Machine Learning Engineer Nanodegree

Capstone Report

Spatika Narayanan
February 7th, 2021

I. Definition

Project Overview

This project will aim to build an algorithm that takes as input a given image, and as output, will determine whether the image contains the face of a human, a dog, or neither. Further, if a human face or dog is detected, the closest resembling dog breed will be output. This problem falls into a domain called image classification, and can be solved using machine learning.

Image Classification Background

Image classification is an example of a supervised learning problem within machine learning – based on training with labelled input data fed to the model and a set of target classes, we expect our model to recognize a new image and classify it correctly.

Though several techniques and types of ML models can be used for image classification, we will use CNN or Convolutional Neural Networks. A huge advantage of CNN is that instead of us defining and engineering features from the raw pixels in order for the model to learn, the network can itself extract and learn features from the image pixels.

CNN Background

Convolutional Neural Networks is a type of machine learning model developed by computer scientist, and later Turing Award-Winner, Yann LeCun in 1988 [1]. It is a type of Artificial Neural Network inspired by the workings of the visual cortex. They are so called, because of their application of a mathematical operation called 'convolution' as opposed to matrix multiplication, in at least one of their layers – i.e. they have at least one 'convolutional layer'. Like other types of neural networks, CNNs consist of input, output and hidden layers (convolutional, pooling, batch normalization and fully connected layers).

The dataset for dog breed image classification has been provided for download in [2] with both dog images and human images, since our pipeline should also detect human images.

Problem Statement

Our specific problem of dog breed image classification has been studied and explored through Kaggle Competitions [3] and in several academic works [4, 5, 6]. Several of these apply CNN as a primary technique with significant classification accuracy. This is a multi-class classification problem, since, as we know, there are definitely more than just two dog breeds. This poses a challenge when it comes to training the model – each breed must be

appropriately represented, and have diversity in images. Since we have defined this as a multi-class classification problem, there exist many possible ML models that are applicable and relevant metrics for each.

But we have already been given the project template code in the form of a iPython notebook on GitHub [2]. As stated in [2], the project goal is to: "…learn how to build a pipeline that can be used within a web or mobile app to process real-world, user-supplied images. Given an image of a dog, your algorithm will identify an estimate of the canine's breed. If supplied an image of a human, the code will identify the resembling dog breed."

We will train our CNN using the dog images and this will form the basis of our breed predictor. We use another pre-trained CNN model to detect whether an image is of a dog or not. Similarly, we use a pre-trained OpenCV model to detect human faces.

The given template notebook [2] describes the sub-tasks needed to complete the project, using CNN as the solution to this classification problem. We will complete sub-tasks so we have functions/modules to:

1. Detect whether a given image is a human face or not using OpenCV
2. Classify a dog breed from a given image using a pre-trained PyTorch CNN model
3. Detect whether a given image is a dog or not, using a pre-trained model, and it's prediction dictionary

The pipeline solution will be as follows for a new image: first module 3 and module 1 will be used to detect whether the image is that of a dog, human or neither. If it is a dog, use module 2 to predict breed else if human, we still use module 2 predictor to output which dog breed the human looks most like. In the case where neither module 3 nor 1 has detected human/dog, the output will indicate error.

The results we are expecting is correct classification of human/dog categories, and if it is a dog image, correct classification of the dog's breed, as far as possible. We can measure model's ability to do this using several metrics, which we discuss in the next section.

Metrics

For multi-class classification problems, **test accuracy %** is a reasonable and commonly used evaluation metric.

$$\%\ Accuracy = \frac{No.\ of\ Correct\ Predictions}{Total\ Number\ of\ Predictions} * 100$$

This should be calculated on the test set – data that the model has not seen before -- to prevent data leakage. A minimum of 60% accuracy is expected for the final CNN dog classification model.

If we are dealing with an imbalanced dataset, however, we must look at other options. Firstly, **precision**: what are the number of true positives with respect to all the 'positives' predicted?

$$Precision = \frac{True\ Positives}{True\ Positives + False\ Positives}$$

However, optimizing this metric doesn't take into account false negatives – those predictions that should have been predicted as a positive. For this, next we have **recall**:

$$Recall = \frac{True\ Positives}{True\ Positives + False\ Negatives}$$

But what would be best is the trade-off between these two metrics [7] the **F1-Score**: it is the harmonic mean of precision and recall. So even if the accuracy is mis-leadingly high for an extremely skewed class (by just predicting, our F1-Score would be 0. It can range between 0 and 1, with values closer to 1 indicating better performance.

$$Recall = 2 * \frac{Precision * Recall}{Precision + Recall}$$

We will decide the metric on which we grade our model performance based on the exploration done in the next section.

II. Analysis

Data Exploration

As mentioned earlier, dataset is provided at [1]. Running through the first few cells of the given project template, we get an overview of our dataset and see that there are 13233 human images and 8351 dog images provided.

```
In [1]: import numpy as np
        from glob import glob

        # load filenames for human and dog images
        human_files = np.array(glob("/data/lfw/*/*"))
        dog_files = np.array(glob("/data/dog_images/*/*/*"))

        # print number of images in each dataset
        print('There are %d total human images.' % len(human_files))
        print('There are %d total dog images.' % len(dog_files))

        There are 13233 total human images.
        There are 8351 total dog images.
```

A simple exploration of the dog_images folder provided, shows that it has already been split into training, validation and testing sets.

