Machine Learning Engineer Nanodegree

Capstone Project - Dog Breed Classifier using CNN

Vedavyas Kamath
April 22nd 2020

I. Definition

Project Overview

This project belongs to the domain of Image classification which refers to a process in computer vision that can classify an image according to its visual content. For example, an image classification algorithm may be designed to tell if an image contains a human figure or not. While detecting an object is trivial for humans, robust image classification is still a challenge in computer vision applications, which actually in fact has great scope in automation & implementing more robust feedback based Control Systems.

The main motivation for me behind choosing this project is mainly because of the fact that I find image classification very interesting and it has many practical applications that can be used to solve real life problems like crime, driving cars and finding tumors.This project will help me explore the field of image classification, while giving me a solid idea and foundation of building an image classifier and preparing me to solve more complex real life problems.

Problem Statement

The problem statement here is to analyse any input image and check for the prescence of a dog in it, and when found return the breed name of the dog which resembles the one found in the image. This is an image classification problem (supervised learning) in which a model is built and then trained using images along with its labels. The model is made to learn the different images and remember their labels (dog breeds in our case), so that whenever a new image is passed, it is able to predict the breed for this new image based on what it has learned while training. So having said this the model will take any image as the input and return the breed of the dog as output.

This problem has a practical potential solution and is measurable too, by all means. By tweaking the model a little and training it with an appropriate image dataset, the solution designed for this problem could also be used for identifying different fruits, cats, animals, vehicles etc.

Evaluation Metrics

The models can be evaluated using an accuracy metric utilizing a test set. The human and dog detectors could be tested on 100 images of each to check the accuracy of the models. Further as stated in the dataset & inputs section, I would split the total image dataset into training, validation and testing sets to be able to train, validate and finally test my models. Usually depending on the dataset, accuracy may not be an adequate measure for a classification model. Like in a medical diagnostic model for example, if the occurrence of a positive result is 1%, then a model that predicts negative 100% of the time is 99% accurate but also a completely worthless model. Other metrics, such as precision, recall or F1 would be more useful for that type of problem.

However, after checking the dog breed dataset, the classes (breeds) seem to be relatively balanced, and hence a simple accuracy score should be sufficient. Simply the accuracy in this case can be viewed as:

Accuracy = $\frac{C}{T} \times 100$

where,
C -> Number of correctly classified images
T -> Total number of images to be classified

II. Analysis

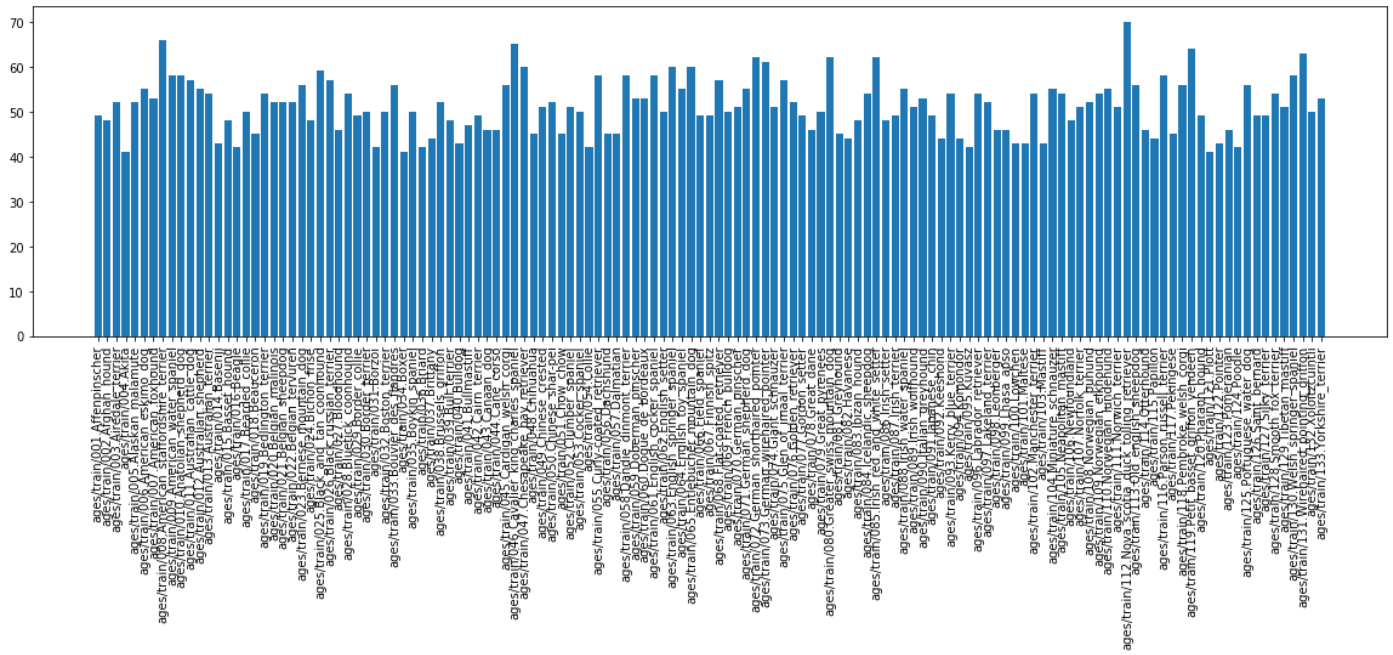
Data Exploration

As this project is to create a Dog Breed Image Classifier, a large set of 8351 dog images and 13233 human images was provided by Udacity. From the data folders, could see there are a total of 133 different dog breeds in the training data set. So as a first step I will split these images into training, validation and test sets to be used through out the project accordingly to first train the model, then validate and eventually test the performance of my model.

The images included in the dataset span various sizes and orientations of which a few samples are as below:

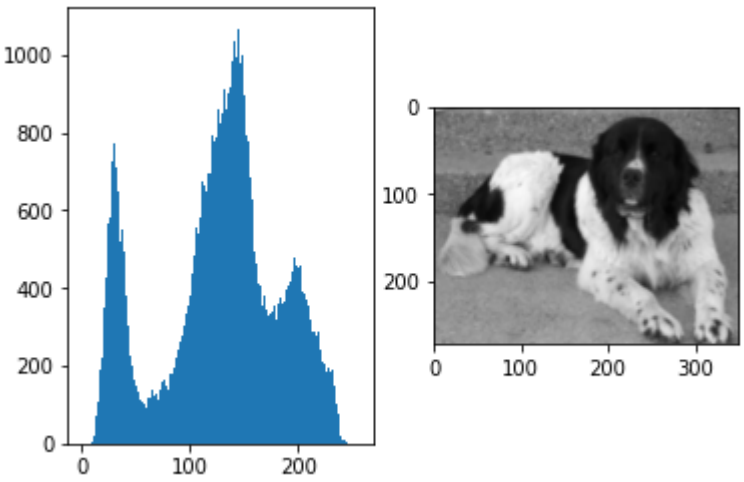
Exploratory Visualization

To understand the data better and gather some descriptive facts/statistics, I have plotted a bar chart which shows the number of images included in the data for every distinct breed. Checked and could see on an average there are 50 images per breed which can be seen in the chart below:

