Dog Breed Classification

Jean-Sebastien Roger

December 8, 2020

Abstract

In this experiment we examined the task of classifying pictures of dogs; we attempted to classify each dog by its breed based on the information seen in an image. The set of images loaded for this analysis is a custom clustering of various dogs, each sourced from Google images and having an associated label of Husky, Labrador, Poodle, or Pug. For each breed we have multiple examples of the image, examples at different angles, under different lighting conditions, and in different locations. Our goal of this experiment is to utilize a common pre-trained model, built from using an 18-layer residual network (ResNet-18) which has been tuned to recognize and classify images. We trained this model on our dataset of flowers in a previous lab and now we are training the same model on a dataset of dog images and then constructing a visualizer which allows us to present random dog selections along side their classification and prediction.

1 Introduction

More than just training a model to identify pre-loaded images, in this lab we also wanted to have a hand in the construction of the dataset to be used in training and validation. All images were sourced from Google images and downloaded using a Google Chrome extension which downloads all images on a webpage. Once the images were downloaded and unzipped, they were then processed, removing images that were incorrectly labeled or that did not qualify for this image classification experiment. Finally, once the data was curated, it was saved to Google drive where it was split into a training set and validation set.

The dataset utilized for this experiment was located in two folders (a 'train' folder and a 'validation' folder), each filled with 4 sub-directories, each sub-directory of a specific breed of dog with at least 400 example JPEG images. Each image was of a dog breed specific to the folder it was placed in, and the folder was named according to the breed assigned. Figure 1 shows a sample of the images that we have in our training set. Once the model is created, we will look to present some analysis showing the accuracy of the model. Also, we have additional stand alone images that can be fed to the model for prediction.

Figure 1: Shows a random sample of cropped images from our training dataset. In total, we had 1,783 training images and 471 validation images.

2 Theory

Building upon the classification we performed in lab 5 with classifying flower genus, we are now interested in performing the same sort of classification of distinguishing between categorical images, but on a dataset which we have curated. Our image classification task is now dictated by the images we choose, and we should be able to see the impact of dataset size and purity of the images on our accuracy and training time.

We are still faced with a more challenging modeling task than that of lab 4 because rather than identify borders of an image based on the gradient in brightness, we must identify multiple key features within an image that will allow us to associate each picture to a label. In order to match the increased complexity of the problem, we now use more advanced classification techniques built upon neural network models. The model we are using for our experiment is the ResNet-18. This model has been pre-trained, and using the pytorch library and torchvision functions, we are able to further train this model on our training set images and produce a model which can be used to predict breed of dog in our validation set as well as random images outside both training and validation sets. Finally, using this newly trained model, we can construct a visualizer which can stitch together the JPEG image that the model is given as input, the prediction, and the true classification.

In order to prove out the accuracy of our model, we also construct a confusion matrix to show our failure rate by category. This also allows us to take our analysis a bit further, calculating sensitivity, specificity, precision, and even F1 score. All of these calculations are done on the final validation set and help us speak towards the accuracy and rates of success.

3 Procedures

We begin our procedure with a review of the data pre-processing and processing steps, this is the step where we set up the dataset for it to be useful in our experiment. Thanks to the extension "Download All Images" we were able to easily download all images related to searches on specific dog breeds. For example, a quick image search on Husky, provided us with over 1000 images, using the extension we were able to download all the images and then review the selection of husky images until all that were selected were truly of a husky and did not have other misleading objects that could potentially skew the results such as other

dogs breeds, text, people, etc.

Data Import
The next step in the data preparation is splitting the full set of images into a training set and a validation set. In order to do this, we put together a simple loop which will move a file from a source to a destination folder based on a random probability. Below is a code snippet using this loop and splitting the husky image set into a training folder and a validation folder.

```
## Import necessary libraries
import os
import shutil

## Set source and destinations
source = '/content/drive/MyDrive/Data1/Dogbreeddataset/Siberian_Husky'
validation = '/content/drive/MyDrive/Data1/Dogbreeddataset/valid/Husky'
training = '/content/drive/MyDrive/Data1/Dogbreeddataset/train/Husky'

## Define the function to split based on randint()
def split_test_train(source, validation, training):
  for filename in os.listdir(source):
    if(random.randint(1,10) > 8):
      shutil.move(os.path.join(source,filename),
                  os.path.join(validation,filename))
    else:
      shutil.move(os.path.join(source,filename),
                  os.path.join(training,filename))

## Run the function defined above
split_test_train(source, validation, training)
```

Once this function has been run for each dog breed, we can proceed to reading in the images and performing the necessary transformations so the model can correctly interpret the objects in the images and train a classification.