We can see sample data for each of the 3 folders. We see that each of train, test and validation data sets have RGB images of varying sizes: e.g. 480x360 pixels or 334x250 pixels.

```
In [31]: from PIL import Image
         from matplotlib.pyplot import imshow

         sample_train_data = train_data.__getitem__(1)

         print(sample_train_data[0])
```

```
<PIL.Image.Image image mode=RGB size=480x360 at 0x7FC430661C50>
```

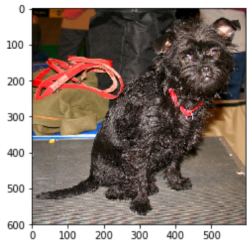
```
In [32]: imshow(sample_train_data[0])
```

```
Out[32]: <matplotlib.figure.AxesImage at 0x7fc4305db0b8>
```


```
<PIL.Image.Image image mode=RGB size=596x600 at 0x7FC4305F6F60>
```

```
: imshow(sample_test_data[0])
```

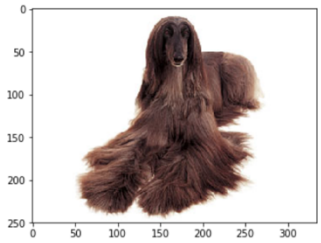
```
: <matplotlib.figure.AxesImage at 0x7fc4304ce860>
```




```
<PIL.Image.Image image mode=RGB size=334x250 at 0x7FC436237FD0>
```

```
: imshow(sample_valid_data[0])
```

```
: <matplotlib.figure.AxesImage at 0x7fc4304ae4a8>
```



We also see the size of each data set: test data has 6680 images, with 835 and 836 respectively for test and validation data.

```
: from torchvision import datasets
  train_data = datasets.ImageFolder(root='/data/dog_images/train')
  test_data = datasets.ImageFolder(root='/data/dog_images/valid')
  valid_data = datasets.ImageFolder(root='/data/dog_images/test')
```

```
: print(len(train_data))
  print(len(test_data))
  print(len(valid_data))
```

```
6680
835
836
```

Each folder contains images from 133 different dog breeds, which will become our different classes. So we already know each label seems to be represented in each of dataset splits.

```
In [29]: train_data.classes
```

```
'114.Otterhound',
'115.Papillon',
'116.Parson_russell_terrier',
'117.Pekingese',
'118.Pembroke_welsh_corgi',
'119.Petit_basset_griffon_vendeen',
'120.Pharaoh_hound',
'121.Plott',
'122.Pointer',
'123.Pomeranian',
'124.Poodle',
'125.Portuguese_water_dog',
'126.Saint_bernard',
'127.Silky_terrier',
'128.Smooth_fox_terrier',
'129.Tibetan_mastiff',
'130.Welsh_springer_spaniel',
'131.Wirehaired_pointing_griffon',
'132.Xoloitzcuintli',
'133.Yorkshire_terrier']
```

However, we do not know if in the training data, each of the 133 classes is *equally* represented. This is important because our CNN will be learning from whatever images are provided. If only two of the 133 classes have many images, but the rest have only a few, the learning and feature extraction of the CNN will also be skewed towards these two classes. This will drastically affect the performance of our model, especially when test data has images from all classes that need to be identified correctly. Our training accuracy may be very high even if the model just predicts the 1-2 majority classes. This would be misleading.

We can check class counts for each data partition as below, however a visual representation makes it easier to inspect the extent of balance/imbalance. Hence we use bar plots for this purpose in the next section.

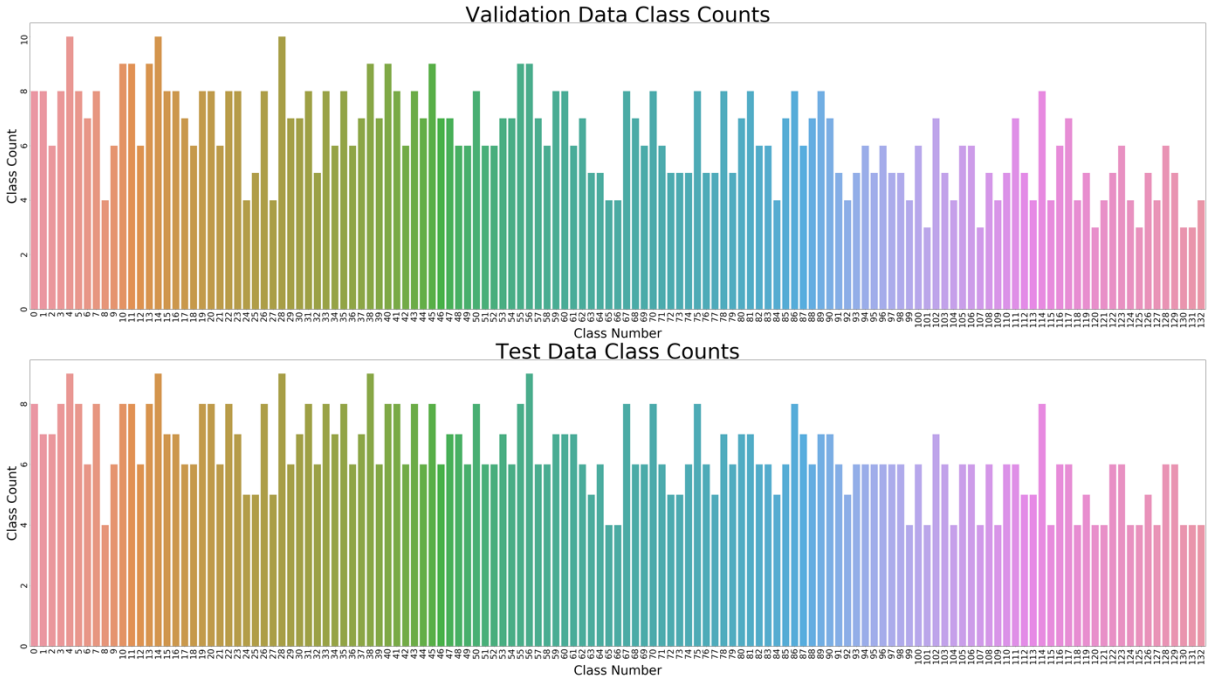
```
In [7]: class_counts

Out[7]: {0: 64,
         1: 58,
         2: 52,
         3: 63,
         4: 77,
         5: 64,
         6: 50,
         7: 66,
         8: 34,
         9: 50,
         10: 66,
         11: 66,
         12: 46,
         13: 69,
         14: 73,
         15: 59,
         16: 62,
         17: 50,
         18: 48,
         19: 62,
         20: 64
```

