Machine Learning Convolutional Neural Network For Dog Breed Classification

Capstone Project

Ben Jacobson
February, 2020

I. Definition

Project Overview

Classification of objects into categories, groups, or types is essential in technology, business, and biology. It allows us to generalize knowledge for quick understanding or application. As individuals, we are constantly assessing our own environment for safety or danger, or perhaps, clues to success or failure. Though we don't actively classify our assessments, the concept and its importance remain the same. Classification is essential to nearly all domains of life. It only makes sense we seek technology that aids us to do this.

Thirty years ago, these concepts inspired science fiction. In Terminator 2, Arnold Schwarzenegger had famously quoted "My CPU is a neural net processor… a learning computer." The real breakthrough in computer vision happened in 2012 and arguably brought forth a wave of confidence with it. The AlexNet neural network outpaced competition by greater than 10% in the ILSVRC (ImageNet Large-Scale Visual Recognition Challenge). The challenge was to accurately classify a set of 1000 classes of images [1,2]. Since this time, neural networks have gained more prominence and acceptance beyond the "black box" it was described as. However, success breeds success, and a slew of deeper and non-sequential neural networks have continued the advancement, led by large multinational companies like Google and Microsoft [3].

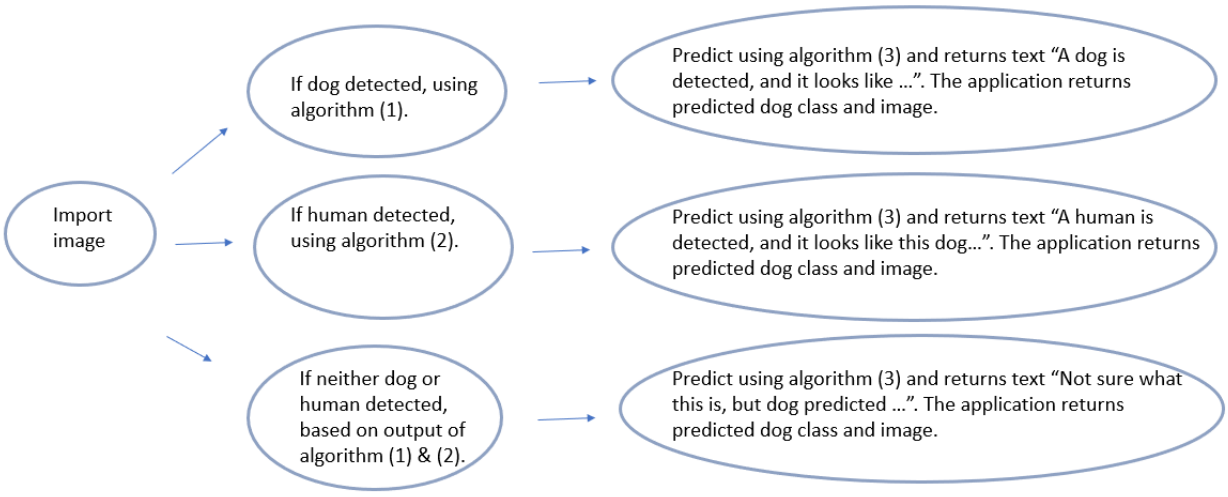
Innovative applications have appeared in various domains, such as in healthcare. In 2017, deep convolutional neural networks (CNN) were used for detection of skin cancer with dermatologist-level accuracy [4]. The application wasn't restricted to CNN application. Neural networks have since been used in healthcare diagnosis and administrative decision making [5].

Since that healthcare problem is “solved” barring the obstacles of regulation, we do have one that is quite similar, yet more fun: dog breed classification! We seek to classify pictures of dogs into one of 133 dog breeds in our data. This problem is less abstract than classifying skin lesions, but can have practical application at a veterinary office, provide useful learnings, or a fun application (which is our intent). This paper is not one to be nominated for a Nobel prize, but can be applied to thousands of similar problems in different domains, that perhaps collectively aid mankind. Is this an easy problem? Can you tell whether this is a Komondor or an Old English Sheepdog?

