

Exploring Computational Efficiency in Object Detection with Convolutional Neural Networks and Tensorflow

Scott Sims

Table of Contents

1 INTRODUCTION.....	3
2 ANATOMY OF AN IMAGE.....	3
3 THE DATA.....	5
4 Basics of a Convolutional Neural Network.....	7
4.1 Edge Detectors.....	8
4.2 Padding.....	9
4.3 Pooling Layer.....	9
4.4 Striding.....	11
5 MODEL ARCHITECTURE.....	13
5.1 CONVOLUTIONAL LAYER OUTPUTS.....	14
6 RGB BASELINE.....	16
6.1 VANISHING GRADIENTS.....	16
6.2 BASELINE PERFORMANCE.....	18
7 GRayscale Model.....	20
7.1 GRAYSCALE PERFORMANCE.....	21
8 HISTOGRAM EQUALIZATION.....	23
8.1 EQUALIZATION PERFORMANCE.....	25
9 TRAINED EQUALIZED MODEL.....	26
9.1 EQUALIZED MODEL PERFORMANCE.....	27
10 RESULTS.....	28
10.1 EXTRA CONSIDERATIONS.....	28
11 Conclusion.....	29
APPENDIX A – Data Processing Class.....	31
APPENDIX B – RGB Tensorflow Class.....	32

1 INTRODUCTION

It seems like we are always sitting in front of screens these days. Computer screens, movie screens, phone screens, etc. are just some of the daily few we interact with. But what if we could do more than just watch? What if we maximized the business and personal interactivity potential with our screens? How many times have you been watching a TV show or movie and loved a character's outfit, car or some kind of gadget? Don't you wish you could just touch it on the screen and your web browser opens up amazon with that exact item for purchase. For the business oriented people, wouldn't you like to maximize the potential of your ads reaching your target customer? Imagine marketing that can "see" what your customer's are watching and only show them ads or products relative to their viewing likes. A person who is always watching football, baseball, and sports related movies might be more interested in a commercial advertising the current sale going on at Dick's than they would at Sephora. Targeted demographic marketing and other capabilities could be improved exponentially if our screens could see the things we are seeing and know what they are. This would need to grow from the foundation of accurate object detection in Video.

This was the original basis for this project. However when initially getting started, an ugly problem reared its head. Training object detectors is an extremely computationally expensive endeavor. Training even just a 3 layer CNN on 3,000 pictures took over 10 hours on a more than capable machine. With the need to try several different combinations of architectures, hyper-parameters, and number of iterations, this didn't start off as a feasible project. From necessity spawns innovation. And with this, comes this investigation into training optimization of Convolutional Neural Networks. All working code with step by step walk through and thought process can be seen in this [Jupyter Notebook](#) on my [Github Repository](#). There's a lot of low level Tensorflow and Object Oriented Programming.

2 ANATOMY OF AN IMAGE

When we render images or stream video on a screen, we see something like this:



Illustration 1: Image of an Owl used in CNN Training

When we look at this, we clearly see the outline, texture, features, and shapes of an owl. When a computer looks at this image, it sees an array of numbers representing pixel intensities.

```
[[ 9  1 29 70 114 76  0  8  4  5  5  0 111 162  9  8 62 62]
 [ 3  0 33 61 102 106 34  0  0  0  0  49 182 150  1 12 65 62]
 [ 1  0 40 54 123 90 72 77 52 51 49 121 205 98  0 15 67 59]
 [ 3  1 41 57 74 54 96 181 220 170 90 149 208 56  0 16 69 59]
 [ 6  1 32 36 47 81 85 90 176 206 140 171 186 22  3 15 72 63]
 [ 4  1 31 39 66 71 71 97 147 214 203 190 198 22  6 17 73 65]
 [ 2  3 15 30 52 57 68 123 161 197 207 200 179 8  8 18 73 66]
 [ 2  2 17 37 34 40 78 103 148 187 205 225 165 1  8 19 76 68]
 [ 2  3 20 44 37 34 35 26 78 156 214 145 200 38  2 21 78 69]
 [ 2  2 20 34 21 43 70 21 43 139 205 93 211 70  0 23 78 72]
 [ 3  4 16 24 14 21 102 175 120 130 226 212 236 75  0 25 78 72]
 [ 6  5 13 21 28 28 97 216 184 90 196 255 255 84  4 24 79 74]
 [ 6  5 15 25 30 39 63 105 140 66 113 252 251 74  4 28 79 75]
 [ 5  5 16 32 38 57 69 85 93 120 128 251 255 154 19 26 80 76]
 [ 6  5 20 42 55 62 66 76 86 104 148 242 254 241 83 26 80 77]
 [ 2  3 20 38 55 64 69 80 78 109 195 247 252 255 172 40 78 77]
 [10  8 23 34 44 64 88 104 119 173 234 247 253 254 227 66 74 74]
 [32  6 24 37 45 63 85 114 154 196 226 245 251 252 250 112 66 71]]
```

Illustration 2: Matrix of Pixel Intensities

The above illustration is a matrix representation of what a computer sees when rendering an image. This 18x18 matrix contains values ranging from 0 to 255. The higher the value, the more intense the pixel is. These 324 pixels however are representative of only a single color channel. A typical color image has 3 color channels; One for red, one for green, and one for blue.

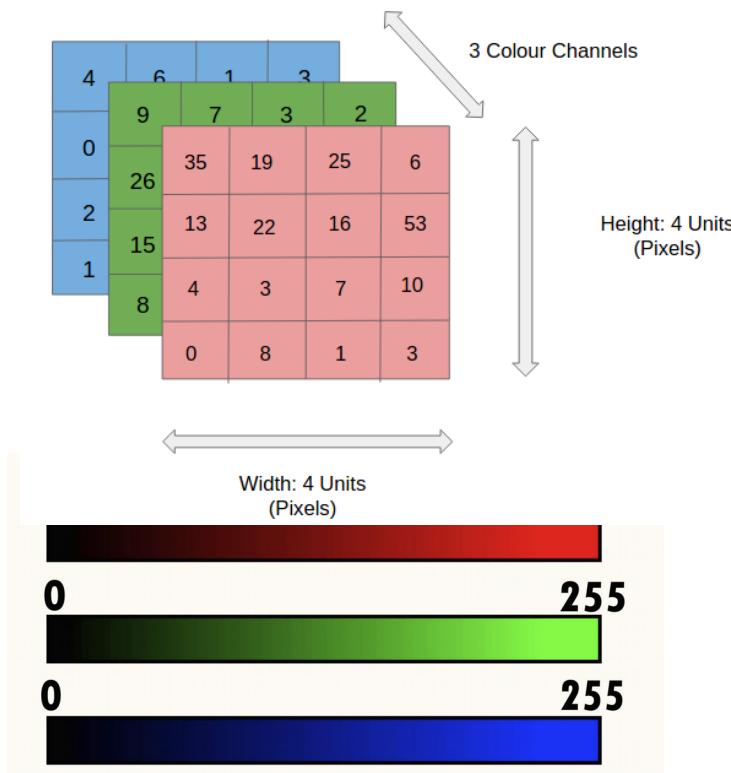


Illustration 3: RGB Channels and Pixel Intensities

One can start to see what the potential computational costs can add up to be. From this mere 18x18 image, there are a total of 972 pixels amongst the 3 color channels. When represented as a float 16 or float 32 matrix, meaning each number is represented as a 16 or 32 bit number respectively, this one image takes up 1.9 and 3.9 kilobytes respectively. Later on, it will be further expanded on how this issue compounds exponentially.

3 THE DATA

Data was chosen from the ILSVRC images database. The ILSVRC or ImageNet Large Scale Visual Recognition Competition, simply known as ImageNet, is a project that curates, labels, and distributes a large visual database for yearly competitions. Their database contains over 20 thousand categories where each category contains several hundreds to thousands of pictures. The images in this study are specifically from the 2013 database, containing the categories of:

- n01503061 – Bird
- n02099602 – Dog
- n02958343 – Car
- n04330267 – Stove

No specific thought process went into the selection of categories for this study other than the desire to have at least two organic objects and two inanimate objects. For the bird category, there are 1,445 pictures, 1,069 for the dog category, 839 car pictures, and 1,196 Stove pictures. Of the 4,549 images, the average image height is 387 pixels and the the average image width is 454 pixels. This brought about the first real decision of the project.

Convolutional layers don't require that the input image be of a fixed size. However the architecture of the CNN contains at least one fully connected layer (synonymous with the hidden layer of a neural network) and the input for this layer must be as expected. Because of this, as an image is fed forward through the CNN, in order to assure that the resultant image size is as expected for the fully connected layer, the images themselves must be reshaped to a standard size. This issue can be circumvented using a method called Spatial Pyramid Pooling. But in order to control the complexity of this study, this method was not used. The paper on Spatial Pyramid Pooling can be read [here](#). With an average pixel height and width of 387 and 454 respectively, initially the images were to be rescaled to 500x500. This was the first computation road block encountered.

By reshaping all images to 500x500, this consequently would create a 4549x500x500x3 array of pixel values with data type, float 32. At 32 bits per integer, this single array would be 13.6gb. Although this is doable, this is not computationally desireable. As a result, the decision was made to reshape the images to 300x300x3. This dimension would reduce the array size to 4.9gb, limit the distortion of the original image as well as prevent them from being too grainy from over enlargement that the 500x500 dimension would cause.

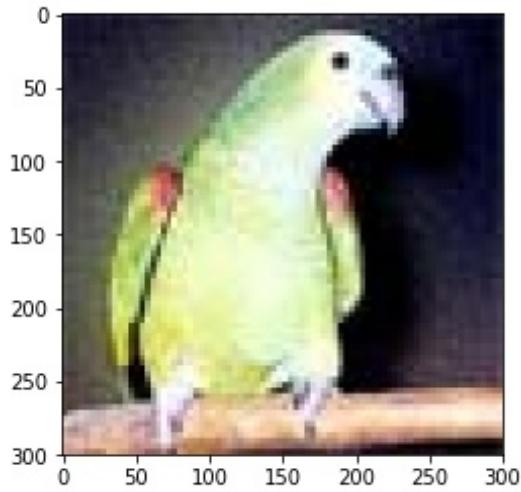


Illustration 4: Image of original size 80x80 enlarged to 300x300 to represent the amount of graininess reshaping images can cause

A data processing class was created in order to load all the images from disk, vectorize them into numpy arrays, and reshape them. After processing the data, it is then saved to disk via a H5 file. This class can be seen in [Appendix A](#). This class performs an 85/15 train test split on the data before saving it to H5. The split of all categories breaks down as shown in Illustration 5.



Illustration 5: Train Test Split of all data for the study

4 Basics of a Convolutional Neural Network

A convolutional layer of a CNN, also known as a filter, is a $(m \times n)$ matrix that is overlaid an $(M \times N)$ image. This filter's size is determined by the CNN architect, and could be viewed as a hyper parameter that is tuned during training. The filter is typically smaller than the image and convolves around the pixel matrices. The general idea of this can be seen below in illustration 6.

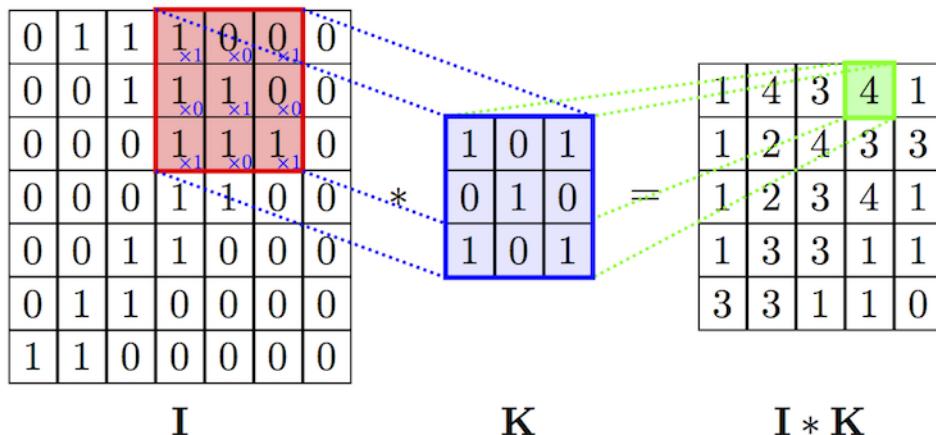


Illustration 6: Applying a convolutional filter

The top left corner of filter K is initially aligned with the top left corner of image I. The values of I are the pixel values for the image. The overlaid filter values are then multiplied with their corresponding pixel cell, and then all the values are summed together. The filter is then slid over by a predetermined amount of columns and the operation starts again. Once the filter has gone as far right as possible, it then resets to the left and shifts down a predetermined amount of rows. It then repeats the same process until it has convolved on the whole image. As shown, the resulting output shrinks the image by a factor of:

$$(n_H - f_H + 1) * (n_W - f_W + 1)$$

where

- n_H & n_W is the pixel height and width of the image respectively
- f_H & f_W are the height and width of the filter respectively

As seen in Illustration 6, the image is 7x7 with a 3x3 filter being applied. Therefore the resulting output is $(7-3+1) \times (7-3+1) = 5 \times 5$ resulting output.

