Machine Learning Engineer Nanodegree

Capstone Project

Minh Gia Hoang

October 28, 2020

Dog Breed Classifier using CNN

1 Definition

1.1 Project Overview

Dog breed classifier is a representative image classification problem in the computer vision (CV). Classifying dogs into their breeds are often challenging even for human by simply looking at them as there are hundreds of distinct breeds. Thus, it needs to be done by computer instead. Solving this problem is useful when rescuing dogs, finding them homes, treating them, etc. or just simply seeing a cute dog and wondering which breed the dog belonged. Given an input image, the first task is to identify if it is a dog image or a human image. If it is a dog image then classify it to one breed of dog. If the image is a human face then identify the most resembling dog breed associated with the face.

To solve the multi-class classification problem in CV, convolutional neural networks (CNNs) have been widely used for their efficiency to deal with 2-D data such as images. CNN can capture the spatial dependencies of pixels in an image by applying 2-D convolution filters. The architecture is suitable for image datasets since a small set of parameters (the kernel) is used to compute outputs of the entire image, so the model has much fewer parameters compared to a fully connected layer. Put differently, CNN can be trained to understand the sophistication of the image better than feedforward neural networks. Thus, CNN will be employed to accomplish this project.

Besides, some different approaches have been explored for this dog breed classifier. The table below summarizes the benchmarks of some early attempts on the Stanford Dogs dataset (Fei-Fei, 2011).

Table 1: Summary of Benchmarks of Stanford Dogs (Hsu, 2015).

Method	Top – 1 accuracy
SIFT + Gaussian kernel (Fei-Fei, 2011)	22%
Unsupervised learning template (Shapiro, 2012)	38%
Gnostic fields (Han, 2015)	47%
Selective pooling vectors (Kanan, 2014)	52%

1.2 Problem Statement

The objective of the project is to build a pipeline that can be used within a web or mobile app to process real-world, user-supplied images. There are two main tasks:

- Human face detector: If given a human face image, the model will predict the most resembling dog breed associated with the face.
- Dog breed detector: If given a dog image, the model should predict its breed.

If neither a dog nor a human is detected, the app outputs an error message. Thus, the model should be able to recognize a dog or a human face in an image then classify the dog according to its breed or provide an estimate of the dog breed that the human face resembles.

1.3 Metrics

Accuracy is one metric used to test models required by the project. In particular, a CNN built from scratch should get more than 10% accuracy while a pre-trained model should achieve an accuracy of more than 60%.

However, in the following section, we will demonstrate that the dog dataset is imbalanced through class label distribution. Due to class imbalance in the dataset, besides accuracy, precision and recall would be appropriate choices for evaluating the proposed model.

The metrics such as accuracy, precision, and recall can be derived from the confusion matrix.

Table 2: An illustrated confusion matrix.

Actual / Predicted	Negative	Positive
Negative	True Negative (TN)	False Positive (FP)
Positive	False Negative (FN)	True Negative (TP)

From the confusion matrix, we have:

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN},$$
$$precision = \frac{TP}{TP + FP},$$
$$recall = \frac{TP}{TP + FN}.$$

2 Analysis

2.1 Data Exploration and Visualization

Human images are distributed in 5749 folders named after human names such as Jonathan Tiomkin, Charlie Williams, etc. There are 13233 human face images. Images are not evenly distributed among the folders.

The dog dataset is structured into a training set, a validation set, and a test set. Each set comprises 133 breeds of dogs organized into subdirectories by breed name. Later, we will see that the number of images per dog breed is imbalanced. There are 6680 dog images in the training set, 835 images in the validation set, and 836 images in the test set.

```
import os

# Data Exploration
print('Number of folders in /lfw:', len(glob('/data/lfw/*')))
print('Number of human images:',len(glob("/data/lfw/*/*")))
print('Numner of folders in /dog_images:', len(glob('/data/dog_images/*')))
print('Folder names in /dog_images:', end=' ')
print(*[x.split('/')[-1] for x in glob('/data/dog_images/*')], sep=', ')
print('Number of folders (breed classes) in /dog_images/train:', len(glob("/data/dog_images/train/*")))
print('Number of images in /dog_images/train:', len(glob("/data/dog_images/train/*/*")))
print('Number of folders (breed classes) in /dog_images/valid:', len(glob("/data/dog_images/valid/*")))
print('Number of images in /dog_images/valid:', len(glob("/data/dog_images/valid/*/*")))
print('Number of folders (breed classes) in /dog_images/test:', len(glob("/data/dog_images/test/*")))
print('Number of images in /dog_images/test:', len(glob("/data/dog_images/test/*/*")))
```

```
Number of folders in /lfw: 5749
Number of human images: 13233
Numner of folders in /dog_images: 3
Folder names in /dog_images: train, test, valid
Number of folders (breed classes) in /dog_images/train: 133
Number of images in /dog_images/train: 6680
Number of folders (breed classes) in /dog_images/valid: 133
Number of images in /dog_images/valid: 835
Number of folders (breed classes) in /dog_images/test: 133
Number of images in /dog_images/test: 836
```

Let's look at some images from the human and dog datasets.

