DEEP VISION PIPELINE FOR SELF-DRIVING CARS BASED ON MACHINE LEARNING METHODS

by

Mohammed Nabeel Ahmed
Bachelor of Engineering, 2017, Ryerson University

A project
presented to Ryerson University
in partial fulfillment of the
requirements for the degree of
Master of Engineering
in
Electrical and Computer Engineering

Toronto, Ontario, Canada, 2017

AUTHOR'S DECLARATION

Abstract

Deep Vision Pipeline for Self-Driving Cars based on Machine Learning Methods
2017
Mohammed Nabeel Ahmed
Master of Engineering
Electrical and Computer Engineering
Ryerson University

The purpose of this thesis project is to design and implement a vision pipeline useful for self-driving cars, based on computer vision methods and deep learning frameworks. This pipeline is useful for identifying the lane, other cars in the view, as well as traffic signs. A final vision pipeline design is proposed that explores a network that can control steering based on vision input.

Firstly, the working model of computer vision techniques used are presented. The mathematical models used are explored, and implementation in source code developed. These models comprise the vision side of the pipeline.

Secondly, this report explores the deep learning models implemented as part of the pipeline. The mathematical approach is presented as well as the source code implementation. The models are industry and academia proven and their implementation is developed in detail.

The final part provides details on full pipeline architecture, and required hardware. A comprehensive discussion is made on the pipeline, the lessons learned, and future work.

Acknowledgements

I'd like to acknowledge my professor Dr. Kaamran Raahemifar, who always believed in my success and abilities, and gave me the chance to prove myself.

Dedication

Dedicated to my parents who’ve worked hard for me to get here and inspired me to achieve, and to my wife who patiently sacrificed time with me so I can dedicate myself to this work.

Table of Contents

List of Figures

Chapter 1
Introduction

Machine Learning and Artificial Intelligence have become a huge focus of the technology industry comprised of internet, software, and hardware companies. Both large companies, as well as startups have realized the potential of applying machine intelligence to existing platforms and devices. AI is used to create smarter homes, smarter personal assistants, smarter data-driven business, and finally smart cars. These systems are relying on mathematical techniques, once thought to be too difficult for computers, to be deployed on latest-generation processors that provide huge computational resources at very marketable cost.

The vision pipeline implementation in this project makes use of proven computer vision techniques relating to distortion correction, edge detection, perspective transformation, and color thresholding among others to prepare camera-view images from a driving vehicle. These are used to identify the driving lane. The implementation is tested for robustness in various road and driving conditions.

Similarly, detecting vehicles in a view requires feature extraction methods to feed into a classifier. Features are obtained using techniques such as color features, histogram of gradients (HOG), and gradient features. An SVM classifier searches a camera view image using a sliding window technique after applying the feature extraction methods and classifies vehicles and non-vehicles. This allows detection and identification of a vehicle in the view.

The use of deep neural networks, such as convolutional neural networks (CNN), is presented in detail. CNN methods are used to implement a traffic sign classifier to detect what type of sign is in each vehicle view. A few academia-proven networks such as LeNet by Yan LeCun [1], and AlexNet by Alex Krizhevsky *et al*. [2,3], are presented and used in the implementation of the traffic sign classifier. These implementations are built using TensorFlow tools in Python and a dependent framework called Keras. All the computer vision models are build using OpenCV or scikit-image Python library.

Chapter 2
Computer Vision Techniques

2.1 Colour Selection

Processing colours in a view image is important to identify important information, at the same time discarding unneeded information. Modifying the colour space helps certain machine learning algorithms to perform better [4]. This section explores the various colour processing techniques useful for identifying lanes and vehicles in a view image.

2.1.1 Colour Spaces

A colour space described a specific combination of colours to describe digital images. The most common is the RGB colour space in 3D. Any colour in the RGB space can be described by the coordinate (255, 255, 255). The R, G, and B values vary from 0 to 255 for red, green, blue.

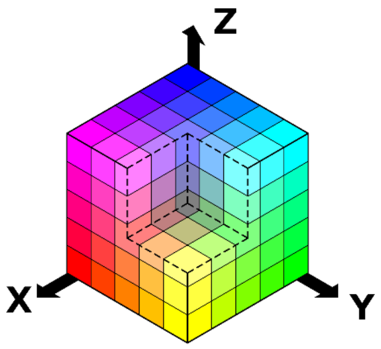


Fig. 1. RGB Colour Space Cube

There other common colour spaces are HSV (Hue, Saturation, and Value), and HLS (hue, lightness, and saturation). These are commonly used in image processing applications. The lightness and value represent ways to measure the lightness or darkness of a colour, while saturation describes the colorfulness. Lowest saturation will reduce all colours to white.

The HLS and HSV colour spaces are shown in figure 2.

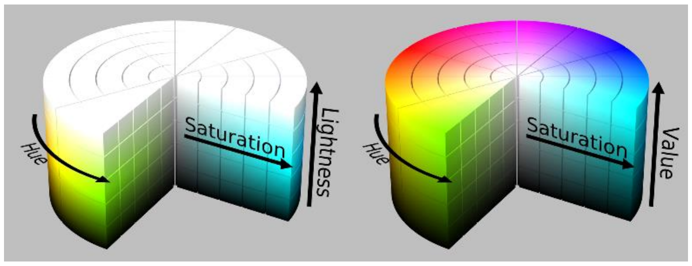


Fig. 2. HLS and HSV Colour Spaces

The following sections describe formulas and source code to convert from RGB to HLS.

2.1.2 Colour Thresholding

For many applications in image processing, converting to HLS provides better utility such as detecting lines regardless of colour. The following formulas describe the conversion from RGB to HLS.

Constants:

$V_{max} \leftarrow \max(R, G, B)$

$V_{min} \leftarrow \min(R, G, B)$

This is the minimum or maximum value across all three RGB values for a given R, G or B.

H channel conversion:

$$H \leftarrow \frac{30(G-B)}{Vmax - Vmin}, V_{max} = R$$

$$H \leftarrow 60 + \frac{30(B-R)}{Vmax - Vmin}, V_{max} = G$$

$$H \leftarrow 120 + \frac{30(R-G)}{Vmax - Vmin}, V_{max} = B$$

The H channel equations differ slightly for R, G, and B.

L channel conversion:

$$L \leftarrow \frac{Vmax+Vmin}{2}$$

S channel conversion:

$$S \leftarrow \frac{Vmax-Vmin}{Vmax+Vmin}, if\ L\ < 0.5$$

$$S \leftarrow \frac{Vmax-Vmin}{2-(Vmax+Vmin)}, if\ L\ \geq 0.5$$

OpenCV provides a very easy to use function to convert and RGB image, names *image*, to HLS colour space image, *hls*, using cv2.cvtColour [5].

```
hls = cv2.cvtColor(image, cv2.COLOR_RGB2HLS)
```

The individual H, L and S components are separated as such

```
H = hls[:,:,0]

L = hls[:,:,1]

S = hls[:,:,2]
```

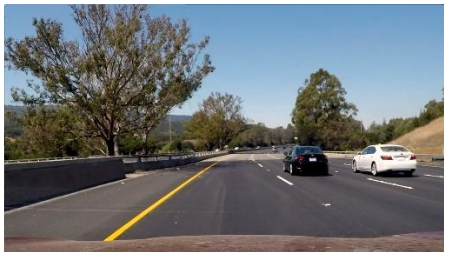


Fig. 3. Original RGB Image

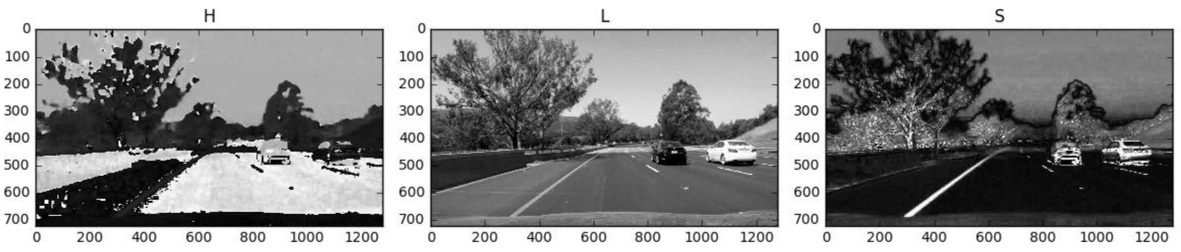


Fig. 4. H, L, and S converted image

It is evident from the S channel that the lane line is very easily distinguishable. We can threshold and binarize the image to obtain an image with much a better identifiable lane line.

```
thresh = (90, 255)

binary = np.zeros_like(S)

binary[(S > thresh[0]) & (S <= thresh[1])] = 1
```

The result is shown in figure 5.

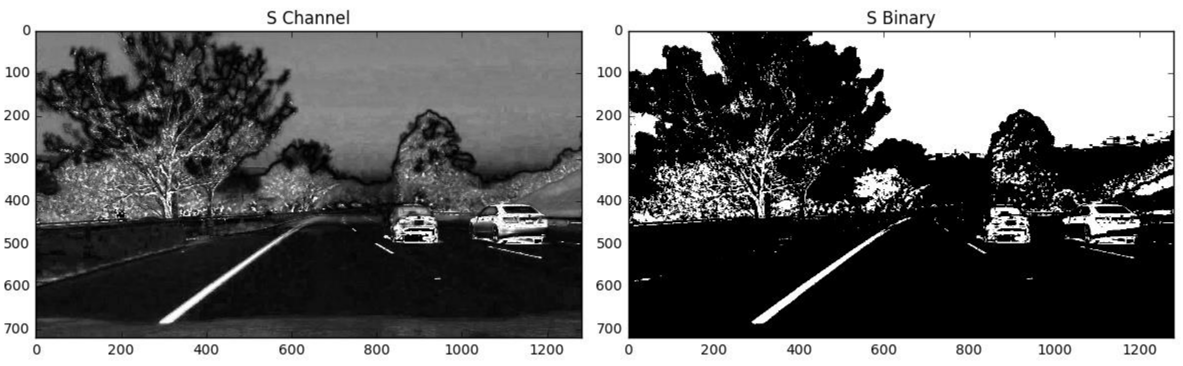


Fig. 5. Threshold and binarize S channel.

2.1.3 Gradient Thresholding

Another type of threshold that can be applied is a gradient threshold to the image to identify the lane lines. This method works by taking a derivative of the image in *x* or *y* direction. Then based on a threshold within an acceptable range of how drastic a colour change occurs (a strong edge/gradient), select either 1 or 0 to obtain a binary image.

This method will be further explored in the following section on Edge Detection.

2.2 Edge Detection

2.2.1 Canny Edge Detection

Canny algorithm [6] is a multi-stage algorithm to detect edges in an image – with useful application to detect lane lines. OpenCV provides an easy to use function that takes in a grayscale image and outputs another image with the detected edges.

The algorithm method is as follows:

1. Noise reduction using a 5x5 Gaussian filter.
2. Intensity Gradient of image by vertical and horizontal derivation.
3. Suppress non-maximum points to produce a cleaner edge in output image.
4. Hysteresis thresholding is used to decide whether a detected is really an edge based on input min and max values.

This method provides an image with well-defined edges. This requires careful choice of threshold values as needed in step 4.

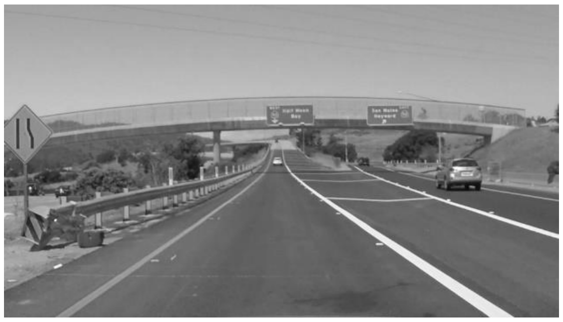


Fig. 6. Input grayscale image

OpenCV provides a function to apply Canny to an input image [7]:

```
edges = cv2.Canny(gray, low_threshold, high_threshold)
```

where *gray* is the input grayscale image, *low_threshold* is the threshold below which gradients are discarded and *high_threshold* is above which gradients are guaranteed edges. The output *edges* is the image containing the detected edges.

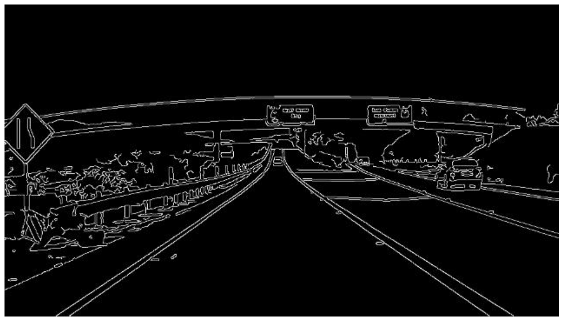


Fig. 7. Canny ‘edges’ output image

2.2.2 Sobel Operation

The Sobel operation is the core of the Canny edge detection algorithm. Sobel operation is used to take a derivation in the horizontal (*x*) and vertical (*y*) directions. The $Sobel_x$ and $Sobel_y$ operators are as follows:

$$S_x = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}$$

$$S_y = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}$$

These are Sobel operators with a 3x3 operator, which is the minimum size. Larger Sobel operators of odd kernel sizes can be used to obtain smoother gradients.

To apply Sobel with OpenCV [8], we read an image, convert to grayscale, and calculate the derivative in *x* and *y* using Sobel. We then take the absolute value, convert to 8-bit and apply threshold. The threshold only passes pixels as ‘1’ that satisfy the threshold.

The absolute value, or magnitude, of the gradient must also be calculated. This provides a combination of gradient in both directions, and is the square root of the sum of squares:

$$abs\_sobelxy = \sqrt{(sobel_x)^2 + \left(sobel_y\right)^2}$$

The Python code is as follows:

```
gray = cv2.cvtColor(im, cv2.COLOR_RGB2GRAY)
sobelx = cv2.Sobel(gray, cv2.CV_64F, 1, ksize)
sobely = cv2.Sobel(gray, cv2.CV_64F, 0, 1, ksize)
gradmag = np.sqrt(sobelx**2 + sobely**2)
scale_factor = np.max(gradmag)/255
gradmag = (gradmag/scale_factor).astype(np.uint8)
binary_output = np.zeros_like(gradmag)
binary_output[(gradmag >= mag_thresh[0]) & (gradmag <= mag_thresh[1])]=1
```

With a thresh_min = 20, thresh_max = 100, and kernel size of 3, we obtain an image as follows:

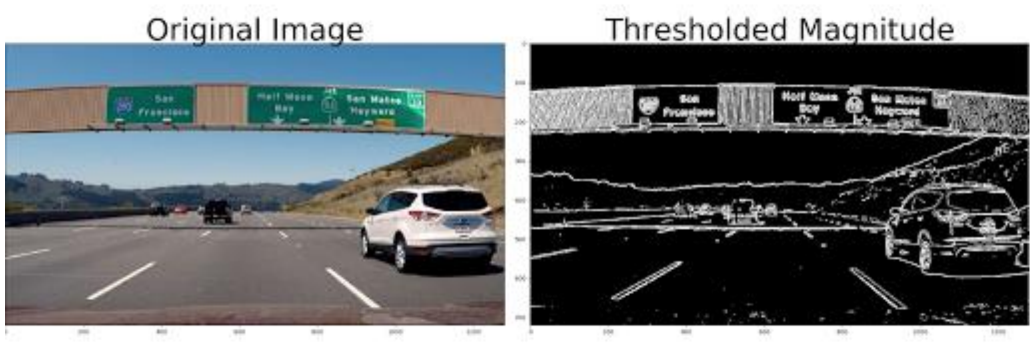


Fig. 8. Sobel operation, magnitude and thresholding results.

