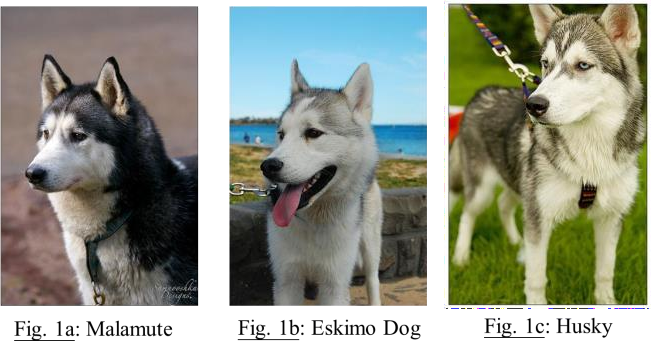
**Abstract**

In the present world, we have wide varieties of species and organisms. This brings into light, the criticality of classification of various Tangible objects. Also, keeping in mind, the ongoing research on genetics and evolution by various scientists across the world, discerning the resemblance among different classes also becomes very crucial. This paper is based on a project which builds a CNN (Convolutional Neural Network) to classify different dog breeds. If the image of a dog is found, this algorithm would find the estimate of the breed. The resembling dog breed is identified if the image of a human is supplied. We have built a pipeline to process real-world images.

**Introduction**

Convolutional neural networks (CNN) have been used to great effect in applications such as object classification, scene recognition, and other applications. In many situations, we can imagine the features (both low-level and higher-level) that are learned by the CNNs in the process of training. However, when the objects the CNNs are trying to categorize share many similar features, such as the breeds of dogs, it becomes hard to imagine the specific features that CNNs must learn in order to categorize these dogs correctly. This is especially true if we take a look at sets of images such as Fig. 1 below, where the 3 dogs share almost all the same visible features, but belong to different classes. It is therefore interesting to see how well CNNs can perform on only dog breeds, compared to labels from all classes of objects in the regular ImageNet.



**Network used:** CNN(Convolutional Neural Network)

Experiment

The Stanford Dogs dataset is an open-access image dataset of dog breeds. There are a total of 120 classes of dogs, with 20580 images in total, partitioned into 8580 test images, and 12000 training images. The image dataset comes with annotations that mark out the bounding boxes which best include the dogs in the image. The dimensions of both the bounding boxes and the original images vary across the board, and the scenes are non-uniform in style within one single class, with occlusion, different poses, different background objects, different colors of fur.

The first step in the learning process is to create a usable set of images for training and testing from the raw image files. The first thing that I did is to crop all images using the annotated bounding boxes. The next step is to resize all resulting images to 256x256 for training and testing purposes. For this, I decided against simply brute-force resizing it because I didn’t think that filters would be able to evaluate a squished or stretched image in the same fashion. Therefore, I decided to throw out all cropped images which had one of the two dimensions under 256 pixels, and for the remaining images, I would resize both dimensions down equally until the smaller dimension was 256, before taking rows and columns [1:256] from the image. I realize that repeated random sampling of rows and columns from the image can create an augmented dataset which should improve training, but for this project I did not do that. After this preprocessing, I ended up with 5678 training images, and 4007 testing images, which I converted into LMDB

format for use with Caffe. The overall image resizing steps are shown in Fig. 3 below.



Fig. 3: Image Resizing Steps Taken

The learning framework/software that I decided to use was Caffe. There were many reasons for this. First, the class encourages students to learn Caffe, which is an open-use software package that has an optimized tradeoff between ease of use and speed. Also, porting the training from CPU to GPU is very easy, as it only involves setting a flag. The prototxt format is also quite easy to understand and make work. However, I had many issues getting Caffe to work for my project. I started off installing Caffe in Ubuntu on Virtualbox, and running the training on the CPU in the virtual machine. However, I abandoned the effort soon after I realized that this would not go anywhere, as the machine would be able to do only 500 or so iterations over 12 hours. The training went much faster after I moved to Terminal and used their GPUs to train, but I had lost a large chunk of time in the process.

All of the results from the testing of the nets are summarized in Table 2. The testing accuracy are based on the cropped and resized testing dataset, using Top-1 accuracy. All learning was done using learning rate of 0.0005, momentum of 0.9, weight\_decay of 0.0005. All CONV layers were regularized with the weight\_decay. The hyperparameters were determined through 10+ fine-tuning iterations using different parameters from the parameter space to determine the optimal one for the problem as I have defined it.

I first started testing with LeNets, going from 3 layers of CONV-RELU-POOL, all the way to 6 layers. I experimented with varying the convolutional filter depths (50 and 500) along the way, while sticking to a filter size of 5x5. The accuracy results I obtained after 100,000 iterations were not good (< 2%) when I have less than 6 layers with 1 fully-connected layer, and when I increased the network to be 6 layers with 2 fully-connected layers, it went up to 9.4% accuracy.