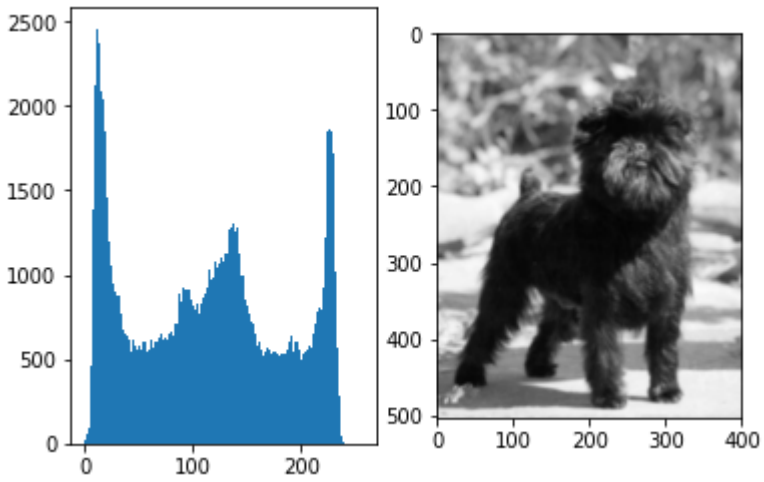


We can get a deeper understanding of these images through the use of histograms. By looking at the histogram of an image, we get an intuition about contrast, brightness of themage. I have plotted a histogram of the value of pixels of an image alongside also displayed the image for which it was plotted to understand from the histogram trends for various images.

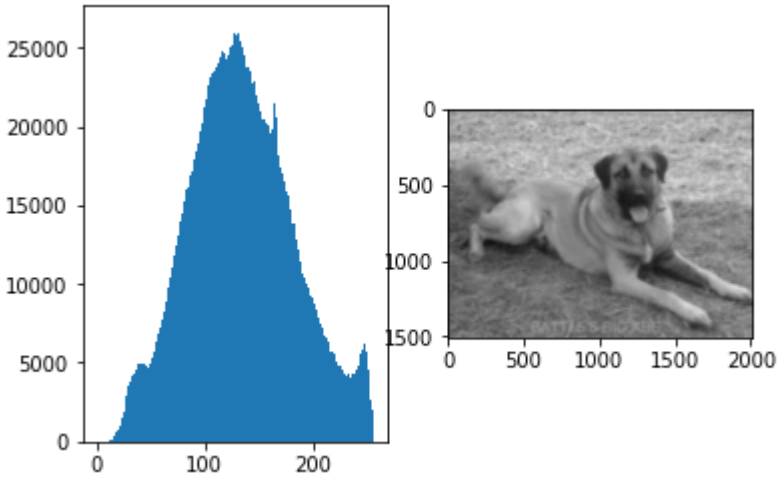
When an image has a balance of light and dark, we get a U-shaped histogram as below, which is the case in the image of the black and white dog below:



An image that has a lot of black will have this type of distribution where the darkest/black pixels are represented by the higher numbers on the x-axis:



An image that is mostly gray with not too much of either black or white will look more like a Normal/Gaussian distribution having a Bell curve as below:



By visualising an image through histogram can give us an idea of how the models decompose, understand and learn images to be able to recognize a new image accordingly.

Algorithms and Techniques

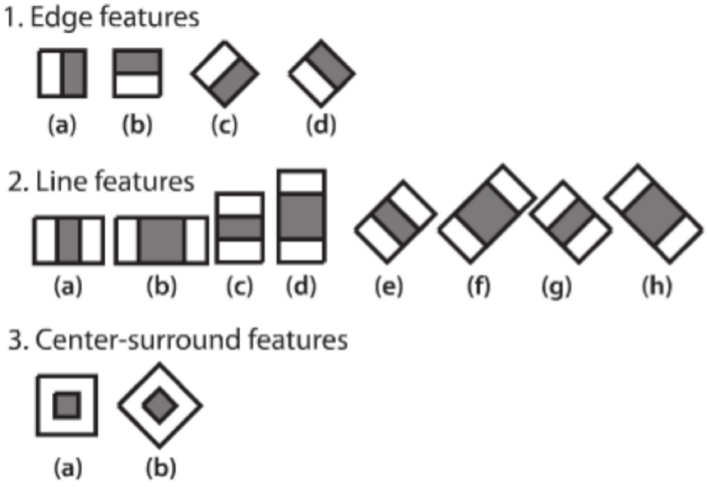
I have developed an algorithm as part of the project that will accept any user supplied image as input but will identify and classify only the dog image. For any image uploaded having a human face, an estimate of the most resembling dog breed will be given and if neither is detected in the image, it provides error output. There are multiple ways of approaching this problem however I have chosen to use convolutional neural network (CNN) to solve this problem and predict the breed for dog images. To acheieve this, I have built individual models for each of the different steps that are designed to handle a single task alone, with high efficiency, which are as below:

Step 1: Human Face Detection
First to detect whether an image has a dog's face or a human's face we need to build a model. I have used the OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

Model Description:
Object Detection using Haar feature-based cascade classifiers is an effective object detection method proposed by Paul Viola and Michael Jones in their paper, "Rapid Object Detection using a Boosted Cascade of Simple

Features" in 2001. It is a machine learning based approach where a cascade function is trained from a lot of positive and negative images which is then used to detect objects in other images.

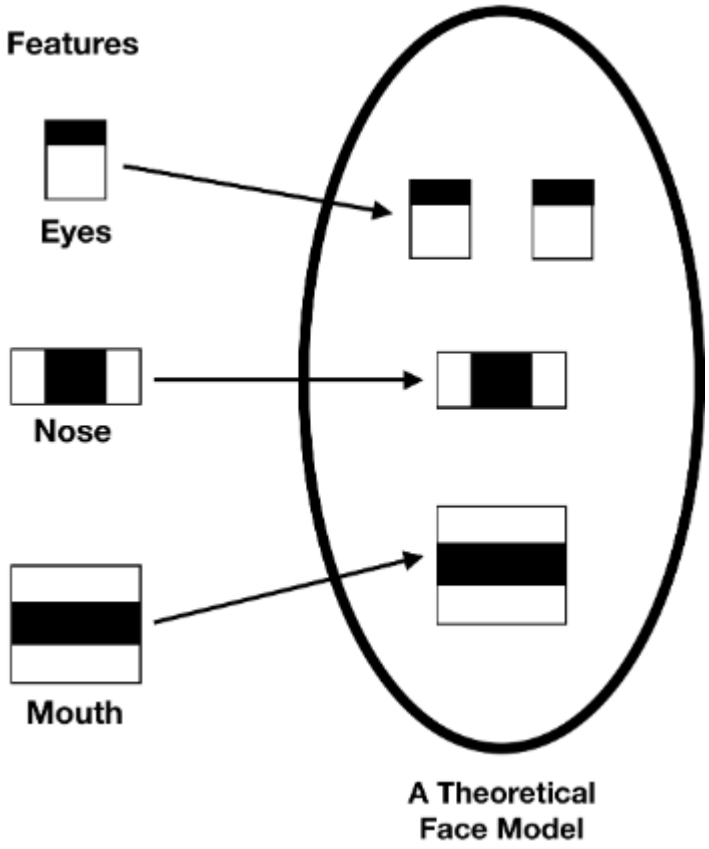
A Haar Cascade is based on “Haar Wavelets” which Wikipedia defines as: A sequence of rescaled “square-shaped” functions which together form a wavelet family or basis. It is based on the Haar Wavelet technique to analyze pixels in the image into squares by function. This uses machine learning techniques to get a high degree of accuracy from what is called “training data”. This uses “integral image” concepts to compute the “features” detected. Haar Cascades use the Adaboost learning algorithm which selects a small number of important features from a large set to give an efficient result of classifiers.