2.2.3 Combination of Thresholds

A combination of the colour and gradient thresholds are applied to images to obtain increased probability of identifiable lane markings in a real-time processing scenario. Since environmental, weather and road conditions can drastically increase the difficulty of finding lane marking, a combination of thresholding and gradient calculation techniques reduces chances unrecognizable lane markings in a view image. A combination of Sobel, magnitude threshold, and direction threshold produces a binary result as shown below in figure 9, where lane markings are clearly identifiable.

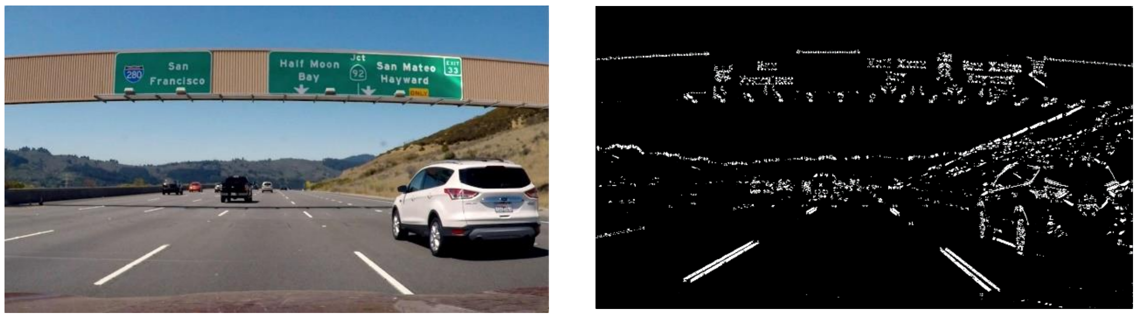


Fig. 9. **Left:** original image. **Right:** combined thresholds result.

2.3 View Transformation

2.3.1 Distortion Correction

Cameras by the nature of lens shape will distort an image slightly. This distortion can change the shape of lines and objects in a view image. For a self-driving car, this can result in computing incorrect steering angles, object detection, and recognition. For this reason, distortion correction is needed to be applied on input view images.

To correct distorted images, we need to calibrate the camera by figuring out its distortion characteristics, and then apply those to an undistort function. OpenCV provides all the necessary tools to do this [9].

We use several images of a chessboard at varying angles to compute

Firstly, we convert an RGB image of a chessboard to a grayscale image as:

```
gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
```

Find corners for an 8x6 chessboard:

```
ret, corners = cv2.findChessboardCorners(gray, (8,6), None)
```

Optionally, draw the chessboard corners (for visual purposes):

```
img = cv2.drawChessboardCorners(img, (8,6), corners, ret)
```

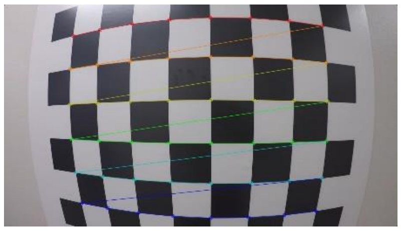


Fig. 10. Chessboard corners identified

Calibrate the camera given object points, image points and grayscale image size:

```
ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints,
gray.shape[::-1], None, None)
```

Object points are the image size of the undistorted chessboard, the image points are those returned by the findChessboardCorners function.

Finally, undistort the image:

```
dst = cv2.undistort(img, mtx, dist, None, mtx)
```

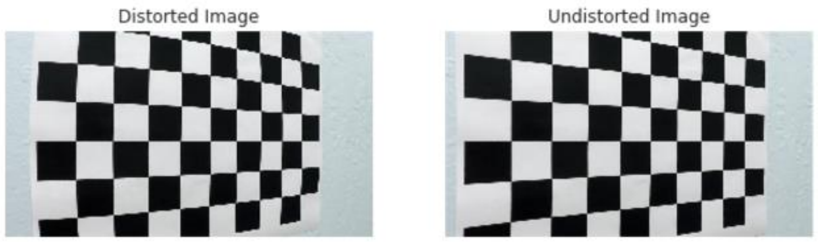


Fig. 11. Undistorting a chessboard image.

Now applying this to a view image from a vehicle, we can see the results as below:

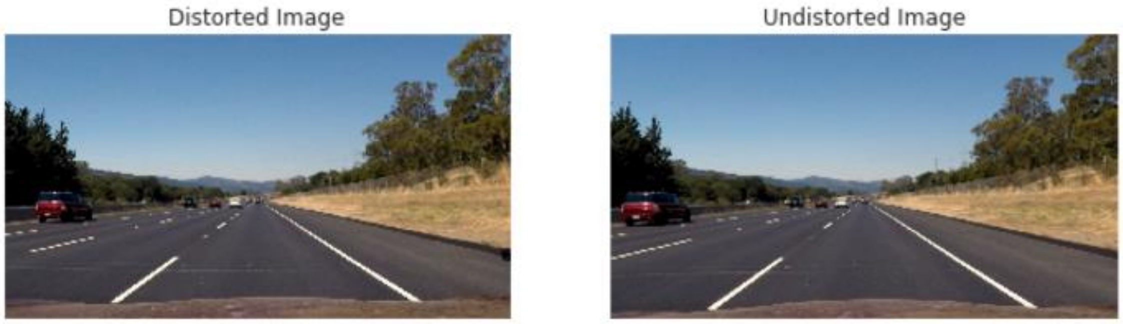


Fig. 12. Applying distortion correction to vehicle view.

2.3.2 Perspective Transform

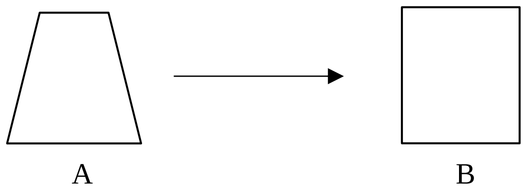
Perspective transform is very useful in transforming a view of a lane from a vehicle to a birds-eye view [10]. This is important for correctly calculating lane curvatures and identification [11].

OpenCV provides get a perspective transform for an image and apply the warp to achieve the transform [12].

First, we provide source and destination points to use for the transformation, and Secondly, use the result to warp the image and achieve the transformation. The source code is as follows:

```
M = cv2.getPerspectiveTransform(src, dst)
warped = cv2.warpPerspective(img, M, img_size, flags=cv2.INTER_LINEAR)
```

The source point are the 4 corners in A (distorted image) to the 4 corners in B (undistorted square):



Next, this is applied this to a lane view image. This makes identifying the lane lines much easier to computer as well as calculating lane curvatures as will be shown in following sections.

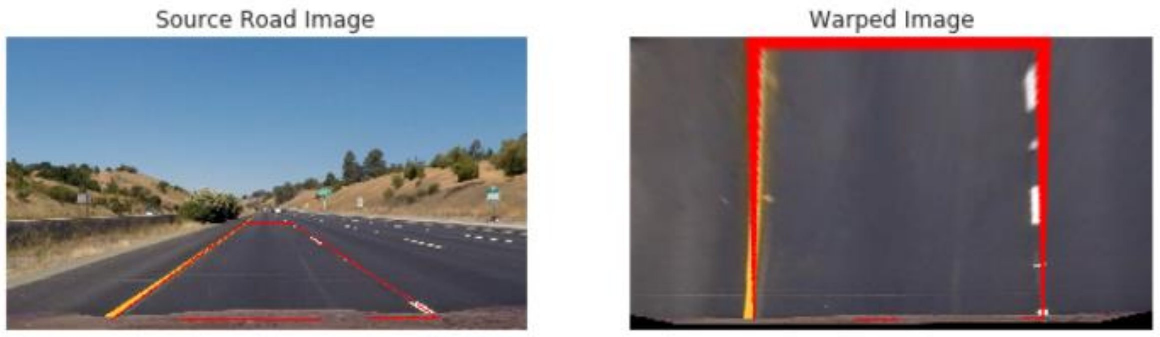


Fig. 13. Birds eye view of lane

The red highlight was applied to the image before processing to demonstrate the effect of perspective transform. In the next section, this is applied to non-highlighted view for which lane recognition and curvature calculation is carried out.

2.4 Lane Recognition

The process for lane recognition builds on all the previous sections. The steps are formalized as follows:

1. Camera Calibration
2. Distortion Correction
3. Colour & Gradient Thresholding
4. Perspective Transform
5. Lane Detection
6. Curvature Calculation

The forgoing section the techniques for camera calibration, distortion correction, colour and gradient thresholding and perspective transform were demonstrated. In this section, the resulting image from perspective transform is taken for lane detection and curvature calculation.

As a vehicle drives, the road can have turns and these need to be accounted for in lane detection as well as measuring curvature to use for steering trajectory [11]. Here we see a few examples of perspective transformed (warped) and binarized images.

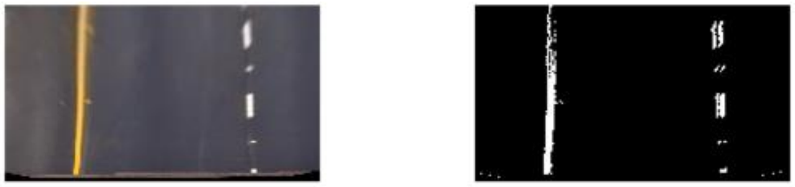


Fig. 14. Straight lane lines

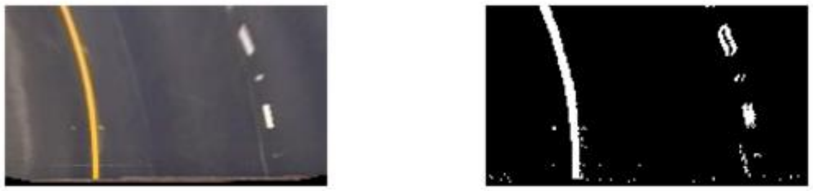


Fig. 15 Curved lane lines

To calculate the curvature of the lanes, we must first identify the lanes. This is done by taking a histogram of the bottom half of the warped and binarized image as follows:

```
histogram = np.sum(warped_image[warped_image.shape[0] // 2:, :, 0],axis=0)
```

The result is as follows for the given image:

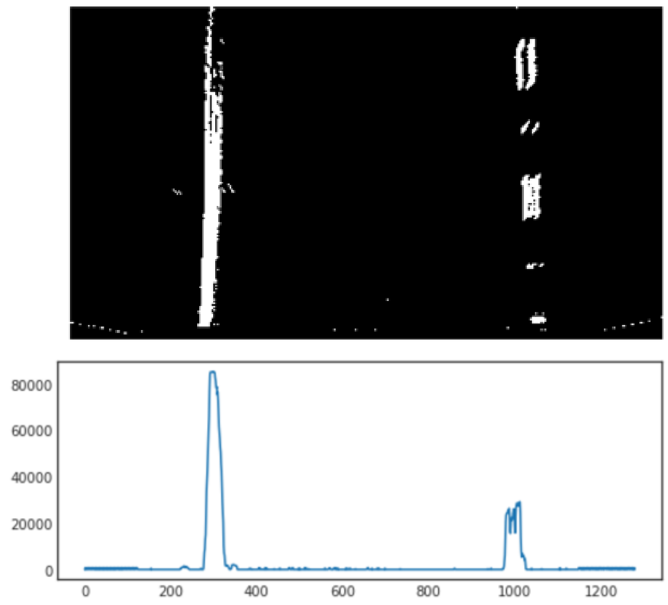


Fig. 16. Histogram of pixel intensity of bottom half of image.

Figure 16 shows two peaks in the histogram that directly describe the location and intensity of the detected lane lines. The next step is to obtain the left and right peak points by dividing the histogram in half and obtaining the peak values using argmax as follows:

```
midpoint = np.int(histogram.shape[0] / 2)
leftx_base = np.argmax(histogram[:midpoint])
rightx_base = np.argmax(histogram[midpoint:]) + midpoint
```

At this point, we are ready to identify the lane lines. Using the histogram and the *x* position of the two peaks of the bottom of half of the image as a starting point, we implement a sliding window search. A sliding window is placed about the center of the line at the bottom and follow the line up to the top of the image frame. As the window steps upwards, the number of “hot” (since pixels are 1 or 0 in binary image) pixels are calculated and the window is re-centered around their mean position. At the end of the sliding window search in a frame, the indices of all the mid-point lines of the windows are concatenated for left and right lane, and a second-order polynomial fit is extracted.

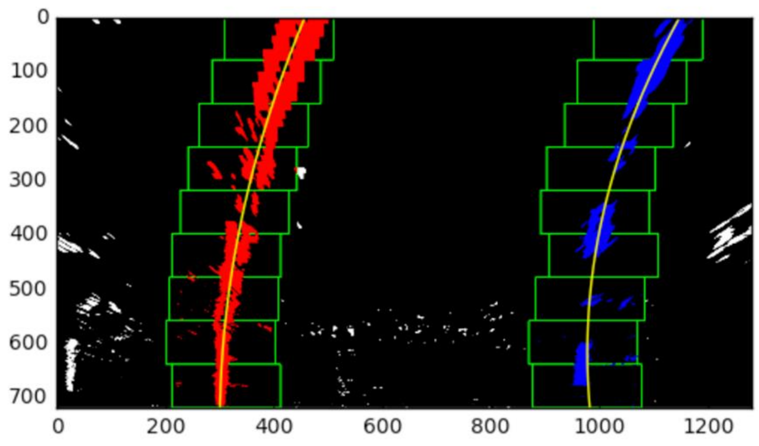


Fig. 17. Sliding window search method and 2nd-order poly-fit.

The Python method in numpy for polyfit is used as such:

```
left_fit = np.polyfit(lefty, leftx, 2)
right_fit = np.polyfit(righty, rightx, 2)
```

The final component is to measure the curvature of the polynomial fit. The polynomial fit is done in the *y*-axis as the lane lines are close to vertical and may exhibit same *x* value for multiple *y* values. The second order polynomial is as follows:

$$f(y) = Ay^2 + By + C$$

Given the above, the radius of curvature is taken as [13]:

$$R_{curve} = \frac{\left[1 + \frac{dx}{dy}^2\right]^{3/2}}{\left|\frac{d^2x}{dy^2}\right|}$$

The first and second order derivatives of the polynomial are taken as:

$$f'(y) = \frac{dx}{dy} = 2Ay + B$$

$$f''(y) = \frac{d^2x}{dy^2} = 2A.$$

Thus, the equation for radius of curvature becomes:

$$R_{curve} = \frac{(1 + (2Ay + B)^2)^{3/2}}{|2A|}$$

The radius of curvature closest to the actual car on the road would be bottom of the image at position 0. This would provide the most accurate prediction of the upcoming curvature that the vehicle should follow. This is illustrated in figure 18.

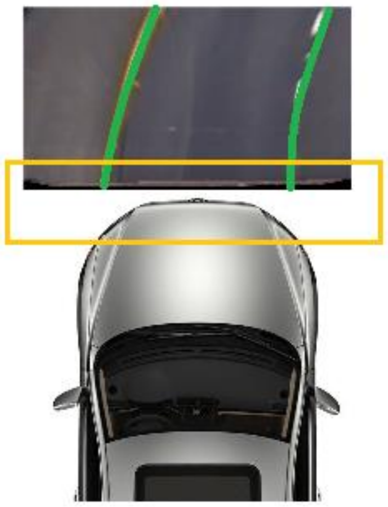


Fig. 18. Relevant curvature portion of lane.