Exploratory Visualization

As checked below, there is a slight imbalance in our input dataset of dog images. However, since the template notebook [1] uses accuracy, we will continue to use this as our evaluation metric.

The below plots show the number of images for each of the 133 classes – labelled from 0 to 132. The x-axes show the class label, and the y-axes show the class counts for each of train, validation and test data sets.



Another insight that these visualizations provide us is that there is a similar distribution across train, validation and test sets – i.e. if a particular class is represented less in training and validation data, it is represented only to the same proportion in test data as well. This gives us confidence that our test performance ideally shouldn't be affected much by the slight imbalances. Though it is important to note in a 'real world' scenario, such as deployment on a dog breed identification app, it's possible our algorithm may be used a lot more to predict class # 131, which has one of the lowest counts. Based on our discussion from the previous section, for a more long-term perspective F1 Score is a better choice.

Algorithms and Techniques

Below we will look at the techniques applied to the different components of this project: the human face detector, the dog face detector, and the CNN model used to predict the dog breed.

OpenCV Image Recognition

OpenCV is an open-source computer vision and machine learning library [8] that provides optimized algorithms in the said domains, including for our image recognition problem. Specifically, there are pretrained models that can be fed an image, and detect whether or not it shows a human face. For our project we will use one such model from [9] which is a "stump based 20x20 gentle adaboost frontal face detector".

Since it is already pretrained, we don't need to train it again with our human images dataset. We can just load it using the OpenCV Python functions [10]. This model uses something called Haar-based cascade classification. This means we use "Haar features" – something like CNN kernels of certain sizes to calculate feature values from all the training images (human faces and non-human faces). The kernels applied to each subsection of the images or 'windows' result in some feature calculations and values.

Out of the resultant features (could be hundreds or thousands) that are calculated using the Haar kernels, the best that can be used to detect a human face are selected using Adaboost [10] – an ensemble of weak classifiers that make up a strong classifier.

The calculated features are applied to each training image window in multiple stages – hence the name 'cascade classifier'. If a window of an image is found to be non-face region in the first stage itself, it is discarded so that processing time to apply each feature to that region isn't wasted. Image windows that pass all stages of features are face regions. And in this cascaded training process, the features that best determine human faces are narrowed down.

OpenCV documentation on this model [10] clearly defines the Python functions needed to transform the input images and then apply the loaded face detector on it.

CNN & Deep Learning Framework – PyTorch

As discussed in the introductory section, CNN is a machine learning model used for image recognition tasks. More specifically it is a deep learning algorithm – multi-layered artificial neural networks learn from large amounts of data – working like the human brain [11].

We need a specialized software, or library, that can help us quickly build deep learning models with supporting functions and techniques (like the ones in the next 2 sub-sections – transfer learning, data augmentation). If you have hardware like GPUs that can speed up deep learning, these frameworks can help you leverage that hardware. PyTorch by Facebook is one such framework [12]. In this project we use PyTorch for transfer learning, data augmentation, and also to build our own CNN from scratch which we will see in more detail in the *Methodology* section.

Transfer Learning

For the dog detection piece of the solution, we apply a pre-trained CNN model called VGG-16. This is considered transfer learning because we are saving training time by leveraging the knowledge gained by the model already trained on a vast dataset called ImageNet which contains 1.2 million labelled images [13]. It has already learned features to correctly classify several classes of dogs. And as we know, deep learning typically requires large amounts of data – transfer learning is often a way to overcome this.

When the target problem is slightly different from what the pre-trained model has learnt, the modifications required (adjusting the learned features) can be done with minimal addition to training time – this process is called fine-tuning [13]. For CNN, this can be as simple as just adjusting the final fully-connected layers, with weights/parameters of all the other layers fixed. PyTorch provides an interface to pre-trained models through torchvision.models [14].

