AIN SHAMS UNIVERSITY

FACULTY OF ENGINEERING

Advanced Driver Assistance System

A Graduation Project is Submitted in Partial Fulfillment of the Requirements for the bachelor's Degree

July 2018

A Thesis

Submitted by

Abdelrahman Emad Eldeen Ahmed Ibrahim

Ahmed Mohamed Ahmed Madbouly

Eman Gamal Mohamed Kenawy

Randa Mohamed Khairy Mostafa

Robica Assaad Mikhail

Supervised By

Prof. Hossam Hassan **Prof. Sherif Hammad**

Acknowledgments

We, the ADAS Team, would like to express our profound gratitude to everyone who had helped us turning our ideas into reality. The encouragement and support we received were our motive to push forward and give our best.

Special thanks to:
Dr. Hossam Hassan for his academic guidance in the Image processing, and for the continuous follow-ups and positive feedback he always gave us.

Prof. Sherif Hammad for his academic supervision, professional tips, and most importantly the positive feedback and motivation he always showed throughout the year.

Dr.Maged Ghoneima and **IHub** for the professional coordination of IGP, offering us all the support needed.

Table of Contents

Table of Figures

Table of Tables

Table of Codes

Introduction

Advanced driver-assistance system, or ADAS, is a system to help the driver in the driving process. When designed with a safe human-machine interface, they should increase car safety and more generally road safety.

ADAS has received considerable attention in recent decades because many car accidents are caused mainly by drivers' lack of awareness or fatigue. Warning the driver of any dangers that may lie ahead on the road is important for improving traffic safety and accident prevention.

ADAS enable better situational awareness and control to make driving easier and safer. ADAS technology can be based upon systems local to the car, that is, "vehicle resident systems" like vision/camera systems, sensor technology, or can be based on smart, interconnected networks as in the case of vehicle-to-vehicle (V2V), or vehicle-to-infrastructure (V2I) systems (jointly known as V2X systems).

One of the most recent trends in automotive industry is the emergence of electric autonomous vehicles. In North America, especially in Canada, development and production of autonomous vehicles is growing at a rapid pace. These vehicles are integrated with ADAS system to enhance vehicle safety and reduce road accidents. From manufacturer's point of view, the new trend of autonomous driving coupled with vehicle safety systems are set to spearhead the adoption rate of advanced driver assistance system, which in turn is expected to push the demand for ADAS testing equipment across the globe.

Modern cars contain a lot of sensors, which are located on many positions on the outside surface of the car. These sensors can be thought of as the eyes of the car to get a continuous picture of the outer world. The graphic below illustrates where these sensors are, and which areas they are supervising.

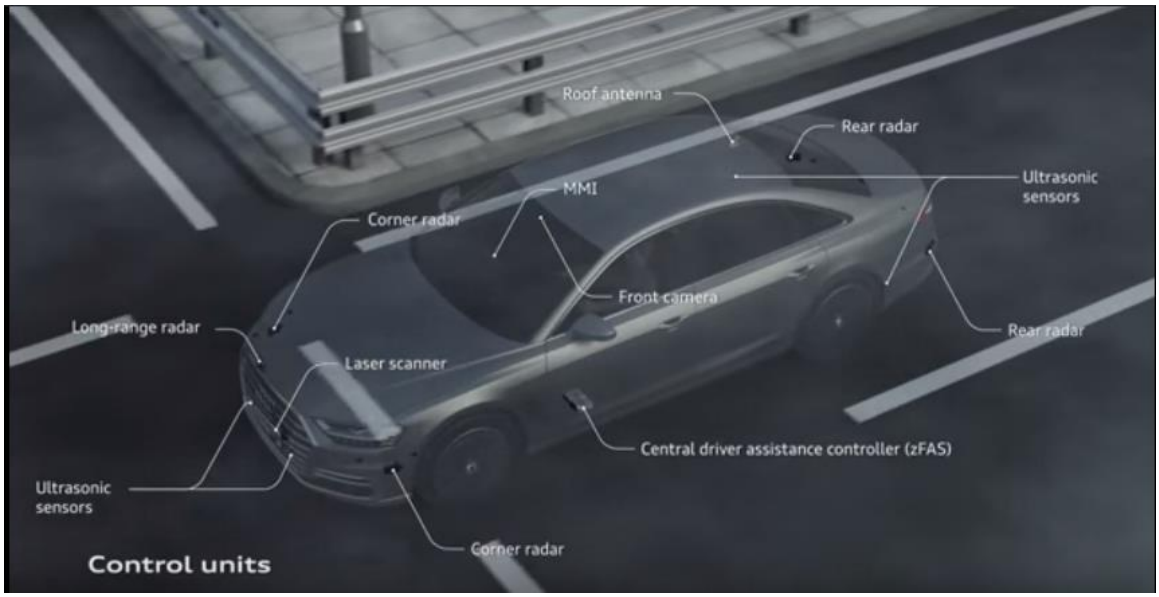


Figure 1: Sensors in a car

Typically, those sensors work in different physical dimensions like ultrasonic sound for the near field used for parking aid beepers, infrared light and cameras for the mid-range to control wipers during rain or detect pedestrians. Radar is used for long distance control to stop the car for obstacles, such as a pedestrian jump in your way or for keeping a safe distance with respect to the car in front. Sensors have some local intelligence to pre-process their detection tasks as long wiring cables limit a full bandwidth transport for information to a centralized processing unit in many cases.

- **Some of ADAS Functionalities:**
  Adaptive cruise control (ACC)
  Adaptive light control: swiveling curve lights
  Anti-lock braking system
  Automatic parking
  Automotive night vision
  Blind spot monitor
  Collision avoidance system
  Forward collision warning system (FCWS)
  Intelligent speed adaptation or intelligent speed advice (ISA)
  Lane departure warning system (LDWS)
  Lane change assistance
  Parking sensor
  Pedestrian protection system
  Traffic-sign recognition
  Vehicular communication systems
  Wrong-way driving warning

CHAPTER 1

Nvidia Development Kit

1. Introduction

Autonomous machines do more than simply complete tasks. They learn, evolve, and react to the world around them with the power of AI. NVIDIA Jetson is the world's leading platform for high-performance, energy-efficient AI computing. Robots, drones, and intelligent cameras can now hamess the power of deep learning with Jetson TX2, the embedded supercomputer design for edge devices. With AI at the edge, you count on fast, accurate inference in autonomous machines without worrying about network constraints.

1.1. Jetson TX2 Developer Kit

Figure 1.1: Nvidia kit

Jetson TX2 is the fastest, most power-efficient embedded AI computing device. The latest addition to the industry-leading Jetson embedded platform, this 7.5-watt supercomputer on a module brings true AI computing at the edge. It's built around an NVIDIA Pascal™-family GPU and loaded with 8 GB of memory and 59.7 GB/s of memory bandwidth. It includes the latest technology for deep learning, computer vision, GPU computing, and graphics—making it ideal for embedded AI computing.

It packs this performance into a small, power-efficient form factor that is ideal for intelligent edge devices like robots, drones, smart cameras, and portable medical devices. It supports all the features of the Jetson TX1 while enabling bigger, more complex deep neural networks.

1.2 Nvidia Jetson TX2 specifications

- **GPU:**
  The biggest change from the TX1 is that the TX2's GPU is based on Nvidia's Pascal architecture. This architecture uses a new manufacturing process that makes it possible to fit in more transistors. The new Pascal architecture involves the 14nm FinFET node process. At the risk of over-simplifying the complex structure, fin field-effect transistors (FinFETs), due to their structure, are essentially better at current control. This property allows manufacturers to pack more processing capabilities in a smaller form factor while simultaneously reducing power consumption.

- **CPU and RAM:**
  The CPU comprises a quad core A57 with a dual-core Denver 2.0 for high performance single threaded functions. The TX1 had an A53 along with the A57, but the A53 was unusable.
  The RAM has been doubled by using four 32-bit channels as opposed to the four 16-bit channels of the TX1. Thus, the 8GB low-power DDR4 memory also brings with it a higher bandwidth making the TX2 one of the fastest development boards out there.

- **Display and Camera Interface:**
  There are quite a few display options that users can choose. The two DSI ports for high-speed serial connection between a display module and the processor have been carried forward from the TX1. The HDMI 2.0 port and Embedded DisplayPort (eDP) have also been continued. The only difference is the addition of a v1.2 DisplayPort for external display.

Figure 1.2: Nvidia camera

The Camera Serial Interface has 12 lanes that allow a maximum of six cameras to be connected. It also uses a new version of the physical layer for data transfer, the MIDI D-PHY v1.2, an industrial standard to connect the cameras to the SoC. This version of the D-PHY bumps up the

transfer rate per lane to 2.5Gbps. However, note that this is not the latest version of the standard. The latest version has a transfer rate of 4.5Gbps. Comparatively, the TX1 uses v1.1 of the standard and thus has a slower transfer rate of 1.1Gbps per lane.

- **Storage:**
  Onboard storage has been doubled with the TX2 now featuring a 32GB eMMC storage. The Embedded MultiMediaCard (eMMC) storage is a cheap yet robust storage solution. However, like an SD card, it is not as fast or equipped as a full-fledged SSD. But donot worry, there is a SATA interface available to expand the storage if you would like to.

- **Dimensions:**
  Regarding physical dimensions, the form factor of the TX1 has been continued in the newer iteration. The standalone TX2 module is roughly the size of a credit card (50mm x 87mm).

1.3. NVIDIA Jetson TX2 Performance

The TX2 has two working modes. The efficiency mode, as the name suggests, prioritizes power consumption over peak performance. This mode allows the TX2 to work at the maximum processing level of the TX1 while consuming just half the power (7.5W) of what the TX1 would have consumed. That is an incredible development!

The performance mode pushes the TX2 into raw, unmitigated power and of course, the power consumption takes a backseat. In this mode, the TX2 works at twice the processing level of the TX1.

1.4. Comparison Between TX1 and TX2

Table 1.1: TX1 & TX2 comparison

	Jetson TX2	Jetson TX1
GPU	NVIDIA Pascal, 256 CUDA cores	NVIDIA Maxwell, 256 CUDA cores
CPU	HMP Dual Denver 2/2 MB L2+ Quad ARM A57/2 MB L2	Quad ARM A57/2 MB L2
Video	4K x 2K 60 Hz Encode [HEVC] 4K x 2K 60 Hz Decode [12-Bit Support]	4K x 2K 30 Hz Encode [HEVC] 4K x 2K 60 Hz Decode [10-Bit Support]
Memory	8 GB 128-bit LPDDR4 58.3 GB/s	4 GB l4-bit LPDDR4 25.6 GB/s
Display	2x DSI, 2x DP 1.2 / HDMI 2.0 / eDP 1.4	2x DSI, 1x eDP 1.4 / HDMI / DP 1.2
CSI	Up to 6 Cameras (2 Lane) CSI2 D-PHY 1.2 (2.5 Gbps/Lane)	Up to 6 Cameras (2 Lane) CSI2 D-PHY 1.1 (1.5 Gbps/Lane)
PCIE	Gen 2 | 1x4 + 1x1 OR 2x1 + 1x2	Gen 2 | 1x4 + 1x1
Data Storage	32 GB eMMC, SDIO, SATA	16 GB eMMC, SDIO, SATA
Other	CAN, UART, SPI, I2C, I2S, GPIOs	UART, SPI, I2C, I2S, GPIOs
USB	USB 3.0 + USB 2.0	USB 3.0 + USB 2.0
Connectivity	1 Gigabit Ethernet, 802.11ac WLAN, Bluetooth	1 Gigabit Ethernet, 802.11ac WLAN, Bluetooth
Mechanical	50 mm x 87 mm (400-Pin Compatible Board-to-Board Connector)	50 mm x 87 mm (400-Pin Compatible Board-to-Board Connector)

2. Installation

2.1. System Requirements

In order to work on PC we need to install:

- Python version 3 or 2
- numpy
- OpenCV
- Tensorflow
- Keras

And to download this project on Nvidia jetson TX2 board install:

- JetPack version 3.2.1 in case of Ubuntu 16.04 host PC
  or any older version for Ubuntu 14.04
- numpy
- Python 3 or 2 (already installed on the board)
- OpenCV
- Tensorflow-gpu
- Keras

2.2. NVIDIA Installation Process

First, we must install some useful tools like pip, wget and curl that will be used later.

```
$ sudo apt-get install python-pip                #python2
OR
$ sudo apt-get install python3-pip   #python 3
$ sudo apt-get install wget
$ sudo apt-get install curl
```

- **JetPack:**

  In our project the host was ubuntu 16.04 then we used jetPack version 3.2.1.

  1. Download JetPack from https://developer.nvidia.com/embedded/jetpack and choose the appropriate version from here https://developer.nvidia.com/embedded/jetpack-archive
  2. Put the downloaded file in a folder named “JetPack”.
  3. Connect the host PC to a switch, the board to the same switch using Ethernet cable and the switch to the internet.
  4. Right click on the file, choose properties and check “Add exec permission” or using terminal write chmod +x JetPack-$JetPack-L4T-3.2.1-linux-x64_b23.run
  5. Open terminal and write ./JetPack-L4T-3.2.1-linux-x64_b23.run. to run JetPack.
  6. Indicate the installation directory and choose whether to enable data collection or not
  7. Select the device you want to flash jetpack on it. In our case, choose Jetson TX2.
  8. A pop-up window will appear asking for permission. Enter the sudo password which in our case is “nvidia”.
  9. A window appears where you can customize which components to download.
     NB: if you do not want to install libraries on the host you can custom your options and choose no action for host list of packages.
  10. Accept the license agreement for the selected components.
  11. The Component Manager will proceed with the installation. Once the host installation steps are completed, click the Next button to continue with the installation of target components.
  12. Select the network layout “Device accesses internet via router/switch”.
  13. Select the interface to use. To specify which interface, open a terminal and write ipconfig all interfaces will be shown, and you can decide the appropriate interface
  14. Another window appears which tells you the actions that will be performed click next.
  15. A pop-up window appears which instruct you to put the board in force recovery mode.
  16. After the post installation, jetpack is successfully flashed on the board.

- **OpenCv:**
  One way to install openCV on jetson is to build from source. A repository on github where there is a download script that can be used to build openCV
  [https://github.com/jetsonhacks/buildOpenCVTX2]
  The script contains the installation of all dependencies that openCV need to be installed. Also, in this script you can configure openCV for your needs
  1. Clone this repository on the board.
     $ git clone https://github.com/jetsonhacks/buildOpenCVTX2.git
  2. Open buildOpenCVTX2 directory.
     $ cd buildOpenCVTX2
  3. Run the script.
     $ ./buildOpenCV.sh
  4. Now you are ready to install openCV, open build directory.
     $ cd ~/opencv/build
  5. $ make
  6. Install openCV.
     $ sudo make install
  7. Search for ccmake file.
     $ apt-cache search ccmake
  8. $ sudo apt-get install cmake-curses-gui
  9. List all settings for opencv.
     $ ccmake .

- **Tensorflow-gpu:**
  You must make sure that NVIDIA drivers, cudaand cuDNN (already provided by the jetpack) are installed. Then
  1. Prepare TensorFlow dependencies and required packages.
     $ sudo apt-get install libcupti-dev
  2. Install TensorFlow (GPU-accelerated version).
     $ pip install tensorflow-gpu
     To choose a specific version then you can write
     $ pip install tensorflow-gpu=={version}
  3. Verify installation.
     Open new terminal and write
     $ python
     then write
     >>> import tensorflow as tf
     if the library is imported successfully then tensorflow is installed.

- **Numpy:**
  In case of python2 from terminal write:
  $ pip install numpy
  In case python3 is used write:
  $ pip3 install numpy

- **Keras:**
  In case of python2 from terminal write:
  $ pip install keras
  In case python3 is used write:
  $ pip3 install keras

2.3. Laptop Installations

- **Python**:
  From terminal write:
  sudo apt-get -y python{version}
  where version is either 2 or 3

- **Numpy:**
  In case of python2 from terminal write:
  $ pip install numpy
  In case python3 is used write:
  $ pip3 install numpy

- **OpenCV:**
  From https://www.lfd.uci.edu/~gohlke/pythonlibs/#opencv download the wheel file and you must choose the appropriate version of opencv corresponding to the python version you are using. In our project we mainly used python3.5.2 then we downloaded "opencv_python-3.4.1-cp35-cp35m-win_amd64.whl"

- **Tensorflow:**
  In case of python2 from terminal write:
  $ pip install tensorflow
  In case python3 is used write:
  $ pip3 install tensorflow

- **Keras:**
  In case of python2 from terminal write:
  $ pip install keras
  In case python3 is used write:
  $ pip3 install keras

CHAPTER 2

Object Detection

1. Introduction

1.1. Overview

Object detection is a main part of scene understanding where the target is to identify an object in the scene. There exist various models that can be used in this purpose as SSD (Single Shot Detection), YOLO (You Only Look Once) and other deep learning models. In our project we have used the YOLO model, which we will discuss in detail in a later section.

1.2. Deep Learning

Deep learning is a machine learning technique that teaches computers to do what comes naturally to humans.It is the key technology behind driverless cars,enabling them to recognize a stop sign,or a pedestrian.Also,it can achieve high accuracy,sometimes exceeding human level performance.

Deep learning consists of models that can learn how to perform a specific task from a set of labeled data,this process is known as Training.Models also consist of neural network architectures thatcontain many layers.

Many applications based on deep learning require high accuracy in terms of safety as self-driving cars. This was the main cause deep learning only became useful recently although it was first theorized in the 1980s.In addition, the need for a substantial computing power.