Chapter 3
Deep and Convolutional Neural Networks

3.1 Overview of Neural Networks

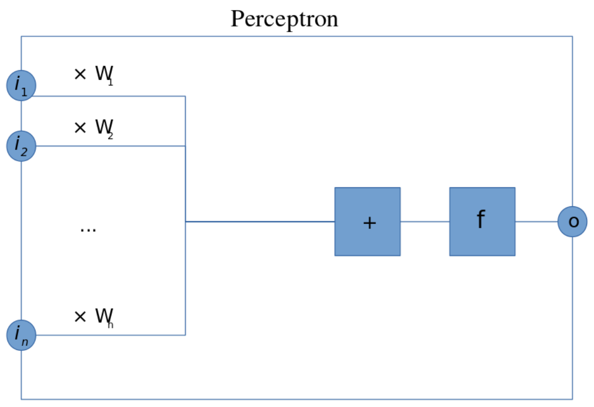
3.1.1 Simple Perceptrons

The basic unit of a neural network is called a perceptron. A perceptron takes input data and, based on a threshold, a categorization decision is the output. Many perceptrons can be connected to construct more complex decision-making networks – a basis for neural networks. A basic perceptron can be defined as:

$$f(x) = \begin{cases} 1 \; if \; w \cdot x + b > 0 \\ 0 \; otherwise \end{cases}$$

Here, *x* is the input vector, *w* is the weight vector, and *f(x)* outputs a single decision value.

weights are an important parameter in neural networks. Weights usually start as random or initial estimate values, and are refined as the neural network learns more about how inputs influence decisions during training of a network. Weighted input data is summed to a value that determines the final output decision. A perceptron may have from 1 to *n* inputs and weights that are summed as described below:



$$o = f\left(\sum_{k=1}^{n} i_k \cdot W_k\right)$$

Fig. 19. Summation of inputs and weights in a general perceptron.

The next section described a basic neural network using a few perceptrons and an activation function to create a decision based on inputs and pre-determined weights.

3.1.2 Simplest Neural Network

Perceptrons always output a 1 or 0 decision based on an input and associated weight. The diagram below described a basic neural network based on a linear combination of inputs *x*, weights *w*, and a bias input *h*, which are put through an activation function *f(h)*, which provides the final output *y*.

A basic step activation function can be described as:

$$f(h) = \begin{cases} 0 \; if \; h < 0 \\ 1 \; if \; h \; \geq 0 \end{cases}$$

Here, *h* is the output of the perceptrons as:

$$h = \sum_i w_i x_i + b$$

Using these, a basic neural network can be constructed as:

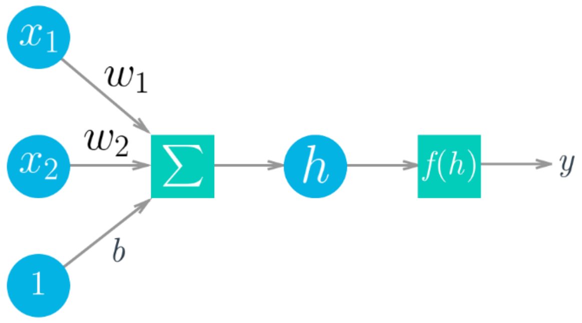


Fig. 20. Simple Neural Network

The activation function in neural network can use any function as in [14]. Most common activation functions used are sigmoid, tanh and softmax. Softmax will be explored more deeply in the section 3.2.5. Here the model for the sigmoid is shown:

$$sigmoid(x) = \frac{1}{(1 + e^{-x})}$$

3.1.3 Gradient Descent

The efficiency and accuracy of neural network operation is the careful selection of weights. As neural networks get larger and more complex, a robust approach to learning weights is needed. Weights are calculated during the training process of a neural networks. Since neural networks create predictions, the aim is to reduce the error as much as possible compared to training data the network has seen. To measure error, or the measure of how wrong a prediction is, an error function such as sum of squared errors (SSE) is used:

$$E = \frac{1}{2}\sum_u \sum_j \left[y_j^\mu - \hat{y}_j^\mu\right]^2$$

Where $\hat{y}$ is the prediction and y is the actual value, and the sum is taken over all output units j and all data points μ. The use of SSE ensures the error is always positive and larger error are penalized more than smaller errors [15].

To further expand the SSE function, we have the output of a neural network as:

$$\hat{y}_j^\mu = f\left(\sum\nolimits_i w_{ij} x_i^\mu\right)$$

Thus, the error in SSE depends on the weights and inputs as:

$$E = \frac{1}{2}\sum\nolimits_u \sum\nolimits_j \left[y_j^\mu - f\left(\sum\nolimits_i w_{ij} x_i^\mu\right)\right]^2$$

The goal thus becomes to minimize the squared error, *E*, by tuning the weights w_{ij} by using a technique known as gradient descent. Gradient descent is the process of finding the weight value or values that give the smallest global error [16], [17]. The graph below summarizes this concept, where *J(w)* = *E*.

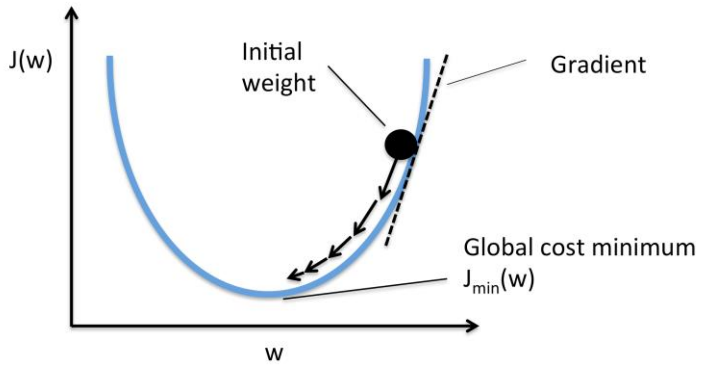


Fig. 21. Gradient descent steps [16].

The weights are updated as:

$$w_i = w_i + \Delta w_i$$

Where the new weight, w_i, is the old weight, w_i, plus the weight step Δw_i.

Now, the weight step is equal to the gradient of the SSE error function by factor of learning rate, η:

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

Here, after several steps of derivation using the chain rule, we get:

$$\frac{\partial E}{\partial w_i} = -(y - \hat{y}) f'(h) x_i$$

$$\Delta w_i = \eta (y - \hat{y}) f'(h) x_i$$

For further simplicity, an error term δ can be defined as:

$$\delta = (y - \hat{y})f'(h)$$

There are other error functions popularly used such as the mean square error (MSE).

The python code for the sigmoid function and its derivative can be defined as:

```
def sigmoid(x):
     return 1/(1+np.exp(-x))
def sigmoid_prime(x):
     return sigmoid(x) * (1 - sigmoid(x))
```

3.1.4 Backpropagation

Backpropagation is an extension of the gradient decent method of training neural networks by tuning the input layer weight parameters to reduce prediction error. For deep neural network with one or more hidden layers, backpropagation allows to update weights in hidden layers. The weights between layers determine the output of a layer, and the error from each unit is scaled by weights moving forward in the network. Using the known error at the output, we can use weights to work backwards into hidden layers in the network [17].

In an output layer, we have errors δ_k^o for each output unit *k*. Therefore, the error for each hidden unit *j* is given as:

$$\delta_j^h = \sum W_{jk}\delta_k^o f'(h_j)$$

And the gradient descent step, similar as before, is taken as:

$$\Delta w_{ij} = \eta \delta_j^h x_i$$

Where w_{ij} are weights between inputs and the hidden layer, and x_i are the inputs. This could be applied for any number of hidden layers in a neural network. The weight steps are equal to the learning rate times the output error of the hidden layer times the value of inputs to that hidden layer:

$$\Delta w_{pq} = \eta \delta_{output} V_{in}$$

Where, we get the δ_{output} parameter by propagating the errors backwards from higher layers, and V_{in} are the inputs to that hidden layer.

3.2 Deep Neural Networks

3.2.1 Rectified Linear Units (ReLu)

The foregoing section on neural networks focused on a linear model. Linear models, though powerful for many simple decision and prediction tasks, are unsuitable for more complex machine learning problems – such as self-driving cars. Adding an output layer after an activation function, and creating many such layers, results in a non-linear function and this non-linearity allows the formation of networks to solve complex problems as described in [19] and [20]

A Rectified Linear Unit (ReLu) is such a type of activation function, defined as:

$$f(x) = \max(0, x)$$

The function returns 0 if *x* is negative, otherwise returns *x*. Google's TensorFlow library provides the ReLu function as `tf.nn.relu()`, used as [21]:

```
hidden_layer = tf.add(tf.matmul(features, hidden_weights), hidden_biases)

hidden_layer = tf.nn.relu(hidden_layer)

output = tf.add(tf.matmul(hidden_layer, output_weights), output_biases)
```

An example of a ReLu nonlinearization of a linear model:

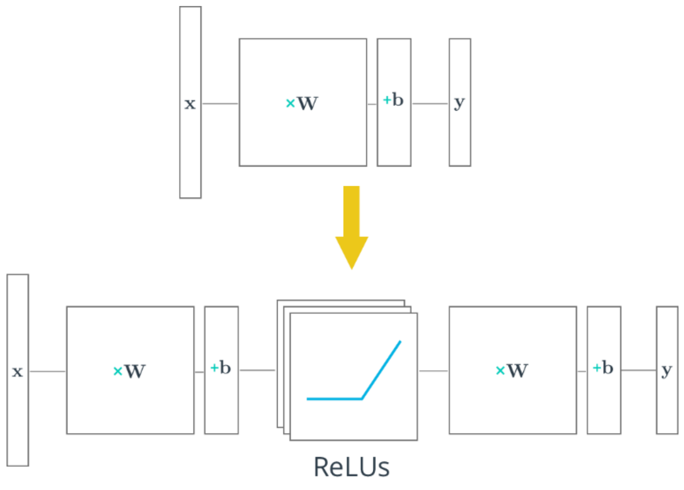


Fig. 22. ReLu applied to create a nonlinear model

3.2.2 TensorFlow Deep Neural Network

TensorFlow is under active development by Google as an open source software library for numerical computation. TensorFlow is used extensively to develop deep learning models and frameworks, and with a flexible architecture is designed to be deployable on multi-CPU or GPU computer systems.

TensorFlow is imported in to a python environment as:

```
import tensorflow as tf
```

In the following sections, several basic components of setting up a basic TensorFlow 256-layer network is setup in python code. This example sets up a pre-built network to identify hand-written digits in the MNIST database [23]. The database is imported as:

```
from tensorflow.examples.tutorials.mnist import input_data

mnist = input_data.read_data_sets(".", one_hot=True, reshape=False)
```

3.2.3 Parameters

Here we define the parameters to be used for the training of the neural network. The learning rate is kept small to avoid gradient descent issues with overshooting. The learning rate is generally accepted to be between 0.0001 and 1. Smaller learnings rates avoid overshooting problems, but can settle in local minima and take very long to converge. Larger learning rates usually cause overshooting problems. In this example, a rate of 0.001 is used. Secondly, epoch size is chosen for how many times the network should retrain before the result saturates, here we choose 20. Batch size is chosen as 128 – a memory friendly number. Lastly, the MNIST data set has images that are 28x28 in size, with 10 classes (digits 0-9). The parameter setup is done as follows:

```
learning_rate = 0.001

training_epochs = 20

batch_size = 128

display_step = 1

n_input = 784  # MNIST data input img shape: 28*28)

n_classes = 10  # MNIST total classes (0-9 digits)
```

3.2.4 Weights and Biases

Here the weights and biases are defined using TensorFlow API for the input, hidden, and output layers. A deeper network would require 2 or more hidden and output layers. The layers weights and biases are setup as:

```
weights = {
    'hidden_layer': tf.Variable(tf.random_normal([n_input,
n_hidden_layer])),
    'out': tf.Variable(tf.random_normal([n_hidden_layer, n_classes]))
}
biases = {
    'hidden_layer': tf.Variable(tf.random_normal([n_hidden_layer])),
    'out': tf.Variable(tf.random_normal([n_classes]))
}
```

For the input layer, the input MNIST data consists of images that 28x28 in size. These are reshaped to be row vectors of 1x784 size, using tf.reshape() API. Such dimensionality reduction is important for better developing and understanding the problem as explained by [22].

Finally, the hidden ReLu activation layer is created to achieve a 2-layer deep neural network:

```
# Hidden RELU activation layer
layer_1 = tf.add(tf.matmul(x_flat, weights['hidden_layer']),\
    biases['hidden_layer'])
layer_1 = tf.nn.relu(layer_1)
# Output layer with linear activation
logits = tf.add(tf.matmul(layer_1, weights['out']), biases['out'])
```

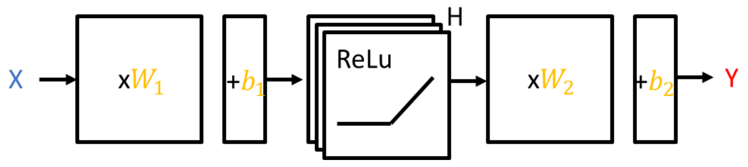


Fig. 23. Deep ReLu network

3.2.5 Loss and Optimization

As this deep neural network is used to classify 10 digits, we need a generalized form of logistic regression called SoftMax. Basically, a SoftMax will provide a probability classification for the 10 possible digits that a given input may be. This functionality is used in the TensorFlow API.

Also, the optimization method is gradient descent implemented using the TensorFlow API.

```
cost = tf.reduce_mean(\
      tf.nn.softmax_cross_entropy_with_logits(logits=logits, labels=y))
optimizer= tf.train.GradientDescentOptimizer(learning_rate=learning_rate)\
    .minimize(cost)
```

3.2.6 Session Run

At this point, the network is assembled and ready for training. The variables are initialized and graph session in TensorFlow is initialized. The graph session consists of training loops in epochs over the total batch.

```
init = tf.global_variables_initializer()
with tf.Session() as sess:
    sess.run(init)
    # Training cycle
    for epoch in range(training_epochs):
        total_batch = int(mnist.train.num_examples/batch_size)
        for i in range(total_batch):
            batch_x, batch_y = mnist.train.next_batch(batch_size)
            sess.run(optimizer, feed_dict={x: batch_x, y: batch_y})
```

This TensorFlow deep network example make use of a single hidden layer of 256 width. In a similar fashion, a much deeper network can be constructed to be deployed for more complex problems, as is often requires in self-driving car applications.

3.2.7 Dropouts

A powerful, yet simple, method to prevent deep multi-layer neural networks from over-fitting training data (and thus become less accurate to for non-training data) is to use dropouts [24]. The idea is to randomly drop units and associated connection in the network during training. This prevents the individual perceptrons over-adapting to neighbor perceptrons. While running the network and testing, keeping all units active approximates the effect of all the dropped-out networks used in the training. This significantly reduces overfitting and provides better accuracy. This technique applies very well to supervised machine learning tasks in vision and other areas of application

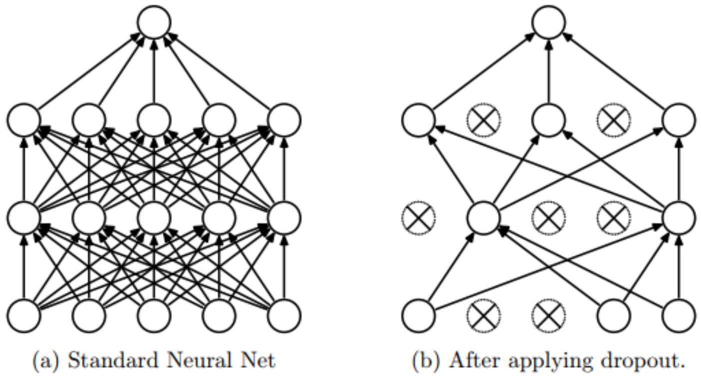


Fig. 24. Dropout model. **Left:** standard neural network. **Right:** thinned network with dropped out units.