Image standardization and transformation
In the following procedures, we will be using much of the same code showcased in Lab 5 in order to construct a model for our dataset. Here, we are establishing a standard way with which we will randomly crop each input image, convert it into a tensor image, and then normalize the resulting tensor so it can be interpreted by the model easier and the model can converge towards a prediction much faster.

```
# Normalization measurements
mean = np.array([0.485, 0.456, 0.406])
std = np.array([0.229, 0.224, 0.225])

# Establishing transformations
```

```
data_transforms = {
    'train': transforms.Compose([
        transforms.RandomResizedCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize(mean, std)
    ]),
    'valid': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize(mean, std)
    ]),
}
```

Image Import

After reading in libraries and setting up the transformations which will crop and normalize our input images, we can run the below lines of code in order to import the images and apply our normalization and labeling for further classification.

```
image_datasets = {x: datasets.ImageFolder(os.path.join(data_dir, x),
                      data_transforms[x]) for x in ['train', 'valid']
                  }

dataloaders = {x: torch.utils.data.DataLoader(image_datasets[x],
                  batch_size=16,
                  shuffle=True,
                  num_workers=4) for x in ['train', 'valid']
               }

dataset_sizes = {x: len(image_datasets[x]) for x in ['train', 'valid']}

class_names = image_datasets['train'].classes

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

Now that we have the loops created which will load random pictures and their associated labels from our imageset, we are able to define a function (code produced in the Appendix) which we can use to quickly visualize the cropped images that we will be training our model on. Figure 2 is a random selection of these images with their associated labels for classification.

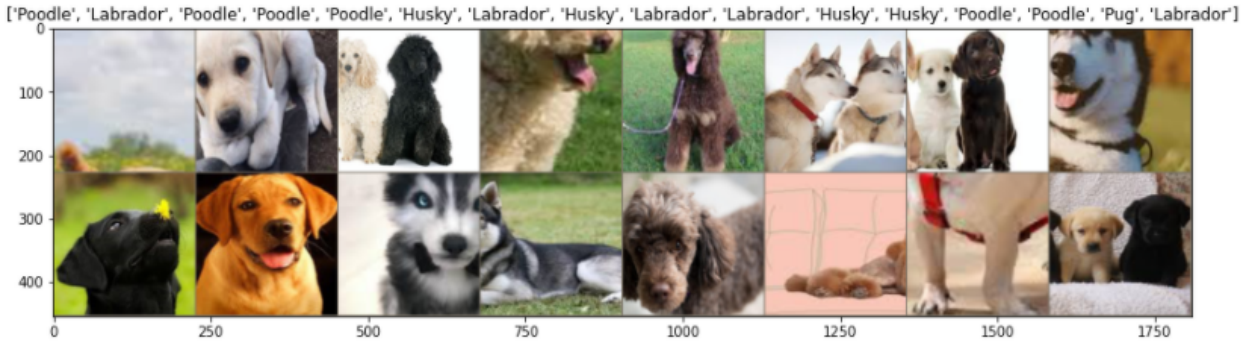

Figure 2: Above are 16 randomly selected images that come from the training set. These images have already been cropped and transformed into tensors which are then normalized.

4 Analysis

In this section we will review the model training, we have moved the package import chunk and various other code chunks to the Appendix.

With our images loaded, cropped, and normalized, we are now prepared to download our ResNet-18 model and train it to make predictions on our prepped dataset. Here we will briefly introduce a key part of the code and discuss calling the model and the training process.

ResNet-18

```
model = models.resnet18(pretrained=True)
num_ftrs = model.fc.in_features
model.fc = nn.Linear(num_ftrs, 102)
model = train_model(model, num_epochs=15)
```

The above chunk of code downloads the pre-trained ResNet-18 model and then performs a linear transformation to the features of the model before training.

Modeling

Once we have loaded the model, we then train the model on our dataset of dog images. The full training definition has been included in the Appendix, however below we show a snippet of the function for the loss criteria used to tune the model, the method of optimization (stochastic gradient descent), and the step function which sets a training schedule using the learning rate and our optimization method.

```
# Specifying our loss function
criterion = nn.CrossEntropyLoss()

# Setting stochastic gradient descent with a learning rate of .001
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)

# Scheduling step function based on descent method and learning rate
scheduler = lr_scheduler.StepLR(optimizer, step_size=7, gamma=0.1)
```

It is important to note that our training set is now iterating to go through 15 epochs, this is because around 15 epochs there is a significant decrease in the additional lift in accuracy with each new iteration. So in order to speed up training time, we were able to decrease the number of epochs from 30 to 15 and we still ended up with a very accurate classification model.

visualizations

Now that our model has been trained and validated, we can proceed to how we wish to present the image and our prediction. Our visualization function is presented in the code chunk below and Figure 3 shows a few samples of the output.