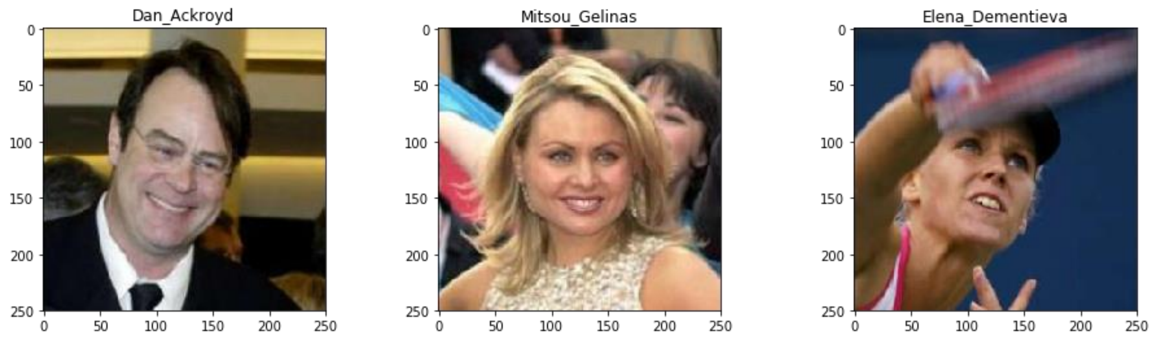


Figure 1: Samples of images in the human dataset.

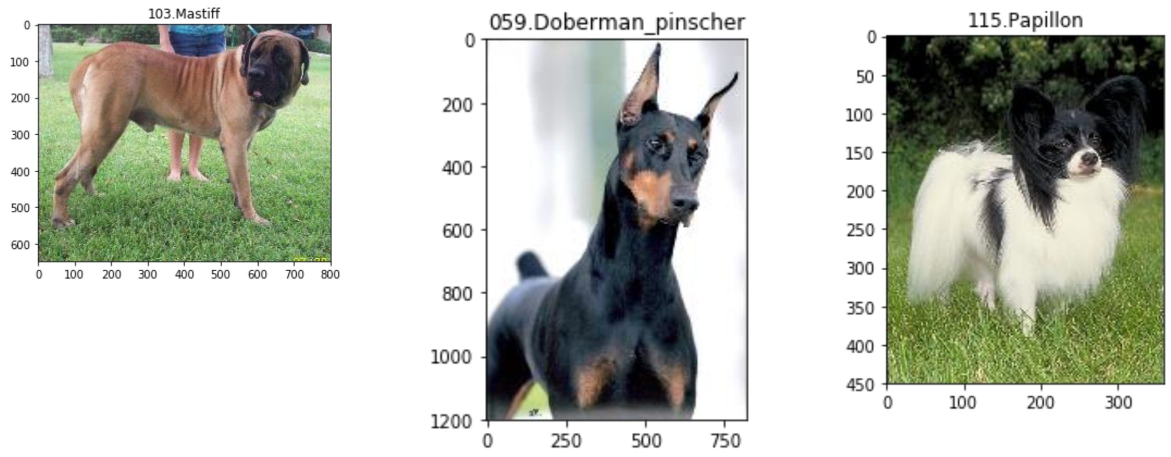


Figure 2: Samples of images in dog dataset.

Human images seem to have the same size 250 x 250, whereas variations in image sizes throughout the dog datasets are high. Thus it is needed to resize all dog images so that they all have the same width and height.

We are interested in the distribution of dog breeds in training and validation sets in the following figure (averages were about 50.22 and 6.28 samples per class for the training set and the validation sets, respectively):

