James Hudnall
Machine Learning Engineer - Capstone Report
10/27/2020

Capstone Report: Dog Image Classifier

I. Definition

Project Overview

My project explores the very difficult problem of taking an input image and imparting a multiclass classification model. I will implement a CNN using Pytorch to identify the presence of humans and dogs, assigning said subject the dog breed they most resemble based on the model. At first, I will train a CNN from scratch; later, I'll improve my results by leveraging a pretrained model via transfer learning.

This model will train on labeled images of dogs and humans. All data was provided by Udacity's "deep-learning-v2-pytorch" repository (Reference 1). The end goal is to provide a new image and have the model infer whether there is either a human or dog present, along with the most likely breed of either. If neither is detected, the model will reflect that in its output. The dataset of human and dog images contains a total of 13,233 and 8,351, respectively.

The domain of image classification is one that comes naturally to us humans. For computers, however, this problem posed a challenge until the advent of convolutional neural nets (CNNs). CNNs were first invented in the 1980s, but gained popularity in the mid 2000s when researchers discovered GPUs could be leveraged to drastically improve training efficiencies over CPUs (Wikipedia). Once this was better understood, CNNs allowed computers to become very good at taking labeled photos and learning to classify them (Reference 2).

At its core, a CNN breaks down an image into the individual pixels for processing (imagine a rubix cube being converted into a single column of blocks). The single row of pixels serves as the input layer to the neural net, which then passes through convolutional layers, eventually arriving at the output layer, which is used (in conjunction with back propagation) to refine the weights of the hidden/convolutional layers to achieve the desired outcome.

Convolutional Neural Networks took the computer vision world by storm in 2012 when Alex Krizhevsky won the ImageNet Large Scale Visual Recognition Challenge (Reference 3). This implementation was the first time that a CNN was used to win an image recognition contest, and the result was that the computer vision community shifted their attention to not only understanding why it performed so well, but what facets they could extract to create even more powerful networks.

Today, CNN's are integral in many modern technologies, such as Tesla's Autopilot and (soon to be) Full Self-Driving capabilities. The implementation techniques of these neural networks have been refined over the past several decades, and they have proven themselves to be a worthy engine with which we can "drive" this multi-class dog classification problem.

James Hudnall
Machine Learning Engineer - Capstone Report
10/27/2020

Problem Statement

This is a supervised learning problem, as we will use labeled data to train the neural network. The problem to be solved is training a model to correctly classify A) if it is a picture of a dog, the dog's breed, or B) detect if the image is the face of a human, alongside the dog breed that the human most resembles. For example, if I input a picture of my face, the output should state that I am human, along with the dog breed most similar to my facial features. I will implement my solution by working through the following project flow:

1. Import the data (both human faces and dog breeds).
2. For human images, I will utilize the OpenCV library to detect human faces in images, and write a human face detector I will call on later.
3. For dog images, I will use a model that has been pre-trained on ImageNet to:
    a. First, detect dogs
    b. Second, classify specific dog breeds by writing a CNN from scratch.
    c. Third, create another CNN to classify breeds, this time using Transfer Learning.
4. After the models are trained and deemed acceptable, I will construct an algorithm to:
    a. Check to see if a human is present. If so, it acknowledges this and also returns the dog breed that the human most resembles.
    b. If a dog is present, the algorithm returns the dog's breed.
    c. If neither is present, this is indicated in the output.
5. Test the algorithm with random pictures to see how accurate the predictions are in a real-world situation.

This will be deemed a success if I can produce a model that beats the benchmarks outlined below to correctly predict dog breeds in images I supply. I would eventually like to port this model to a web app that I can serve family and friends. I believe this would cement the practicality of the underlying technology, and encourage me to continue my learning.

Metrics

For each implementation, I will use the cross entropy loss function to tune the model as it trains. To judge the overall model's effectiveness, I will compare my results with those of the benchmark models, using the accuracy metric. The dataset is slightly imbalanced, but in my opinion not to the point of calling upon other evaluation metrics. Furthermore, many of the benchmarks I explored all used accuracy as a direct comparison, cementing it as the candidate of choice for this task.