So what is this filter doing? It is basically an edge detector/feature extractor. Since the filter is multiplying the pixel values and summing the result, the areas with high pixel intensity will be amplified and areas with low pixel intensity will be attenuated. The orientation of this amplification/attenuation depends on the orientation of the filter.

4.1 Edge Detectors

Let's look at a Vertical Edge Detector example taken from Andrew Ng's CNN deep learning Course for an illustration of what's going on.

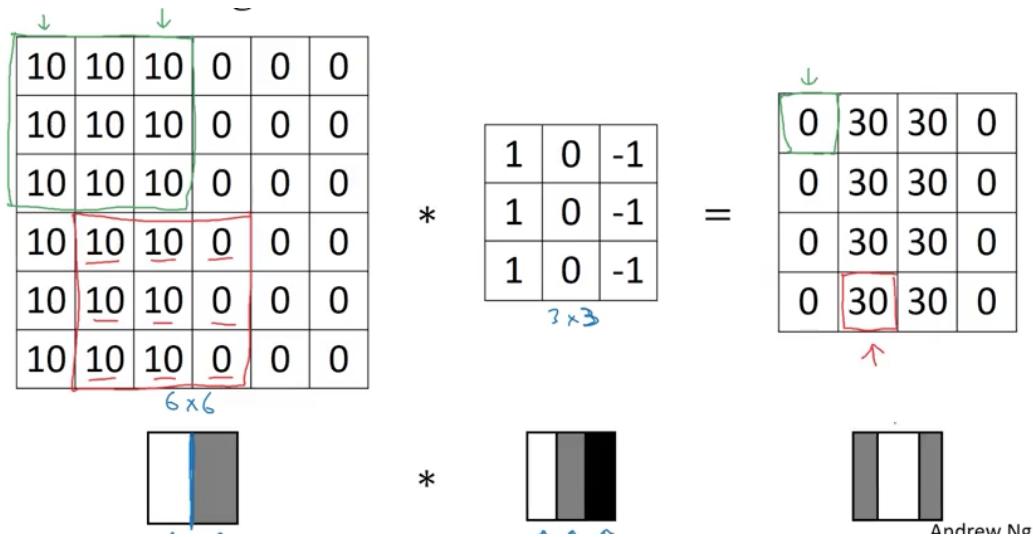
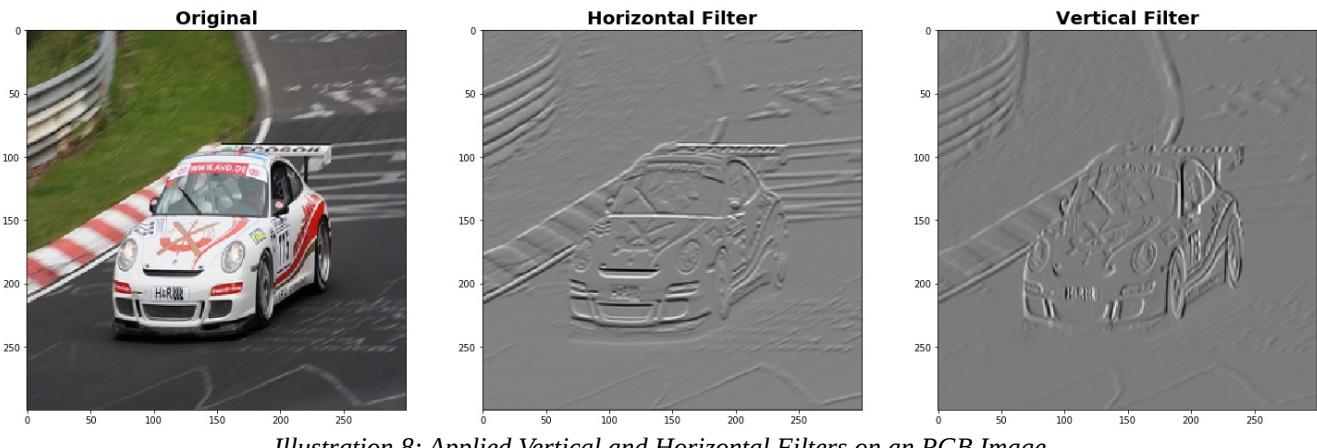


Illustration 7: Vertical Edge Detector as Explained by Andrew Ng; Convolutional Neural Networks - Deep Learning Specialization

The first 3 columns have pixel intensity 10 and the last 3 have pixel intensity 0. This results in the image seen below it. The left side of the image is white where the right side of the image is gray, with an obvious edge boundary between the two. As the filter convolves around the image, it reduces to pixel values to 0 everywhere except for where the edge existed between the two pixel intensities. This filter is a vertical edge detector. If this filter was rotated 90 degrees, it would then become a horizontal edge detector. When these filters are convolved around an actual RGB image, the following is produced:



After applying the horizontal and vertical filters on the image, it is shown how these filters highlight those edges which appear horizontal and vertical for each respective filter. All other edges, patterns,

and shapes are attenuated. The images are fed in as 300x300x3 array. Notice that the output of these images retain this 300x300 shape. This is due to padding.

4.2 Padding

Padding is the act of taking our ($M \times N$) image (300x300) in this case and adding extra rows and columns of 0 values around it. This is done because when convolving the filter around the image, the middle pixel values are involved in several instances of the convolutional process. The pixels on the sides however are convolved on less frequently due to their position at the borders. Because of this, a lot of information at the borders are lost. By adding 0 padding, we are able to convolve on these border pixels more often, thus preserving information at the borders. Padding also serves as a way to control output of the convolutional layers. If we need a specific output size, we can ensure we get it by incorporating padding. This process is highlighted in illustration 9 which 0 pads a 32x32 image.

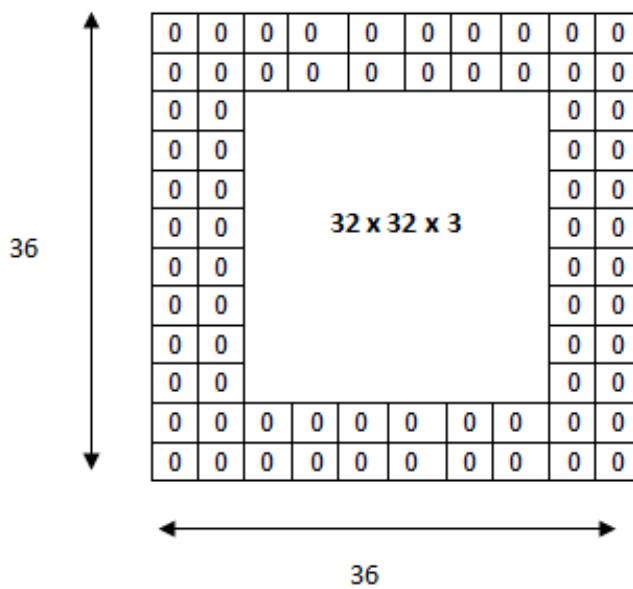


Illustration 9: Zero Padding for CNNs

When applying a 5x5 filter to this 32x32 image, the resulting output after adding the 4 columns and rows of zeros, remains 32x32. Padding however is not the only alteration/transformation that is typically applied to a fed image.

4.3 Pooling Layer

A very common practice after feeding an image into a convolutional filter is to then feed that output into what is known as a “Pooling Layer”. This layer acts as yet another filter. There are two types of pooling layers; Max pooling and Average pooling. Max pooling will be discussed here, but the idea of average pooling can be inferred from the explanation.

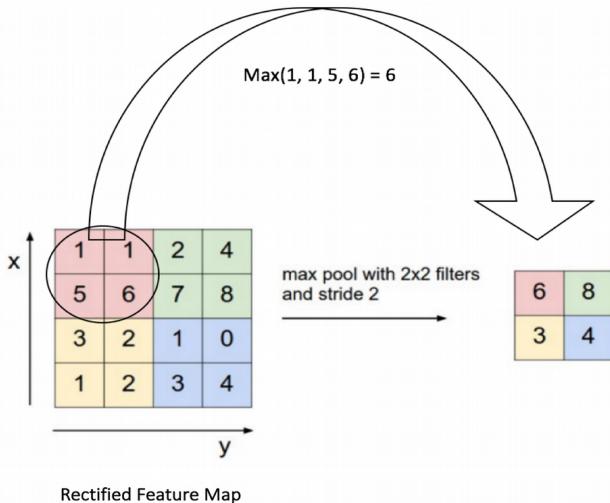


Illustration 10: Example of a 2x2 Max Pooling Filter

A max pool filter simply overlays a filter on the image as shown above and simply returns the largest value in the filter. The filter is then slid over by a predetermined amount of cols and the step is repeated. This pooling layer is used for 2 different reasons

1. It reduces the dimensionality of the data. The pool layer is a non trainable layer that simply takes the highest value in an array. As seen above we are reducing the size of our image (in that example) by 75%. This will have great computational effects during training. However there is an inherent issue with this. We are throwing away 3/4 of our data.
2. The model needs to be invariant to small perturbations in the data. The model shouldn't be trained so precisely that it looks for an exact pixel in an exact location. It should generalize well to any image with the object of interest in any orientation, size and shape. The pooling layer injects this variance into the model. The convolution layer tries to identify some important structure in some region of the data while the pooling layer obscures the exact location of this structure.

After applying a pooling layer to the original images with the edge detectors, it results with effectively the original image, but at compressed dimensionality.

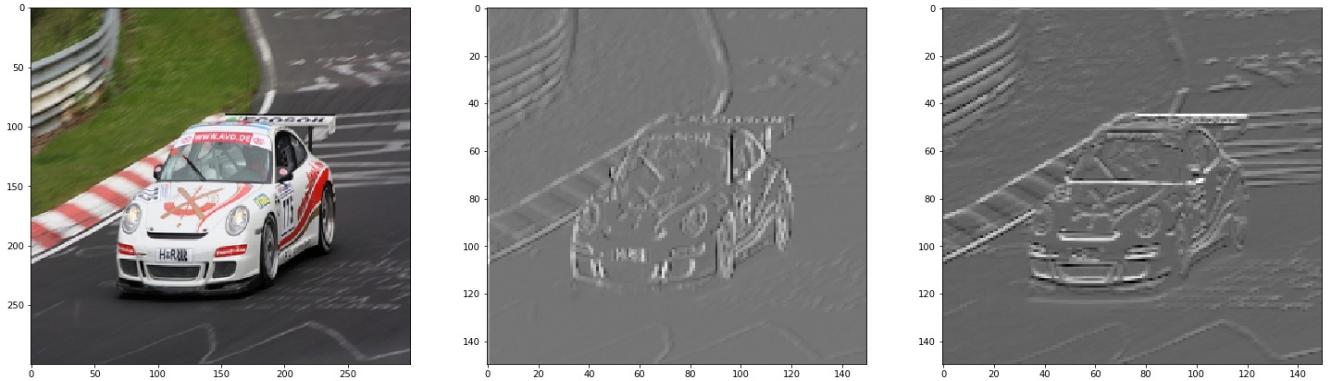


Illustration 11: Image filtered through a Edge Detector Convolutional Filters and a Max Pool Layer

The results from the convolutional layer is fed into the pooling layer where the empirical features and edges that were extracted are preserved, but the location of these features have changed. The output size of the images are now 150x150.

4.4 Striding

Until now the intuition has been that the filter is slid in each direction by only 1 column or row. This doesn't have to be the case. It can be increased how many positions the filters move by designating the "strides" of the filter. Illustration 13 shows the implementation of a 3x3 filter with a stride of 2.