1.3. How Deep Learning Works?

Many deep learning models use neural network architectures, which is why deep learning models are often referred to as deep neural networks. Neural Networks are made up of neurons that have learnable weights and biases. Each neuron receives some inputs, performs a dot product between weights and inputs, then follows it with non-linearity. They have a loss function on the last fully connected layer to compute the error between the output expected from the network and the actual output coming from the network.

The term "deep" usually refers to the number of hidden layers in the neural network. Traditional neural networks only contain 2-3 hidden layers, while deep networks can have as many as 150.

As discussed in section 2.2 deep learning models are trained by using large sets of labeled data and neural network architectures that learn features directly from the data without the need for manual feature extraction.

1.4. Convolutional Neural Network (CNN)

Convolutional neural networks are very similar to neural networks the difference is that they assume that the inputs are images.A CNN convolves learned features with input data using 2D convolutional layers, making this architecture suitable to process 2D data.This way eliminates the need for manual feature extraction, therefore you donot need to identify features used to classify images.

The CNN extracts feature directly from images. CNN is composed of layers that filters the images to getuseful information. Theses filters' (= kernels) parameters are learned automatically during training.This automated feature extraction makes deep learning models highly accurate for computer vision tasks as object classification.

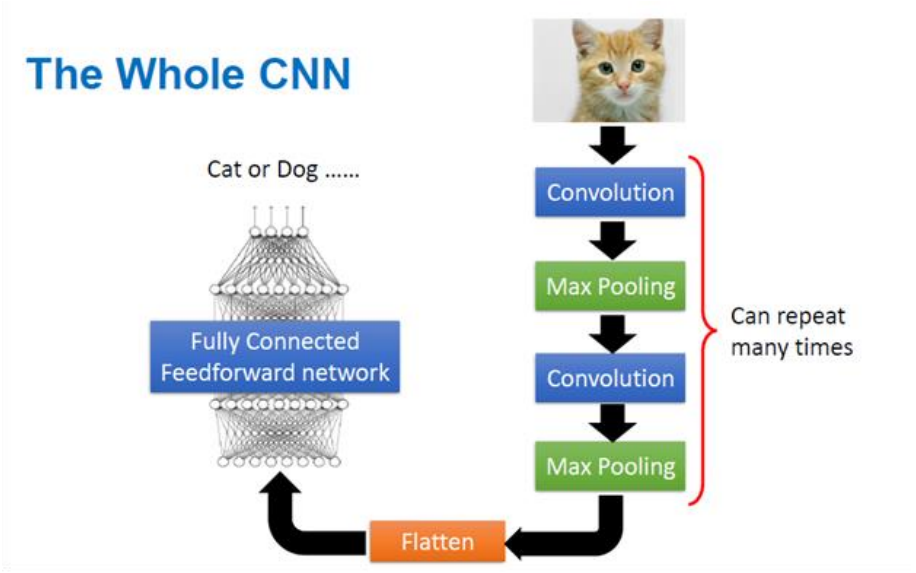


Figure 2.1: CNN Layers

1.5. CNN Common Architecture

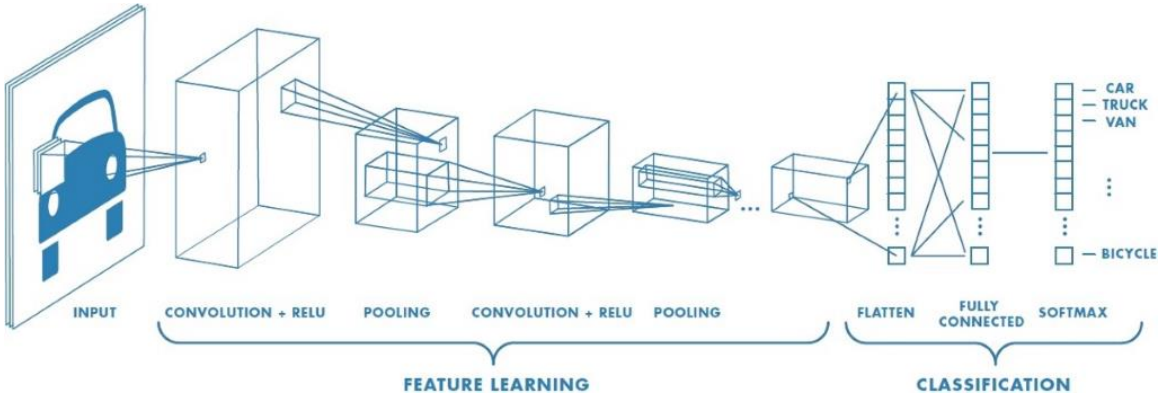


Figure 2.2: CNN Architecture

Normally the pattern is convolution: Input - (CONV) – Rectified Liner Model (ReLU) – Pool – CONV – ReLU – Pool – FC – Softmax loss

- **Convolution Layer (CONV layer)**
  Is the most important operation. The CONV layer's parameters consist of a set of learnable filters. During the forward pass, we slide more precisely convolve each filter across the width and height of the input image and compute dot products between the entries of the filter and the input at any position. The output is a 2-dimensional activation map that gives the responses of that filter at every spatial position. The network will learn filters that extract some type of visual feature such as an edge or color on the first layer, or an entire shape on the higher layers of the network.

- **The ReLu (Rectified Linear Unit) Layer**
  ReLu is the most commonly deployed activation function for the outputs of the CNN neurons.

- **The Pooling Layer**
  This layer main utility lies in reducing the spatial dimensions (Width * Height) of the input for the next convolution layer. It does not affect the depth dimension of the input. The decrease in size leads to less computational overhead for the next layers also it solves the overfitting problem.

- **The Fully Connected Layer (FC)**
  A layer where its neurons fully connected to the output of previous layer. It is used in the last stage of the CNN to connect to the output layer that compute the class score.

1.6. Some of the object detection deep learning models

- **R-CNN:**
  Based on the extraction of possible objects using a region proposal method, features are extracted from each region using CNN and finally classify regions with SVMs.

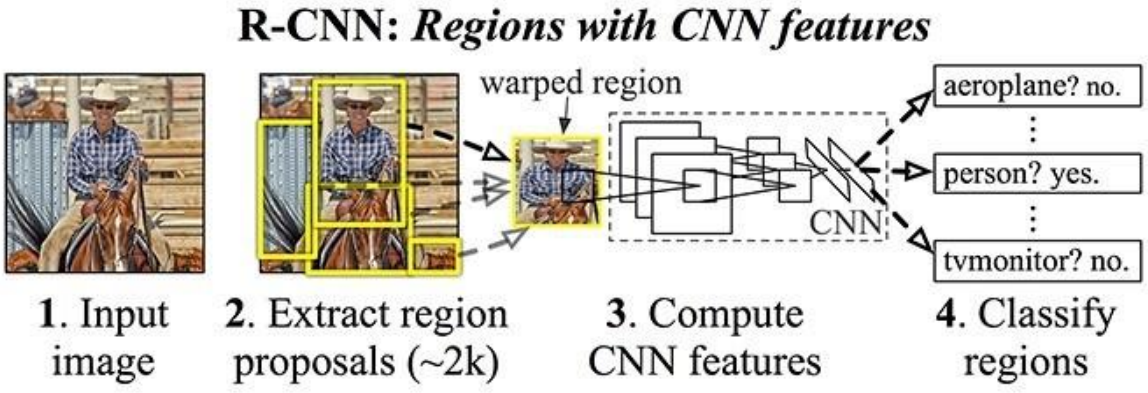


Figure 2.3: R-CNN Architecture

- **Fast R-CNN**
  Applied the CNN on the whole image the both Region of Interest (ROI) Pooling on the feature map with a final feedforward network for classification and regression are applied.

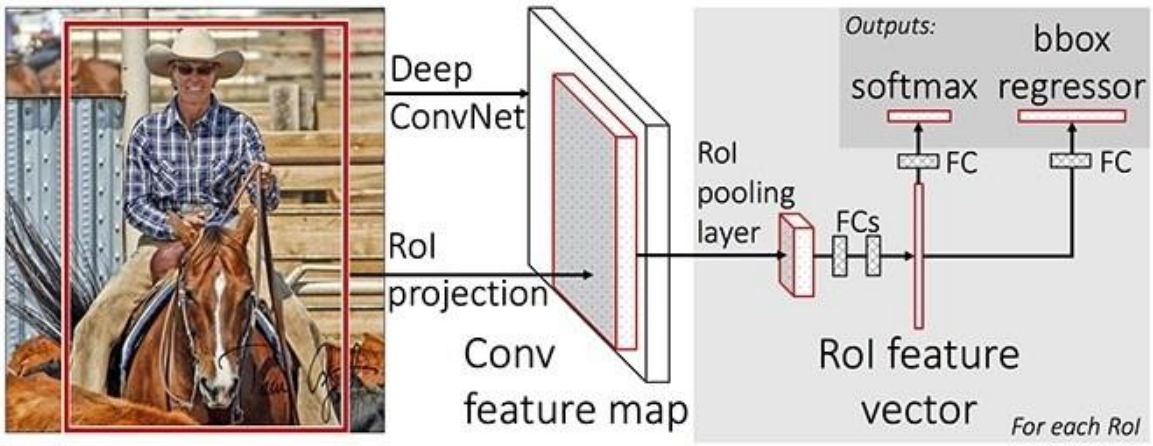


Figure 2.4: Fast-RCNN Architecture

- **YOLO**
  Proposed a simple convolutional neural network approach which has a great results and speed allowing for the first time real time object detection.

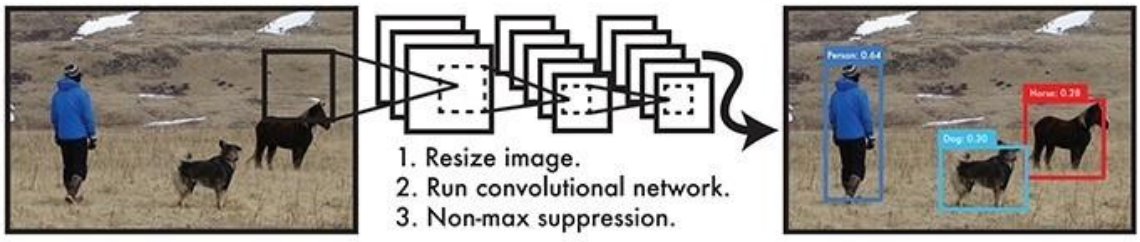


Figure 2.5: YOLO Architecture

- **Faster R-CNN**
  - A Region Proposal Network (RPN) is added to get rid of the selective search algorithm and make the model is completely trainable. The RPN task is to output the objects based on a score called objectness.

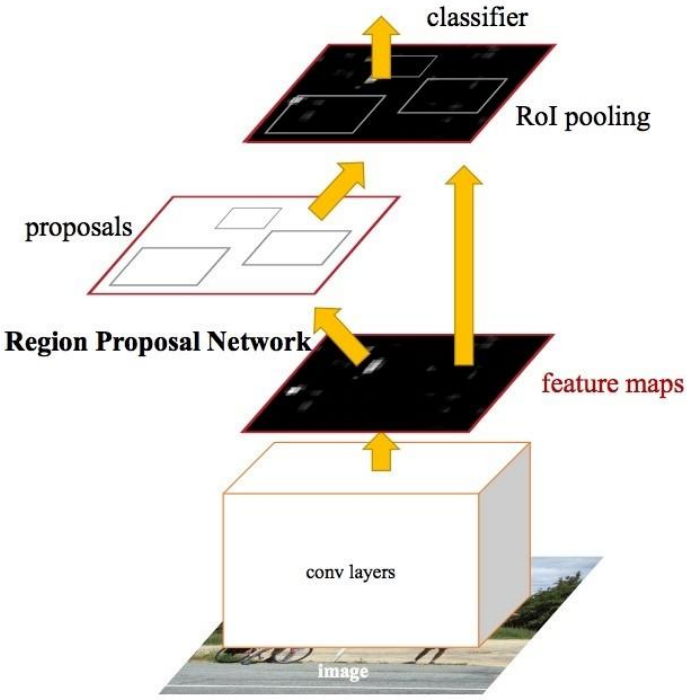


Figure 2.6: Faster-RCNN Architecture

- **Single Shot Detection (SSD):**
  Takes on YOLO but uses multiple sized convolutional feature maps.The main idea is that region proposals are no more needed instead it uses different bounding boxes and then adjust then adjust the bounding box a part of prediction.

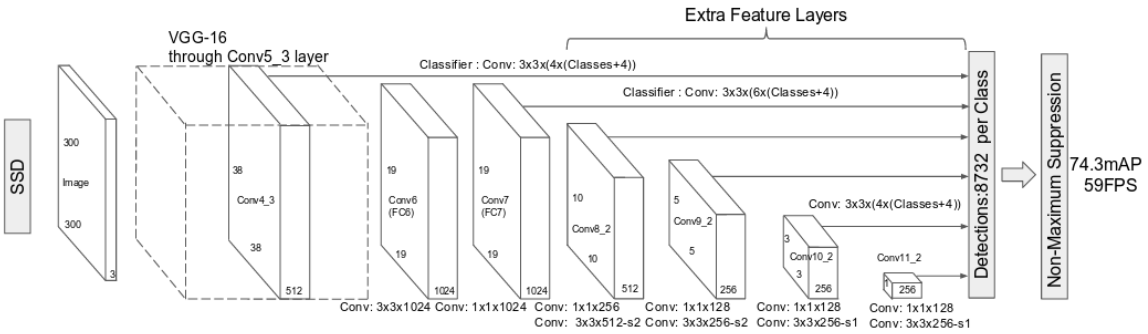


Figure 2.7: SSD Architecture

- **Region based Fully Convolutional Network (R-FCN)**
  Takes the architecture of Faster-RCNN but with only convolutional networks. Transforms the image to a feature map, a region proposal network uses the feature map to locate

boundingcandidates in the image, then a classifier uses this information to classify each bounding boxcandidate to a class.

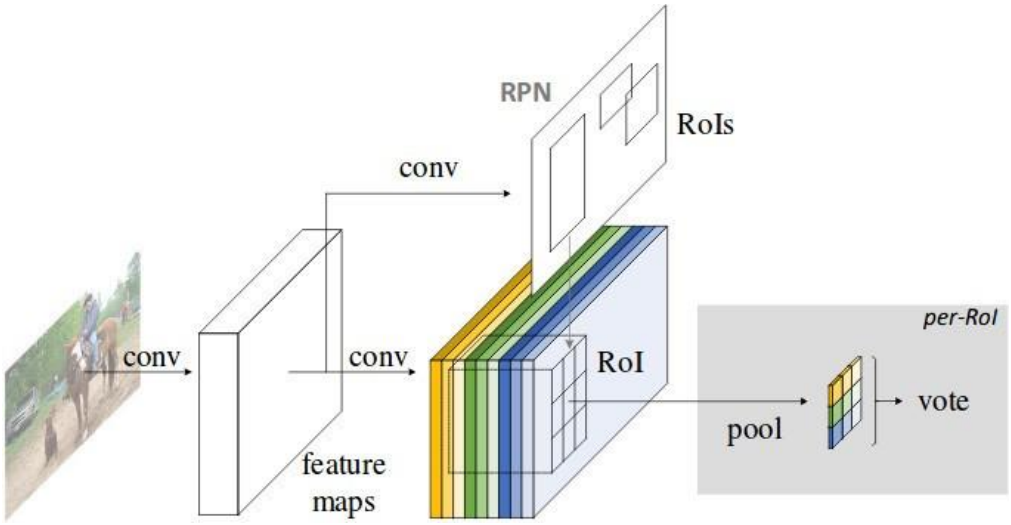


Figure 2.8: R-FCN Architecture

1.7. Speed and Accuracy Comparison

Faster RCNN has the highest accuracy but has a low speed. YOLO has the highest speed but has a lower accuracy than Faster RCNN and SSD. Therefore, the choice of the appropriate model depends on the project's requirements and the accuracy speed trade-off.

Figure 2.9: Models comparison

2. YOLO (You Only Look Once)

2.1. Overview

"You Only Look Once" is a new approach to object detection. As discussed in section 1.1.6 more recent approaches like R-CNN use region proposal methods to first generate potential bounding boxes in an image and then run a classifier on these proposed boxes. After classification, post-processing is used to refine the bounding box, eliminate duplicate detections, and rescore the box based on other objects in the scene. These complex pipelines are slow and hard to optimize because each individual component must be trained separately. On the other hand, Yolo reframe object detection as a single regression problem, straight from image pixels to bounding box coordinates and class probabilities. So, you only look once at an image to predict if there any objects, if so what objects are present and where they are.

2.2. How it works

YOLO uses a single convolutional neural network for prediction of multiple bounding boxes and class probabilities for those boxes.

YOLO detection network has 24 convolutional layers followed by 2 fully connected layers as shown below.