Data Augmentation

As noted before, deep learning requires a large amount of training data. So a technique that can be leveraged to expand our dataset is data augmentation. This is done by applying various transformations to the existing training images such as horizontal or vertical flips, rotation by a certain number of degrees, blurring or distorting the images, randomly changing the contrast, brightness or saturation of an image, or any combination of these. These and more transforms can be done using PyTorch's torchvision.transforms [15]. Data augmentation's merits for CNN training are well-documented in research work such as [16].

Benchmark

One benchmark or 'baseline' will be the validation and test accuracies for a 'from scratch' CNN – i.e. a very simple and naïve model without much tuning – but a minimum accuracy of **10%** is expected. We will use 3 convolutional layers and appropriate pooling and dropout layers. This model is built in Step 3 of the template notebook using PyTorch and explained in more detail in the *Implementation* section.

Deep learning models are susceptible to several pitfalls – a major one is that they require a large amount of training data to realize their potential. This is where leveraging pre-trained models, or transfer learning, comes in. We will see more about this in *Implementation.*

For a transfer learning model, we can also use past research work as a benchmark. For example, in the work done in [17], they have compared performance of models such as DenseNet-121, DenseNet-169, ResNet-50 and GoogleNet. We can use the below table from [17], which summarizes the test accuracy for each model. An accuracy of around **80 to 85%** would be good to aim for.

TABLE I: COMPARISON OF STATE OF THE ART MODELS WITH OUR MODIFIED MODELS, DA AND FT IS THE ABBREVIATION OF THE DATA AUGMENTATION AND FINE-TUNING RESPECTIVELY.

Model Name	Test Accuracy
DenseNet-121	74.28%
DenseNet-169	76.23%
GoogleNet	72.11%
ResNet-50	73.28%
DenseNet-121+FT+DA	84.01%
DenseNet-169+FT+DA	85.37%
ResNet-50+FT+DA	**89.66%**
GoogleNet+FT+DA	82.08%

III. Methodology

Data Pre-processing

Human Detector

OpenCV's cascade classifier requires that the input images are in grayscale – this can be done by using the cvtColor function with COLOR_BGR2GRAY argument: this means convert BGR (colour image) to grayscale. No other pre-processing is required for this step.

Dog Detector

For this step, since we use a pre-trained model from PyTorch (VGG16), there are certain requirements of the input image that we obtain using PyTorch's torchvision.transformer:

- Image Size – 224x224; minimum expected for PyTorch pre-trained models
- Must be converted to an image tensor – a numpy array representation of the image
- Normalize values – to standardize all tensor image values; the mean and std deviation sequences below below are expected for pre-trained models. There are 3 values in each sequence for the 3 channels of input (RGB image)

```
transformer = 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])])
```

Dog Breed Classifier – From Scratch and Transfer Learning

In this step, the test and validation datasets are pre-processed using the same transformer above. However, for the training data set, we will apply some extra data augmentation steps:

- Use a random resized crop instead of centre crop that maintains the aspect ratio: size is still the same
- Use a random horizontal flip
- Use a random rotation between 10 to 20 degrees

```
transformer = transforms.Compose(
    [transforms.Resize(256), transforms.RandomResizedCrop(224, ratio=(1,1)),
     transforms.RandomHorizontalFlip(),
     transforms.RandomRotation((10,20)), # degrees to rotate by: min, max fo
     transforms.ToTensor(),
     transforms.Normalize(mean=[0.485, 0.456, 0.406],
                          std=[0.229, 0.224, 0.225])])
```

We used RandomHorizontalFlip as there are bound to be images of dogs facing in either direction. We also used RandomRotate with a rotation range, again to simulate additional images of dogs that may not be in the exact same orientation, while still keeping the scenarios realistic.

The reason we chose 224x224 for the input tensor is so that we can use the same data loaders for Step 3 and Step 4's models (see *Implementation*). Minimum image size for most of the pre-trained models is 224x224. Also, consistency matters: we can also more easily compare performance by doing this.

Implementation

The following steps 0 to 5, provided in the template notebook [2], were completed towards the project workflow.

0. *Import Datasets*

File names (image paths) of each image in both datasets (human and dog) were stored in NumPy arrays. The images were loaded when needed using either OpenCV's 'imread' function or PIL's Image.open function, using the image path.

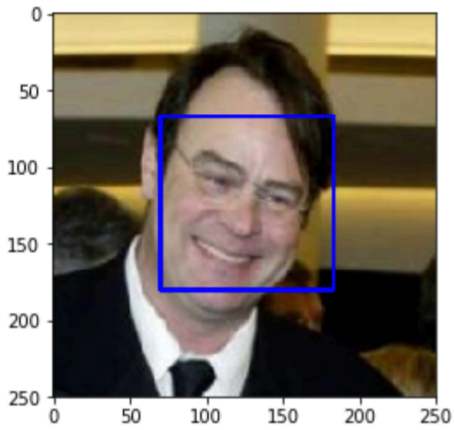
1. Detect Humans

Extracted OpenCV's Haar-feature based cascade classifiers (pre-trained), read images using OpenCV with previously created array of image paths, and used the classifier (detectMultiScale function) to detect whether the image has a human face or not.

```
# 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
```

Sample output shown below:

```
Number of faces detected: 1
```



Evaluation of the detector will be covered in *Model Evaluation and Validation.*

2. Detect Dogs

In this step we loaded a pre-trained model from PyTorch – VGG16 – and used it to predict the class of an image, given its file path.

