into probabilities for each class. Fortunately, we do not need to train most of the weights as they are already the established "good" representations of what an image contains. We only need to pass through our image through a pretrained VGG16, get an output vector of length 4,096, and then build one last fully connected layer with it, yielding 133 dog breeds.

Benchmark

The benchmark model to be compared against is the Convolutional Neural Network that I built from Pytorch from scratch. It only has 3 Convolutional layers with ReLU followed by 2 Fully Connected Layers

with suitable activation functions. This model has achieved a test accuracy rate of 17% after being trained for around 60 epochs (each time I save my model and pick up where I left off by loading the model next time). At first glance, this looks like a terrible accuracy rate, but considering that we have 133 classes and some dogs have minimal distinctions.

Figure 3 – Brittany vs Welsh Springer Spaniel

For example, at least I cannot distinguish between a Brittany and a Welsh Springer Spaniel. And the following pair is yet another example of difficult classification

Figure 4 – Curly-Coated Retriever vs American Water Spaniel

Data Preprocessing

Although deep learning algorithms seem extremely intelligent, frameworks such as Tensorflow and Pytorch are not. First, you need to read in the image and convert it to “RGB” format as shown below, then apply transformations including Resize, CenterCrop and ToTensor in order to convert the image into a multi-dimensional array that Pytorch can recognize. And you cannot convert images to whatever size you would like. For example, pretrained deep learning architectures have very specific requirements for what the valid input size is (224x224x3 for VGG16).

```
from PIL import Image
import torchvision.transforms as T

def load_image(img_path):
    image = Image.open(img_path).convert('RGB')
    # resize to (244, 244) because VGG16 accept this shape
    transform = T.Compose([T.Resize(256), T.CenterCrop(224), T.ToTensor()])

    image = transform(image).unsqueeze(0)
    return image
```

Figure 5 – Preprocessing image and convert it to Tensor

Next, it comes that dataloaders. As shown in the image below, it is a way to apply some transformations to your train/validation/test datasets. The DataLoader itself can be thought as an iterator. In each iteration, it gives the model a batch of 32 (in this example) training examples and the model will adjust its parameters accordingly so that the training loss will be smaller after most of the iterations.

```
import os
from torchvision import datasets

### TODO: Write data loaders for training, validation, and test sets
## Specify appropriate transforms, and batch_sizes
data_dir = '/data/dog_images'

#standard normalization parameters for RGB images
standard_normalization = T.Normalize(mean=[0.485, 0.456, 0.406],
                                      std=[0.229, 0.224, 0.225])

# TODO: Define transforms for the training data and testing data
train_transforms = T.Compose([T.RandomRotation(30),
                              T.RandomResizedCrop(224),
                              T.RandomHorizontalFlip(),
                              T.ToTensor(),
                              standard_normalization])

test_transforms = T.Compose([T.Resize(255),
                             T.CenterCrop(224),
                             T.ToTensor(),
                             standard_normalization])


# Pass transforms in here
train_data = datasets.ImageFolder(data_dir + '/train', transform=train_transforms)
valid_data = datasets.ImageFolder(data_dir + '/valid', transform=test_transforms)
test_data = datasets.ImageFolder(data_dir + '/test', transform=test_transforms)

trainloader = torch.utils.data.DataLoader(train_data, batch_size=32,shuffle=True)
validloader = torch.utils.data.DataLoader(valid_data, batch_size=32,shuffle=True)
testloader = torch.utils.data.DataLoader(test_data, batch_size=32,shuffle=True)
```

Figure 5 – DataLoader example

Implementation

The metrics that I have implemented is simply the test accuracy, as described in previous sections. Algorithm is a pretrained VGG16 neural network followed by a customized fully connected layer to give out probability estimation for each class. The other parts, I basically followed the instructions of the Jupyter Notebook so there is not much to report.

One thing that I would like to mention is that I have added a try-except block in my training code. By applying it, each time I run the

training code, instead of starting from fresh, it actually load the previous model if it exists and then start from there. This is a huge time saver especially when there are some accidents such as the workplace of Udacity has been inactive for a certain period of time and it got timed out. It would potentially be a huge time saver.

```
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

    #Do this so that I don't have to train 100 epochs and see the timeout in workspace
    try:
        model.load_state_dict(torch.load(save_path))
        print("Load from previously trained model")
    except:
        pass
```

Figure 7 – Try-Except block to read the saved model file to avoid retrain

Refinement

The main refinement of this project is to apply the VGG16 model specifically to this use case with Transfer Learning. The initial solution cannot be directly used for this task, but by simply adding one fully connected layer, it will.

Model Evaluation and Validation

The Transfer Learning version of the model, has achieved a test accuracy of 87% in my latest run. This is quite impressive considering the fact that we have 133 classes and the distinctions between some classes are minimal. At the very end, I also included some applications to upload random pictures and then getting a prediction, as shown in

Figure 1.

Justification

By using Transfer Learning, the model has achieved an accuracy rate of 87% on the test dataset, which is a tremendous improvement on 17% of a CNN I built from scratch. This justifies that for most computer vision tasks, it makes a lot of sense to use pretrained deep neural network architectures to save time and dramatically improve accuracy.

GitHub repo: https://github.com/tianhaoluo/Udacity-Dog-Project