Capstone Project

Kim Harrington
September 10, 2020

I. Definition

1.1 Project Overview

Image recognition is a field in computer science that applies specific technologies such as machine learning algorithms and strategies to recognizing and/or identifying certain objects of interest such as animal or numerous other vision-based subjects. This field applies deep learning neural network architectures to solving image recognition challenges.

The UDACITY capstone project required to satisfy the Machine Learning Engineer certificate is to build a deep learning Convolutional Neural Network (CNN). I selected the Dog-Breed Classification project that is offered by UDACITY and this project will use deep learning neural networks to solve image recognition by classifying dog images into 133 different breed classifications.

The project objective is to write an algorithm for a dog identification application using CNN. A quote from the ***Wall Street Journal*** dated May-30, 2020 in an article entitled 'AI Won't Help You Reopen Your Business', Gary Marcus, professor at New Your University said, regarding Deep Machine Learning systems, that "they're just big engines for finding statistical correlations". Essentially, the scope of this project is to develop the dog identification algorithm by building a CNN model using two different approaches each of which determines the probability that the input image matches a specific class of dog breed or not. The two approaches to be applied are: (1) build the CNN from scratch and use various metrics to determine accuracy; (2) Leverage a selected pre-trained CNN models by replacing the model's classifier with one that is specifically trained for dog-breed classification; this technique is known as transfer- learning. The algorithm will receive various images and determine if the input is a dog, if it is which of the 133 breeds does it match. If the image is a human-face, it will specify that the image is human and ascertain which dog-breed that input matches probabilistically. Should the image be anything other than a dog or human-face, the algorithm will indicate that the input is neither and communicate that the input is an error.

The project work was preformed using the UDACITY 'Dog Project Workspace' notebook. The technology applied to the project are Python V3 and the Pytorch machine learning libraries and the benchmark model is VGG-16 which is well documented on the internet achieving accuracies of 90%+ across the ImageNet data set. ResNet50 was also considered as the benchmark model, but too many students applied it so I selected VGG-16. The issue with the benchmark model versus the models created by this project is the size of the data set for this project; it is roughly three orders of magnitude smaller than the VGG-16 model using ImageNet and this difference will be reflected in the accuracies achieved by each model. The data in the workspace is for training, validation, and testing. All the data is labeled so there is no need to write code to label it with on-hot-encoding.

1.2 Problem Statement

The objective is to write an algorithm that predicts image input as a specified dog breed using two approaches: (1) build the CNN from scratch and; (2) apply transfer - learning using a pre-trained robust model to make probabilistic predictions on input images. The Pytorch machine learning libraries are used to construct the CNN model from scratch and building a new classifier for the transfer learning model. Since all the images are already labeled there will be no need to write code to label each image. Python version 3+ will be used throughout this initiative.

To build the CNN from scratch I will use the traditional architecture consisting of one or more convolutional layers followed by Fully-Connected layers. As for the convolutional layers I will apply Conv2D exclusively since this is commonly used to solve these types of image classification. Essentially, I will determine the number of Conv2D layers required, determine the filter/Kernel sizes, apply pooling techniques to conduct feature reduction, and use the ReLu function for activation. For the Fully-Connected layer(s) I will specify the hyper-parameter dropout and apply the softmax function to the final output.

To solve the transfer model approach, I will select the pre-trained model from one of the models identified in the table below. To determine which model to select I will consider two factors, size in megabytes and accuracy. The table below presents the accuracy and size of each mode and it is from 'https://keras.io/api/applications/'

MODEL	REPORTED ACCURACY (TOP-5)
DENSENET-169	93.2% Size = (57 MB)
RESNET50	92.1% Size = (98MB)
VGG16	90.1% Size = (528 MB)

UDACITY has provided a human-face detector and a corresponding data set to use in the algorithm development. I will write a dog-breed detector using a pretrained model, VGG16, as well as the ImageNet data set and its indexes, specifically those index ranges that classifies dog breeds.

Algorithm development will use the CNN transfer learning model as well as the UDACITY supplied human-face detector and the dog-breed detector built in this project. Code will be written to display the input images along with a text message stating the dog-breed type if a human or dog is detected, else communicate an error. I will save the best classifier model identified during training and load it into a standard CPU environment which the algorithm will also execute within. To save the model I will define procedures to save and reload the trained classifier. The completed algorithm will take as input an image and it will:

- Determine if the input image is a dog or human-face; then predict the breed and present the image;
- If the image is none of the above communicate an error.

The UDACITY notebook has outlined six executable steps these steps serve as tasks to materialize the project:

1. Import Datasets:
    a. Import the data sets supplied in the workspace
    b. Import the human-faces images into the workspace using OpenCV's implementation of Haar feature-based cascade classifiers
    c. Load the images into numpy arrays to be used by the next three steps.
2. Detect Humans:
    a. Use the UDACITY supplied human-face detector code to detect human faces,
    b. Test the code against the dog and human-face images.
3. Detect Dogs:
    a. Download the pretrained image classifier VGG16,
    b. Build an image display procedure and a predictor procedure using VGG16 that returns an index into ImageNet,
    c. Develop a dog detector procedure,
    d. Test the code against the dog and human-face images.
4. Create a CNN to Classify Dog Breeds (from Scratch)
    a. Conduct data exploration and visualization, preprocess the data,
    b. Build the data transforms for the entire data set and build the data loaders,
    c. Define and build the CNN model architecture,

    d. Train the model and save the best model with the highest accuracy,
    e. Use the UDACITY Test procedure and achieve an accuracy >= 10%.
5. Create a CNN to Classify Dog Breeds (using Transfer Learning):
    a. Select a pretrain image classifier as the transfer model,
    b. Define the dog -breed specific classifier architecture,
    c. Train the model saving the best one with the highest accuracy,
    d. Use the UDACITY Test procedure and achieve an accuracy >= 60%.
6. Write Your Algorithm:
    a. Write the algorithm using the transfer learning dog-breed classifier, the human-face and dog detectors, and image display,
    b. Test against sample images.
7. Test Your Algorithm
    a. Test the algorithm against various images of human-faces, other animals, dog images.

2.

1.3 Metrics

The metrics used in the project are:

1. Accuracy score, number or correctly predicted images/ total number of images
2. Test Loss; where$\sum_{1}^{\text{Total Num Batches}}$ test_loss + ((1 / (batches + 1)) * (loss.data - test_loss))
3. F1-Score; where F1 = 2 * (precision * recall) / (precision + recall)
4. Confusion matrix:
    a. Specificity: True Negative rate per class,
    b. Sensitivity: True Positive rate per class,
    c. The number of True Positives on the Diagonal.

    The confusion matrix will show which categories the model is predicting correctly and which categories the model is predicting incorrectly. For the incorrect predictions the matrix will identify which categories are confusing the model.

Items 1-4 were applied on the scratch model and this is primarily due to the nature of the training data set which is imbalanced and left skewed. The F1 score determines the model accuracy between zero and one probability for imbalanced data sets; the closer to one the more accurate the model. The confusion matrix basically shows accuracy results on the diagonal of the 133x133 breed classes, if for example the model accuracy result was near 90+% the diagonal will show all the true-positives 'hits' while the specificity would be low and the sensitivity high. Since the transfer model used a proven pretrained CNN model with an advertised accuracy of 90+%, I only applied the accuracy score and test loss to the transfer learning exercise.

II. Analysis

2.1 Data Exploration

Since the data for this project is supplied in the workspace there is two types human-face and dog images. The directory for data follows with information on the number of images per data type:

Directory path	Number of files	Type
• /data/dog_images/	8,351	Dog Images
• /data/lfw/	13,233	Human Images

Directory path	Number of files	Purpose
• /data/dog_images/train/	~6,680	Training images data set
• /data/dog_images/valid/	~ 835	Validation images data set
• /data/dog_images/test/	~ 836	Test images data set

To support the detector exercises numpy arrays were created to store the human and dog images using glob into human_files and dog_files respectively. The human images have a shape of 250x250 pixels with a Red/Green/Blue channel. I determined this by reading in images, i.e., img = cv2.imread(human_files[0]) then printed the img.shape and; then I built a data-frame to obtain the statistics. Repeating this process for dog images and examining various files the shape of the dog images varied significantly, for example I looked at img = cv2.imread(dog_files[1010]) and its shape was 805x871 pixels. Thus, concentrating on the data/dog_images/train/data set I built a data-frame for the dog images. Using this code; df_dog_img.median(), df_dog_img.mean(), df_dog_img.std() I arrived at the data set characteristics:

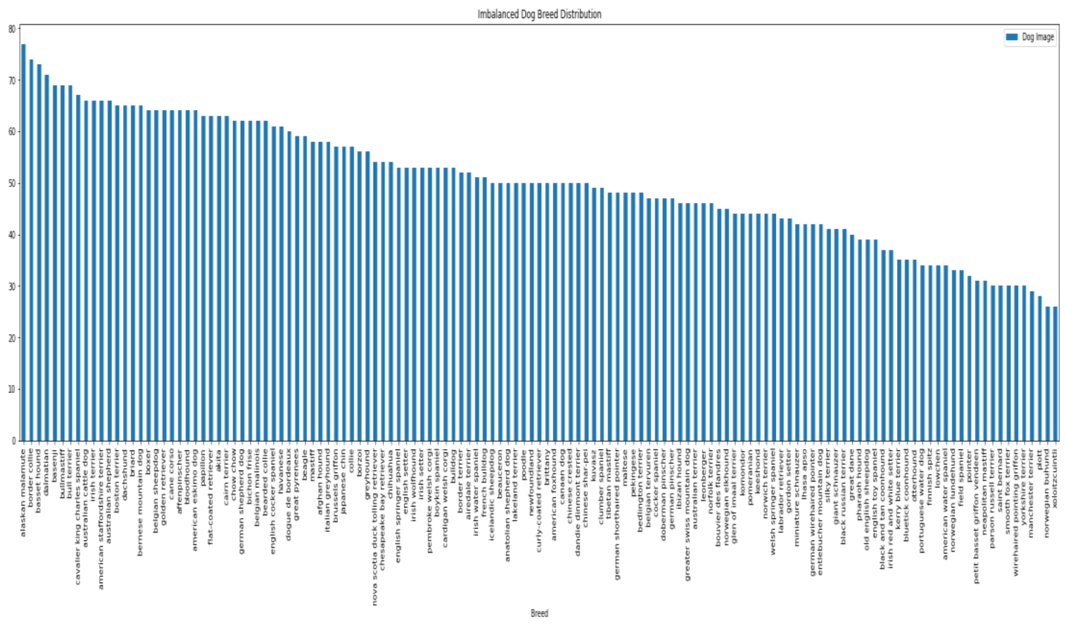
DOG IMAGES	PIXEL HEIGHT	PIXEL WIDTH
MEAN	532.16	571.38
STANDARD DEVIATION	341.6	397.5
MEDIAN	466	500

Using the dog image data-frame, I counted the occurrence / frequency of images per class and arrived at the average number of images per class is 50 with a standard deviation of 12. Noteworthy is the high standard deviation value of the dog image data set indicating significant image size variations (sigma squared), however, on the other hand the median works well as a guide on how to crop the images for training.

The final point about the data is that all of it are images so feature processing will have to be addressed during the CNN convolution process, therefore, there is no further data processing for feature reduction in advance of convolution.

2.2 Exploratory Visualization

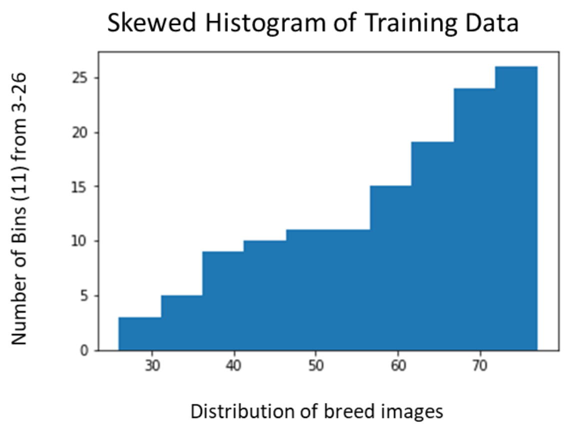
Data visualization will reveal the true nature of the data set. What we are interested in is determining if the data set is balanced or imbalanced, i.e., skewed from a normal distribution and obtaining a sense of which classes have the greatest frequency of occurrence. The following bar graph represents the distribution of dog-breed classes of the training data. There are 133 breed types in total. Looking at the distribution indicates an imbalanced data set. The data analysis was conducted against the training data set of a total 6,680 images.



Producing a proper histogram of the training data and obtaining the distribution of dog breed class occurrences resulted in this array output of the plot, showing the data distribution imbalance. Notice that the range is from 26 to 77 images per dog-breed.

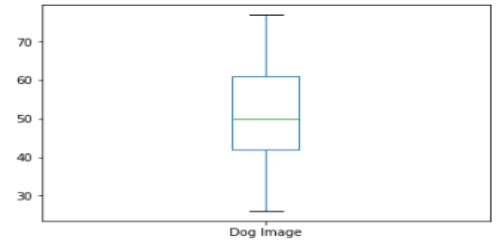
```
array([ 26. ,  31.1,  36.2,  41.3,  46.4,  51.5,  56.6,  61.7,  66.8,
        71.9,  77. ]),
<a list of 10 Patch objects>)
```

This histogram below has 11 bins ranging from 3 to 26 on the Y-Axis and distribution of breed images on the X-Axis.