For my testing with GoogLeNets, I used a similar approach with the number of layers, going from 3-7 layers. I kept the size of the filters per layer the same as in Fig. 2, and only varied the depths of the filters in my testing.

One thing I noticed early on with training is that the quality of convergence of the nets vary wildly depending on the learning rate. For example, if I use a learning rate of 0.0001, even at 100,000 iterations, the accuracy of the model will still be less than 2%, while if I use a learning rate of 0.0005, the model converges quite well to the maximum accuracy, and a learning rate of 0.001 causes the loss to blow up within 2000 iterations.

I visualized some filters below from one example of LeNet that I trained. Unfortunately I experienced a weird bug in pycaffe when I tried to visualize some of the larger LeNets or GoogLeNets. In these situations, if I load the model with pycaffe, all the weights would appear to be NaN’s and the test accuracies would appear to be uniformly distributed across all classes, but when I use the caffe –test interface in terminal, it would give an accuracy with a low testing loss (~0.5) as well as the accuracy it displayed while testing (~9%). I am not sure why this would be the case, and I wonder if my computer has memory issues which make it unable to load larger nets. Therefore, the visualization below is only from a 3-layer LeNet, with depth-50 filters, which achieved below 2% accuracy during testing. Although I would have preferred a visualization of a CNN that did better, I figured that it was better to have some visualization than none at all.

The visualizations below show the original image (Fig. 4) that was fed into the filters, the first and second layer filters themselves, and outputs from the first and second convolutional layers. The first layer filters (Fig. 5) are low enough in number, with small number of channels (3 channels for RGB since they are the first layer), that we can visualize them by simply combining the 3 channels and representing them in RGB format. Therefore, the first layer filters are shown in color, and there are 50 of them, representing a depth of 50. The outputs from the first layer Fig. 6 are plotted in grayscale, and show the result of convolving the image below with the filters. To make it convenient for viewing, I am only showing the outputs from the first 36 channels, so the resulting image matrix is square. We can already see some interesting results from the output of the first layer. Firstly, in some of the outputs, the contrast between the white and black portions of the dog are maintained (though they may be reversed so that black becomes white and vice versa), but in other outputs there are barely any differences between the two. We can see that in the 50 inputs to the second layer for this particular image, some inputs will have barely any signal and contrast, while others will maintain their strong contrasts for edge detections and other non-linear operations.

The second layer filters, while visualized in Fig. 7, are not really able to be viewed in an easy manner. This stems from the fact that there are 50 input channels per filter, and there are 50 filters in the layer as well. The Caffe tutorial suggests that these filters be plotted in a row-column matrix format,

where every row indicates all of the input channels of the filter, and the column represents the depth of the filters. do note here that the outputs from the second layer (Fig.8) are not too different from the first layer. This is a good representation of a network that is not well-learned, because there is not much transition of hierarchy of low-level to high-level features as we move through the levels of the network.