```
def visualize_model(model, num_images=16):
    model.eval()
    index = 0
    for i, (inputs, labels) in enumerate(dataloaders['valid']):
        inputs = inputs.to(device)
        labels = labels.to(device)
        outputs = model(inputs)
        preds = torch.max(outputs, 1)[1]

        for j in range(inputs.size()[0]):
            index += 1
            title1 = 'predicted: ' +
            dataset_labels[int(class_names[preds[j]])-1] +
            '      class: ' +
            dataset_labels[int(class_names[labels[j]])-1]
            imshow(inputs.cpu().data[j], title1)

            if index == num_images:
                return
```

As we can see in Figure 3, we are able to read in a set of JPEG images from a folder (our 'valid' folder) and output a prediction with a single function call. We can see from just this sample how all images seem to be fairly well classified by the model. From images of a full dog body and background, to partial images depicting just the face of a dog, the model can properly classify the breed of the dog in the image.

Figure 3: Validation images alongside a given prediction and true class.

Below we also show how we are able to read a single image taken on a phone and uploaded to Google drive. The two examples below are pictures of my own pets never supplied to the model, after performing a similar tensor transformation in order to normalize the image, we can run our model on these images and output a prediction.

Cooper Test

```
imagePath = '/content/drive/MyDrive/Cooper.jpeg'
image = io.imread(imagePath)
plt.imshow(image);

img = apply_transforms(image).clone().\
        detach().requires_grad_(True).to(device)
outputs = model(img)
preds = torch.max(outputs, 1)[1]
print('predicted: ' + dataset_labels[preds])
predicted:   'Labrador'
```

Figure 4: Cooper - out of sample out of time image

Einstein Test

```
imagePath = '/content/drive/MyDrive/Einstein.jpeg'
image = io.imread(imagePath)
plt.imshow(image);

img = apply_transforms(image).clone().\
        detach().requires_grad_(True).to(device)
outputs = model(img)
preds = torch.max(outputs, 1)[1]
print('predicted: ' + dataset_labels[preds])
predicted:   'Husky'
```

Figure 5: Einstein - out of sample out of time image

Figure 4 and Figure 5 are the pictures I have taken of my own pet and, after applying the model, we see that it correctly classified the breed in both situations

5 Results

With our validation completed and with us having viewed random selections of images and predictions, we can move on to presenting the empirical results of our validation set and seeing how the accuracy of our model holds up against the full validation set.

```
Confusion Matrix :
array([[118,   0,   0,   0],
       [  0, 120,   0,   1],
       [  0,   0, 117,   0],
       [  0,   0,   0, 115]])
```

Figure 6: Confusion matrix showing the true positives and false positives for each breed used within the model

We can see from our confusion matrix that our model does very well at predicting the correct breed of dog, amongst: husky, labrador, poodle, and pug. Within our validation set there was only 1 mis-classification where a pug was incorrectly classified as a labrador. When we begin to calculate the accuracy and other evaluation metrics we see just how accurate the model is. Below is the code used to calculate all of the following metrics from a confusion matrix.

```
from sklearn.metrics import f1_score

cnf_matrix = cm
FP = cnf_matrix.sum(axis=0) - np.diag(cnf_matrix)
FN = cnf_matrix.sum(axis=1) - np.diag(cnf_matrix)
TP = np.diag(cnf_matrix)
TN = cnf_matrix.sum() - (FP + FN + TP)

FP = FP.astype(float)
FN = FN.astype(float)
TP = TP.astype(float)
TN = TN.astype(float)

# Sensitivity, hit rate, recall, or true positive rate
TPR = TP/(TP+FN)
# Specificity or true negative rate
TNR = TN/(TN+FP)
# Precision or positive predictive value
PPV = TP/(TP+FP)
# Negative predictive value
NPV = TN/(TN+FN)
# Fall out or false positive rate
FPR = FP/(FP+TN)
# False negative rate
```

```
FNR = FN/(TP+FN)
# False discovery rate
FDR = FP/(TP+FP)
# Overall accuracy
ACC = (TP+TN)/(TP+FP+FN+TN)
# Global F1 score using totals
F1 = f1_score(preds.reshape(-1),labels, average = 'micro')
```

We have the following summary statistics:

```
1) Accuracy: [1, 0.99787686, 1, 0.99787686]
2) True positives: 470
3) True negatives: 1412
4) False positives: 1
5) False negatives: 1
6) Recall: [1, 0.99173554, 1, 1]
7) Precision: [1, 1, 1, 0.99137931]
8) Specificity: [1, 1, 1, 0.99719101]
9) F1 score: 0.9978768577494692
```

6 Conclusions

This final lab was a good follow up to our previous image classification lab where we were tasked with classifying flower images. Like in lab 5, in this lab we have taken our previous experience dealing with with classifying rice seed image as either 'proper' or 'broken' and expanded on the task of image classification. Here, we have not only built a classifier that will take an image, or set of images, and output a prediction, but we have increased the difficulty of our prediction engine. From a simple threshold calculation up to a complete pre-trained residual network with additional tuning on our dataset. This scales up our prediction capabilities, which now will allow us to output a classification of dog breed after taking into account multiple characteristics within the image.

From this lab, we are able to put together the dataset and the categories we wished to predict, then take a pre-trained model and re-tune it to use our labels and image sets. Using this procedure, we can see the power of residual network models and their ability to identify key characteristics within an image and associate them to a label. The next steps in this lab would be to check our classifications, perhaps we could add additional dog breeds to give more variety within the model. Also, we can speak to the accuracy of the model here. We see from the evaluation metrics that the model is performing very well, with an overall accuracy of 99%. Unlike lab 5, we now have a confusion matrix which helps us have a wider understanding of model performance and gives us empirical metrics which support our claims of accuracy. Another step in this process could be to see which images were mis-classified, and since we have control of our training set, we could fine tune our training set even more to remove noise.

References

[1] He, K., Zhang, X., Ren, S., Sun, J. (2016). Deep Residual Learning for Image Recognition [PDF]. Las Vegas: Institute of Electrical and Electronic Engineers.

[2] "Torchvision.models." Torchvision.models - PyTorch 1.6.0 Documentation

7 Appendix

Package import

```
!pip install git+https://github.com/williamedwardhahn/mpcr
from mpcr import *
!pip install flashtorch
!pip install barbar
from flashtorch.utils import apply_transforms
from flashtorch.saliency import Backprop
import itertools
from sklearn.metrics import confusion_matrix
import os
import shutil
import random
```

Image presentation function

```
def imshow(inp, title = " "):
    fig, ax = plt.subplots()
    inp = inp.numpy().transpose((1, 2, 0))
    inp = std * inp + mean
    inp = np.clip(inp, 0, 1)
    ax.imshow(inp)
    plt.title(title, loc='center')
    fig.set_size_inches(20, 20)
    plt.show()
```

Full training function

```
def train_model(model, num_epochs=30):

    model = model.to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
    scheduler = lr_scheduler.StepLR(optimizer, step_size=7, gamma=0.1)

    for epoch in range(num_epochs):

        print('Epoch: ',epoch+1,'/',num_epochs)
```

```
        ###Train
        model.train()
        running_corrects = 0
        for inputs, labels in Bar(dataloaders['train']):
            inputs = inputs.to(device)
            labels = labels.to(device)

            optimizer.zero_grad()

            outputs = model(inputs)

            preds = torch.max(outputs, 1)[1]
            running_corrects += torch.sum(preds == labels.data)

            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

        print("Train ", 'Acc: {:.2f}'.\
        format(running_corrects.double()/dataset_sizes['train']))

        scheduler.step()

        ###Val
        model.eval()
        running_corrects = 0
        for inputs, labels in Bar(dataloaders['valid']):
            inputs = inputs.to(device)
            labels = labels.to(device)

            outputs = model(inputs)
            preds = torch.max(outputs, 1)[1]
            running_corrects += torch.sum(preds == labels.data)

        print("Valid ", 'Acc: {:.2f}'.\
        format(running_corrects.double()/dataset_sizes['valid']))
        print("##########################")
    return model
```

Final prediction visualizer

```
def visualize_model(model, num_images=16):
    model.eval()
    index = 0
    for i, (inputs, labels) in enumerate(dataloaders['valid']):
        inputs = inputs.to(device)
```

```
labels = labels.to(device)

outputs = model(inputs)

preds = torch.max(outputs, 1)[1]

for j in range(inputs.size()[0]):
    index += 1
    title1 = 'predicted: ' +
    dataset_labels[int(class_names[preds[j]])-1] +
    '      class: ' +
    dataset_labels[int(class_names[labels[j]])-1]

    imshow(inputs.cpu().data[j], title1)

    if index == num_images:
        return
```