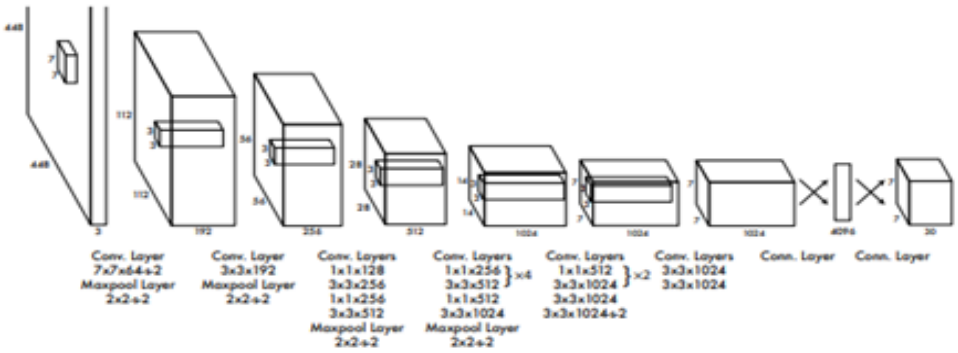


Figure 2.10: YOLO layers

The final output of the network is 7 × 7 × 30 tensor of predictions. So, Yolo divides the image into 7 × 7 cell, each cell is responsible of predicting 2 bounding boxes. The bounding box represents the rectangle that encloses an object. For each bounding box, there is 4 values indicating the dimensions and the position of the bounding box within an image (X and Y coordinates of the center of the bounding box, width and height of bounding box) and a confidence score indicating if there is an object enclosed in this box or no as shown in figure 15.

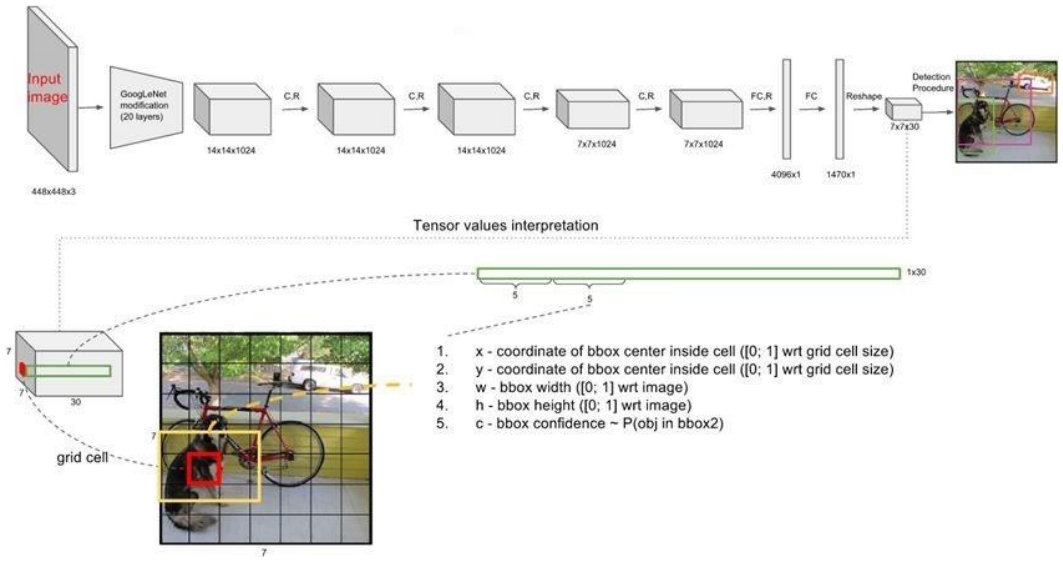


Figure 2.11: Tensor of prediction

As example, if the model is trained on the Pascal VOC data set which can detect 20 different classes, Yolo gives a probability distribution over all possible classes assigning each class with the probability of being enclosed in one of the two bounding boxes detected by the cell. So, for each cell, there is 5 values for the first bounding box and its confidence score, 5 other values for the second bounding box and 20 values presenting the prediction of each class. The confidence score for each bounding box and the class prediction are combined into one final score that tells us the probability that this bounding box containsa specific type of object.

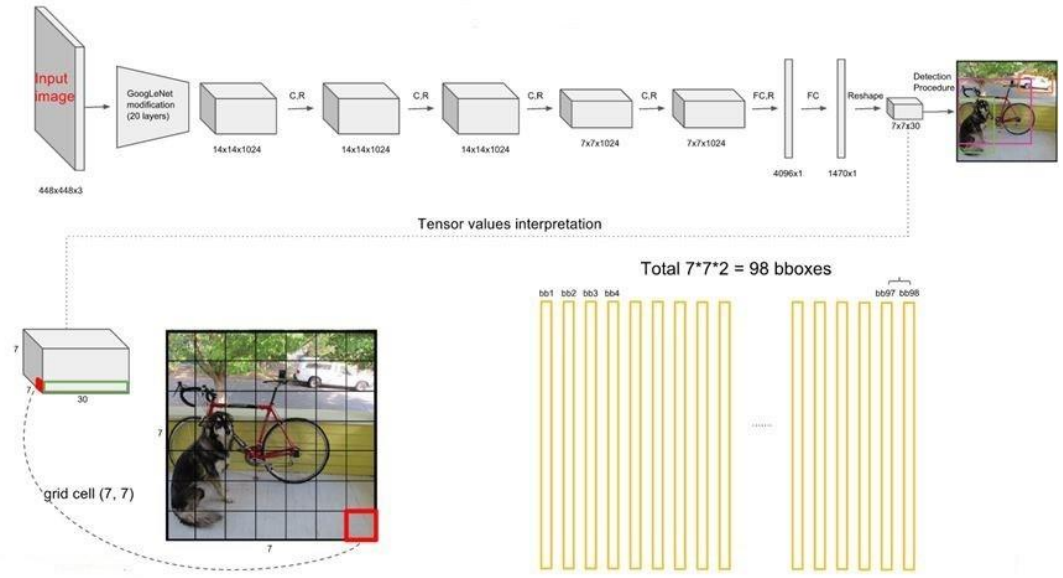


Figure 2.12: Total number of bounding boxes

Now we have class scores for each bounding box, we set a threshold value and set the score of class to zero if the score value is less than the threshold, and then sort them in a descending order and then apply the NMS (Non-Maximum Suppression).

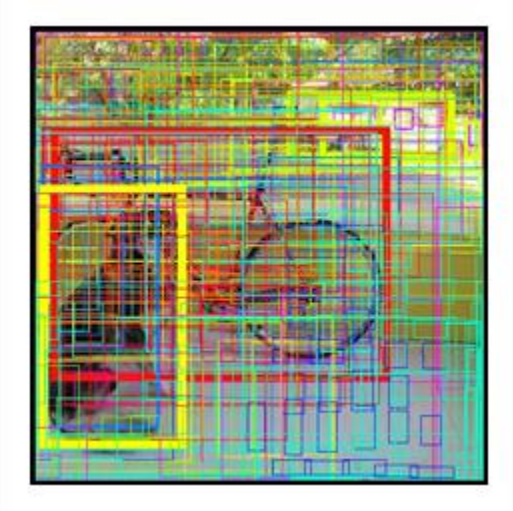


Figure 2.13: First stage image

- **Non-Maximum Suppression**
  As shown in figure 17 there are a lot of overlapped bounding boxes that detects the same object, or another that detect nothing. Therefor non-maximum suppression is applied to get rid of the empty bounding boxes and to reduce the overlapped bounding boxes to only one that best fit the object.
- **How Non-Maximum Suppression Works?**
  For each bounding box in the image there is a class score for each class. For each object get the bounding boxes scores for this object as shown in figure 18.

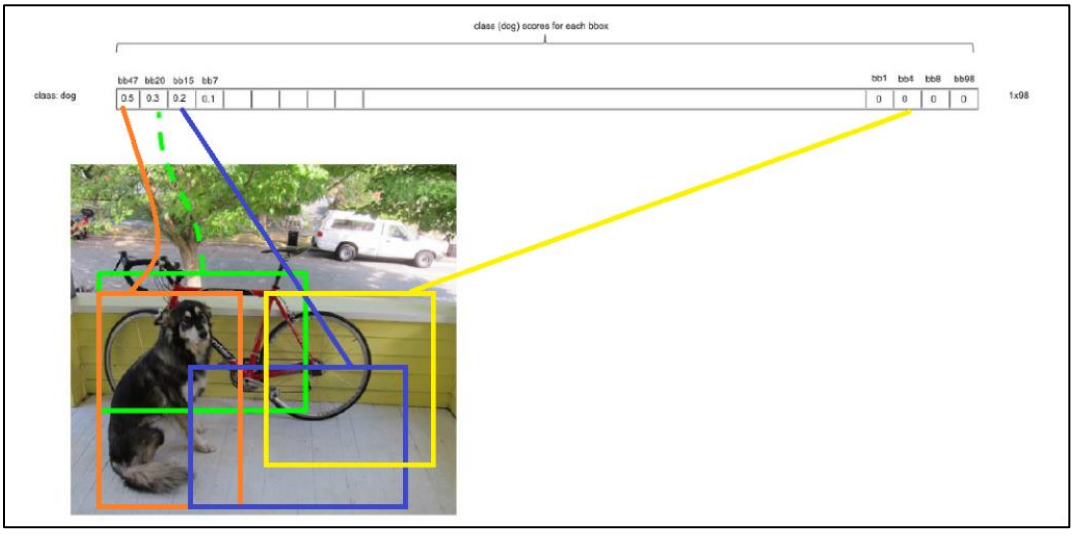


Figure 2.14: Bounding box score for class dog

Compare between the highest score bounding box and others less score that are not zeros and compute the intersection over union of these bounding boxes, if the result more than a predefined

threshold then set the less score to zero, if not then the score will not be changed check figure 19. This process is repeated for all less score bounding boxes with the highest score. Then the same is applies to the next highest score. After comparison almost all pairs of bounding boxes, there will be non-zero class score value. Do this procedure for the next class until all classes are covered.

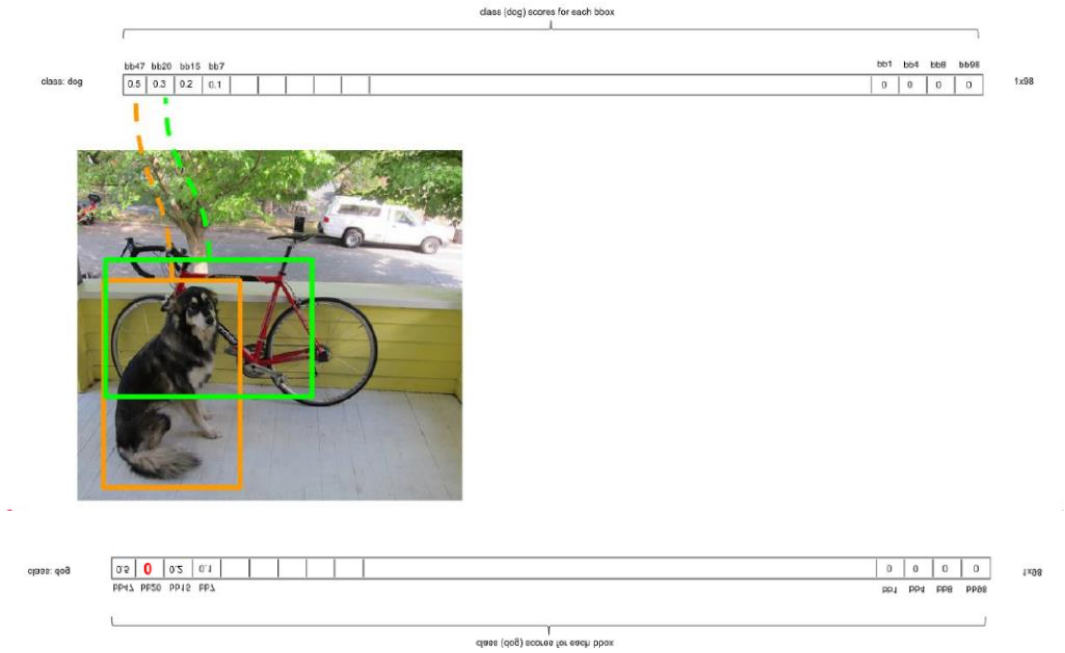


Figure 2.15: Bounding box thresholding If the intersection over union between the orange and green boxes more than the threshold then set the score of the green box to zero.

Figure 2.16 shows the output at the end.

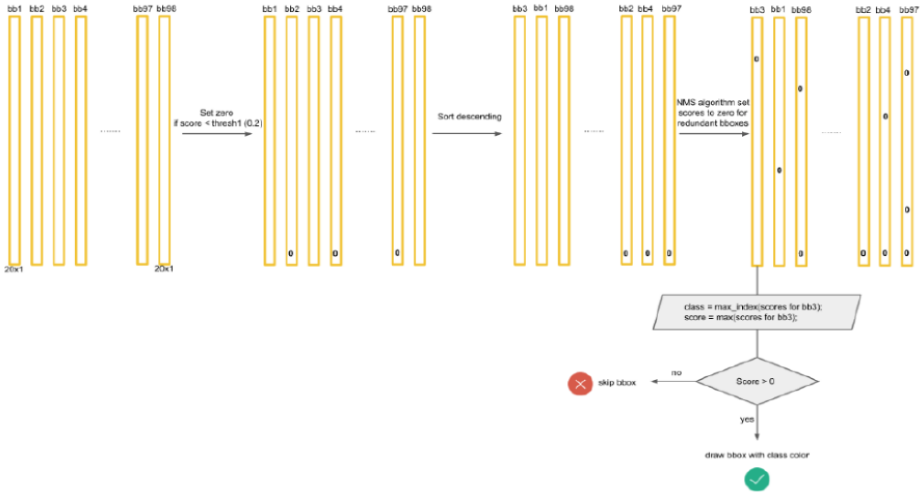


Figure 2.16: Output after thresholding

Each bounding box can contain multiple object, therefor we first get the object with the highest score then we check if this score is higher than zero; draw the box, if not skip this box. The same is repeated for all bounding boxes.

The output image at the image will be as shown in Figure 2.17.

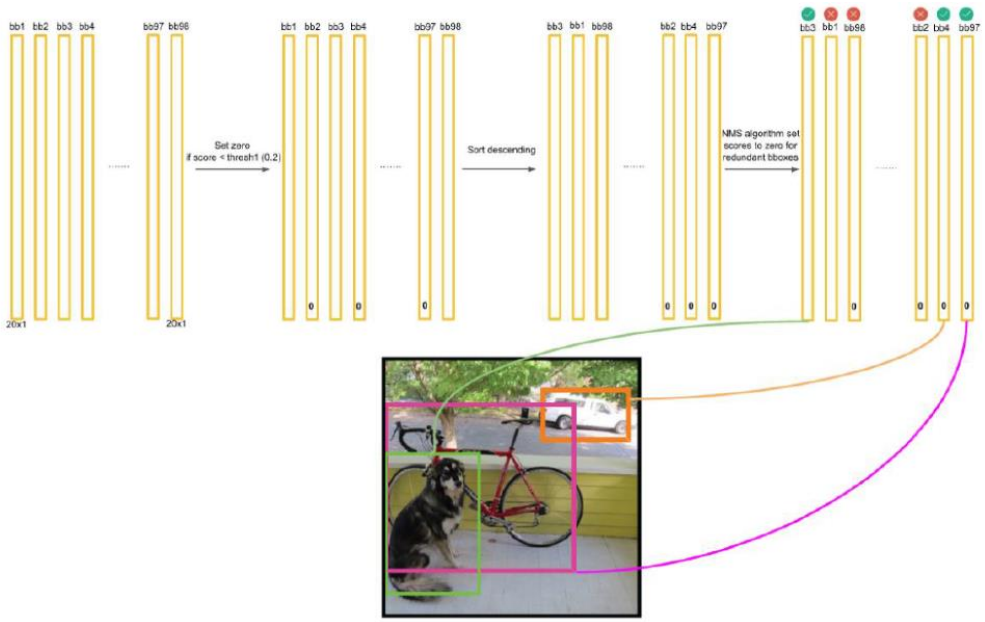


Figure 2.17: NMS output

3. Object Detection With YOLO

3.1. Overview

Asdiscussed before there are a lot of object detection models and the choicebetween them depends on the needs of the project. In our project our target is to make the system a real time system. Object detection in an autonomous car all its concern is the speed not only the accuracy. If there is a pedestrian in front of the car we need to detect it as quickly as possible. Therefor YOLO was the most appropriate model to use. There are two implementations of YOLO one using C and the other using python; we used both implementation and the result of each one will be discussed later in this section in details.

3.2. Progress

First, we installed darknet which is an open source neural networks framework written in C and cuda.It supportsboth CPU and GPU computation.In our project we installed darknet on a ubuntuoperating system,then the process was:

write on the linux terminal the following commands:

1. git clone https://github.com/pjreddie/darknet

2. Cd darknet
3. Make

By this we have the YOLO and there is a folder named cfg where there are text files of all yolo versions' configuration. To download the weights file:

4. wget https://pjreddie.com/media/files/yolov3.weights

Now we can run Yolo using the below command:

5. ./darknet detect cfg/yolov2.cfg yolov2.weights data/dog.jpg

We got this output on the terminal:

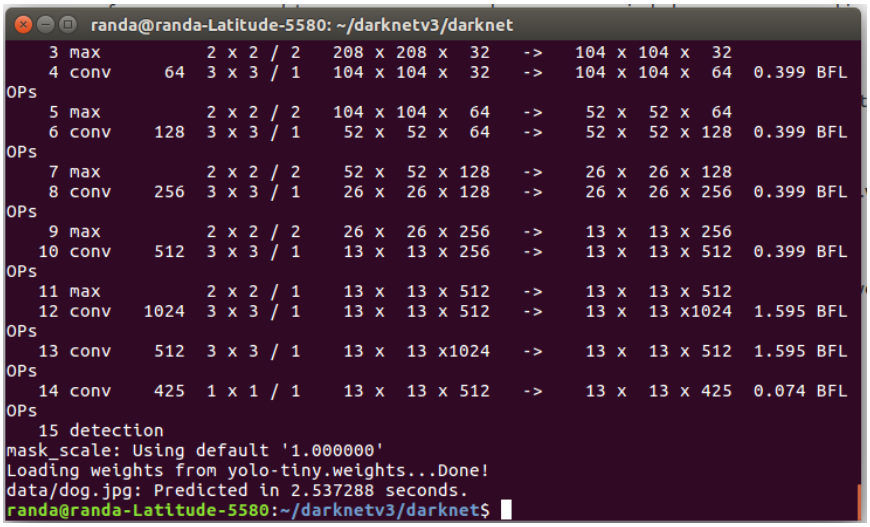


Figure 2.18: Output tiny yolo version 2 on images

To run it on a video or the webcam change the OPENCV line on the make file to 1 instead of 0, then run the command make again.

