DEEP LEARNING

CAR MODEL RECOGNITION

SUMMER REPORT

Team Impetuors : *under Prof. Nishchal Verma*

Preetham Paul **Praphul Singh**

Harshit Saini **Aadrish Sharma**

Guided by : Vikas Singh

Introduction

Deep Learning is a bleeding edge field, which took Artificial Intelligence to the next level. With deep networks expanding their utility for different tasks, deep networks have reserved their place in almost every field. With growing need for technology that aids the human-kind, deep Learning has proved to be loyal in promoting AI in every field.

Our code demonstrates how deep networks can be used to contribute to one of the most anticipated tasks that every India teenager wants to do - Know the details of a stunning car when it appears in front of him, on road. Our project's goal is to enable him to simply take a picture of it in his smartphone, and get an handful of information, about the car and its model. Well, this may appear to be quite challenging for an ordinary smartphone, but with about 200 lines of code, trained upon over 800 lines of code running in the app's

server on the internet, this is not challenging anymore. Deep Learning has made this possible.

Our code is written in **tensorflow and python.**

OUR AIM : Recognise the model of the car from a photograph through Deep Learning techniques

Reading Time:

We've marked the inception of the project with some reading tasks. We've spent 4 weeks spent, studying about Deep Learning, as we're total amateurs in this field, and this is our first practical project in Deep Learning. We've covered different topics like **Encoders, CNN's, RNN's, LSTM's, Transfer Learning, HOG, SVM's...**

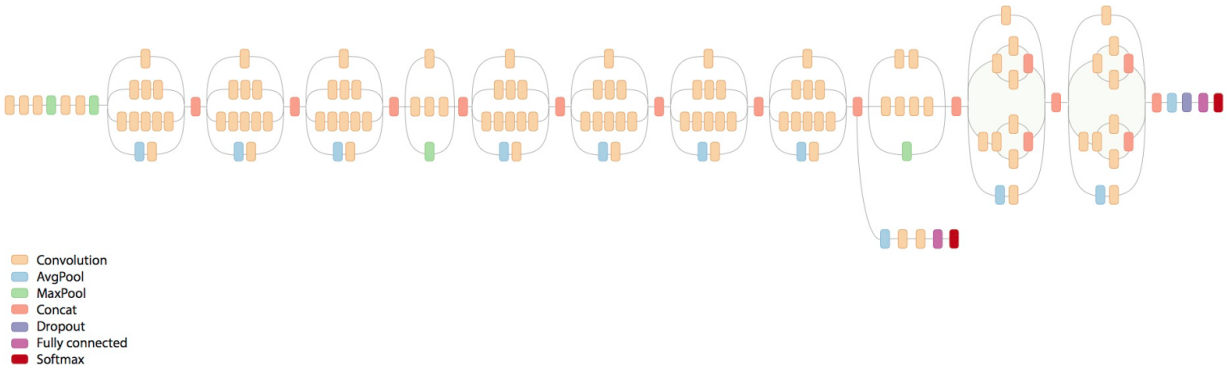
Data Collection :

Data collection is one of the challenging things that any machine learning engineer faces. For reaching goals that no one has ever reached (that is, no one has ever collected data for different models of cars), we have spent days collecting data from the internet. As data from internet is uncoordinated, we had to spend to spend lot of time removing irrelevant images from the data. This made the data, in simple terms, quite 'pure'.

We have collected ***1,50,000+*** (the number of images before the data turned 'pure' is more than ***2,00,000***) images of about ***140*** models, belonging to ***14*** brands of cars which one can find Indian roads. These are quite big figures for a team of 4 members, working in a week's span. With all the data collected, organised and sorted, we have set to train the data with Google's Inception-v3 model.

Inception-v3:

Proposed in 2014 by Christian Szegedy and other Google researchers and used in the GoogLeNet architecture that won both the ILSVRC 2014 classification and detection challenges.



Transfer Learning :

We are using the Inception-v3 model to train each brand of car for its different models which gives a good accuracy for small number of classes approximately less than 30. Inception-v3 model was trained on imagenet data in the 2012 Imagenet challenge, upon 6 GPUs for about 8 weeks. As we lack such huge amount of computation power, we resorted to something better instead of trained the whole model from the scratch - Transfer Learning.

Transfer learning is a technique that shortcuts a lot of this work by taking a fully-trained model for a set of categories like Inception-v3, and retraining from the existing weights pre-trained over a larger data, for new classes. So, we took the saved graph from the imagenet 2012 challenge, and used it for feature extraction. In this project we'll be retraining the final layer from scratch, while leaving all the others untouched.

The problem we faced was that we were having about 14 brands and there were about 10 models for each class. So, finally the number of classes became 140 training which, getting a good accuracy was not possible by implementing just Transfer Learning.

We needed something special. So we decided to train for the brands separately.

Retraining Inception-v3 :

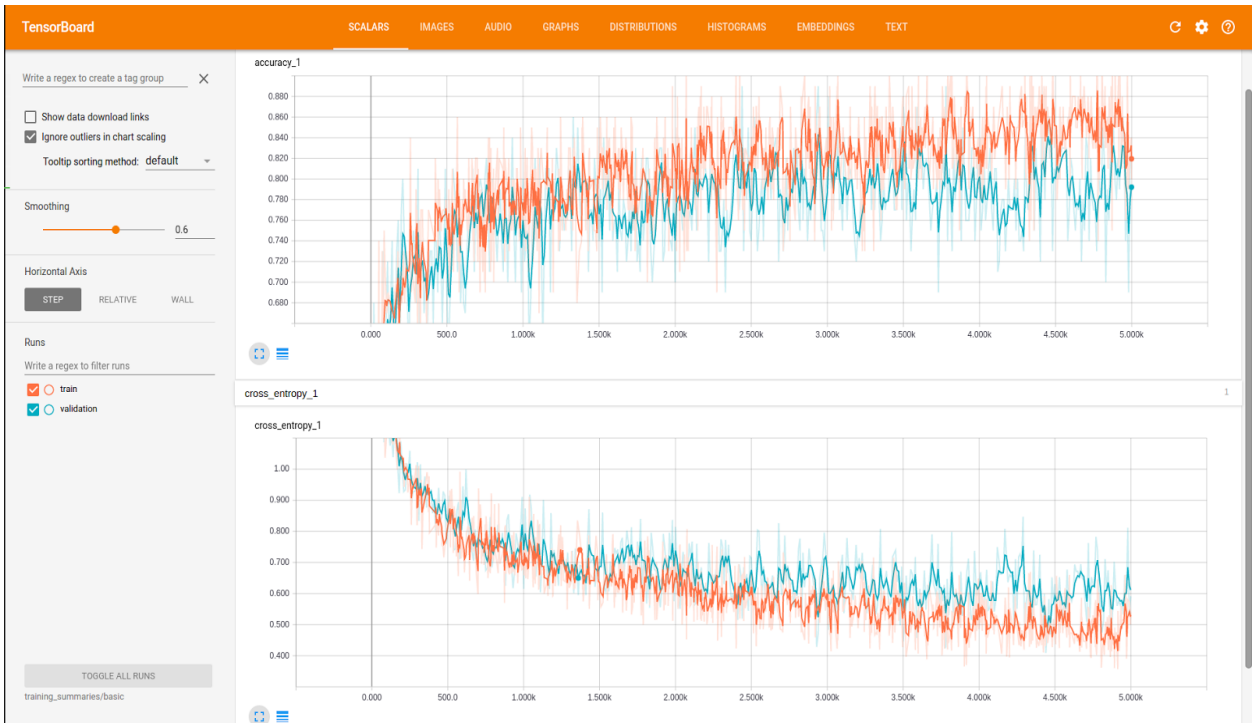
We have added an extra layer at the end of the inception-v3 layer as mentioned before. We've trained the last layer first for all the models at once. But the model's accuracy dropped below 55% as the number of classes was high. So, we've thought of classifying images based on brand first, and then trained them for the models in the respective brands.