VGG-16 dog breed predictions are all of a continuous sequence from 151 to 268 inclusive. So we checked whether the predicted index is a 'dog' index in VGG16's dictionary, and return a 'true' if so. Evaluation of the detector will be covered in *Model Evaluation and Validation.*

3. Create a CNN to Classify Dog Breeds - from Scratch

In this step, we completed the first 'from scratch' model described in *Benchmark* section. For the architecture we followed a simple model with 3 convolutional layers with kernel size 3. Smaller kernel sizes are preferred over larger kernel sizes, and odd over even [18]. 3x3 convolutional filter is a popular choice.

For a starting point, we referred to the following PyTorch documentation on how to define a CNN [19] and a Medium tutorial showing the same with more layers [20].

With each layer, the number of output channels increases, but the image output size decreases (if stride of the Conv2D layer set to 2).

Next, we added operations for each convolutional layers as follows.

Each Conv2D layer has:

- Batch Normalization: to prevent the optimizer from oscillating in a plateau, each batch is normalized so all features are on the same scale
- ReLU: Commonly used activation function for CNNs
- Max Pooling: For down-sampling, a best practice after non-linearity is applied. This is found to be better than average pooling for tasks like image recognition. If stride is 2 for this layer, output size is further scaled down by 2.

We further defined 2 dropout layers, applied to each fully connected layer, to prevent overfitting, with a small dropout probability of 0.2 (20%). These layers are needed to transition from feature maps, to the output prediction.

We use two fully connected layers, so that the first FC layer can abstract out key features and reduce size (6272 --> 512) before the final FC layer predicts the output class. Output size of 512 for FC1 is commonly used in CNN architectures.

Adam optimizer was chosen, as it is known to be better than SGD. Cross Entropy Loss was the chosen loss function, which is appropriate for multi-class classification.

We also reduced the number of epochs from 100 (template) to 30, to reduce training time and as the validation loss was already low. Test accuracy (below) also was above the threshold required for this task: 10% was expected, accuracy produced is 22%:

```
# call test function
test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 3.423710


Test Accuracy: 22% (189/835)
```

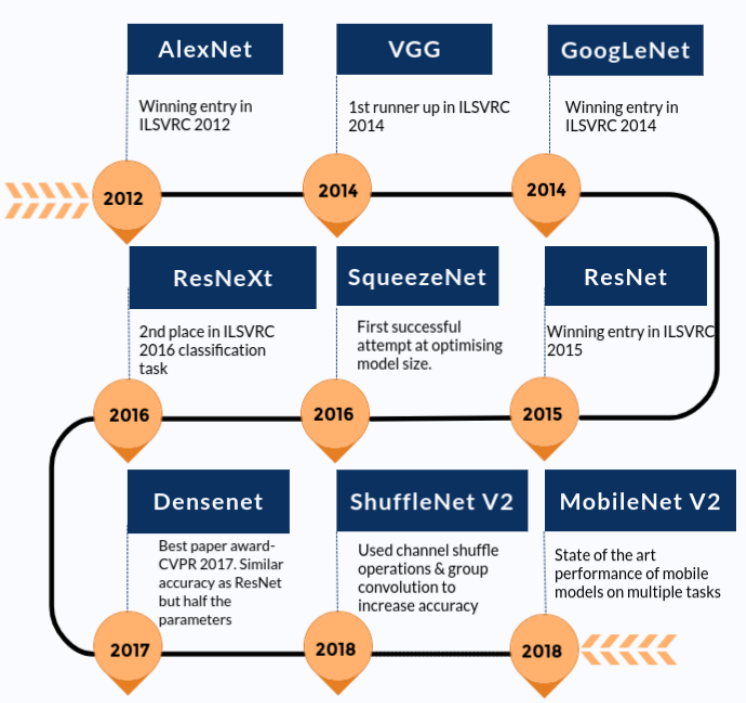
This forms the test accuracy to beat for our subsequent models – this one is the first benchmark or rather, baseline. We are looking to get an accuracy similar to our second benchmark [17], which uses transfer learning. So we tried this next.

4. Create a CNN to Classify Dog Breeds - Transfer Learning

PyTorch has several pre-trained models that can be used and compared for this problem [14] including VGG, AlexNet, ResNet, GoogLeNet, and Inception v3. Again loss function and optimizer as specified as CrossEntropyLoss and Adam respectively.

For Adam optimizer, learning rate of 0.001 was chosen, and since we are fine-tuning, only the final classifier/output layers were optimized. Other parameters were frozen. A test accuracy of at least 60% is expected.

We referred to the below image from [21] to shortlist which pre-trained models to apply and compare:



We eliminated AlexNet, VGG and GoogleNet in favour of the newer and more recent models which are built on top of improvements from these. The choice was also limited by the

constraint of PyTorch used in the Udacity workspace provided, which is 0.4.0. Based on this, we finalized the following:

- SqueezeNet – a more lightweight version of AlexNet – "AlexNet level accuracy with 50x fewer parameters" [22]
- ResNet-101: Variation of a winning entry in ILSVRC (ImageNet Large Scale Visual Recognition Challenge)
- DenseNet-121: Best Paper Award – Conference of Computer Vision and Pattern Recognition 2017; similar accuracy to ResNet but half the parameters

Fine-tuning: For all layers besides the final classification/fully connected layers, the parameters were frozen. Each transfer learning model used required slightly different fine-tuning. For example, DenseNet-121 below, the classifier is modified to a linear layer with output of size 133 – since we have 133 classes to predict.

```
num_ftrs = model_transfer.classifier.in_features
model_transfer.classifier = nn.Linear(num_ftrs, 133)
```

Since we are using the same data loaders that we used for the from-scratch CNN, the same data augmentation and transformations will apply here too. Batch size was chosen as 32. We started with SqueezeNet (10 epochs) which had an accuracy of 65% - still short of our benchmarks in [17]. We will see the improvements made in the next section.