3.3 Convolutional Neural Networks

3.3.1 Filters and Convolutions

Recognizing objects in an image – such as traffic signs, pedestrians, other vehicles, and other objects – is a very complex problem. Training a neural network to recognize objects requires very complex networks known as convolutional neural networks. An image may contain a traffic sign, pedestrian, or other vehicle anywhere in the image view, and the recognition of the object need to be the same regardless of where in the image is appears. This is referred to as translational invariance. Generally, for applications where recognition of entity is important regardless of where or how many times they appear (time or space), these are known as statistical invariants.

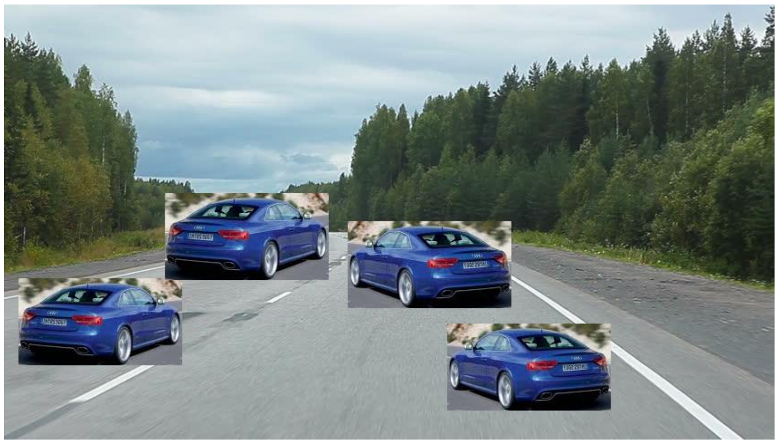


Fig. 25. Translational Invariance: object is a car, regardless of where it appears in image.

Specialized deep neural networks known as convolution neural networks (CNNs) are especially useful in these applications with spatial and statistical invariance. A convolutional filter reduces spatial information from an image to allow for better recognition of features in an image. A deep multi-layer convnet classifies components of increasingly complex shapes at each level in the hierarchy. The first level will recognize basic lines and blobs of colour, higher levels will recognize shapes with combinations of lines and features, and eventually through many layers complete objects can be recognized. A powerful such network is ImageNet proposed in [2].

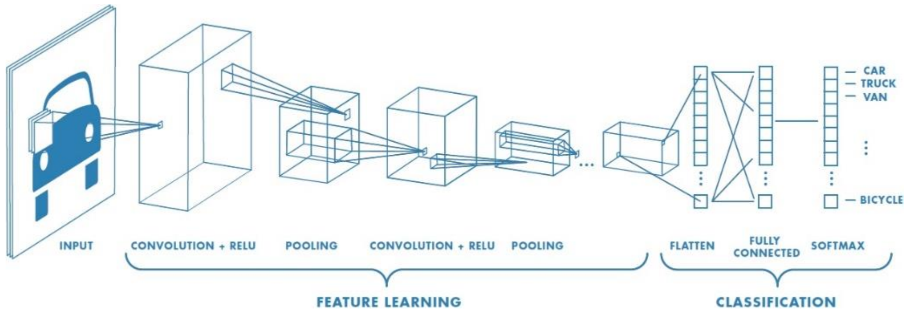


Fig. 26. Convolutional layers and final classification [25].

Essentially, the first step of a CNN is to break an image in to smaller square patches. The width of this patch, and a depth defines the first-later convolutional filter. This filter is then slid across the image by a step size hyperparameter called a 'stride'. The smaller the stride, the more accurate but larger the network becomes. Several filters of varying sizes may be used in the first layer to obtain features of different characteristics from the image. The filter depth, k, in the first layer connects to k neurons in the next layer, thus giving a height of k in the next layer. Most CNNs use similar starting filter values and are tuned as needed.

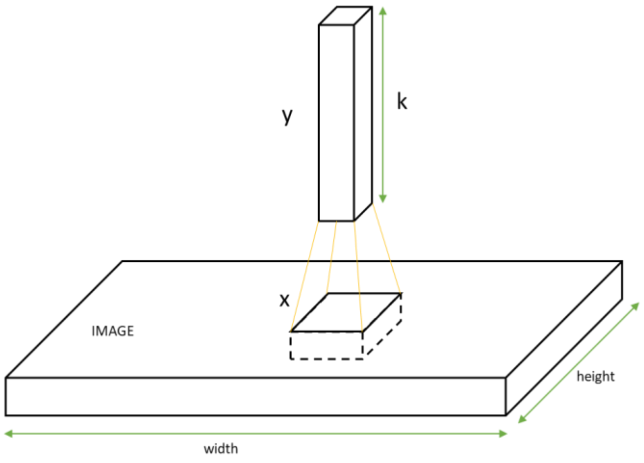
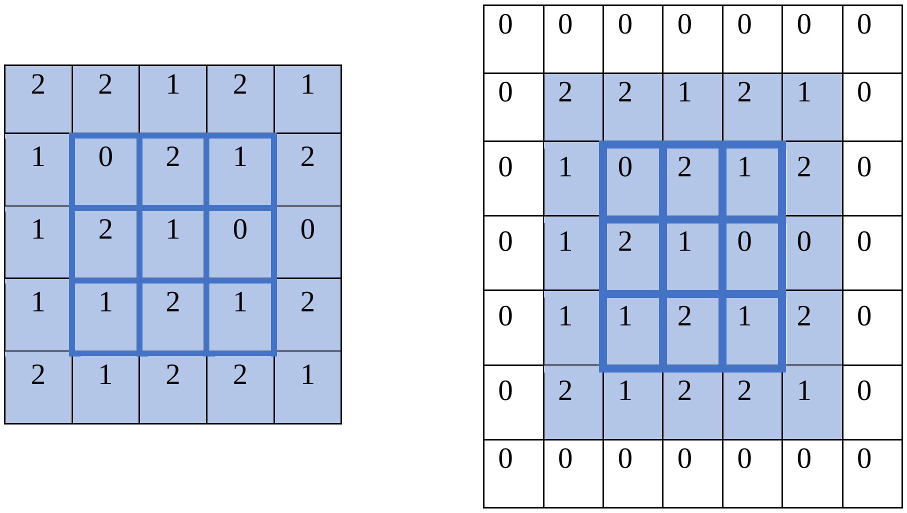


Fig. 27. CNN first layer convolutional filter of size *x, k*.

3.3.2 Parameters

The benefit of CNNs, as discussed, is the translational invariance of objects appearing anywhere in an image. The way this is possible is that the network weights and biases for a given layer are shared for all patches. The added benefit is that new weights and biases are needed to be trained for each patch, reducing training complexity and computational time. This allows more complex CNNs to be trained.



2	2	1	2	1
1	0	2	1	2
1	2	1	0	0
1	1	2	1	2
2	1	2	2	1

0	0	0	0	0	0	0
0	2	2	1	2	1	0
0	1	0	2	1	2	0
0	1	2	1	0	0	0
0	1	1	2	1	2	0
0	2	1	2	2	1	0
0	0	0	0	0	0	0

Fig. 28. **Left:** 5x5 grid, 3x3 filter, no padding. **Right:** 7x7 grid, 3x3 filter, with padding.

Stride is the step size in moving the filter across an input to produce the next layer, and the filter needs to start to start at a corner. At the corner, the filter may either start right at the edge, or overlap the edge slightly. To overlap, we add padding of 0s around the input image. If there is no overlap, the next layer is smaller thereby reducing is dimensionality, if we allow overlap, we can maintain dimensionality in the next layer. This is demonstrated in figure 26, if input grid is 5x5, with a 3x3 filter and stride of 1, we get a 3x3 in the following layer. If input grid is 7x7 with a padding of 0s, with same filter and stride, we get a 5x5 in the following layer, thus maintaining dimensionality.

Given an input image with the following parameters:

Description	Parameter
Input layer of width and height	W, H
Convolutional layer filter size	F
Stride	S
Padding	P
Number of filters	K

Table 1. CNN Parameters.

The width of the following layer is obtained as:

$$W_{out} = \frac{W - F + 2P}{S + 1}$$

Output height is:

$$H_{out} = \frac{H - F + 2P}{S} + 1$$

Where output depth is equal to number of filters as $D_{out} = K$.

3.3.3 Pooling and Max pooling

Another common method to create spatial invariance in convolutional networks is to introduce a pooling layer, the most common of which is max pooling – with added benefits of reducing computational time and prevent overfitting [26]. A pooling layer is added between CNN layers periodically and operates independently. Max pooling is commonly used with a size of 2x2 and a stride of 2. This down-samples a layer by 75% by choosing a max value in each 2x2 section of a layer. This is demonstrated in figure 27.

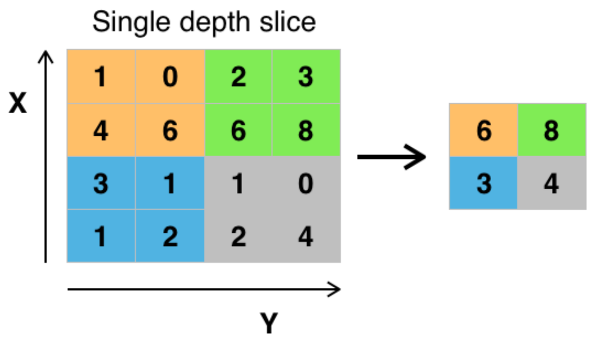


Fig. 29. Max pooling on a layer.

TensorFlow provides a function tf.nn.max_pool() to apply max pooling to convolutional layers as such:

Given a convolutional layer as `conv_layer`

```
conv_layer = tf.nn.max_pool(
    conv_layer,
    ksize=[1, 2, 2, 1],
    strides=[1, 2, 2, 1],
  padding='SAME')
```

Other pooling methods also exist such as average pooling [27], that takes an average of the numbers rather than max. Furthermore, other methods are also commonly used to reduce dimensionality, increase performance, reduce overfitting, and reduce computational time. Some of these methods are 1x1 convolutions to reduce dimensionality, and inception modules that combine several convolutions in 1 layer.

3.3.4 Convolution Network in TensorFlow

This section described implementing a basic CNN using TensorFlow in Python. This network will be a mix of convolutional layers, max pooling and fully connected layers built using resources from [28].

We begin by preparing the MNIST dataset [23] as before, by using a TensorFlow function to batch, scale, and one-hot encode (a dataset labelling method) the input data:

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets(".", one_hot=True, reshape=False)

import tensorflow as tf

# Parameters
learning_rate = 0.00001
epochs = 10
batch_size = 128

# sample size to calculate validation and accuracy
test_valid_size = 256

# Network Parameters
n_classes = 10  # MNIST total classes (0-9 digits)
dropout = 0.75  # Dropout probability
```

Now we prepare the weights and biases to be used in the CNN, and store them in variables:

```
weights = {
    'wc1': tf.Variable(tf.random_normal([5, 5, 1, 32])),
    'wc2': tf.Variable(tf.random_normal([5, 5, 32, 64])),
    'wd1': tf.Variable(tf.random_normal([7*7*64, 1024])),
    'out': tf.Variable(tf.random_normal([1024, n_classes]))}

biases = {
    'bc1': tf.Variable(tf.random_normal([32])),
    'bc2': tf.Variable(tf.random_normal([64])),
    'bd1': tf.Variable(tf.random_normal([1024])),
    'out': tf.Variable(tf.random_normal([n_classes]))}
```

Next, we define the convolution and max pooling functions:

```
def conv2d(x, W, b, strides=1):
    x = tf.nn.conv2d(x, W, strides=[1, strides, strides, 1],
padding='SAME')
    x = tf.nn.bias_add(x, b)
    return tf.nn.relu(x)
```

The max pooling function also adds padding to input image:

```
def maxpool2d(x, k=2):
    return tf.nn.max_pool(
        x,
        ksize=[1, k, k, 1],
        strides=[1, k, k, 1],
        padding='SAME')
```

Using the functions created, we can define the architecture of the CNN in a multi-layer network as follows. This network contains 2 layers with max pooling, a fully connected layer with ReLu activation and drop outs.

```
def conv_net(x, weights, biases, dropout):
    # Layer 1 - 28*28*1 to 14*14*32
    conv1 = conv2d(x, weights['wc1'], biases['bc1'])
    conv1 = maxpool2d(conv1, k=2)

    # Layer 2 - 14*14*32 to 7*7*64
    conv2 = conv2d(conv1, weights['wc2'], biases['bc2'])
    conv2 = maxpool2d(conv2, k=2)

    # Fully connected layer - 7*7*64 to 1024
    fc1 = tf.reshape(conv2, [-1, weights['wd1'].get_shape().as_list()[0]])
    fc1 = tf.add(tf.matmul(fc1, weights['wd1']), biases['bd1'])
    fc1 = tf.nn.relu(fc1)
    fc1 = tf.nn.dropout(fc1, dropout)

    # Output Layer - class prediction - 1024 to 10
    out = tf.add(tf.matmul(fc1, weights['out']), biases['out'])
    return out
```

We are finally ready to run the training session on the network, and print validation and test accuracy. This code is given in Appendix A.

Chapter 4
Traffic Sign Classifier

4.1 LeNet Architecture

LeNet is a deep CNN first proposed by LeCun in 1998 [1]. LeNet was first used for alphanumeric recognition in documents or hand-written alpha numerals. This CNN is simple but quite powerful for image recognition purposes and will be constructed for use for traffic sign recognition task with a high 90th percentile accuracy.

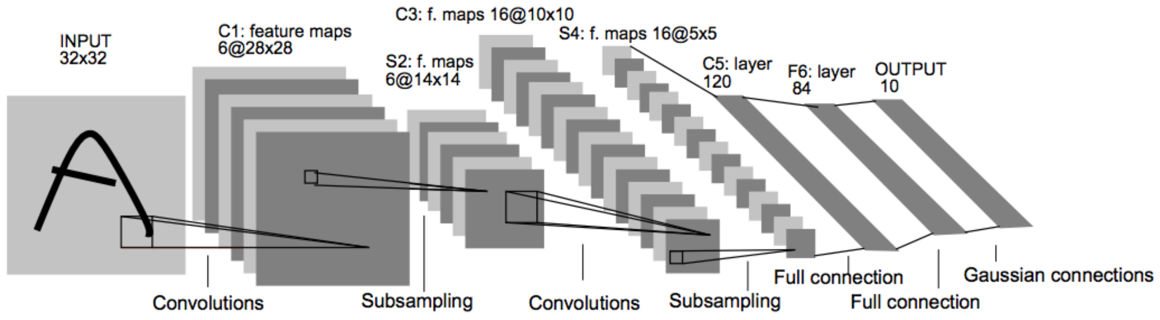


Fig. 30. Architecture of LeNet-5 proposed by Yan LeCun for digit recognition.

This network is constructed as follows for the application of traffic sign recognition using Python and TensorFlow. This LeNet architecture accepts a 32x32xC image for input, with C colour channels. The sample of traffic signs is shown below, taken from the German Traffic Sign Dataset made available by the INI institute.

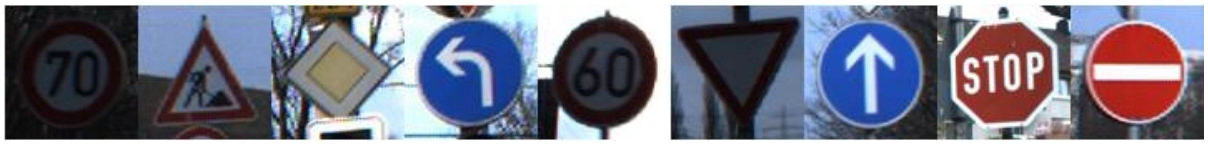


Fig. 31. Sample of traffic signs from the German Traffic Sign Dataset.

The complete LeNet architecture is described as:

Layer 1: Convolutional output with 28x28x6.

Activation: ReLu activation.

Pooling: Output with 14x14x6.

Layer 2: Convolutional output with 10x10x16.

Activation: ReLu activation.

Pooling: Output with 5x5x16.

Flatten: Flattened output to 1x400

Layer 3: Fully Connected with 120 outputs.

Activation: ReLu activation.

Layer 4: Fully Connected with 84 outputs.

Activation: ReLu activation.

Layer 5: Fully Connected Logits with 10 outputs.

The above is then implemented in Python using TensorFlow and is provided in Appendix A.2.

4.2 Training and Evaluation Pipeline

The training pipeline is setup as in the following Python code. Here, we set a learning rate of 0.001. In this pipeline, the logits output from the LeNet architecture is given to a TensorFlow function that calculated the softmax cross entropy, this provides the probability error for the image classification of traffic signs. The output error probabilities are then averaged using tf.reduce_mean(), and the loss_operation output from here is given to the optimizer to minimizer to minimize using the provided learning rate. The optimizer used is an Adam optimize – stochastic gradient optimization method developed by Kingma *et al.* [29].