Here, we are providing the accuracies for different brands, trained only for models within.

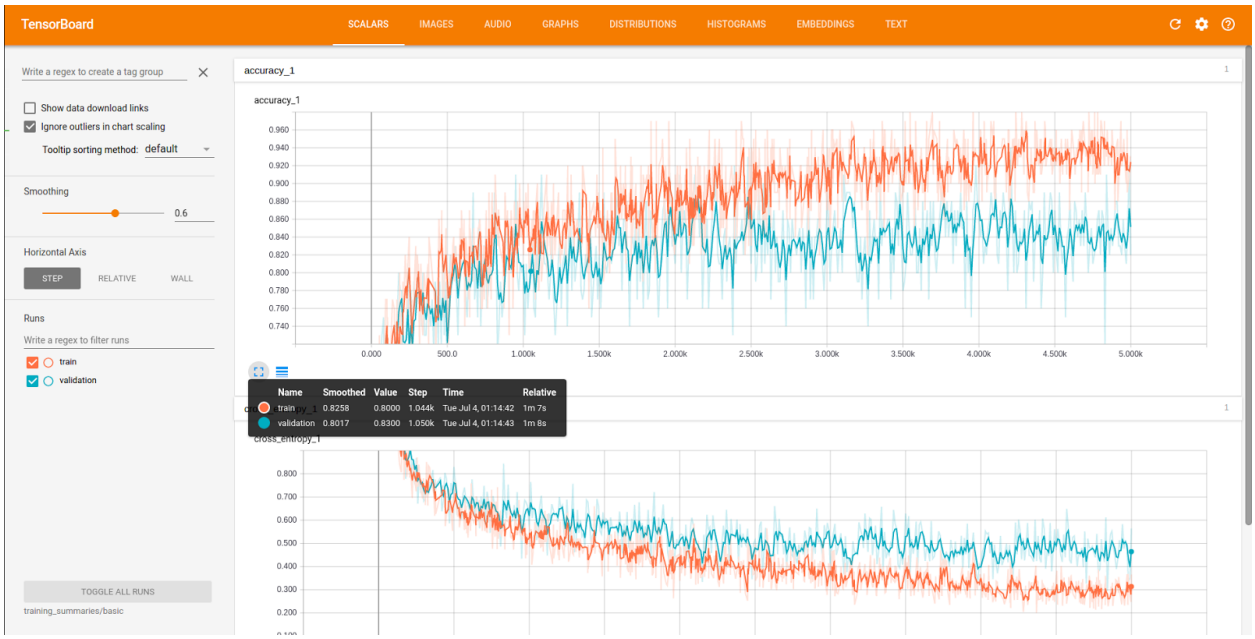
We have trained for models individually in each brand : Fiat, Ford, Honda, Hyundai, Mahindra, Maruti Suzuki, Volkswagen, Land Rover, Chevrolet, Tata, Datsun etc.

There has been problem of overfitting in the beginning, which has been resolved using a ***dropout layer*** .