Again, the median for the image count per dog breed class is 50 with a standard deviation of 12. To get a sense of the level of skewness we would have a value of ZERO for balanced data sets, but this is an asymmetrical distribution which is a negative skew toward the left. Skewness analysis revealed the following:

```
Skewness should be 0 for a Balanced data set,i.e., denotes a symmetrical distribution
  the value of skew is negative so the skewness is an asymmetrical distribution;
  the asymmetrical distribution is toward the left side of the curve.

Skewness = :  Dog Image   -0.07762
dtype: float64

Kurtosis = :  Dog Image   -0.728101
dtype: float64

Median = :  Dog Image    50.0
dtype: float64
```



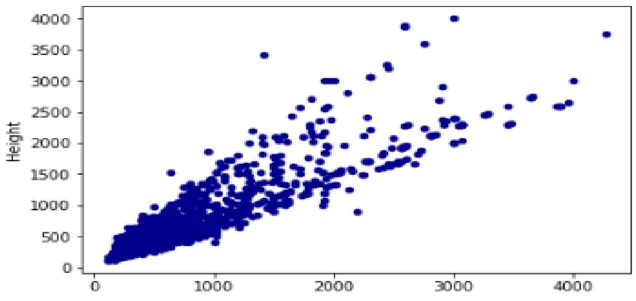
The chart above clearly shows the extent of the data image outliers of the dog-breed images which is show below as a scatter plot of the data. The Kurtosis score is also negative at -0.728 indicating left skewness.

Calculating the training data set image dimension variation of width against height will provide data for the parameters of the Pytorch transformations to be used for the training, test, and validation data sets.

The scatter plot along with the image statistics follow.

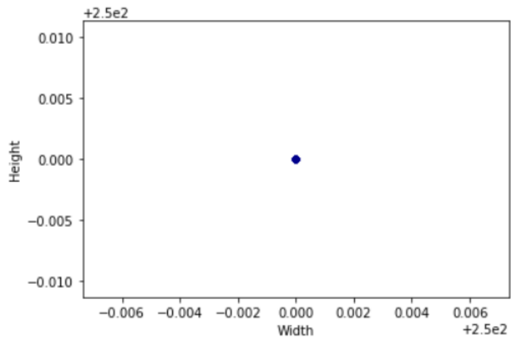
IMAGE	**MEAN**	**STANDARD DEVIATION**
HEIGHT	532.2 pixels	341.6 pixels
WIDTH	571.4 pixels	397.5 pixels

```
Dog Image Median: Height    466.0
Width     500.0
dtype: float64  Mean: Height    532.157485
Width     571.382335
dtype: float64  Standard Deviation Height    341.626020
Width     397.480566
dtype: float64
```



Examining the human-face images by plotting the data-frame histogram showed statistically that these images are all square 250 x 250 pixels with a standard deviation of ZERO.

```
Median : Height    250.0
Width     250.0
dtype: float64    Mean : Height    250.0
Width     250.0
dtype: float64   Standard Deviation Height    0.0
Width     0.0
dtype: float64
```

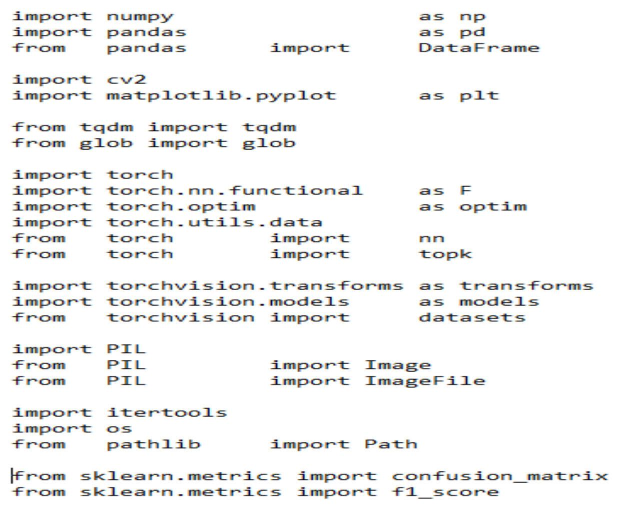


2.3 Algorithms and Techniques

This project has six different sub-projects to solve, I will describe the algorithms for each of the following:

1. Detect Humans,
2. Detect Dogs,
3. Create a CNN to Classify Dog Breeds (from Scratch),
4. Create a CNN to Classify Dog Breeds (using Transfer Learning),
5. Write Your Algorithm,
6. Test Your Algorithm

The following libraries will be used throughout this project:



```
import numpy                          as np
import pandas                         as pd
from   pandas         import          DataFrame

import cv2
import matplotlib.pyplot              as plt

from tqdm import tqdm
from glob import glob

import torch
import torch.nn.functional            as F
import torch.optim                    as optim
import torch.utils.data
from   torch          import          nn
from   torch          import          topk

import torchvision.transforms as transforms
import torchvision.models      as models
from   torchvision import        datasets

import PIL
from   PIL            import Image
from   PIL            import ImageFile

import itertools
import os
from   pathlib        import Path

from sklearn.metrics import confusion_matrix
from sklearn.metrics import f1_score
```

For items one and two the numpy arrays that loaded the images into memory are human_files and dog_files, these will be used to create dog_files_short and human_files_short where each will contain the first 100 images.

2.3.1 Detect Humans

As specified within the notebook, OpenCV's implementation of Haar feature-based cascade classifiers is used to detect human faces in images. OpenCV provides many pre-trained face detectors, stored as XML files on github. The notebook has downloaded detectors which are stored it in the haarcascades directory.
UDACITY has provided the code for the face_detector().

The 'tqdm' library will be imported into the workspace environment so that I can conduct the detector performance and time it against the 100 images in human_files_short. A 'tqdm' instance will be given a range from zero to 100 and used in a loop. Within the loop there will be a conditional statement using the face_detector to test each image. A counter will count the number of images that are human faces. This count will be divided by 100 to determine if percentage of all the images that are human. 'tqdm' methods will be used to clear, refresh and close the tqdm object instance.

2.3.2 Detect Dogs

To construct the dog detector, I will use of the pretrained VGG16 model. A few procedures will be built to use this model and display the images these we are follows:

PROCEDURES	**FUNCTION**	**COMMENT**
VGG16_PREDICT	Pass an image path and returns the index corresponding to the predicted ImageNet class.	Called by dog_detector Run the image through the model with gradient decent turned off.
DOG_DETECTOR	Pass an image path and returns True or False	Calls VGG16_predict to get index into ImageNet

I will repeat the same technique in the human face performance test using the 'tqdm' library and its methods for the dog detector.

2.3.3 Create a CNN to Classify Dog Breeds (from Scratch)

For this architecture I will use Conv2D, i.e., Convolutional Layer – 2-Dimentional. A 2D convolutional layer is the most common approach to writing an image classifier. I spent many hours reading articles on the internet and within the knowledge section of this course learning the 'what', 'how' and 'why' of constructing convolution layers and their processing relationship to fully connected network layers. The CNN layers are all about feature reduction or preprocessing of the input data. Each CNN layer transforms and reduces the input and uses the filters for feature recognition. I will use various 'kernel' sizes (starting with 5) or moving-windows on each filter. I will experiment with various filters sizes and increase the sized in subsequent layers. Pooling, which is another sliding window technique, will be used to reduce the images from one layer to the next; max-pooling is the most used to literally cut the features size / Feature-Reduction down where a 2x2 max-pool filter is planned. Within each CNN layer I will apply the ReLu activation function. Ultimately, I plan on five-layers of convolution with max-pooling followed by three layers of a Fully-Connected Network.

I calculated the total number of training parameters using a spreadsheet since Pytorch does not have a straight forward utility like tensor-flow has. I used the format from tensor flow to show how I calculated the total number of training parameters manually. For each Conv2D layer I applied this formula:

$$\text{Training\_paramteres} = \left[\sum_{i=1}^{\text{Total-Layer}} (\text{Kernel-Size}^2{}_i) * (\text{in\_filters}+\text{Bias}_i) * \text{Out\_filters} \right] + \quad \text{\# Convd2D Layers}$$

$$\left[\text{image\_size}^2 * (\text{input} + \text{Bias}) * \text{output} \right]_{FC1} \quad + \quad \text{\# FC1 layer}$$

$$\left[\sum_{i=2}^{\text{Total-FC}} (\text{input} + \text{Bias}) * \text{output} \right] \quad \text{\# FC2 …FC3}$$

Conv2d Parameters:

- Stride: = 1
- Padding: = 1

- Kernel Size: = 3; Started with 5 and worked my way down
- Image input: 224x224, that is the Height to Width
- Dilation:	= None
- Bias		= 1

Below is a table that I built to calculate the potential number of total training parameters. This table will be used to define my scratch-model CNN NET-Class and all associated parameters. The key parameters are: Kernel-size, in_filters and out_filter size, image size, stride, padding, and the height and width of the input image, and bias. Each of these parameters are specified in the 'Calculation of Total CNN-Model Training Parameters' table. Actually, this is the final results table.

Calculation of Total CNN-Model Training Parameters

Layer (type)	***Output Shape***	***In_Filters***	***Out_Filters***	***Image Size***	***Training Parametes***			
conv2d_1 (Conv2D)	(None, 224, 224, 32)	3	32	***112***	**896**	(Kernel**2 * Initial input Filters or RGB-Chan * Out-filters		
max_pooling2d_2 (MaxPooling2	(None, 112, 112, 32)							
conv2d_2 (Conv2D)	(None, 112, 112, 64)	32	64	***56***	**18,496**	(Kernel**2 * Previous Filter +1)*out-filters		
max_pooling2d_3 (MaxPooling2	(None, 56, 56, 64)	0						
conv2d_3 (Conv2D)	(None, 56, 56, 128)	64	128	***28***	**73,856**	(Kernel**2 * Previous Filter +1)*out-filters		
max_pooling2d_4 (MaxPooling2	(None, 28, 28, 128)	0						
conv2d_4 (Conv2D)	(None, 28, 28, 256)	128	256	***14***	**295,168**	(Kernel**2 * Previous Filter +1)*out-filters		
max_pooling2d_5 (MaxPooling2	(None, 14, 14, 256)	0						
conv2d_5 (Conv2D)	(None, 14, 14, 512)	256	512	***7***	**1,180,160**	(Kernel**2 * Previous Filter +1)*out-filters		
max_pooling2d_5 (MaxPooling2	(None, 7,7, 512)	0						
dropout_1 (Dropout)	(None, 7, 7,512)							
global_average_pooling2d_1 (	(None, 512)	0	0			***FC1***	25,740,288	'(input +bias)*output
						FC2	512,500	
dense_1 (Dense)					<u>**26,319,421**</u>	***FC3***	<u>66,633</u>	
							26,319,421	
					27,887,997	= number of trainable parameters		

I will Train the model using the dog-image train data set and save the best model with the highest accuracy and use the UDACITY Test procedure to achieve an accuracy >= 10%.

2.3.4 *Create a CNN to Classify Dog Breeds (using Transfer Learning)*

The process applied to determine the transfer model was to first determine which pretrained CNN model to use and replace its classifier with a specific model designed for dog-breed classification. There are multiple choices to select form namely DenseNET-169, ResNet50, VGG16. The table below presents the accuracy of each mode and it is from 'https://keras.io/api/applications/'

MODEL	**REPORTED ACCURACY (TOP-5)**
DENSENET-169	93.2% Size = (57 MB)
RESNET50	92.1% Size = (98MB)
VGG16	90.1% Size = (528 MB)

I intend to choose the VGG16 model. I will build built a net NET-Class using the Sequential specification to design the classifier which will replace the VGG16 model classifier. I will extract the number of in_features as specified in the VGG16 model, i.e., 25,088. I plan to start with 2 and 3 hidden layers. I will use the NLLLoss criterion which requires the softmax normalize process to be applied after the output layer for probabilities. I will adjust the Dropout parameter between 20% to 50% during the training/learning process. I plan on using the best-trained model for the project Algorithm so that I can run it in CPU mode and not GPU. I plan to reuse the same transforms used to create the scratch CNN model. I will Train the mod el using the dog-image train data set and save the best model with the highest accuracy and use the UDACITY Test procedur e to achieve an accuracy >= 60%.

2.3.5 Write Your Algorithm

To write my own algorithm I will define the ‘run_app(image_path)’ procedure. This code will use ‘dog_detector()’ and ‘face_detector()’ procedures. I will use the ‘best’ saved model of the transfer classifier. I will write a couple of new procedures to save and load the model making it run in CPU mode, i.e., ‘save_model(model, save_path)’ and ‘load_checkpoint(filename)’. The save_model specifically saves the new classifier’s state_dic and the newly trained classifier. The load_checkpoint() procedure returns the transfer classifier model. To set up the environment to ‘run_app()’ I will use this command:

model_transfer = load_checkpoint('model_transferBEST.pt')

I plan to write a predict_breed_transfer (image_path) procedure to test and see if it would predict a dog breed when submitting a human. Class_names as indexes will be used to capture from the training data set, i.e., train_data.classes. I will test the run_app on many dog and human images.

2.3.6 Test Your Algorithm

Test your algorithm at least six user provided images on your computer and for each image show the predicted dog breed else an error statement if not a human-face or a dog image.

