Machine Learning Engineer Nanodegree

Capstone Project: Classifying Dog Images by Breed

By Ross Weijer

June 25, 2020

I. Definition

Project Overview

In this capstone project, I will apply machine learning to a problem within the field of computer vision, more specifically the subfield of image recognition. Computer vision is one area of research in which machine learning has yielded the most concrete and tangible gains over the past decade. A primary venue in which these gains have been recognized is the ImageNet Large Scale Visual Recognition Challenge, which since 2010 has consistently raised the standards for algorithmic approaches to object detection and classification across an increasing number of categories[1]. Not only is the ImageNet database and finer curated subsets of the data widely available for academic and personal use, but so are many of the deep Convolutional Neural Network (*hereafter CNN*) architectures that have proven highly successful in the image classification challenges supported by ImageNet. Top-performing algorithms on ImageNet's data, including Inception[2] and GoogLeNet[3] developed by Google and various iterations of the VGG architecture from Oxford University[4], are all widely available within popular deep learning frameworks such as PyTorch[5], which we used heavily in this course, albeit not for supervised classification of images.

Although I have spoken at a very general level about image recognition, for this project I will focus my efforts on a very specific classification problem and restrict my input data only to images of dogs. Crucially, these dog images come with labels that specify what breed of dog is present in the image. The algorithms I develop will solve a multi-class classification problem, taking in an input image and placing it within one of the multiple dog breeds present within our data. This challenge is one of the capstone project domains recommended by Udacity for the

[1] http://www.image-net.org/challenges/LSVRC/
[2] https://arxiv.org/abs/1512.00567
[3] https://arxiv.org/abs/1409.4842
[4] https://arxiv.org/abs/1409.1556
[5] https://pytorch.org/docs/stable/torchvision/models.html

course, and Udacity sources this data back to a Kaggle contest[6]. However, my work will be using the data as downloaded from the Stanford Dogs Dataset website[7], hosted by the Stanford researchers that originally curated and annotated this subset of the broader ImageNet database. The main reason for my choice is that the Stanford researchers also provide accompanying bounding box coordinates that allow me to crop out the non-canine portions of each image.

Problem Statement

The simplest framing of the central problem is as follows: given an image of a dog, can a machine learning algorithm tell me what breed of dog the image contains? As stated in the previous section, this is an example of multi-class classification, and deep CNNs have emerged as the cutting-edge approach to these types of classification problems when images are the type of data being classified. Consequently, the algorithms explored in this project will all be CNNs. More specifically, I will compare how a CNN that has been trained from the ground up—the only approach to training neural networks we covered in our coursework—performs relative to a neural network architecture trained via a transfer learning process that will be more clearly delineated in the Algorithms & Techniques section of this report.

Metrics

In order to compare how different CNNs perform as dog breed classifiers, one needs to specify a standard way to score each algorithm. The metrics I will be using in this report are **top-1 accuracy** and **top-5 accuracy** calculated as follows. For a multi-class classification with N total possible classes, each CNN can produce a probability of any given image belonging to each class. So in my specific case where the classes are dog breeds, I can simply look at the most probable breed and five-most probable breeds predicted by my CNNs and check whether these subsets contain the ground truth breed label for each image.

For example, let's say I fed an image of a *German Shepherd* into one of my CNNs.

- If the network produced predicted class labels [*German Shepherd, Doberman, Rottweiler, Norwegian Elkhound, Miniature Pinscher*], then this prediction would be **top-1 accurate and top-5 accurate**.
- If the predicted class labels were [*Doberman, Rottweiler, German Shepherd, Norwegian Elkhound, Miniature Pinscher*], then this prediction would be **top-5 accurate but not top-1 accurate**.
- If the predicted class labels were [*Doberman, Rottweiler, Leonberg, Norwegian Elkhound, Miniature Pinscher*], then this prediction would be **neither top-1 accurate nor top-5 accurate**.

[6] https://www.kaggle.com/c/dog-breed-identification/overview
[7] http://vision.stanford.edu/aditya86/ImageNetDogs/

Variants of top-K accuracy, where K is some integer much smaller than the total number of classes N, are frequently used in evaluating multi-class classification problems. Most notably, top-1 and top-5 accuracy are used to evaluate how many of the CNNs in PyTorch perform on the ImageNet database, which in its entirety comprises 1,000 classes.[8] Accuracy is an appropriate performance metric to use for classification problems on *balanced data*: data sets whose points are relatively evenly distributed across the N classes being predicted. I will specifically demonstrate the balanced nature of the Stanford Dogs dataset in the next section of this report.

II. Analysis

Data Exploration

I will begin this section with a more detailed discussion of the size, shape, and other characteristics of the Stanford Dogs dataset. As downloaded from the site hosted by the original researchers, the data covers 20,580 color JPEG images comprising 120 different breeds of dog. Not only is the data pre-organized into folders by breed, the researchers also provide a means of splitting the data into 12,000 training images and 8,580 testing images[9]. Further, they provide bounding box annotations for each of the images in the form of XML files that allow users of the data to remove people, buildings, and other extraneous background features that might complicate the breed classification problem. A sample image from the data set is presented in Figure 2.1.

There are two other characteristics worth mentioning about the dataset that slightly alter the high-level train and testing counts presented above. First of all, it is possible for a single image to contain multiple dogs. In these cases, the associated annotation XML will contain multiple bounding boxes, and I chose to split that image into *multiple train or test instances*. By choosing to save a separate image for each bounding box in the course of preprocessing, I actually ended up with 12,877 train images and 9,249 test images (growth in size of 7% and 8%, respectively). Not only was this growth relatively small, it was also relatively evenly distributed across breeds, so it didn't meaningfully upset the balanced nature of the data set.

Second, in applying the bounding boxes, I found one particular case in which an image labelled as Chihuahua contained two bounding boxes that each held different breeds of dog, as shown in Figure 2.2. In the raw dataset, both of these sub-images were labelled as Chihuahua. I did not make the effort to visually inspect all of the images for any mislabellings but rather accepted the data

[8] The most widely reported summary table of various CNNs on ImageNet is found here in the PyTorch documentation. I note that this performance is on an *unspecified random subset* of ImageNet, so we can't be sure whether it includes all of the 1,000 classes in the full dataset. This table also shows error, which is the complement of accuracy.
[9] http://vision.stanford.edu/aditya86/ImageNetDogs/

as it was labelled. These sort of mislabelling events are often a “cost of doing business” when using widely available, human annotated images for machine learning problems.

Exploratory Visualization

My exploratory data analysis process was centered around answering two key questions. First and foremost, to justify my choice of top-1 and top-5 accuracy as evaluation metrics, I wanted to answer the question: *is the Stanford Dogs dataset balanced*? To answer these questions I produced barplots of the number of images by breed label across both the entire dataset (Figure 2.3-a) and looking just at the training data (Figure 2.3-b). From these plots we see that most breeds are represented by between 150 and 200 images, and that there are consistently 100 training images per breed (*at least before we split images by applying bounding boxes*). As such, we have an adequate number of examples of each example breed to fuel learning and our dataset is balanced so we don’t need to alter our proposed evaluation metrics.