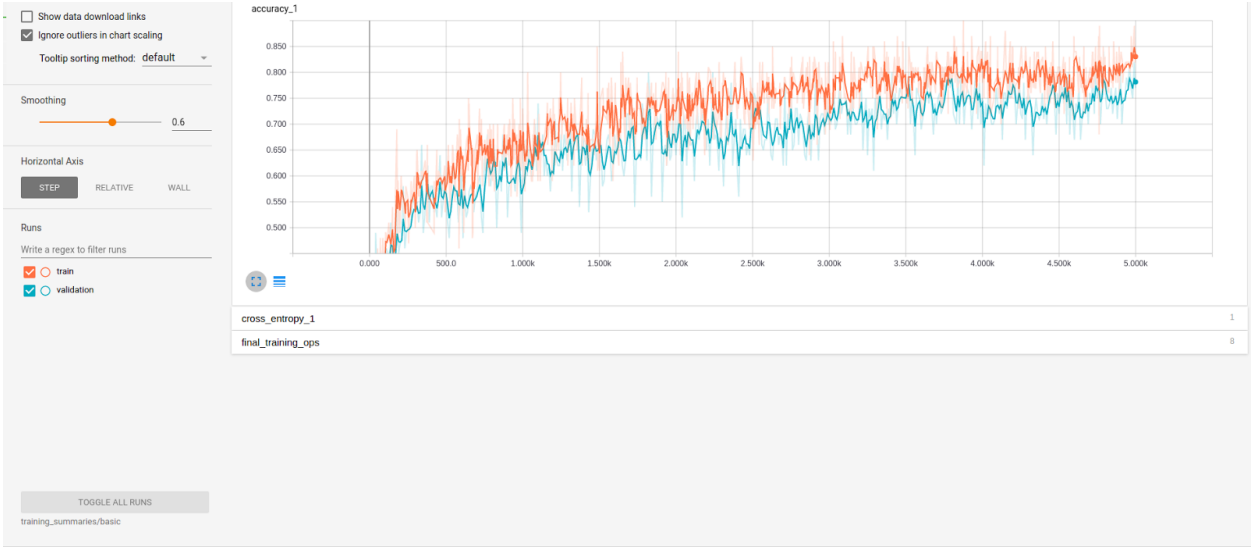
Fiat : Test Accuracy : 75%



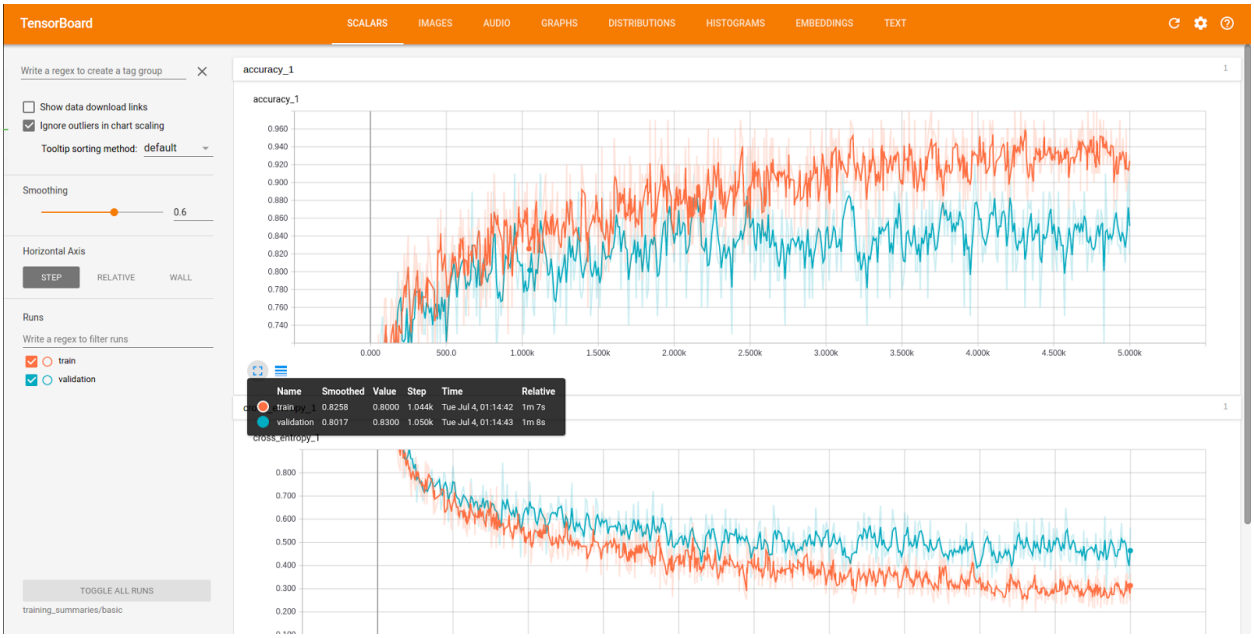
Ford : Test Accuracy : 76%



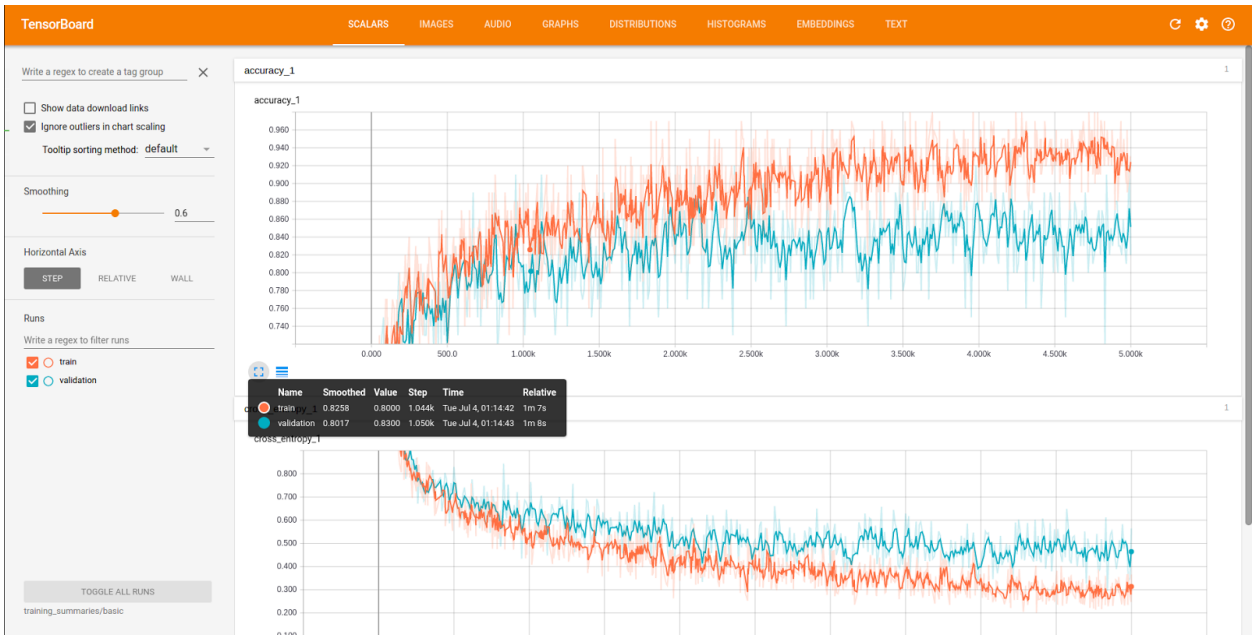
Hyundai : Test Accuracy = 73%



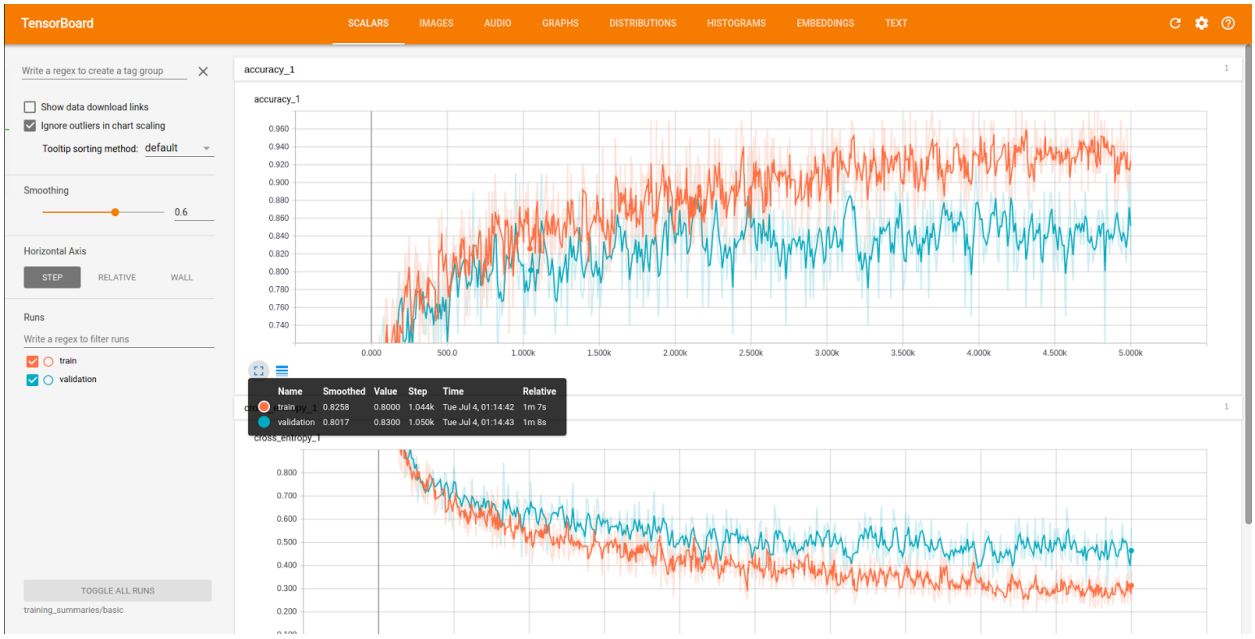
Honda : Test Accuracy : 78%



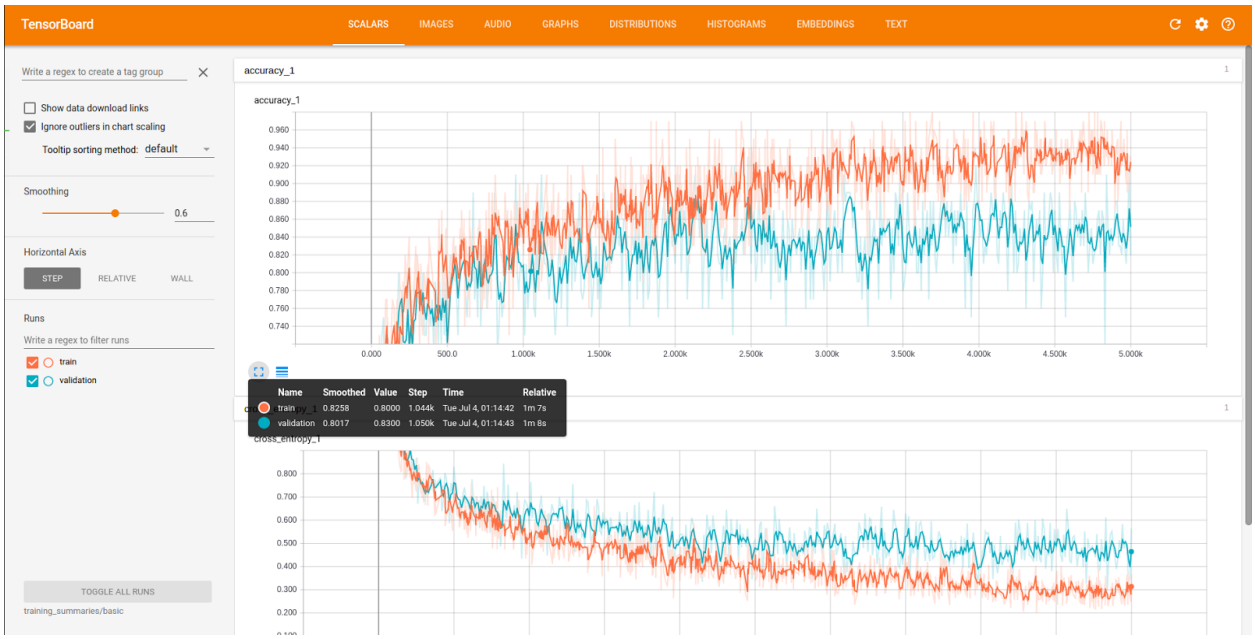
TATA : Test accuracy = 80%



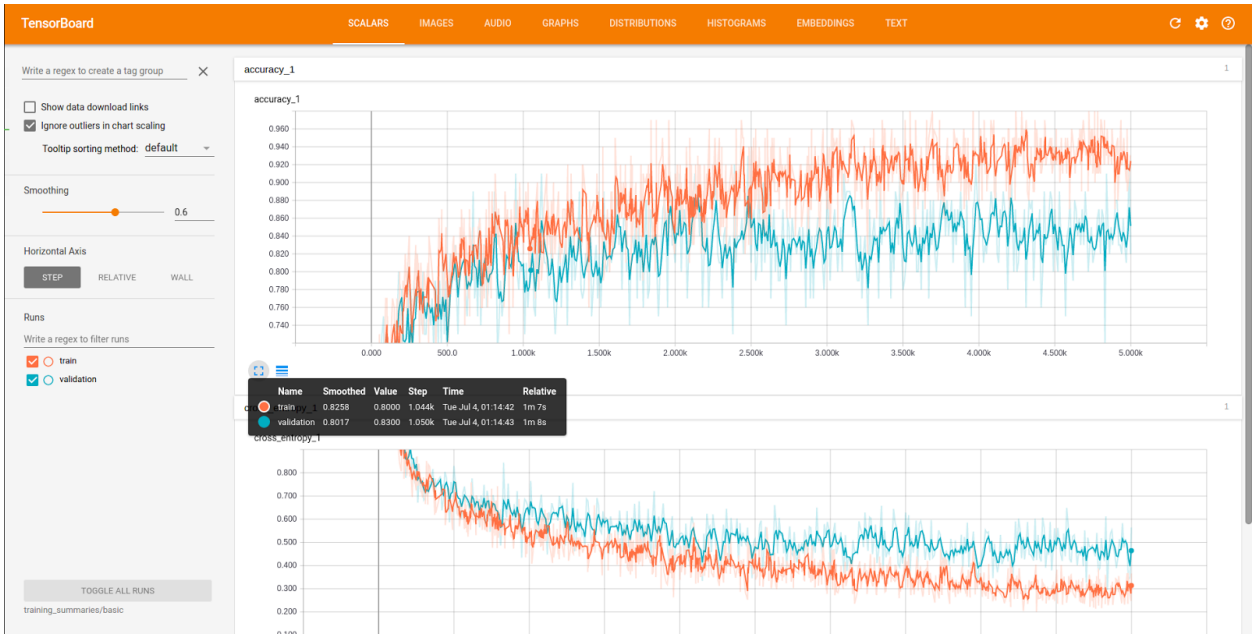
Mahindra : Test accuracy = 80%



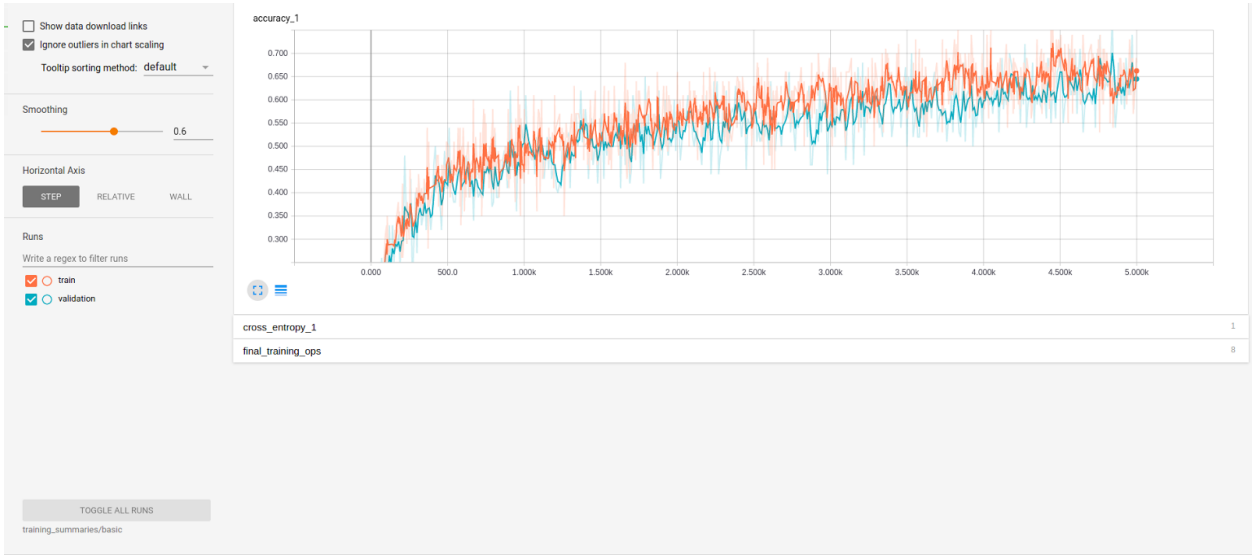
Chevrolet : Test Accuracy = 77%



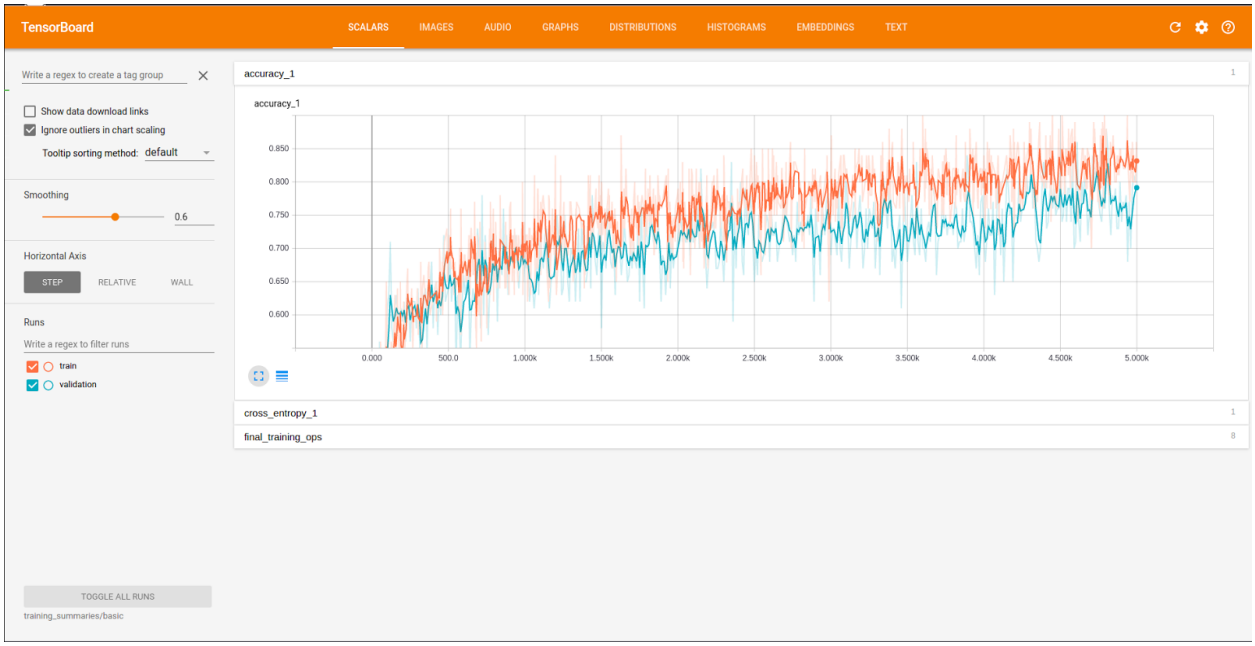
Maruti Suzuki Test accuracy = 75%



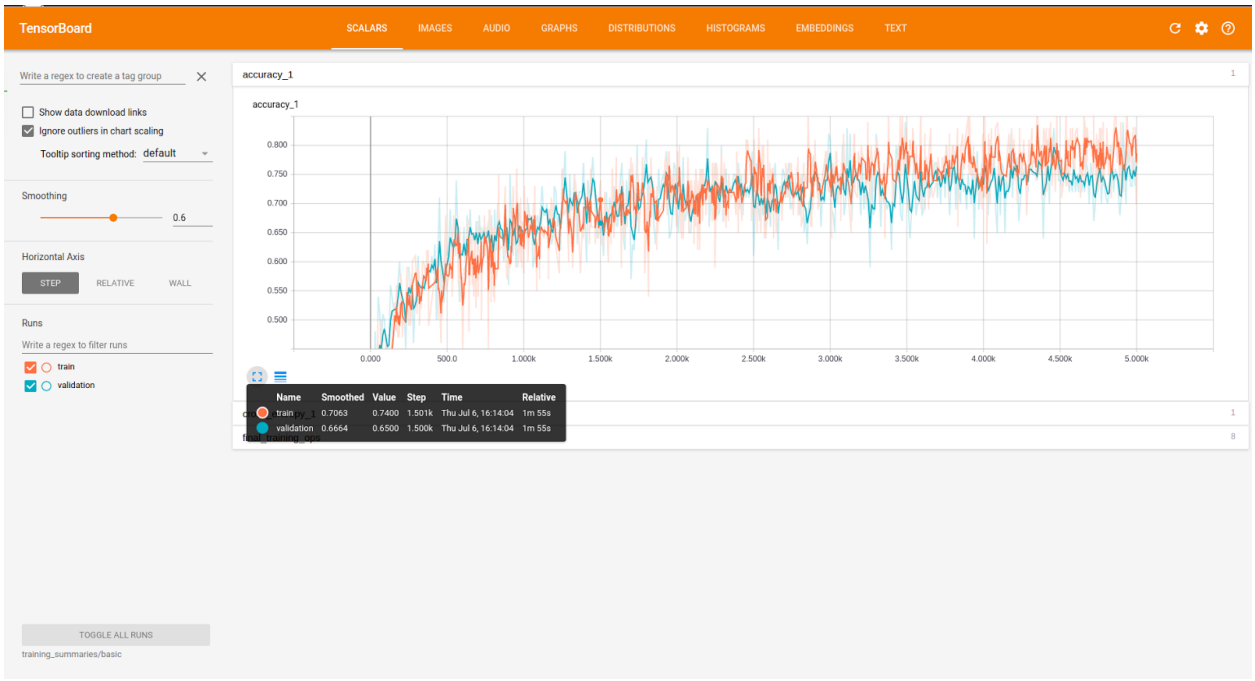
Volkswagen Test accuracy = 78%



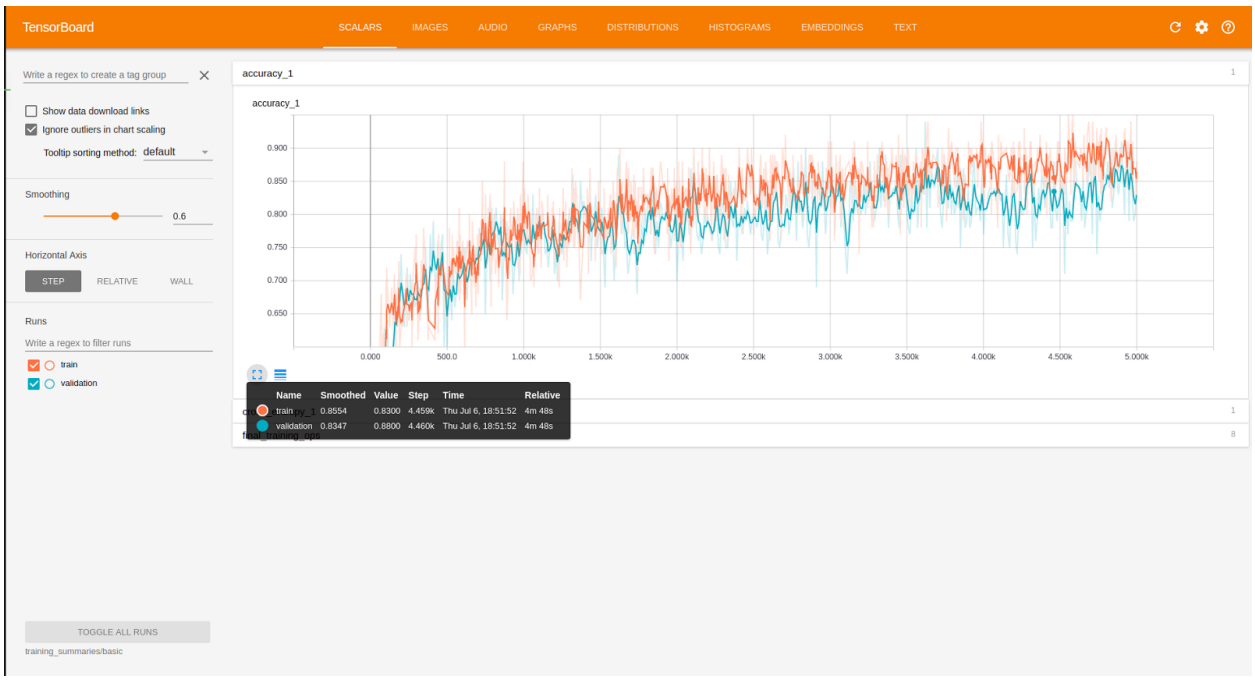
LandRover Test accuracy = 82%