2.4 Benchmark

Initially, the bench mark model that I chose for this project was the Kaggle Dog vs. Cat classifier; the problem seemed very similar at the time to the dog breed classification task. There are numerous examples on the internet using pytorch or Keras tensorflow. The Kaggle Dog vs. Cat project classifier provided the following results without transfer learning:

- Accuracy of 80%;
- Training and testing loss of 0.2 and 0.2 respectively for a total of ~17 epochs;
- https://towardsdatascience.com/image-classifier-cats-vs-dogs-with-convolutional-neural-networkscnns-and-google-colabs-4e9af21ae7a8.

Although, this is an image recognition project, it is just binary classification and not as complex as Dog-Breed classification so I will not use it for my non – transfer learning model. Instead I’ll use the ‘Machine Learning Mastery’ project as a bench mark, ‘How to Develop a CNN From Scratch for CIFAR-10 Photo Classification’ written by Dr. Jason Browniee, May 13, 2019. The URL is: https://machinelearningmastery.com/how-to-develop-a-cnn-from-scratch-for-cifar-10-photo-classification/. I will use this analysis as the benchmark for my project where the accuracy of 67% - 73%. The number of classes are only ten, but the CIFAR-10 dataset is widely used for computer vision algorithms in machine learning.

For transfer learning I’ll use the image classifier that I built in the Udacity Nano degree, ‘Introduction to Machine Learning’. The Intro to ML Nano degree project classifier for flower category provided the following results using transfer learning across 100+ different flower types:

- Accuracy of 83%;
- Training and testing loss of 0.7 and 0.69

I will use this as the bench mark for comparison to this project's transfer learning exercise since the number classes are relatively the same and both projects use the pretrained VGG16 model.

Therefore, for transfer learning I will bench mark 80%+ and for non-transfer learning 67+% accuracy.

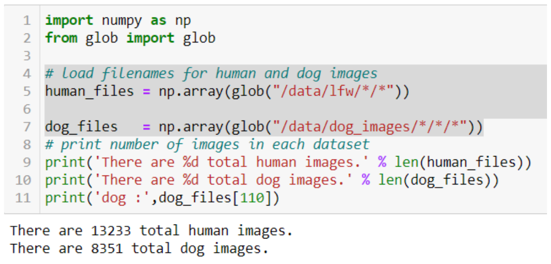
III. Methodology

3.1 Data Preprocessing

The processing steps are similar for the human and dog detectors, i.e., (1) Load and prepare the data; (2) specify the algorithm; (3) report the results. As for the CNN models it is to load and prepare the data.

3.1.1 Human Face Detector

Step-1: Loading the data an determine the number of images for each:

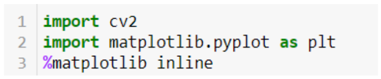


```
import numpy as np
from glob import glob

# 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))
print('dog :',dog_files[110])
```

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

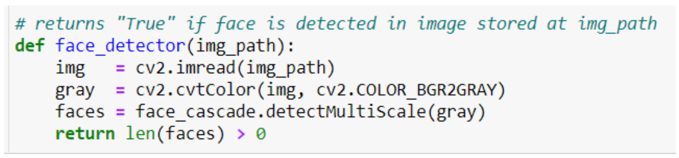
As above there are 13,233 human images and 8,351 dog images.

Step-2: Prepare the data:



```
import cv2
import matplotlib.pyplot as plt
%matplotlib inline
```

Step-3: specify the algorithm:
I will use the code provided by UDACITY as follow for testing the detector passing to it a path to an image:

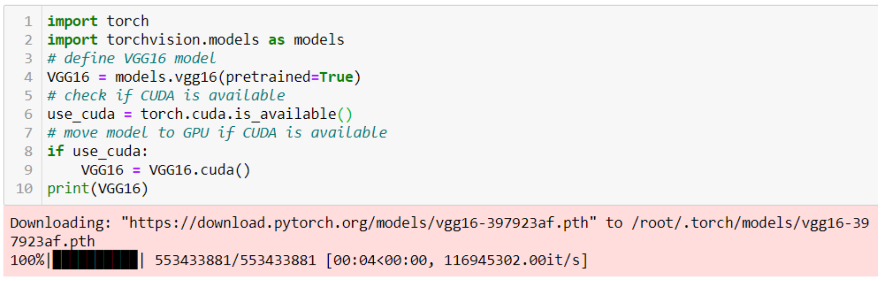


```
# returns "True" if face is detected in image stored at img_path
def face_detector(img_path):
    img   = cv2.imread(img_path)
    gray  = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    faces = face_cascade.detectMultiScale(gray)
    return len(faces) > 0
```

3.1.2 DOG Detector

Step-1: Loading the data an determine the number of images for each:

I loaded the pretrained VGG16 model and set the environment.

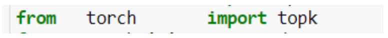


```
import torch
import torchvision.models as models
# define VGG16 model
VGG16 = models.vgg16(pretrained=True)
# check if CUDA is available
use_cuda = torch.cuda.is_available()
# move model to GPU if CUDA is available
if use_cuda:
    VGG16 = VGG16.cuda()
print(VGG16)
```

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg16-39
7923af.pth
100%|██████████| 553433881/553433881 [00:04<00:00, 116945302.00it/s]
```

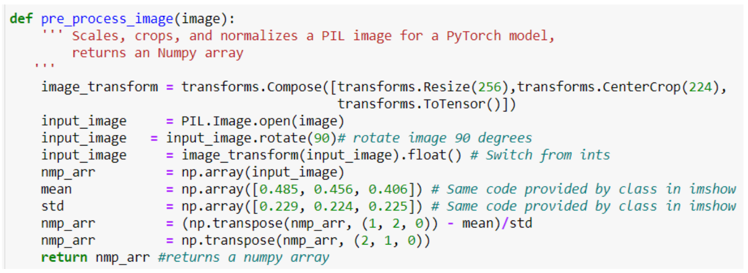
I set the _device = 'CPU'_ and the _model = VGG16._

Step-2: Prepare the data:
I imported this library below to have the VGG16 model return the top 'one' image predicted as well as the index to the corresponding image, this will be used in the VGG16_predict(image_path) procedure for dog detection.



```
from   torch         import topk
```

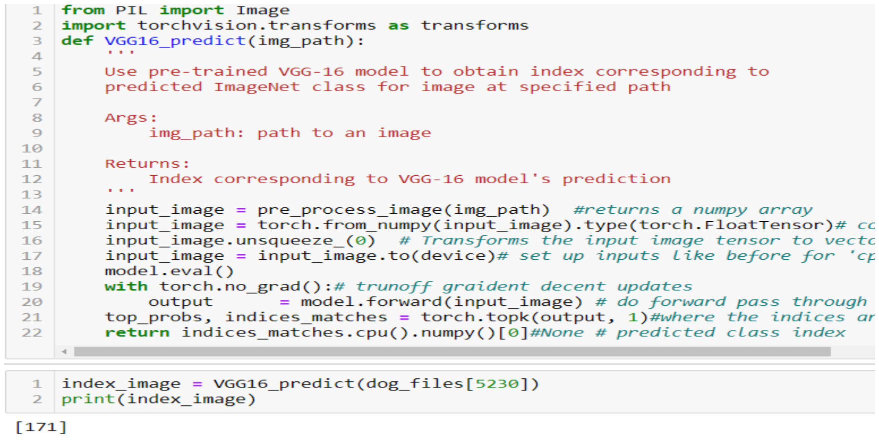
Step-3: specify the algorithm:
I wrote this code to preprocess the image which will be used in the VGG16_predict procedure.



```
def pre_process_image(image):
    ''' Scales, crops, and normalizes a PIL image for a PyTorch model,
        returns an Numpy array
   '''
    image_transform = transforms.Compose([transforms.Resize(256),transforms.CenterCrop(224),
                                          transforms.ToTensor()])
    input_image      = PIL.Image.open(image)
    input_image    = input_image.rotate(90)# rotate image 90 degrees
    input_image      = image_transform(input_image).float() # Switch from ints
    nmp_arr          = np.array(input_image)
    mean             = np.array([0.485, 0.456, 0.406]) # Same code provided by class in imshow
    std              = np.array([0.229, 0.224, 0.225]) # Same code provided by class in imshow
    nmp_arr          = (np.transpose(nmp_arr, (1, 2, 0)) - mean)/std
    nmp_arr          = np.transpose(nmp_arr, (2, 1, 0))
    return nmp_arr #returns a numpy array
```

This code builds a transform for the image, processes it as a PIL image translates it into a numpy array and set the mean and standard deviation of the image returning a numpy array so it can be converted into a pytorch tensor when called by VGG16_predict(image_path).

The following code snippet is the code used to predict dog images and will be called by the dog_detector.



```
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
    '''
    input_image = pre_process_image(img_path)  #returns a numpy array
    input_image = torch.from_numpy(input_image).type(torch.FloatTensor)# co
    input_image.unsqueeze_(0)  # Transforms the input image tensor to vecto
    input_image = input_image.to(device)# set up inputs like before for 'cp
    model.eval()
    with torch.no_grad():# trunoff graident decent updates
        output      = model.forward(input_image) # do forward pass through
    top_probs, indices_matches = torch.topk(output, 1)#where the indices ar
    return indices_matches.cpu().numpy()[0]#None # predicted class index
```

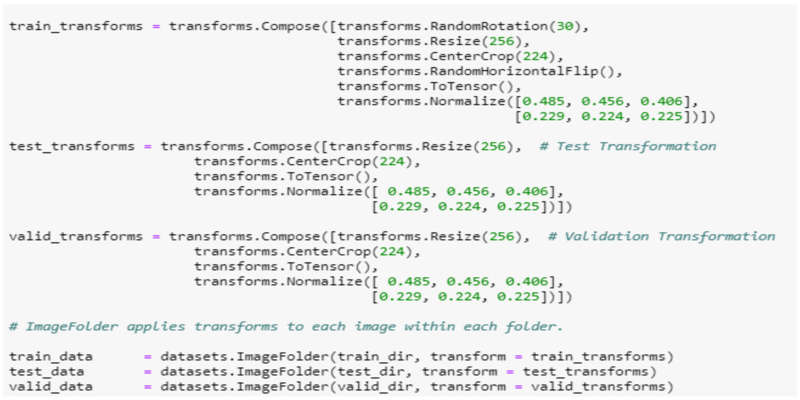
```
index_image = VGG16_predict(dog_files[5230])
print(index_image)
```

```
[171]
```

The code will be used by the dog_dector and it will receive an image path returning the predicted model's index.

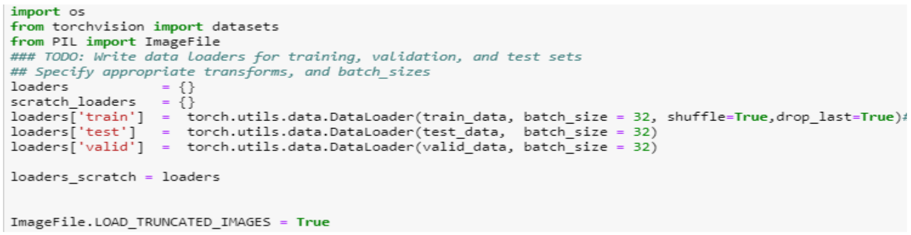
3.1.3 Create a CNN to Classify Dog Breeds (from Scratch)

To set up the Pytorch transforms for each data set determining the size of the image and center-crop was based on the information from data exploration findings. Training data images are resized to 256 pixels and center cropped to 244 pixels. To better train the model the transform is set to random rotation and horizontal flip. The mean is set to [0.485, 0.456, 0.406] and standard deviation set to [0.229, 0.224, 0.225] from the center, this is normalization; the images are also flipped and randomly rotated. Test and validation image transforms are resized to 256 pixels and center crop to 224 with normalization as the training; this is for consistency across all data sets to better control the operating environment and examine the results and performance of the model.



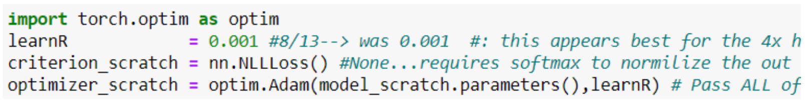
```
train_transforms = transforms.Compose([transforms.RandomRotation(30),
                                       transforms.Resize(256),
                                       transforms.CenterCrop(224),
                                       transforms.RandomHorizontalFlip(),
                                       transforms.ToTensor(),
                                       transforms.Normalize([0.485, 0.456, 0.406],
                                                            [0.229, 0.224, 0.225])])

test_transforms = transforms.Compose([transforms.Resize(256),  # Test Transformation
                  transforms.CenterCrop(224),
                  transforms.ToTensor(),
                  transforms.Normalize([ 0.485, 0.456, 0.406],
                                      [0.229, 0.224, 0.225])])

valid_transforms = transforms.Compose([transforms.Resize(256),  # Validation Transformation
                  transforms.CenterCrop(224),
                  transforms.ToTensor(),
                  transforms.Normalize([ 0.485, 0.456, 0.406],
                                      [0.229, 0.224, 0.225])])

# ImageFolder applies transforms to each image within each folder.

train_data      = datasets.ImageFolder(train_dir, transform = train_transforms)
test_data       = datasets.ImageFolder(test_dir, transform = test_transforms)
valid_data      = datasets.ImageFolder(valid_dir, transform = valid_transforms)
```

Data loaders we consistently set to a batch size of 32. For training data, shuffle and drop-last parameter are set to True for truncation errors. Th PIL ImageFile.LOAD_TRUNCATED_IMAGES to True to eliminate image load truncation errors of the last 150 bytes of some arbitrary images. The initial input data arrives with three channels for Red/Green/Blue and with the application of the with the applied transforms, the image size is consistently 224x224 square pixels. Data loader code specification follow:



```
import os
from torchvision import datasets
from PIL import ImageFile
### TODO: Write data loaders for training, validation, and test sets
## Specify appropriate transforms, and batch_sizes
loaders          = {}
scratch_loaders  = {}
loaders['train']  =  torch.utils.data.DataLoader(train_data, batch_size = 32, shuffle=True,drop_last=True)#
loaders['test']   =  torch.utils.data.DataLoader(test_data,  batch_size = 32)
loaders['valid']  =  torch.utils.data.DataLoader(valid_data, batch_size = 32)

loaders_scratch = loaders


ImageFile.LOAD_TRUNCATED_IMAGES = True
```

The hyperparameters set in preprocessing are learning rate, criterion, and the optimizer as indicated in the code. This code was also used by the transfer learning, but for the scratch model the learnR was set to 0.0001.