Secondly, I wanted to get a sense of how large and uniform in size the images in the dataset were. Even though the images would be cropped and resized as part of any standard image classification workflow, I was curious to get a sense of how high-resolution they were before any of those steps were applied. The resulting joint distribution of image width and height, shown in Figure 2.4, reveals that most of our images in the Stanford Dogs dataset are approximately 500px wide by 375px high, with the second-most common image size being in the vicinity of 375px wide by 500px high. We note again that this was *before* application of bounding boxes and also excluded a small minority of images that had either dimension above 1000px.

Algorithms & Techniques

The two basic approaches that I took to utilizing CNNs were constructing a relatively shallow network and training it from scratch, and applying a transfer learning approach that’s fundamentally about adapting a pre-trained network to a more specific classification task. In this section, I will detail each of these approaches in both general and specific terms.

As inputs to machine learning tasks, images are essentially very high-dimensional vectors of pixel values, with the dimensionality determined by each image’s width, height, and number of color channels. At its core, the architecture of a CNN is designed to reduce that dimensionality in a clever way by taking advantage of the fact that we can learn interesting higher-order features of images by looking at regions of pixel values. Put concretely, these CNNs employ convolutional layers, which rely on sliding a small-window matrix operator repeatedly over the image to abstract out general visual patterns like edges, shapes, and gradients, and MaxPooling layers, which reduce small regions of an image to the maximum of all pixel values contained therein, in combination with the more traditional fully connected layers of other neural network architectures.[10]

While there are many ways to assemble the different layers of a CNN, many architectures rely heavily on repetition of smaller sub-architectures. However, CNNs that are deep enough to achieve generalizable classification performance across many different types of images require a large volume of input data and significant amount of time to train. Keeping both of these principles in mind, I constructed a relatively shallow CNN (*hereafter referred to as BasicConvNet*) to serve as my initial algorithm to implement. As shown in Figure 2.5, BasicConvNet includes two Convolution - ReLU-MaxPool layers that are applied to the input image before it is flattened and passed through two fully connected linear layers that primarily handle the classification.

The high-level architecture of BasicConvNet, such as number of layers and overall depth, was inspired by a simple CNN used to classify sprite images from the Pokémon game series[11]. Each convolution layer requires the choice of a few hyperparameters—the number of convolution filters K, spatial extent of each filter F_c, the stride of each filter S_c, and the amount of zero padding P_c to apply to each dimension of the input—while pooling layers also require the choice of spatial extent F_p and stride S_p. Following recommendations from Andrej Karpathy for smart defaults for these parameters[12], I applied the following values throughout BasicConvNet: $F_c = 3$, $F_p = S_p = 2$, and $S_c = P_c = 1$. I also set K = 16 for each Convolution Layer.

Now that my initial algorithmic approach has been detailed at both general and specific levels, I'd like to give a similar overview of transfer learning. Keep in mind that very deep CNNs take a very long time to train, and that the convolution and pooling layers at the front end of the network are generally specialized to extract higher-order visual patterns like edges or blobs of color. The premise behind transfer learning is to take a pre-existing architecture that has learned its constituent features on millions of images, keep the uppermost convolutional layers that extract visual information static, replace one of the final fully connected layers of the architecture, and only retrain that replacement layer on a more specific subset of data[13]. The approach gets its name from the fact that we transfer some of the inherent features that the CNN learned from a larger data set and transfer it to a new problem[14] without incurring the high time investment of starting from scratch.

[10] A very good high-level overview of Convolutional Neural Networks along with a basic application example can be found in this Journal of Geek Studies article. A more in-depth and interactive explanation of the mathematical and computational principles at hand is available at Andrej Karpathy's open-source CS231n curriculum.
[11] https://jgeekstudies.org/2017/03/12/who-is-that-neural-network/
[12] See specifically the respective summaries of the Convolution Layer and Pooling Layer portion of his lectures.
[13]The explanation above is modified after the "fine tuning" approach defined here.
[14] As detailed in the *Approach to Transfer Learning* section of this Medium post.

Once I decided on following the approach of fine-tuning a pre-existing architecture to a more specific classification problem, two remaining decisions were: which architecture(s) do I want to modify, and in what way? There honestly wasn't a high amount of domain-specific knowledge that motivated the first of these decisions. I had seen people leverage the VGG-16 architecture in transfer learning applications before[15] within the python library Keras, and I found reference implementations for both VGG-16 and ResNet50 using PyTorch that were even closer to my intended use case[16]. I then chose GoogLeNet as a third candidate for transfer learning because I wanted to compare "small", "medium", and "large" approaches: measuring by number of total parameters contained, VGG-16 was "large", ResNet50[17] has about an order of magnitude fewer parameters than VGG-16 and was thereby "medium", and GoogLeNet has an order of magnitude fewer parameters than ResNet50 and was therefore "small"[18].

With architectures chosen as candidates for transfer learning, I needed to choose how to modify each of these architectures. Following the general example specified here[19] I added on two small fully connected layers that were separated by a ReLU activation and dropout operation that reduced the output dimensionality of each network down from the 1,000 possible classes of ImageNet to the 120 possible breeds in the Stanford Dogs dataset. All subsequent mentions of VGG-16, ResNet50, and GoogLeNet in this report refer *not* to the original architectures but instead to the architectures with their final fully connected layers replaced by the layers in Figure 2.6.

Benchmark

The first benchmark that I established for this project was the absolute simplest one I could think of: implementing a Probabilistic Random Guess classifier, based solely on the frequency of each of the different breeds across the entire Stanford Dogs dataset. So, for example, if Chihuahua images comprised 5% of the overall data set, I would simply guess Chihuahua for any input image 5% of the time. I was able to do this by simply looking at the frequency of breeds in the dataset and using that as an input to the random.choice( ) method in NumPy[20]. Using this to generate five predicted breed labels for each of the testing images and then comparing that with the ground truth label yielded **top-1 accuracy of 0.98%** and **top-5 accuracy of 4.18%**. This benchmark is incredibly naive, though, and a more useful benchmark was quickly established as detailed in the Implementation section of this report.

[15] https://datascopeanalytics.com/blog/building-a-visual-search-algorithm/
[16] See the *Function to Load a Pretrained Model* section of this GitHub Jupyter notebook holding the code behind the Medium post in footnote 14.
[17] https://arxiv.org/abs/1512.03385
[18] Including the custom classifier layer that I added to each architecture, VGG-16 has approximately 136 million total parameters, ResNet50 has approximately 25 million parameters, and GoogLeNet has just over 6 million parameters. These calculations followed the implementation specified in cell 22 of the same Github Jupyter notebook above.
[19] See specifically the *Pre-trained Models for Image Recognition* portion of this Medium post.
[20] https://numpy.org/doc/stable/reference/random/generated/numpy.random.choice.html

III. Methodology

Data Preprocessing

The preprocessing required for this project was less centered around feature selection and more around reorganizing the images as they were downloaded from Stanford. The raw files came categorized according to breed subdirectories inside one large *Images* folder, with bounding box files stored in a separate *Annotation* folder and lists specifying which paths pointed to train images and test images. Because I would be relying heavily on PyTorch for implementing deep learning, I wanted to follow the conventions specified by PyTorch's ImageFolder class[21] in which images are organized into train and test directories with subdirectories specifying the class labels (*in this particular case, the names of each breed of dog*).