Problem Statement

The aim is to accurately detect a dog in an image and predict the breed of that dog, based on 133 classes. This is accomplished in three parts – separate algorithms that determine whether (1) a dog is in an image (2) or a human is in an image and (3) what dog breed is predicted for the image. Our main focus will be on (3).

The final application will be deployed, and use these algorithms in combination, similar to the following flow chart.



The general framework of the project follows these steps, and iteratively through multiple models as defined in the problem statement. These steps are as follows:

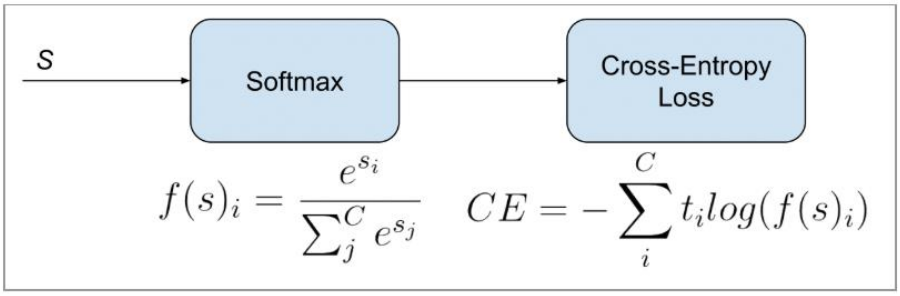
1. Retrieve Input Data
2. Clean and Explore
3. Prepare/Transform
4. Develop/Train Model
5. Validate/Evaluate Model
6. Deploy to Production
7. Monitor and Update Model & Date [7]

Evaluation Metrics

Both the benchmark and solution model(s) will be evaluated for accuracy (select correct dog breed class or not) using Softmax Cross-Entropy Loss. Compared to MAE and other metrics, Softmax Cross-Entropy Loss (cross categorical entropy) has been noted to allow better training with more complex datasets, and more generalizable results, as well as is able to be differentiated. [6]

Since this is a multi-label classification problem in which only one predicted class is allowed as an output, we will use Softmax Cross-Entropy Loss for this type of problem. Softmax Cross-Entropy Loss takes into account the groundtruth and CNN score for the groundtruth class (of the 133 dogs. The Softmax equation provides normalized probabilities that sum to 1, and we apply cross-entropy loss on the groundtruth class it should be, *-ln(probability of class)*. In pytorch, *nn.CrossEntropyLoss* applies both the Softmax equation and cross-entropy loss. The more accurate the network is, the smaller

the loss. The following depicts our loss function applied using pytorch built in functions. [11]



Based on this loss/cost equation, we calculate our training loss with each iteration after the forward pass, and with this calculate the gradients, and change the weights based on the gradients. (This is our model learning!)

The running train and validation loss are displayed. The function **test** is defined in the benchmark (scratch) model and applied in the models for that are created in the transfer learning stage. The final calculation for accuracy is simple: if the prediction is correct, then 1, if not 0. This is summed and divided by the total number of predictions, giving our accuracy.

II. Analysis

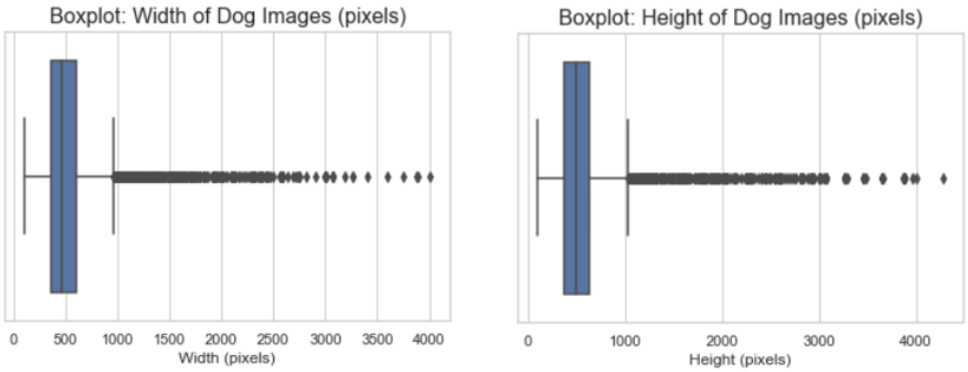
Data Exploration: Dataset and Inputs

The following lists the data and inputs for model creation. Both datasets are comprised of images of various size and perspective, but all color (RGB, 3 channel). The datasets are used in the project in order to train, validate, and test predictive models.