```
import torch.optim as optim
learnR            = 0.001 #8/13--> was 0.001  #: this appears best for the 4x h
criterion_scratch = nn.NLLLoss() #None...requires softmax to normilize the out
optimizer_scratch = optim.Adam(model_scratch.parameters(),learnR) # Pass ALL of
```

In the NET class definition, I set the dropout parameter equal to 20%.

3.1.4 Create a CNN to Classify Dog Breeds (using Transfer Learning)

For this section, I reused the scratch model's transforms, data loaders, dataset, criterion, and optimizer. The hyper parameters Dropout and learning rate are altered to 0.20 and 0.0001 respectively.

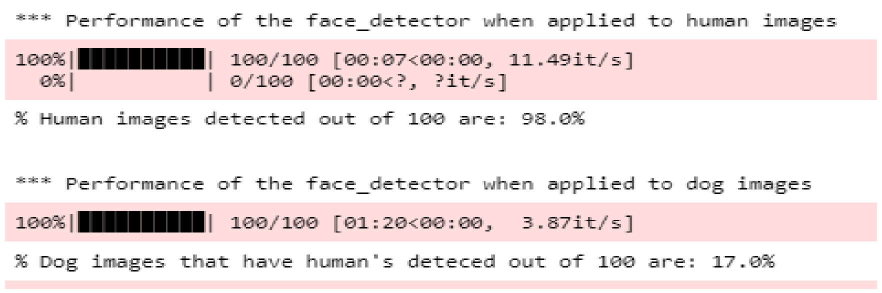
3.2 Implementation

3.2.1 *Assess the Human Face Detector*

Executing the human face detector resulted in the statistics as follows:

Human images detected out of 100 images are: 98%.
Dog images that have human-faces detected out of 100 images are: 17.%.
The results are very accurate for human faces, but when applied to dog-images there is room for improvement.

Testing the performance of the face detector algorithm on the images (roughly the first 100) of the human_files versus dog_files resulted in the output below; detecting faces was much quicker than that of dog detection.



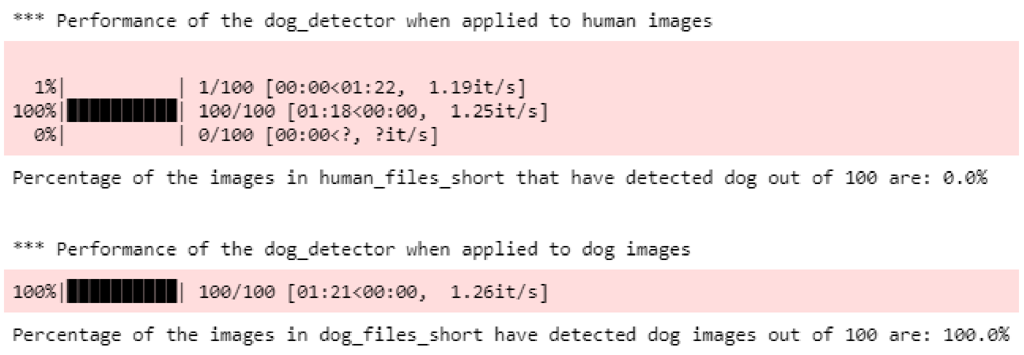
```
*** Performance of the face_detector when applied to human images
100%|██████████| 100/100 [00:07<00:00, 11.49it/s]
  0%|          | 0/100 [00:00<?, ?it/s]
% Human images detected out of 100 are: 98.0%

*** Performance of the face_detector when applied to dog images
100%|██████████| 100/100 [01:20<00:00,  3.87it/s]
% Dog images that have human's deteced out of 100 are: 17.0%
```

3.2.2 *Detect Dogs*

Using the previously built dog detector, I got the following results on dog images:

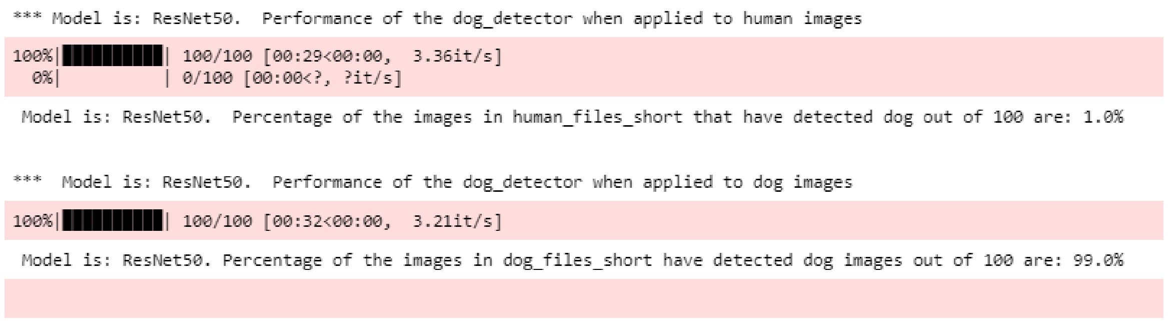
Dog images detected in human_files out of 100 images are: 0%.
Dog images detected in dog_files out of 100 images are: 100%.

Testing the performance of the dog detector algorithm on the images (roughly the first 100) of the human_files versus dog_files resulted in the output below. The performance is roughly the same.



```
*** Performance of the dog_detector when applied to human images

  1%|          | 1/100 [00:00<01:22,  1.19it/s]
100%|██████████| 100/100 [01:18<00:00,  1.25it/s]
  0%|          | 0/100 [00:00<?, ?it/s]
Percentage of the images in human_files_short that have detected dog out of 100 are: 0.0%

*** Performance of the dog_detector when applied to dog images
100%|██████████| 100/100 [01:21<00:00,  1.26it/s]
Percentage of the images in dog_files_short have detected dog images out of 100 are: 100.0%
```

Repeating this process for the dog_detector using the ResNet50 model. In short, the ResNet50 model performance was slower and for all intents and purposes just as accurate. See results below.



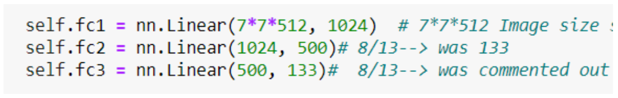
```
*** Model is: ResNet50.  Performance of the dog_detector when applied to human images
100%|██████████| 100/100 [00:29<00:00,  3.36it/s]
  0%|          | 0/100 [00:00<?, ?it/s]
 Model is: ResNet50.  Percentage of the images in human_files_short that have detected dog out of 100 are: 1.0%

***  Model is: ResNet50.  Performance of the dog_detector when applied to dog images
100%|██████████| 100/100 [00:32<00:00,  3.21it/s]
 Model is: ResNet50. Percentage of the images in dog_files_short have detected dog images out of 100 are: 99.0%
```

3.2.3 *Create a CNN to Classify Dog Breeds (from Scratch)*

In the **Algorithms and Techniques** section of this report I clearly discussed how I would build the scratch model architecture and the purpose behind the approach.   The steps that I took to build the scratch model follows:

1. Set criterion_scratch = nn.NLLLoss() #None...requires softmax to normalize the output between 0 and 1;
2. Set the learning rate to 0.001;
3. Define the NET Class for the scratch model
    a. Build the feature reduction and processing capability in the convolutional layers using this definition:
    *self.layer1 = nn.Sequential(nn.Conv2d(**in_filter**, **out_filter**, kernel_size=3, stride=1, padding=1),*
    nn.ReLU(),   # Activation function.
    nn.MaxPool2d(kernel_size=2, stride=2))# spatial variance used to recognize an
    b. *Kernel size will be '3' for all layers with a stride of one, i.e., move the window one square at a time;*
    c. *Padding for all layers around the kernel is one, i.e., add 1-row and column around the kernel window size;*
    d. *For all layers the activation function will be ReLu;*
    e. *For all layers the MaxPool2D function is specified above;*
    f. *Each layer will have additional characterization uniquely defined as follows:*
        i. Layer1: Characterize the initial input layer:
            1. Image in: 224x224 # Implied
            2. Image out: 112x122 # Implied
            3. In_filters   = 3 for Red/blue/Green images
            4. Out_filters= 32, there will be initial filters to train
        ii. Layer2:
            1. Image in: 112x112 # Implied
            2. Image out: 56x56   # Implied
            3. In_filters   = 32 output from the previous layer;
            4. Out_filters= 64, there will be initial filters to train;
        iii. Layer3:
            1. Image in: 56x56 # Implied
            2. Image out: 28x28 # Implied
            3. In_filters   = 64 output from the previous layer;
            4. Out_filters= 128, there will be initial filters to train;
        iv. Layer4:
            1. Image in: 28x28 # Implied
            2. Image out: 14x14 # Implied
            3. In_filters   = 128 output from the previous layer;
            4. Out_filters= 256, there will be initial filters to train;
        v. Laye5:

        1. Image in: 14x14 # Implied
        2. Image out: 7x7 # Implied
        3. In_filters   = 256 output from the previous layer;
        4. Out_filters= 512, there will be initial filters to train;
    g. Define the Dropout parameter for the class as 0.20
    h. Define the fully connected layers (FC)
       As for the FC layers, I arrived at the following where the final FC layer maps 500 to the 133 dog breed classes.



```
self.fc1 = nn.Linear(7*7*512, 1024)  # 7*7*512 Image size :
self.fc2 = nn.Linear(1024, 500)# 8/13--> was 133
self.fc3 = nn.Linear(500, 133)#  8/13--> was commented out
```

    i. Apply batch normalization to FC-1 and FC2
    j. Define the forward method that invokes the feature processing layer conv2D 1-through-5
    k. Flatten the output from the convolutional layers
    l. Invoke the FC layers
    m. Return the final out put using the softmax function because I used the criterion nn.NLLLoss().
4. Instantiate the NET class as *model_scratch = NET()*
5. Set *optimizer_scratch = optim.Adam(model_scratch.parameters(),learnR);*
6. Set the environment to Cuda GPUs.
7. Run 25 epochs of the training model using its test loss and accuracy calculations. Set Dropout = 0.2 and Learn rate = 0.001. Save the best performing model.
8. Metric applied for the implementation are Test Loss, Accuracy, and evaluation using a confusion matrix and the associated Specificity, Sensitivity, and examining the diagonal for true positives for all 133 dog breed classes.

To build a confusion matrix and the F1-score to measure accuracy of an imbalanced dataset, I will do the following:

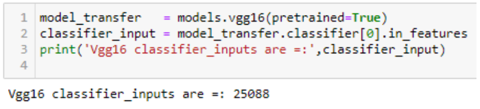
1. Import: from sklearn.metrics import confusion_matrix
   from sklearn. metrics import f1_score;
2. Write get_all_preds(model, loader) and get_num_correct(preds, labels) which is code from Deeplizard module on confusion matrix and it is modified for this project. This code will get all the predications and labels when run in mode = eval / torch.no_grad() and provide the correct number of correct predictions; *'train_preds, test_labels = get_all_preds(model_scratch, prediction_loaders)';*
3. Build a dataframe to examine the percent of dog-breed class predicted out of 133 classes;
4. Create and plot the confusion matrix using sklean: cm = confusion_matrix(test_labels,
   train_preds.argmax(dim=1));
5. Secure the sensitivity, specificity, the true-positive diagonal, and determine the accuracy.

3.2.4 Create a CNN to Classify Dog Breeds (using Transfer Learning)

This model will be used with the human-face and dog detectors. In the **Algorithms and Techniques** section of this report I clearly discussed how I would build the scratch model architecture and the purpose behind the approach. The goal is to build a specialized dog-breed classifier using a pretrained model. The steps that I took to build the transfer learning model follows:

1. Reuse the data loaders from the scratch model; *loaders_transfer = loaders;*
2. Set criterion_scratch = nn.NLLLoss() #None...requires softmax to normalize the output between 0 and 1;
3. Set the learning rate to 0.001;
4. Set dropout equal to 0.30;

5. Define the NET Class for the scratch model:
    a. Calculate the number of input-features the pretrained model has:

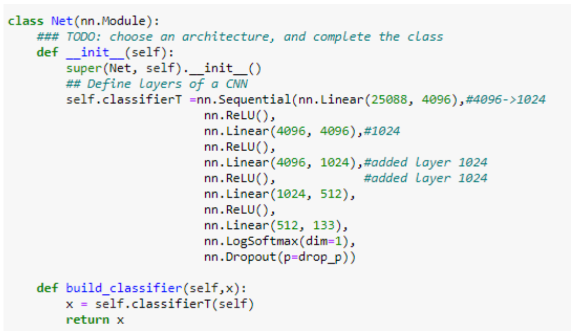


```
model_transfer   = models.vgg16(pretrained=True)
classifier_input = model_transfer.classifier[0].in_features
print('Vgg16 classifier_inputs are =:',classifier_input)

```

```
Vgg16 classifier_inputs are =: 25088
```

Use the 25,088 in_features as input to the specialize classifier

    b. Now it is just a matter of defining the FC layers.