In order to accomplish this, I wrote a separate preprocessing module that parsed the raw train and test lists supplied by the Stanford Dogs dataset, matched the image pathways within these files to their appropriate bounding box XML information, cropped every bounding box region out of the raw images, and saved these cropped sub-images into a new directory structure organized the data set type (*train* or *test*) and then the breed label[22]. At the end of this pipeline, the raw files provided by the researchers were deleted, to minimize the amount of data to be sitting around on a Sagemaker notebook instance that needed to be uploaded to S3.

Implementation

With this preprocessing pipeline set up to organize the data properly and tested locally, I was ready to move things into the Sagemaker notebook environment and prepare to actually train and test a CNN. I decided to start by implementing by BasicConvNet following many of the Sagemaker/PyTorch coding conventions we learned in this course: moving the training directory with all of its breed-specific subdirectories up to S3, populating a *sourceGroundUp* directory with model.py, train.py, and predict.py files; creating a training job on a GPU-accelerated ml.p2.xlarge instance that trained the network for 20 epochs in 128-image batches; using the resulting model to host a prediction endpoint on a normal ml.t2.medium instance, and passing the test images from the properly configured directory on my notebook instance to that endpoint in small batches to return predictions while avoiding the 5MB network request cap for these inference endpoints.

[21] https://pytorch.org/docs/stable/torchvision/datasets.html#imagefolder
[22] This routine is well documented in the run_pipeline( ) method of the preprocess.py module in this project's Github repository.

The only step in this process that merits a little bit extra attention is an image augmentation routine that I worked into the training script utilized by sagemaker. Image augmentation helps ensure that when a CNN is being trained, we are presenting a slightly different variation of the same base image across epochs through the data set. In general, this helps make sure that the network isn't overfitting to highly specific artifacts of each photos—such as lighting, orientation, and angle—but is instead learning about properties of the particular dog contained in the photo. I implemented the image augmentation pipeline recommended here[23] which can be roughly summarized as: taking a random 85 to 100% of each input image; resizing it to 256px by 256px; randomly rotating the image some amount within 20 degrees in either direction; randomly jittering the color values; randomly flipping the image along its horizontal axis with probability of 50%; and finally center cropping the image to an input size of 224px by 224px and normalizing it according to values recommended by PyTorch when working with ImageNet data[24]. Several sample outputs of this augmentation process applied to a reference image can be viewed in Figure 3.1.

Training our BasicConvNet from the ground up for 20 epochs took about 38 minutes, and training top-1 accuracy on the final epoch reached 64%. In what was the most initially unexpected development in this process, generating predictions for the approximately 9,000 testing images took about 35 minutes—almost as long as the entire training routine took. I quickly realized that the prediction endpoint should probably *also* be served on an ml.p2.xlarge instance that can support GPU computations. The BasicConvNet achieved **top-1 accuracy of 11.6%** and **top-5 accuracy of 31.7%**. While this is a significant improvement from our Probabilistic Random Guess benchmark, it was still a bit disappointing overall. The fact that our testing accuracy was so much lower than our training accuracy indicates a possibility that our BasicConvNet may have overfit to the training data, but because I didn't implement an independent validation set it's difficult to know for sure. What I do know for sure is that this experience reinforced why people rarely train CNNs from scratch in practice[25].

Refinement

With my initial benchmark surpassed, but not by as much as I would have liked, I wanted to see for myself how much better we could do with using transfer learning on the three architectures discussed earlier. In broad strokes, the process I followed was very similar to the one outlined in the first and second paragraphs of the prior section. The only key difference was that instead of pointing to a directory where the train.py script loaded in the BasicConvNet model, I instead pointed to a *sourceTransferLearn* directory in which the train.py script loaded in a pre-trained model of a specified type—"vgg16", "resnet50", or "googlenet"—and froze all of its parameters, then replacing the final layer with the classifier architecture we discussed earlier. This way,

[23] See the *Image Preprocessing* section of this Medium post.
[24] For guidelines on input size and normalization, see https://pytorch.org/docs/stable/torchvision/models.html#classification
[25] Don't take my word for it, take Andrej Karpathy's.

when training occurred, the only weights being learned belonged to the layers in this more specific classifier.

The only other change to the process that I *tried* to implement was to serve the prediction endpoint on an ml.p2.xlarge instance, believing that my AWS Resource Limit of 1 could be used for both training and hosting instances. Unfortunately, I found out the hard way that that wasn't the case. While I put in the proper AWS Resource Increase request as soon as I found this out, I eventually decided to bite the proverbial bullet and use a non-GPU accelerated instance for deploying my inference endpoint.

Training the GoogLeNet CNN for 20 epochs to specialize it for dog breed classification took about 37 minutes, about as long as our BasicConvNet took, and our training accuracy got to 68.2% on the final epoch. On an ml.t2.medium instance, generating test predictions with GoogLeNet took approximately 40 minutes, yielding **top-1 accuracy of 64.5%** and **top-5 accuracy of 89.2%**. It's worth remembering that GoogLeNet is the smallest of the network architectures that we are using for transfer learning.

Performing transfer learning with ResNet50 for 20 epochs to fine tune it took about 52 minutes, about 15 minutes longer than GoogLeNet, and our training accuracy got to 77.3% on the final epoch. On an ml.t2.medium instance, generating test predictions with ResNet50 took just about 64 minutes, admittedly a long time but not prohibitively so. ResNet50 achieved **top-1 accuracy of 72.9%** and **top-5 accuracy of 93.0%**, outperforming GoogLeNet by a noticeable margin.

Finally, we come to the largest of our transfer learning targets, VGG-16. Training this behemoth for 20 epochs took about 63 minutes, only 10 minutes longer than ResNet50, yielding a final epoch training accuracy of 75.3%. In what's arguably the hardest call I had to make during this project, I decided to sink *3 hours and 12 minutes* into generating predictions from VGG-16 on a non-GPU accelerated instance, about three times as long as it took for ResNet50. VGG-16 achieved **top-1 accuracy of 73.1%** and **top-5 accuracy of 94.2%** on the 9,200 testing images, marginally beating out ResNet with respect to our core evaluation metrics.

IV. Results

Model Evaluation & Validation

A neater summary of the top-1 and top-5 accuracy performance of our four CNNs, along with other metrics on how our inference cycle played out, can be found in Figure 4.1. Looking at these results side-by-side we can see that the transfer learning approaches are in a class of their own with respect to our evaluation metrics. For all three of our transfer-learned CNNs, testing accuracy was within five percentage points of our final training accuracy, which serves as evidence that the useful features these networks "learned" on the totality of ImageNet are generazing well to the more specific subset of the Stanford Dogs dataset. This stands in stark contrast to our BasicConvNet, which saw a delta of over 50 percentage points between train accuracy and test accuracy.