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

```
Test Loss: 1.391858


Test Accuracy: 65% (543/835)
```

```
Epoch: 1 took 123.402 seconds.  Training Loss: 4.030740         Validation Loss: 2.592509
Decrease in validation loss. Saving new model.
Epoch: 2 took 123.277 seconds.  Training Loss: 2.569622         Validation Loss: 2.030397
Decrease in validation loss. Saving new model.
Epoch: 3 took 122.959 seconds.  Training Loss: 2.092322         Validation Loss: 1.909686
Decrease in validation loss. Saving new model.
Epoch: 4 took 122.526 seconds.  Training Loss: 1.882073         Validation Loss: 1.825580
Decrease in validation loss. Saving new model.
Epoch: 5 took 122.435 seconds.  Training Loss: 1.753138         Validation Loss: 1.561332
Decrease in validation loss. Saving new model.
Epoch: 6 took 121.874 seconds.  Training Loss: 1.659257         Validation Loss: 1.535264
Decrease in validation loss. Saving new model.
Epoch: 7 took 122.310 seconds.  Training Loss: 1.570020         Validation Loss: 1.507682
Decrease in validation loss. Saving new model.
Epoch: 8 took 122.558 seconds.  Training Loss: 1.514521         Validation Loss: 1.677062
Epoch: 9 took 122.054 seconds.  Training Loss: 1.527164         Validation Loss: 1.500449
Decrease in validation loss. Saving new model.
Epoch: 10 took 121.439 seconds.         Training Loss: 1.471973         Validation Loss: 1.652334
```

5. Write your Algorithm

Once the model was chosen (see *Refinement* for final model details), we wrote a simple rule-based function leveraging the dog detector and human detectors constructed in steps 1 and 2, as well as predictions from step 4's CNN dog breed classifier. This is our 'solution pipeline' to do the following [2]:

- 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 output that indicates an error.

Refinement

After SqueezeNet, we ran Resnet101 which had an accuracy of 79%, in 10 epochs as well.

Resnet101

is greater than 60%.

```
In [146]: test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
          Test Loss: 0.697021


          Test Accuracy: 79% (663/835)
```

```
Epoch: 1 took 206.002 seconds.  Training Loss: 2.959849         Validation Loss: 1.113686
Decrease in validation loss. Saving new model.
Epoch: 2 took 203.854 seconds.  Training Loss: 1.536510         Validation Loss: 0.966883
Decrease in validation loss. Saving new model.
Epoch: 3 took 204.704 seconds.  Training Loss: 1.335289         Validation Loss: 0.808964
Decrease in validation loss. Saving new model.
Epoch: 4 took 204.101 seconds.  Training Loss: 1.237176         Validation Loss: 0.749130
Decrease in validation loss. Saving new model.
Epoch: 5 took 203.787 seconds.  Training Loss: 1.174434         Validation Loss: 0.926364
Epoch: 6 took 203.769 seconds.  Training Loss: 1.160488         Validation Loss: 0.701556
Decrease in validation loss. Saving new model.
Epoch: 7 took 202.923 seconds.  Training Loss: 1.152797         Validation Loss: 0.795050
Epoch: 8 took 203.271 seconds.  Training Loss: 1.097594         Validation Loss: 0.904159
Epoch: 9 took 203.271 seconds.  Training Loss: 1.048416         Validation Loss: 0.782814
Epoch: 10 took 202.571 seconds.         Training Loss: 1.086542         Validation Loss: 0.734915
```

Next, we chose DenseNet which is more recent than either of the previous models (2017). It is supposed to provide Resnet-level accuracy for half the parameters. Indeed we see that the accuracy is about 80% for 10 epochs. And the per-epoch time taken also seems to be less than Resnet.

DenseNet-121

```
In [26]: test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
         Test Loss: 0.607499


         Test Accuracy: 80% (670/835)
```

```
Epoch: 1 took 154.280 seconds.  Training Loss: 1.109051         Validation Loss: 0.747743
Decrease in validation loss. Saving new model.
Epoch: 2 took 153.890 seconds.  Training Loss: 1.087954         Validation Loss: 0.705361
Decrease in validation loss. Saving new model.
Epoch: 3 took 153.979 seconds.  Training Loss: 1.055388         Validation Loss: 0.750278
Epoch: 4 took 155.493 seconds.  Training Loss: 1.023641         Validation Loss: 0.714662
Epoch: 5 took 155.591 seconds.  Training Loss: 1.010298         Validation Loss: 0.666508
Decrease in validation loss. Saving new model.
Epoch: 6 took 150.850 seconds.  Training Loss: 1.012698         Validation Loss: 0.686527
Epoch: 7 took 149.416 seconds.  Training Loss: 0.977068         Validation Loss: 0.652553
Decrease in validation loss. Saving new model.
Epoch: 8 took 150.706 seconds.  Training Loss: 0.957790         Validation Loss: 0.688061
Epoch: 9 took 150.509 seconds.  Training Loss: 0.948236         Validation Loss: 0.657680
Epoch: 10 took 150.324 seconds.         Training Loss: 0.927668         Validation Loss: 0.642559
Decrease in validation loss. Saving new model.
```