```
rate = 0.001
logits = LeNet(x)
cross_entropy = tf.nn.softmax_cross_entropy_with_logits(logits, one_hot_y)
loss_operation = tf.reduce_mean(cross_entropy)
optimizer = tf.train.AdamOptimizer(learning_rate = rate)
training_operation = optimizer.minimize(loss_operation)
```

The next step is to calculate the training accuracy based on the number of correct prediction during the training and validation stages.

```
correct_prediction = tf.equal(tf.argmax(logits, 1), tf.argmax(one_hot_y,
1))
accuracy_operation = tf.reduce_mean(tf.cast(correct_prediction,
tf.float32))

def evaluate(X_data, y_data):
    num_examples = len(X_data)
    total_accuracy = 0
    sess = tf.get_default_session()
    for offset in range(0, num_examples, BATCH_SIZE):
        batch_x, batch_y = X_data[offset:offset+BATCH_SIZE],
        y_data[offset:offset+BATCH_SIZE]
        accuracy = sess.run(accuracy_operation,
        feed_dict={x: batch_x, y: batch_y})
        total_accuracy += (accuracy * len(batch_x))
    return total_accuracy / num_examples
```

4.3 Train LeNet Model

At this point, we are ready to train the model for 10 Epochs and achieve a validation accuracy of 94%. This could be improved by longer training time and better image data preparation. The validation is done using the x_validation data and associated y_validation labels split from training.

```
with tf.Session() as sess:
    sess.run(tf.initialize_all_variables())
    num_examples = len(X_train)

for i in range(EPOCHS):
    X_train, y_train = shuffle(X_train, y_train)
    for offset in range(0, num_examples, BATCH_SIZE):
    end = offset + BATCH_SIZE
    batch_x, batch_y = X_train[offset:end], y_train[offset:end]
    sess.run(training_operation, feed_dict={x: batch_x, y: batch_y})

    validation_accuracy = evaluate(X_validation, y_validation)
```

4.4 Evaluate Model

Finally, the model is evaluated using the test set of 12,630 images. The training data set was a size of 39,209 images split into 80% for training and 20% for validation as described in previous section. Here we evaluate the model on the 12,630 images:

```
   test_accuracy = evaluate(X_test, y_test)
```

Chapter 5
Support Vector Machines and Vehicle Recognition

5.1 Support Vector Machines (SVM)

Support vector machines (SVMs) are a class of supervised learning models associated with data analysis involving classification. SVMs can classify samples in many dimensions and highly versatile using various kernel types. This class of classifiers are useful for recognizing cars in an image view, as well as pedestrians, types of cars, etc. In this implementation, we will be recognizing cars in a view image. We also require several computer vision methods to extract features from the image to classify cars so these methods are also developed.

Specifically, SVMs classify based on a separating hyperplane. During supervised learning using labelled training data, the algorithm creates a maximal distance hyperplane to best separate the classes of data [30]. Given the various dividing lines that are possible as shown in figure 32(a), the algorithm finds the optimal hyperplane as in figure 31(b). In higher-order classifications, the hyperplane may be higher-order and more than one would be needed as shown in figure 33.

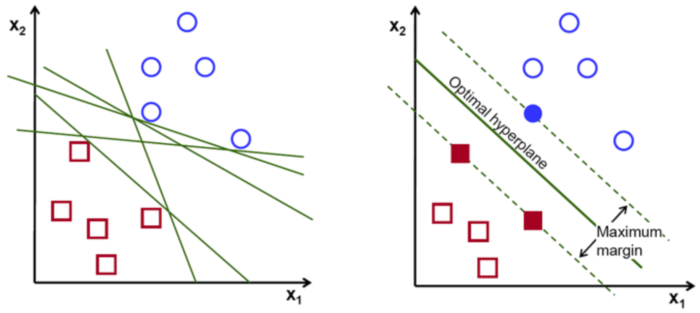


Fig. 32. (a) possible hyperplanes. (b) optimal hyperplane.

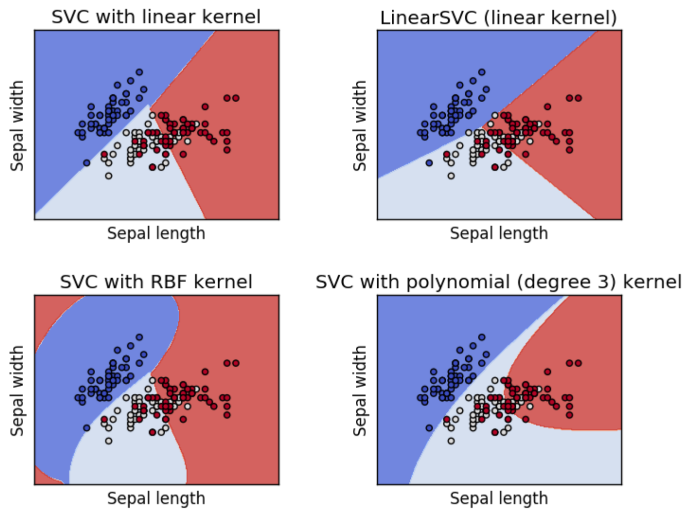


Fig. 33. SVM with higher degree kernels yield better classification.

In this implementation, we use the scikit-learn Python library to implement and train the SVM classifier, and scikit-image as well as OpenCV for the various computer vision techniques to be implemented [31], [32].

5.2 Colour Histogram Features

A histogram of colour pixel intensities as features can define an object of a specific type. For example, all white cars, or red cars, or cars of any colour should share a similar histogram features for the same colour. However, there are hundreds of difference colours a car can be, and other random objects in a view image can satisfy a histogram match, so this feature alone cannot be relied upon. However, histograms of colour intensities can be used as a part of a feature set for vehicle classification.

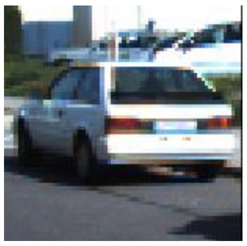


Fig. 34. Example car image

Given a vehicle image as in figure 34, the histogram of RGB colour channels can be calculated as:

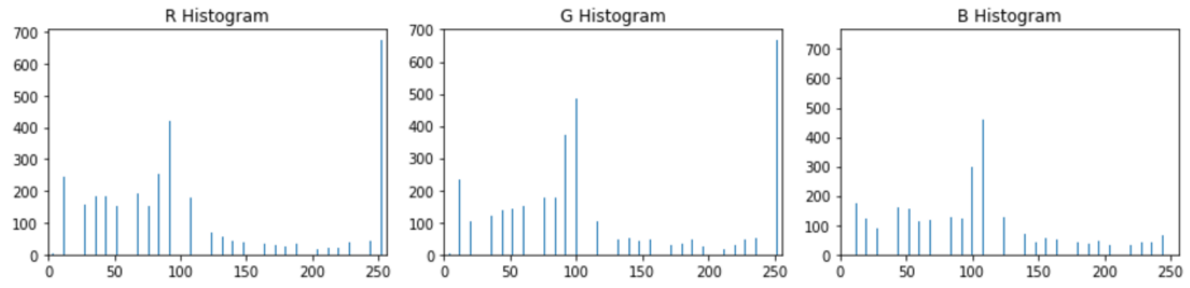


Fig. 35. Colour histogram channels for figure 33 car image.

This is done using Python Numpy library as:

```
def color_hist(img, nbins=32, bins_range=(0, 256)):
    # Compute the histogram of the color channels separately
    channel1_hist = np.histogram(img[:,:,0], bins=nbins, range=bins_range)
    channel2_hist = np.histogram(img[:,:,1], bins=nbins, range=bins_range)
    channel3_hist = np.histogram(img[:,:,2], bins=nbins, range=bins_range)
    # Generating bin centers
    bin_edges = channel1_hist[1]
    bin_centers = (bin_edges[1:]  + bin_edges[0:len(bin_edges)-1])/2
    # Concatenate the histograms into a single feature vector
    hist_features = np.concatenate((channel1_hist[0], channel2_hist[0], 
channel3_hist[0]))
    # Return the individual histograms, bin_centers and feature vector
    return channel1_hist, channel2_hist, channel3_hist, bin_centers, 
hist_features


rh, gh, bh, bincen, feature_vec = color_hist(car_image, nbins=32, 
bins_range=(0, 256))
```

This code calculates the histogram of the three colour channels, the bin centers, and concatenates the histograms into a single feature vector, useful later. The histograms are plotted using the matplotlib library in Python.

Histograms could also be calculated in alternate colour spaces such as HLS which may give better insight where cars have more saturated colours than the background. Also, histograms may help differentiate cars vs. non-car objects in a view.

5.3 Gradient and HOG Features

Using colour only captures a limited aspect of a colour image. However, images of similar structure but different colours can be better captured using gradients. The direction of gradients in an image better captures the shape of the object in an image. If the gradient of a shape image is split into cells and vectorized into a 1-dimension array, this can provide a signature for the class of objects with similar shape.

A robust method for object classification was developed by Dalal *et al.* [33], where gradient of an image, figure 36(a), is taken and the gradient magnitude and direction for each pixel is calculated, shown in figure 36(b). The individual pixels are then grouped into smaller cells of 8x8 pixels, as shown in figure 36(c-d). Next, the histogram of gradient orientations is calculated for each cell in the image, achieving the result shown in figure 36(e). The histogram is distributed into orientation bins of, for example, 9 directions from about -160^o to +160^o. We add up the individual orientation contributions from each pixel in a cell in a weighted vote to obtain a start map with arrows of different length – as shown in figure 36(e). Strong gradients contribute more to the map than small random gradients, thereby reducing contributions from noise. This gives us the Histogram of Oriented Gradients (HOG). A HOG is robust to variations in shape and provides a great object, in this case vehicle, detection technique.

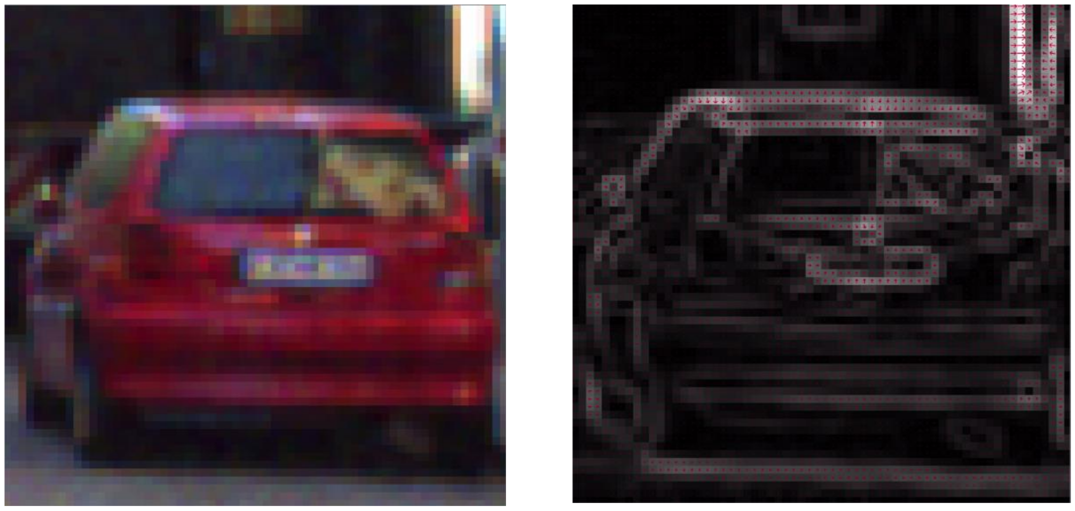

(a) (b)

Fig. 36. (a) Original image. (b) Gradient magnitude and direction of each pixel.

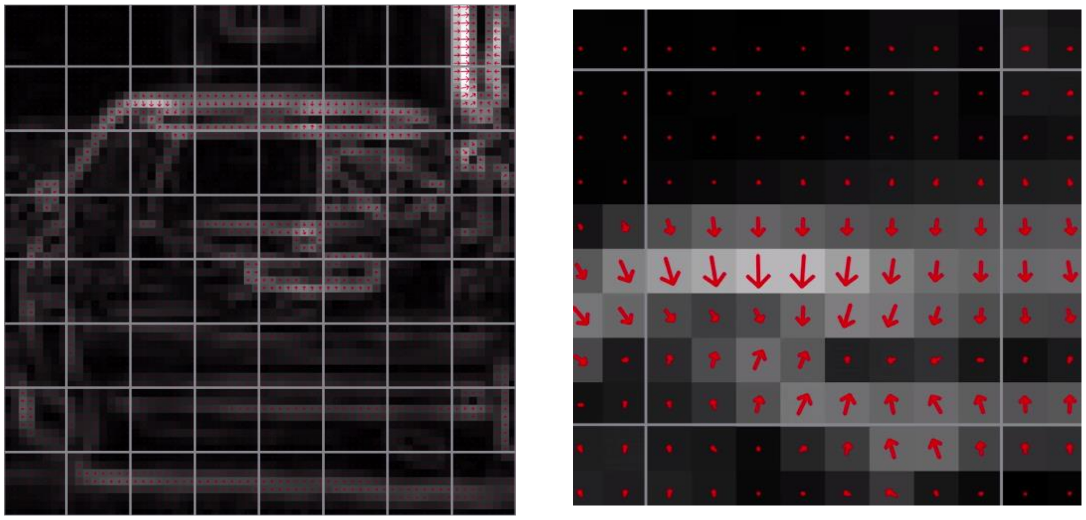

(c) (d)

Fig. 36. (c) grouped cell of 8x8 size. (d) zoom-in of a cell with individual orientation gradients.

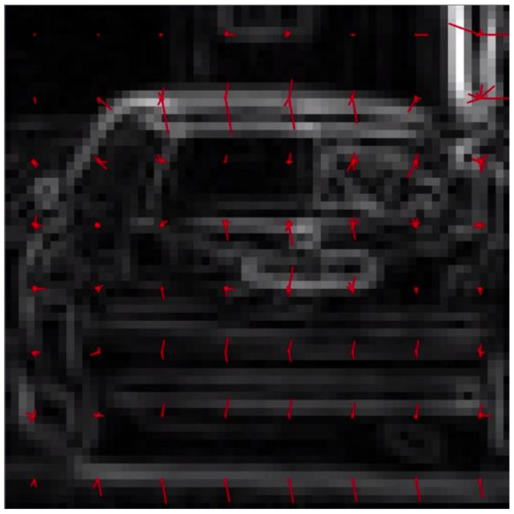


Fig. 36. (e) Final Histogram of Oriented Gradients image.

We utilize the scikit-image library skimage in Python to obtain the HOG features form an image. Several parameters are needed to be provided such as:

- Image: input image
- Orientation: number of orientation bins
- Grid Cells: number of grid cells
- Pixels per cell, and finally cells per block.

The source is given as follows.