Furthermore, because we're implementing image augmentation, we know that we have made minor tweaks to the training instances with each forward pass through our CNNs, and at least in the case of our transfer learning networks we are still able to see solid performance on an independent test set. This fact, along with the relatively small decline between train and test accuracy, leads me to believe that any of our transfer-learned networks would be relatively robust to new inputs.

The only other evaluation consideration I will raise here relates to the time it took for these networks to perform inference. Even though VGG-16 achieved slightly higher top-1 and top-5 accuracy rates than ResNet50, I can't help but note that this gain in performance came at the cost of a 200% increase in time required to generate predictions for our test images. Admittedly some of this difference might be obviated when the prediction service would be moved to GPUs and VGG-16's inference time might be brought down to more acceptable levels. Nevertheless, I'm inclined to choose ResNet50 as the overall "winning" model here because it strikes the best balance between performance and speed, even in a non-GPU inference context.

Justification

Overall, taking a transfer learning approach to dog breed classification yielded impressive performance, vastly outstripping both a purely probabilistic approach and a ground-up training approach at comparable levels of training time. It bears mentioning that in the specific cases of VGG-16 and ResNet50 we're not only seeing relatively durable test accuracy, we're also seeing similar performance as was reported for these architectures on the broader ImageNet database itself[26]. This stands as perhaps the strongest evidence that these two networks in particular have successfully "transferred" their learning to a more specific sub-classification problem of recognizing the breed of a given dog from a photo. These results are strong enough for me to declare either of the transfer learning CNNs a successful solution to this problem, with particular emphasis on the strong performance of VGG-16 and ResNet50.

V. Conclusion

Visualization

To get a bit of a deeper look at how accurate each of these CNNs were, I not only looked at global top-1 and top-5 accuracy rates but also visualized these rates within each of the different breed classifications. These breed-level top-1 accuracy plots are shown for the BasicConvNet,

[26] This conclusion can be reached by converting the top-1 and top-5 error rates listed here in the PyTorch documentation into top-1 and top-5 accuracy rates.

GoogLeNet, ResNet50, and VGG-16 in Figure 5.1.a, Figure 5.1.b, Figure 5.1.c, and Figure5.1.d, respectively. Not only do these barplots drive home the incredible variety of the classification challenge undertaken there, they also underscore how the transfer learning CNNs are really in a league of their own. Our BasicConvNet achieved 0% top-1 accuracy for a handful of breeds and struggled to surpass 30% for this metric in the best case. Conversely, our transfer learning CNNs only fell *below* 30% top-1 accuracy at the breed level a handful of times. GoogLeNet saw about 50% of dog breeds in the dataset fall between 60% and 80% top-1 accuracy, whereas ResNet50 and VGG-16 saw 50% of breeds place solidly above 70% top-1 accuracy. The final interesting comment I can make on these visuals is that specific breeds tended to rise to the top or fall to the bottom of the accuracy rankings *across* the different architectures. For example, the *Keeshond* breed reported the highest top-1 accuracy rate within each of our three transfer learning architectures, whereas *Chow*, *Dhole*, *Japanese Spaniel*, and *Schipperke* placed within the top five across two of the three CNNs. There is less uniformity at the bottom of the ranking, with *Miniature Poodle* and *Siberian Husky* showing up in the bottom five breeds for two of our CNNs. Further analysis of the visual similarities between these breeds, perhaps by a qualified human expert or dog show judge, could shed some light on why certain breeds are more difficult to classify correctly than others on a semi-consistent basis.

Reflection

Overall, this capstone project confirmed a lot of the things that I thought I knew about image recognition and deep learning, specifically the overwhelming supremacy of transfer learning, but had only heard indirectly through colleagues' experiences or read in other articles. I am glad that I was able to get my own first-hand experience and put an end-to-end practical project under my belt . Much of how the results turned out, even down to fact that the best accuracies being clocked were in the 70-80% range, conformed to my prior expectations going in.

I think an aspect of this work that was most interesting and surprising to me waas how easy it was to handle pre-processing, manipulation, and augmentation of the input images using torchvision's Transpose API[27]. That was something that I was certain I was going to sink a lot of coding and implementation time into, but in actuality it was pretty easy to write a relatively custom PyTorch data loader that automatically chained many of these standard image operations together in a repeatable fashion.

The most difficult aspect of the crowd is that, even in the best case scenario, training and predicting with these deep CNN architectures takes a considerable amount of time, a burden that only gets magnified when you're spinning up separate servers to handle various steps of this process. I would have liked to prototype, tweak, and iterate a bit more, but there's a big difference between tweaking a hyperparameter that alters how a few cells in a notebook run locally and making a change that requires several hours of end-to-end deployment in order to yield results. It felt like I sort of had to know the exact resources I needed and experiments I

[27] https://pytorch.org/docs/stable/torchvision/transforms.html

wanted to run before I was able to make a lot of progress, because there wasn't as much time to adjust an analysis plan on the fly.

Improvement

There are two primary paths of improvement that I would pursue if continuing on this project. The first and arguably easier to implement would be to alter my data preprocessing pipeline to create a stratified validation set that holds back 20% of the training images within each breed category. With that done, I could modify my training scripts relatively easily to perform validation within the course of each epoch, potentially allowing me to implement an early stop to training if the validation loss holds constant across several epochs. The use of a validation set would allow me to be more confident in my hypothesis that my BasicConvNet was overfitting but my transfer learning CNNs were not. Additionally, given how time intensive training is, it seems like it would be a big advantage to be able to stop that process early with some degree of confidence if it seems like you're hitting diminishing returns.

The only way in which I would repeat this process with an independent validation set, however, would be if I had access to the compute resources that it takes to get predictions back. The most obvious way to do this is to switch the deployment instance type from ml.t2.medium to a ml.p2.xlarge with GPU acceleration. The fact that AWS provisions training resources separately from hosting resources is one of those inconvenient truths that I discovered midway through my progress on sagemaker, and a request to fix it was mid flight via AWS support as this report was being drafted. I was willing to wait three hours for VGG-16's predictions just once for the sake of being able to finish this draft, but I would *not* be making that tradeoff again.

Another possible way I could have sped up inference was to look more closely at utilizing a Sagemaker batch transform job instead of relying on repeated calls to a running inference endpoint. Most of what I read suggested that this is the better course of action when one doesn't need real-time predictions, but I was unable to find good reference examples for how to implement a batch transform job with image data. On the contrary, much of the specific guidance around sagemaker's PyTorch inference endpoint seems to me much more wedded to a deploy-centric workflow.