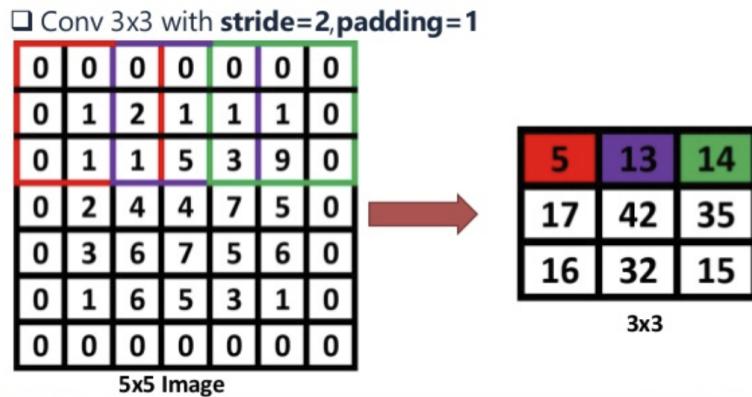


Illustration 12: Implementing strides on a filter. It can be seen that the purple filter starts at column 3 instead of column 2, representing a stride of 2 after the red filter

It is not uncommon in more recent CNNs to remove the use of a pool layer all together in favor of just increasing strides on the convolutional layer. It serves the same purpose of introducing variance into the model, but does have the downside of bearing weights that need to be saved in memory and trained. This can make it computationally expensive in comparison to a pooling layer. With the incorporation of padding and stride, the resultant output shape of an image function needs to be updated to account for these transformations.. With both padding and strides, the resulting image size is now:

$$\frac{n_H + 2p - f_H}{s} \times \frac{n_W + 2p - f_W}{s}$$

where:

- p is padding. It is $2*p$ because if you want one layer of padding, a layer of zeros is added to the left and right side of the image for vertical axis, and a layer of zeros is added to the top and bottom side of the image for horizontal axis.
- s is the number of strides for the filter to take while convolving around the image

And this is the basic premise of a CNN. Filters convolve around an image searching for features and edges. The resulting convolution sometimes is then fed through a pooling layer which is a non trainable feature reduction layer. These convolution layers and pooling layers are stacked together to create the architecture of a CNN. The vertical and horizontal filters were specifically chosen for this basic introduction. But in the context of a real CNN, filters are randomly initialized and learned during training (excluding the pooling layer). Each filter learned will be responsible for searching for specific features in various orientations of an image. This wasn't meant to be an in depth lesson on CNNs, and as a consequence, a lot of intuition is left to the reader.

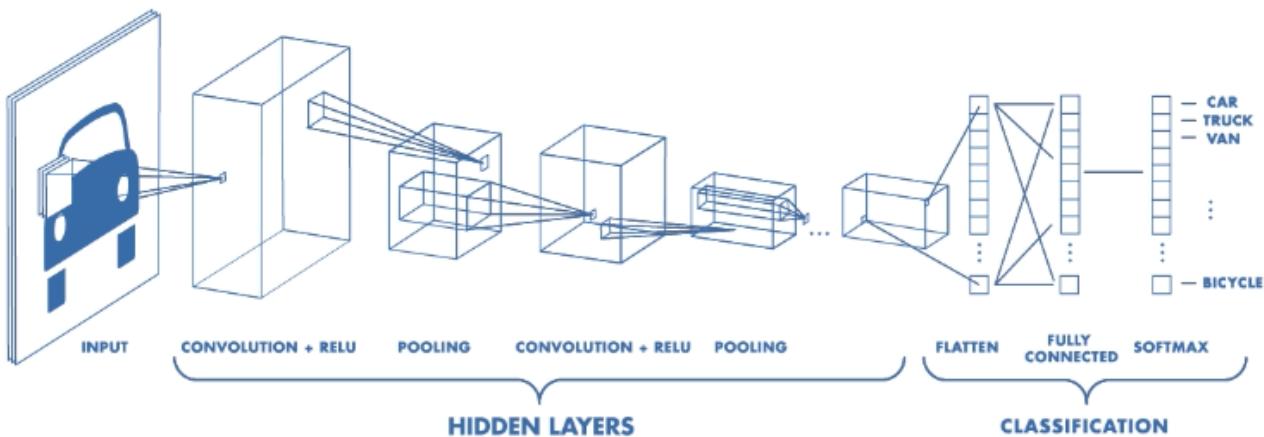


Illustration 13: A full Convolutional Neural Network (CNN)

5 MODEL ARCHITECTURE

A few different CNN structures were rapidly prototyped in order to determine initial performance. Ultimately a 4 layer model was settled upon in order to conduct the optimization study. The input layer is obviously the $? \times 300 \times 300 \times 3$. The “?” is a placeholder in tensorflow signaling the model that there will be a certain amount of training samples passed to the model determined at runtime. However, all other dimensions will be set and should be expected as such. The first convolutional layer contained 10 different convolutional filters followed by a pooling layer. The second layer also contained 10 different convolutional filters. This time it neglected the use of a pooling layer. Finally, the third layer was constructed using 20 different filters. The third layer did make use of a pooling layer. After the final convolutional layer, the outputs are fed into a fully connected layer with the 4 output nodes representing one of the 4 different categories. The logits from this fully connected layer are then fed into a softmax function. A full breakdown of the CNN architecture is as follows

Layer	Filters	Filter Shape	Strides	Padding	Variables	Trainable
1st Convolutional Layer	10	7x7	2	SAME	493	Yes
1st Pooling Layer	1	4x4	2	VALID	NONE	No
2nd Convolutional Layer	10	2x2	2	SAME	170	Yes
3rd Convolutional Layer	20	3x3	2	SAME	190	Yes
2nd Pooling Layer	1	2x2	2	VALID	NONE	No
Fully Connected Layer 1	N/A	N/A	N/A	N/A	720	Yes

A quick note on the architecture. Where padding says “SAME”, this simply means that whatever the input shape is of the incoming data, it will be zero padded so that the output will still be the same shape. At least it would be the same shape if a stride of 1 was implemented. However since a stride of 2 is used, the resulting shape won’t be quite the same shape as the input. Where padding says “VALID”, this means no padding is implemented and the resultant shape will follow that of the equations mentioned in section 4.4 with a value for “p” of 0.

In relation to the structure of the graph, Tensorflow has built in functionality that allows one to monitor training realtime with Tensorboard. It creates the graph in the direction of which the tensors

flow. The W's are the filters for each convolutional layer that is learned during training. Following the input "x" of the graph at the bottom right, the following happens:

- Input Image (300x300x3) is convolved upon by 10 separate filters in the first layer (W1) and outputs a (75x75x10) tensor
- Second layer receives the output from the first layer and is convolved upon by another 10 separate filters (W2) and outputs a (38x38x10) tensor
- Third layer receives the output from the second layer and is convolved upon by 20 separate filters (W3) and outputs a flatten array of 720 values
- Fully Connected layer receives the flatten vectored and outputs a prediction

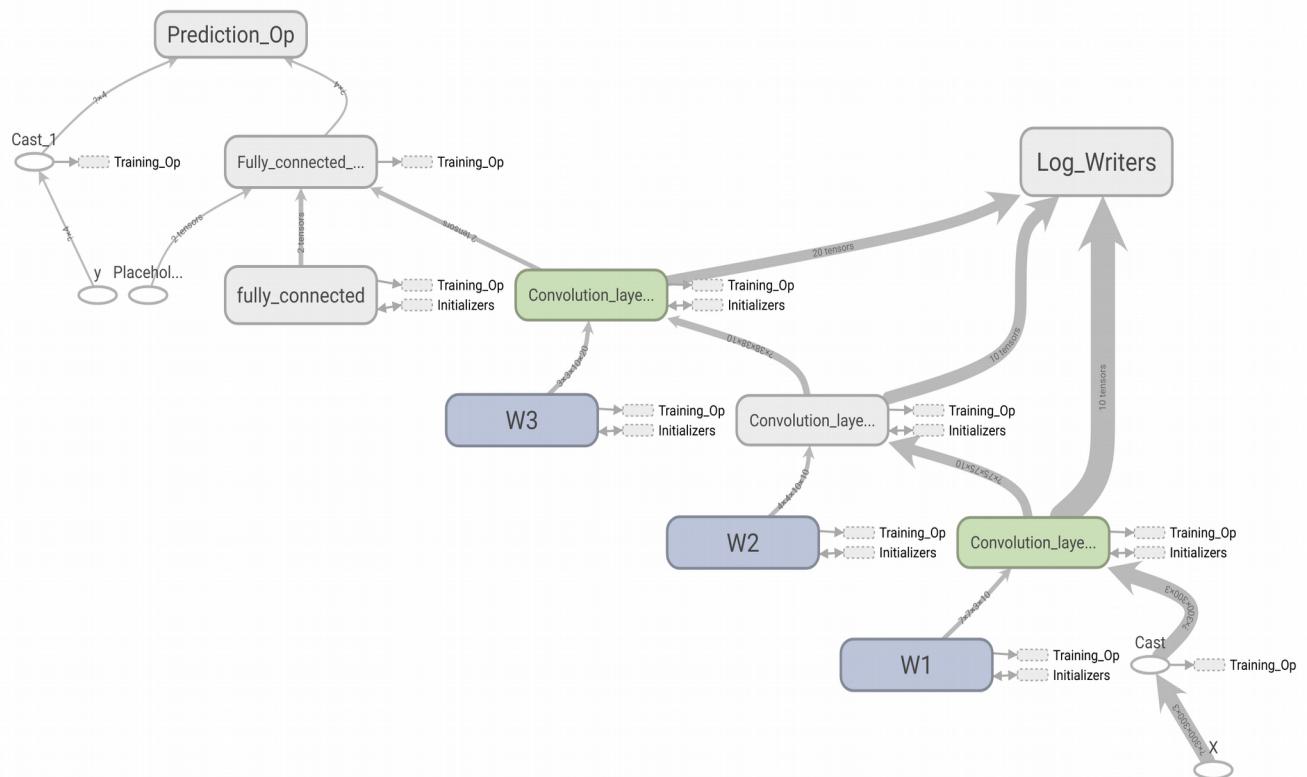
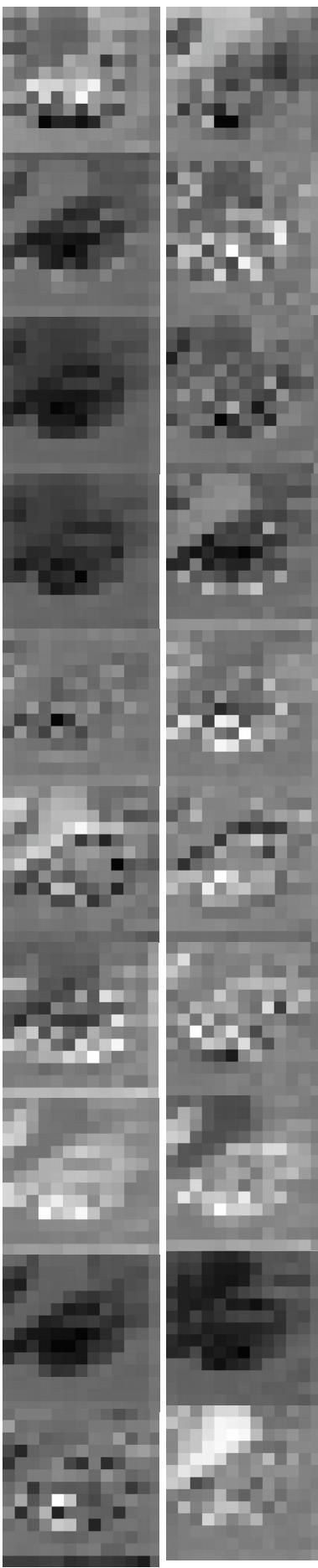


Illustration 14: Tensorboard Graph Representation of the Trained CNN Model

5.1 CONVOLUTIONAL LAYER OUTPUTS

The following images are an example of the filter outputs for all 3 convolutional layers. Column 1 is the first layer with 10 different filters, 2 is the second layer with 10 different filters, and column 3 is the third layer with 20 different filters. It can be seen that the first and second layers are more of the edge detecting layers whereas the third layer is in search of more of the abstract.