```
from skimage.feature import hog


# Function returns HOG features and output image.
def get_hog_features(img, orient, pix_per_cell, cell_per_block, vis=False,
feature_vec=True):
    # Use skimage.hog() to get both features and a visualization
    features, hog_image = hog(img, orientations=orient,
pixels_per_cell=(pix_per_cell, pix_per_cell),
    cells_per_block=(cell_per_block, cell_per_block), visualise=vis,
feature_vector=feature_vec)
    return features, hog_image
# Grayscale car image
gray = cv2.cvtColor(car_image, cv2.COLOR_RGB2GRAY)
# HOG parameters
orient = 9
pix_per_cell = 8
cell_per_block = 2
# Call get_hog_features.
features, hog_image = get_hog_features(gray, orient, pix_per_cell,
cell_per_block, vis=True, feature_vec=False)
```

This provides an output as figure 37(b) for a given input image figure 37(a).

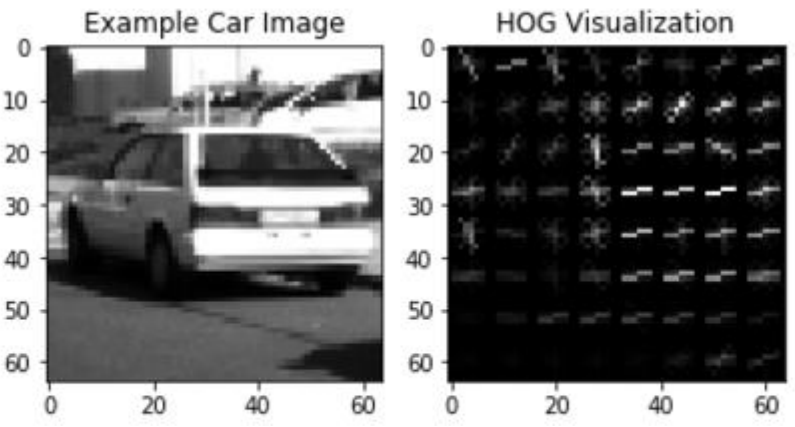


Fig. 37. (a) input grey car image. (b) Output HOG feature visualization

5.4 Combine and Normalize Features

To obtain more robust models for classification, we combine multiple features into a feature set. This may include colour and shape based features. We combine colour-space features such as HSV, with HOG features, and concatenate the feature vector. One problem that arises is that the features are different quantities and larger number may improperly skew results. We require scaling the features magnitudes to avoid dominating effects of one feature over others.

The Python library sklearn provides a scaler method called standardScaler() to normalize data for machine learning tasks [34]. Before supplying this method with a feature vector, the data needs to be in a format where each row is a single feature vector. This is done using the following method:

```
  import numpy as np
  feature_list = [feature_vec1, feature_vec2, ...]
  # Create array stack, NOTE: StandardScaler() expects np.float64
X = np.vstack(feature_list).astype(np.float64)
```

We are then ready to use the standardScaler() method as:

```
from sklearn.preprocessing import StandardScaler
# Fit a per-column scaler
X_scaler = StandardScaler().fit(X)
# Apply the scaler to X from np.vstack earlier.
scaled_X = X_scaler.transform(X)
```

5.5 Colour and HOG Classification

Here the colour histogram feature and HOG feature extractions developed earlier will be combined and used for SVM training. The SVM will be trained to classify car vs. non-car images. The colour and HOG features will be extracted for each, scaled to zero mean and 1 variance.

In the pipeline follows first by doing a colour space conversion, we then apply spatial binning to obtain spatial colour features, next we obtain the colour histogram features, and finally we obtain the HOG features of the image. Features from each stage are appended to a features vector named *combined_features* which is concatenated and returns list of feature vectors. Lastly, the features vector is scaled.

The source code process is as follows. The fully implemented function is provided in Appendix A.3.

```
feature_image = cv2.cvtColor(image, cv2.COLOR_RGB2HSV) #colour space
# Obtain spatial colour features
spatial_features = bin_spatial(feature_image, size=spatial_size)
combined_features.append(spatial_features) #append spatial features
# Obtain histogram features
hist_features = color_hist(feature_image, nbins=hist_bins,
                bins_range=hist_range)
combined_features.append(hist_features) #append Histogram features
for channel in range(feature_image.shape[2]):
    # Obtain HOG Features
    hog_features.append(get_hog_features(feature_image[:,:,hog_channel],
                                          orient, pix_per_cell,
                                          cell_per_block,
                                          vis=False, feature_vec=True))
combined_features.append(hog_features)
features.append(np.concatenate(combined_features)) #concatenate and append
      # Create an array stack of feature vectors
      X = np.vstack((car_features, notcar_features)).astype(np.float64)
      X_scaler = StandardScaler().fit(X) # Fit a per-column scaler
      scaled_X = X_scaler.transform(X) # Apply scaler to X
```

At this point, the svc fitting is done using the training data set, and accuracy is tested with a testing data set. For this, the car and *not_car* features are extracted, feature scaling is done, and the data is split into training and test sets randomly. The full implementation is provided in Appendix A.4.

```
car_features = extract_features() #extract all car image features
notcar_features = extract_features() #extract all  non-car image features
# Create an array stack of feature vectors
X = np.vstack((car_features, notcar_features)).astype(np.float64)
# Fit a per-column scaler
X_scaler = StandardScaler().fit(X)
# Apply the scaler to X
scaled_X = X_scaler.transform(X)
# Define the labels vector
y = np.hstack((np.ones(len(car_features)),
np.zeros(len(notcar_features))))

# Split up data into randomized training and test sets
rand_state = np.random.randint(0, 100)
```

```
X_train, X_test, y_train, y_test = train_test_split(
   scaled_X, y, test_size=0.2, random_state=rand_state)
# Use a linear SVC, fit, score, and predict using the SVC
svc = LinearSVC()
svc.fit(X_train, y_train)
svc.score(X_test, y_test)
svc.predict(X_test[0:n_predict]) #for labels in y_test.
```

5.6 Sliding Window

In the preceding sections, the techniques and code are presented to be used for vehicle detection in a front view image. To detect a vehicle in an image, a sliding window search method needs to be implemented to search for vehicles, with modification as shown in [35]. This requires deciding a region of interest in the image, the minimum and maximum size of vehicles to be detected in the view, and how the sliding window should behave. For each window portion collected for analysis, it is processed and used for classification. This is done by a vehicle vs. non-vehicle SVM classifier as developed in section 5.1.

The sliding window method will search only the bottom half of the image where cars would normally appear – thereby saving much time and computational effort searching unneeded areas. Secondly, Vehicles will appear larger when they're closer, and become progressively smaller closer to the horizon vanishing point. To account for this, sliding windows of a minimum and maximum size will be decided along with number of intermediary sizes. Thirdly, we may choose to overlap search windows and decide the sliding step size. These factors will determine both the accuracy and the speed of processing in inverse proportion. The concept is illustrated in figure 38.

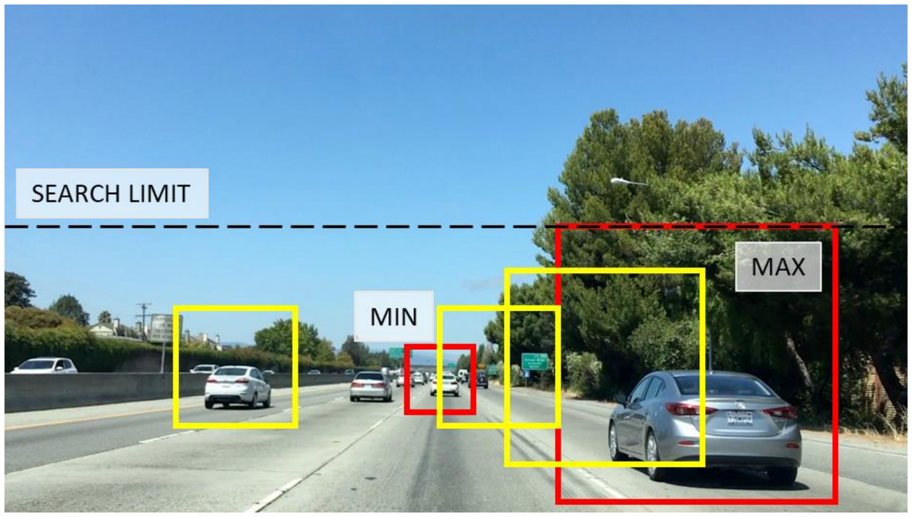


Fig. 38. Multiple sliding windows and region of interest

As the various windows slide over the image, the portion of the window is processed for feature extractions so far developed such as: colour space conversion, colour histogram, colour spatial bins and HOG features. Once a feature vector is obtained, this is passed to a trained SVM for classification. A faster method would be to pre-HOG feature extract the entire image, before doing sliding window search and extracting the colour and histogram features.

Since we've already developed function to extract image feature in extract_features() as defined in Appendix A.3, we will need to add a function for the sliding windows, and another function to search each sliding window for features using extract_features(). We define two functions: slide_windows() and search_windows(), where the output of slide_windows() is passed to search_windows() as well all parameters required for extract_features(). The sliding_windows() function is provided in Appendix A.5.

The search_windows() function captures the process of processing all incoming windows, extracting the window portion of original image, passing for feature extraction, scaling features, classification using SVM, and passing all positive detection. The positive detections are then used for heat mapping and removing false detection explained in the following section.

```
def search_windows(img, windows, clf, scaler, color_space='RGB',
                    spatial_size=(32, 32), hist_bins=32,
                    hist_range=(0, 256), orient=9,
                    pix_per_cell=8, cell_per_block=2,
                    hog_channel=0, spatial_feat=True,
                    hist_feat=True, hog_feat=True):

    # 1. Empty list to store positive detection windows
    on_windows = []
    #2. Iterate all sliding windows
    for window in windows:
        #3. Extract window portion from original image
        test_img = cv2.resize(img[window[0][1]:window[1][1],
        window[0][0]:window[1][0]], (64, 64))
        #4. Extract features for window using extract_features()
       features = extract_features(test_img, color_space=color_space,
                            spatial_size=spatial_size, hist_bins=hist_bins,
                            orient=orient, pix_per_cell=pix_per_cell,
                            cell_per_block=cell_per_block,
                            hog_channel=hog_channel, spatial_feat=spatial_feat,
                            hist_feat=hist_feat, hog_feat=hog_feat)
        #5. Feature scaling
        test_features = scaler.transform(np.array(features).reshape(1, 1))
        #6. Classify using SVM
        prediction = svc.predict(test_features)
        #7. If positive (prediction == 1) save window
        if prediction == 1:
            on_windows.append(window)
    #8) Return windows
    return on_windows
```

5.7 Positive Detection

Finally, when the vehicle detection method as outlined is run, we may have multiple detections of a vehicle. This is a result of the multiple size windows search detecting the car multiple times, as well as false positives as shown in figure 39. Multiple detection can be combined into one by creating a “heat-map” system whereby higher number of detection in an overlapping region create higher likelihood of a correct prediction. If a false positive is found without sufficient number of multiple detection, dictated by a decided threshold, the detection in that region is discarded. The size of the detection is decided by drawing a square over the “heat blobs” in the heat map. This gives an approximate of the detected vehicle size, as shown by the final rectangle in figure 40.

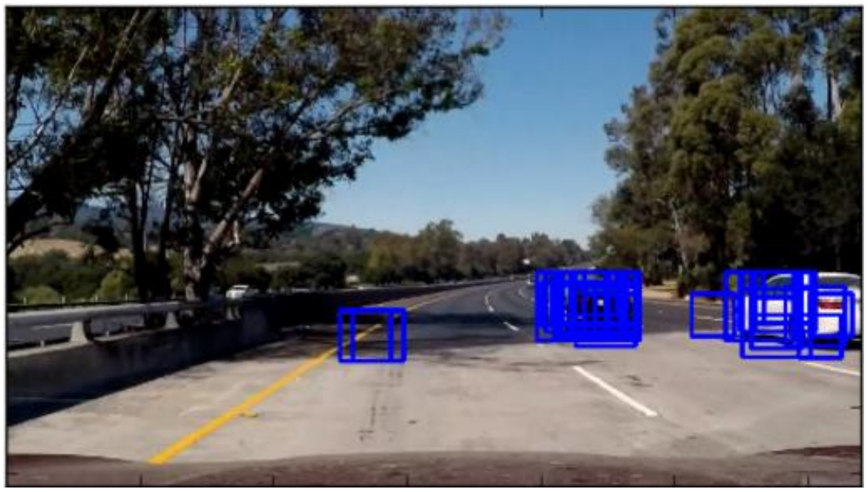


Fig. 39. Multiple detection and false positives

We define a add_heat() function that inputs a list of multiple detections (as drawn boxes) and creates a heat map. This is then fed to a thresholder function apply_threshold() to filter any detections below the threshold – thereby reducing false detections.

The source is as follows:

```
def add_heat(heatmap, detect_list):
    # Iterate through detect_list
    for box in detect_list:
        # Add += 1 for all pixels inside detection
        # box is form ((x1, y1), (x2, y2))
        heatmap[box[0][1]:box[1][1], box[0][0]:box[1][0]] += 1