Below is a table summarizing the refinements covered in *Implementation:*

We see that DenseNet per-epoch time is about 25% lower than Resnet – which makes sense since DenseNet is a lower parameter version. SqueezeNet, which is just a more lightweight version of AlexNet has the lowest accuracy, which makes sense, since AlexNet is the oldest. Several improvements were proposed after that to the likes of ResNet and DenseNet.

Model	Epochs	Per-Epoch Avg. Time (seconds)	Test Accuracy (%)
SqueezeNet with DA	10	122.48	65
Resnet-101 with DA	10	203.82	79
DenseNet-121 with DA	10	152.34	80

IV. Results

Model Evaluation and Validation

Human Face Detector

We evaluated this on a small sample of first 100 images in the human face and dog images datasets. We ran these through the human face detector. The expectation is that more of the human images should be correctly identified, and very few (or none) of dog images should be called out as human faces.

98 of the human images are correctly detected as human, 17 of the dog images are incorrectly 'detected' as human. This is an acceptable result.

Percentage of human images with detected face = 98%
Percentage of dog images with detected face = 17%

Dog Detector

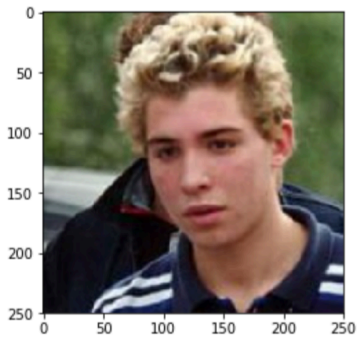
A similar process is repeated for our dog detector built with VGG-16 and transfer learning:

Detected dog percentage in human files = 0%
Detected dog percentage in dog files = 100%

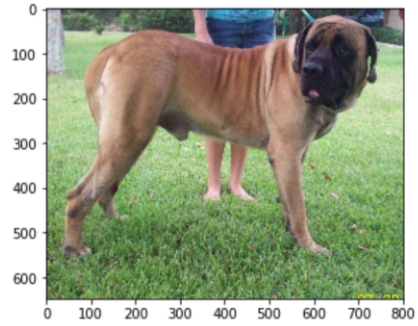
This too is an acceptable result.

Dog Breed Classifier with CNN

Our final model was built upon DenseNet-121 trained to 30 epochs. It showed a test accuracy of 81% and also performed well on other unseen data:



You look like a: American water spaniel



This dog is a: Mastiff

Note from the above human-detected image: the model has correctly identified in the human the wavy/curled fur characteristic of American water spaniels.

On the sensitivity of the model: the test accuracy remained about the same with 10 epochs and 30 epochs of training, indicating that the model is not prone to overfitting. This is a good indicator that we have a robust model. Note below how the training loss remains stable after 11 epochs.

DenseNet121 – 30 epochs

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

```
Test Loss: 0.694098


Test Accuracy: 81% (677/835)
```

```
Epoch: 1 took 154.131 seconds.  Training Loss: 0.970520         Validation Loss: 0.671456
Decrease in validation loss. Saving new model.
Epoch: 2 took 153.605 seconds.  Training Loss: 0.929674         Validation Loss: 0.667275
Decrease in validation loss. Saving new model.
Epoch: 3 took 153.513 seconds.  Training Loss: 0.906856         Validation Loss: 0.707873
Epoch: 4 took 154.551 seconds.  Training Loss: 0.916665         Validation Loss: 0.658589
Decrease in validation loss. Saving new model.
Epoch: 5 took 154.418 seconds.  Training Loss: 0.891734         Validation Loss: 0.675161
Epoch: 6 took 153.854 seconds.  Training Loss: 0.872875         Validation Loss: 0.677334
Epoch: 7 took 152.225 seconds.  Training Loss: 0.884683         Validation Loss: 0.676404
Epoch: 8 took 155.368 seconds.  Training Loss: 0.868458         Validation Loss: 0.658091
Decrease in validation loss. Saving new model.
Epoch: 9 took 155.795 seconds.  Training Loss: 0.871647         Validation Loss: 0.642369
Decrease in validation loss. Saving new model.
Epoch: 10 took 155.753 seconds.         Training Loss: 0.871571         Validation Loss: 0.690371
Epoch: 11 took 155.057 seconds.         Training Loss: 0.842851         Validation Loss: 0.626722
Decrease in validation loss. Saving new model.
Epoch: 12 took 155.762 seconds.         Training Loss: 0.870416         Validation Loss: 0.630427
Epoch: 13 took 155.795 seconds.         Training Loss: 0.830792         Validation Loss: 0.642734
Epoch: 14 took 153.584 seconds.         Training Loss: 0.830920         Validation Loss: 0.661983
Epoch: 15 took 155.400 seconds.         Training Loss: 0.823737         Validation Loss: 0.634466
Epoch: 16 took 155.870 seconds.         Training Loss: 0.836884         Validation Loss: 0.657519
Epoch: 17 took 155.512 seconds.         Training Loss: 0.809466         Validation Loss: 0.648136
Epoch: 18 took 154.120 seconds.         Training Loss: 0.812964         Validation Loss: 0.670721
Epoch: 19 took 151.447 seconds.         Training Loss: 0.820433         Validation Loss: 0.693831
Epoch: 20 took 150.370 seconds.         Training Loss: 0.812768         Validation Loss: 0.700712
Epoch: 21 took 150.689 seconds.         Training Loss: 0.814899         Validation Loss: 0.715333
Epoch: 22 took 150.927 seconds.         Training Loss: 0.865168         Validation Loss: 0.695509
Epoch: 23 took 149.714 seconds.         Training Loss: 0.823243         Validation Loss: 0.669434
Epoch: 24 took 148.745 seconds.         Training Loss: 0.792847         Validation Loss: 0.643744
Epoch: 25 took 151.182 seconds.         Training Loss: 0.827808         Validation Loss: 0.694788
Epoch: 26 took 153.597 seconds.         Training Loss: 0.796783         Validation Loss: 0.640350
Epoch: 27 took 151.809 seconds.         Training Loss: 0.768825         Validation Loss: 0.651599
Epoch: 28 took 153.203 seconds.         Training Loss: 0.805455         Validation Loss: 0.650374
Epoch: 29 took 152.521 seconds.         Training Loss: 0.807868         Validation Loss: 0.684826
Epoch: 30 took 150.563 seconds.         Training Loss: 0.821109         Validation Loss: 0.645741
```