Run on the webcam write: ./darknet detector demo cfg/coco.data cfg/yolov2.cfg yolov2.weights
Run on video file write: ./darknet detector demo cfg/coco.data cfg/yolov2.cfg yolov2.weights <video file>

When we ran it on a video we got the following result:

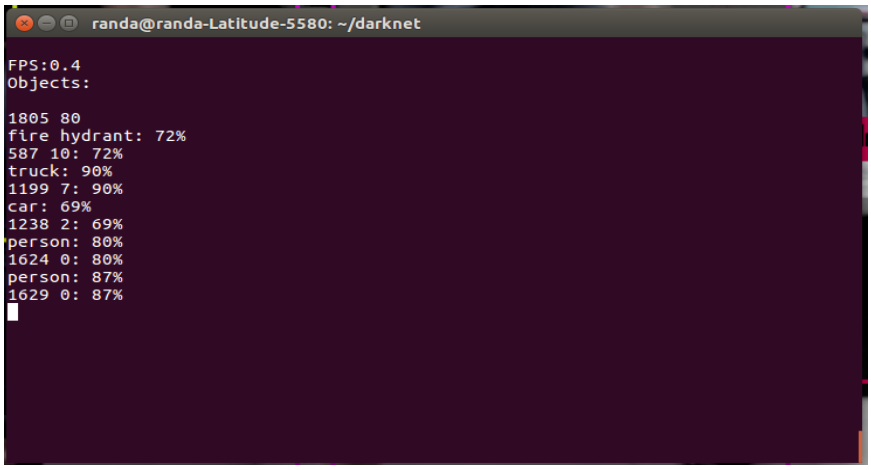


Figure 2.19: YOLO version2 on video

To run yolo on the board we performed the same steps and changed in the make file GPU and CUDNN line to 1.

We tried the different yolo architecture to compare between them and choose the fastest architecture. Mainly we tried yolo version2, tiny yolo version2, yolo version3 and tiny yolo version3. When we used these architectures, we got the following:

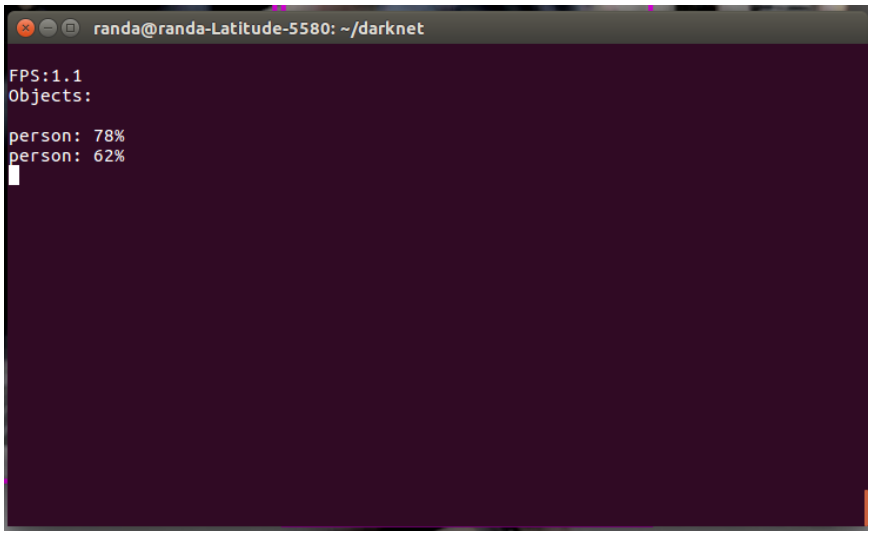


Figure 2.20: Tiny yolo version2

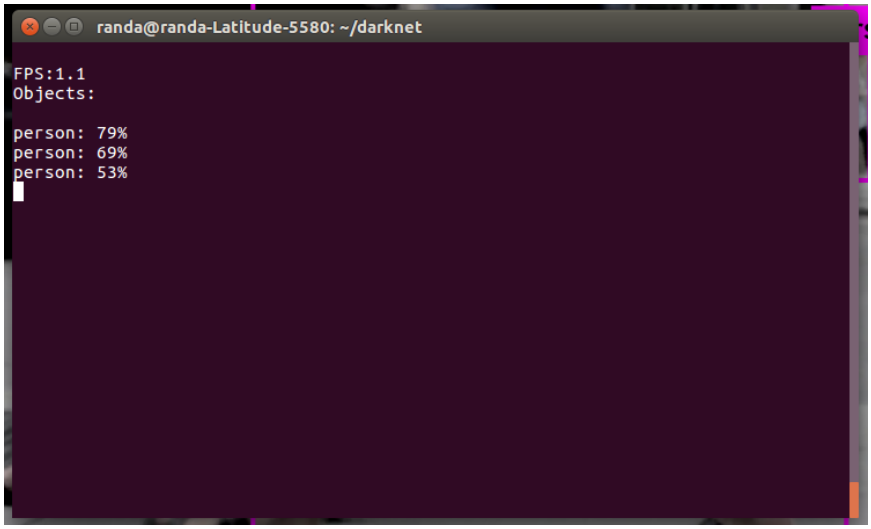


Figure 2.21: Tiny yolo version3

As seen in the results that the performance represented in the number of frames per second (fps) is very low compared to our target which is a real time system.

The first idea proposed was to re-train the system to detect only five objects that are frequently exists in our environment. These objects were: cars, pedestrians, bicycles, motorbikes and dogs.

3.3. Training process:

1. Download VOC data
   wget https://pjreddie.com/media/files/VOCtrainval_11-May-2012.tar
   wget https://pjreddie.com/media/files/VOCtrainval_06-Nov-2007.tar
   wget https://pjreddie.com/media/files/VOCtest_06-Nov-2007.tar
   tar xf VOCtrainval_11-May-2012.tar
   tar xf VOCtrainval_06-Nov-2007.tar
   tar xf VOCtest_06-Nov-2007.tar
2. Generate labels for data.
   First download voc_label.py: wget https://pjreddie.com/media/files/voc_label.py
   Modify the array of objects in line 9 with desired objects only and remove the rest then run voc_label.py: python voc_label.py
3. Write on the terminal: cat 2007_train.txt 2007_val.txt 2012_*.txt > train.txt
4. modify voc.names (in the path: darknet/data) with the desired objects only in the same order as the orders of objects in the voc_label.py. Rename this file obj.names instead of voc.names or according to your preference.
5. modify voc.data (in the path: darknet/cfg). Rename it obj.data or according to your preference.
   classes = 5
   train = (train.txt path)/train.txt
   valid = (2007_test path)/2007_test.txt
   names = data/obj.names
   backup = backup

6. modify tiny-yolo-voc.cfg (in the path: darknet/cfg). Rename it yolo-obj.cfg or according to your preference.
   line 120: classes=5
   line 114: filters=50
7. Download pretrained convolution weights darknet19_448.conv.23:
   wget https://pjreddie.com/media/files/darknet53.conv.74
8. Change detector.c (in the folder examples) line 138 to if(i%1000==0 || (i < 1000 && i%100 == 0))
9. start training:
   ./darknet detector train cfg/obj.data cfg/yolo-obj.cfg darknet19_448.conv.23 gpus 0,1,2,3
   (if training is interrupted restart from backup weights with command: ./darknet detectortrain cfg/obj.data cfg/yolo-obj.cfg backup/yoloobj.backup gpus 0,1,2,3)

We used the VOC datasets where the number of images is approximately 9963 images.

- **The result:**
  After the training we got the results shown in the below figures.

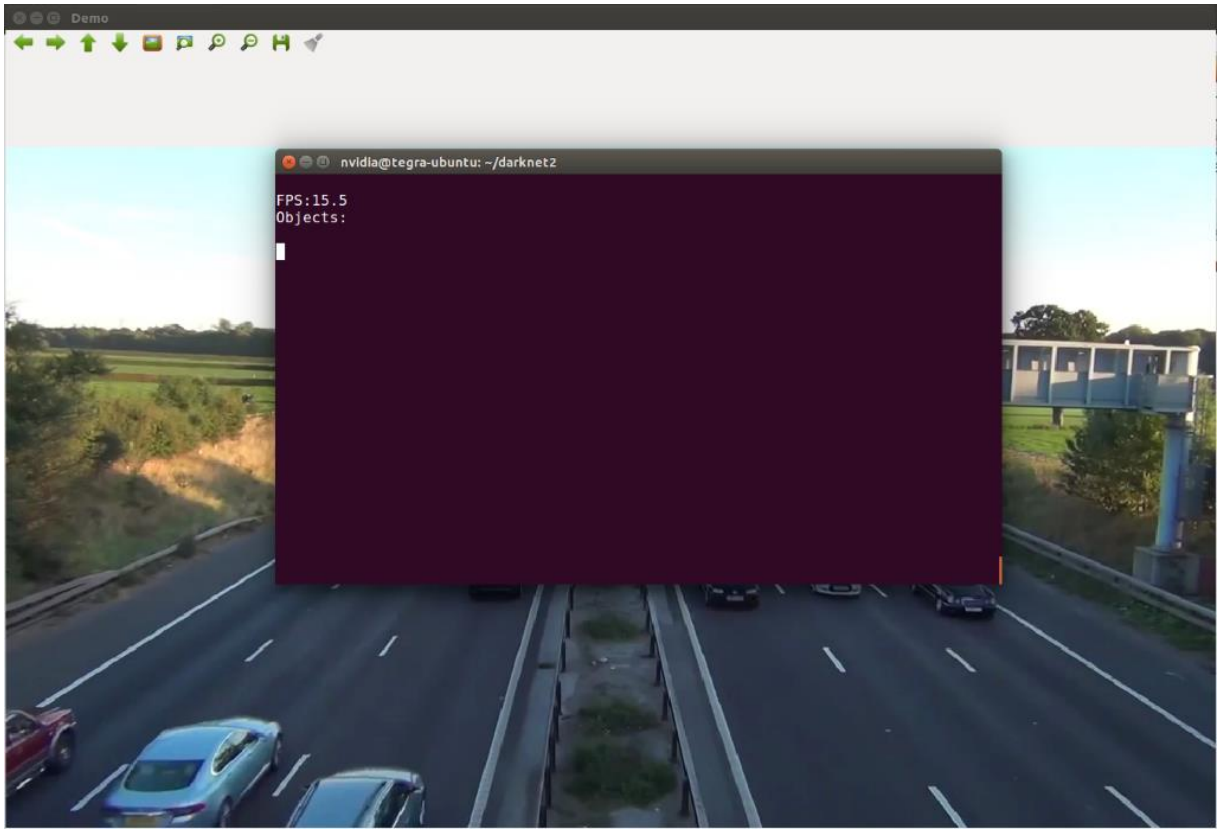


Figure 2.22: FPS after training

So, the fps number is increased to 15.5 but at the same time we got the followingresult.

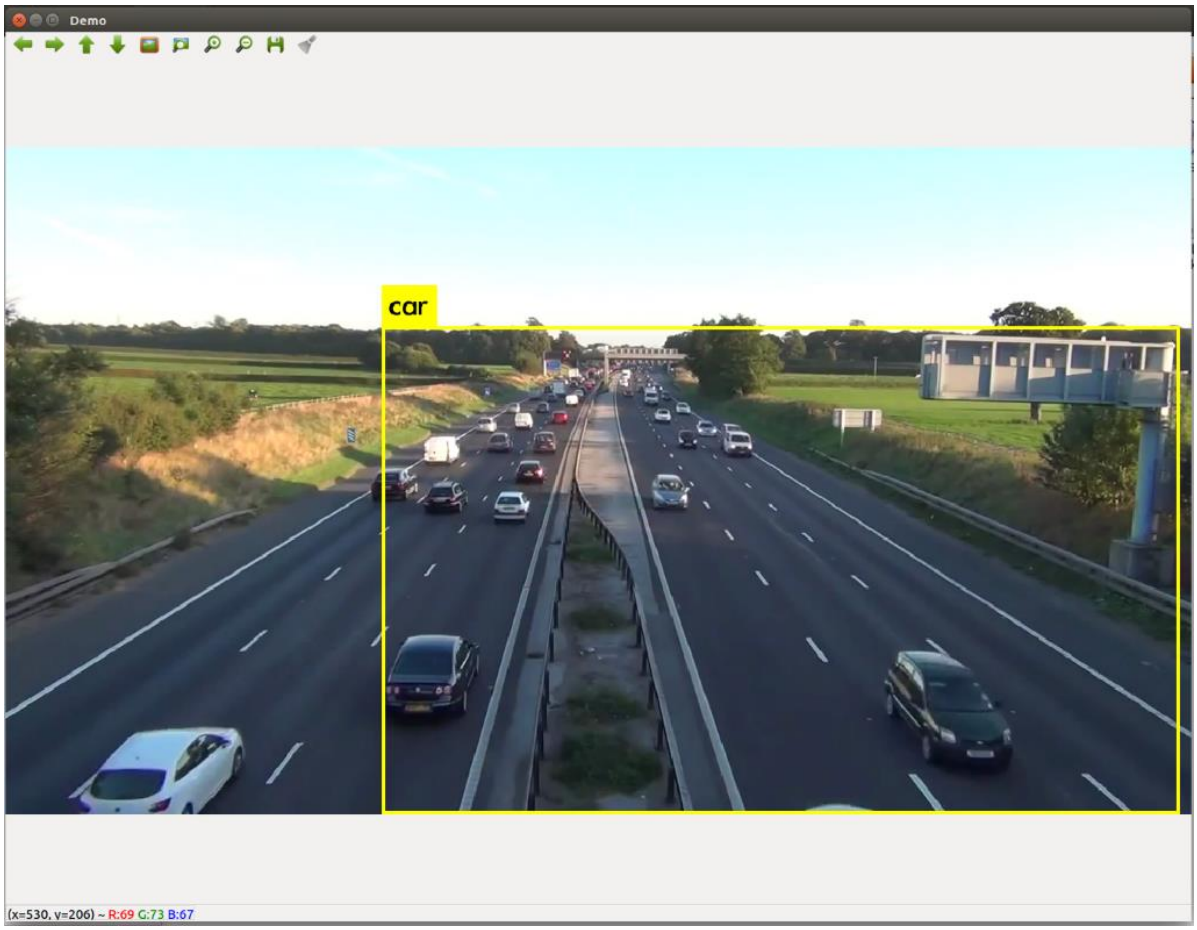


Figure 2.23: Output after training when the threshold is set to 0.1

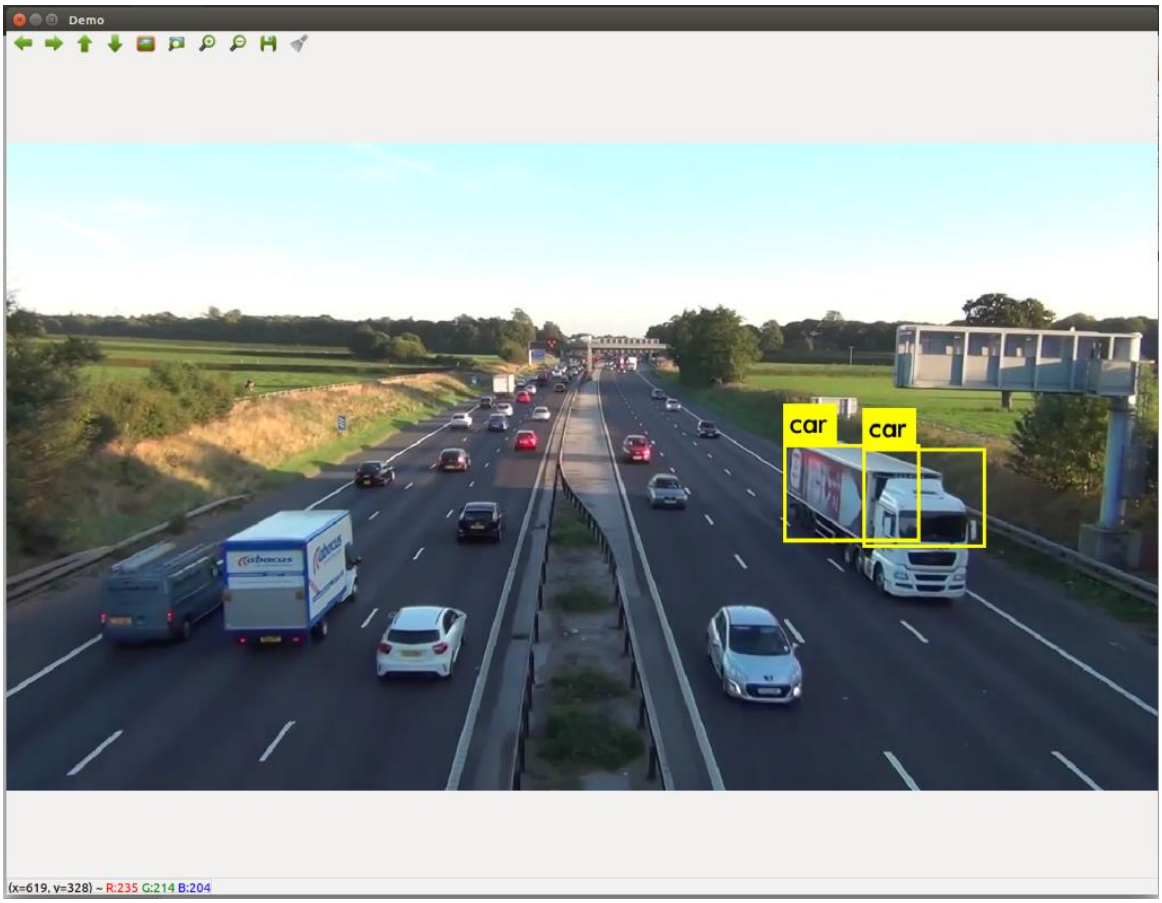


Figure 2.24: Output after training without threshold

Although the fps is increased but the accuracy is very low. The cause of this problem is the numberof training data as it was a small number compared to the number of objects required. Then we had to use another dataset with VOC, so we used Kitti dataset plus a few images taken by us from the faculty.
But the problem was that each dataset has different labeling format and YOLO requires a specific format. Therefore, we need to convert the format of each dataset to the YOLO format.