    # Return heatmap
    return heatmap
```

```
def apply_threshold(heatmap, threshold):
    # discard pixels below the threshold
    heatmap[heatmap <= threshold] = 0
    # Return thresholded heatmap
    return heatmap
```

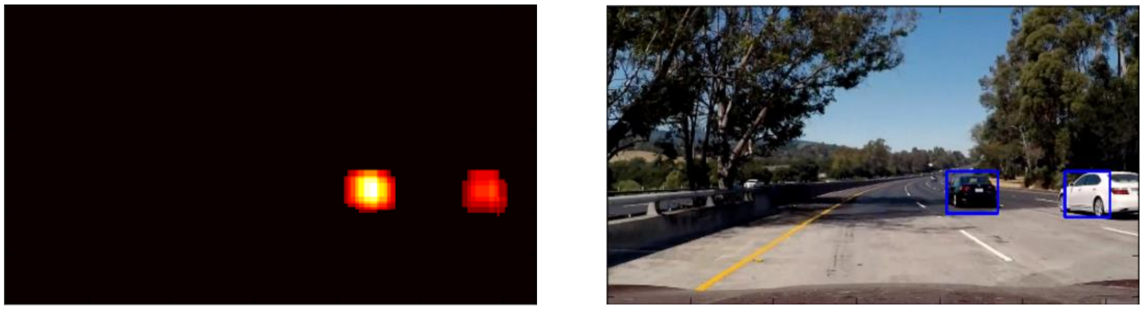


Fig. 40. **Left:** Heap map filter. **Right:** single detection from heat map pixels

The images in figure 40 show the heat map created by multiple detections in the same general area. Any detections below a threshold such as 10 continuous detections, are discarded. A rectangle is drawn over the "hot" areas to create the detection as shown on the right in figure 40.

Chapter 6

Combined Pipeline and Hardware

6.1 Pipeline Architecture

Building on the techniques and methods developed so far in this report, a full-featured pipeline can be assembled that recognizes road lanes, traffic signs, and other vehicles. Further, training such a system while including steering angles and road view image feed, the system learns to drive autonomously as demonstrated in [36]. In this section, an architecture is presented for training a system to drive using camera feed and steering wheel angles only as implemented in [36], and then such a system is expanded to include the developed techniques for lane detection, traffic sign recognition, and vehicle detection.

To train a CNN to drive a vehicle based on camera images and steering angles, a data collection method is used as described in figure 41. Here, the steering angle is recorded from training data and adjusted for vehicle geometry by converting to $^1/_r$ instead of using r directly. Next, 3 cameras are used to collect road view images from right, center, and left views, which are then randomizes, shifted, rotated, jittered to create a more general data set and prevent overfitting. Finally, a CNN is trained by giving it the image data to output a steering angle command, this command is compared with the actual command from training data and used to adjust waits in the network to reduce the error using back propagation.

The CNN used in [36] comprises of 3 fully-connected layers, 4 convolutional layers, and normalization and input planes. This totals 27 million connections and 250 thousand parameters. The method is shown in figure 41.

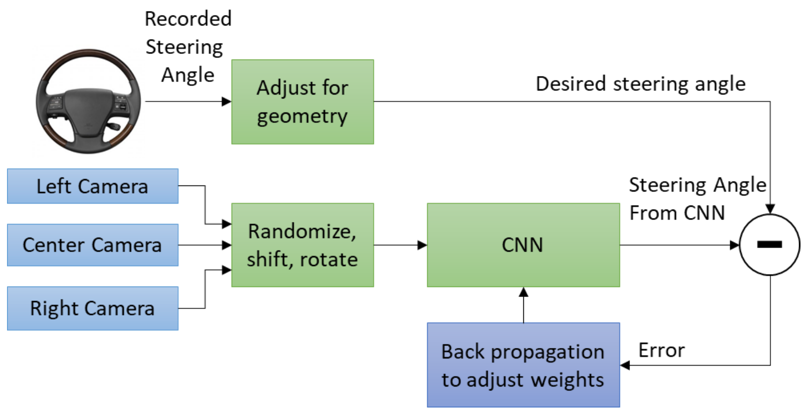


Fig. 41. Training the CNN to control steering.

The method in figure 41 can be modified to include the lane detection and lane curvature calculation for more accurate steering wheel angle inferencing. Here, we use image processing techniques developed for the pipeline to calibrate the camera, apply distortion correction, extract colour and gradient thresholding images, apply perspective transform and binarize image, and finally use a histogram to detect lanes and calculate the curvature using polynomial method presented. This information can then be fed to the same CNN as in method in figure 41 and used for steering angle prediction.

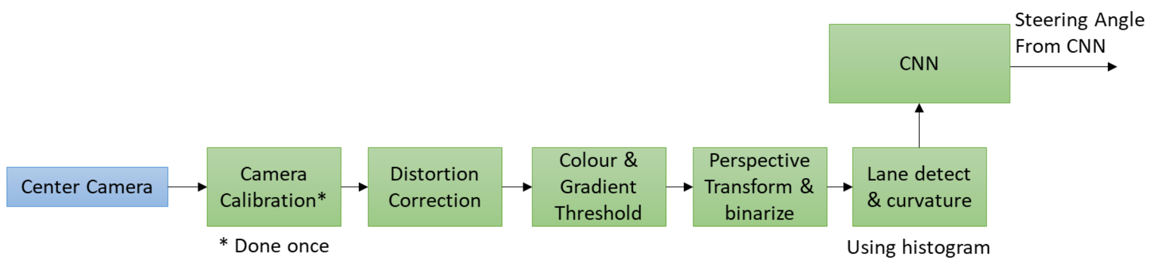


Fig. 42. Lane detection and curvature calculation.

Additionally, in figure 42, we may apply the same pipeline for right and left camera to identify all lanes in a multi-lane setting such as a highway – though these would not be used for steering angle calculation.

Next, we create the pipelines for traffic sign detection. Here, traffic sign images are used to generate additional data with the same labels by jittering, transforming, rotating the original images. This creates more generalized data and prevents overfitting. In a real-world scenario traffic signs may appear in a view in many different conditions and angles and the network should still be able to recognize these. Next, these images are grayscaled, normalized and one-hot encoded with labels. These are then fed to the LeNet-5 architecture, the output logits of which feed a cross-entropy and Adam optimizer to train the network. Each pixel of the traffic sign image (32x32xC) is treated as a single input to the LeNet-5 architecture, which is a CNN architecture.

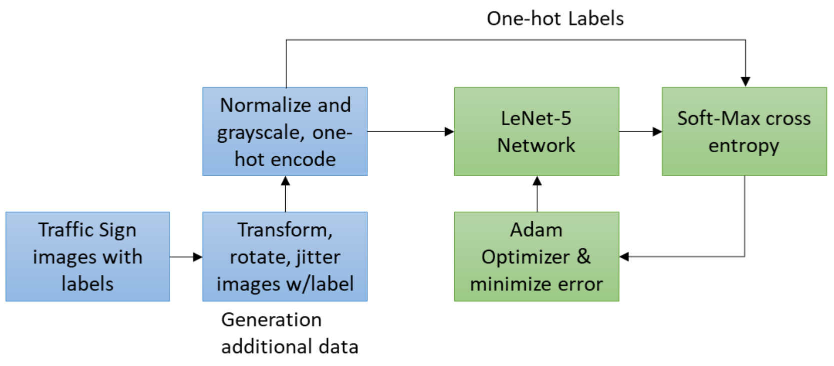

One-hot Labels
Normalize and grayscale, one-hot encode
LeNet-5 Network
Soft-Max cross entropy
Traffic Sign images with labels
Transform, rotate, jitter images w/label
Adam Optimizer & minimize error
Generation additional data

Fig. 43. Training network for traffic sign recognition.

Finally, the pipeline for vehicle detection is presented. Here, example images of vehicles and non-vehicles typically found in a front view are used to train the SVM. First, the images features are extracted by colour conversion, spatial binning, obtaining the colour histogram, extracting the HOG features, and finally scaling all the features. This is fed to a SVM fitter to train the SVM using the vehicle vs. non-vehicle labels. The method pipeline is shown in figure 44.

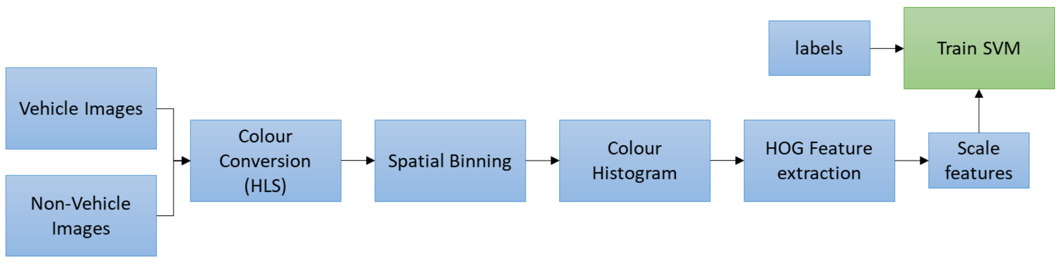


Fig. 44. Training SVM for vehicle detection.

6.2 Hardware

Though CNNs and neural networks in general were developed decades earlier, hardware with sufficient computing power did not exist that could utilize the full potential of these deep learning techniques. Advances in CPU and GPU devices has helped create an industry for relatively low-cost, low-power and small footprint high-performance computing devices. Companies such as NVIDIA and ARM have developed high performance processors and graphics-processing units very suitable for machine learning applications. Due to the parallel computing architecture of GPUs, they have become very useful for application in both neural network training and inferencing.

One product family from NVIDIA, the Drive PX / PX2, are designed to be used in self-driving cars and are being deployed by companies such as Tesla, Audi, Volkswagen and Mercedes-Benz. This is system has 2 GPUs and 2 Quad-core ARM-based processors. The Drive PX2 is shown in figure 45.

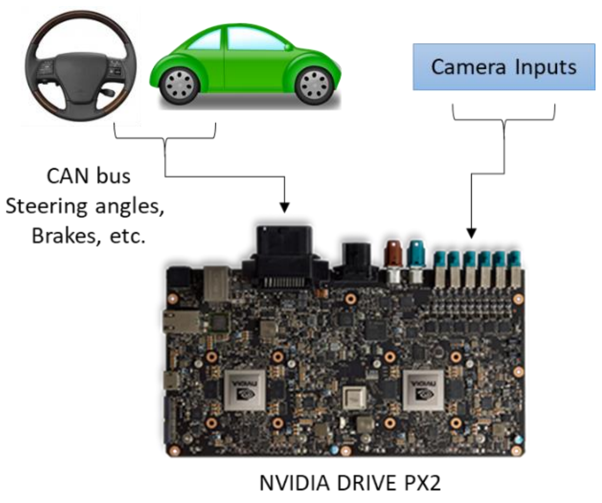


Fig. 45. Nvidia Drive PX2 self-driving car computer.

The Drive PX2 computer plugs directly into the vehicles Controller Area Network (CAN) bus. This is the network that connects all the vehicle’s control functions together. The Drive PX2 also interfaces with several cameras from right, center, left, and even rear views. In a tier-3 or tier-4 autonomous vehicle, 2 or 3 Drive PX computers may be used to facilitate all the various functions of a fully autonomous vehicle with little to no input from the driver.

6.3 Conclusions & Deployment

This report presented many techniques and methods in computer vision and deep learning to create a vision pipeline for self-driving cars. To this end we utilized many approached in computer vision and image processing afforded by industry and academia proven libraries such as OpenCV and scikit-image. These afforded an ability to rapidly extract features from images, transform images, and manipulate images to be used for classification and detection. For many deep learning methods, proven libraries such as TensorFlow and scikit-learn were presented and used. Important deep CNNs such as AlexNet and LeNet were explored and used, as well as SVMs. These techniques proved powerful in traffic sign recognition and vehicle detection. These techniques could be easily applied to other areas such as pedestrian detection.

The deep vision pipeline developed in this report can be used as part of the design in an autonomous vehicle, along with control, localization, and path planning functions required for a fully autonomous vehicle. The source code of all the pipelines for lane detection, traffic sign recognition, and vehicle detection are provided throughout the report, as well as in Appendix A. Much of these are easily deployed on a Linux based computer, preferably with a GPU+CPU processing capability to operate at or near real-time preferably at 30 frames per second.

6.4 Future Work

A future expansion of this work would include pedestrian detection pipeline built similarly to the vehicle detection pipeline to classify pedestrian versus non-pedestrian objects in the view. This kind of information could be used to teach a CNN to apply brakes when a pedestrian crosses a path, or avoid pedestrians or cyclists on the side of the road.

Furthermore, the vehicle detection method could be changed to identify the type of vehicle such as motorcycle, car, van, truck, bus, street-car, etc. This information is more useful in real-world driving scenarios and safety. Such a system would expand on the SVM design to send all positive detections for classification to a CNN. The classifier would be implemented similarly to the traffic sign recognition network.

Many other improvements may also be made to increase compute performance and robustness for large variance in real-world scenarios the systems would be exposed to.

Appendix A

A.1

Run training session on the network, and print validation and test accuracy from section 3.3.4

```
# tf Graph input
x = tf.placeholder(tf.float32, [None, 28, 28, 1])
y = tf.placeholder(tf.float32, [None, n_classes])
keep_prob = tf.placeholder(tf.float32)

# Model
logits = conv_net(x, weights, biases, keep_prob)

# Define loss and optimizer
cost = tf.reduce_mean(\
    tf.nn.softmax_cross_entropy_with_logits(logits=logits, labels=y))
optimizer =
tf.train.GradientDescentOptimizer(learning_rate=learning_rate)\
    .minimize(cost)

# Accuracy
correct_pred = tf.equal(tf.argmax(logits, 1), tf.argmax(y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))

# Initializing the variables
init = tf. global_variables_initializer()

# Launch the graph
with tf.Session() as sess:
    sess.run(init)

    for epoch in range(epochs):
        for batch in range(mnist.train.num_examples//batch_size):
            batch_x, batch_y = mnist.train.next_batch(batch_size)
            sess.run(optimizer, feed_dict={
                x: batch_x,
                y: batch_y,
                keep_prob: dropout})

            # Calculate batch loss and accuracy
            loss = sess.run(cost, feed_dict={
                x: batch_x,
                y: batch_y,
                keep_prob: 1.})
            valid_acc = sess.run(accuracy, feed_dict={
                x: mnist.validation.images[:test_valid_size],
                y: mnist.validation.labels[:test_valid_size],
                keep_prob: 1.})

            print('Epoch {:>2}, Batch {:>3} -'
                  'Loss: {:>10.4f} Validation Accuracy: {:.6f}'.format(
                epoch + 1,
                batch + 1,
                loss,
                valid_acc))

    # Calculate Test Accuracy
    test_acc = sess.run(accuracy, feed_dict={
        x: mnist.test.images[:test_valid_size],
        y: mnist.test.labels[:test_valid_size],
        keep_prob: 1.})
    print('Testing Accuracy: {}'.format(test_acc))
```

A.2

LeNet architecture from section 4.1

```
from tensorflow.contrib.layers import flatten
def LeNet(x):
     # Hyperparameters
     mu = 0
     sigma = 0.1
      # SOLUTION: Layer 1: Convolutional. Input = 32x32x1. Output = 
28x28x6.
      conv1_W = tf.Variable(tf.truncated_normal(shape=(5, 5, 3, 6), mean = 
mu, stddev = sigma))
      conv1_b = tf.Variable(tf.zeros(6))
      conv1   = tf.nn.conv2d(x, conv1_W, strides=[1, 1, 1, 1], 
padding='VALID') + conv1_b


      # SOLUTION: Activation.
      conv1 = tf.nn.relu(conv1)


      # SOLUTION: Pooling. Input = 28x28x6. Output = 14x14x6.
      conv1 = tf.nn.max_pool(conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 
1], padding='VALID')


      # SOLUTION: Layer 2: Convolutional. Output = 10x10x16.
      conv2_W = tf.Variable(tf.truncated_normal(shape=(5, 5, 6, 16), mean 
= mu, stddev = sigma))
      conv2_b = tf.Variable(tf.zeros(16))
      conv2   = tf.nn.conv2d(conv1, conv2_W, strides=[1, 1, 1, 1], 
padding='VALID') + conv2_b


      # SOLUTION: Activation.
      conv2 = tf.nn.relu(conv2)


      # SOLUTION: Pooling. Input = 10x10x16. Output = 5x5x16.
      conv2 = tf.nn.max_pool(conv2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 
1], padding='VALID')
```