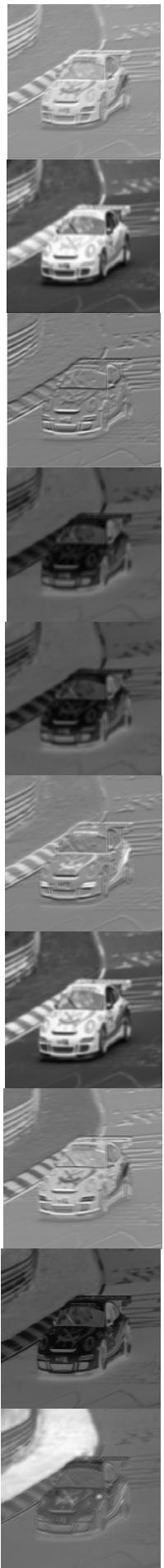
Third Convolutional Layer



Second Convolutional Layer



First Convolutional Layer



6 RGB BASELINE

Not only is memory space a primary concern when training CNNs, but also training speed. Training full RGB images took several hours of training. This is not uncommon in the world of object detection and computer vision algorithms. But the nasty downside to this is hyper-parameter tuning can take hours to days to really dial the model in. All tests were performed on a system with an 8-core Xeon processor, 16GB Ram, 1TB SSD HD, and Linux Ubuntu 16.04 LTS Kernel. In regards to this study's CNN, the typical hyper-parameters of learning rate and regularization constant were tuned. However, the optimizer for back propagation was Adam optimizer which comes with its own hyper-parameters. Finding a good initial baseline for the RGB model took 3 days due to training time, hyper-parameter tuning and overall CNN architecture. Large CNNs can learn very complex data, but at a tricky cost. Initial tested models produced extremely poor results even after hours of epochs. The problem encountered was that the model was big enough to the point that it suffered from vanishing gradients.

6.1 VANISHING GRADIENTS

A full comprehensive mathematical discussion on vanishing (and exploding) gradients are out the scope of this paper. It is merely brought up in this instance because it was an initial problem seen while trying to establish a baseline model. A CNN works in the same way as a traditional deep neural network. Data is fed forward through the network where a prediction is made. The error is calculated between the predictions and true values. The model is then traversed backwards performing back propagation to train the model on its “mistakes”. On sufficiently large enough models, when back propagation commences, the later layers in the model learn more quickly than the earlier layers. In the simplest explanation, this is due to multiple derivatives being multiplied by the derivative of an early layer. As the derivatives are taken, traversing backwards, they result in very small numbers ($<< 0$). Small numbers multiplied by small numbers equals an even smaller number. As a consequence, the earlier layers in the model are updated very slowly with these small numbers, thus, they learn even slower. This is a simple explanation of the vanishing gradient problem. To see a great step by step intuitive and mathematical explanation of this gradient problem, see this [article](#).

To combat this problem, a 4 layer network was discovered to work the most suitably in the days of searching for the best architecture. This deep of a model allowed all layers to learn at a suitable rate while being able to generalize to the complexity of the data. In conjunction with this, RELU activation functions were chosen for each convolutional layer instead of sigmoid or tanh due to it's ability to suppress the vanishing gradient problem. Sigmoids and tanh functions restrict the output of numbers in the activation layer to a value between 0 to 1 and -1 to 1 respectively. A consequence of this is the derivative of such small numbers results in even smaller numbers. Weight updating was monitored realtime during training to ensure neither vanishing nor exploding gradients were being realized via Tensorboard

During training, the weights are updated in a tensorflow log writer. This log stores information such as cost, weights distributions and histograms as well as image rendering through each layer of the CNN. The weights from the RGB Model are displayed on the right. Each update to the log is written after every 10th Epoch. The y-axis of the graphs represent the Epoch and the x-axis is the histogram distribution.

It can be seen that layer 3 weights are going through the most dramatic changes after each Epoch. Layer 1's updates appear less dramatic the longer the model learns. This exemplifies the potential pitfall of vanishing gradients as well as the explanation that earlier layers learn much slowly than later layers.

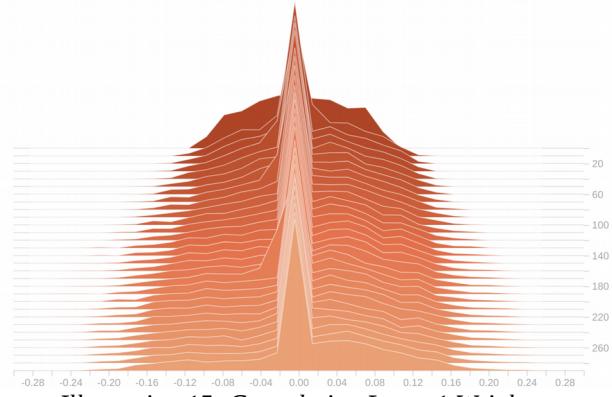


Illustration 15: Convolution Layer 1 Weights

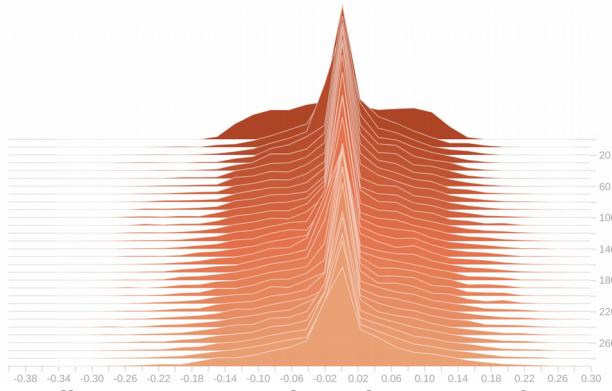


Illustration 16: Convolutional Layer 2 Weights

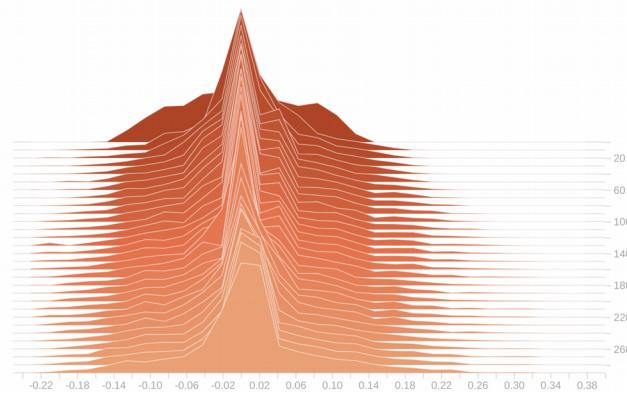


Illustration 17: Convolutional Layer 3 Weights

6.2 BASELINE PERFORMANCE

From the model as outlined in previous sections, the initial baseline performance was respectable, but took sometime to train. A custom Tensorflow class was constructed in order to facilitate the training, logging, and prediction methods of the model. This class can be seen in [Appendix B](#). The class implements early stopping for both increasing cost and floor value cost. If the cost increases 20 epochs, training is terminated. Conversely if the difference between $\text{cost}(t + 1)$ and $\text{cost}(t)$ is less than 0.0001, training is terminated. The results for RGB baseline after 300 Epochs are shown in illustration 16

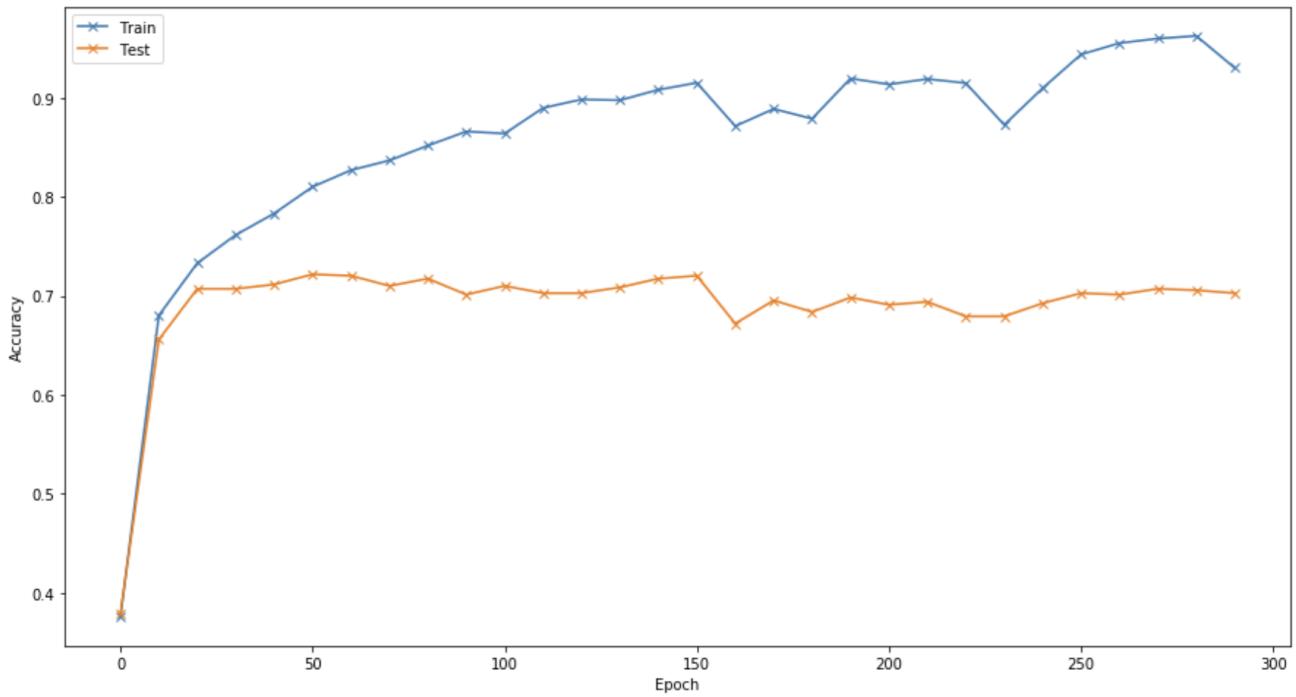


Illustration 18: Train and Test set accuracy for RGB CNN

The accuracy for the training set reached over 96% at Epoch 280 while accuracy for the test set reached 72% at Epoch 50. Afterwards it continue pretty flat during the rest of training. This is a sign of overfitting on the training set. But for the current purposes, it served as a valid baseline. Training spanned 300 Epochs and took 6.3 hours. This breaks down to 75.6 seconds per Epoch.



Illustration 19: RGB Training Cost With Respect to Time

During training, the model class saves a checkpoint of the model every 10th Epoch. This checkpoint saves all weights and biases for that Epoch. Further investigation was done on model metrics for both the best performing model on the train set and the best performing model on the test. Those results are as follows. The Train Set metrics are on the left and the Test Set metrics are on the right.



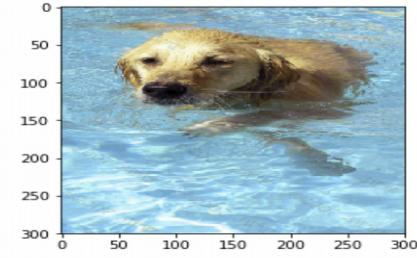
This is a Bird



This is a Bird



This is a Stove



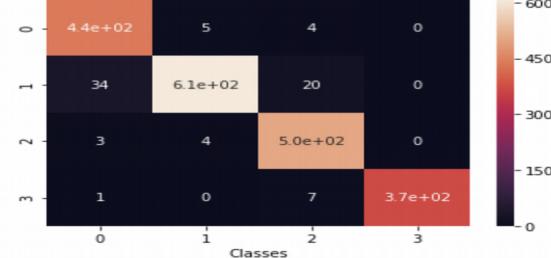
This is a Dog

Accuracy: 96.1%

Out[110]:

	F-Beta	Precision	Recall	Support
Dog	0.949079	0.920168	0.979866	447
Bird	0.950666	0.985390	0.918306	661
Stove	0.963740	0.942164	0.986328	512
Car	0.989362	1.000000	0.978947	380

Confusion Matrix



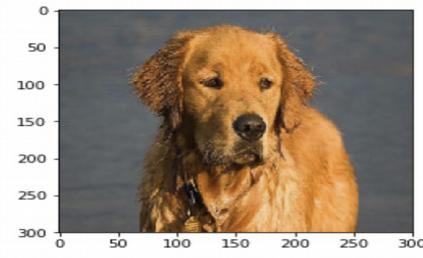
This is a Bird



This is a Car



This is a Stove



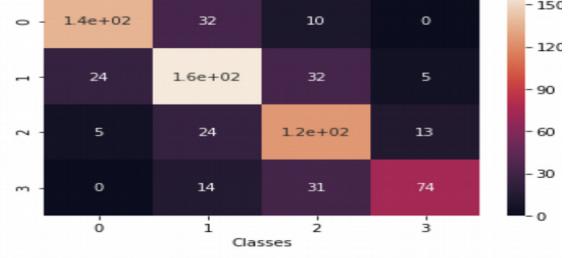
This is a Dog

Accuracy: 72.18155197657394%

Out[113]:

	F-Beta	Precision	Recall	Support
Dog	0.793003	0.824242	0.764045	178
Bird	0.706935	0.692982	0.721461	219
Stove	0.684932	0.631313	0.748503	167
Car	0.701422	0.804348	0.621849	119

Confusion Matrix



The best model on the training set performed amazing, with high precision and Accuracy for all 4 categories. It even resulted in perfect precision when classifying cars. The test set metrics aren't too bad either. It is a decent car and dog classifier. However, it did take 6.3 hours to run through 300 Epochs and clearly there is an overfitting problem. Typical object detectors are trained on tens of thousands of images. Only +3000 were used here. What is wanted is the ability to still achieve high accuracy while reducing training time. The first idea that came to mind was performing training on the same images, but only in Grayscale.