3.4. Data Handling

First, we got the train images from KITTI Dataset which is 7481images with TEXT Files contain what is inside each image (objects labels, dimensions, location).There was a problem which was inconsistency between the number of car class (28742) and the pedestrian class (4487).

So, we used also the VOC dataset which is 17037 images with XMLS Files contain what is inside each image (objects labels, dimensions, location).

The problem was solved, and we had more consistency numbers of classes train images.

Table 2.1: Dataset used for training darkflow

	VOC	KITTI	Our data	total
Car	2492	28742	480	31714
Pedestrian	17296	4487	0	21783

Now the problem is in different ways of writing data between 'KITTI Dataset' and 'VOC Dataset'. On the other hand, YOLO Dark Flow has certain format for the data which is different from 'KITTI Dataset' and 'VOC Dataset'.

- **KITTI Dataset format:**

```
Car 0.00 2 1.75 600.14 177.09 624.65 193.31 1.44 1.61 3.66 0.24
1.84 66.37 1.76
Car 0.00 0 1.78 574.98 178.64 598.45 194.01 1.41 1.53 3.37 -2.19
1.96 68.25 1.75
DontCare -1 -1 -10 710.60 167.73 736.68
182.35 -1 -1 -1 -1000 -1000 -1000 -10
DontCare -1 -1 -10 758.52 156.27 782.52
179.23 -1 -1 -1 -1000 -1000 -1000 -10


Car 0.00 2 1.75 600.14 177.09 624.65 193.31 1.44 1.61 3.66 0.24
1.84 66.37 1.76
Car 0.00 0 1.78 574.98 178.64 598.45 194.01 1.41 1.53 3.37 -2.19
1.96 68.25 1.75
DontCare -1 -1 -10 710.60 167.73 736.68
182.35 -1 -1 -1 -1000 -1000 -1000 -10
DontCare -1 -1 -10 758.52 156.27 782.52
179.23 -1 -1 -1 -1000 -1000 -1000 -10
```

Figure 2.25: KITTI format

- **The format code:**

1 type Describes the type of object: 'Car', 'Van', 'Truck', 'Pedestrian', 'Person_sitting', 'Cyclist', 'Tram', 'Misc' or 'DontCare'

1 truncated Float from 0 (non-truncated) to 1 (truncated), where truncated refers to the object leaving image boundaries

1 occluded Integer (0,1,2,3) indicating occlusion state: 0 = fully visible, 1 = partly occluded 2 = largely occluded, 3 = unknown

1 alpha Observation angle of object, ranging [-pi..pi]

4 bbox 2D bounding box of object in the image (0-based index): contains left, top, right, bottom pixel coordinates

3 Dimensions 3D object dimensions: height, width, length (in meters)

3 Location 3D object location x,y,z in camera coordinates (in meters)

1 Rotation_y Rotation ry around Y-axis in camera coordinates [-pi..pi]

1 Score Only for results: Float, indicating confidence in detection, needed for p/r curves, higher is better.

- **VOC Dataset format:**

```
<?xml version="1.0" encoding="ISO-8859-1"?>
- <annotation>
    <folder>VOC2012</folder>
    <filename>2007_000027.jpg</filename>
  - <source>
      <database>The VOC2007 Database</database>
      <annotation>PASCAL VOC2007</annotation>
      <image>flickr</image>
    </source>
  - <size>
      <width>486</width>
      <height>500</height>
      <depth>3</depth>
    </size>
    <segmented>0</segmented>
  - <object>
      <name>person</name>
      <pose>Unspecified</pose>
      <truncated>0</truncated>
      <difficult>0</difficult>
    - <bndbox>
        <xmin>174</xmin>
        <ymin>101</ymin>
        <xmax>349</xmax>
        <ymax>351</ymax>
      </bndbox>
    - <part>
        <name>head</name>
      - <bndbox>
          <xmin>169</xmin>
          <ymin>104</ymin>
          <xmax>209</xmax>
          <ymax>146</ymax>
        </bndbox>
      </part>
```

Figure 2.26: VOC format

The KITTI datasets easier for parsing data as it is text file with space between every information than the VOC dataset, so we used Open source code from VOC dataset to KITTI dataset.

This Code Converts between object dataset formats from the folder Command Prompt

```
python vod_converter/main.py -from voc --from-path datasets/mydata-
voc --to kitti --to-path datasets/mydata- kitti
```

Figure 2.27: Format conversion code

This Code generates new XMLs files from KITTI dataset to YOLO darkflow format and takes specified certain number of classes, such as the Car and Pedestrian Classes and make other classes XML files empty.

```
for text,image in zip(texts,image):
  objects=[]
  x=0
  data=open(text,'r')
  script=data.readlines()
  for i in script:
   car_index=i.find('Car')
 Cyclist_index=i.find('Cyclist')
  Cyclist_index=Cyclist_index*int(i.find('bicycle'))
  Pedestrian_index=i.find('Pedestrian')
  motorbike_index=i.find('motorbike')
  dog_index=i.find('dog')
```

Code 2.1: XML generation 1

```
if car_indexb== 0:
objects.append('Car')
for count in i:
if count ==' ':
x +=1
continue
if(x == 4):
tl_x +=count
elif(x==5):
tl_y +=count
elif(x==6):
br_x +=count
elif(x==7):
br_y +=count
elif (x==8):
tl_x=round(float(tl_x))
tl_y=round(float(tl_y))
br_x=round(float(br_x))
br_y=round(float(br_y))
tlc=(tl_x,tl_y)
brc=(br_x,br_y)
tl.append(tlc)
```

```
br.append(brc)
tl_x=' '
tl_y=' '
br_x=' '
br_y=' '
x=0
break
```

Code 2.2: XML generation 2

Then we draw the parsed boxes to ensure that we have the correct data

```
imag=cv2.imread(image.path)
  for p in range(len(objects)):
   cv2.rectangle(imag,tl[p],br[p],(255,0,0),3)
   frame = cv2.putText(imag,objects[p],tl[p], cv2.FONT_HERSHEY_COMPLEX, 1, (0, 0, 0), 2)
   cv2.imshow("1",imag)
   if cv2.waitKey(0)==ord('q'):
     cv2.destroyAllWindows()
     z=0
     break
  """g.write_xml('images',image,objects,tl,br,'annotations')"""
  tl=[]
  br=[]
```

Code 2.3: Draw boxes

Then we write the xml file using the write_xml function.