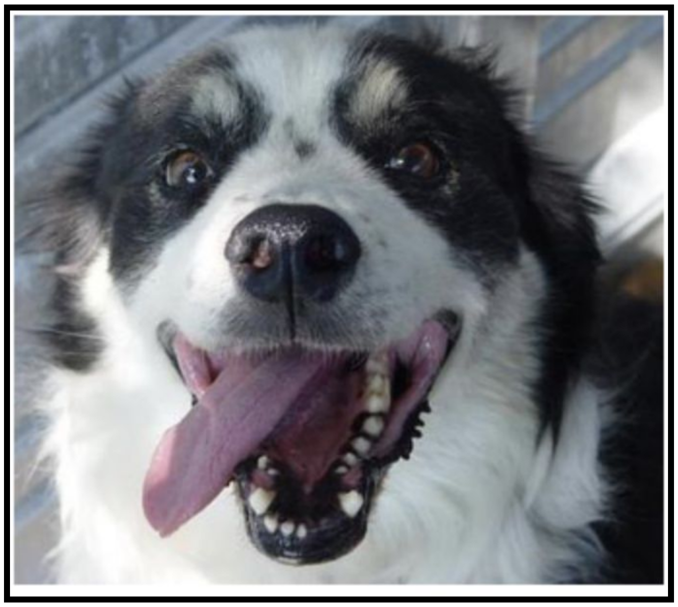
Had performing inference on a GPU accelerated machine been possible, I would have also liked to explore an alternative approach to transfer learning that relies on using the CNN as a feature extraction technique instead of fine-tuning the existing architecture for classification. The general principle behind this feature extraction strategy is to remove the final softmax layer of a pre-trained architecture so that the CNN instead outputs roughly 1,000 dimensional feature vectors that can be fed into a classifier like an SVM[28]. Given that the Stanford Dogs dataset is a reasonably large and relatively well defined subset of ImageNet, I don't necessarily think that

[28] See the first bullet in these notes from Andrej Karpathy for more on using ConvNets as feature extractors.

the feature extraction approach would have outperformed the fine-tuning approach to transfer learning, but it would be interesting to run the experiment and use it as a benchmark in any case.

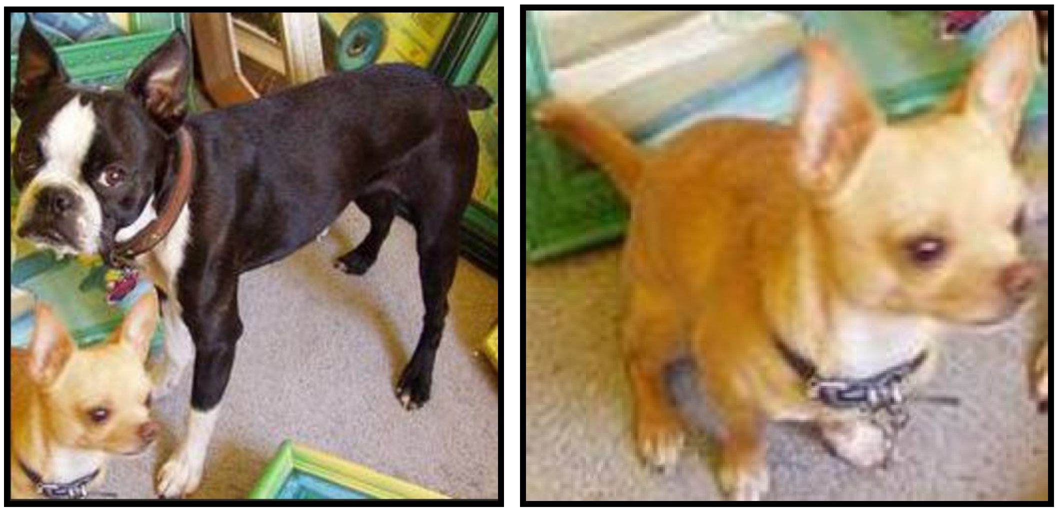
VI. Figures

Figure 2.1



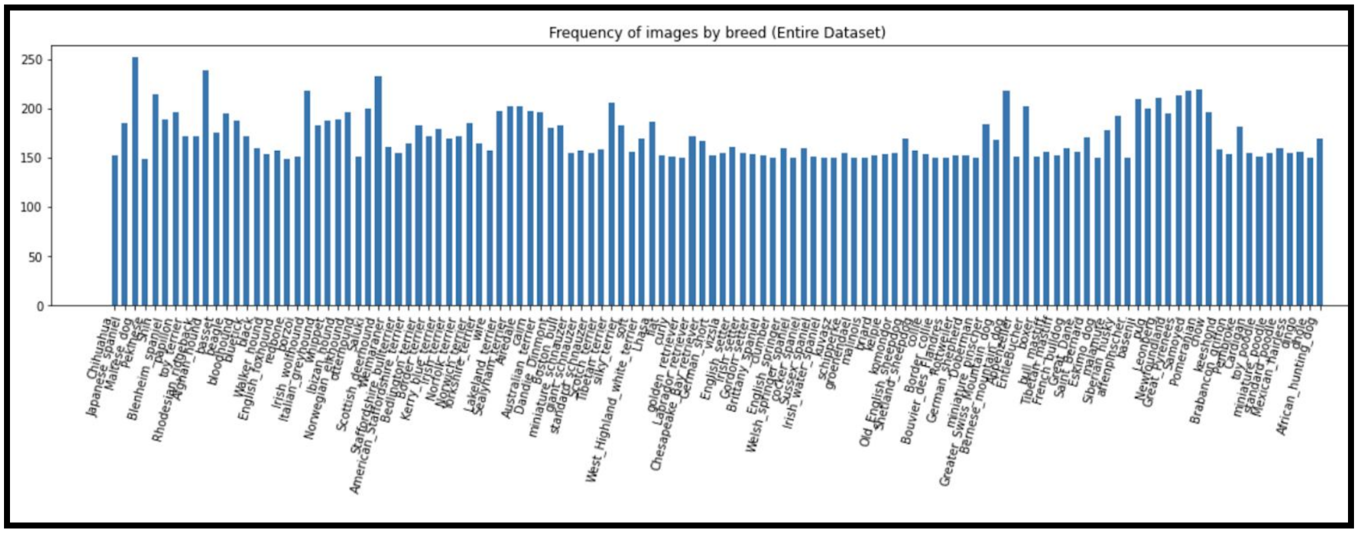
A sample image from the Stanford Dogs dataset that has already had its bounding box cropping applied. The breed label for this image is *Border Collie*. (return to text)