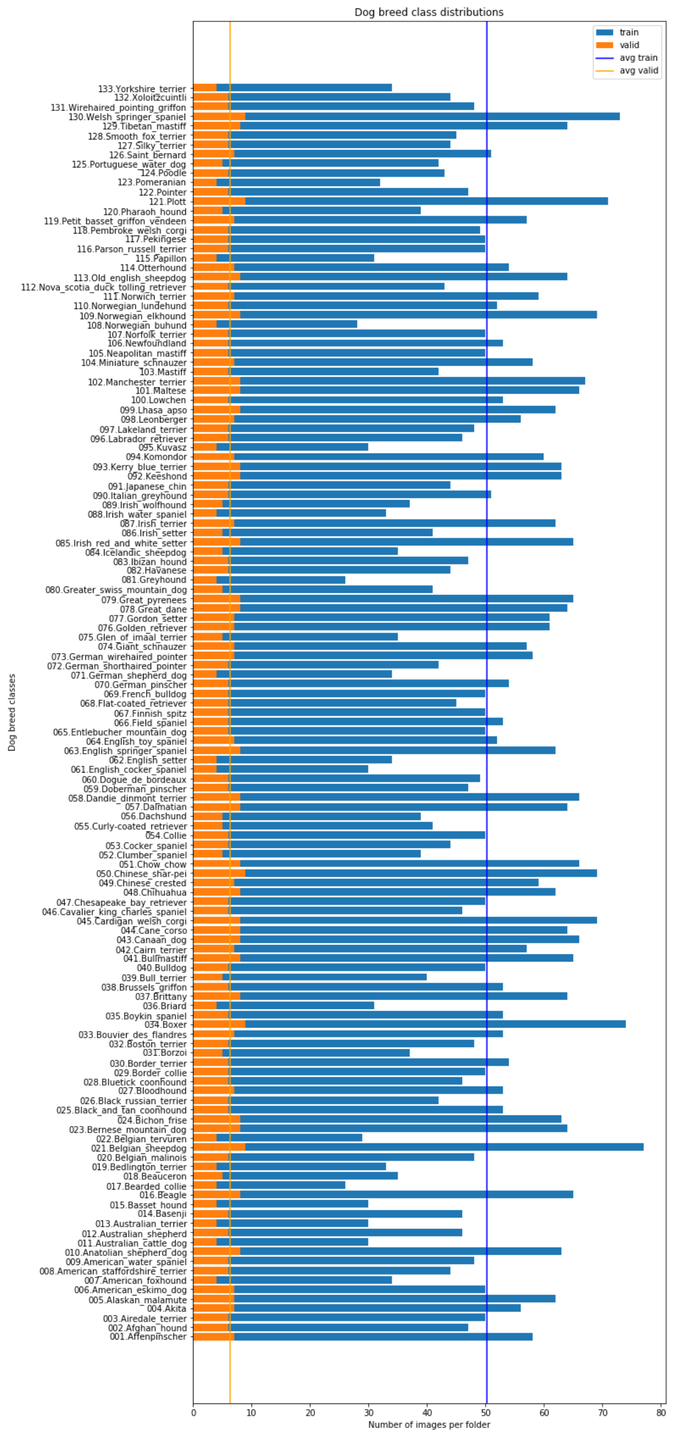


Figure 3: Dog breed distribution.

2.2 Algorithms and Techniques

To solve the multi-class classification problem in CV, CNNs have been widely adopted. CNN can capture the spatial dependencies of pixels in an image by applying 2-D convolution. The architecture is suitable for image datasets since a small set of parameters (the kernel) is used to compute outputs of the entire image, so the model has much fewer parameters compared to a fully connected layer. Put differently, CNN can be trained to understand the sophistication of the image better than feedforward neural networks. The solution involves 3 stages:

- Human face detection using OpenCV's implementation of Haar feature-based cascade classifiers.
- Dog detection using the VGG-16 model, along with weights that have been trained on ImageNet.
- Dog breed classification using first a CNN built from scratch, then trying transfer learning with pre-trained models from ImageNet competition to significantly models boost the accuracy to meet the requirements of the project. Some good candidates are VGGNet and Residual Network (or ResNet).

Besides, data augmentation is also adopted to extend a dataset and improve generalization to avoid overfitting. The used algorithms and techniques will be discussed in detail in the methodology section.

2.3 Benchmark

A CNN is built from scratch and should get more than 10% accuracy. This accuracy can confirm that the model is working since a random guess yields a correct answer about 1 in 133 times, which corresponds to an accuracy of less than 1%. The details

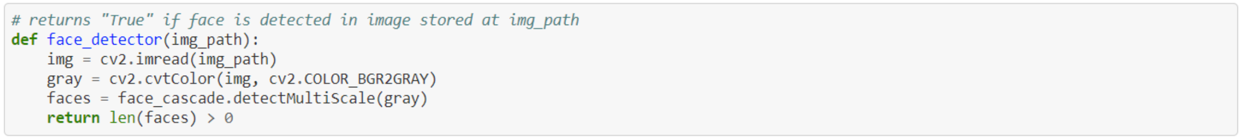
3 Methodology

3.1 Human Detection

A Haar feature-based cascade classifier is used to detect human faces. It is a machine learning-based approach where a cascade function is trained from a lot of positive (with a human face) and negative (without human face) images. OpenCV library provides many pre-trained detectors including this classifier.

```
face_cascade = cv2.CascadeClassifier('haarcascades/haarcascade_frontalface_alt.xml')
```

The images are first converted to greyscale before being passed to a face detector. The method `face_cascade.detectMultiScale` returns a list of 4 bounding box coordinate values for all detected faces in the same image. The function `face_detector` takes a string-valued file path to an image as input and returns True if a human face is detected in an image and False otherwise appearing in the code block below.



```
# 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
```

3.2 Dog Detection

A pre-trained VGG-16 model is used to detect dogs. This model is trained on ImageNet, which is a large and popular dataset used for image classification tasks. ImageNet has more than 10 million URLs where each URL links to an image containing an object from one of 1000 categories. The input to the VGG-16 model is a fixed-size 224 × 224 RGB image. The preprocessing is subtracting the mean RGB value, computed on the training set, from each pixel. The function `VGG16_predict` in the code block below uses the pre-trained VGG-16 model to obtain an index corresponding to the predicted ImageNet class for the inputted image path.

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

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
    transform = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
    ])

    img = Image.open(img_path).convert('RGB')
    img = transform(img)
    img = img.unsqueeze(0)

    if use_cuda:
        img = img.cuda()

    VGG16.eval()
    output = VGG16(img)

    return torch.max(output,1)[1].item() # predicted class index
```

Dog breeds occur consecutively on a dictionary from ImageNet from keys 151 to 268 inclusively, that is, from “Chihuahua” to “Mexican hairless”. Hence, the VGG-16 model is excepted to return an index between 151 to 268 (inclusive). Otherwise, it means no dog detected.