7 GRayscale MODEL

An image is converted from RGB to grayscale most commonly by taking the average of the 3 pixel color intensities and converting them to a single value. Therefore, we are reducing our size constraints for each image by 2/3. This is removing some of the "information" contained in the data, but since CNNs are mostly edge and shape detectors, the idea was hopefully the loss of information didn't translate to a loss of performance. The intrinsic shape of the object in the image is preserved, but the color is just removed.

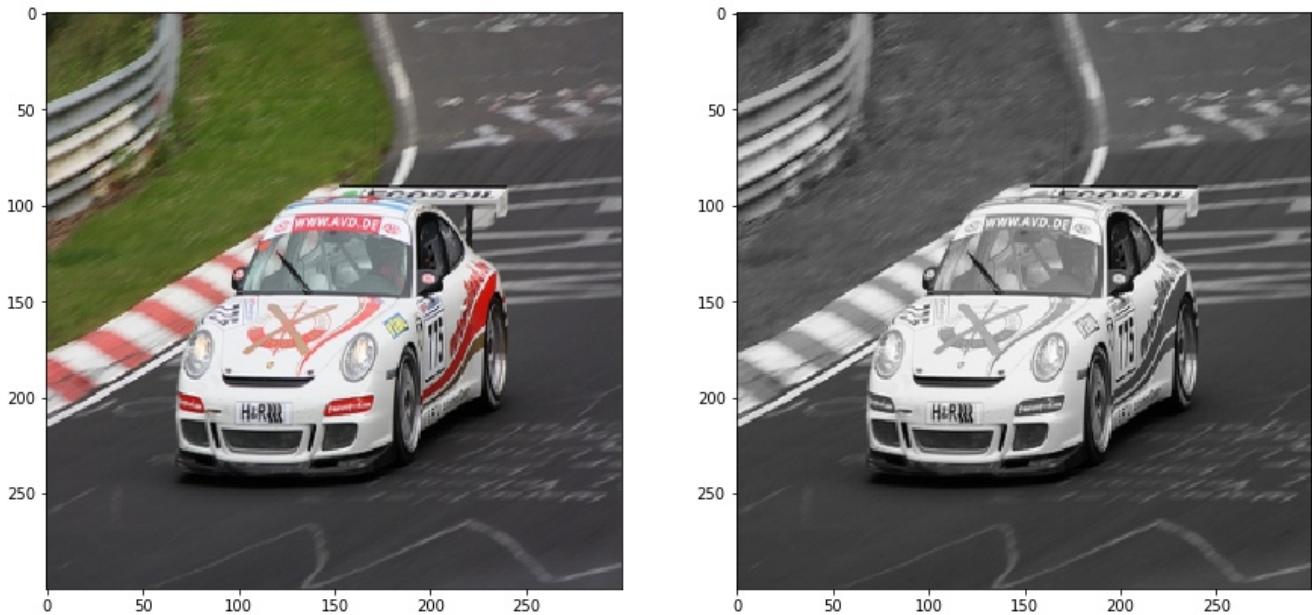


Illustration 20: RGB Image Converted to Grayscale

The two images above are the same, but not in size. As an array, the color image has dimension 300x300x3 whereas the grayscale image on the right is 300x300x1. The resulting array sizes are 1MB and 360KB respectively.

7.1 GRayscale Performance

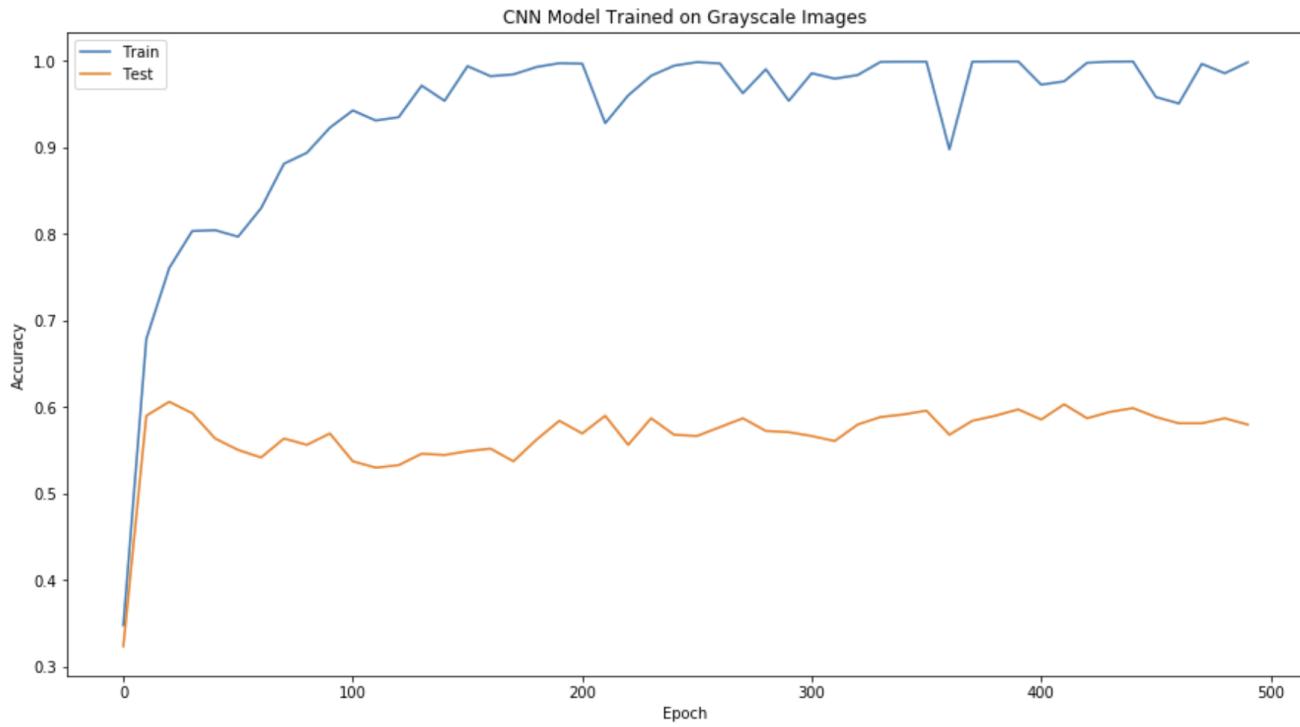


Illustration 21: Training and Test Set Accuracy for Grayscale Model

By converting to Grayscale and using the same CNN architecture, it was only able to achieve 61% accuracy on the test set. Meanwhile, it did achieve 99.9% accuracy on the training set. Again, it is grossly overfitting the training data as seen with the RGB images. On the plus side however, with full RGB data, training took 6.3 hours to go 300 Epochs, where it took only 5.1 hours to go 500 Epochs with the grayscale data. This breaks down to about 36.7 seconds per Epoch, a reduction of 51.4% training time. This would obviously save some time training various different CNN architectures. The complete metrics for this model are as follows:

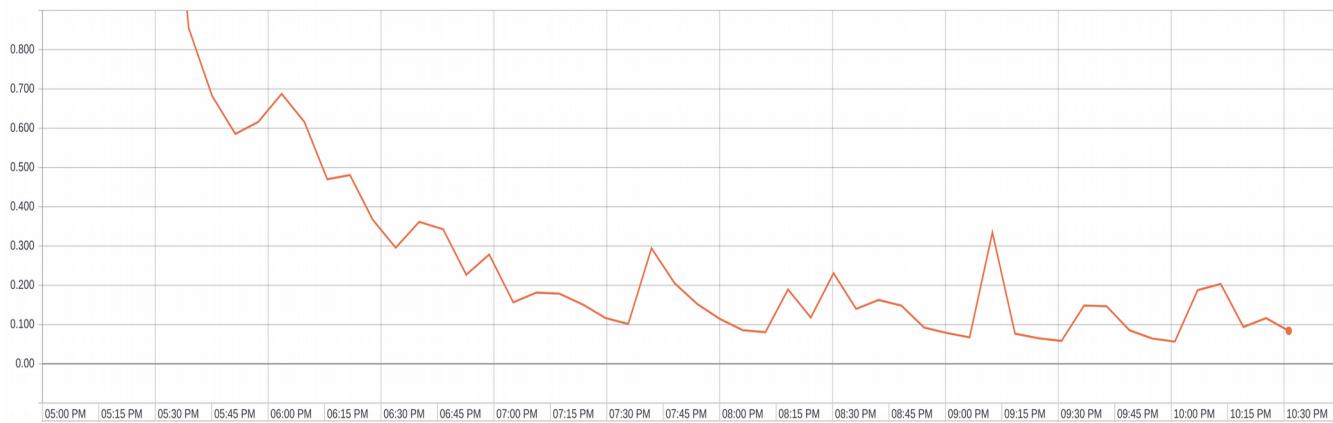
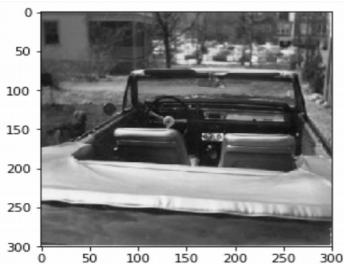


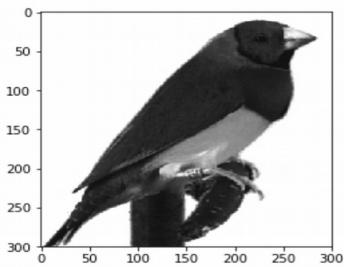
Illustration 22: Grayscale Training Cost with Respect to Time



This is a Car



This is a Dog



This is a Bird



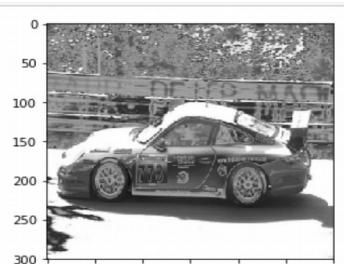
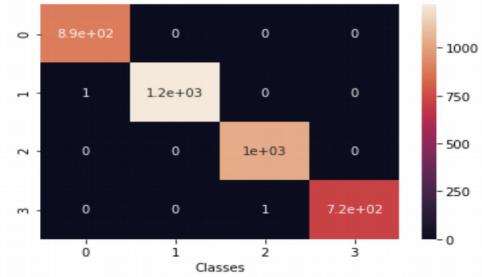
This is a Stove

Accuracy: 99.9482669425763%

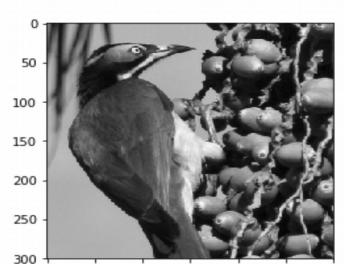
Out[129]:

	F-Beta	Precision	Recall	Support
Dog	0.999439	0.998879	1.000000	891
Bird	0.999592	1.000000	0.999184	1226
Stove	0.999514	0.999029	1.000000	1029
Car	0.999305	1.000000	0.998611	720

Confusion Matrix



This is a Stove



This is a Bird



This is a Stove



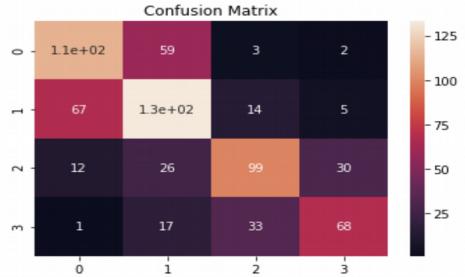
This is a Dog

Accuracy: 60.61493411420204%

Out[133]:

	F-Beta	Precision	Recall	Support
Dog	0.612903	0.587629	0.640449	178
Bird	0.585903	0.565957	0.607306	219
Stove	0.626582	0.664430	0.592814	167
Car	0.607143	0.647619	0.571429	119

Confusion Matrix



Further metrics investigation confirmed that with a 2/3 data reduction size came a 10% accuracy penalty when training the same model with grayscale data. Again, the training set metrics are on the left and the test set metrics are on the right. This isn't surprising since we did reduce data density in the images for the model to learn from. The hope was that even with the reduction in "information" the edges of each object in the images would still be defined enough for the model to detect accurately. Since the goal is for the model to learn the edges of dogs, birds, stoves, and cars essentially, maybe we can enhance the grayscale images to make it easier for the model to detect them.