Figure 2.2



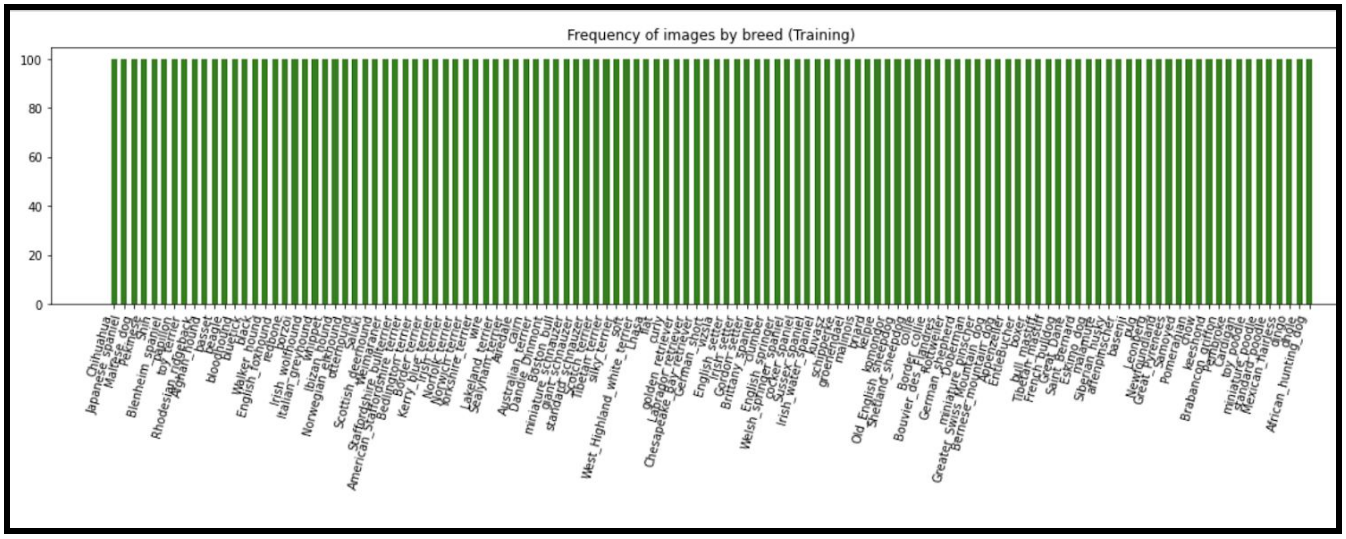
Two bounding boxes contained in image *n02085620_5661.jpg* from the Stanford Dogs dataset. The first bounding box contains a *Boston Bull Terrier*, the second contains a *Chihuahua*, but both of these bounding boxes were labelled as *Chihuahua* in the original dataset. (return to text)

Figure 2.3-a



Barplot showing the number of overall images in the Stanford Dogs dataset by breed. The overwhelming majority of the 120 dog breeds are represented by between 150 and 200 images in the data set, *before applying bounding boxes perturbs these counts slightly*. (return to text)

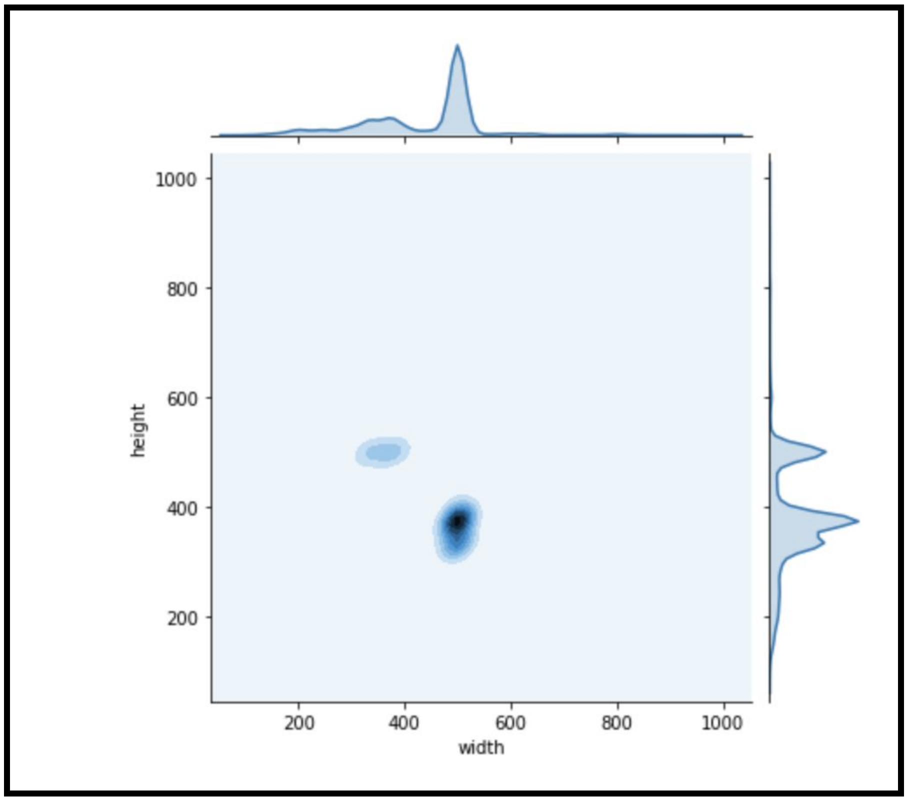
Figure 2.3-b



Barplot showing the number of training images in the Stanford Dogs dataset by breed. The training set provided by the researchers includes exactly 100 instances per class, *before applying bounding boxes perturbs these counts slightly*. (return to text)

Figure 2.4

Joint Distribution of Image Width and Height
across Stanford Dogs Dataset



Joint distribution of width and height for images in the Stanford Dogs dataset, before bounding boxes are applied. The most common size range for images is approximately 500px by roughly 375px high. The transpose of these dimensions, 375px by 500px, is the second most common size. Excludes approximately one percent of the data that is above 1000 pixels in either dimension, to improve readability of the plot. (return to text)

Figure 2.5

Layer Type	Input Shape	Output Shape
Conv2d	(224, 224, 3)	(224, 224, 16)
ReLU activation	*No shape change, element-wise operation*	
MaxPool2d	(224, 224, 16)	(112, 112, 16)
Conv2d	(112, 112, 16)	(112, 112, 16)
ReLU activation	*No shape change, element-wise operation*	
MaxPool2d	(112, 112, 16)	(56, 56, 16)
Flatten	(56, 56, 16)	(50176,)
Linear Layer	(50176,)	(3136,)
ReLU activation	*No shape change, element-wise operation*	
Linear Layer	(3176,)	(120,)
LogSoftmax	*No shape change, converts to log probabilities*	