```
class Net(nn.Module):
    ### TODO: choose an architecture, and complete the class
    def __init__(self):
        super(Net, self).__init__()
        ## Define layers of a CNN
        self.classifierT =nn.Sequential(nn.Linear(25088, 4096),#4096->1024
                                   nn.ReLU(),
                                   nn.Linear(4096, 4096),#1024
                                   nn.ReLU(),
                                   nn.Linear(4096, 1024),#added layer 1024
                                   nn.ReLU(),            #added layer 1024
                                   nn.Linear(1024, 512),
                                   nn.ReLU(),
                                   nn.Linear(512, 133),
                                   nn.LogSoftmax(dim=1),
                                   nn.Dropout(p=drop_p))

    def build_classifier(self,x):
        x = self.classifierT(self)
        return x
```

6. Instantiate the NET class as *model_scratch = NET()*
7. Access the sequential specifications: *model_scratch = model_scratch.classifierT*
    # This is necessary to access the nn.Sequential specifications
8. Set *optimizer_scratch = optim.Adam(model_transfer.parameters(),learnR);*
9. Set the environment to Cuda GPUs.
10. Freeze the only the pretrained model parameters using a procedure that I wrote:
    a. freeze_params(model_transfer)
11. Now set the model transfer classifier to the scratch_model classifier for training:
    a. model_transfer.classifier = model_scratch# model_scratch.classifierT
9. Run 25 epochs of the training model using its test loss and accuracy calculations. Save the best performing model.
10. Metric applied for the implementation are Test Loss, Accuracy, and evaluation using a confusion matrix and the associated Specificity, Sensitivity, and examining the diagonal for true positives for all 133 dog breed classes.

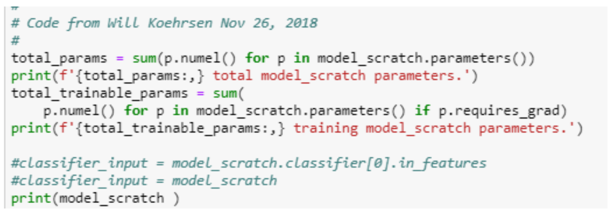
3.3 Refinement

3.3.1 Create a CNN to Classify Dog Breeds (from Scratch)

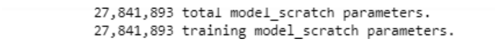
During the implementation process I experimented with multiple CNN layers followed by three-four FC (Fully-Connected) layers using various kernel sizes (starting with 5). It is here that I learned all about the management of the number of kernel sizes and its relationship to the number of in_feature calculations when creating the model. Noteworthy, is designing the number of filters that each CNN layer had. It is during the different sizing of each CNN-layer's input and output filters that I realized the idea that each convolution layer's filter are trained to search for different feature aspects in an image which is then used for classification and passed to the next layer; this is where I realized that I needed to start the processing with the smallest number of filters in layer one and increase them in each subsequent layer. In summary, I experimented with the following elements in an effort to arrive at an improved architecture: (1) Kernel Size; (2) Number of in and out filters per layer; (3) Number of convolution layers; (4) Max-Pooling was fixed because all the articles or internet examples that I studied show 2X2 is best for this application; (5) Stride, when set to one was best as the window shift parameter (which I learned also affected my in_feature size and the number of model parameter) and Padding around the kernel; (6) Setting the Transformations image size and crop from the center; and (7) learning rate. Of course, I used the dataset that was preprocessed as described above. I experimented with altering the training data transforms, but leant that it was best for now to use those specified in the scratch mode. The model described in the implementation section is the final solution model securing a 37% accuracy rate over the goal of 10%.

Adjusting the number of conv2D layers had an impact on the filter sizes as well as the image reduction and most important the performance accuracy of the model. I started at 12% accuracy with just three layers and worked by way up to 17% accuracy with a 3-layer conv2d followed by a 2-layer FC. The initial filters of the second layer started at 16, as per a mentor suggestion. When I adjusted the feature processing to five layers followed by three FC, I hit 37% accuracy.

I ran 25 epochs using Dropout = 0.2 and Learn rate = 0.001. I settled on criterion_scratch to nn.NLLLoss() because I wanted to use the softmax feature specifically, normalizing the output between specifically, 0 and 1. I did experiment with CrossEntropy as well. Lastly, from the code below I obtained the actual number of training parameter, and compared it to by manual calculation; I had a **~1.6% error** in my calculation.



```
#
# Code from Will Koehrsen Nov 26, 2018
#
total_params = sum(p.numel() for p in model_scratch.parameters())
print(f'{total_params:,} total model_scratch parameters.')
total_trainable_params = sum(
    p.numel() for p in model_scratch.parameters() if p.requires_grad)
print(f'{total_trainable_params:,} training model_scratch parameters.')

#classifier_input = model_scratch.classifier[0].in_features
#classifier_input = model_scratch
print(model_scratch )
```



```
27,841,893 total model_scratch parameters.
27,841,893 training model_scratch parameters.
```

3.3.2 Create a CNN to Classify Dog Breeds (using Transfer Learning)

In the **Algorithms and Techniques** section of this report I clearly discussed how I would build the transfer model architecture and the purpose behind the approach. During the implementation process I experimented with multiple FC layers using various in and out feature sizes. In summary, I experimented with the following elements in an effort to arrive at an improved architecture: (1) Number of in and out features for the classifier; (2) Dropout rates; (3) learning rates; (4) the number of epochs. Of course, I used the dataset that was preprocessed as described above. I experimented with altering the training data transforms again, but leant that it there was little to no improvement in performance than the specifications used in the scratch mode.

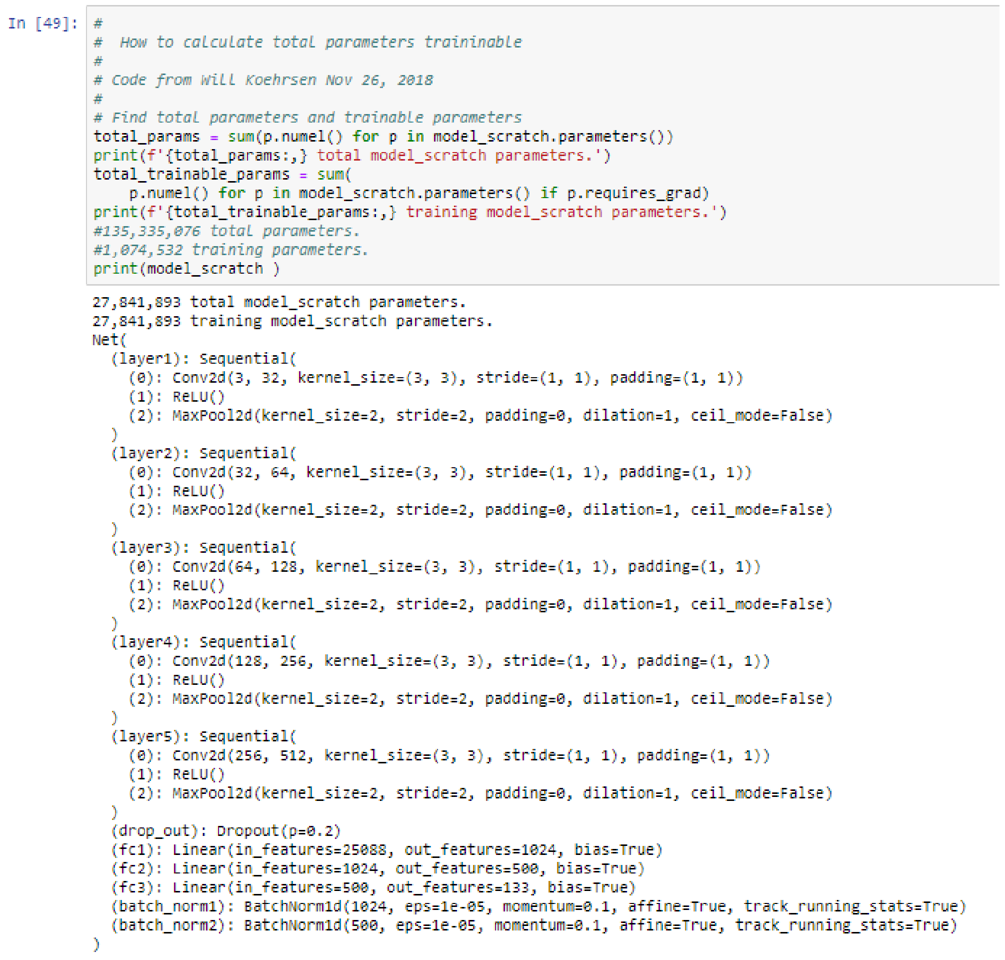
I ran 20-35 epochs using Dropout equal to range 0.25 to 0.5 and Learn rate in between 0.001 and 0.0001. After multiple runs, the number of epochs to produce a good performing solution was nine. I kept the criterion_transfer set to nn.NLLLoss() because I wanted to normalizing the output between 0 and 1 so that I could use this to distinguish between dog breed images and other animals in the project Algorithm. I used top_probs = torch.topk(output, 1) to capture the probabilities, then used top_probs.exp() to get the inverse since the criterion is NLLL. I eventually gravitated to a five-layer conv2D classifier, but starting at three and extending to five. As for determining the number of Fully Connected layers, I started with two and had poor results. I incremented these layers one at a time adjusting the learning rate and dropout hyper-parameters each time. This is when I recognized that three FC layers with five conv2D produced the best results. From there I focused on dropout holding learning rate constant and settled on a value of 25%. I presume that this value is best suited for the small training dataset size. This model NET Class definition achieved a 79% accuracy. Lastly, adjusting the conv2D number of filters proved imported. I had read that starting with a small number and increasing it in each layer worked best. I used a mentor suggestion to start with 16 by abandoned that replacing it with 32 and got better results. This exercise became a balancing act between the number of conv2D layers and the in and out filter size for layer; a lot of tuning!

IV. Results

4.1 Model Evaluation and Validation

4.1.1 Create a CNN to Classify Dog Breeds (from Scratch)

The final model architecture summary details follow:

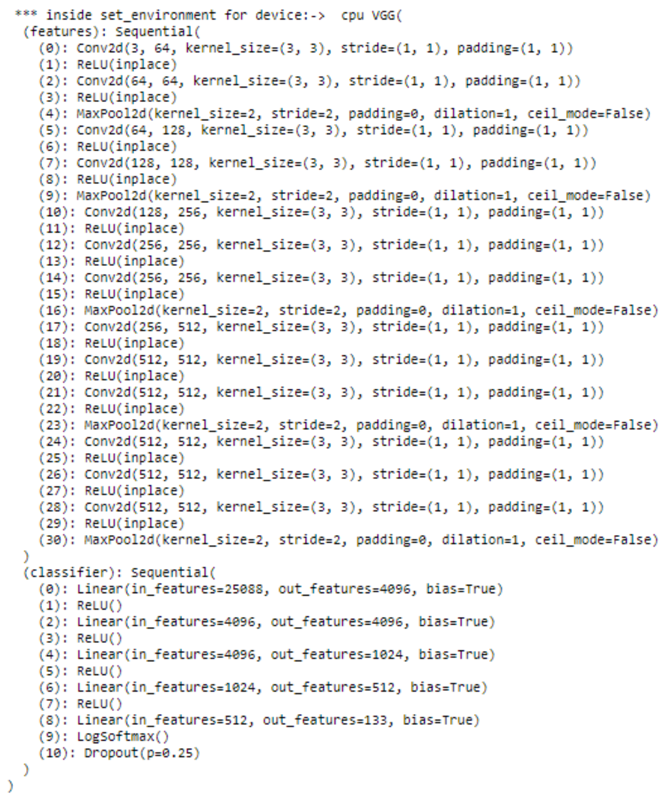


```
In [49]: #
         #  How to calculate total parameters trainable
         #
         # Code from Will Koehrsen Nov 26, 2018
         #
         # Find total parameters and trainable parameters
         total_params = sum(p.numel() for p in model_scratch.parameters())
         print(f'{total_params:,} total model_scratch parameters.')
         total_trainable_params = sum(
             p.numel() for p in model_scratch.parameters() if p.requires_grad)
         print(f'{total_trainable_params:,} training model_scratch parameters.')
         #135,335,076 total parameters.
         #1,074,532 training parameters.
         print(model_scratch )
```