Datsun Test accuracy = 100%



Renault Test accuracy = 92%



Histogram of Oriented Gradients

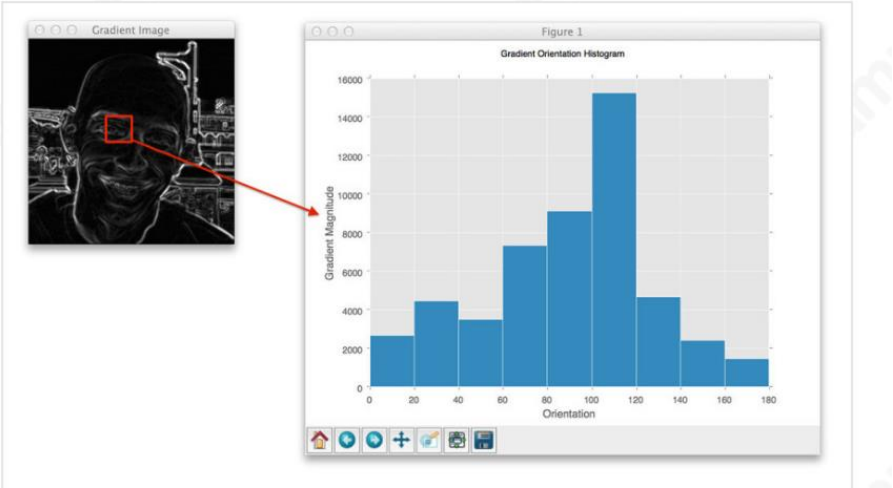
This is a feature descriptor discussed in computer vision and image processing for the purpose of object detection. The technique counts occurrences of gradient orientation in localized portions of an image. This method is similar to that of edge orientation histograms, scale-invariant transform descriptors, and shape contexts, but differs in that it is computed on a dense grid of uniformly spaced cells and uses overlapping local contrast normalization for improved accuracy.

We have seen the convolutional layers in which filters are used to map different parts of images to feature vectors. We needed a feature which could be taken as the unique thing to distinguish the brands.

HOG can be taken similar to one of those filters which we have used to map logos of the cars to extract features. We trained the logo data using HOG filters in a standard model named SVM which gave us an **accuracy of 88%**.

How does HOG work?

The cornerstone of the HOG descriptor algorithm is that appearance of an object can be modeled by the distribution of intensity gradients inside rectangular regions of an image:



Implementing this descriptor requires dividing the image into small connected regions called cells, and then for each cell, computing a histogram of oriented gradients for the pixels within each cell. We can then accumulate these histograms across multiple cells to form our feature vector.

Dalal and Triggs also demonstrated that we can perform *block normalization* to improve performance. To perform block normalization we take groups of overlapping cells, concatenate their histograms, calculate a normalizing value, and then contrast normalize the histogram. By normalizing over multiple, overlapping blocks, the resulting descriptor is more robust to changes in illumination and shadowing. Furthermore, performing this type of normalization implies that each of the cells will be represented in the final feature vector multiple times but normalized by a slightly different set of neighboring cells.

Step 1: Normalizing the image prior to description.