```python
### 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)
    return index >= 151 and index <= 268 # true/false
```

3.3 Dog breed classification

3.3.1 Data Preprocessing

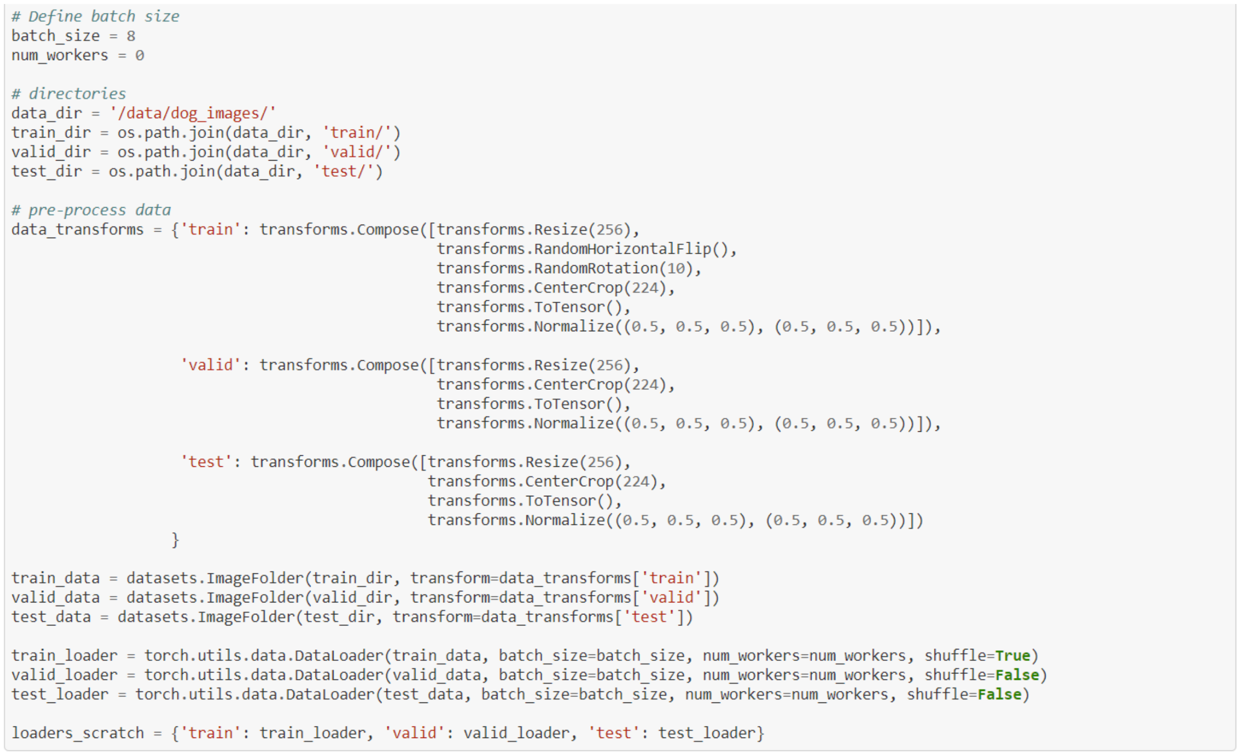
In this section, we present three separate data loaders for the training, validation, and test datasets of dog images. Depending on the sets, different procedures for data preprocessing can be applied. For example, we perform data augmentation on the training set only but the 3 sets still share some particular image processing steps to feed the models.

In the original VGG-16 paper (Zisserman, 2015), the authors used fixed-size 224 x 224 images as the input. Thus we rescaled the original images to 256 x 256 before cropping to get 224 x 224 images. The image rescaling is critical since cropping a 224 x 224 image out of a much larger original is unlikely to contain the features we are interested in to train the model. Images are also normalized in all channels.

Data augmentation helps to increase the size of the training dataset and improve generalization to avoid overfitting. Therefore, we augmented the dataset using random horizontal flipping and random rotation up to 10 degrees. They provided dog images are in the correct position so we can flip them horizontally or slightly rotate them to force the model to be more robust to variations in the position and orientation.

In each iteration, the data loader returns one batch of data, with the given batch size which is first set to 8 after some tunning. If shuffle is set to True, it shuffles the training data before creating batches. Shuffling helps randomize the input to the optimization algorithm, which can lead to a faster reduction in the loss.

The data preprocessing is done by the code block below.