```
27,841,893 total model_scratch parameters.
27,841,893 training model_scratch parameters.
Net(
  (layer1): Sequential(
    (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (layer2): Sequential(
    (0): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (layer3): Sequential(
    (0): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (layer4): Sequential(
    (0): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (layer5): Sequential(
    (0): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (drop_out): Dropout(p=0.2)
  (fc1): Linear(in_features=25088, out_features=1024, bias=True)
  (fc2): Linear(in_features=1024, out_features=500, bias=True)
  (fc3): Linear(in_features=500, out_features=133, bias=True)
  (batch_norm1): BatchNorm1d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (batch_norm2): BatchNorm1d(500, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
)
```

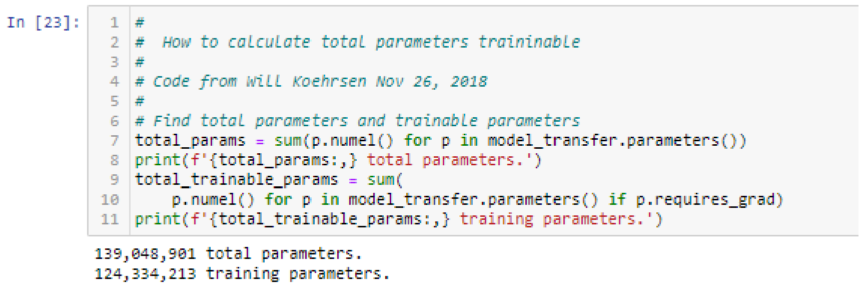
This model has been explained in detail throughout this report and it meets expectation. There are 27.8 million parameters to trained. Five conv2D layers followed by three fully connected layers where the last FC layer outputs the processing result to the 133 dog-breed classes. Noteworthy is how this model compare to the pretrained. The scratch model has to be explicit in how feature reduction is preformed increasing the number of filters from the first layer to layer five while simultaneously decreasing the image dimensions using MaxPooling 2x2. The model has been trained and tested with the UDACITY supplied training, test and validation data. This dataset is very small compared to the millions of images the top pretrained CNN models have used. This is a significant factor in achieving the accuracy of this scratch model. Therefore, given the limitation of the training and testing data, this model is simply not robust for predicting images from a new set of data.

4.1.2 Create a CNN to Classify Dog Breeds (using Transfer Learning)



```
*** inside set_environment for device:->  cpu VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU()
    (2): Linear(in_features=4096, out_features=4096, bias=True)
    (3): ReLU()
    (4): Linear(in_features=4096, out_features=1024, bias=True)
    (5): ReLU()
    (6): Linear(in_features=1024, out_features=512, bias=True)
    (7): ReLU()
    (8): Linear(in_features=512, out_features=133, bias=True)
    (9): LogSoftmax()
    (10): Dropout(p=0.25)
  )
)
```

The total number of training parameters are:



```
In [23]:
#
#  How to calculate total parameters traininable
#
# Code from Will Koehrsen Nov 26, 2018
#
# Find total parameters and trainable parameters
total_params = sum(p.numel() for p in model_transfer.parameters())
print(f'{total_params:,} total parameters.')
total_trainable_params = sum(
    p.numel() for p in model_transfer.parameters() if p.requires_grad)
print(f'{total_trainable_params:,} training parameters.')
```

```
139,048,901 total parameters.
124,334,213 training parameters.
```

This model builds a specific dog-breed classifier with 124.3 million training parameters. There are five fully connected layers where the final layer is 512 to 133 dog-breed classes. All the feature reduction and processing are handled by the VGG16 pretrained model to which there are 13 conv2d layers, significantly more than the scratch model. During processing the pretrained VGG16 model had its parameters ‘frozen’, i.e., these were not re-trained only the new classifier was trained. The new classifier model has been trained and tested with the UDACITY supplied training, test and validation data. Although, the dataset is very small compared to the millions of images the top pretrained CNN models have used, it is sufficient to get pretty good results in terms of accuracy for the new classifier. Therefore, this model is somewhat robust for predicting images from a new set of data. The accuracy is 79% and when tested with images used outside of those provided it worked well every time making accurate predictions. I believe that the results from this classifier can be trusted.

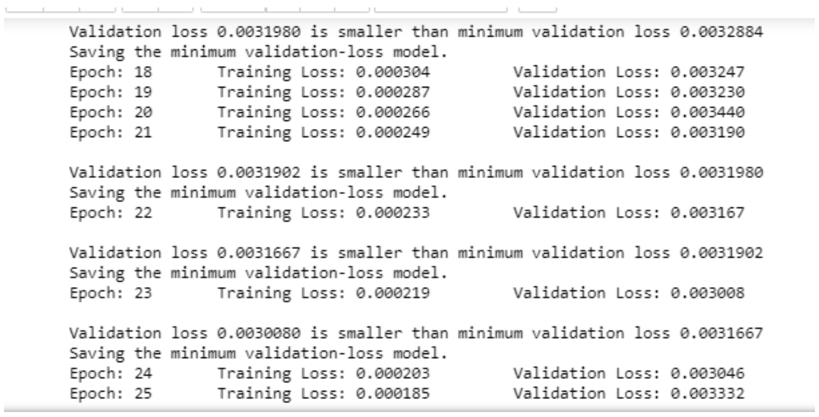
4.1.3 Write Your Algorithm

The transfer model was tested within the Algorithm. Part of the Algorithm procedures included the human-face and dog detectors. The Algorithm performed well using the transfer model correctly predicting all of the images submitted. The CNN transfer classifier proved to be robust enough to solve the project problem. I had to rewrite the dog detector from the initial one because the VGG16 pretrained model and ImageNet indexes where used. The new code adds a conditional statement to check the returned prediction probability to make sure only dog breeds for dogs was returned and uses the transfer model classifier. I replaced the vgg16_predict with predict_breed_transfer(image_path). This new procedure uses the transfer model and its indices and returns the probability of a match and the index of the transfer model.

4.2 Justification

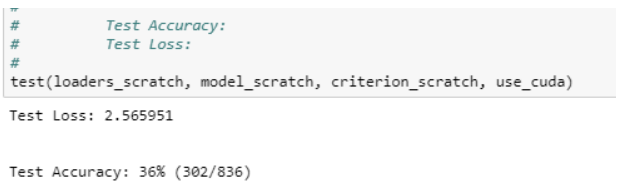
4.2.1 Create a CNN to Classify Dog Breeds (from Scratch)

The bench mark model that I chose for scratch model is from the 'Machine Learning Mastery' project as a bench mark, 'How to Develop a CNN From Scratch for CIFAR-10 Photo Classification' written by Dr. Jason Browniee, May 13, 2019. Browniee achieved an accuracy of 67% initially and through modification reached 73%. My final model achieved an accuracy of 36%. Epochs 22 through 23 presented the best results where the curve of the validation loss starts to move in the opposite direction indicating issues with the architecture.



```
Validation loss 0.0031980 is smaller than minimum validation loss 0.0032884
Saving the minimum validation-loss model.
Epoch: 18       Training Loss: 0.000304         Validation Loss: 0.003247
Epoch: 19       Training Loss: 0.000287         Validation Loss: 0.003230
Epoch: 20       Training Loss: 0.000266         Validation Loss: 0.003440
Epoch: 21       Training Loss: 0.000249         Validation Loss: 0.003190

Validation loss 0.0031902 is smaller than minimum validation loss 0.0031980
Saving the minimum validation-loss model.
Epoch: 22       Training Loss: 0.000233         Validation Loss: 0.003167

Validation loss 0.0031667 is smaller than minimum validation loss 0.0031902
Saving the minimum validation-loss model.
Epoch: 23       Training Loss: 0.000219         Validation Loss: 0.003008

Validation loss 0.0030080 is smaller than minimum validation loss 0.0031667
Saving the minimum validation-loss model.
Epoch: 24       Training Loss: 0.000203         Validation Loss: 0.003046
Epoch: 25       Training Loss: 0.000185         Validation Loss: 0.003332
```

The training algorithm saved the best trained-model which was used to submit to the test procedure resulting in Test-Loss of 2.566 and accuracy of 36%. The test-code is offered by the UDACITY Dog-Breed workspace.

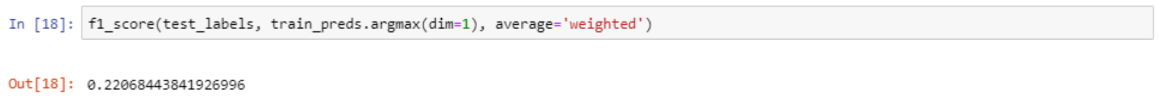


```
#         Test Accuracy:
#         Test Loss:
#
test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)

Test Loss: 2.565951


Test Accuracy: 36% (302/836)
```

Computing the F1 score, known as balanced F-score or F-measure, is interpreted as a weighted average of the precision and recall, where an F1 score reaches its best value at **1** and worst score at **0.** The relative contribution of precision and recall to the F1 score are equal. Note: text is straight out of the library definition. The formula for the F1 score is:

F1 = 2 * (precision * recall) / (precision + recall)

Since this is a multi-class / multi-label analysis, I set the average parameter to the weighted which calculates metrics for each label, and finds their average number of true instances for each label. For label imbalance; it can result in an F-score that is not between precision and recall.



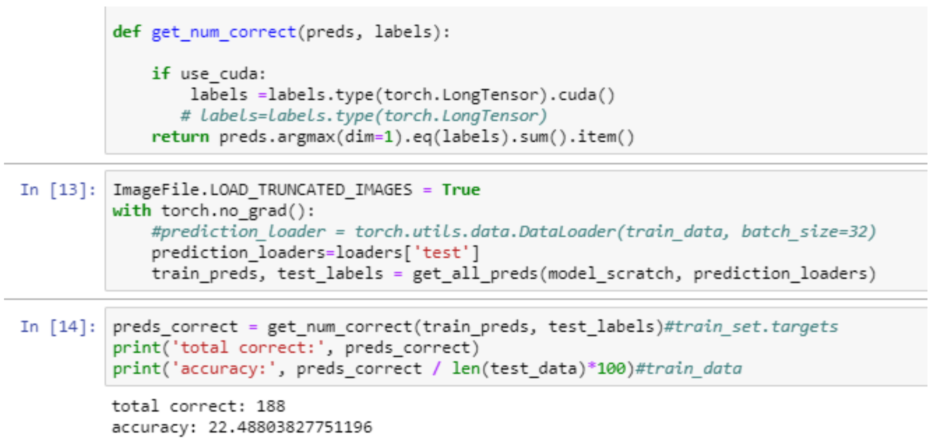
```
In [18]: f1_score(test_labels, train_preds.argmax(dim=1), average='weighted')

Out[18]: 0.22068443841926996
```

I received a value of 0.22 / 22% which is very close to zero.

Code from the website 'Deeplizard' was leveraged below and it is from their training module on confusion matrixes where I modified it for this project and recalculated the accuracy as follows:

1. Loaded the model from model_scratch.pt
2. Set the environment to torch.no_grad() for Forward execution only
3. Set the data loader to 'Test' data set
4. Invoked get_all_preds(model, loader) that returns all the predictions and labels
5. Invoked get_num_correct predictions that returns all correct predictions

Out of 836 test images 188 were correct yielding a 22.5% Accuracy which is closer to the F1-score than the UDACITY provided 'test' algorithm.

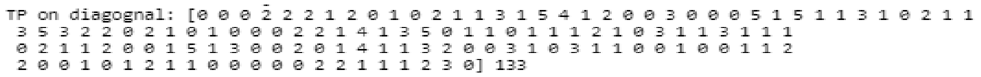


```
def get_num_correct(preds, labels):

    if use_cuda:
        labels =labels.type(torch.LongTensor).cuda()
       # labels=labels.type(torch.LongTensor)
    return preds.argmax(dim=1).eq(labels).sum().item()
```

```
In [13]: ImageFile.LOAD_TRUNCATED_IMAGES = True
         with torch.no_grad():
             #prediction_loader = torch.utils.data.DataLoader(train_data, batch_size=32)
             prediction_loaders=loaders['test']
             train_preds, test_labels = get_all_preds(model_scratch, prediction_loaders)
```

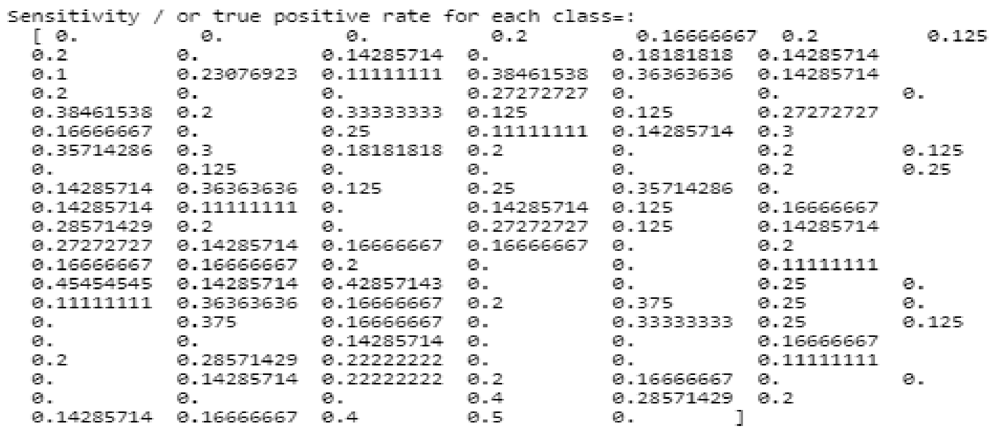
```
In [14]: preds_correct = get_num_correct(train_preds, test_labels)#train_set.targets
         print('total correct:', preds_correct)
         print('accuracy:', preds_correct / len(test_data)*100)#train_data

         total correct: 188
         accuracy: 22.488038277511961
```

The confusion matrix analysis proved to illustrate the scratch model weakness as observed by the True-Positive (TP) diagonal and sensitivity. Looking at all the classes in the TP diagonal array below one can discern that the occurrence per class of the correct predictions and it is very low [read array left to right for each dog-breed]. In the conclusion section of this report expands on this result.