II. Analysis

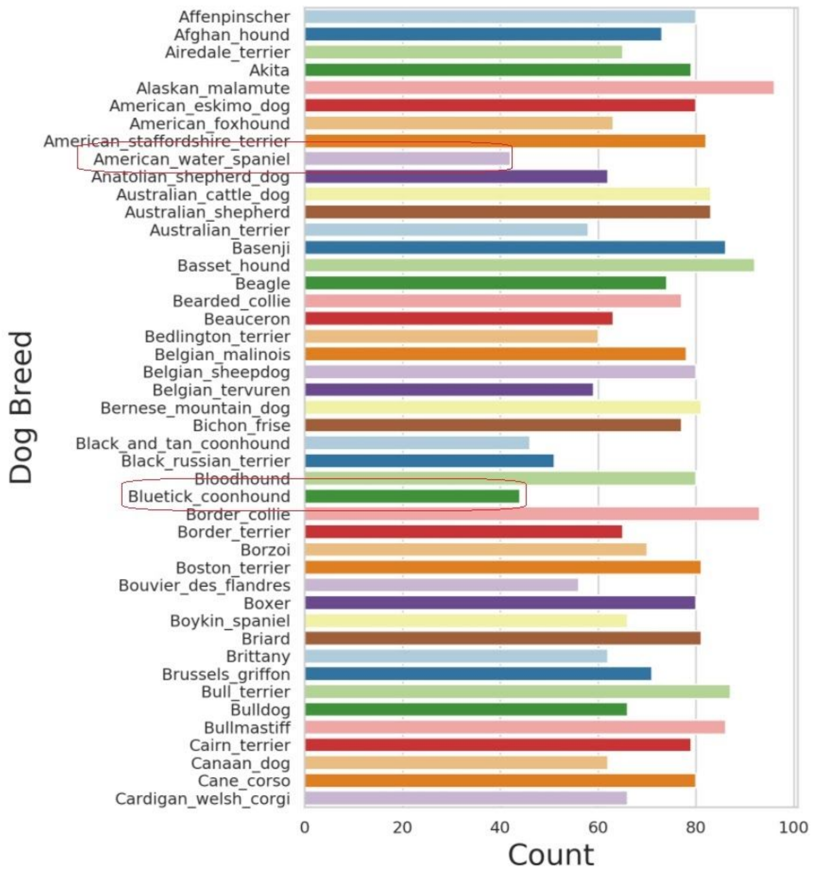
Data Exploration

All data was provided by Udacity's "deep-learning-v2-pytorch" repository. The human data consists of 5,749 folders containing a total of 13,233 images. All images of humans were 250x250 pixels. The dog pictures are pre-sorted into train, test and validation sets, each containing 133 different dog breeds, with a total image count of 8,351 dog images distributed as follows:

- Train Set: 6,680 images
- Test Set: 836 images
- Validation Set: 835 images

Exploratory Visualization

Neither the human nor dog images were evenly distributed; certain breeds and actors/actresses alike contain more individual photographs than others. Taking a look at a slice of the dog breed count distribution, you can see that some breeds have less than 50 examples, while others have almost 100:



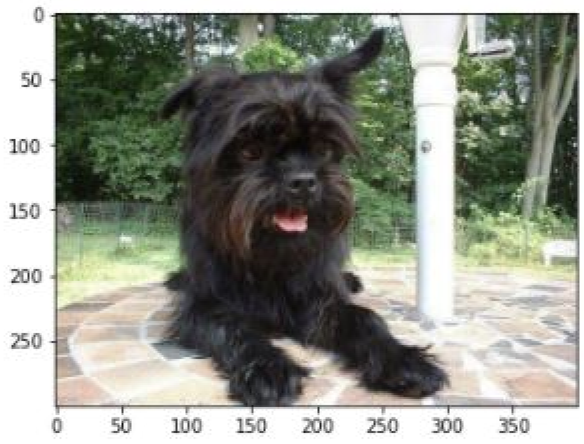
This variance in examples of different breeds will likely affect my model. While researching this, I discovered that it is recommended that dataset validation and test sets should have the same mix/frequency of observations that will be seen. Training sets should have an equal number in each class; if not, a method to improve the model's effectiveness is to replicate the less common one until it is more equal (reference 4). I decided to bookmark this idea, and come back to it if the model falls short of expectations.

Human Pictures

All human pictures were 250 x 250 pixels. A few examples:

Dog Pictures

The dog images were of different sizes.