Face Detection determines the locations and sizes of human faces in arbitrary (digital) images.

In **Face Recognition**, the use of Face Detection comes first to determine and isolate a face before it can be recognized.

Haar Cascades use machine learning techniques in which a function is trained from a lot of positive and negative images. This process in the algorithm is used for feature extraction. An illustration of this is below:



The training file used in this project is an XML file called: `haarcascade_frontalface_alt.xml`

Used the detectMultiscale module from OpenCV which simply creates a rectangle with coordinates (x,y,w,h) around the face detected in the image which contains code parameters that are the most important to consider.

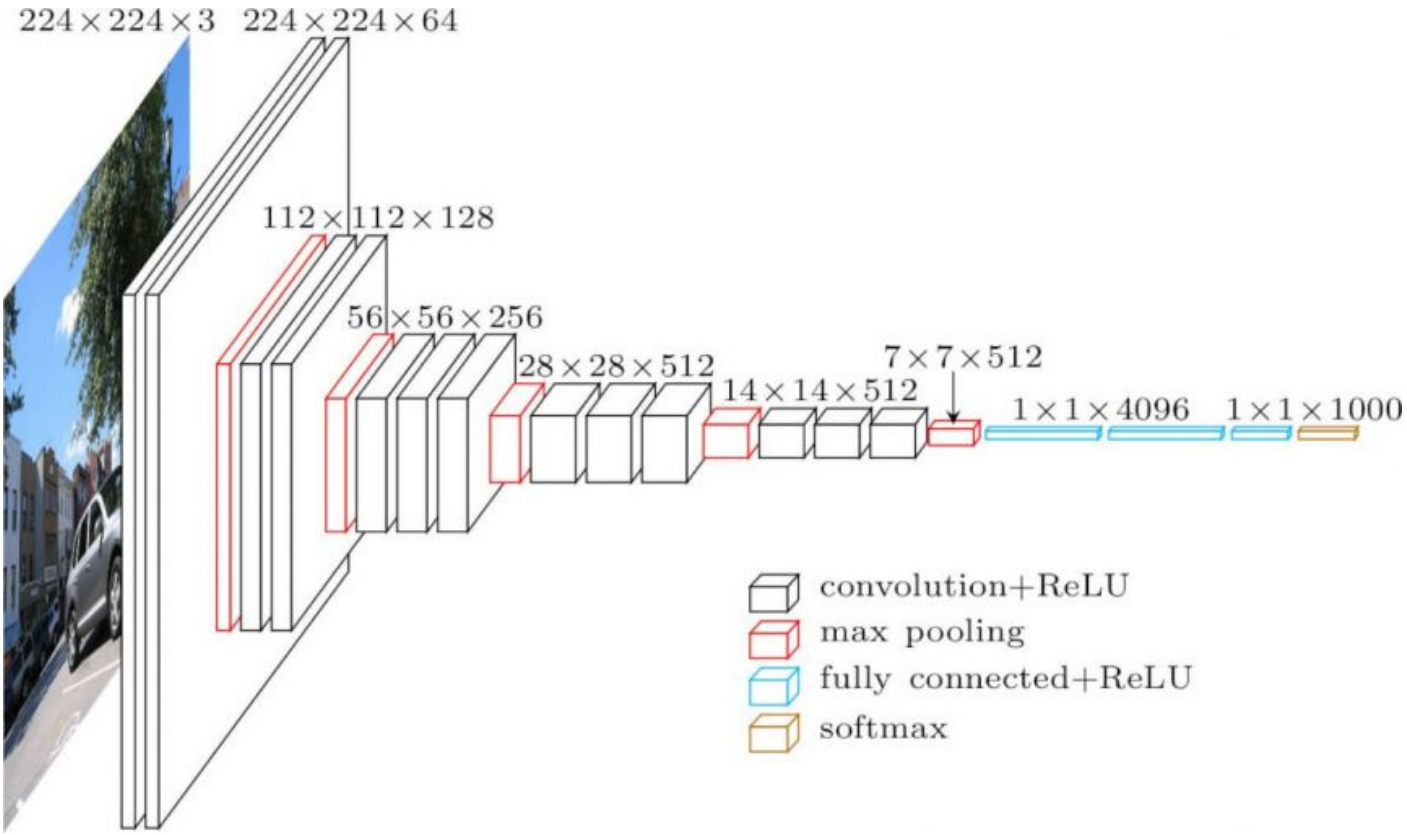
`scaleFactor` : The value indicates how much the image size is reduced at each image scale. A lower value uses a smaller step for downscaling. This allows the algorithm to detect the face. It has a value of x.y, where x and y are arbitrary values you can set.
`minNeighbors` : This parameter specifies how many “neighbors” each candidate rectangle should have. A higher value results in less detections but it detects higher quality in an image. You can use a value of X that specifies a finite number.
`minSize` : The minimum object size. By default it is (30,30). The smaller the face in the image, it is best to adjust the minSize value lower.

When a face is detected, a red rectangle will be generated around the face which can be seen in the implementation section where I have attached an image of the same.

Reason for Choice:
OpenCV's HAAR cascade classifiers haarcascade_frontalface_alt.xml is especially designed for the sake of frontal face recognition by being able to detect important facial features such as eyes, nose, cheeks and the mouth thereby making it a good candidate to be used to recognize human faces.

Step 2: Dog Face Detection
And simillarly for detecting the prescence of dog face, used the the pre-trained VGG16 model which is a very Deep Convolutional Networks for Large-Scale Image Recognition already trained on ImageNet, a very large & popular dataset used for image classification and other vision tasks.

Model Description:
VGG16 is a convolutional neural network model proposed by K. Simonyan and A. Zisserman from the University of Oxford in the paper “Very Deep Convolutional Networks for Large-Scale Image Recognition”. The model achieves 92.7% top-5 test accuracy in ImageNet, which is a dataset of over 14 million images belonging to 1000 classes. It was one of the famous model submitted to ILSVRC-2014. It makes the improvement over AlexNet by replacing large kernel-sized filters (11 and 5 in the first and second convolutional layer, respectively) with multiple 3×3 kernel-sized filters one after another. VGG16 was trained for weeks and was using NVIDIA Titan Black GPU’s. And ImageNet is a dataset of over 15 million labeled high-resolution images belonging to roughly 22,000 categories. The images were collected from the web and labeled by human labelers using Amazon’s Mechanical Turk crowd-sourcing tool. Starting in 2010, as part of the Pascal Visual Object Challenge, an annual competition called the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) has been held. ILSVRC uses a subset of ImageNet with roughly 1000 images in each of 1000 categories. In all, there are roughly 1.2 million training images, 50,000 validation images, and 150,000 testing images. ImageNet consists of variable-resolution images. Therefore, the images have been down-sampled to a fixed resolution of 256×256. Given a rectangular image, the image is rescaled and cropped out the central 256×256 patch from the resulting image. A simple illustration of architecture of VGG16 is below:



The input to cov1 layer is of fixed size 224 x 224 RGB image. The image is passed through a stack of convolutional (conv.) layers, where the filters were used with a very small receptive field: 3×3 (which is the smallest size to capture the notion of left/right, up/down, center). In one of the configurations, it also utilizes 1×1 convolution filters, which can be seen as a linear transformation of the input channels (followed by non-linearity). The convolution stride is fixed to 1 pixel; the spatial padding of conv. layer input is such that the spatial resolution is preserved after convolution, i.e. the padding is 1-pixel for 3×3 conv. layers. Spatial pooling is carried out by five max-pooling layers, which follow some of the conv. layers (not all the conv. layers are followed by max-pooling). Max-pooling is performed over a 2×2 pixel window, with stride 2.