```
TP on diagognal: [0 0 0 2 2 2 1 2 0 1 0 2 1 1 3 1 5 4 1 2 0 0 3 0 0 0 5 1 5 1 1 3 1 0 2 1 1
 3 5 3 2 2 0 2 1 0 1 0 0 0 2 2 1 4 1 3 5 0 1 1 0 1 1 1 2 1 0 3 1 1 3 1 1 1
 0 2 1 1 2 0 0 1 5 1 3 0 0 2 0 1 4 1 1 3 2 0 0 3 1 0 3 1 1 0 0 1 0 0 1 1 2
 2 0 0 1 0 1 2 1 1 0 0 0 0 0 2 2 1 1 1 2 3 0] 133
```

Examining the corresponding sensitivity illustrates the TP rate for each class and it has very poor performance.



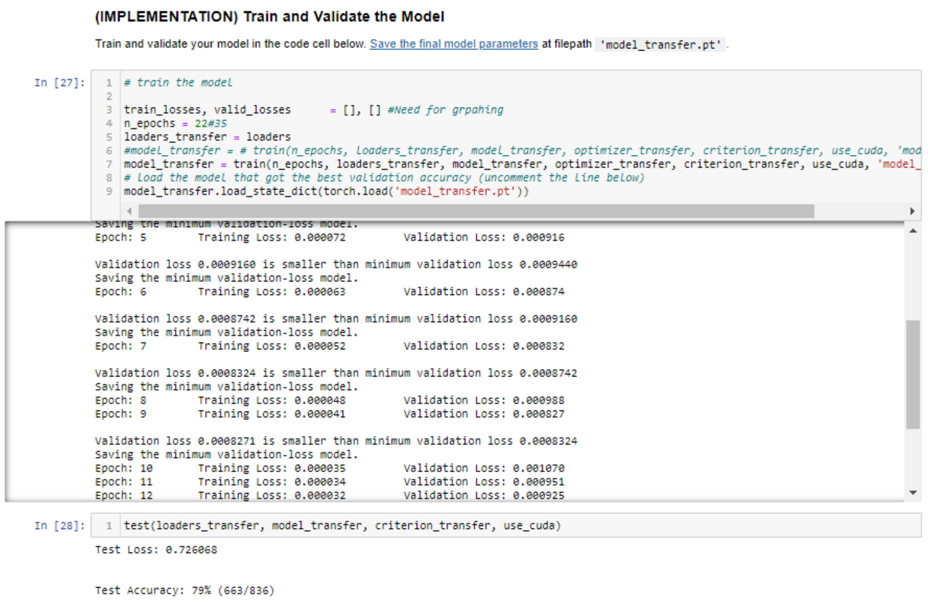
```
Sensitivity / or true positive rate for each class=:
 [ 0.          0.          0.          0.2         0.16666667  0.2         0.125
  0.2         0.          0.14285714  0.          0.18181818  0.14285714
  0.1         0.23076923  0.11111111  0.38461538  0.36363636  0.14285714
  0.2         0.          0.          0.27272727  0.          0.          0.
  0.38461538  0.2         0.33333333  0.125       0.125       0.27272727
  0.16666667  0.          0.25        0.11111111  0.14285714  0.3
  0.35714286  0.3         0.18181818  0.2         0.          0.2         0.125
  0.          0.125       0.          0.          0.          0.2         0.25
  0.14285714  0.36363636  0.125       0.25        0.35714286  0.
  0.14285714  0.11111111  0.          0.14285714  0.125       0.16666667
  0.28571429  0.2         0.          0.27272727  0.125       0.14285714
  0.27272727  0.14285714  0.16666667  0.16666667  0.          0.2
  0.16666667  0.16666667  0.2         0.          0.          0.11111111
  0.45454545  0.14285714  0.42857143  0.          0.          0.25        0.
  0.11111111  0.36363636  0.16666667  0.2         0.375       0.25        0.
  0.          0.375       0.16666667  0.          0.33333333  0.25        0.125
  0.          0.          0.14285714  0.          0.          0.16666667
  0.2         0.28571429  0.22222222  0.          0.          0.11111111
  0.          0.14285714  0.22222222  0.2         0.16666667  0.          0.
  0.          0.          0.          0.4         0.28571429  0.2
  0.14285714  0.16666667  0.4         0.5         0.        ]
```

4.2.2 *Create a CNN to Classify Dog Breeds (using Transfer Learning)*

For transfer learning I'll use the image classifier that I built in the Udacity Nano degree, 'Introduction to Machine Learning'. This project used the same VGG16 pretrained model for a transfer learning exercise. The Intro to ML Nano degree project classifier for flower category provided the following results using transfer learning across 106+ different flower types:

- Accuracy of 83%.

The following screen shot from the notebook captures the best model secured at epoch number nine.
I am positive that running more epochs will not increase accuracy, due to overfitting.



(IMPLEMENTATION) Train and Validate the Model

Train and validate your model in the code cell below. Save the final model parameters at filepath `'model_transfer.pt'`.

```
In [27]:
# train the model

train_losses, valid_losses      = [], [] #Need for grpahing
n_epochs = 22#35
loaders_transfer = loaders
#model_transfer = # train(n_epochs, loaders_transfer, model_transfer, optimizer_transfer, criterion_transfer, use_cuda, 'mod
model_transfer = train(n_epochs, loaders_transfer, model_transfer, optimizer_transfer, criterion_transfer, use_cuda, 'model_
# load the model that got the best validation accuracy (uncomment the line below)
model_transfer.load_state_dict(torch.load('model_transfer.pt'))
```

```
Saving the minimum validation-loss model.
Epoch: 5        Training Loss: 0.000072         Validation Loss: 0.000916

Validation loss 0.0009160 is smaller than minimum validation loss 0.0009440
Saving the minimum validation-loss model.
Epoch: 6        Training Loss: 0.000063         Validation Loss: 0.000874

Validation loss 0.0008742 is smaller than minimum validation loss 0.0009160
Saving the minimum validation-loss model.
Epoch: 7        Training Loss: 0.000052         Validation Loss: 0.000832

Validation loss 0.0008324 is smaller than minimum validation loss 0.0008742
Saving the minimum validation-loss model.
Epoch: 8        Training Loss: 0.000048         Validation Loss: 0.000988
Epoch: 9        Training Loss: 0.000041         Validation Loss: 0.000827

Validation loss 0.0008271 is smaller than minimum validation loss 0.0008324
Saving the minimum validation-loss model.
Epoch: 10       Training Loss: 0.000035         Validation Loss: 0.001070
Epoch: 11       Training Loss: 0.000034         Validation Loss: 0.000951
Epoch: 12       Training Loss: 0.000032         Validation Loss: 0.000925
```

```
In [28]:
test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.726068


Test Accuracy: 79% (663/836)
```

I achieve an accuracy of 79% with a Test Loss of 0.72, not bad. I had hoped to realize accuracy in the 80+ percentile. I tried many iterations adjusting the number of FC layers and the in/out feature count per layer as well as the Dropout rate. Dropout rate had a considerable impact on the performance, due to the small amount of data. The best results for the transfer learning was a Dropout rate of 0.25 and a five-layer classifier. When changing Dropout rate between 25% to 45% made a huge difference in acceptable performance as related to the benchmark, this is due to the small about of data used for training cutting out 25-50 percent has impact on the classifier's quality. 79% accuracy is sufficient to have solved this problem using transfer learning and it is substantially better than building a scratch model. The optimal model presented it self at epoch ten where the validation loss dropped to 0.000827.

4.2.3 Write Your Algorithm

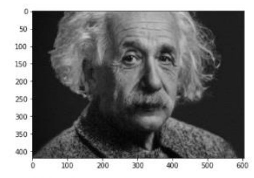
To write my own algorithm I defined the ‘run_app(image_path)’ procedure. This code did not reuse the previously written ‘dog_detector()’, only the ‘face_detector()’ procedure. I rewrote the dog detector and wrote predict_breed_transfer (image_path) code as described above. I used the ‘best’ saved model of the transfer classifier. I added a couple of new procedures to save and load the model making it run in CPU mode versus GPU, i.e., ‘save_model(model, sav_path)’ and ‘load_checkpoint(filename)’.   The load_checkpoint() procedure returns the transfer classifier model and maps the GPU saved model to local CPU storage. To set the environment up for the ‘run_app()’ I ran this command:

‘model = load_checkpoint('model_transferBEST.pt')’.

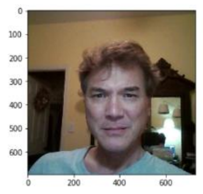
Class_names was set as indexes captured from the training data set, i.e., train_data.classes so that I could retrieve the dog breed names.

4.2.3 Test Your Algorithm

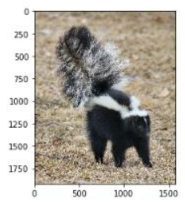
To test my algorithm, I provide images of animals and human faces, the model worked very well. As shown below.



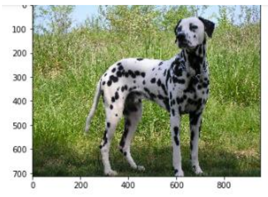
```
A human image has been dectected
This human resembles a dog-breed and would belong to class : Pharaoh hound
```



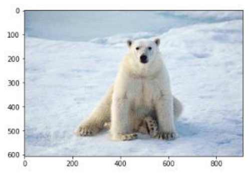
```
A human image has been dectected
This human resembles a dog-breed and would belong to class : Xoloitzcuintli
```



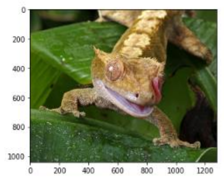
```
**** ERROR *** invalid image submitted:
```



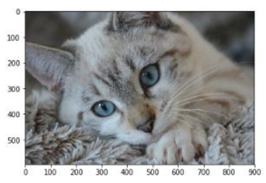
```
A dog image has been dectected
Predicting the breed as belonging to class : Dalmatian
```



```
**** ERROR *** invalid image submitted:
```



```
**** ERROR *** invalid image submitted:
```



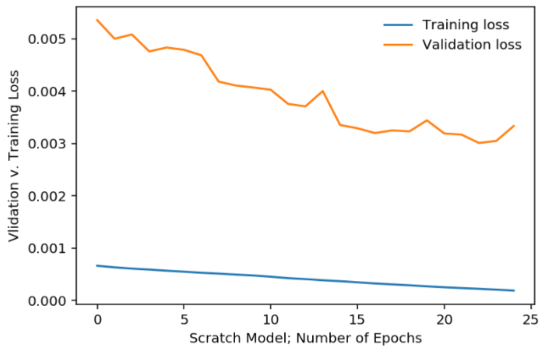
```
**** ERROR *** invalid image submitted:
```

V. Conclusion

5.1 Free-Form Visualization

5.1.1 Create a CNN to Classify Dog Breeds (from Scratch)

The scratch model is evaluated using accuracy which is the ratio of number of correct predictions to the total number of images processed. The UDACITY Test Code was used to assess this model and it reached 36%. However, Test accuracy versus Training accuracy was also used and plotted. The goal would be to have the test loss and validation loss converge and improve together. These values converge as the number of training epochs increases. I experimented with the number of epochs ranging from 15 to 40 in an attempt to identify the optimal number of epochs that produced the best results. The graph below illustrates the training v. validation loss. It took 24 epochs to reach maximum performance for this model. So, epochs 24 is where the curves, training and test loss start to diverge into opposite directions indicating issues with the model architecture.

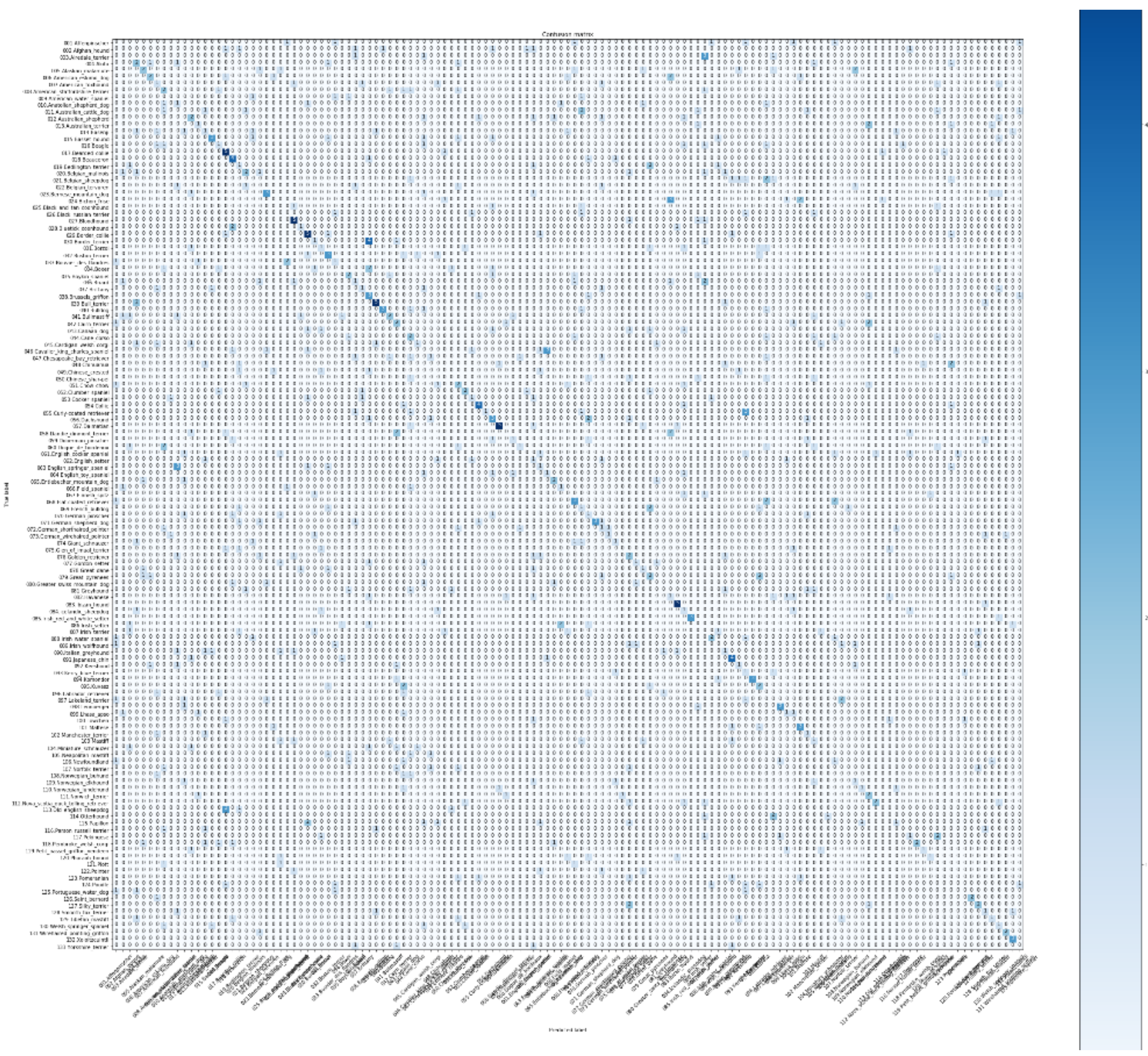


Below is the confusion matrix graphic for a 133 X 133 class representation with all the class names; this is built using the test dataset which is run against the scratch model. The size of the number of classes makes reading the chart difficult. The matrix measures the effectiveness or performance of a classification model. The better the effectiveness, the better the performance; this is what needs to be ascertained. The Y-Axis are the true-labels and the X-Axis are the predicted-labels. True labels are in rows and the columns contain predictions. The diagonal contains the correct predictions; note the True-Positive (TP) diagonal. This graph was built using the test data set where there are 836 total images. It is difficult to read the matrix graph an array of the Ture Positives on the diagonal array below better communicates the performance of correct predictions. Of course, the array represents the dog breeds from zero to 133, left to right. Very poor prediction success. When comparing the True-Positive (TP) diagonal with the array of True-Negatives (TN), the TNs prediction accuracy rate is high.