```python
# Define batch size
batch_size = 8
num_workers = 0

# directories
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/')

# pre-process data
data_transforms = {'train': transforms.Compose([transforms.Resize(256),
                                                transforms.RandomHorizontalFlip(),
                                                transforms.RandomRotation(10),
                                                transforms.CenterCrop(224),
                                                transforms.ToTensor(),
                                                transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))]),

                   'valid': transforms.Compose([transforms.Resize(256),
                                                transforms.CenterCrop(224),
                                                transforms.ToTensor(),
                                                transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))]),

                   'test': transforms.Compose([transforms.Resize(256),
                                               transforms.CenterCrop(224),
                                               transforms.ToTensor(),
                                               transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
                  }

train_data = datasets.ImageFolder(train_dir, transform=data_transforms['train'])
valid_data = datasets.ImageFolder(valid_dir, transform=data_transforms['valid'])
test_data = datasets.ImageFolder(test_dir, transform=data_transforms['test'])

train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, num_workers=num_workers, shuffle=True)
valid_loader = torch.utils.data.DataLoader(valid_data, batch_size=batch_size, num_workers=num_workers, shuffle=False)
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size, num_workers=num_workers, shuffle=False)

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

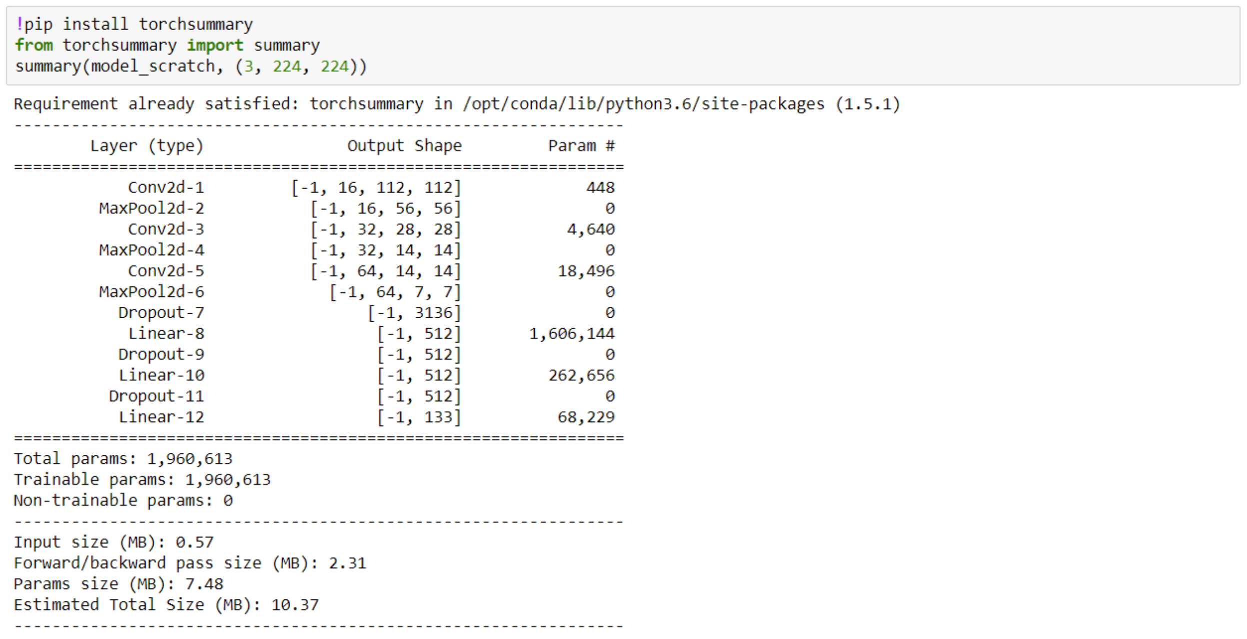
3.3.2 Implementation

We built a CNN from scratch to classify dog breeds accordingly with the problem statement. The scratch model is structured as follows:

- 3 convolutional (conv) layers transform a 3-channel image to a 64-channel feature map. All conv layers have a kernel size of 3 and a padding of 1 (to preserve spatial resolution after convolution). The first 2 conv layers have stride 2 while the third one has stride 1.

- Max-pooling is performed over a 2 x 2-pixel window, with stride 2 after conv layers which halves the height and width. The feature map gets smaller as we add more layers until we are finally left with a small feature map, which can be flattened into a vector.
- 3 fully connected layers at the end produce an output vector of size 133 for each image.
- 3 dropout layers of 0.25 are added to avoid overfitting.
- All hidden layers are equipped with the rectification (ReLU) non-linearity.

The summary of the scratch model is as follows.



```
!pip install torchsummary
from torchsummary import summary
summary(model_scratch, (3, 224, 224))
```

```
Requirement already satisfied: torchsummary in /opt/conda/lib/python3.6/site-packages (1.5.1)
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 16, 112, 112]             448
         MaxPool2d-2           [-1, 16, 56, 56]               0
            Conv2d-3           [-1, 32, 28, 28]           4,640
         MaxPool2d-4           [-1, 32, 14, 14]               0
            Conv2d-5           [-1, 64, 14, 14]          18,496
         MaxPool2d-6             [-1, 64, 7, 7]               0
           Dropout-7                 [-1, 3136]               0
            Linear-8                  [-1, 512]       1,606,144
           Dropout-9                  [-1, 512]               0
           Linear-10                  [-1, 512]         262,656
          Dropout-11                  [-1, 512]               0
           Linear-12                  [-1, 133]          68,229
================================================================
Total params: 1,960,613
Trainable params: 1,960,613
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.57
Forward/backward pass size (MB): 2.31
Params size (MB): 7.48
Estimated Total Size (MB): 10.37
----------------------------------------------------------------
```

For the classification problem, a cross-entropy loss is adopted. Besides, we also use the following hyperparameters to train the model.