- Dog Breed Image Dataset
    - This is a dataset of 133 classes/breeds of dogs, a total of 8,351 images. Each of the 133 dog breeds will be divided into a train, validation and test data set.
- Human Face Dataset
    - This is a dataset of 13233 images of humans.

The following shows the range of input images in our data that we will use to train, validate, and test the model. All images are three channels, which indicates RGB. The "width" dimension has an average of 529 pixels and "height" dimension of 567 pixels. However, statistically speaking, there is a lot of variation in the data, with standard

deviation of 333 and 389 pixels, respectively. So, what does this mean to us? Some images have more information than others. Preprocessing steps may affect some images differently as well.



Exploratory Visualization

In examining just a few images, there are some important elements to note. From left to right there are many differences, (1) has text in the image, (2) the image is at an offset angle and just the upper body, (3) has two dogs, and (4) has plants in the foreground, with many puppies, I think. So, in examining just a few images, we find there is considerable variation in the "framing" of the image and what is in it, not just the dog breed.

In comparison, the human faces dataset is composed of RGB images of the same size 250x250. On initial inspection, they appear a bit cleaner. However, we do see the same rotation, and multiple people in the same image. This is not the focus of the project, but worth mentioning since it is used to determine if the image is of a human, rather than a dog.

Algorithms and Techniques

The anticipated solution will use a machine learning framework based on a convolutional neural network (or multiple) to take an input image, detect if is a dog, and then classify that image based on the available classes. We will use transfer learning, namely, fine-tuning.

The early layers of a CNN detect edges, patterns, and going deeper, squares or circles. More complex representations are found, such as eyes and noses. Finally, more abstract features such as cars, dog types, or toothbrushes are detected. So, these early layers generalize well to most data, while we may need to change what is lastly created in the classifying layers at the end.

While the Benchmark (scratch) model had randomly initialized weights, we use a pre-trained network (its model artifacts and learned weights). We use the pretrained weights in the early layers, and freeze them, while allowing the last dense layers to be trained. This is where we are "fine-tuning" the CNN weights, and reclassifying only for 133 classes. Since the dataset is mostly small, we will be freezing most the layers, which does vastly improve training time as well.

The recommended solution is intended to be operational, and fast, in order to be used in an application. In particular a MobileNetV2, VGG, or SqueezeNet1.0 model will be trained, tested and selected. Data transformation will be used, as discussed in the following section "Data Preprocessing" as well as to avoid overfitting.

The aim is to measure performance via appropriate metrics, in a replicable and quantifiable manner. As a result, datasets are provided in the GitHub repository, as well as code to replicate the model creation as selected in the final application.

Default parameters for the VGG16, SqueezeNet1.0, and MobileNetV2 models can be found in the pytorch documentation, with the model artifacts displayed in the GitHub repository for this project [13]. However, all three models had the classification layers unfrozen, and a new layer added for 133 classes.

Benchmark

We need a model, preferably parsimonious or historical, to compare the proposed solution with. As requirements for the project, a CNN from scratch is to be created. Random chance indicates that 0.75% of the time the correct class will be selected (based on 133 classes). However, our expectation is to have a benchmark model built and trained from scratch, that achieves greater than 10% accuracy. This will be our benchmark model.

The benchmark model will employ random initialization of weights, approximately five convolutional units, max pooling, batch normalization, dropout, and a final dense layer for classification. Multiple epochs and a smaller batch size will be used. The exact architecture will be built to exceed 10% accuracy based on cross categorical entropy. This is mandated in the project guidelines. (The final model would require greater than 60% accuracy and be based on transfer learning.)