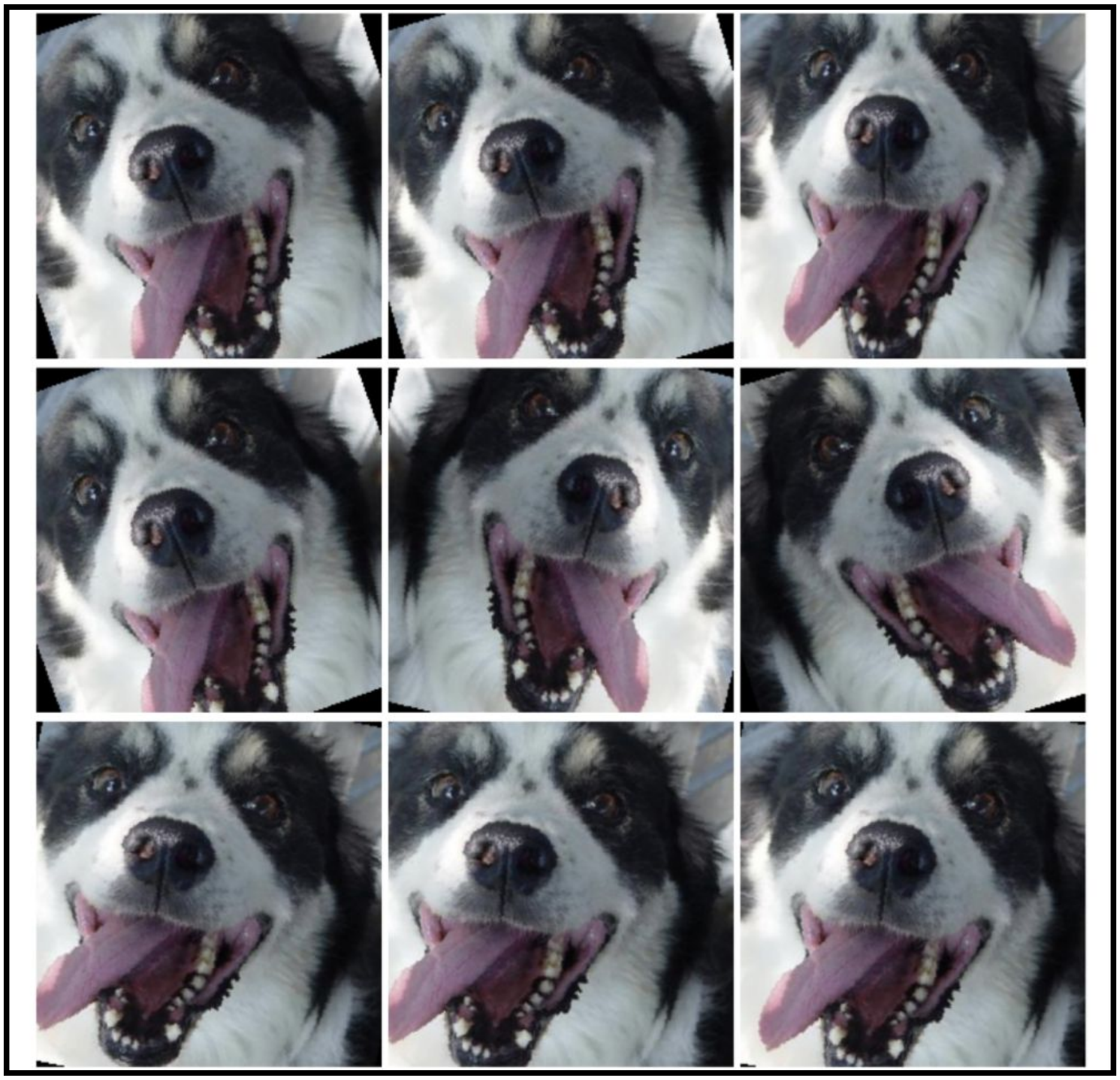
Table diagramming the architecture of BasicConvNet, the initial algorithm I implemented to solve the problem of classifying dog images into breeds. (return to text)

Figure 2.6

Layer Type	Input Shape	Output Shape
Linear Layer	(N,)	(512,)
ReLU activation	*No shape change, element-wise operation*	
Dropout	*33% chance of removing a connection between units*	
Linear Layer	(512,)	(120,)
LogSoftmax	*No shape change, converts to log probabilities*	

Table diagramming the sequential classifier that replaced the final fully connected layer of each pre-trained architecture I used in transfer learning. The value of N at the entry point to this classifier varied according to which architecture was used and was not something I had any control over: for VGG-16, N = 4096; for ResNet50, N = 2048; for GoogLeNet, N = 1024 (return to text)

Figure 3.1



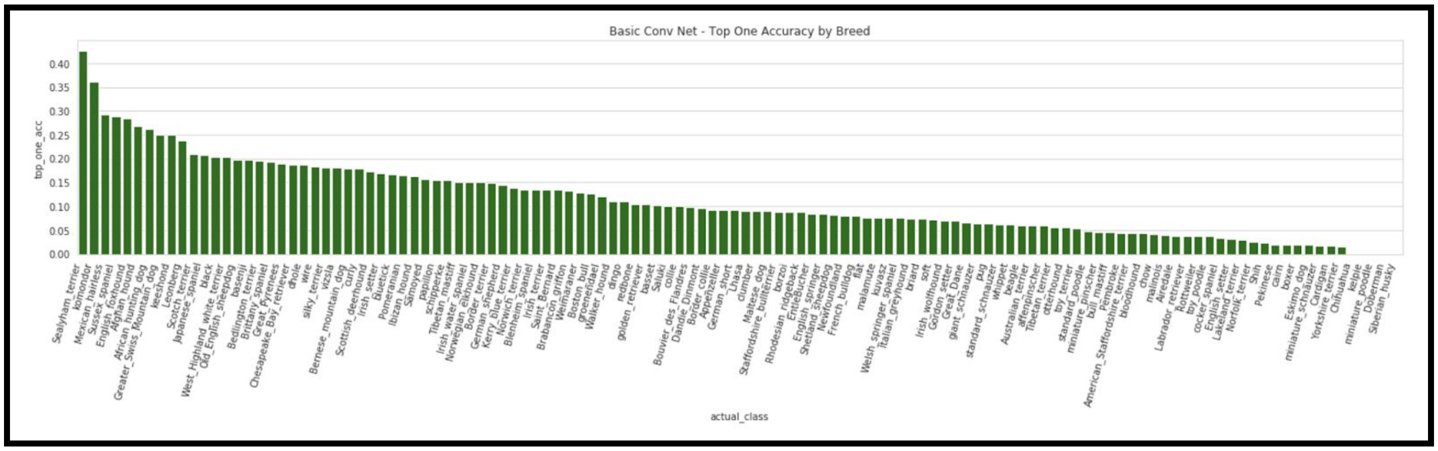
Nine examples of how the sample image from Figure 2.1 may potentially be modified when passed through the image augmentation pipeline we used to train our CNNs. (return to text)

Figure 4.1

Model	Training Time (ml.p2.xlarge)	Final Epoch Training Accuracy	Inference Time (ml.t2.medium)	Test Top-1 Accuracy	Test Top-5 Accuracy
RandomGuess (*Benchmark*)	N/A	N/A	N/A	0.98%	4.18%
BasicConvNet	38 min	64.4%	35 min	11.6%	31.7%
GoogLeNet	37 min	68.2%	40 min	64.5%	89.2%
ResNet50	52 min	77.3%	64 min	72.9%	93.0%
VGG-16	63 min	75.3%	192 min	73.1%	94.2%