Justification

Our final model is DenseNet-121 with data augmentation and fine-tuning for the current dataset, trained to 30 epochs. The test accuracy is 81%. This adequately crosses the threshold of our first benchmark model – CNN from scratch (22%) and the expected level for the Udacity project (60%).

Recalling our DenseNet-121 accuracy from [17], we find that we are a few percentage points short – their accuracy was 84% for the same model. The differences could be due to the variations in data augmentation applied by the team, and the dataset itself – they tested also on a subset of ImageNet dataset. A key difference in augmentation in fact, is that [17] also applied transformations to produce noisy images. Number of epochs is the same (30).

However, our present model still adequately serves the purpose as seen on the sample images above, identifying a dog as a Mastiff correctly.

V. Conclusion

Reflections & Improvements

We have so far built a human detector using a pre-trained OpenCV classifier, a dog detector using a pre-trained VGG16 (PyTorch) and a dog breed classifier using a pre-trained and fine-tuned DenseNet-121. We have also put these together as a successful algorithm that shows the closest resembling dog breed for human and dogs, and shows error if any image besides these is the input.

We have also built a CNN from scratch using PyTorch functions. This was the most challenging part of the project as to get this right, the output size at each layer must be calculated based on the kernel size, stride, padding. This is then needed to define the input size at subsequent layers. There was also some research needed on what are best practices for these architectural parameters, and even for values like the batch size and output channel size for convolutional layers. We also needed to know how and when to use batch normalization, max pooling and dropout.

But what was interesting was that despite this effort, it was not a very accurate classifier (22% accuracy), as it is still considered a shallow classifier compared to the likes of standard architectures like ResNet or even VGG-16. And we did not do any hyperparameter tuning like finding the most optimized value for the dropout rate, or activation function. This is one potential improvement.

For both the from-scratch CNN and fine-tuned models like DenseNet-121 we can also employ more types of data augmentation like colour jitter to introduce different levels of brightness & saturation, or introduce random noise to images by using a Gaussian filter and blurring them.

And though we used a single validation set, it would also be good practice to introduce k-fold cross-validation so that we have a better idea of 'unseen' accuracy on multiple validation sets.

References

1. http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf
2. https://github.com/udacity/deep-learning-v2-pytorch/tree/master/project-dog-classification
3. https://www.kaggle.com/c/dog-breed-identification/overview
4. https://link.springer.com/article/10.1007/s41095-020-0184-6
5. https://link.springer.com/chapter/10.1007/978-3-319-16841-8_52
6. http://vision.stanford.edu/aditya86/ImageNetDogs/
7. https://towardsdatascience.com/the-5-classification-evaluation-metrics-you-must-know-aa97784ff226
8. https://opencv.org/about/
9. https://github.com/opencv/opencv/blob/master/data/haarcascades/haarcascade_frontalface_alt.xml
10. https://docs.opencv.org/master/db/d28/tutorial_cascade_classifier.html
11. https://www.ibm.com/cloud/learn/deep-learning
12. https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html
13. https://arxiv.org/pdf/1608.08614.pdf
14. https://pytorch.org/docs/stable/torchvision/models.html
15. https://pytorch.org/docs/stable/torchvision/transforms.html
16. https://arxiv.org/abs/1906.11052
17. https://www.researchgate.net/publication/325384896_Modified_Deep_Neural_Networks_for_Dog_Breeds_Identification
18. https://towardsdatascience.com/deciding-optimal-filter-size-for-cnns-d6f7b56f9363
19. https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html#define-a-convolutional-neural-network
20. Medium Article

21. https://learnopencv.com/pytorch-for-beginners-image-classification-using-pre-trained-models/
22. https://arxiv.org/abs/1602.07360