III. Methodology

Data Preprocessing

- The dog image set was randomly divided, without replacement, into a train, validation and test set.
- The batch size used is 20 images.
- Each image goes through the following transformations before being batched and used in the CNN models.
    - The image is resized to 256x256x3
    - The image is then cropped to 224x224x3, in alignment with the transfer learning models used
    - The dataset is augmented by using a Random Rotation of 20 degrees. This is likely important based on the exploratory data analysis, in which most image have the dog at a variety of angles.
    - The dataset is augmented by using a Random Horizontal Flip with a 50% chance of occurring. This allows for mirror images, which makes sense in this context.
    - The image is converted to a tensor.
    - The image is normalized.

Implementation & Refinement

This section combines the implementation and (iterative) refinement of the different models using our framework:

4. Develop/Train Model
5. Validate/Evaluate Model

This is applied to the following models, and discussed:

- Benchmark (scratch model)
- VGG16

- SqueezeNet1.0
- MobileNetV2

Refinement was necessary to have better predictive accuracy, but also to deploy the model on Heroku. The initial VGG-16 model was too large to deploy, surpassing the "slug size of 500MB", debugging this issue was quite tricky since documentation is sparse. As a result, the use of SqueezeNet1.0 and MobileNetV2 became necessary. These models have similar performance, yet smaller architecture and size, so I decided to develop both and choose the most appropriate.

The Baseline (Scratch) Model

I started with a basic design, and 3x32x32 input tensor. A fully connected layer with 133 nodes was used for the 133 dog classes. After ~10 epochs, the accuracy was low at 1%, no better than chance. An additional 128-depth convolutional layer was added with ReLu activation and pooling, was trained for 50 epochs, which brought the test accuracy to 2%. Clearly, more layers were needed. Also, I reasoned that such a small image may not contain enough information to discern the features needed to classify different breeds. As a result, I increased the size to 3x224x224.

Dropout layers were included to avoid overfitting as in most models, and I increased the depth, extracting more features, and, as a result, higher accuracy. Padding is included to avoid loss of information in the convolutions.

Following this approach, I saw an improvement, reaching 10% accuracy with 10 epochs and a learning rate of .001. I researched other ways to improve the model and saw that batch normalization between layers proved useful in keeping the original distribution and efficiency (regularization), which upon implementation, at 40 epochs, the model was at 18% accuracy.

Lastly, the model still had not converged yet, so I ran for 100 epochs achieving an accuracy of 26% on the test data. This represents our benchmark model, and with 26% accuracy is better than the random chance of < 1% accuracy, meeting our project requirements. (See appendix 1 for final architecture.)

Transfer Learning Models

VGG16

I initially utilized the VGG16 architecture to be consistent with the early stages of the project. We froze all early layers and unfroze last layer for classification into our 133 dog

classes. After training the model for 100 epochs, the model maintained 57% accuracy, which is pretty good, and had yet to converge.

However, this model is too large for operationalizing via Flask/Heroku. Unfortunately, while attempting to deploy the model to Heroku, I learned that the operational "slug" size is only allowed at 500 MB, so a different model was needed in order to operationalize the model. (It would be fine on AWS but our chosen platform is Heroku for this project). So, I needed to select architectures that had similar performance, yet smaller size.

SqueezeNet1.0 & MobileNetV2

So, I decided to train two smaller models (1) **SqueezeNet1.0** and (2) **MobileNetV2**due to the smaller model size. (As a result, the jupyter notebook has been recycled multiple times, which can be found in GitHub under the commit history). Both have slight nuances in the classification layers and the predictive output, but overall have the same

- After training the SqueezeNet1.0 model for 10 epochs, we obtained an accuracy of 57%. This model about the same as VGG16, but would work operationally.
- After training, MobileNetV2 resulted in higher accuracy at 82%. This did train two full days (not GPU, much longer than either VGG16 or SqueezeNet1.0, which was surprising.

As a result, the model chosen for operationalization was MobileNetV2 due to performance, and also the ability to operationalize on Heroku. Deployment of the model was successful, and currently in production (active).