Table 3: Hyperparameters for the scratch model.

Hyperparameter	**Values**
Number of epochs	20
Optimizer	stochastic gradient descent (SGD)
Batch size	8
Learning rate	0.02

This presented model is used as a benchmark model. In Section 4.1, we will show that this model is working and has an accuracy better than the 10% required by this project.

Finally, we would present a non-working model. This model is rather similar to the model above except it has only one conv layer. We tried this model based on the review of our project proposal.

The reviews suggest that *"If you identify that your data is strongly imbalanced, you could set as a benchmark the simplest model. For instance, CNN with 1 Convolutional Layer and use other metrics instead of accuracy that distinguishes correctly between the numbers of correctly classified examples of different classes, as you mentioned (precision and recall)."*

We found that it is difficult to train the suggested CNN architecture because a single conv and followed by max-pooling are not enough to have a small feature map, which can be flattened into a vector. The code block below illustrates this problem where the input of the first fully connected layer is a vector of size 102764544!

```
summary(model_scratch2, (3, 224, 224))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 8, 112, 112]             224
         MaxPool2d-2            [-1, 8, 56, 56]               0
           Dropout-3               [-1, 25088]               0
            Linear-4                [-1, 4096]     102,764,544
           Dropout-5                [-1, 4096]               0
            Linear-6                [-1, 1024]       4,195,328
           Dropout-7                [-1, 1024]               0
            Linear-8                 [-1, 133]         136,325
================================================================
Total params: 107,096,421
Trainable params: 107,096,421
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.57
Forward/backward pass size (MB): 1.23
Params size (MB): 408.54
Estimated Total Size (MB): 410.34
----------------------------------------------------------------
```

3.3.3 Refinement

The CNN model built from scratch can be improved significantly by transfer learning. The idea is to reuse the lower layers of a pre-trained model for feature extraction and only train a few higher layers for classification as depicted in Figure 4. Modern CNNs training on huge datasets like ImageNet have great performance.

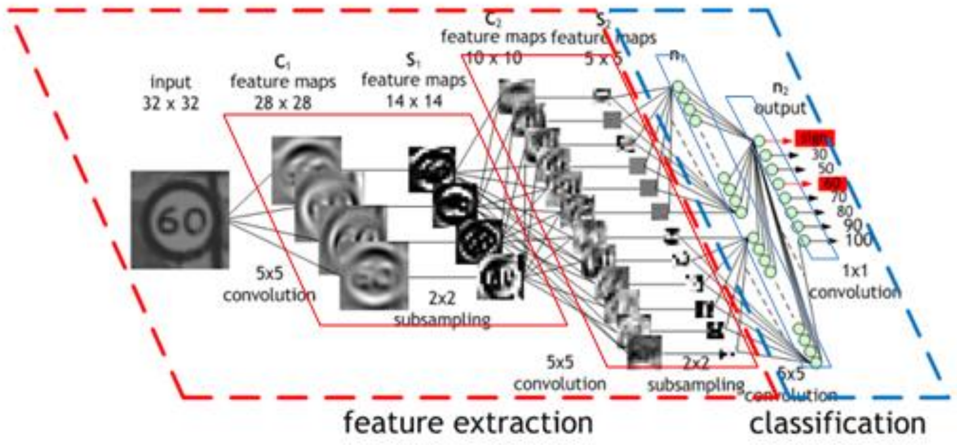


Figure 4: How CNN learns. Image by Maurice Peemen.

Here, we are interested in a couple of state-of-the-art pre-trained architectures VGGNet (Zisserman, 2015) and (K. He, 2016) as a fixed feature extractor for our models. The new hyperparameters to train the models are summarized in the table below.

Hyperparameter	**Values**
Number of epochs	20
Batch size	32
Learning rate	0.001

On the one hand, VGGNet is worth trying because it has a very simple and classical architecture and has great performance, coming in second in the ImageNet 2014 competition. VGGNet is already adopted in the dog detection task and thus can be reused. The idea here is that we keep all the convolutional layers, but replace the final fully-connected layer with our classifier. This way we can use VGGNet as a fixed feature extractor for our images then easily train a simple classifier on top of that. The following code block implements our model.

```
import torchvision.models as models
import torch.nn as nn

## TODO: Specify model architecture
# Load the pretrained model from PyTorch
model_transfer = models.vgg16(pretrained=True)

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

n_inputs = model_transfer.classifier[6].in_features

# add last linear layer (n_inputs -> 133 dog breed classes)
# new layers automatically have requires_grad = True
model_transfer.classifier[6] = nn.Linear(n_inputs, 133)

# print out the model structure

if use_cuda:
    model_transfer = model_transfer.cuda()
```

On the other hand, ResNet is the winner of the ImageNet 2015 competition and has amazing performance on image classification. It helps to train very deep networks by using skip connections (or shortcut connections). Specifically, it adds the original input back to the output feature map obtained bypassing the input through one or more conv layers as shown in Figure 5. This seemingly small change produces a drastic improvement in the performance of the model.