This normalization step is entirely optional, but in some cases this step can improve performance of the HOG descriptor. There are three main normalization methods that we can consider:

- **Gamma/power law normalization**: In this case, we take the $\log(p)$ of each pixel p in the input image. However, as Dalal and Triggs demonstrated, this approach is perhaps an "over-correction" and hurts performance.
- **Square-root normalization**: Here, we take the $\sqrt{(p)}$ of each pixel p in the input image. By definition, square-root normalization compresses the input pixel intensities far less than gamma normalization. And again, as Dalal and Triggs demonstrated, square-root normalization actually increases accuracy rather than hurts it.
- **Variance normalization**: A slightly less used form of normalization is variance normalization. Here, we compute both the mean μ and standard deviation σ of the input image. All pixels are mean centered by subtracting the mean from the pixel intensity, and then normalized through dividing by the standard

deviation: $p' = (p - \mu)/\sigma$. Dalal and Triggs do not report accuracy on variance normalization; however, it is a form of normalization that I like to perform and thought it was worth including.

In most cases, it's best to start with either no normalization or square-root normalization. Variance normalization is also worth consideration, but in most cases it will perform in a similar manner to square-root normalization (at least in my experience).

Step 2: Gradient computation

The first actual step in the HOG descriptor is to compute the image gradient in both the *x* and *y* direction. These calculations should seem familiar, as we have already reviewed them in the Gradients .

We'll apply a convolution operation to obtain the gradient images:

$G_x = I \star D_x$ and $G_y = I \star D_y$

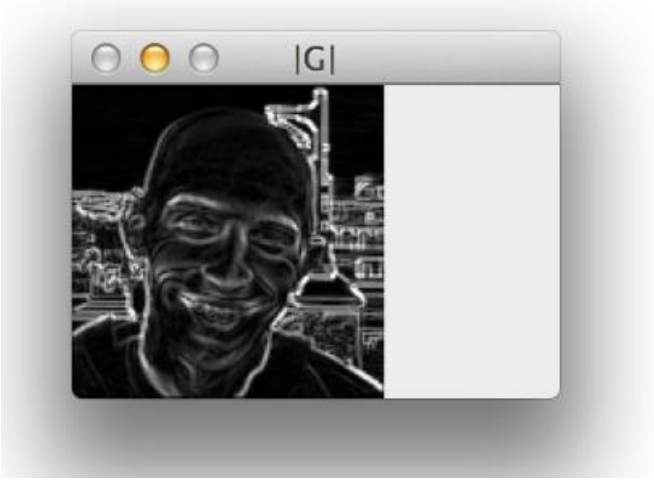
where I is the input image, D_x is our filter in the *x*-direction, and D_y is our filter in the *y*-direction.

As a matter of completeness, here is an example of computing both the *x* and *y* gradient of an input image:

Now that we have our gradient images, we can compute the final gradient magnitude representation of the image:

$$|G| = \sqrt{G_x^2 + G_y^2}$$

Which we can see below:



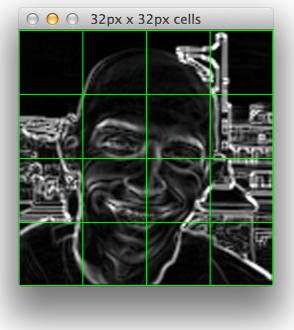
Finally, the orientation of the gradient for each pixel in the input image can then be computed by:

$$\theta = arctan2(G_y, G_x)$$

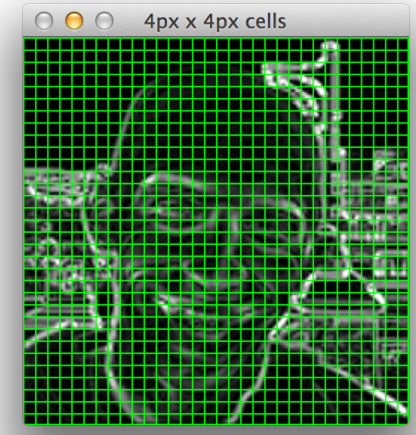
Given both $|G|$ and θ, we can now compute a histogram of oriented gradients, where the bin of the histogram is based on θ and the *contribution* or *weight* added to a given bin of the histogram is based on $|G|$.

Step 3: Weighted votes in each cell

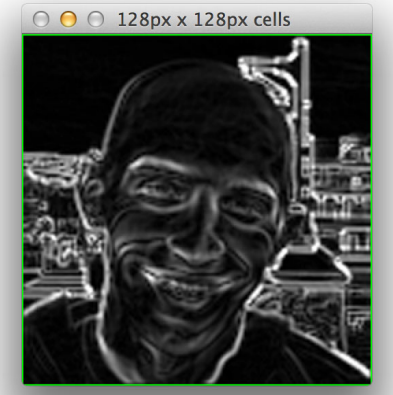
Now that we have our gradient magnitude and orientation representations, we need to divide our image up into cells and blocks.



A "cell" is a rectangular region defined by the number of pixels that belong in each cell. For example, if we had a *128 x 128* image and defined our `pixels_per_cell` as *4 x 4*, we would thus have *32 x 32* = *1024* cells:



If we defined our `pixels_per_cell` as *32 x 32*, we would have *4 x 4 = 16* total cells:



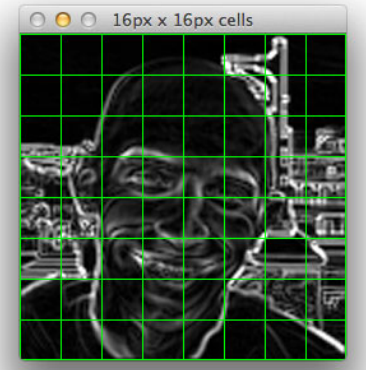
And if we defined `pixels_per_cell` to be *128 x 128*, we would only have *1* total cell:

Obviously, this is quite an exaggerated example; we would likely never be interested in *1 x 1* cell representation. Instead, this demonstrates how we can divide an image into cells based on the number of pixels per cell.

Now, for each of the cells in the image, we need to construct a histogram of oriented gradients using our gradient magnitude $|G|$ and orientation θ mentioned above.