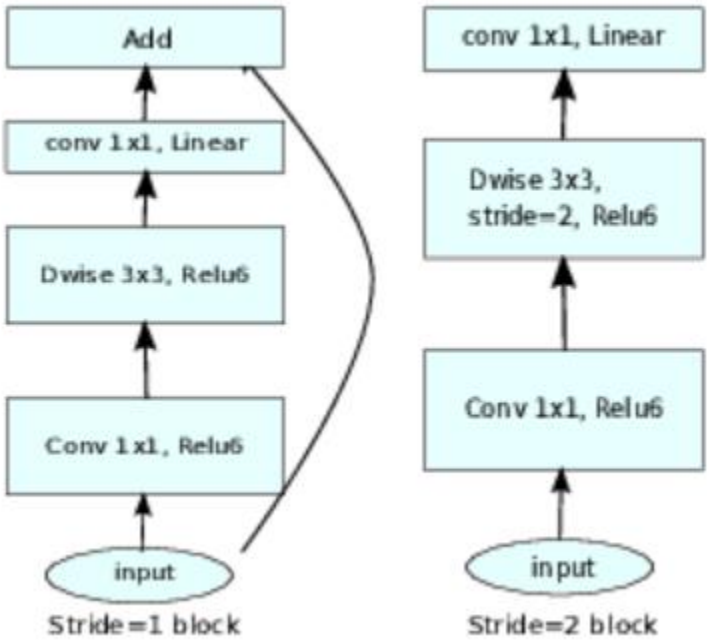
IV. Results

Model Evaluation and Validation

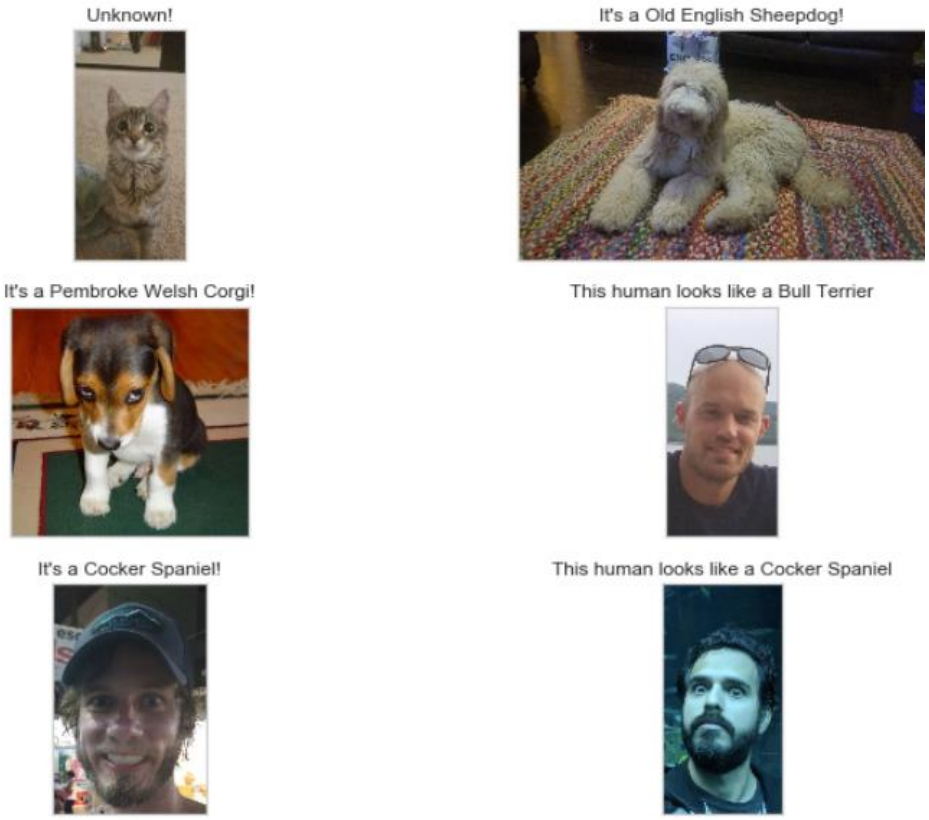
Final model evaluation was based on the evaluation metric, categorical cross entropy, as well as the ability to be operationalized on Heroku. The following is the accuracy (based on test data) for each model:

Model	Accuracy (%)	Size (MB)
Scratch	26%	26.3
VGG16	57%	526.6
SqueezeNet1.0	57%	3.2
MobileNetV2	82%	9.5

Not only is MobilNetV2 the most accurate, but also able to be operationalized. Thus, it is a natural choice for deployment. This architecture [11] is fairly new from Jan 2018, and utilizes "…inverted residual structure where the input and output of the residual block are thin bottleneck layers opposite to traditional residual models which use expanded representations in the input an MobileNetV2 uses lightweight depthwise convolutions to filter features in the intermediate expansion layer…"[12] The model uses a novel architecture that does a tradeoff between accuracy and latency, with the goal of use in mobile applications.



The output of the MobilNetV2 model on various images is accurate for dogs, and, I suppose subjectively, predicts the humans "correctly". I had a few good laughs as well. Here, the final model is tested with various inputs of unseen data to evaluate whether the model generalizes well, and apart from a few errors, seems to do well. So it is fairly robust, but does not so fair in some instances.



Justification

Though many models were trained, this MobileNetV2 transfer-learning model was chosen due to performance, resource availability and operational ability (not too big)! Apart from additional refinement, this model does quite well.

The model was trained using the pytorch framework, and maintained an accuracy of 82% on unseen test data. To put this success into perspective, random chance would correctly select the dog breed less than 1% of attempts. This is pretty good! When compared to our benchmark model trained on 100 epochs of 26% accuracy, this is an obvious difference, and justifies final selection and deployment.