Three Fully-Connected (FC) layers follow a stack of convolutional layers (which has a different depth in different architectures): the first two have 4096 channels each, the third performs 1000-way ILSVRC classification and thus contains 1000 channels (one for each class). The final layer is the soft-max layer. The configuration of the fully connected layers is the same in all networks.

All hidden layers are equipped with the rectification (ReLU) non-linearity. It is also noted that none of the networks (except for one) contain Local Response Normalisation (LRN), such normalization does not improve the performance on the ILSVRC dataset, but leads to increased memory consumption and computation time.

Reason for Choice:
VGG16 significantly outperforms the previous generation of models in the ILSVRC-2012 and ILSVRC-2013 competitions. The VGG16 result is also competing for the classification task winner (GoogLeNet with 6.7% error) and substantially outperforms the ILSVRC-2013 winning submission Clarifai, which achieved 11.2% with external training data and 11.7% without it. Concerning the single-net performance, VGG16 architecture achieves the best result (7.0% test error), outperforming a single GoogLeNet by 0.9% thereby making it a particularly good candidate to solve our problem here of identifying dog faces.

Step 3: Dog Breed Identification
Once we have detected a dog/human face in the image, the next step would be to predict the breed of the dog for which I have tried 2 approaches as below:

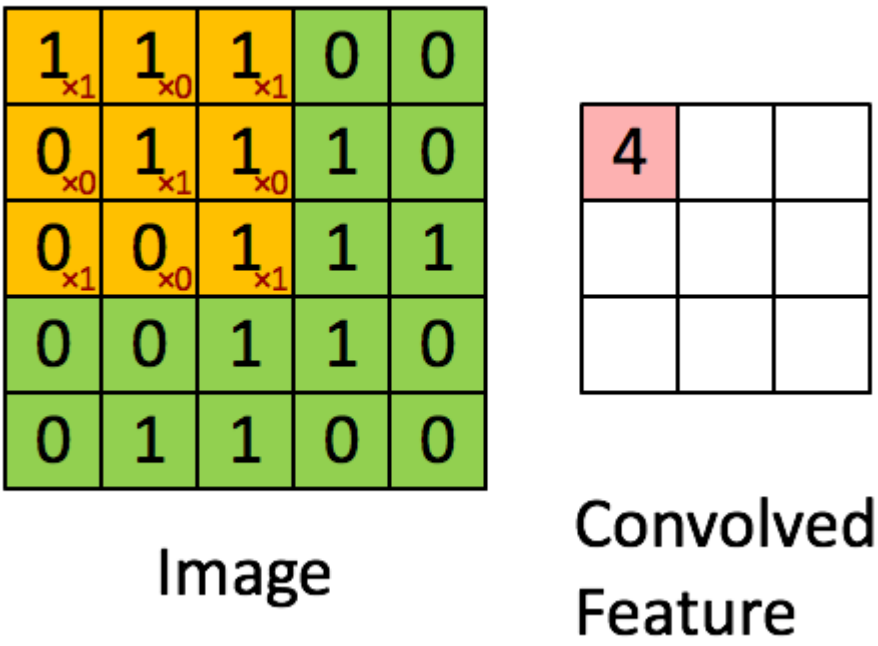
a.) CNN from Scratch:
I have first tried to build a customized CNN network in which I have defined th architecture including the input, output and hidden layers all by myself.

Model Description:
A typical CNN consists of 3 parts:

1. Convolution layer
2. Pooling layer
3. Fully connected layer

Generally speaking, the convolutional layer is responsible for extracting local features in the image; the pooling layer is used to significantly reduce the parameter magnitude (dimension reduction); the fully connected layer is similar to the traditional neural network portion and is used to output the desired result.

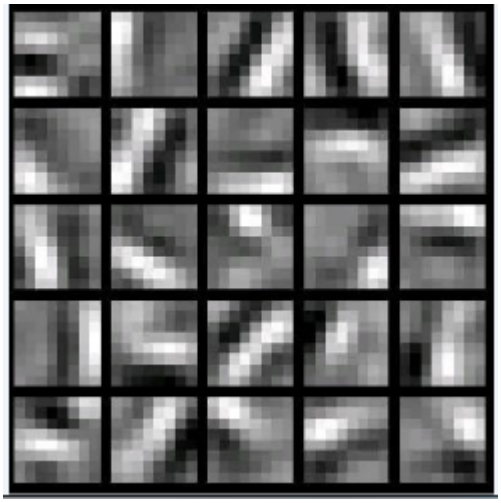
1. Convolution layer:
The operation of the convolutional layer is as shown in the figure below, using a convolution kernel to scan the entire picture:



This process we can understand is that we use a filter (convolution kernel) to filter the small areas of the image to get the eigenvalues of these small areas.

In a specific application, there are often multiple convolution kernels. It can be considered that each convolution kernel represents an image mode. If an image block is convolved with the convolution kernel, the image block is considered to be Very close to this convolution kernel. If we design 6 convolution kernels, we can understand

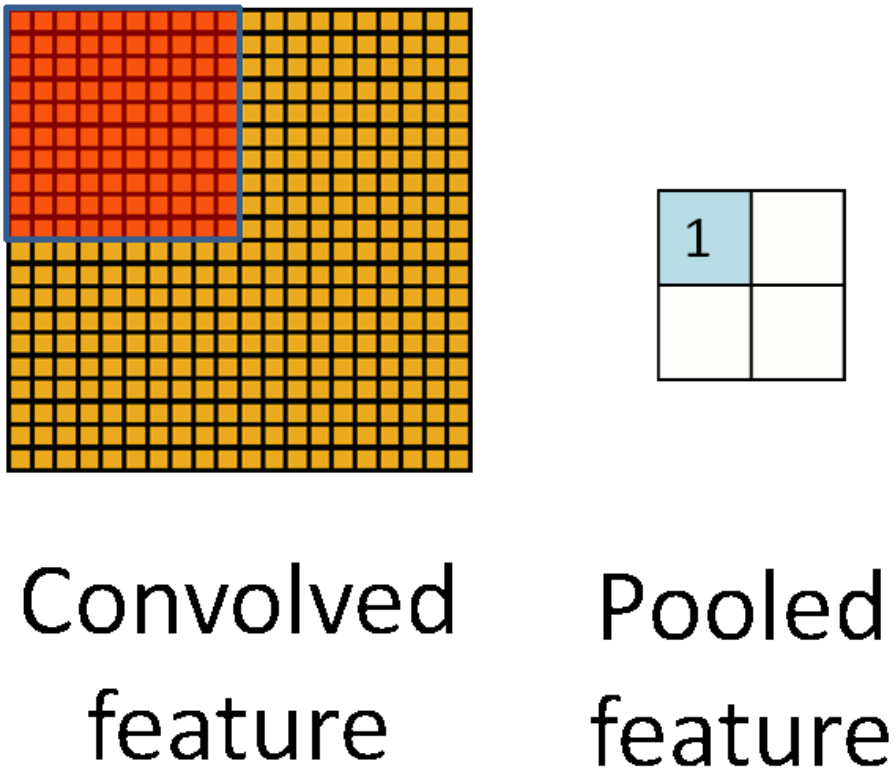
that we think there are 6 underlying texture patterns on this image, that is, we can draw an image using the basic mode in 6. The following are examples of 25 different convolution kernels:



Summary: The convolutional layer extracts the local features in the image by filtering the convolution kernel, similar to the feature extraction of human vision mentioned above.