```
TP on diagognal: [0 0 0 2 2 2 1 2 0 1 0 2 1 1 3 1 5 4 1 2 0 0 3 0 0 0 5 1 5 1 1 3 1 0 2 1 1
 3 5 3 2 2 0 2 1 0 1 0 0 0 2 2 1 4 1 3 5 0 1 1 0 1 1 1 2 1 0 3 1 1 3 1 1 1
 0 2 1 1 2 0 0 1 5 1 3 0 0 2 0 1 4 1 1 3 2 0 0 3 1 0 3 1 1 0 0 1 0 0 1 1 2
 2 0 0 1 0 1 2 1 1 0 0 0 0 0 2 2 1 1 1 2 3 0] 133
TN : [819 823 824 815 818 821 819 813 830 818 820 816 825 818 817 821 808 813
 820 820 823 824 819 825 824 827 811 828 808 823 822 814 823 825 821 819
 825 810 813 822 819 816 820 813 825 815 824 823 827 827 815 823 825 817
 827 812 815 826 826 824 826 816 820 825 824 827 827 815 818 819 822 819
 828 826 830 812 827 824 816 828 827 818 816 819 826 823 814 823 825 821
 815 825 819 825 819 819 823 823 825 826 820 826 819 828 831 822 825 832
 824 829 820 824 829 829 822 831 824 825 829 828 825 831 822 829 830 829
 824 825 825 822 828 824 828] 133
```

This leads us to examine precision; precision = (TP) / (TP+FP). FP is the number of false positives. Out of all the 133 classes, how many were predicted positively? The value of each class prediction in terms of percentages follows in this 133-element array derived from the confusion analysis. For example, element number six in the array is an Alaskan Malamute and has a Positive Predictive value of 28.57% while element 15 is an American Eskom Dog with a 33.33% precision hit rate. Recall the F1 score value of 22% this array of classes reflects this score. The model is not very precise.

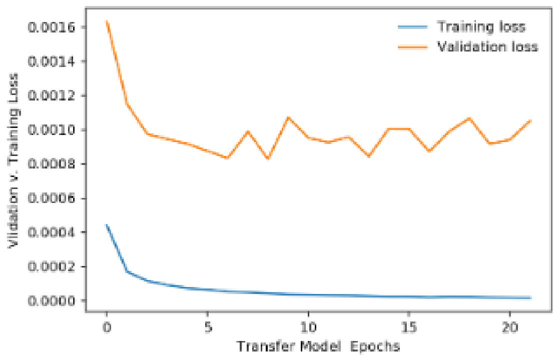
```
Percision / positive predictive value for each class=:
   [ 0.         0.         0.         0.15384615  0.25        0.28571429
  0.1         0.13333333  0.          0.08333333  0.          0.18181818
  0.2         0.11111111  0.33333333  0.14285714  0.25        0.25        0.1
  0.25        0.          0.          0.33333333  0.          0.          0.
  0.29411765  0.25        0.27777778  0.16666667  0.14285714  0.21428571
  0.125       0.          0.22222222  0.11111111  0.2         0.15789474
  0.35714286  0.42857143  0.25        0.16666667  0.          0.13333333
  0.25        0.          0.2         0.          0.          0.
  0.15384615  0.28571429  0.2         0.33333333  0.5         0.2
  0.41666667  0.          0.25        0.25        0.          0.07142857
  0.11111111  0.16666667  0.28571429  0.2         0.          0.23076923
  0.09090909  0.09090909  0.5         0.09090909  0.33333333  0.2         0.
  0.125       0.25        0.14285714  0.16666667  0.          0.          0.1
  0.35714286  0.09090909  0.5         0.          0.          0.28571429
  0.          0.14285714  0.28571429  0.16666667  0.07692308  0.5
  0.18181818  0.          0.          0.375       0.16666667  0.          0.3
  0.14285714  0.1         0.          0.          0.125       0.          0.
  0.14285714  0.33333333  0.18181818  0.4         0.          0.
  0.16666667  0.          0.16666667  0.5         0.33333333  0.33333333
  0.          0.          0.          0.          0.          0.5
  0.28571429  0.14285714  0.2         0.11111111  0.4         0.33333333
  0.        ]
```



Although difficult to read the graph, the Ture Positives on the diagonal array illustrates the lack of precision.

5.1.2 Create a CNN to Classify Dog Breeds (using Transfer Learning)

The graph below illustrates the Validation versus Training loss. I researched the characteristics of this graph and all the articles on the internet indicated that I have a typical overfitting problem with the model. I need to apply more L1 or L2 regularization techniques. Again, adjusted Dropout several times along with the learning rate to reach this final solution accuracy to 79%. Noteworthy is how sensitive this transfer model accuracy is to slight hyper-parameter adjustments.



At epoch number 9 as shown in the graph is approximately where the maximum model performance is achieved.

5.2 Reflection

In summary the problem statement for this capstone project is to produce an Algorithm that can correctly predict input images as human-faces or specific dog-breeds. Should an image be a human-face specify which dog-breed is predicted for that human face. Otherwise if the image input is neither a dog nor human communicate the that the image is neither as an error. To solved the problem this program required that a human-face and dog breed defector be built, tested and, measure the accuracy and the performance time for each detector execution. UDACITY provided the face-detector and the image library for testing. The project required that a dog-breed detector be written using a pretrained CNN image classifier architecture, i.e., VGG16. The dog detector dataset was provided in the UDACITY workspace. Testing the dog detector was validated against the public ImageNet index scheme. Ultimately, the Algorithm is to be built using both defectors and a dog-breed specific classifier that is written using a choice of public pretrained image classifiers.

The challenging part of the capstone project was: (1) build from scratch a CNN architecture;(2) picking the best pretrained model to write a transfer-learning classifier. I am convinced that I should have picked ResNet50 over the VGG16 model because it has a many more hidden layers than the VGG16 pretrained model and its accuracy is reported as higher.

The process that I followed to build the final solution uses the UDACY notebook specification as outlined in the problem statement. Following is the outline of that process, but augmented to emphasize key steps.
The UDACITY notebook has outlined six executable steps these steps serve as tasks to materialize the project:

1. Import Datasets:
   a. Import the human-faces images using OpenCV's Haar feature-based cascade classifiers;
   b. Load the images into numpy arrays to be used by the next three steps.
2. Detect Humans:
   a. Use the UDACITY supplied human-face detector code; Test the code against the dog and human-face images. Test the performance using VGG16.
3. Detect Dogs:
   a. Download the pretrained image classifier VGG16,
   b. Build an image display and a predictor procedure using VGG16 that returns an index into ImageNet,

    c. Develop a dog-detector procedure, Test the code against the dog and human-face images,
    d. Test the performance using VGG16 and ResNet50; outcome was roughly the same.
4. Create a CNN to Classify Dog Breeds (from Scratch)
    a. Conduct data exploration and visualization, preprocess the data,
        i. Examine the statistic using histograms, skewness to ascertain if the data set is balanced or not;
        ii. Examine the image pixel sizes, obtain: medium, mean, standard deviation and use as input for building the transforms;
        iii. Build a scatter plot of the training dataset for dog images. Determine the distribution of dog breeds frequency of occurrence;
    b. Build the data transforms; set resize, and crop from center consistently for the entire dataset. Set training transforms to flip and rotate;
    c. Build data loader; set consistent batch size. For the training loader set to shuffle and drop-last=True;
    d. Define and build the CNN model architecture:
        i. Define the conv2D layers, kernel size, stride, number of feature filters, calculate the input size;
        ii. Calculate the in & out features per layer;
        iii. Apply MaxPooling for image reduction and the ReLu activation function;
        iv. Calculate the number of trainable parameters per layer and the image size for each layer;
        v. Determine dropout and which criterion to use;
        vi. Design the optimal number of fully connected layers with all associated in and out features.
    e. Train the model and save the best model with the highest accuracy, adjust hyper-parameters and conv2 number of layers; experiment with multiple epochs.
    f. Use the UDACITY Test procedure and achieve an accuracy >= 10%.
5. Create a CNN to Classify Dog Breeds (using Transfer Learning):
    a. Select a pretrain image classifier as the transfer model, use VGG16;
    b. Define the dog -breed specific classifier class structure, i.e., number of FC layers;
    c. Download the VGG16 pretrained model and freeze the model parameters;
    d. Instantiate the NET class and assign the classifier object to the pretrained model;
    e. Train the model saving the best one with the highest accuracy. Make sure that the stat_dic and trained classifier are saved for the best performing model,
    f. Use the UDACITY Test procedure and achieve an accuracy >= 60%.
6. Write Your Algorithm:
    a. Write the algorithm using the transfer learning dog-breed classifier,
    the human-face and dog detectors, and image display,
    b. Write a new dog_detector procedure that considers the probability the prediction is a dog breed.
    c. Replace vgg16_predictor with a procedure that uses the transfer model classifier, returns its indexes and the inverse of the NLLL results so that a proper statistic can be rendered.
    d. Test against sample images.
7. Test Your Algorithm
    a. Test the algorithm against various images of human-faces, other animals,
    dog images.

The most interesting aspect of the project was also the most challenging, i.e., to build from scratch a CNN model. This is where I learned the most. I learned how to calculate the number of trainable parameters per conv2 layer, why choosing a conv2D over a 1D or 3D is needed. Experimenting Kernel size and stride to arrive at the optimal convolution window from layer to layer was enlightening. The learnt the application of the max_pooling function is for image/feature reduction. I realized that one function of the convolutional layer is in fact a feature reduction preprocessing process, but it is coupled

with feature recognition by the filters. I became aware that increasing the number of conv2D layers increases the quality of the model, but that the 'in & out' filter sizes must increase from layer one to the nth layer. It was also challenging to determine the number of fully connected layers that follow the convolutional layers.
I believe that for this application the solution is 'good-enough' given the accuracy achieved and that it almost met my expectation, however, it can be used in a general educational setting, but it is not good enough for an industry application.

5.3 Improvement

Clearly, this project was a daunting task, building a CNN classifier. I have learned a lot! Even when using a pretrained model the initiative seems more of an art-form then an engineering effort due to all the parameter adjustments and deciding on how many hidden layers and the associated optimum **in** and **out features**. As for building the scratch CNN model I learnt and realized the art of feature processing of the images using convolution with two dimensions. Considering the pretrained transfer classifier performance I would suggest the following improvements.

To improve this algorithm, I would do the following changes:

1. Experiment with the number of layers, in_features and out_features to eliminate overfitting;
2. Adjust / Tune the Dropout regularization around the 0.25 area to fight overfitting;
3. Experiment with different settings of learning rate to various values, e.g., 0.000125 -0.0002;
4. The data set for training is relatively small so I'd apply image augmentation to the data set adding:
    a. Random noise,
    b. Blur the image, and
    c. Flipping the image upside down, right and left.
5. Definitely try other pretrained models, namely DESENET-169 or ResNet50;
6. Set the validation transform to be consistent with the training transform; I read that this improves accuracy and helps with converging the train and validation loss curves.

I definitely think there is a better solution. I have read report successes using ResNet50 as well as DesNet-169 with a 93+% reported accuracy. When I used VGG16 in for the flower classification project I achieved 80+% accuracy and I expected this, but to no avail.