Algorithms and Techniques

I will use the following algorithms throughout the project:

face_detector - This function will be built on top of OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces (reference 5). This function will take an image as input, convert it to grayscale using detectMultiScale function, and returns True if the classifier detects a human face. This will be used later on in our final implementation algorithm to determine if there are any humans present in target photos.

VGG16_predict - This function takes an image as input and determines the index of the image, within a segment of ImageNet's classes (total classes: 1000). This function uses a pre-trained model, so there is no training involved on my end.

dog_detector - This function will build on the VGG16_predict function above, returning True if the output index falls within the range at which dogs are located (151-268, inclusive).

CNN to Classify Dog Breeds (from Scratch) - This PyTorch model trains on our dog image dataset outlined above. The model was trained using Udacity's GPU-enabled environment (which greatly increased the training speed).

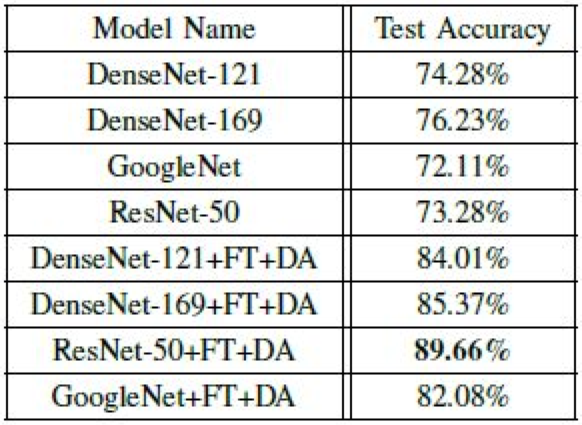
CNN to Classify Dog Breeds (using Transfer Learning) - This PyTorch model draws upon a pretrained ResNet50 model to leverage previous training. I change the final layer weights and output to adapt to the desired 133 dog breed classes.

run_app - This final function takes an image as input. It then uses the transfer model to predict the breed, is_human to detect humans (using the existing face_detector function) and is_dog to detect dogs (using the existing dog_detector function). The app checks to see if a human is detected, showing their breed. If no humans are detected, it checks for dogs, along with their breed. Finally, if neither is detected, it conveys that in the function output.

Benchmark

To be deemed successful according to the project rubric, the CNN created from scratch must exceed 10% accuracy. As a baseline, a model that completely guesses would return an accuracy of 1/133 (approximately 0.75%).

The CNN model that utilizes transfer learning must have an accuracy exceeding 60% to be deemed successful, again per the project rubric. I discovered the following benchmark chart for a similar problem, which I will use as my own benchmark when testing my ResNet50 Model:



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

Source: https://www.preprints.org/manuscript/201812.0232/v1

III. Methodology

Data Preprocessing

For preprocessing, I performed several transforms via the torchvision module. This included the VGG16_predict that was used to identify which pictures contained dogs. Within the VGG16_predict function, during the image I also discovered I needed to add an additional dimension at the beginning for the PyTorch to correctly accept the image as input.

To feed the CNN's themselves, the preprocessing started with loading the train, validation and test datasets into data loaders. **train_dataset** transforms included randomly resizing/cropping to 224x224 pixels, flipping horizontally (also, randomly with a P=0.5), and rotating with a P=0.5. They were then transformed to a PyTorch tensor, and normalized for training. **valid_dataset** and **test_dataset** were each Resized to 256 x 256 pixels, center cropped to 224, transformed to the PyTorch tensor and then normalized. I chose not to introduce the random flip or rotation, as the model was not being trained on either of these.

Implementation

Setup Functions

I started this project by creating two functions; one to detect human faces, returning True if any were present in a given photo, and the other to detect the presence of dogs. As outlined above, the face detector leveraged OpenCV's implementation of Haar feature-based cascade classifiers. The function took an image as input, converted it to grayscale, and applied the CascadeClassifier to detect faces. As outlined in the provided notebook, the **faces** variable represented a numpy array of detected faces, represented as a 1-dimensional array with 4 entries each representing a "side" of the bounding box. This model successfully identified human faces in the first 100 human files with 98% accuracy. Surprisingly, it also detected faces in 17% of 100 samples of dog pictures. After exploring the dog data images in more detail, I noticed that some of them did indeed contain human faces. Regardless, I felt that this function was accurate enough to serve its purpose of identifying human faces.