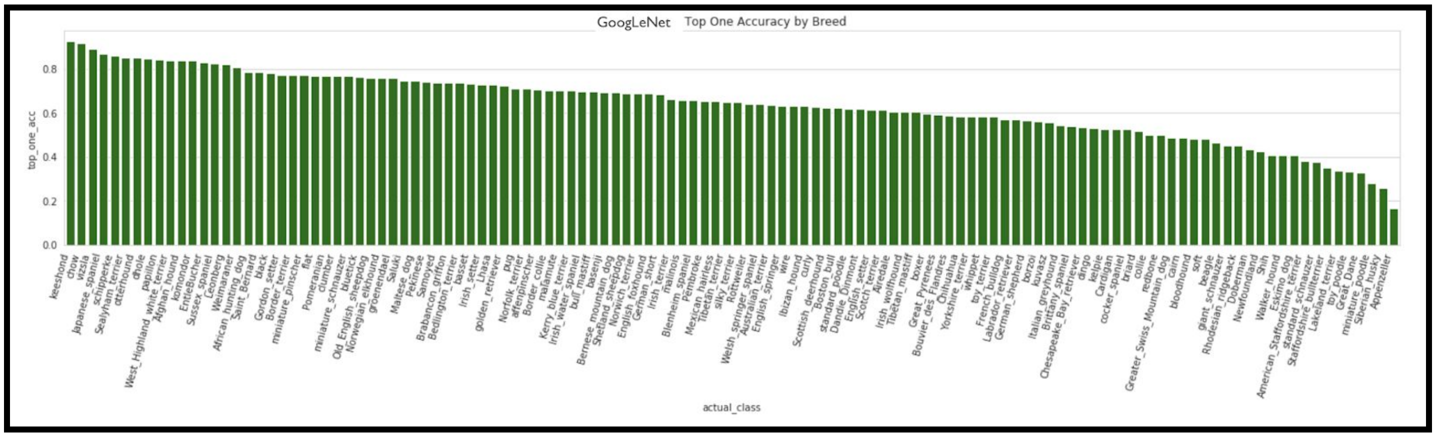
Model Evaluation summary for our initial benchmark, our CNN trained from the ground up, and our three experiments in transfer learning. (return to text)

Figure 5.1.a



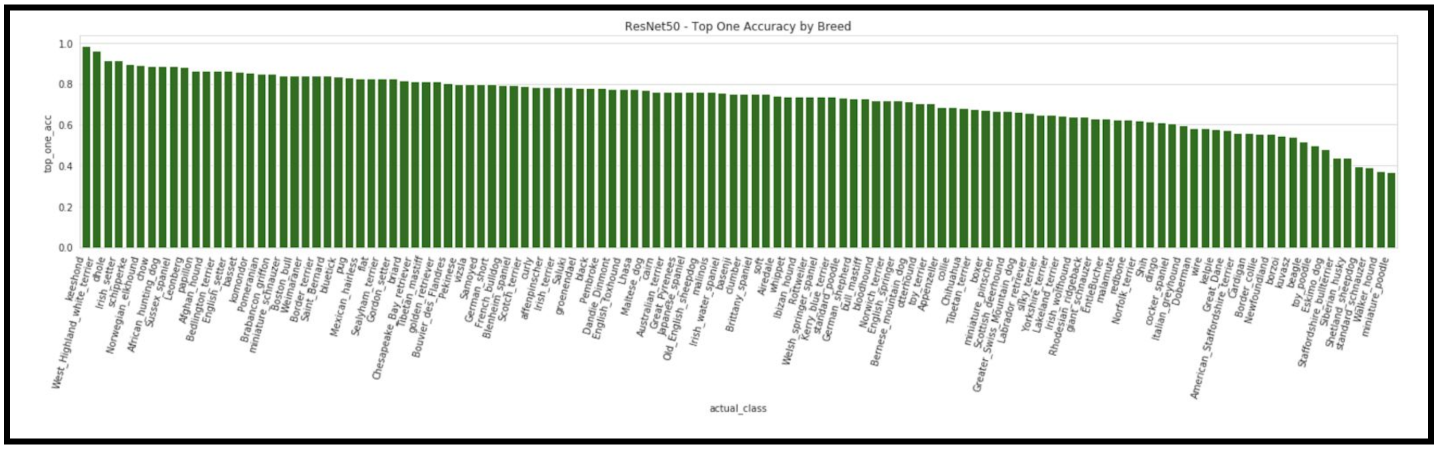
Top-1 accuracy rates by breed achieved by BasicConvNet on the test images. Several breeds had accuracy of 0%. Over half of breeds have top-1 accuracy below 10%. Only two breeds have top- accuracy rates above 35%. (return to text)

Figure 5.1.b



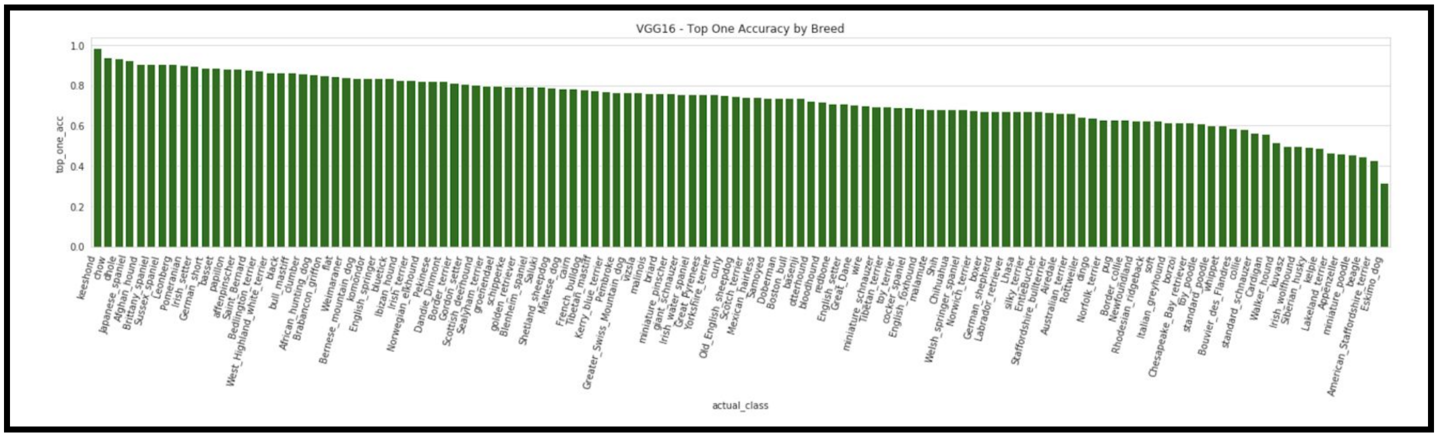
Top-1 accuracy rates by breed achieved by GoogLeNet on the test images. Only three breeds had top-1 accuracy rates below 30%. The CNN achieved top-1 accuracy rates of at least 60% but less than 80% for about half of the breeds in the dataset. About one-eighth of breeds have top- accuracy rates of at least 80%. (return to text)

Figure 5.1.c



Top-1 accuracy rates by breed achieved by ResNet50 on the test images. Only about five percent of breeds saw top-1 accuracy rates below 50%. Nearly 30% of breeds witnessed top-1 accuracy rates between 50 and 70%. The remaining 65% of breeds reported top-1 accuracy rates of at least 70%. (return to text)

Figure 5.1.d



Top-1 accuracy rates by breed achieved by VGG-16 on the test images. Only about one eighth saw top-1 accuracy rates below 60%. There's a relatively even 25% of breeds that fall in between the 60 - 70%, 70-80%, and 80-90% tranches of top-1 accuracy. About nine breeds of top-1 accuracy rates of at least 90%. (return to text)