The Output is Xml Files:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
- <annotation>
    <folder>images</folder>
    <filename>000001.png</filename>
    <segmented>0</segmented>
  - <size>
      <width>1242</width>
      <height>375</height>
      <depth>3</depth>
    </size>
  - <object>
      <name>Car</name>
      <pose>Unspecified</pose>
      <truncated>0</truncated>
      <difficult>0</difficult>
    - <bndbox>
        <xmin>388</xmin>
        <ymin>182</ymin>
        <xmax>424</xmax>
        <ymax>203</ymax>
      </bndbox>
    </object>
  - <object>
      <name>Cyclist</name>
      <pose>Unspecified</pose>
      <truncated>0</truncated>
      <difficult>0</difficult>
    - <bndbox>
        <xmin>677</xmin>
        <ymin>164</ymin>
        <xmax>689</xmax>
        <ymax>194</ymax>
      </bndbox>
    </object>
</annotation>
```

Figure 2.28: Output XML

3.5. Darkflow

we tried another approach as the darknet version was harder to integrate it with the other parts of the project. Therefore, we tried the darkflow version of YOLO which is written in python. This implementation makes it easy to integrate.

- **Download Process:**
  1. Download the darkflow repo: https://github.com/thtrieu/darkflow.
     git clone https://github.com/thtrieu/darkflow.
  2. Open a terminal inside the darkflow folder and type: python setup.py build_ext –inplace
  3. Just like darknet we used the yolo weights.
  4. Create new folder called bin and move the downloaded weights to this folder.
  5. To test darkflow with video open a terminal and type:
     python flow --model cfg/yolo.cfg --load bin/yolov2.weights --demo videofile.mp4 --gpu 1.0 --saveVideo
     where videofile.mp4 is the name of the video used.
     In case we are not using a GPU then –gpu 1.0 part is removed.

Without GPU the output was:

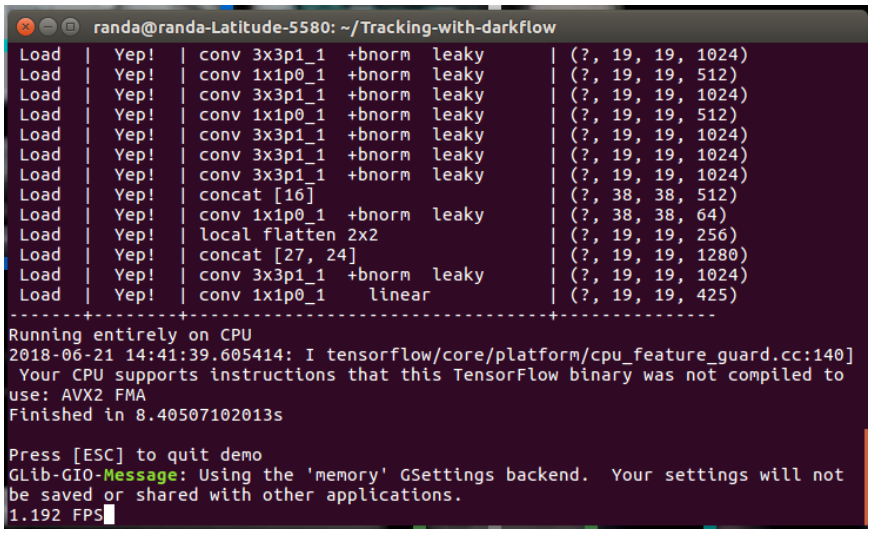


Figure 2.29: Output without GPU

- **How to use darkflow**

Create new python project and inside this project import TFNet from thepath darkflow/net/build.Then create a dictionary named options or any other name you want, inside thedictionary identify the model, the weights, threshold and the gpu. Then pass this optionsdictionaryto the TFNet function.

To run it on video then use opencv library to capture video from file or from a camera and for each frame send it to the predict function to detect the object inside this frame.

To compute numbers of fps we got the time before prediction (=start) and after it (=end), the inverse of the difference between end and start time is the fps value.

To draw detections, the output from the prediction function is the bounding box coordinate represented in the top left and bottom right values of the box, object label and the confidence score. We got these values separately and used the bounding box coordinates to draw the rectangle and got the label to place the detected class on the image.

The same code is applied in case we want to read input from a camera but there will be a few changes. Indicate the camera port in the capture video function, then we need to resize the input frame.

3.6. DarkflowTraining

The start is from data handling part as discussed in this section. After we got the training data images annotated with the appropriate labels convenient with YOLO format, darknet training steps 4,5 and 6 are applied.

Then type the command: ./flow --model cfg/yolo-obj.cfg --load bin/yolov2.weights --train --annotation <annotations path> --dataset <dataset images path> --gpu <number between 0 and1> --epoch <specify the number>

The number of epochs depends on the amount of training data, if the amount of data is small then the epoch number must be big value and if the amount of data is big then we need less number of epochs.

Start training and monitor the output in the terminal window. Continue the training until the loss value is as small as possible or until it is not lowering anymore. During training there is a folder called "ckpt" is created, this folder contains all the weights file that will come from the training.
After the training is finished, modify the options dictionary in the python code file with the new model. Also, modify the load entry with the number step of the last weight in the ckpt folder. And the rest of the code is the same.

- **The result**
  After we have done the training we got the following results.

Figure 2.30: Output after darkflow training (1)

Figure 2.31: Output after darkflow training (2)

Figure 2.32: Output after darkflow training (3)

Figure 2.33: Output after darkflow training (4)

The results as shown in the figures above are more accurate as we increased the amount of training data. But the frame rate is increased, when we ran darkflow after training on our laptops we got 5fps and when we ran it on the board we got 7 fps.

This decrease in performance is caused by the fact that darkflow is wrapping the c code to python code. Therefore, it is slower. To overcome this problem while still using darkflow, tracking with Kalman filter can be used. This idea is discussed in detail in the next chapter.

Chapter 3

Tracking Using Kalman Filter

1. Tracking

1.1. Overview

A tracking algorithm wants to know where something is heading. Which means that its main goal is to keep a steady watch on one or many moving objects over time in a video stream. But to keep track of an object we must first detect that object. Therefore, it is often used with object detection and recognition.

One of the things we must take into consideration while tracking objects is its dynamics. The dynamics of an object means the way this object moves across frames.

1.2. Tracking with Dynamics

The key idea of tracking with dynamics that we must have a model that represent the motion of the object like the velocity or the acceleration. Using that model, we want to predict where the objects will be in the next frame. The prediction will allow us to restrict where we are going to search for that object in the next frame. Also, it improves the estimation because we are combining a prediction with the measurements we got from the model.

1.3. Detection vs Tracking

The use of the dynamics is the difference between tracking and just detection. In detection we independently detect the objects in each frame. But in tracking we take the first frames to detect objects then predict its new positions in the new frames using estimated dynamics. Therefore, the basic loop of tracking is first detection to get our model then prediction to assume the location in the next frame.

1.4. Tracking Assumptions

Tracking assume continuous motion across frames such that objects do not disappear and reappear in different places in the scene. Also, camera is not moving incautiously to a new viewpoint.

1.5. State vs Measurement

Measurement is the things that we can directly observe while state is the things we want to keep track of. Therefore, states are the true parameters we care about,and we want to estimate, and measurementsare the observations of the underlying state.

At each time step, the state changes and we get a new observation. Our goal is to estimate the distribution of the new state. If we have the distribution or belief of what the new state might be, I can estimate the highest probability or the average guess of the new state.

1.6. Tracking steps

Tracking is mainly split into two steps Prediction and Correction. Prediction means what is the next state of the object given past measurements. Therefore, given that $X_t$is the state at time t and $Y_0 = y_0$ ….., $Y_{t-1} = y_{t-1}$ are all the previous measurements, then the prediction will be

$$P (X_t| Y_0 = y_0 \ldots\ldots, Y_{t-1} = y_{t-1}) \quad (3.1)$$

Correction is an updated estimate of the state from prediction and measurements.If $Y_t = y_t$is the current measurement, then the correction will be

$$P (X_t| Y_0 = y_0 \ldots\ldots, Y_{t-1} = y_{t-1}, Y_t = y_t) \quad (3.2)$$

Therefore, the tracking process is

1. Start with an estimated prior distribution.
2. At first frame take a measurement.
3. Then correct our estimate.

Keep repeating these steps, at every time t we make a prediction for time t+1, take a measurement and we correct for time t+1.

2. Kalman Filter

2.1. Overview

Kalman filter is a method of tracking linear dynamical models with gaussian noise. We can use Kalman filter whenever there is uncertain information about some dynamic system, and we want to make a guess about what the system is going to do next.

Kalman filters have the advantage that they are light on memory as they don't need to keep any history other than the previous state, and they are very fast, making them well suited for real time problems and embedded systems.

We have two distinct set of equations: Time Update that can be thought of as prediction equations, and Measurement Update that can be thought of as correction equations. Both equation sets are applied at each time k.

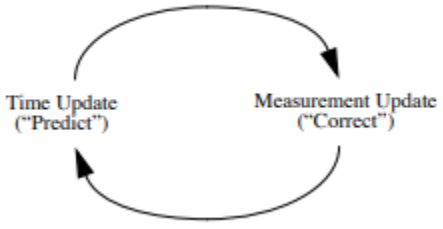


Figure 3.1: Kalman filter cycle

2.2. How Kalman Filter Works

As a small example to understand how it works. If state of our model has only position and velocity.

$$\vec{x} = \begin{bmatrix} p \\ v \end{bmatrix} \tag{3.3}$$

Where x is the state, p is the position and v is the velocity.

We do not know what the actual position and velocity are; there are a whole range of possible combination of position and velocity that might be true, but some of them are more likely than others.

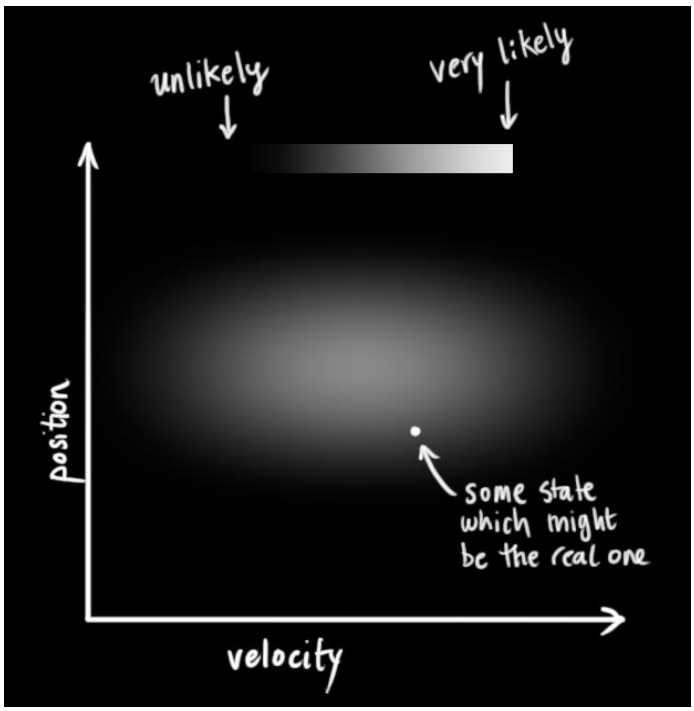


Figure 3.2: Probability distribution

In this case Kalman filter assumes that both variables are random and gaussian distributed. Each variable has a mean which is the center of the random distribution, and a variance which is the uncertainty.

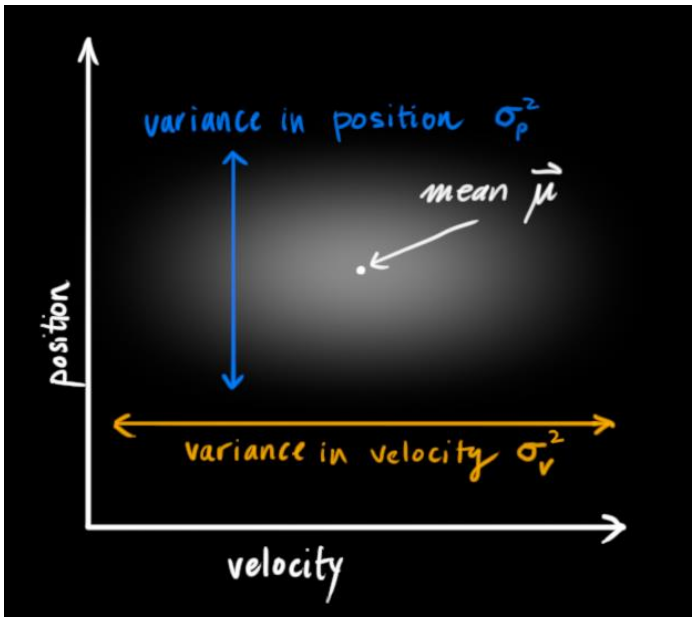


Figure 3.3: Gaussian distribution mean and variance

If the state variables are correlated, which means that one variable is dependent on the other, the shape of the distribution will be different and instead of the variance we will be caring about the covariance between variables. Each element of the covariance matrix represents the degree of relation between two states variables.

$$\overrightarrow{x_k} = \begin{bmatrix} position \\ velocity \end{bmatrix} \tag{3.4}$$

$$P_k = \begin{bmatrix} \Sigma_{pp} & \Sigma_{pv} \\ \Sigma_{vp} & \Sigma_{vv} \end{bmatrix} \tag{3.5}$$

P_k is the covariance matrix and Σ_{vp} is the covariance between velocity andposition.

2.3. Time Update

Now we have the current state at time k-1 and we need to predict the next state at time k. The prediction function works on the current distribution and gives us a new one.

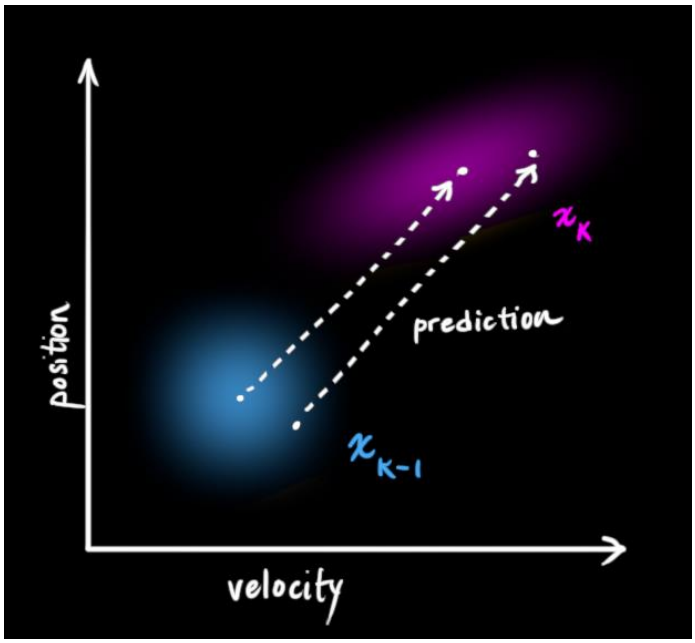


Figure 3.4: Prediction

This prediction can be represented by a matrix F_k. It takes every point in our original estimate and moves it to a new predicted one, which is where the system would move if that original estimate was the right one.

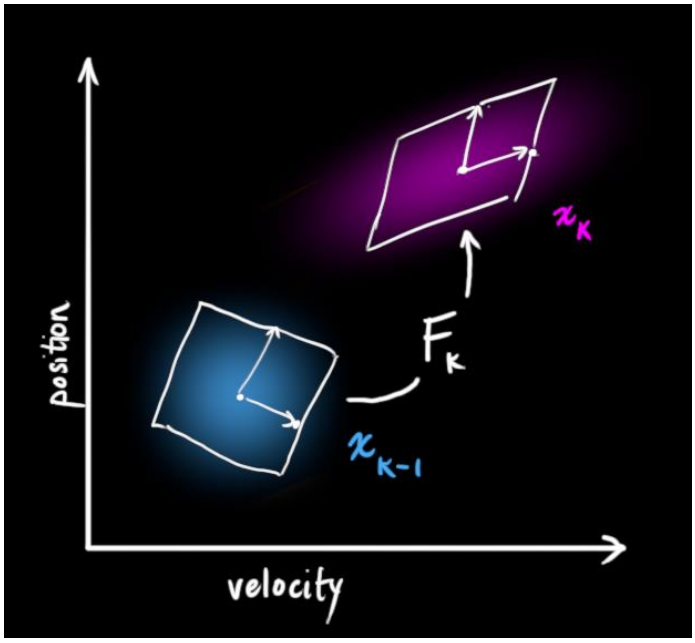


Figure 3.5: Prediction matrix

Thus,

$$\overrightarrow{x_k} = F_k \overrightarrow{x_{k-1}} \tag{3.6}$$

Also, we need to update the covariance matrix. Therefore,

$$P_k = F_k P_{k-1} F_k^T \quad (3.7)$$

- **External Influence**
  There might be some changes that are not related to the state itself, the outside world could be affecting the system. If we know this additional information about the external world, we could add it into a vector called $u_k$and add it to our prediction as correction.

$$\overrightarrow{x_k} = F_k \overrightarrow{x_{k-1}} + B_k \vec{u}_k \quad (3.8)$$