Fig. 4: Original Image for Visualization

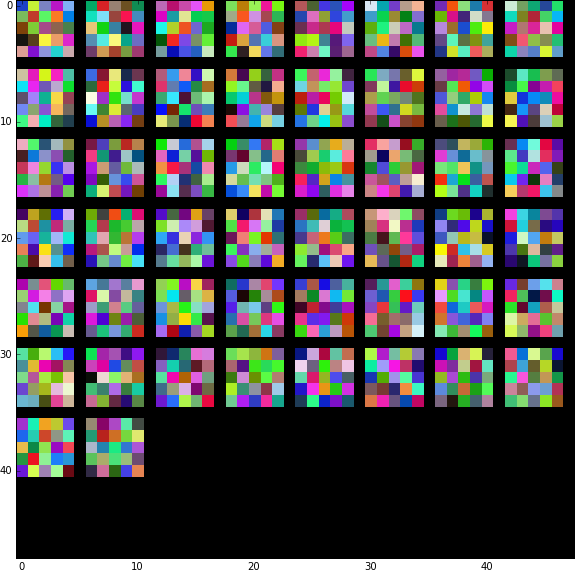


Fig. 5: Visualizing First Layer CONV Filters

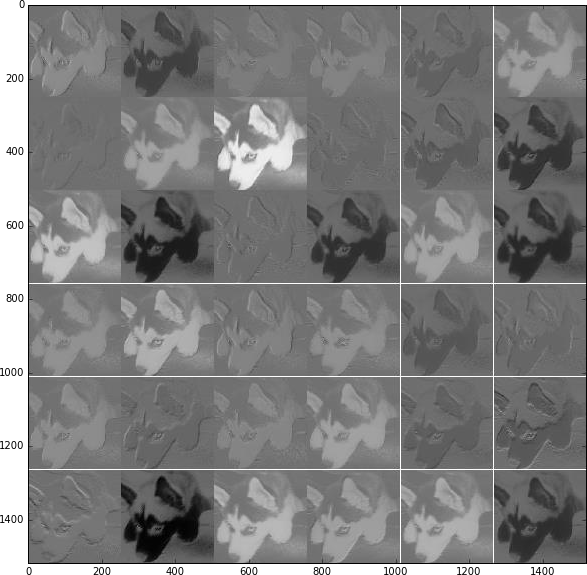


Fig. 6: Output from First CONV Layer

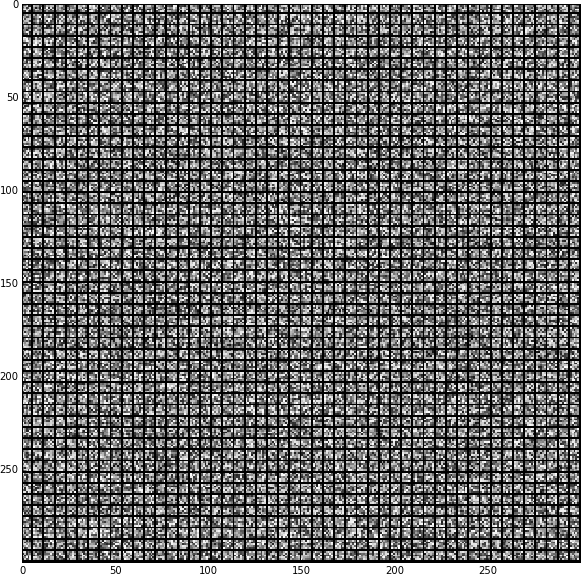


Fig. 7: Visualizing the Second CONV Layer

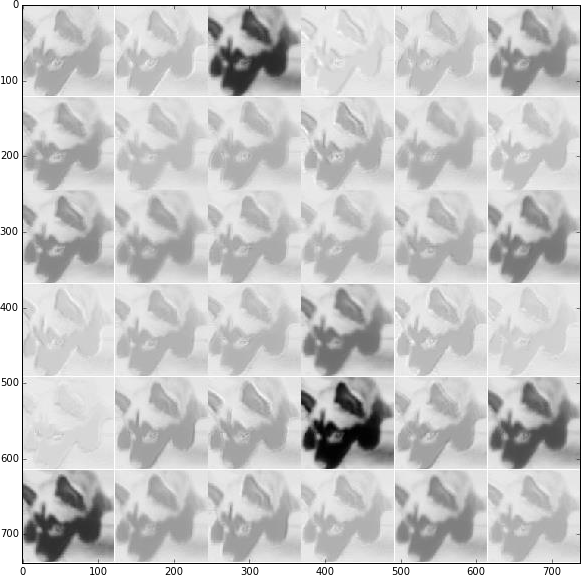


Fig. 8: Output from the Second CONV Layer

**Layers in the Network:**

Model: "sequential\_7"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

conv2d\_29 (Conv2D) (None, 60, 60, 128) 9728

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d\_29 (MaxPooling (None, 58, 58, 128) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dropout\_35 (Dropout) (None, 58, 58, 128) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

batch\_normalization\_29 (Batc (None, 58, 58, 128) 512

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_30 (Conv2D) (None, 56, 56, 64) 73792

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d\_30 (MaxPooling (None, 54, 54, 64) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dropout\_36 (Dropout) (None, 54, 54, 64) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

batch\_normalization\_30 (Batc (None, 54, 54, 64) 256

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_31 (Conv2D) (None, 52, 52, 32) 18464

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d\_31 (MaxPooling (None, 50, 50, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dropout\_37 (Dropout) (None, 50, 50, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

batch\_normalization\_31 (Batc (None, 50, 50, 32) 128

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_32 (Conv2D) (None, 48, 48, 32) 9248

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d\_32 (MaxPooling (None, 46, 46, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dropout\_38 (Dropout) (None, 46, 46, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

batch\_normalization\_32 (Batc (None, 46, 46, 32) 128

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_33 (Conv2D) (None, 44, 44, 32) 9248

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d\_33 (MaxPooling (None, 42, 42, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dropout\_39 (Dropout) (None, 42, 42, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

batch\_normalization\_33 (Batc (None, 42, 42, 32) 128

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

global\_average\_pooling2d\_1 ( (None, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_12 (Dense) (None, 10) 330

=================================================================

Total params: 121,962

Trainable params: 121,386

Non-trainable params: 576

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Accuracy score:**

**Confusion matrix:**

**Conclusion:**