```
      # SOLUTION: Flatten. Input = 5x5x16. Output = 400.
      fc0   = flatten(conv2)

      # SOLUTION: Layer 3: Fully Connected. Input = 400. Output = 120.
      fc1_W = tf.Variable(tf.truncated_normal(shape=(400, 120), mean = mu,
stddev = sigma))
      fc1_b = tf.Variable(tf.zeros(120))
      fc1   = tf.matmul(fc0, fc1_W) + fc1_b

      # SOLUTION: Activation.
      fc1    = tf.nn.relu(fc1)

      # SOLUTION: Layer 4: Fully Connected. Input = 120. Output = 84.
      fc2_W  = tf.Variable(tf.truncated_normal(shape=(120, 84), mean = mu,
stddev = sigma))
      fc2_b  = tf.Variable(tf.zeros(84))
      fc2    = tf.matmul(fc1, fc2_W) + fc2_b

      # SOLUTION: Activation.
      fc2    = tf.nn.relu(fc2)

      # SOLUTION: Layer 5: Fully Connected. Input = 84. Output = 10.
      fc3_W  = tf.Variable(tf.truncated_normal(shape=(84, 43), mean = mu,
stddev = sigma))
      fc3_b  = tf.Variable(tf.zeros(43))
      logits = tf.matmul(fc2, fc3_W) + fc3_b

    return logits
```

A.3

Feature extraction function referred from Section 5.5

```
from sklearn.preprocessing import StandardScaler

# Define a function to extract features from a list of images
def extract_features(imgs, cspace='RGB', spatial_size=(32, 32),
                        hist_bins=32, hist_range=(0, 256),
                        orient=9, pix_per_cell=8, cell_per_block=2,
hog_channel=0,
                        spatial_feat=True, hist_feat=True, hog_feat=True):
```

```
    # Create a list to append feature vectors to
    features = []
    # Iterate through the list of images
    for file in imgs:
        combined_features = []
        # Read in each one by one
        image = mpimg.imread(file, format='PNG')
        # apply color conversion if other than 'RGB'
        if cspace != 'RGB':
            if cspace == 'HSV':
                feature_image = cv2.cvtColor(image, cv2.COLOR_RGB2HSV)
            elif cspace == 'LUV':
                feature_image = cv2.cvtColor(image, cv2.COLOR_RGB2LUV)
            elif cspace == 'HLS':
                feature_image = cv2.cvtColor(image, cv2.COLOR_RGB2HLS)
            elif cspace == 'YUV':
                feature_image = cv2.cvtColor(image, cv2.COLOR_RGB2YUV)
            elif cspace == 'YCrCb':
                feature_image = cv2.cvtColor(image, cv2.COLOR_RGB2YCrCb)
        else:
            feature_image = np.copy(image)

        if spatial_feat:
            # Apply bin_spatial() to get spatial color features
            spatial_features = bin_spatial(feature_image, size=spatial_size)
            combined_features.append(spatial_features)
        if hist_feat:
            # Apply color_hist() also with a color space option now
            _,_,_,_,hist_features = color_hist(feature_image, nbins=hist_bins,
            bins_range=hist_range)
            combined_features.append(hist_features)
        if hog_feat:
            # Call get_hog_features() with vis=False, feature_vec=True
            if hog_channel == 'ALL':
                hog_features = []
                for channel in range(feature_image.shape[2]):
                    hog_features.append(get_hog_features(feature_image[:,:,channel],
                                                         orient, pix_per_cell,
                                                         cell_per_block,
                                                         vis=False, feature_vec=True))
                hog_features = np.ravel(hog_features)
            else:
                hog_features = get_hog_features(feature_image[:,:,hog_channel],
                                                orient, pix_per_cell, cell_per_block,
                                                vis=False, feature_vec=True)
            combined_features.append(hog_features)

        # Append the new feature vector to the features list
        features.append(np.concatenate(combined_features))
    # Return list of feature vectors
    return features

car_features = extract_features(cars)
notcar_features = extract_features(notcars)

if len(car_features) > 0:
    # Create an array stack of feature vectors
    X = np.vstack((car_features, notcar_features)).astype(np.float64)
    # Fit a per-column scaler
    X_scaler = StandardScaler().fit(X)
    # Apply the scaler to X
    scaled_X = X_scaler.transform(X)
```

A.4

Feature extraction, SVN training, and SVN test prediction from section 5.5

```
from sklearn.model_selection import train_test_split
from sklearn.svm import LinearSVC

# parameters
color_space = 'YCrCb' # Can be RGB, HSV, LUV, HLS, YUV, YCrCb
orient = 9
pix_per_cell = 8
cell_per_block = 2
hog_channel = "ALL" # Can be 0, 1, 2, or "ALL"
spatial_feat = True
spatial_size = (32, 32)
hist_feat = True
hist_bins = 32
hist_range = (0, 256)

car_features = extract_features(cars, cspace=color_space, spatial_size=spatial_size,
                                hist_bins=hist_bins, hist_range=hist_range,
                                orient=orient, pix_per_cell=pix_per_cell,
                                cell_per_block=cell_per_block, hog_channel=hog_channel,
                                spatial_feat=spatial_feat, hist_feat=hist_feat)
notcar_features = extract_features(notcars, cspace=color_space, spatial_size=spatial_size,
                                   hist_bins=hist_bins, hist_range=hist_range,
                                   orient=orient, pix_per_cell=pix_per_cell,
                                   cell_per_block=cell_per_block, hog_channel=hog_channel,
                                   spatial_feat=spatial_feat, hist_feat=hist_feat)



# array of feature vectors
X = np.vstack((car_features, notcar_features)).astype(np.float64)
# Fit a per-column scaler
X_scaler = StandardScaler().fit(X)
# Apply the scaler to X
scaled_X = X_scaler.transform(X)

# Define labels vector
y = np.hstack((np.ones(len(car_features)),
np.zeros(len(notcar_features))))

# Split data into randomized training and test sets
rand_state = np.random.randint(0, 100)
```

```
X_train, X_test, y_train, y_test = train_test_split(
    scaled_X, y, test_size=0.2, random_state=rand_state)

# Use a linear SVC
svc = LinearSVC()
svc.fit(X_train, y_train)
# Check the score of the SVC
print('Test Accuracy of SVC = ', round(svc.score(X_test, y_test), 4))
n_predict = 10
print('SVC prediction: ', svc.predict(X_test[0:n_predict]))
```

A.5

Sliding window functions from section 5.6

```
def slide_window(img, x_start_stop=[None, None], y_start_stop=[None, 
None],
                    xy_window=(64, 64), xy_overlap=(0.5, 0.5)):
    if x_start_stop[0] == None:
        x_start_stop[0] = 0
    if x_start_stop[1] == None:
        x_start_stop[1] = img.shape[1]
    if y_start_stop[0] == None:
        y_start_stop[0] = 0
    if y_start_stop[1] == None:
        y_start_stop[1] = img.shape[0]
    # Span of the region to be searched
    xspan = x_start_stop[1] - x_start_stop[0]
    yspan = y_start_stop[1] - y_start_stop[0]
    # Number of pixels per step in x, y
    nx_pix_per_step = np.int(xy_window[0]*(1 - xy_overlap[0]))
    ny_pix_per_step = np.int(xy_window[1]*(1 - xy_overlap[1]))
    # Compute the number of windows in x, y
    nx_buffer = np.int(xy_window[0]*(xy_overlap[0]))
    ny_buffer = np.int(xy_window[1]*(xy_overlap[1]))
    nx_windows = np.int((xspan-nx_buffer)/nx_pix_per_step)
    ny_windows = np.int((yspan-ny_buffer)/ny_pix_per_step)
    # Initialize list to append window positions
    window_list = []
    # Loop through finding x, y window positions
    for ys in range(ny_windows):
        for xs in range(nx_windows):
            # Calculate window position
            startx = xs*nx_pix_per_step + x_start_stop[0]
            endx = startx + xy_window[0]
            starty = ys*ny_pix_per_step + y_start_stop[0]
            endy = starty + xy_window[1]

            # Append window position to list
            window_list.append(((startx, starty), (endx, endy)))
    # Return list of windows
    return window_list
```

Bibliography

[1] LeCun, Y., Bottou, L., Bengio, Y. and Haffner, P. (1998). Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11), pp.2278-2324.

[2] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." In Advances in neural information processing systems, pp. 1097-1105. 2012.

[3] Simonyan, K. and Zisserman, A. (2015). VERY DEEP CONVOLUTIONAL NETWORKS FOR LARGE-SCALE IMAGE RECOGNITION. [online] pp.1-14. Available at: https://arxiv.org/pdf/1409.1556.pdf [Accessed 5 Aug. 2017].

[4] Mu Guo, Zhang Xinyu, Deyi Li, Tianlei Zhang, An Lifeng, "Traffic light detection and recognition for autonomous vehicles", The Journal of China Universities of Posts and Telecommunications, vol. 22, no. 1, pp. 50-56, 2015.

[5] Docs.opencv.org. (2017). Miscellaneous Image Transformations — OpenCV 2.4.13.3 documentation. [online] Available at: http://docs.opencv.org/2.4/modules/imgproc/doc/miscellaneous_transformations.html [Accessed 5 Aug. 2017].

[6] Canny, J. (1986). A Computational Approach to Edge Detection. IEEE Transactions on Pattern Analysis and Machine Intelligence, PAMI-8(6), pp.679-698.

[7] Docs.opencv.org. (2017). Canny Edge Detection — OpenCV 3.0.0-dev documentation. [online] Available at: http://docs.opencv.org/3.0-beta/doc/py_tutorials/py_imgproc/py_canny/py_canny.html [Accessed 5 Aug. 2017].

[8] Docs.opencv.org. (2017). Image Gradients — OpenCV 3.0.0-dev documentation. [online] Available at: http://docs.opencv.org/3.0-beta/doc/py_tutorials/py_imgproc/py_gradients/py_gradients.html [Accessed 5 Aug. 2017].

[9] Docs.opencv.org. (2017). Camera Calibration and 3D Reconstruction — OpenCV 2.4.13.3 documentation. [online] Available at: http://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html [Accessed 5 Aug. 2017].

[10] Duong, T., Pham, C., Tran, T., Nguyen, T. and Jeon, J. (2016). Near real-time ego-lane detection in highway and urban streets. 2016 IEEE International Conference on Consumer Electronics-Asia (ICCE-Asia).

[11] Seo, D. and Jo, K. (2014). Inverse Perspective Mapping based road curvature estimation. 2014 IEEE/SICE International Symposium on System Integration.

[12] Docs.opencv.org. (2017). Geometric Image Transformations — OpenCV 2.4.13.3 documentation. [online] Available at: http://docs.opencv.org/2.4/modules/imgproc/doc/geometric_transformations.html

[13] Bourne, M. (2017). 8. Radius of Curvature. [online] Intmath.com. Available at: http://www.intmath.com/applications-differentiation/8-radius-curvature.php [Accessed 5 Aug. 2017].

[14] Hoon Chung, Sung Joo Lee and Jeon Gue Park (2016). Deep neural network using trainable activation functions. 2016 International Joint Conference on Neural Networks (IJCNN).

[15] CS168: The Modern Algorithmic Toolbox. (2015). [ebook] Stanford, p.Lecture #15. Available at: http://theory.stanford.edu/~tim/s15/l/l15.pdf [Accessed 5 Aug. 2017].

[16] Sebastian Raschka's Website. (2017). Machine Learning FAQ. [online] Available at: https://sebastianraschka.com/faq/docs/closed-form-vs-gd.html [Accessed 5 Aug. 2017].

[17] Coursera (2017). Lecture 10 - Gradient Descent. [video] Available at: https://www.coursera.org/learn/machine-learning/lecture/8SpIM/gradient-descent [Accessed 5 Aug. 2017].

[18] Rojas, R. (1996). Neural Networks - A Systematic Introduction. [ebook] New York: Springer-Verlag, pp.153-183. Available at: https://page.mi.fu-berlin.de/rojas/neural/chapter/K7.pdf [Accessed 5 Aug. 2017].

[19] Dahl, G., Sainath, T. and Hinton, G. (2013). Improving deep neural networks for LVCSR using rectified linear units and dropout. 2013 IEEE International Conference on Acoustics, Speech and Signal Processing.

[20] Hara, K., Saito, D. and Shouno, H. (2015). Analysis of function of rectified linear unit used in deep learning. 2015 International Joint Conference on Neural Networks (IJCNN).

[21] TensorFlow. (2017). tf.nn.relu | TensorFlow. [online] Available at: https://www.tensorflow.org/api_docs/python/tf/nn/relu [Accessed 5 Aug. 2017].

[22] Olah, C. (2014). Visualizing MNIST: An Exploration of Dimensionality Reduction. [Blog] colah's blog. Available at: http://colah.github.io/posts/2014-10-Visualizing-MNIST/ [Accessed 5 Aug. 2017].

[23] LeCun, Y. (2017). MNIST handwritten digit database, Yann LeCun, Corinna Cortes and Chris Burges. [online] Yann.lecun.com. Available at: http://yann.lecun.com/exdb/mnist/ [Accessed 5 Aug. 2017].

[24] Srivastava, Nitish, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. "Dropout: a simple way to prevent neural networks from overfitting." Journal of Machine Learning Research 15, no. 1 (2014): 1929-1958.

[25] "Convolutional Neural Network - MATLAB & Simulink". 2017. Mathworks.Com. https://www.mathworks.com/discovery/convolutional-neural-network.html.

[26] Graham, Benjamin. "Fractional max-pooling." arXiv preprint arXiv:1412.6071(2014).

[27] Lin, Min, Qiang Chen, and Shuicheng Yan. "Network in network." arXiv preprint arXiv:1312.4400 (2013).

[28] GitHub. (2017). aymericdamien/TensorFlow-Examples. [online] Available at: https://github.com/aymericdamien/TensorFlow-Examples [Accessed 6 Aug. 2017].

[29] Kingma, Diederik, and Jimmy Ba. "Adam: A method for stochastic optimization." arXiv preprint arXiv:1412.6980 (2014).

[30] Section 4.5 (Seperating Hyperplanes) of the book: Elements of Statistical Learning by T. Hastie, R. Tibshirani and J. H. Friedman.

[31] Docs.opencv.org. (2017). Introduction to Support Vector Machines — OpenCV 2.4.13.3 documentation. [online] Available at: http://docs.opencv.org/2.4/doc/tutorials/ml/introduction_to_svm/introduction_to_svm.html [Accessed 6 Aug. 2017].

[32] Scikit-learn.org. (2017). 1.4. Support Vector Machines — scikit-learn 0.18.2 documentation. [online] Available at: http://scikit-learn.org/stable/modules/svm.html [Accessed 6 Aug. 2017].

[33] Dalal, Navneet, and Bill Triggs. "Histograms of oriented gradients for human detection." Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on. Vol. 1. IEEE, 2005.

[34] Scikit-learn.org. (2017). 4.3. Preprocessing data — scikit-learn 0.18.2 documentation. [online] Available at: http://scikit-learn.org/stable/modules/preprocessing.html [Accessed 6 Aug. 2017].

[35] Noh, SeungJong, Daeyoung Shim, and Moongu Jeon. "Adaptive sliding-window strategy for vehicle detection in highway environments." IEEE Transactions on Intelligent Transportation Systems 17.2 (2016): 323-335.

[36] Bojarski, Mariusz, et al. "End to end learning for self-driving cars." *arXiv preprint arXiv:1604.07316* (2016).