VGG16 Function

I then leveraged the pretrained VGG16 model to detect dogs. I loaded the image, defined the transform process as resizing images to 224x224 (in preparation for using them in my end-models), transforming them into tensors, and then normalizing them using the mean/std recommended in the PyTorch documentation. After defining this transformation, I also needed to change the number of channels in the input tensor to a single channel by slicing, adding an additional dimension at the beginning via the unsqueeze method. Finally, the function returns the predicted class index from 2nd element in .max() tuple.

CNN Model (Scratch)

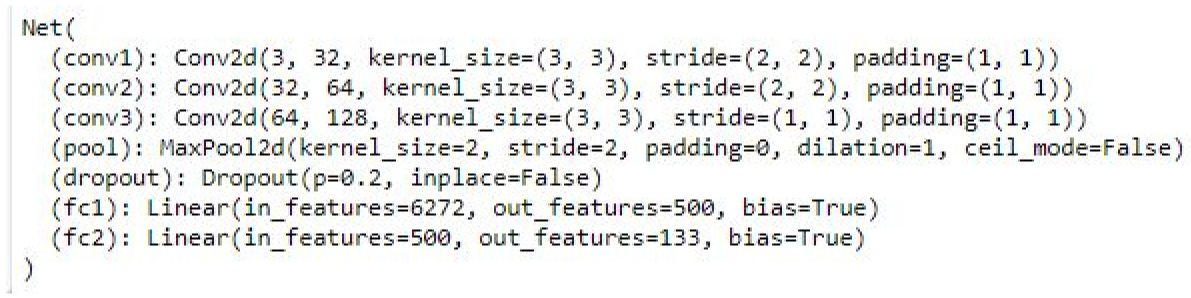
For the CNN model from scratch, I transformed and loaded the train, validation and test datasets. The transformation methodology is outlined above. I used a batch size of 20 based on a bit of trial and error and researching different techniques. I only shuffled the train_dataloader, as this was the data that would train the model and determine the eventual model weights and parameters. For the model itself:

- There are a total of 3 convolutional layers, building out to a 128x7x7 tensor. The in_channels of the first convolutional layer is configured to accept the image data, which contains 3 color channels. The out_channels builds up to 128 feature maps.

- Max pooling is then used to identify the most prominent features from the feature map (downsizes) and to make the model a bit less resource heavy.
- Dropout is used in an effort to reduce overfitting to the somewhat limited training data. According to the PyTorch docs, and the paper Improving neural networks by preventing co-adaptation of feature detectors, this is a proven technique for regularization.
- Transitioning from the convolutional layers to the FC layers requires flattening the first in_channels parameter.
- 2 fully-connected/linear layers are used on the flattened input to return to the objective class classification (our classification problem consists of 133 possibile breeds)

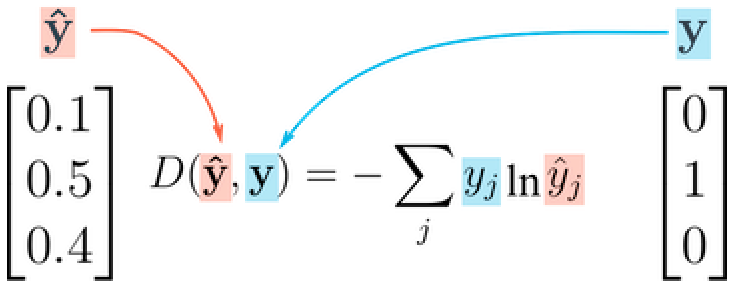
For the forward pass:

- ReLu is used as the activation function
- The network is already trained with the softmax() function inherently included, as it is performed implicitly by F.cross_entropy()**. As such, the final FC layer is returned.



```
Net(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (dropout): Dropout(p=0.2, inplace=False)
  (fc1): Linear(in_features=6272, out_features=500, bias=True)
  (fc2): Linear(in_features=500, out_features=133, bias=True)
)
```

I selected cross entropy loss as my loss function, and SGD (stochastic gradient descent) as my optimizer.