2. Pooling layer (downsampling) : data dimensionality reduction to avoid overfitting

The pooling layer is simply a downsampling, which can greatly reduce the dimensions of the data. The process is as follows:



In the above figure, we can see that the original picture is 20×20, we downsample it, the sampling window is 10×10, and finally downsample it into a feature graph of 2×2 size.

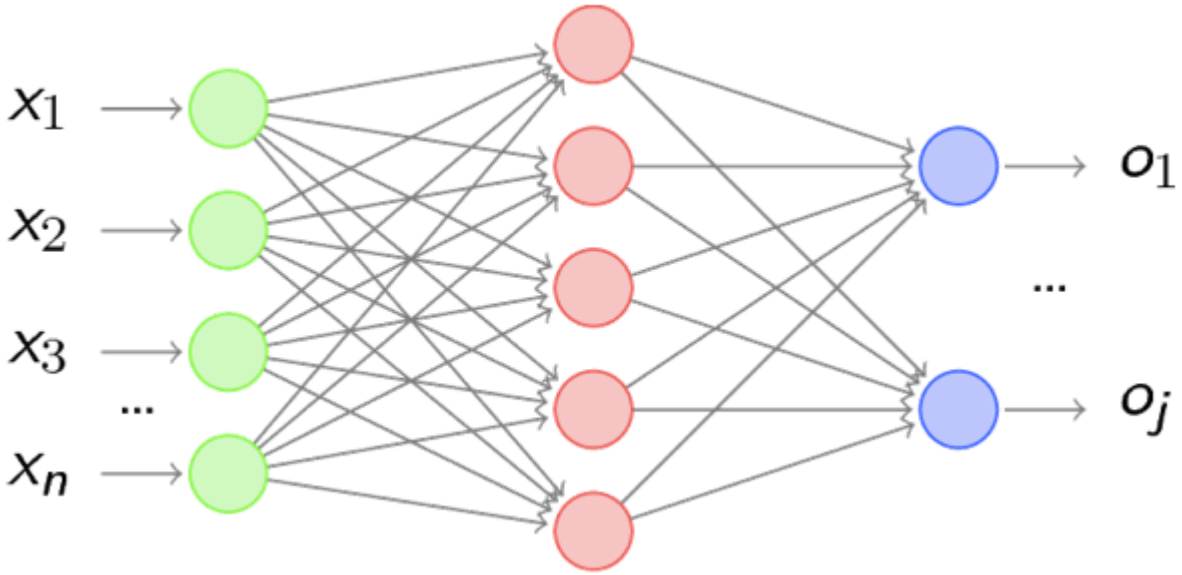
The reason for this is that even after the convolution is done, the image is still large (because the convolution kernel is small), so in order to reduce the data dimension, the downsampling is performed.

Summary: The pooling layer can reduce the data dimension more effectively than the convolutional layer. This can not only greatly reduce the amount of computation, but also effectively avoid overfitting.

3. Fully connected layer - output

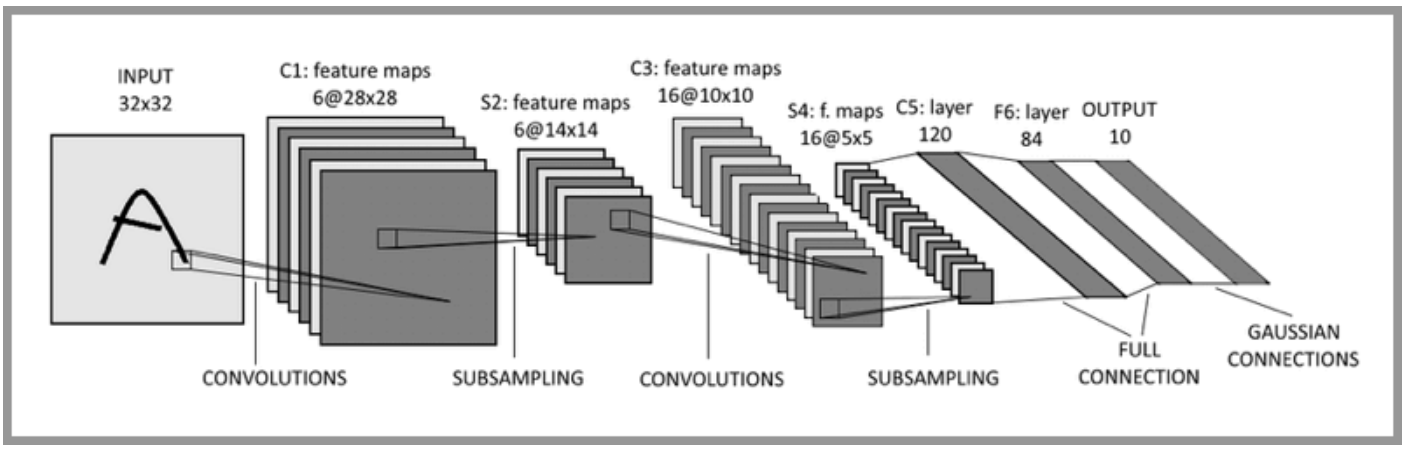
Finally, the data processed by the convolutional layer and the pooled layer is input to the fully connected layer to get the final desired result.

After the data is reduced by the convolutional layer and the pooled layer, the full connection layer can “run”, otherwise the data volume is too large, the calculation cost is high, and the efficiency is low.



A typical CNN is not just the 3 layer structure mentioned above, but a multi-layer structure, such as the structure of LeNet-5 as shown below:

For example: Convolution Layer - Pooling Layer - Convolution Layer - Pooling Layer - Convolution Layer - Fully Connected Layer



Reason for Choice:

Main reason for using CNN for image classification is:

1. Can effectively reduce the size of large data into small data (without affecting the results)

2. Can effectively retain image features, in line with the principle of image processing while preserving the characteristics of the image, similar to the human visual principle.

b.) By Transfer Learning:
In the 2nd approach I made use of an already built & trained model (resnet50 - trained on general images data) and NOT dog specific data. Therefore, modified the classifier slightly to give output as one out of 133 classes.

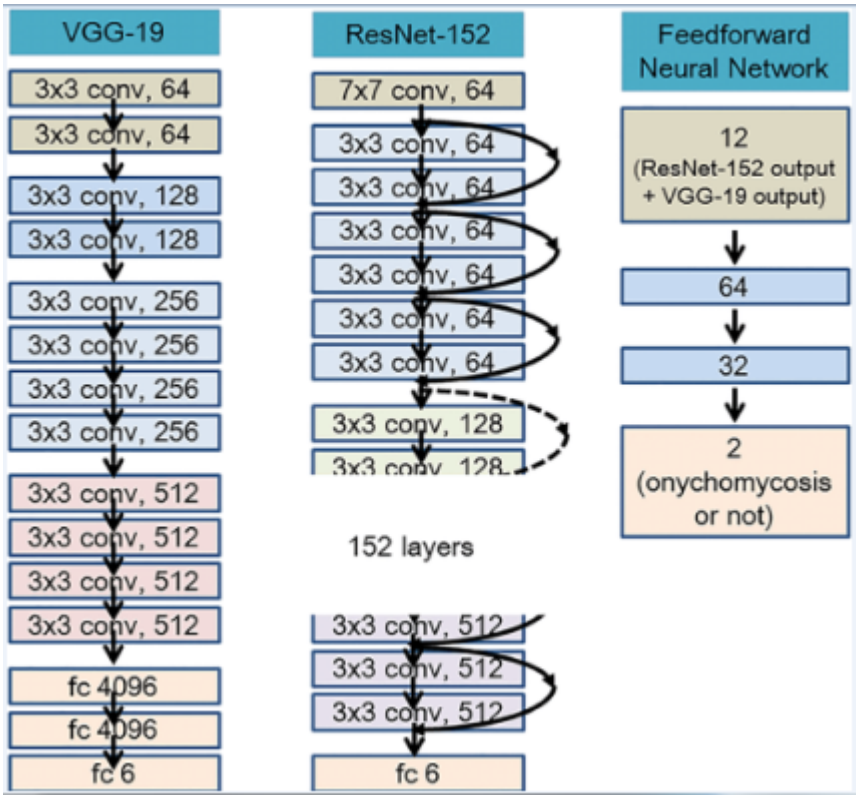
Model Description:

Residual Network (ResNet) is a Convolutional Neural Network (CNN) architecture which can support hundreds or more convolutional layers. ResNet can add many layers with strong performance, while other architectures usually have a drop off in the effectiveness with each additional layer.

Resnet models were proposed in “Deep Residual Learning for Image Recognition”. We have 5 different versions of resnet models, which contains 5, 34, 50, 101, 152 layers respectively.

ResNet proposed a solution to the “vanishing gradient” problem. Neural networks train via backpropagation, which relies on gradient descent to find the optimal weights that minimize the loss function. When more layers are added, repeated multiplication of their derivatives eventually makes the gradient infinitesimally small, meaning additional layers won’t improve the performance or can even reduce it.

ResNet solves this using “identity shortcut connections” – layers that initially don’t do anything. In the training process, these identical layers are skipped, reusing the activation functions from the previous layers. This reduces the network into only a few layers, which speeds learning. When the network trains again, the identical layers expand and help the network explore more of the feature space.



ResNet was the first network demonstrated to add hundreds or thousands of layers while outperforming shallower networks. Although since its introduction in 2015, newer architectures have been invented which beat ResNet's performance, it is still a very popular choice for computer vision tasks.

A primary strength of the ResNet architecture is its ability to generalize well to different datasets and problems.

I have used the resnet50 model available in torchvision.models provided by PyTorch which includes multiple deep learning models, pre-trained on the ImageNet dataset and ready to use. The number 50 in resnet50 indicate the numbers of layers in the model.

The resnet50 pre-trained models expect input images normalized in the same way, with mini-batches of 3-channel RGB images of shape (3 x H x W), where H and W are expected to be at least 224. The images have to be loaded in to a range of [0, 1] and then normalized using mean = [0.485, 0.456, 0.406] and std = [0.229, 0.224, 0.225]

Pre-training lets us leverage transfer learning – once the model has learned many objects, features, and textures on the huge ImageNet dataset, you can apply this learning to your own images and recognition problems.

A simple way to perform transfer learning with PyTorch's pre-trained ResNets is to switch the last layer of the network with one that suits our requirements which is included in the Implementation section.

Reason for Choice:
I have chosen the Resnet50 model specifically because it has a 'Top-5' error rate of just 7.13% and a 'Top-1' error rate of about 23% which seems well-suited for the problem I am trying to solve here.

Benchmark

As stated in the above section, I have built/made use of 3 different models, each one being for a particular task. Having built these models, there is a need for a benchmark/threshold value to be set for each of the models individually with respect to the chosen evaluation metric (in our case - Accuracy), so that each step is implemented with a certain level of accuracy that is expected to be there while intergrating everything into a final application. So I am listing out the benchmark thresholds (in terms of accuracy in (%) for each of the 3 models built) below:

- **Step 1**:
  Human Face Detector (*Atleast 80%*)

- **Step 2**:
  Dog Detector (*Atleast 80%*)

- **Step 3**: Dog Breed Identifier:
  1. Model built from scratch: (*Atleast 10%*)
  2. Model used by Transfer Learning: (*Atleast 60%*)

III. Methodology

Data Preprocessing

The application being built in this project is an Image Classifier which therefore, will take input data in the form of images. Hence, the main data preparation task required would be to convert the jpg images into proper formats and sizes and then transform it into a tensor so that it can be fed to the model.

The pre-processing steps (in more detail) performed on the data before applying it to the model have been listed below:

Step 1:
Firstly, split the data into training, validation and test datasets so that data could be used accordingly to train, evaluate and finally test/evaluate the performance of my model and also created data loaders for each set so that it can be called upon to load the data in a similar way for feeding to the different models built for different tasks as specified in the Algorithms section.

Step 2:

- Most pretrained models including the ones I have used in this project (VGG16/RESNET50) require the input to be 224 x 224 images. As we now want the input images to have dimension as (224, 224, 3), re-sized the images to 224, followed by center-cropping.
- Added some data augmentation such as random flipping and rotation of the images which is usually done as per the standard practice while working with image data.I have taken reference from the labs that I have completed previously (as part of pre-requisite nanodegrees) such as the MNIST case study & the Fashion case study.
- Further also matched the normalization used when the models were trained. Normalized each color channel separately, for which the means were [0.485, 0.456, 0.406] and the standard deviations were [0.229, 0.224, 0.225].

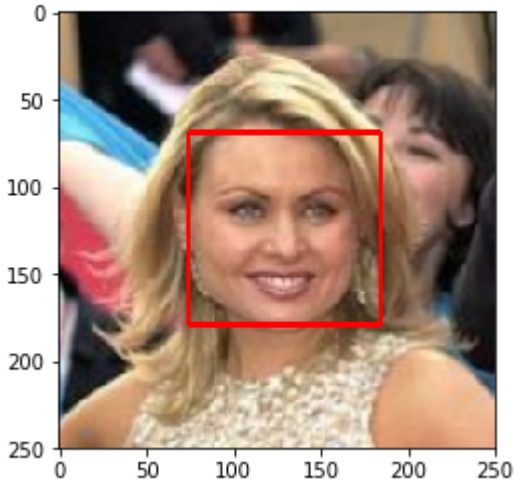
Implementation

Step 1 : Human Face Detection
For detecting the prescence of a human face in the image, used OpenCV's implementation of Haar feature-based cascade classifiers. A Haar Cascade is basically a classifier which is used to detect particular objects from the source. It is a machine learning based approach where a cascade function is trained from a lot of positive and negative images. It is then used to detect objects in other images. Initially, the algorithm needs a lot of positive images (images of faces) and negative images (images without faces) to train the classifier. OpenCV provides many pre-trained face detectors, stored as XML files on github. We have downloaded one of these detectors `haarcascade_frontalface_alt.xml` and stored it in the haarcascades directory. haarcascade_frontalface_default.xml is a haar cascade designed by OpenCV to detect the frontal face which is available on github. A Haar Cascade works by training the cascade on thousands of negative images with the positive image superimposed on it and is capable of detecting features from the source.

It is a standard practice that the colour input images should be converted to gray-scale before being passed to the face detector. Thus defined a function to convert an image to gray-scale by passing the image to function `cv2.cvtColor()` and making use of the `cv2.COLOR_BGR2GRAY` to convert it to gray-scale. Then used the `detectMultiScale()` function which executes the classifier stored in face_cascade and takes the grayscale image as a parameter to give the output.