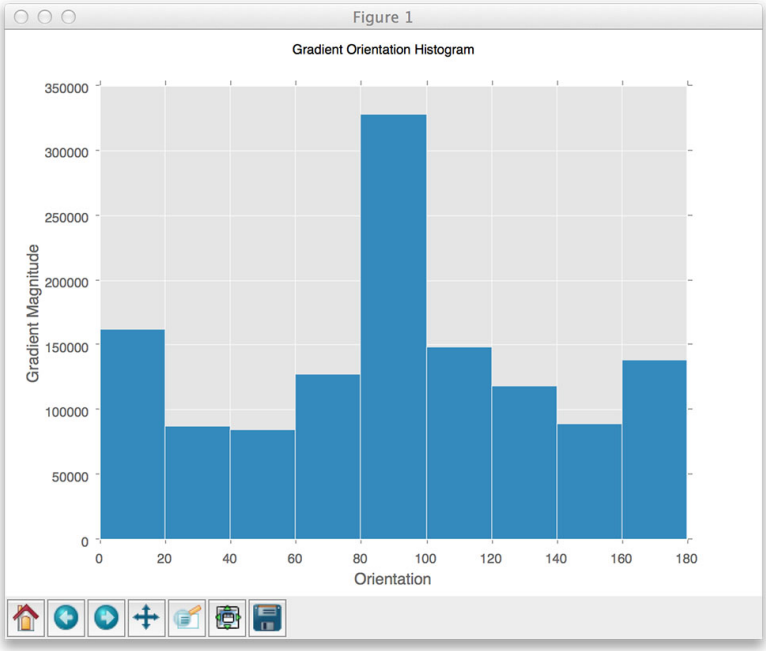
But before we construct this histogram, we need to define our number of `orientations`. The number of `orientations` control the number of bins in the resulting histogram. The gradient angle is either within the range $[0, 180]$ (unsigned) or $[0, 360]$ (signed). In general, it's preferable to use unsigned gradients in the range $[0, 180]$ with `orientations` somewhere in the range $[9, 12]$. But depending on your application, using signed gradients over unsigned gradients can improve accuracy.

Finally, each pixel contributes a *weighted vote* to the histogram — *the weight of the vote is simply the gradient magnitude* $|G|$ *at the given pixel.*

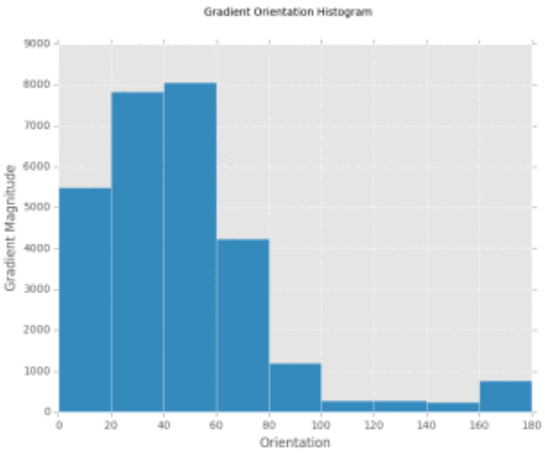


Let's make this more clear by taking a look at our example image divided up into *16 x 16* pixel cells:

And then for each of these cells, we are going to compute a histogram of oriented gradients using 9 orientations (or bins) per histogram:



Here is a more revealing animation were we can visually see a different histogram computed for each of the cells:



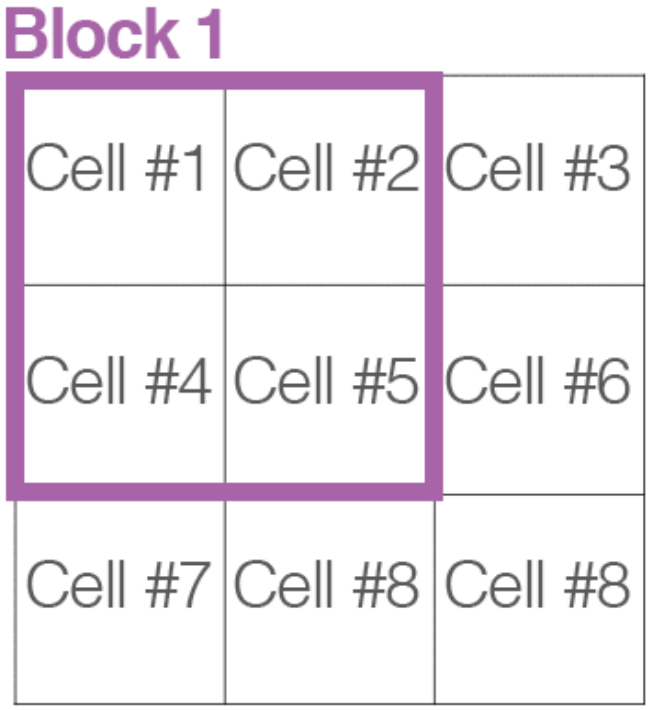
At this point, we could collect and concatenated each of these histograms to form our final feature vector. However, it's beneficial to apply block normalization, which we'll review in the next section.

Step 4: Contrast normalization over blocks

To account for changes in illumination and contrast, we can normalize the gradient values locally. This requires grouping the "cells" together into larger, connecting "blocks". It is common for these blocks to overlap, meaning that each cell contributes to the final feature vector more than once.

Again, the number of blocks are rectangular; however, our units are no longer pixels — they are the cells! Dalal and Triggs report that using either 2 x 2 or 3 x 3 cells_per_block obtains reasonable accuracy in most cases.

Here is an example where we have taken an input region of an image, computed a gradient histogram for each cell, and then locally grouped the cells into overlapping blocks:



For each of the cells in the current block we concatenate their corresponding gradient histograms, followed by either L1 or L2 normalizing the entire concatenated feature vector. Again, performing this type of normalization implies that each of the cells will be represented in the final feature vector multiple times but normalized by a different value. While this multi-representation is redundant and wasteful of space, it actually increases performance of the descriptor.

Finally, after all blocks are normalized, we take the resulting histograms, concatenate them, and treat them as our final feature vector.

Note: The sliding window for car logo detection can be understood well by the figure given below:

We have created hog feature descriptor, but now we're left to implement the hog descriptor to extract features to train a single layer that recognises logo's brand.

In the HOG descriptor, we've made, the primary size of the sliding window is 32x32. We've calculated gradient using [-1 0 1] kernel, after normalising gamma and normal contrast, and evaluated magnitude of gradient for each pixel. Then, histograms are calculated with 8 bins, and range : (0,180) based on gradient with magnitude as weights, over 8X8 pixels. Then, 2x2 cells are joined as blocks which are normalised to avoid brightness highlight differences. Then, a total of 288 features are obtained.

SVM :

We've used a Linear SVM as classification technique to classify logos in a single layer. Linear SVM has been used , because the number of features is quite higher than that of the number of classes.

Future Plans :

Finally, as we've planned, first a sliding window uses hog descriptor and searches for logo and classifies the image based on brands. Then, the relevant node from the brands hidden layer will be activated at the end of the inception model, and further classification is done based on models, in the last layer, which has been trained individually for models in each brand.

To make the project more precise and descriptive we are trying to implement the entire model by an Android app. We will give a description of the image using image captioning which covers the topic LSTM and link the sentence to the google search API to recommend cars of similar features like colour, brand and models.