$\hat{\mathbf{y}}$
$\mathbf{y}$
$\begin{bmatrix}0.1\\0.5\\0.4\end{bmatrix}$
$D(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_j y_j \ln \hat{y}_j$
$\begin{bmatrix}0\\1\\0\end{bmatrix}$

Cross Entropy

Cross entropy is a continuous function that facilitates incremental model improvement, as follows:

1. The cross entropy loss function identifies the predicted probability for the correct label (1, in the blue Y above), ignoring the rest.
2. Next, it takes the Log of the predicted probability and multiplies by -1, resulting in larger loss values for bad predictions.
3. Finally, the average cross entropy across all output rows gives us the overall loss for a given set of data.

CNN Model (Transfer)

For the transfer CNN model, I debated using the VGG16 model for simplicity (having already used it above), but I ultimately landed on the ResNet50 model. This model is 50 layers deep, and has more than 23 million parameters, making it a prime candidate for image classification tasks such as the one I am trying to tackle reference 6.

After doing a bit of research, I learned that the ResNet architecture tackles the vanishing gradient problem (introduced by increasingly deep neural network layers) by introducing an “identity shortcut connection” that can skip layers selectively (reference 7). These shortcut connections allow the power inherent with the 50 layer depth mentioned above, without introducing the aforementioned gradient issues.

I start by copying the initial data loaders and importing the pretrained ResNet50 model. After checking the input features (there were 2,048), I froze all of the layers by setting the requires_grad method to False. I then instantiated a fully-connected layer at the end, taking the 2,048 input features, and specifying 133 as the desired output (to match the dog breeds). After making sure I was utilizing the GPU (model_transfer.cuda()), specifying the same CrossEntropyLoss and SGD loss/optimizers, I trained the transfer model over 15 epochs.

In terms of documenting the code, I added comments where I felt specific design decisions were made. I also added comments when I had to stop and research topics I was less familiar with. Overall, I believe the comments included describe the overall thought process throughout the application of each model.

Refinement

Both models surpassed the 10% and 60% minimum viable benchmarks by a fair amount. As such, I did not feel the need to do extensive refinement for improved results (my original intent was to dedicate said time to learning how to serve the end model via a web app on AWS). I did, however, experiment with the learning rate for the scratch_model.

IV. Results

Model Evaluation and Validation

Throughout the training of both CNN models, I used a robust train function that kept track of both training and validation loss. For each pass, if validation loss decreased, the model was saved. At the end of the training, I recalled the model with the lowest validation loss by loading this model. I tested each model with a **train** function that enumerated the test dataset and compared model predictions to the actual labels.

CNN Model (Scratch)

The scratch model correctly classified dog breeds with 12% accuracy. Given the fact that I only trained this over 15 epochs, I was willing to accept this performance. I would've preferred to train for many more epochs, but I had some trouble with Udacity's GPU instance in which my notebook was located. I also didn't have physical access to a cuda-compatible GPU.

As mentioned above, I selected the SGD (stochastic gradient descent) optimizer for this model, as I read in several different sources that it performed well on image classification tasks.

CNN Model (Transfer)

The transfer model achieved 76% accuracy. Given the fact that these were my first two implementations of neural networks, I was quite pleased with this result. As outlined above, the model was trained using the **train** function, whereby the model with the lowest validation loss was automatically saved. The end model was tested using the **test** function.

App

In testing the app itself, overall I was pleased with the results. When I fed it some of my own images, and images I found on the internet, the model was mostly successful. I've verified that my rescue is indeed an American staffordshire terrier, although one picture also classified her as an Italian greyhound.

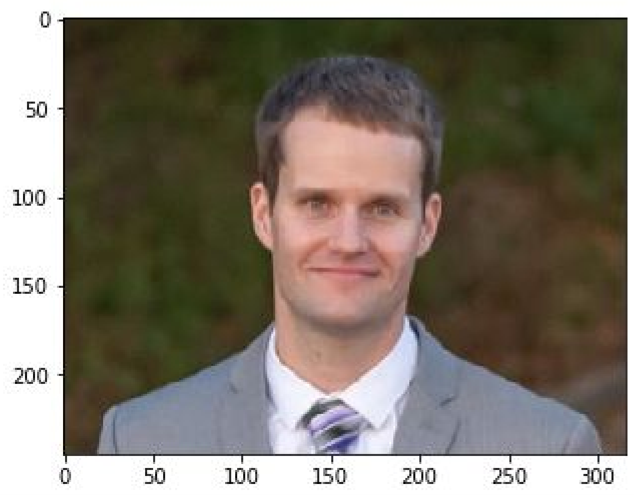
Justification

The results of the scratch and transfer models surpassed the benchmarks of 10% and 60%, respectively. Although I feel each can be fine-tuned and trained for a longer duration, I would say for the given end use, the results of the model are "good enough," though I plan to improve upon them in the future. This project is meant to be a "fun" app - not something that is determining a major decision, such as surgery or stock market projections for a brokerage firm.