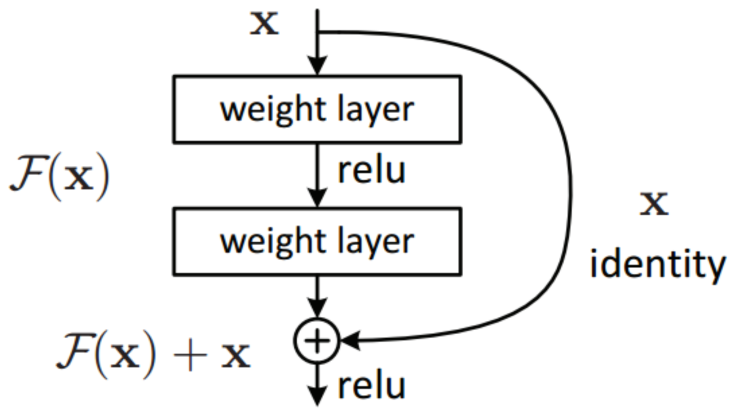

x
weight layer
relu
weight layer
$\mathcal{F}(\mathbf{x})$
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

Figure 5: Residual learning (K. He, 2016).

We will use the 101-layer-deep ResNet architecture (ResNet101) pre-trained on the ImageNet dataset. Similar to VGGNet, we only need to replace the final fully-connected layer with our classifier to produce 133-dimensional output. The following code block implements the solution.

```
model_transfer2 = models.resnet101(pretrained=True)

for param in model_transfer2.parameters():
    param.requires_grad = False

model_transfer2.fc = nn.Linear(2048, 133)

if use_cuda:
    model_transfer2 = model_transfer2.cuda()

criterion_transfer2 = nn.CrossEntropyLoss()
optimizer_transfer2 = optim.Adam(model_transfer2.fc.parameters(), lr=0.001)
```

4 Results

4.1 Model Evaluation and Validation

4.1.1 Human detection

The human face detector based on OpenCV's implementation of Haar feature-based cascade classifiers has 98% of humans detected in the first 100 human images and 17% of humans detected in the first 100 dog images.

4.1.2 Dog detection

The dog detector based on the pre-trained VGG-16 model has 100% dog detected in the first 100 dog images and 0% of dog detected in the first 100 human images.

4.1.3 Dog breed classification

The model performances are summarized in Table 4. Due to an imbalanced dataset, precision and recall metrics are also considered besides accuracy.

Table 4: Model performance comparison on the test dataset.

Model	Accuracy	Precision	Recall
Benchmark	13%	12%	12%
VGG-16	83%	85%	83%
ResNet101	85%	87%	84%

The 13% accuracy of the benchmark model meets the requirement of 10%. Besides, its precision and recall are not too bad considering a simple CNN model and a small number of training epochs. As expected, the pre-trained models using VGG-16 and ResNet101 performs very well on the test dataset with all metrics exceeding 80%. The ResNet101 is slightly better than the VGG-16 so we choose ResNet101 as the final solution for this project.

4.2 Justifications

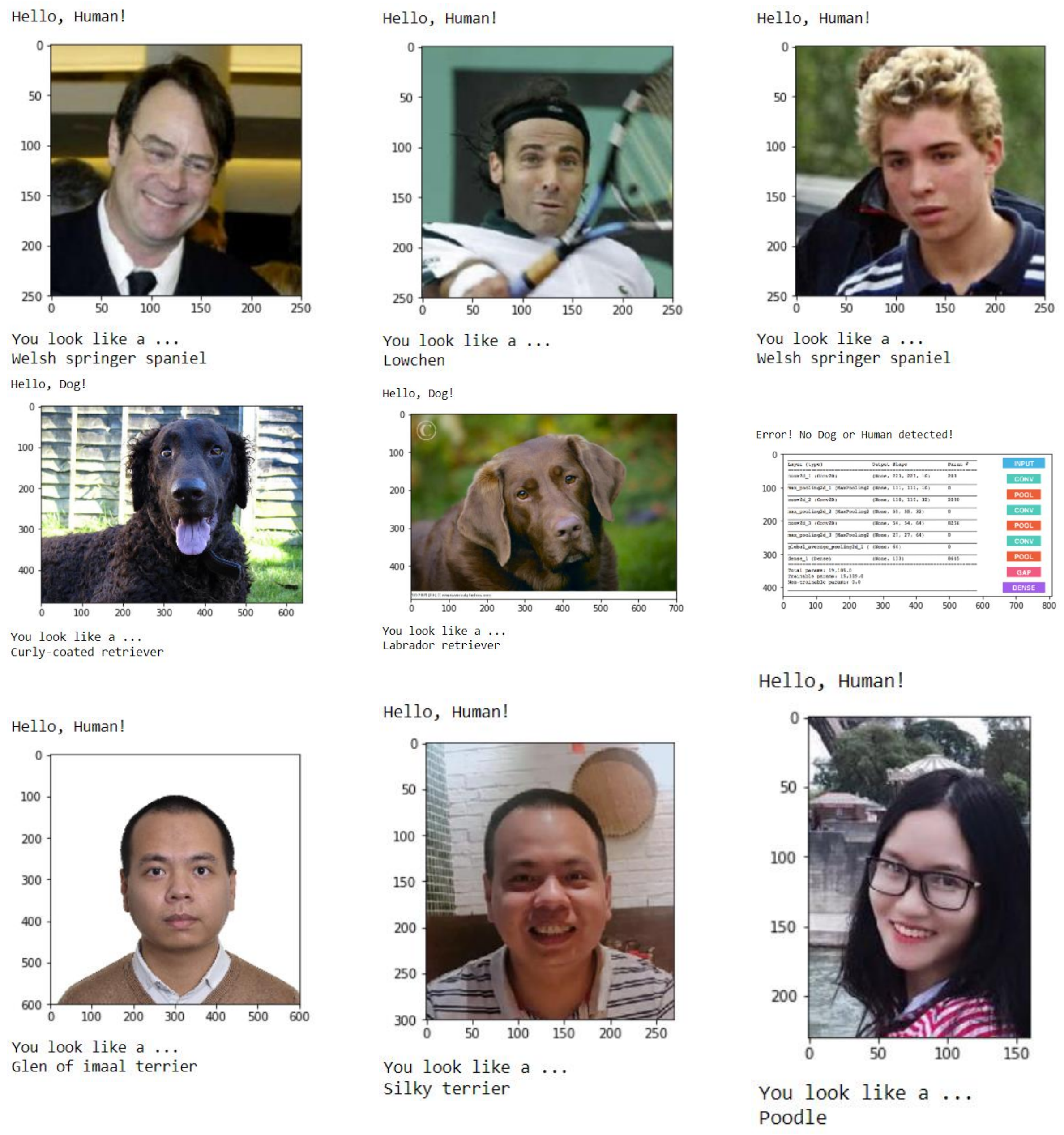
The final solution (ResNet101) has far exceeded the benchmark by accuracy (85% vs 13%), precision (87% vs 12%), and recall (84% vs 12%), so we can conclude that the final solution is significant enough to have adequately solved the dog breed classification problem.