8 HISTOGRAM EQUALIZATION

Histogram Equalization is a process of enhancing the contrast of grayscale images. The proper definition, as defined by the University of Utah Scientific Computing and Image Processing Institute, states

"Histogram equalization is a method to process images in order to adjust the contrast of an image by modifying the intensity distribution of the histogram. The objective of this technique is to give a linear trend to the cumulative probability function associated to the image. The processing of histogram equalization relies on the use of the cumulative probability function (cdf). The idea of this processing is to give to the resulting image a linear cumulative distribution function. Indeed, a linear cdf is associated to the uniform histogram that we want the resulting image to have."

This can be better explained with the help of visual aids. From a grayscale image, a histogram of all its pixel intensities can be rendered and their associated CDF.

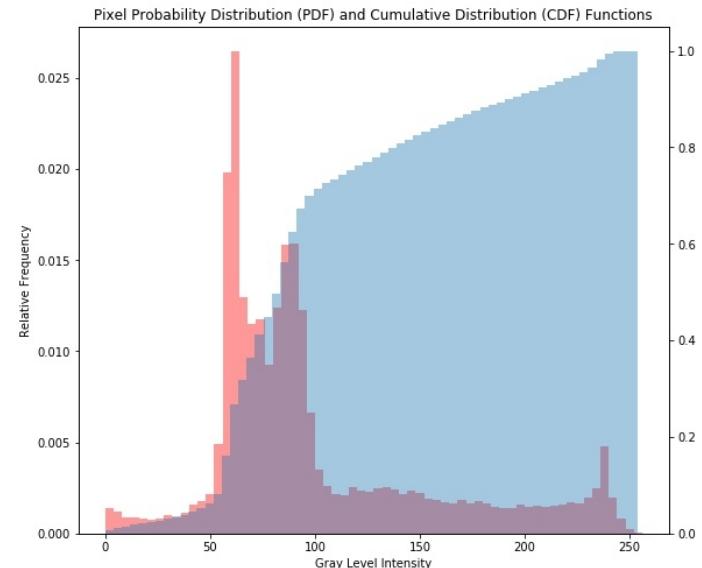
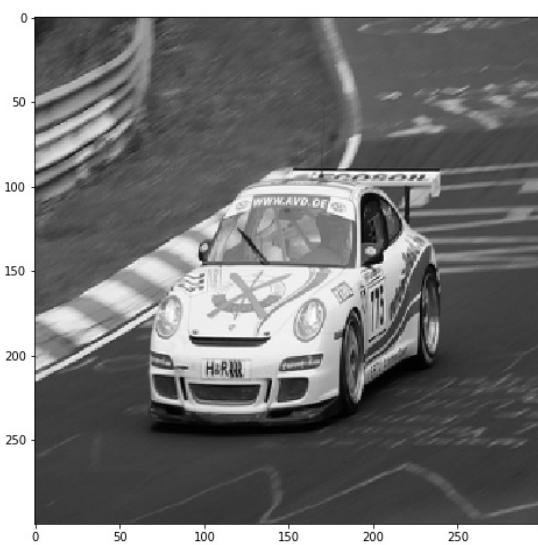


Illustration 23: Grayscale Image with Pixel Intensity Histogram and CDF

The CDF or cumulative distribution function is the probability that the a variable takes a value less than or equal to a specified x in the domain. The blue bars in the right image are the CDF for the grayscale car. The process of histogram equalization interpolates the histogram values so that the

resulting CDF is a linear line. The mathematics behind this are straightforward, but out of scope for this paper.

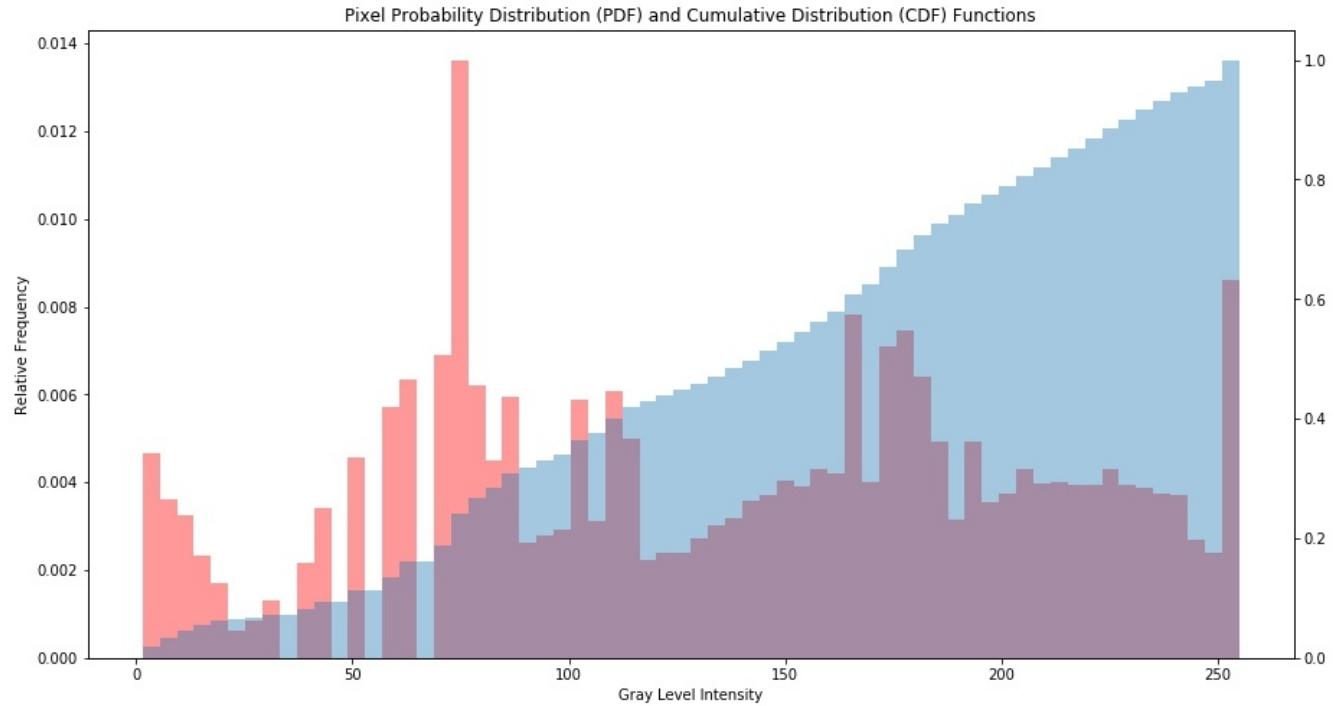


Illustration 24: Equalized Histogram on Grayscale Image

Notice that after the equalization process, the CDF is nearly linear. This process maintains the original image but improves contrast and in some cases sharpens the image as well.

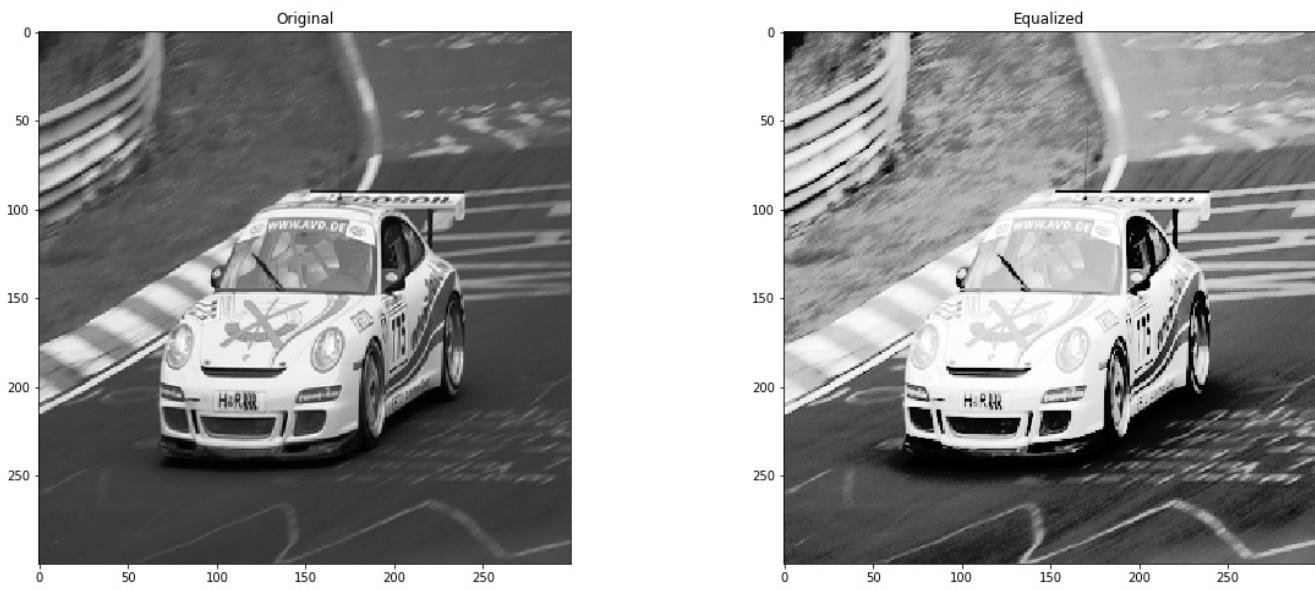


Illustration 25: Before and After of Equalized Image

8.1 EQUALIZATION PERFORMANCE

All grayscale images were equalized for both the training set and the test set and then ran through the grayscale model trained in the previous section. There were two models (Epochs) to test it on, the best performing model for the training set (Epoch 380) and the best performing model for the test set (Epoch 20) as seen in Illustration 18. The Equalized data performed as follows:

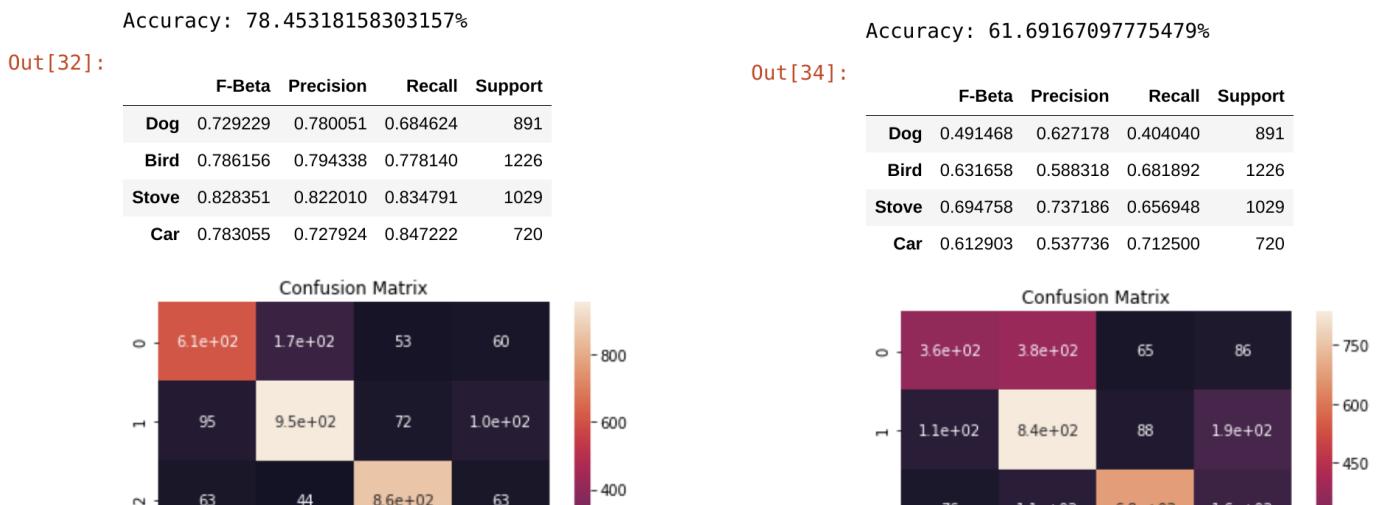


Illustration 26: Equalized Training Data - Epoch 380

Illustration 27: Equalized Training Data - Epoch 20

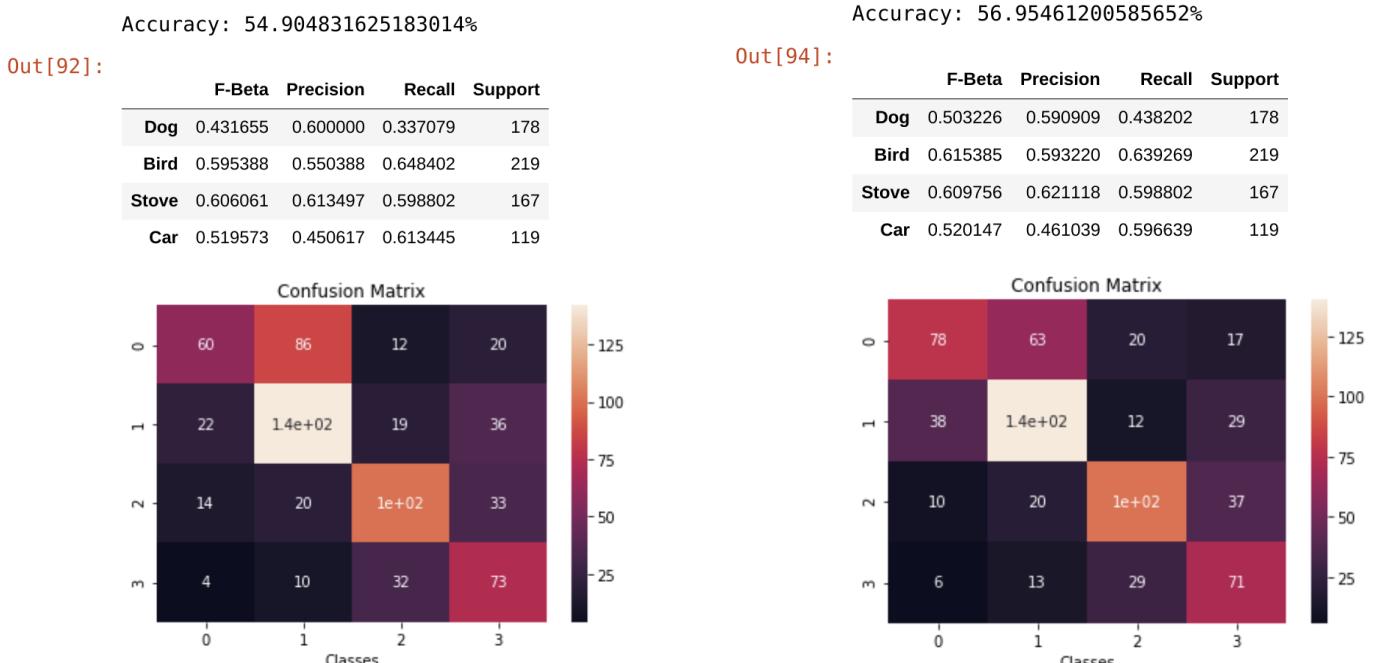


Illustration 29: Equalized Test Data - Epoch 380

Illustration 28: Equalized Test Data - Epoch 20

From the metrics above, it can be seen that performance decreased for all instances. The training data dropped from 99.9% accuracy to 78.4% on its best training Epoch. Conversely, the test data dropped from 60.6% to 56.95% on its best Epoch. The latter had a less significant drop in performance, but a drop nonetheless. The intent was to increase the models ability to detect the object's edges by improving image contrast and improve performance. Obviously these efforts didn't come to fruition. The next step was to retrain a model using both equalized and original images.

9 TRAINED EQUALIZED MODEL

Unfortunately, equalizing the images didn't provide the desired results. Still, some important qualities were extrapolated from the exercise.

1. It is much faster to train grayscale models than RGB so this would facilitate rapid prototyping and hyper-parameter tuning if necessary
2. Sharpening all the images may reduce the overfitting effect currently seen during the training process.
3. We have doubled our data for the model to train and be tested on by altering the original images. This should also hopefully alleviate the overfitting issue.

The same model architecture again was used to train the model on the now 7,732 images. Half are the original grayscale and the other half are the equalized gray scales.

9.1 EQUALIZED MODEL PERFORMANCE

To train the 7,732 images it took approximately 75.5 Seconds/Epoch. This is exactly the time it took to train the 3,866 RGB images and is quite expected since we've added more data for the model to learn from. Unfortunately the results weren't as desired.

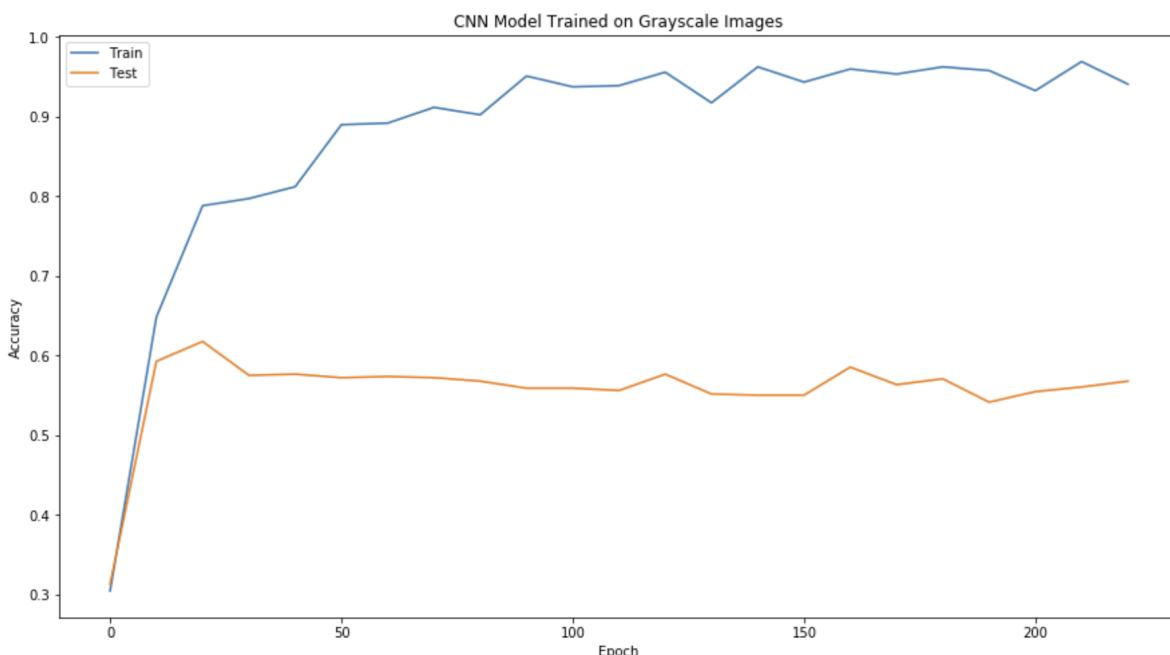


Illustration 30: Train and Test Set Accuracy on Original and Equalized Data

10 RESULTS

The summary statistics of all models are shown in the table below. Converting to grayscale did significantly speed up training while saving space – both hard disk and RAM. The increase of speed came at about 10% accuracy decrease on the test set. This isn't terrible considering 2/3 of the information in each image is condensed into a single array of pixel intensities. Even though a decrease in accuracy was present, that's not to say this method is inadequate. If several thousand images were added to the training data, this would linearly increase training time with respect to the number of images added – not to mention the strain on memory space this addition would cause.

Model	Training Examples	Test Examples	Epochs	Training Time (hours)	Seconds per Epoch	Train Accuracy (%)	Test Accuracy (%)
RGB	3866	683	300	6.3	75.6	96	72
Grayscale	3866	683	500	5.1	36.7	99.9%	60
Equalized	7732	1366	229	4.8	75.5	97%	62

10.1 EXTRA CONSIDERATIONS

- Due to the time expense of running each model, not all possibilities were explored. In order to train a truly robust object detection algorithm, several thousands of training pictures are used. Even though the final model was trained using double the amount of training examples, intrinsically they were still the same images. This could be the reason for only a 2% increase in accuracy on the test set. Another model could be trained using more training images in order to explore this potential fault.
- Another point of control could be the random weight initialization. Initialization of weights has been a big topic of study in regards to CNNs and Deep Neural Networks. The starting weights can play a big role in overall performance of the model due to an early subject, vanishing and exploding gradients. Xavier initialization was utilized for all models, but the random seed generator was not kept constant. Xavier initialization is out of scope for this paper, but the original research paper can be read [here](#). Essentially, although the same initializer was used, the same initial values themselves were not used model to model. This undoubtedly impacted the resultant of each training session.
- Touched on earlier, a point of focus is also the initial alteration of the original images themselves before being fed into the model. Each image was reshaped to a 300x300 image regardless of original size. This does have a negative impact of stretching, contorting, distorting and attenuating the resolution of the impending image. Spatial Pyramid Pooling would have alleviated this problem, but as stated earlier, this method was not implemented in order to keep the complexity to a minimum of the study.

- Finally, model architecture plays a big role as well. Stated earlier, several models were rapidly prototyped in order to get a good initial baseline established. This took several days in itself and was not at all exhaustive. After a baseline model architecture was established, it was used for all other models with no further investigation as to if there was a better architecture more suitable for the data reduction. There are several famous model blueprints that have returned exceptional performance. These structures however require a vast amount of computing power. Some of these model structures and their respective papers are:
 - AlexNet - [ImageNet Classification with Deep Convolutional Neural Networks](#)
 - VGG Net - [VERY DEEP CONVOLUTIONAL NETWORKS FOR LARGE-SCALE IMAGE RECOGNITION](#)
 - GoogleNet - [Going Deeper with Convolutions](#)
 - Microsoft ResNets - [Deep Residual Learning for Image Recognition](#)
 - And several others as outlined [here](#)

11 Conclusion

Even though RGB images provided the best performance, converting to Grayscale is a viable option to investigate further for computational efficiency. There was an attenuation in performance, but not significant enough to completely deter from this method. Taking into account memory requirements and training time, grayscale could potentially perform better than full RGB given enough training samples. Another component not discussed in this paper is prediction time. When checking accuracy on both training and test sets, it took a significant amount of time to predict on RGB vs Grayscale. Although this time constraint was not quantified, it was noticeable and very significant. In upcoming tests to further expand on this study, all code will be scaled up to cloud computing where several GPUs will be utilized in order to minimize training/prediction time to its economical minimum. This would be necessary to further combat the vanishing gradient problem due to the need of several more training iterations to counteract the minimal gradient updating in the early layers of the CNN.