Finally, wrote a function that returns True if a human face is detected in an image and False otherwise. This function, aptly named `haar_face_detector()` , takes a string-valued file path to an image as input and gives either True or False as output. A simple illustration of how the model works behind the scenes to identify faces can be seen in the below image:

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



Step 2: Dog Face Detection

For dog face detection, I have used a pre-trained model called VGG16 available within `torchvision.models` - a subpackage that contains definitions of models for addressing different tasks, including: image classification, pixelwise semantic segmentation, object detection, instance segmentation, person keypoint detection and video classification.

The VGG16 model that I have chosen is a Very Deep Convolutional Networks used for Large-Scale Image Recognition which is already trained on ImageNet, a very large & popular dataset used for image classification.

When given an image, this pre-trained VGG-16 model returns a prediction (derived from the 1000 possible categories in ImageNet) for the object that is contained in the image.

I have written a function that will accept the path to the image file to be analysed while returning the index corresponding to the ImageNet class that is predicted by the pre-trained VGG-16 model which will always be an integer between 0 and 999.

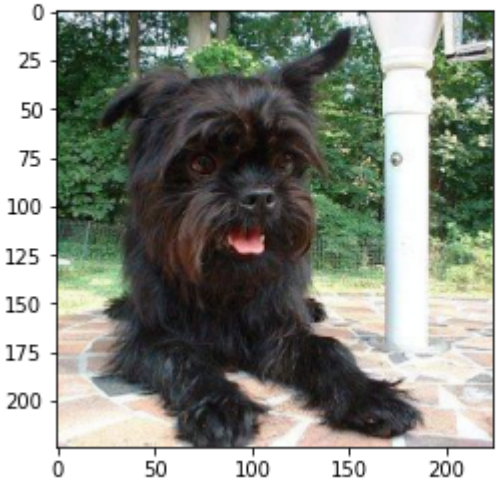
Before passing an image to the above function, I have passed the image to a pre-processing function that would first convert the image into a tensor after being re-sized, center-cropped and all standard data augmentation techniques applied. I have also defined another function that would un-normalize the image and convert it from a image tensor to a NumPy Image so that it could be dsiplayed by using the plt.imshow() function. An example of an image converted from a a regular .jpeg format into a tensor, which is then converted into NumPy array for plotting by using the above 2 pre-process functions is below:

Image displayed before pre-processing:

Tensor Size and Image after Pre-processing:

```
torch.Size([1, 3, 224, 224])

<matplotlib.image.AxesImage at 0x7fc6681dfbe0>
```



Finally, defined the `VGG16_predict()` function which would take an input image path & filename, apply the 2 pre-processing functions above and then pass the pre-processed image data to the VGG16 model to obtain index corresponding to the ImageNet class that resembles the input image we passed.

As stated above, there are 1000 distinct classes (categories) that the VGG16 may classify the input image into out of which classes corresponding to dogs appear in an uninterrupted sequence starting from 151 untill 268. Hence, in this particular task we need our function to only check the input images for which our model predicted class values between the range of 151 and 268 and return a True value, thereby implying that a dog image was found in the image. For any other class value predicted by the model outside this range (151-268) would mean that there is no dog found, thereby causing a False value to be returned.

Step 3: Dog Breed Identification

Now that we have built models for human face & dog face detection, the next step is to finally predict the dog breed given that either a human face or a dog face was detected in an image. I have tried to acheive this goal by using 2 different approaches which are covered in more detail below:

a.) By Building a CNN from Scratch:

First I have built a CNN right from scratch by deciding and defining the architecture on my own including the different types and number of layers. This was an attempt to just see and compare how a customised model performs against a pre-trained models that are used in general.

Before begining to decie the architecture took a look at the shape (dimensions) of the input and output for reference that would give me a starting point for building the network. As the input to this model would be an image, decided the 1st layer to have the shape as (224, 224, 3) while the last layer to have output as 133 possible classes.

Started adding the Convolutional layers and Maxpooling layers (reducing the x-y size of an input, while keeping only the most active pixels from the previous layer), along with the Linear & Dropout layers which are added as usual to avoid overfitting and ultimately give us a 133 dimensional output. Also made use of the MaxPool2d layer to down-sample the input representation thereby reducing its dimensionality, which is very commonly used in such type of Image Classification problems.

Therefore, chose the very 1st layer of my CNN to be an input convolutional layer having shape of (224, 224, 3). I wanted this new layer to have 16 filters each having a height and width of 3 and wanted the filter to jump 1 pixel at a time. Then wanted the layer to have the same width and height as the input layer, so I padded it accordingly to construct the 2nd layer. The layers in between the input & output layers (i.e. hidden layers) usually consist of convolutional layers, ReLU layers, pooling layers, and fully connected layers.

The number of convolutional layers we choose to add, decides the complexity of the images (in terms of color and shape) that our model is able to detect. In simple words, adding more layers results in a more complex model which will be able to learn the images in more detail thereby being able to detect/predict classes for the images more accurately.

I even added a pooling layer that would take in the kernel_size and stride after every new Convolutional lyer was defined so that it could down-sample the input's dimensions by factor of 2. Finally, added fully connected Linear Layer to produce a 133-dim output, along with Dropout layer to avoid overfitting. Made the network feed forward by defining the overall path in which data will be propagted through the network to finally arrive to any one of the 133 possible dog breed classes.

A final summary of the network I had desiged looked as shown in the below image:

```
Net(
  (conv1): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=50176, out_features=500, bias=True)
  (fc2): Linear(in_features=500, out_features=133, bias=True)
  (dropout): Dropout(p=0.25)
  (batch_norm): BatchNorm1d(500, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
)
```

b.) By Using Transfer Learning:

After trying to build a custom CNN, decided to go ahead and make use of a pre-trained model by transfer learning for the sake of getting a higher accuracy and a more efficient model. Transfer learning is a machine learning method where a model developed for a task is reused as the starting point for a model on a second task. This is a popular approach in deep learning where pre-trained models are used as the starting point on

computer vision and natural language processing tasks given the vast compute and time resources required to develop neural network models on these problems and from the huge jumps in skill that they provide on related problems.

Pre-trained networks are efficient and well suitable to solve challenging problems because once trained, these models work as feature detectors even for images they werent trained on.

Here I have used transfer learning to train a pre-trained network so that it is able to classify our dog images. I have used the "resnet50" model trained on ImageNet which is readily available for use in torchvision.models The classifier part of this model is a single fully-connected layer which is (fc): Linear(in_features=2048, out_features=1000, bias=True) that was trained on the ImageNet data which did not have specific dog breed data.

This meant that I now had to replace the classifier with (133 classes), while the features continue to work perfectly on their own. And so chose the nn.CrossEntropyLoss() criterion which combines the functions :nn.LogSoftmax and nn.NLLLoss in a single class which has been proven to be useful when training for image classification problem with multiple classes.

A simple way to perform transfer learning with PyTorch's pre-trained ResNets is to switch the last layer of the network with one that suits our requirements by freezing the ResNet layer as we don't want to train and pass the remaining parameters to our custom optimizer. Therefore, froze the parameters for the model and replaced the last fully connected layer with a linear layer of 133 output features through the below code which did the thing:

```
# Freeze parameters so we don't backprop through them
for param in model_transfer.parameters():
    param.requires_grad = False
# Replace the last fully connected layer with a Linnear layer with 133 out features
model_transfer.fc = nn.Linear(2048, 133)
if use_cuda:
    model_transfer = model_transfer.cuda()
```

Algorithm of Final Application:

Implemented an algorithm, that does the following:

1. Applies pre-processing to the input image.
2. Checks for a dog's face and returns its breed if found.
3. Checks for human face, and if found returns the dog breed which resembles the human's face.
4. Handles the error in an interactive manner if neither dog or human face is found in the image.