B_k is called the control matrix and $u_k$the control vector.

- **External Uncertainty**
  External forces can affect the system, but we are not aware of these forces, as wind or gravity. We cannot keep track of these things, and if any of this happens, our prediction could be off. We can model the uncertainty associated with the world that we arenot keeping track of by adding some new uncertainty after every prediction step. By treating the untracked influences as noise with covariance we got the new expression for the covariance matrix.

$$P_k = F_k P_{k-1} F_k^T + Q_k \quad (3.9)$$

Where $Q_k$is the untracked uncertainty from the environment.

2.4. Measurement Update

We might have several sensors which give us information about the state of our system. Each sensor tells us something indirect about the state, in other words, the sensors operate on a state and produce a set of readings. We will model the sensors with a matrix H_k.

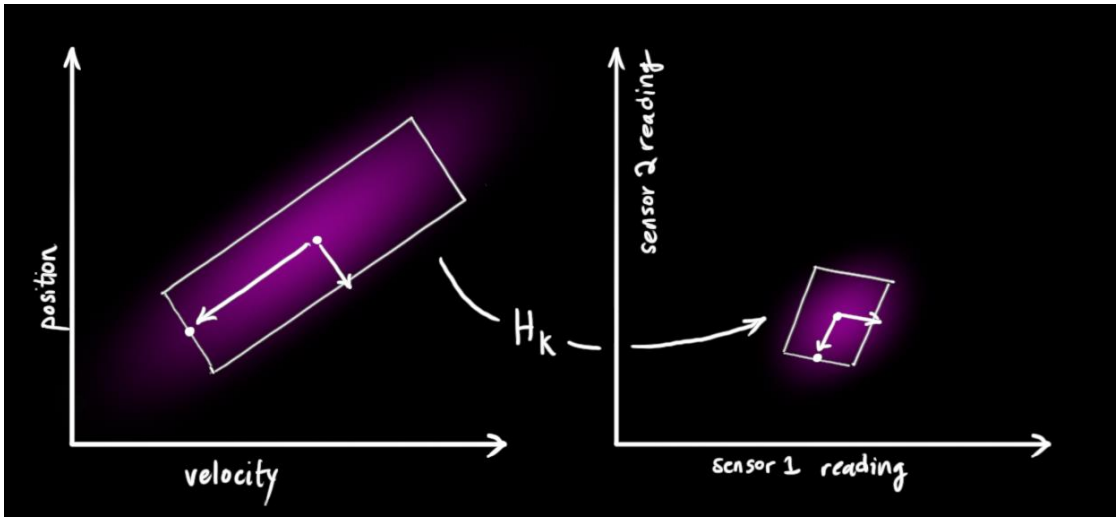


Figure 3.6:Transition between sensor reading and state variables

One thing that Kalman filters are great for is dealing with sensor noise, as sensors can be unreliable. From each reading we observe, we might guess that our system was in a particularstate. But because there is uncertainty, some states are more likely than others to haveproduced the reading we saw. We

will call the covariance of this uncertainty R_k. The distribution has a mean equal to the reading we observed, we will call this mean z_k.

Now we have two gaussians,one surrounding the mean of our prediction, and one surrounding the actual sensor reading we got.

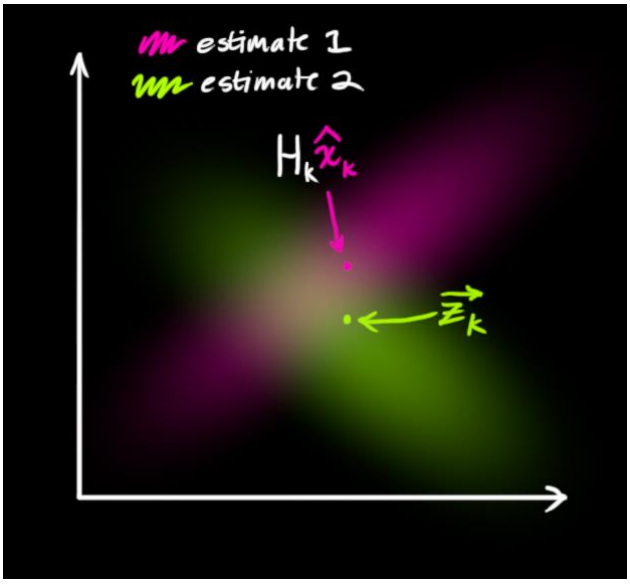


Figure 3.7: The sensor reading and prediction gaussians

The overlap between both gaussians is the region where both are likely. The mean of this distribution the best guess of the true configuration given all the information we have. If we multiply both gaussians we get a new gaussian distribution with new mean and covariance matrix.

New mean and covariance will be

$$\mu' = \mu + K(\mu_1 - \mu_0) \quad (3.10)$$

$$\sigma'^2 = \sigma_0^2 - K\sigma_0^2 \quad (3.11)$$

Where K is a matrix called Kalman gain.

- **Putting it all together:**
  All equations of the update step

$$\vec{x}'_k = \vec{x}_k + K'(\vec{z}_k - H_k\vec{x}_k) \quad (3.12)$$
$$P'_k = P_k - K'H_kP_k \quad (3.13)$$
$$K' = P_kH_k^T\left(H_kP_kH_k^T + R_k\right)^{-1} \quad (3.14)$$

  $\vec{x}'_k$ is the new best estimate and we can go on and feed it back into another roundof predict or update.

3. Object Tracking

3.1. Overview

Now that we talked about object tracking and Kalman filter in details, we want to use this idea in object detection. Tracking accelerate the process of object detection, because instead of applying object detection algorithms on each frame we could now apply tracking which is faster than detection.

Tracking is faster than detection because when you are tracking an object that was detected in the previous frame, you know a lot about the appearance of the object. You also know the location in the previous frame and the direction and speed of its motion. So, in the next frame, you can use all this information to predict the location of the object in the next frame and do a small search around the expected location of the object to accurately locate the object.

3.2. Algorithm

Kalman filter consists of two steps: prediction and update. The first step uses previous states to predict the current state. The second step uses the current measurement, such as detection bounding box location, to correct the state.

First, we must implement Kalman filter. Indicate the state vector and initialize the equations' matrices.

After the Kalman filter implementation a detection to tracker assignment must be implemented. Detection to tracker assignment takes the current list of trackers and new detections and output matched detections, unmatched trackers, and unmatched detections. If there are multiple detections, we need to match (assign) each of them to a tracker. We use intersection over union (IOU) of a tracker bounding box and detection bounding box as a metric. We solve the maximizing the sum of IOU assignment problem using the Hungarian algorithm (also known as Munkres algorithm). The machine learning package scikit-learn has a build in utility function that implements Hungarian algorithm.

Based on the linear assignment results, we keep two list for unmatched detection and unmatched trackers, respectively. When a car enters a frame and is first detected, it is not matched with any existing track, thus this particular detection is referred to as unmatched detection.

In addition, any matching with an overlap less than iou threshold signifies the existence of an untracked object. When a car leaves the frame, the previously established track has no detection to associate with. In this scenario, the track is referred to as unmatched track. Thus, the tracker and detection associated in the matching are added to the lists of unmatched trackers and unmatched detection, respectively.

Also, we need to indicate two parameters, minimum hits and maximum age. Minimum hits number of consecutive matches needed to establish a track. The parameter maximum age is number of consecutive unmatched detection before a track is deleted. Both parameters need to be tuned to improve the tracking and detection performance.

3.3. Tracking Algorithms

Simple Online and Realtime Tracking (SORT) algorithm and Simple Online and Realtime Tracking with a Deep Association Metric (Deep SORT)are two object tracking algorithms using Kalman filter that we used with YOLO in this project.

SORT is a barebones implementation of a visual multiple object tracking framework based on rudimentary data association and state estimation techniques. It is designed for online tracking applications where only past and current frames are available and the method produces object identities on the fly. While this minimalistic tracker does not handle occlusion or re-entering objects its purpose is to serve as a baseline and testbed for the development of future trackers.This approach obviously yields a multi-purpose algorithm: SORT doesnot need to know which type of object we track. It doesn't even need to learn anything: to perform the associations SORT uses mathematical heuristics such as maximizing the IOUmetrics between bounding boxes in neighboring frames. Each box is labeled with a number (object id), and if there is no relevant box in the next frame, the algorithm assumes that the object has left the frame.

Deep SORT is a new and improved version of SORT.It was designed specially to reduce the number of switching between identities, ensuring that the tracking is more stable.

3.4. Results

By applying the above algorithms, we got the following results using YOLO and yolov2 weights on our laptops.

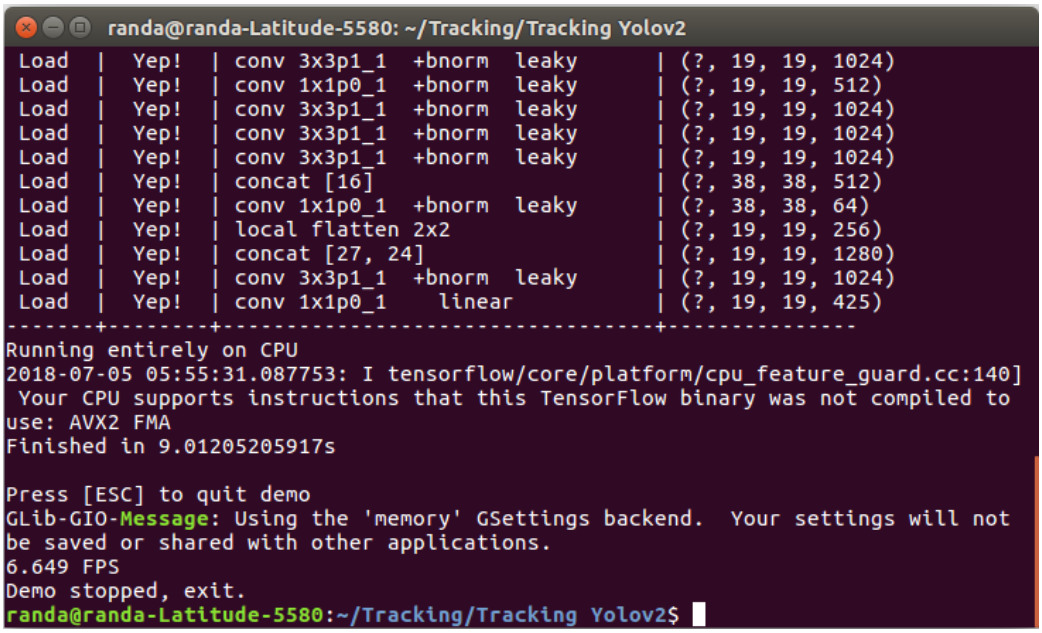


Figure 3.8: Using SORT algorithm, and skipping 5 frames without tracking

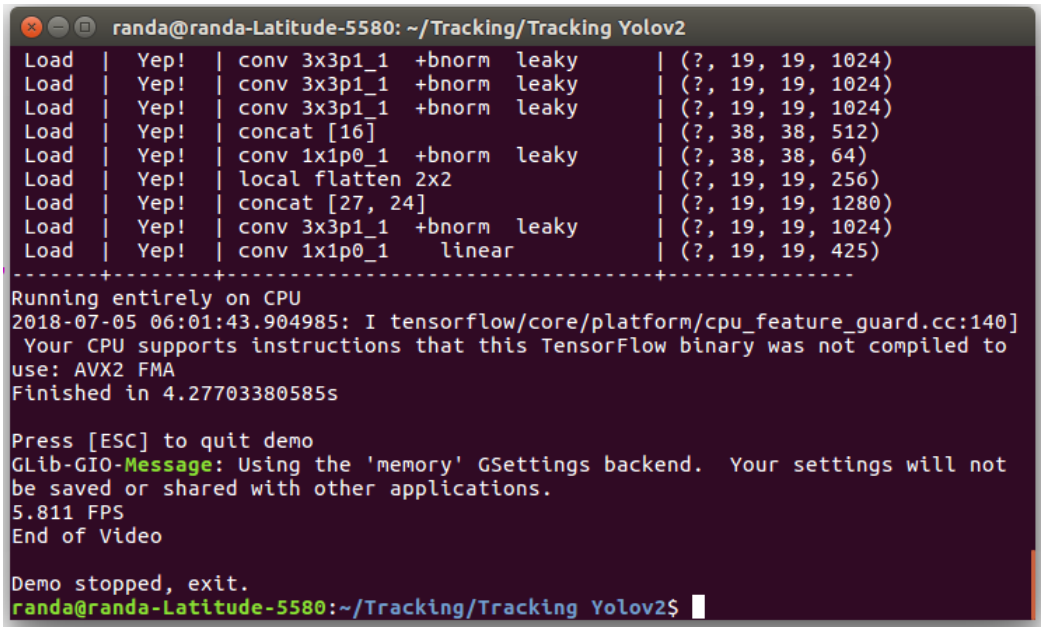


Figure 3.9: Using SORT algorithm and skipping 5 frames with tracking.

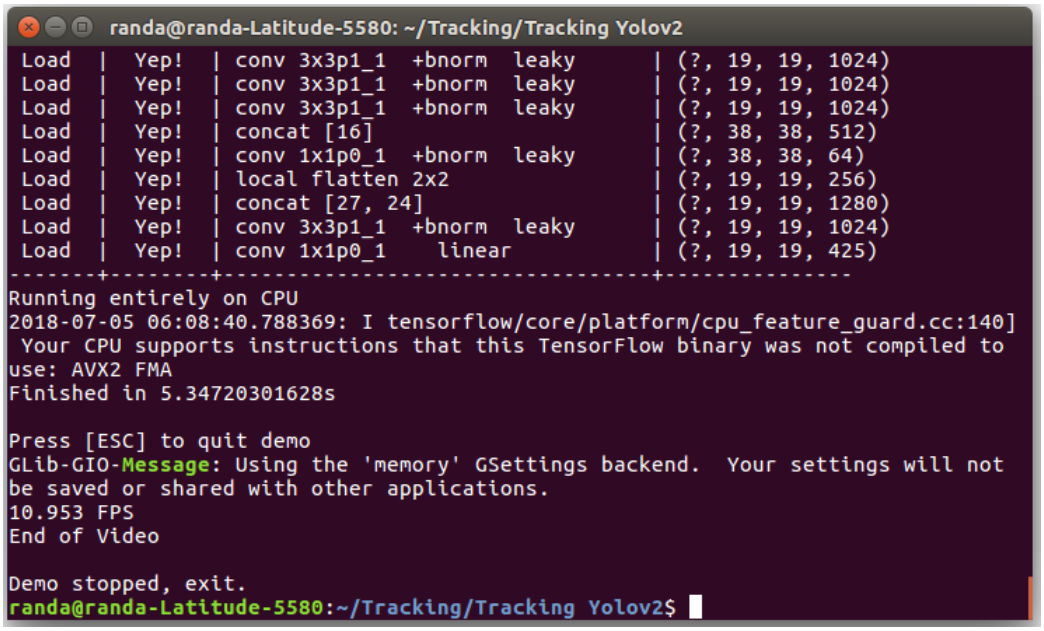


Figure 3.10: Using SORT algorithm, and skipping 10 frames without tracking.

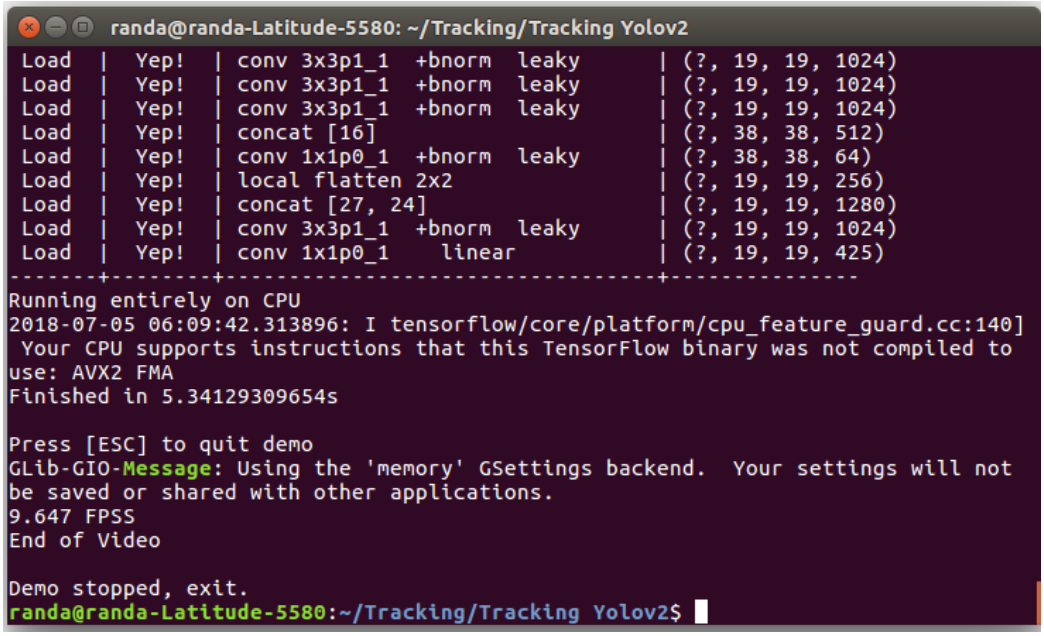


Figure 3.11: Using SORT algorithm, and skipping 10 frames with tracking.

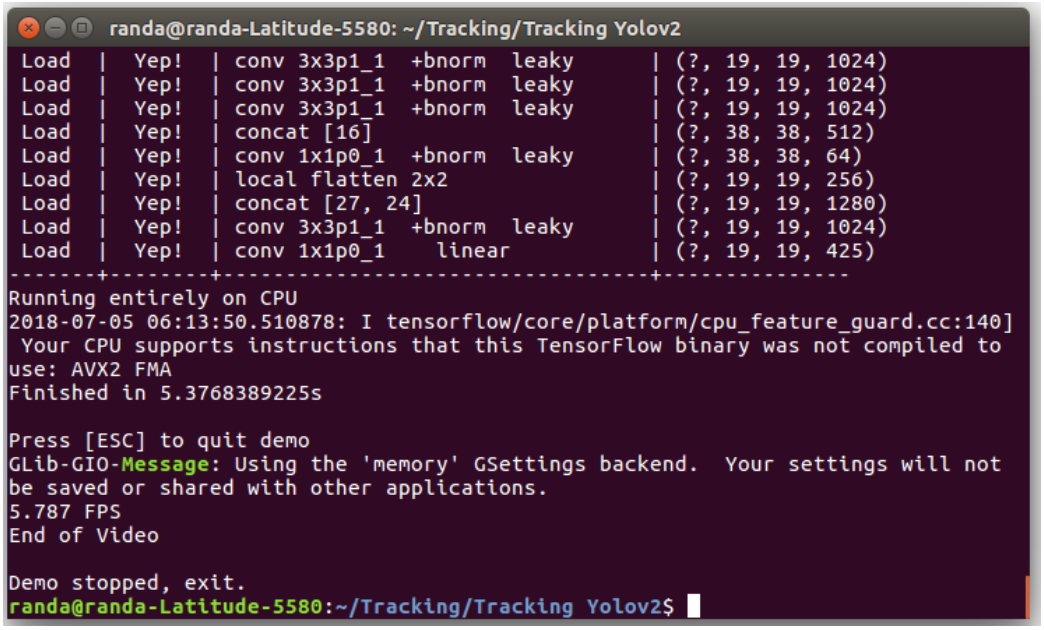


Figure 3.12: Using deep SORT algorithm, and skipping 5 frames without tracking.

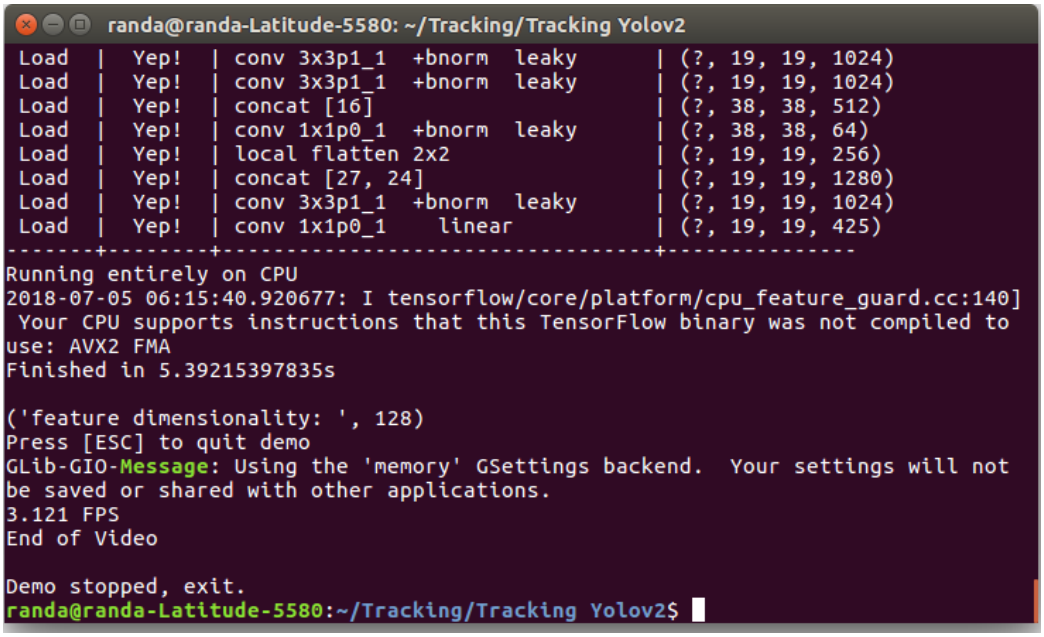


Figure 3.13: Using deep SORT algorithm, and skipping 5 frames with tracking.

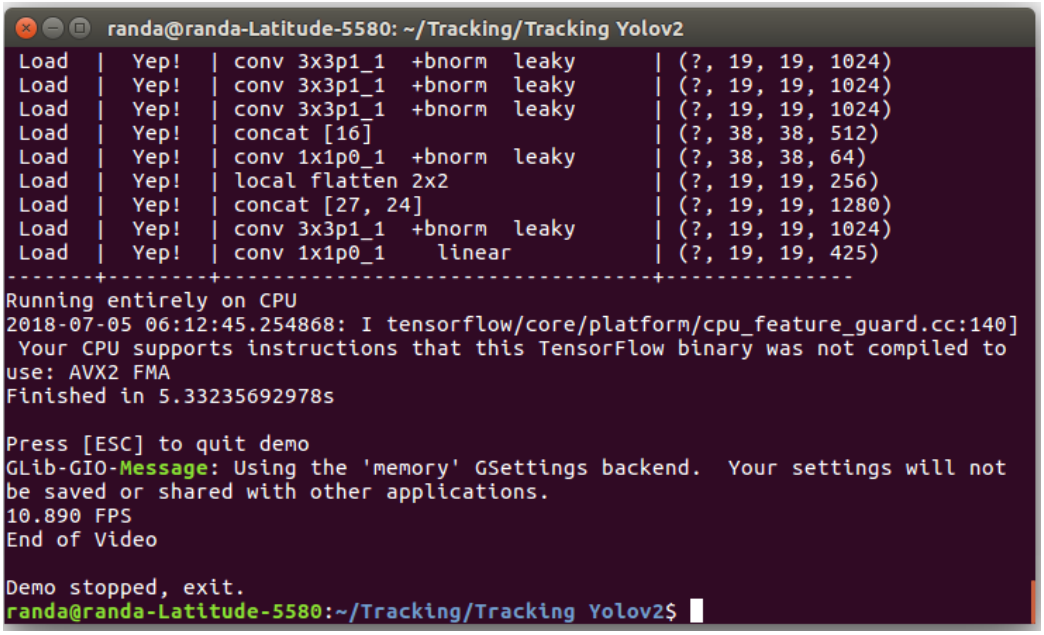


Figure 3.14: Using deep SORT algorithm, and skipping 10 frames without tracking

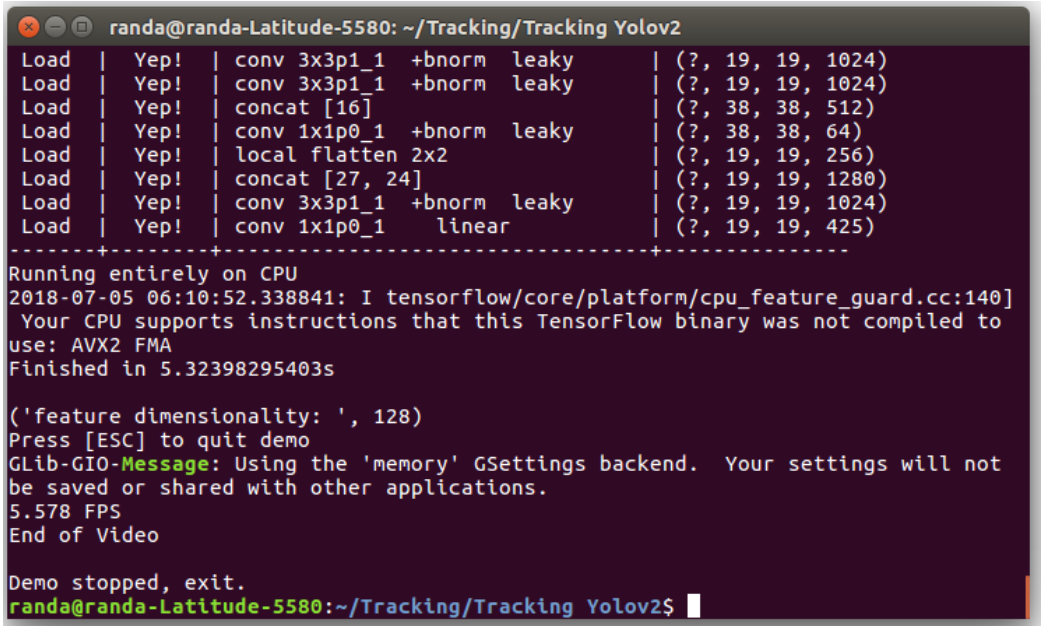


Figure 3.15: Using deep SORT algorithm, and skipping 10 frames with tracking.

Skipping only means that we skip number of frames without applying detection on them. While skipping with tracking means that we stop detection on these frames, but we apply tracking on them. Therefore, skipping alone is faster than skipping with tracking.

CHAPTER 4

Depth Estimation

1. Introduction

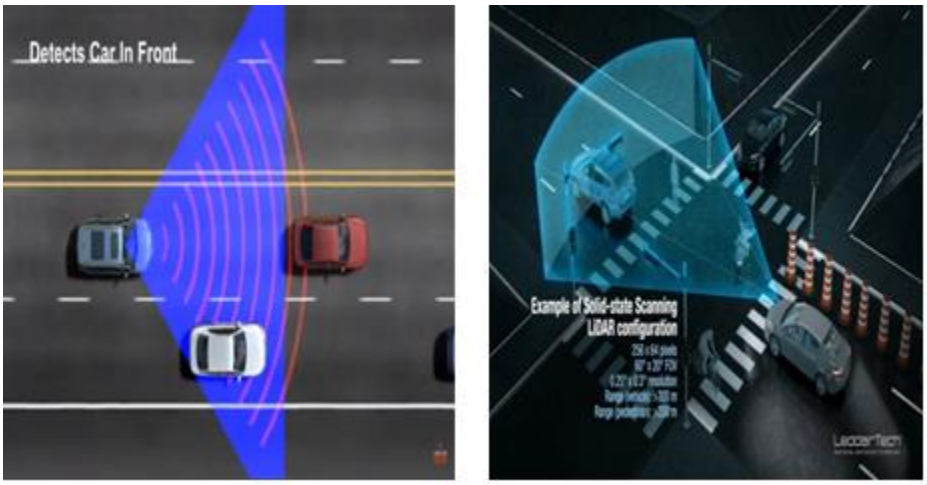
1.1. Overview

Alot of accidents are happening nowadays because drivers donot keep enough distance between their cars and other cars in the road. A fatal collision can occur within a split of second if the drivers didnot hit the brakes in time.so for an autonomous car, the car must know how far objects to be able to take different decisions like (increase or decrease speed, sudden breaking, steering, ….).

To makes depth estimation possible, there are three main groups of sensor systems: camera, radar, and lidar based systems

1.2. Depth estimation sensor systems

- **RADAR**
  The word RADAR stands for Radio Detection and Ranging, which means detect objects and determine their range, angle, and/or velocity using radio waves.
  The RADAR can measure distance up to 500 meters.
- **LIDAR**
  The word LIDAR stands for Light Detection and Ranging which is a laser-based system that means detect objects and determine their range, angle, and/or velocity using laser beams.
  The LIDAR can measure distance up to 300 meters.



a b

Figure 4.1: Example for (a) Radar and (b) Lidar

- **Stereo Vision:**
  Stereovision means detecting objects and their position using two cameras.
  It can measure distance up to 40 meters.

Comparison:

Table 4.1: Depth estimator sensors comparison

	RADAR	LIDAR	Stereo Vision
Resolution	Bad	Fine	Good
Velocity	Good	Fine	Bad
All- Weather	Good	Bad	Bad
Size	Good	Bad	Good
Range	Good	Fine	Bad
Noise handle	Bad	Bad	Good
Cost	Good	Bad	Fine

In this project, we used Stereo Vision method.

1.3. Stereo Vision Method

As we mentioned before, this method is using two cameras to detect objects and calculate the distance.So, we have two images from the right and left camera. Then we match the imagespixels to create a depth map and from the camera specification we can get length.

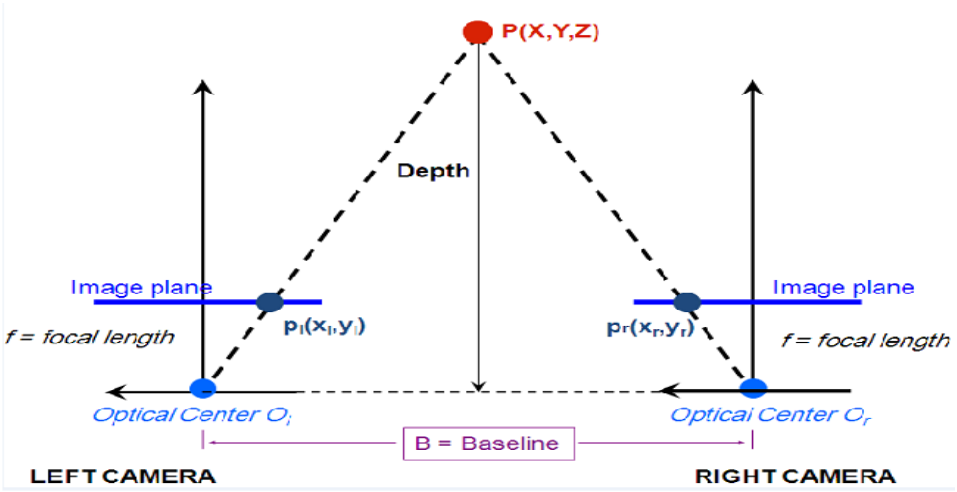


Figure 4.2: Stereo Vision

We get the distance value from the equation:

$$D = f * B/(disparity) \qquad (4.1)$$

D: Distance value

f: Camera focal length

B: Camera baseline

We used ZED Cam in our project as it is stereo vision-based camera.

2. ZED Camera

2.1. Overview

The ZED is a camera that reproduces the way human vision works. Using its two “eyes” and through triangulation, the ZED provides a three-dimensional understanding of the scene it observes, allowing your application to become space and motion aware.

Figure 4.3: ZED camera

The camera has several video modes available:

Table 4.2: Camera video modes

Video Mode	Output Resolution (side by side)	Frame Rate (fps)	Field of View
2.2K	4416x1242	15	Wide
1080p	3840x1080	30, 15	Wide
720p	2560x720	60, 30, 15	Extra Wide
WVGA	1344x376	100, 60, 30, 15	Extra Wide

Depth can be captured at longer ranges, up to 20m.Field of view is much larger, up to 110°.Base line equal to 12 cm.

2.2. ZED Camera Calibration

Stereolabs offers a program for calibration camera by trying to adjust the tilt angle of the camera to match the blue circle or ellipse with the red circle or ellipse.

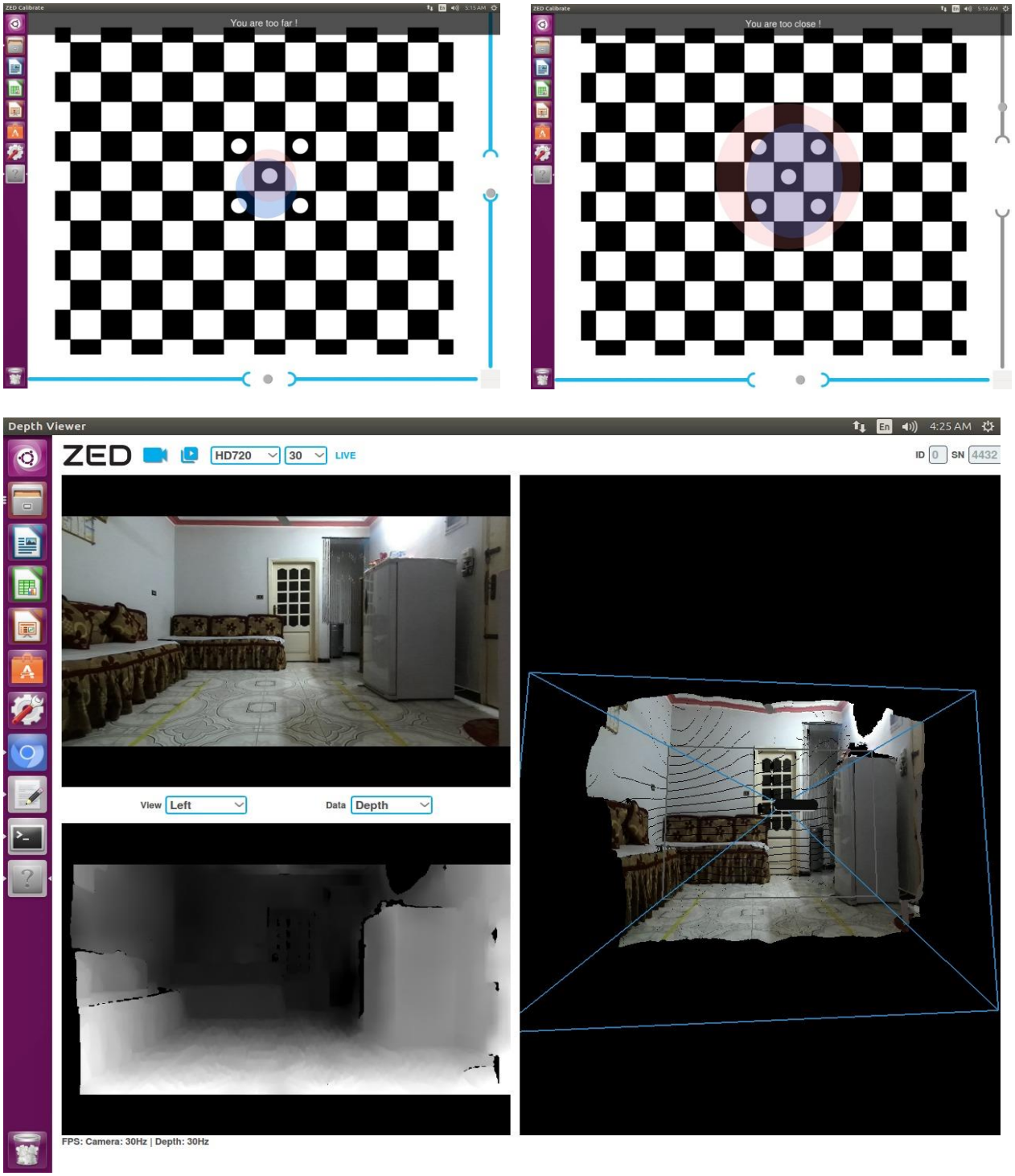


Figure 4.4: Zed camera calibration

ZED Camera Calibration of Stereolabs is very accurate and suitable for our system.

2.3. ZED Camera Algorithm

The Algorithm divided into five stages as showing in the below block diagram.

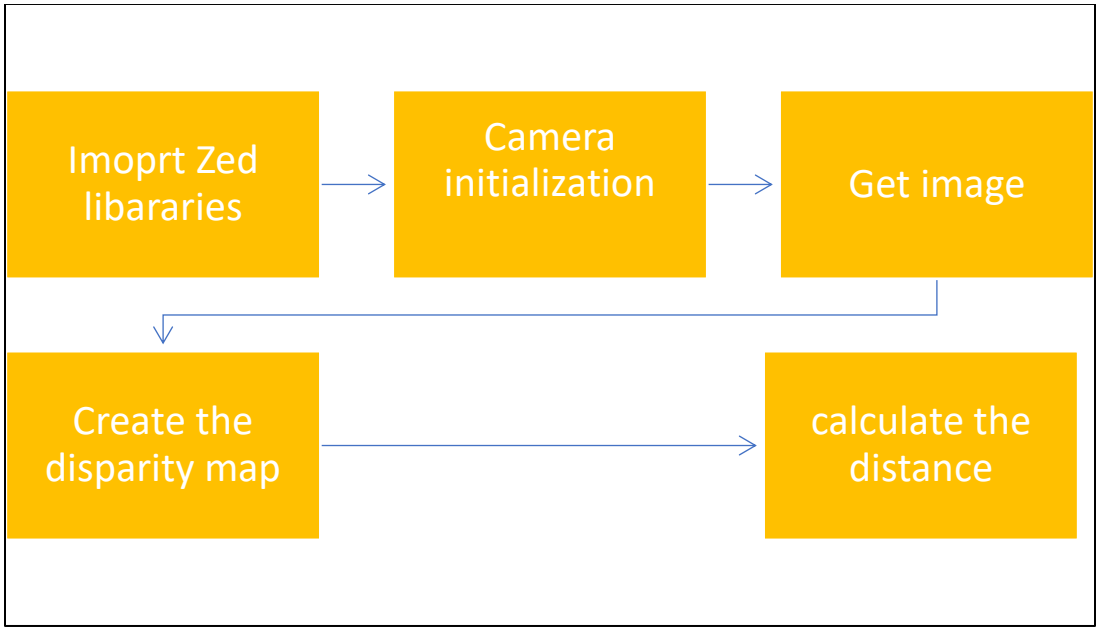


Figure 4.5: Zed camera algorithm stages

- **Import ZED libraries:**