APPENDIX A – Data Processing Class

```
1 class load_training_data():
2     def __init__(self,directory = '/home/scott/Pictures/Training Pictures'):
3         print('Navigating to Directory and Compiling Images')
4
5         #Initialize some empty variables
6         self.i = 0
7         self.training_data_paths = []
8         self.training_labels = []
9         self.X_train = None
10        self.X_test = None
11        self.y_train = None
12        self.y_test = None
13
14
15    try:
16        #Change to the directory with the images
17        os.chdir(directory)
18        #Make a list of all the subdirectories of different picture types
19        self.sub_directories = os.listdir()
20    except Exception as e:
21        print(e)
22        print('Please Enter a Valid Directory')
23
24
25
26    for pos, d in enumerate(self.sub_directories):
27        #Make a list of all the image names in the current sub directory
28        images = os.listdir(d)
29        #Create a list of full path names for every image in the current sub directory
30        self.training_data_paths.extend([os.path.join(directory, d+'/'+img) for img in images])
31        #Encode the target label for the current picture type
32        #Dogs = (1,0,0,0) | Birds = (0,1,0,0) | Stove = (0,0,1,0) | Cars = (0,0,0,1)
33        encoding = np.zeros(len(self.sub_directories))
34        encoding[pos] = 1
35        #Extend the list of one hot encoded training labels
36        self.training_labels.extend([encoding]*len(os.listdir(d)))
37
38
39
40    try:
41        #Assert that the total number one hot encoded target labels equals the total of training examples
42        assert len(self.training_data_paths) == len(self.training_labels)
43    except AssertionError:
44        raise(AssertionError('Number of Training Examples m = {} Does Not Match Number of Training Labels (m) = {}'.format(len(self.traini
45        print('Finished Compiling Images')
46
47
48
49
50    def process_images(self):
51        training_images = []
52
53        print('\nProcessing Images into n-Dimensional Arrays')
54        print('-----')
55        #Create a list of the 3-dimensional numpy array of pixel values for each image
56        training_images.extend(cv2.cvtColor(cv2.imread(path),cv2.COLOR_BGR2RGB) for path in self.training_data_paths)
57        #Delete unnecessary variables for RAM allocation control
58        del self.training_data_paths
59
60        #Show the average height and width in pixels of all images in the training data
61        height = np.mean([d.shape[0] for d in training_images])
62        width = np.mean([d.shape[1] for d in training_images])
63        print('\naverage image height is',height)
64        print('average image width is', width)
65        #Delete unnecessary variables for RAM allocation control
66        del height
67        del width
68
69        print('\nreshaping images to 300x300x3')
70        print('-----')
71        #Reshape all images to 300x300x3
72        training_images_reshaped = [cv2.resize(img, (300,300), interpolation = cv2.INTER_AREA) for img in training_images]
73        #Delete unnecessary variables for RAM allocation control
74        del training_images
75
76        print('\nCreating {} x 300 x 300 x 3 array'.format(len(self.training_labels)))
77        print('-----')
78        #Create a m X 300 X 300 X 3 array of all training data --> m being the total number of all pics
79        tensor_data = np.vstack([img for img in training_images_reshaped]).reshape(len(self.training_labels),300,300,3)
80        #Create a m X 1 X 4 array of all training labels
81        self.training_labels = np.vstack([label for label in self.training_labels]).reshape(len(self.training_labels),1,len(self.sub_directories))
82        #Delete unnecessary variables for RAM allocation control
83        del training_images_reshaped
84
85        print('\nexecuting Train Test Split')
86        print('-----')
87        #Create Train Test Splits
88        self.X_train, self.X_test, self.y_train, self.y_test = train_test_split(tensor_data, self.training_labels, test_size = 0.15, random_state
89        print('done Creating Training and Testing Data at 85/15 Split!')
90        #Delete unnecessary variables for RAM allocation control
91        del self.training_labels
92
93
94    return self
95
96    def next_batch(self, batch_size):
97        #Method for batch feeding into model if RAM capacity is sufficient to load data all at once
98        x = self.X_train[self.i:self.i+batch_size]
99        y = self.y_train[self.i:self.i+batch_size]
100       self.i = (self.i + batch_size) % len(self.X_train)
101       return x,y
102
103    def write_h5(self):
104        print('\nWriting data to HDF5')
105        print('-----')
106        # Method to write transformed data to hdf5
107        hf = h5py.File('data.h5', 'w')
108
109        hf.create_dataset('X_train', data=np.divide(self.X_train,255))
110        hf.create_dataset('X_grayscale_train', data = np.expand_dims(np.array([cv2.cvtColor(x, cv2.COLOR_RGB2GRAY) for x in self.X_train]), axis
111        hf.create_dataset('X_test', data=np.divide(self.X_test,255))
112        hf.create_dataset('X_grayscale_test', data = np.expand_dims(np.array([cv2.cvtColor(x, cv2.COLOR_RGB2GRAY) for x in self.X_test]), axis=1
113        hf.create_dataset('y_train', data=self.y_train)
114        hf.create_dataset('y_test', data=self.y_test)
115        hf.close()
116
117        print('\nfinished Writing h5 in {}'.format(os.getcwd()))
```

APPENDIX B – RGB Tensorflow Class

```

class cnnModel:
    def __init__(self, savefile=None, weight_initializer = tf.contrib.layers.xavier_initializer(seed = 0), learning_rate=0.01, beta = 0.01):
        #Create save file path variable
        self.savefile = savefile
        #Reset any current graphs
        tf.reset_default_graph()
        #Set Random Seed
        np.random.seed(0)

        #Print Useful Model Information
        print('Creating CNN Graph')
        print("Tensorboard Log files stored in ./tf_logs/RGB/")
        print('To Activate Tensorboard Session: Stensorboard --logdir "./tf_logs/RGB/" --port 6006')
        print('Optimizer Learning Rate: {}' .format(learning_rate))
        print('L2 Regularization Constant: {}' .format(beta))

        if savefile is not None:
            print('Model Save Directory: {}' .format(os.path.join(os.getcwd(),savefile)))
        else:
            print('WARNING: No save directory - Model and Weights will be lost after training\n')

        #Placeholders
        self.inputs = tf.cast(tf.placeholder(dtype = tf.float64, shape = [None, 300,300,3], name = 'X'), tf.float32)
        self.labels = tf.cast(tf.placeholder(dtype = tf.float64, shape = [None,4], name = 'y'), tf.float32)
        self.hold_prob = tf.placeholder(tf.float32)

        #Convolutional Layer #
        with tf.name_scope('Convolution layer 1'):
            self.W1 = tf.get_variable(dtype = tf.float32, name = 'W1', shape = [7,7,3,10], initializer=weight_initializer)
            self.b1 = tf.Variable(tf.constant(0.1, shape = [10]))
            self.C1 = tf.nn.conv2d(input = self.inputs, filter = self.W1, strides = [1,2,2,1], padding = 'SAME', name = 'C1')
            self.A1 = tf.nn.relu(features = self.C1, name = 'A1') + self.b1
            self.P1 = tf.nn.max_pool(self.A1, ksize = [1,2,2,1], padding = 'VALID', strides = [1,2,2,1], name = 'Pool_1')

        #Convolutional layer #
        with tf.name_scope('Convolution layer 2'):
            self.W2 = tf.get_variable(dtype = tf.float32, name = 'W2', shape = [4,4,self.P1.shape[3],10], initializer=weight_initializer)
            self.b2 = tf.Variable(tf.constant(0.1, shape = [10]))
            self.C2 = tf.nn.conv2d(input = self.P1, filter = self.W2, strides = [1,2,2,1], padding = 'SAME', name = 'C2')
            self.A2 = tf.nn.relu(features = self.C2 , name = 'A2') + self.b2

        #Convolutional Layer #
        with tf.name_scope('Convolution layer 3'):
            self.W3 = tf.get_variable(dtype = tf.float32, name = 'W3', shape = [3,3,self.A2.shape[3],20], initializer=weight_initializer)
            self.b3 = tf.Variable(tf.constant(0.1, shape = [20]))
            self.Z1 = tf.contrib.layers.flatten(self.A2)
            self.C3 = tf.nn.conv2d(input = self.Z1, filter = self.W3, strides = [1,3,3,1], padding = 'SAME', name = 'C3')
            self.A3 = tf.nn.relu(features = self.C3 , name = 'A3') + self.b3
            self.P3 = tf.nn.max_pool(value = self.A3, ksize = [1,2,2,1], padding = 'VALID', strides = [1,2,2,1], name = 'Pool_3')

        #Fully Connected Layer
        with tf.name_scope('Fully connected layer'):
            self.flaten = tf.contrib.layers.flatten(self.P3)
            self.Z1 = tf.contrib.layers.fully_connected(self.flaten, num_outputs = 4, activation_fn = None)
            self.hold_out_fc = tf.nn.dropout(self.Z1, keep_prob = self.hold_prob)

        #Prediction Operation
        with tf.name_scope('Prediction Op'):
            self.predictions = tf.argmax(tf.nn.softmax(logits = self.hold_out_fc), 1)
            self.correct_prediction = tf.equal(self.predictions, tf.argmax(self.labels, 1))
            self.accuracy = tf.reduce_mean(tf.cast(self.correct_prediction, 'float'))

        #Training Operation
        with tf.name_scope('Training Op'):
            self.regularizers = tf.nn.l2_loss(self.W1) + tf.nn.l2_loss(self.W2) + tf.nn.l2_loss(self.W3)
            self.loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(logits = self.Z1, labels = self.labels))
            self.cost = tf.reduce_mean(self.loss + (beta * self.regularizers))
            self.optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(self.cost)

        #Global Variables Initializer
        with tf.name_scope('Initializers'):
            self.init = tf.global_variables_initializer()
            self.saver = tf.train.Saver(max_to_keep=500, filename='Step_number')

    def fit(self,x,y, num_epochs = 100, batch_size = 50):
        print('***** TRAINING MODEL *****')
        #get total number of batches from input size
        num_of_batches = int(x.shape[0] / batch_size)
        #initialize some needed variables
        prev_cost = 0
        cost_tracker = 0
        #Initialize an extra summary writer
        cost_summary_2 = tf.summary.scalar('Cost 2', self.cost)
        accuracy_summary_2 = tf.summary.scalar('Accuracy 2', self.accuracy)

        #Run the graph
        with tf.Session() as sess:
            #Initialize all Global Variables
            sess.run(self.init)

            #Initialize log writer for Tensorboard
            writer = tf.summary.FileWriter('./tf_logs/RGB/', sess.graph)
            for i in range(num_of_epochs):
                #Initialize useful variables
                cost = 0
                batch_start = 0

                for j in range(num_of_batches):
                    #Train the model and track cost
                    _,c = sess.run([self.optimizer,self.cost], feed_dict={self.inputs:x[batch_start:batch_start + batch_size], self.labels:y[batch_start:batch_start + batch_size], self.hold_prob:0.5})
                    batch_start+=batch_size
                    cost+=c

                #Check for flat cost values over each iteration
                if np.absolute(cost - prev_cost) <= 0.001:
                    print('Cost has reached floor value @ {} - Terminate learning on Epoch {}'.format(cost,i))
                    break

                #Check for increasing cost values over each iteration
                if cost > prev_cost:
                    cost_tracker += 1
                    prev_cost = cost
                    if cost_tracker == 20:
                        self.saver.save(sess, self.savefile,global_step=i)

                        #write log files for tensorboard
                        summary = sess.run(self.merged_summaries, feed_dict = {self.inputs:x, self.labels:y, self.hold_prob:1})
                        writer.add_summary(summary, i)
                        print("Cost has increased 20 Epochs in a row - Terminate learning on Epoch {} with cost value {}".format(i,cost))
                        break

                else:
                    prev_cost = cost
                    cost_tracker = 0

                #Every 10 epochs print cost and write tensorflow logs
                if i%10 == 0:
                    print('Cost For Epoch {} : {}'.format(i,cost))

                #save a checkpoint of the model
                self.saver.save(sess, self.savefile,global_step=i)

                #write log files for tensorflow
                summary = sess.run(self.merged_summaries, feed_dict = {self.inputs:x[:1933], self.labels:y[:1933], self.hold_prob:1})
                cost_2, accuracy_2 = sess.run([cost_summary_2,accuracy_summary_2], feed_dict = {self.inputs:x[1933:], self.labels:y[1933:], self.hold_prob:1})
                writer.add_summary(cost_2,i)
                writer.add_summary(accuracy_2,i)

            def predict(self, x):
                with tf.Session() as sess:
                    #Restore last saved model checkpoint
                    self.saver.restore(sess, self.savefile)
                    #Run feed forward part of the graph for predictions
                    preds, w1, w2, w3 = sess.run([self.predictions,self.W1, self.W2, self.W3], feed_dict = {self.inputs:x, self.hold_prob:1})
                    results = {'Predictions':preds, 'W1':w1, 'W2':w2, 'W3':w3}
                    return results

```