Refinement

First, I had tried running my custom CNN without adding the pooling layers and could see that it ran much slower and the accuracy was much lower somewhere around 5%. Then I utilized max pooling to reduce the dimensionality and also because it makes the CNN run faster. Could see that Pooling was safe in this case since it is likely that neighboring pixels are similiar and downsampling will not reduce the classifiers ability to detect the image.

I even increased the filter selection from 3 to 16 to 32 to 64 which I read to be a standard practice while designing CNNs after which I was finally able to get an accuracy score of 15%.

Furthermore, the CNN without transfer learning consumed lot of time and was not satisfactory in terms of performance too with meagre accuracy of 15%. Thus to reduce training time without sacrificing accuracy, I trained a CNN using transfer learning. The model used was a pre-trained resnet50 model which gave a much better result of 81% with transfer learning.

IV. Results

Model Evaluation and Validation

I have built/made use of 4 different different models in this project, each one being for a particular task as stated before. I have tested and evaluated each of these 4 models individially so that I can compare each model to its bench-mark that I had set.

Step 1: Human Face Detector

As stated earlier used openCV's haar based cascade classifier to detect human faces. Tested this model by using 100 human images and 100 dog images each and see how correctly the model was able to classify the images. Ideally, we would want our model to classify 100% of human images with a detected face and 0% of dog images with a detected face. However in reality, could see that our model:

- Detected human faces correctly in 98 human files correctly out of the 100 human files given.
- Detected 17 human faces in 100 dog files i.e. 17% (misclassification).

Therefore, the final accuracy of the score as per the formula stated in the Metrics section can be calculated as :

Accuracy = $\frac{C}{T} \times 100$ = $\frac{98+83}{200} \times 100$ = 90.5%

where,
C -> Number of correctly classified images
T -> Total number of images to be classified

Step 2: Dog Detector
Similarly evaluated the pre-trained VGG16 model used to detect dog faces again with a total of 100 dog image files and 100 human image files. In this case ideally, wanted our model to classify 100% of dog images with a detected dog face and 0% of human images with a detected face. The model did a splendid job where in it was able to give us the below result:

- Detected a dog face only in 1 image out of the 100 human image files (mis-classification).
- Whereas was able to correctly detect dog faces in all 100 dog images.

Hence, the final accuracy of the score as per the formula used above can be calculated as :

Accuracy = $\frac{C}{T} \times 100$ = $\frac{99+100}{200} \times 100$ = 99.5%

Step 3: Dog Breed Identifier

a.) CNN from scratch:
For the CNN built from scratch, I had managed to get an overall accuracy of about 15% on the test set even though I had run 12 epochs while triaing the model.

b.) By use of Transfer Learning:
The pre-trained resnet50 model I used for dog breed identification by transfer learning gave me an overall accuracy score of about 81% on the test with 12 epochs while training. I had trained the CNN I designed and also the pre-trained network used by transfer learning by using the same number of epochs (12), so that it would be clear which model is a better performer. Resnet50 used by transfer learning method was a clear-cut winner by giving a significantly high accuracy score of 81%

Justification

From the test evaluation results we can safely say that all the models built for each of the tasks have performed better than the benchmark model that was initially specified.

Step 1: **Human Face Detector** (*Expected atleast 80%*)
The human detector that I had built evaluated to work with an accuracy score of 90.5% which is considerably higher than the expected score of 80%. The model has performed atleast 10% better than the benchmark specified earlier. And therefore, by looking at the results I can say that the final solution does seem significant enough to have solved the problem efficiently.

Step 2: **Dog Detector** (*Expected atleast 80%*)
The dog face detector built using the pre-trained VGG16 model evaluated an accuracy score of whopping 99.5% which is exceptionally higher than the expected score of 80%. The model has performed the best it can wherein it ended up classifying an image only once in 200 atempts which makes it much better than the benchmark specified earlier. Hence, in this case too, I can undoubtedly say that the final solution for this particular problem seems more than just significant to have solved the problem efficiently.

Step 3: **Dog Breed Identifier**
a.) **Model built from scratch**: (*Atleast 10%*)
The CNN that I had built from scratch with a custom architecture was obviously not expected to perform as well as any other pre-trained model would have. And becasue of this I had set a benchmark threshold for this particular model to have an accuracy score of just 10% (atleast) which would be much better than guessing randomly. My custom-amde CNN however performed better than the expected benchmark threshold with an accuracy score of 15% on the test set. This is not record-breaking or a very good model but managed to surpass the set threshold by a margin of 5%. I had built this particular model as part of this project to only get an experience of building a network by myself and see how it compares against using pre-trained models using transfer learning.

b.) **Model used by Transfer Learning**: (*Atleast 60%*)
Finally could see that the model built using the pre-trained resnet50 model by transfer learning proved to be the best for predicting dog breeds having an accuracy score of 81% on the test set. It was expected for this model to perform significantly better than my custom CNN model as it is a known fact that pre-trained models are much better at such tasks than the ones we design ourselves. With the accuracy score as 81%, this model performed 21% better than our pre-defined benchmark threshold of 60% which is considerably higher to have solved the problem of identifying dog breeds, given the fact that we as humans would also not be able to be classify dogs with such a high accuracy especially for the ones that look very similar to each other.

V. Conclusion

Reflection

This was a very challenging problem statement for which I have come up with an end-to-end solution wherein from the time an image is passed, its pre-processing, step-wise identification of either a dog's or a human's face is done while finally identifying the breed which resemlbes the image most closely. This is a task which we as humans would struggle as it is very difficult to firstly memorize all 133 breeds and then be able to accurately classify/predict the breeds that to with an accuracy as high as 81%.

It has been a great learning experience while trying to solve this challenging problem statement. I was forced to look up the net for solutions whenever I was getting stuck in certain stages of the project. One such occassion which was a little difficult was where I had to design and build a CNN of my own right from scratch, tailor-made to the needs of this project. Although this particular task was challenging, it also did seem to be the most interesting part of the project where I literally built something on my own, just however before making use of a pre-trained model for obvious reasons!

The accuracy of 81% is certainly more impressive than the 15% accuracy that I could achieve by using my own CNN. The reason the accuracy was so low with my CNN is because the resnet50 model I used, was built on the ImageNet library, & the knowledge of which I had managed to transfer to my problem here (dog breed identification). The imagenet libary has quite a large number of images (approx. 1.2 million) including images of dogs. This meant that the transfer learning I did here was based on a similar dataset, at least partially.

Improvement

There are countless number of possibilities & ways any particular work/task maybe improved to make it better and I am sure there maybe a lot that could be done in this case too. A couple of potential improvements which I could come up with and could be made to improve this particular piece of work are:

- Currently model cannot handle the scenario where in there are either multiple humans/dogs in one image. So the model could be improved to deal with such scenarios as well and return an optimum output.
- Currently the model is just giving us only 1 breed which it think resembles the one in the input image the most. This can be changed to make the model return the top N predicted classes (as specified by user) along with their corresponding probabilities, instead of giving just 1 single breed as output.
- Currently the project can be accessed only via Python Console, however this could be deployed as an API in AWS with a beautiful little UI where the user can either upload the image to be checked or pass a link to the image.