Now we adopt this model in our dog app. This app 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 an output that indicates an error.

Let us see some results!



```
Hello, Human!
You look like a ...
Welsh springer spaniel

Hello, Human!
You look like a ...
Lowchen

Hello, Human!
You look like a ...
Welsh springer spaniel

Hello, Dog!
You look like a ...
Curly-coated retriever

Hello, Dog!
You look like a ...
Labrador retriever

Error! No Dog or Human detected!

Hello, Human!
You look like a ...
Glen of imaal terrier

Hello, Human!
You look like a ...
Silky terrier

Hello, Human!
You look like a ...
Poodle
```

Figure 6: Sample outputs predicted using our dog app.

Overall, the dog app works rather well. It can detect human, dog, or neither in the given images. Then it provides estimates on the breed for the dog image, the resembling dog breed for the human image, or returns an error message.

5 Conclusions and Perspectives

In this project, we built a dog app to process real-world, user-supplied images. Given an image of a dog, the app will identify an estimate of its breed. If supplied with an image of a human, the app will identify the resembling dog breed. Our models perform very well and can classify the dog breed with an accuracy of 85% and a precision of 87%. To achieve this, we have explored quite a lot of practical topics in machine learning, such as data exploration and visualization, data preprocessing and augmentation, state-of-the-art CNN architecture for classification, etc. We also learned how to assemble a series of models designed to perform various tasks in a data processing pipeline.

There is a lot of room to improve our model performance and reduce training time:

- Try to increase the training data by other data augmentation techniques like random shift, random size, generated images with various contrasts, etc.
- Try other variants of ResNet or different CNN architectures (GoogLeNet, Xception, SENet – winner of ImageNet 2017, etc.) or a combination of CNN models.
- Playing with hyperparameters like batch size, learning rate, optimizer, number of epochs, activation function, etc.
- Instead of using a fixed learning rate, try to use a learning rate scheduler, which will change the learning rate after every batch of training.

6 Bibliography

Fei-Fei, A. K. (2011). Novel Dataset for Fine-Grained Image Categorization. *First Workshop on Fine-Grained Visual Categorization, IEEE Conference on Computer Vision and Pattern Recognition.* Colorado Springs, CO.

Han, G. C. (2015). Selective Pooling Vector for Fine-Grained Recognition. *IEEE Winter Conference on Applications of Computer Vision*, (pp. 860-867).

Hsu, D. (2015). *Using Convolutional Neural Networks to Classify Dog Breeds.* Stanford University.

K. He, X. Z. (2016). Deep Residual Learning for Image Recognition. *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (pp. 770-778). Las Vegas, NV: IEEE.

Kanan, C. (2014). Fine-grained object recognition with Gnostic Fields. *IEEE Winter Conference on Applications of Computer Vision*, (pp. 23-30).

Shapiro, S. Y. (2012). Unsupervised Template Learning for Fine-Grained Object Recognition. *Advances in Neural Information Processing Systems 25*, (pp. 3122-3130).

Zisserman, K. S. (2015). Very Deep Convolutional Networks for Large-Scale Image Recognition. *International Conference on Learning Representations.*