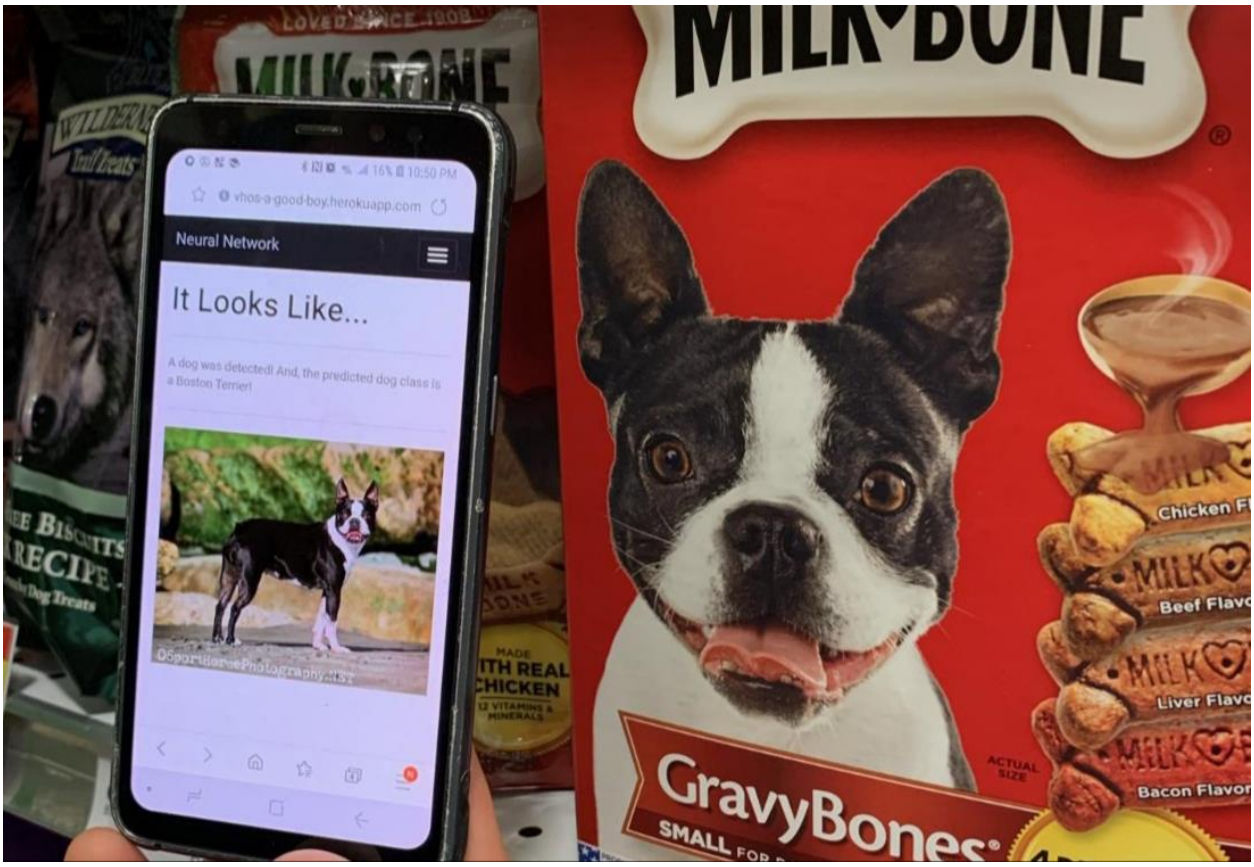
V. Conclusion

The Final Application

The application made use of Heroku, an open source PaaS, with a Bootstrap framework. The CSS and HTML was customized for simplicity but maintain the connectivity between the various components (algorithms as stated in the problem statement). This was the

application's "scaffolding", with all documents maintained on GitHub. The Heroku framework was chosen due to ease of development and low cost (none). At last, the combination of models (1), (2), and (3) were used to evaluate an image and predict the dog class. The following image shows the application (desktop) used to take a picture of a bag of dog food, and predict the image.

Try it for yourself at https://whos-a-good-boy.herokuapp.com/!



Reflection

The entire process worked very well and was a combination of iteration between algorithmic performance and the ability to operationalize. The steps used at times was "failing fast" and learning iteratively. The most difficult aspect was operationalizing on Heroku and having to completely re-think the models being deployed. Also, the error messages were not always clear.

Improvement

The following are areas for improvement:

- (1) improving the models used for **face_detector** and **dog_detector**, perhaps custom CNNs perhaps based off transfer learning,
- (2) improving the architecture of the **model_scratch** through additional layers
- (3) utilizing data augmentation/larger dataset to improve the model. In order to operationalize the model on Heroku, I needed to consider size, so deploying on AWS could ease that concern.
- (4) using addition data transformation, and augment the image set to be larger.
- (5) putting the deployment code in the same repository as the development code

Of course, each model could be improved perhaps by training with different hyperparameters (learning rate, epochs,…), considering different model architectures, increasing the dataset, or even modifying the underlying problem statement. That said, the application and algorithm, by and large, achieve our project objectives.

Citations

[1] https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neuralnetworks.pdf

[2] http://www.image-net.org/challenges/LSVRC/

[3] https://adeshpande3.github.io/The-9-Deep-Learning-Papers-You-Need-To-Know-About.html

[4] https://www.nature.com/articles/nature21056

[5] https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0212356

[6] https://papers.nips.cc/paper/8094-generalized-cross-entropy-loss-for-training-deep-neuralnetworks-with-noisy-labels.pdf

[7] https://Udacity.com

[8] https://cloud.google.com/ml-engine/docs/ml-solutions-overview

[9] https://gombru.github.io/2018/05/23/cross_entropy_loss/

[10] https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/CNN20Whitepaper.pdf

[11] https://www.semanticscholar.org/paper/MobileNetV2%3A-Inverted-Residuals-and-Linear-Sandler-Howard/dd9cfe7124c734f5a6fc90227d541d3dbcd72ba4/figure/6

[12] https://arxiv.org/abs/1801.04381

[13] https://github.com/radleap/Machine-Learning-CNN-Dog-Classification-Project

Appendix

[1] Scratch Model Architecture

```
Net(
  (features): Sequential(
    (0): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (4): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (5): ReLU(inplace=True)
    (6): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (7): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (11): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (12): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (15): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (16): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (17): ReLU(inplace=True)
    (18): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (19): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (classifier): Sequential(
    (0): Dropout(p=0.25, inplace=False)
    (1): Linear(in_features=12544, out_features=500, bias=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.25, inplace=False)
    (4): Linear(in_features=500, out_features=133, bias=True)
  )
)
```