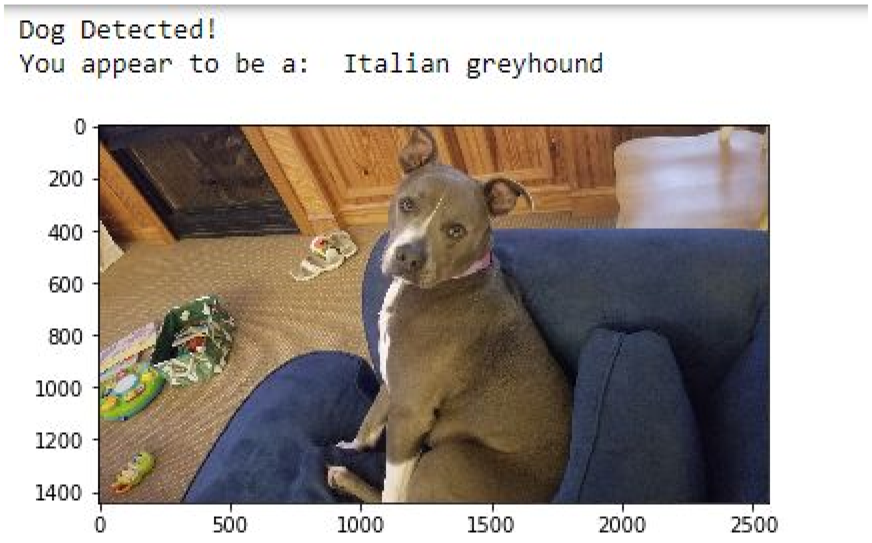
V. Conclusion

Free-Form Visualization

```
Human Detected!
In terms of dogs, you most resemble:  Cairn terrier
```



```
Dog Detected!
You appear to be a:  French bulldog
```



```
Dog Detected!
You appear to be a:  Italian greyhound
```

Surprising result for the breed selection

Reflection

I've only been studying machine learning for 3-4 months at this point, so this project was quite a challenge. Despite that, I thoroughly enjoyed working through this project, along with the skills I picked up along the way.

I particularly liked how each function built upon the previous, culminating in the ability to perform a task like dog classification. 6 months ago, this process would've been a complete black box to me. After going through this project (and more importantly, this course), I have gained a much better grasp on the key underlying principles that comprise Machine Learning. With that said, I am even more motivated by the fact that there is so much more to learn!

Improvement

I definitely feel like I can improve upon the accuracy of both models, particularly the scratch_model. I've been reading about how to better account for unbalanced data, and I now have a better understanding of how I can account for classes with lower representation by means of replicating them, bringing the overall distribution closer to an even distribution. I think this would improve that model.

I also feel like I could have played more with the hyperparameters of both CNNs, particularly with:

- The number of epochs
- The learning rate (implementing PyTorch's built-in **Scheduler** to incorporate automatic learning rate decay.
- Momentum (to avoid getting "stuck" in local minimums, reducing the amount of epochs required to achieve a given train/validation loss).

For the CNN structure, I plan to experiment with introducing batch norm into the process, in an effort to make the training process more efficient (reference 8).

Finally, I would eventually like to serve this model via web application to family and friends.

References

1. https://github.com/udacity/deep-learning-v2-pytorch/tree/master/project-dog-classification
2. https://en.wikipedia.org/wiki/Convolutional_neural_network
3. https://en.wikipedia.org/wiki/AlexNet
4. https://arxiv.org/pdf/1710.05381.pdf
5. http://docs.opencv.org/trunk/d7/d8b/tutorial_py_face_detection.html
6. https://medium.com/@nina95dan/simple-image-classification-with-resnet-50-334366e7311a#:~:text=What%20is%20ResNet%2D50%20and,applied%20to%20analyzing%20visual%20imagery
7. https://towardsdatascience.com/an-overview-of-resnet-and-its-variants-5281e2f56035
8. https://machinelearningmastery.com/batch-normalization-for-training-of-deep-neural-networks/