```
import pyzed.camera as zcam
import pyzed.defines as sl
import pyzed.types as tp
import pyzed.core as core
import math
```

Code 4.1: Import ZED libraries

pyzed.camera: is used to Initiate the camera Parameters.
Sensing mode:

1. Standard Mode: the default mode in the camera. this mode preserves distance metrics and shapes. It is fast but contains holes due to visual occlusions and filtering.
2. Fill Mode: this mode provides a fully dense depth map with a Z value for every pixel (X, Y) in the left image

Coordinate units: Set the measure unit type (mm, cm, m)

pyzed.defines: is used to extract the camera.Defines like (Camera Parameters,Measure Units, Sensing Modes…….).

pyzed.types: Check if the camera is opened and there is no failure.
pyzed.core: It will contain the image and disparity data.

- **Camera Initiation:**

Wemust initiate the camera parameters to use it.

```
def zed_init():
dep.zed = zcam.PyZEDCamera()
init_params = zcam.PyInitParameters()
init_params.depth_mode = sl.PyDEPTH_MODE.PyDEPTH_MODE_PERFORMANCE
init_params.coordinate_units = sl.PyUNIT.PyUNIT_MILLIMETER
err = dep.zed.open(init_params)
if err != tp.PyERROR_CODE.PySUCCESS:
exit(1)
dep.runtime_parameters = zcam.PyRuntimeParameters()
dep.runtime_parameters.sensing_mode = sl.PySENSING_MODE.PySENSING_MODE_STANDARD
dep.image = core.PyMat()
dep.depth = core.PyMat()
dep.point_cloud = core.PyMat()
```

Code 4.2: Camera initialization

- **Get the image**

We get the image frame from the camera using the retrieve_image function which it stores the image data in the core lib. In four channel RGB and depth map in the last channel. So, we exclude the last channel from the copy which we get from the core lib. if we want to get the image.

Figure 4.6: Left and Right images

We choose from which camera we will take the image.

```
dep.zed.retrieve_image(dep.image, sl.PyVIEW.PyVIEW_LEFT)
dep.zed.retrieve_measure(dep.depth, sl.PyMEASURE.PyMEASURE_DEPTH)
frame=dep.image.get_data()
frame=cv2.cvtColor(frame,cv2.COLOR_BGR2RGB)
frame=cv2.cvtColor(frame,cv2.COLOR_RGB2BGR)
return frame
```

Code 4.3: Choose which camera

- **Depth map:**
  The ZED stores a distance value (Z) for each pixel (X, Y) in the image. The distance is expressed in metric units (meters for example) and calculated from the back of the left eye of the camera to the scene object.
  Depth maps captured by the ZED cannot be displayed directly as they are encoded on 32 bits. To display the depth map, a monochrome (grayscale) 8-bit representation is necessaryvalues between [0, 255], where 255 represents the closest possible depth value and 0 the most distant possible depth value.

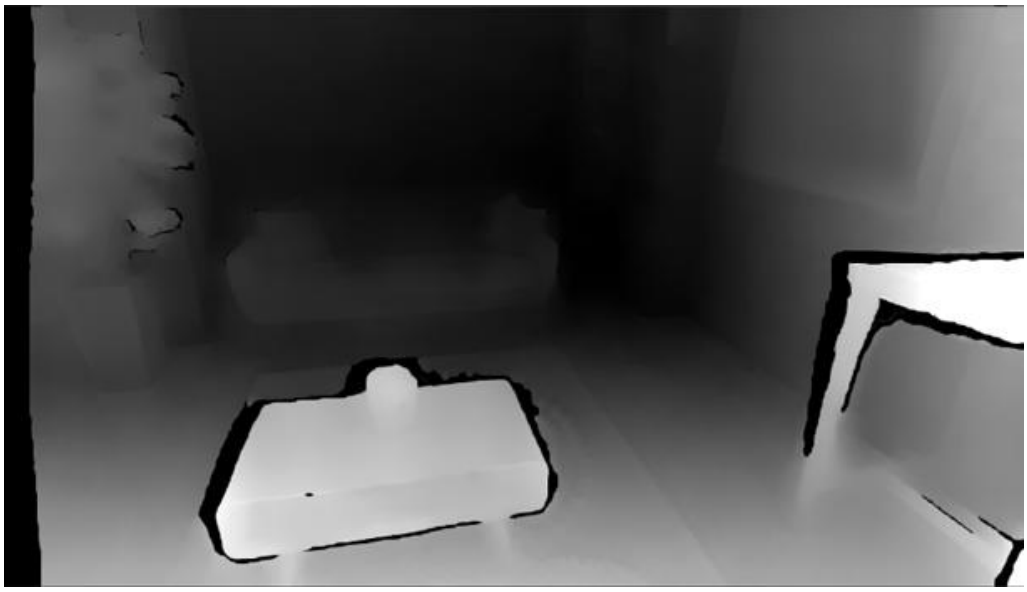


Figure 4.7: Depth map

We generate the depth map from the two images using PyMEASURE_DEPTH parameter then using the depth map we calculate the distance from the camera center for each pixel in the RGB channels using PyMEASURE_XYZRGBA

```
def disparity():
dep.zed.retrieve_measure(dep.depth,sl.PyMEASURE.PyMEASURE_
DEPTH)
dep.zed.retrieve_measure(dep.point_cloud,sl.PyMEASURE.PyME
ASURE_XYZRGBA)
```

Code 4.4: Disparity function

- **Calculate the distance:**
  To calculate the distance, we need to specify the pixel coordinates (x, y) then get the distance value in the three channel (RGB) so we create 3- point cloud which is Another common way of representing depth information i. A point cloud can be seen as a depth map in three dimensions. While a depth map only contains the distance or Z information for each pixel, a point cloud is a collection of 3D points (X,Y,Z) that represent the external surface of the scene and can contain color information.

To calculate the distance, we need to specify the pixel coordinates(x, y) then get the distance value in the three channel (RGB) so we create 3- point cloud which is Another common way of representing depth information i. A point cloud can be seen as a depth map in three dimensions. While a depth map only contains the distance or Z information for each pixel, a point cloud is a collection of 3D points (X,Y,Z) that represent the external surface of the scene and can contain color information.

Figure 4.8: Point cloud

```
def distance(x,y):
      err, point_cloud_value = dep.point_cloud.get_value(x, y)
      distance = math.sqrt(point_cloud_value[0] * point_cloud_value[0] + point_cloud_value[1] *
point_cloud_value[1] + point_cloud_value[2] * point_cloud_value[2])
      if not np.isnan(distance) and not np.isinf(distance):
            distance = distance/1000
            distance =round(distance,2)
            distance = str(distance)
            distance=distance+'meter'
          else:
              distance = "Can't estimate distance at this position, move the camera\n"
          return distance
```

Code 4.5: Distance function

3. Depth Estimation Testing

3.1. Goal

We must know the camera depth accuracy and its maximum range. so, we compared the camera results with the real results that we measured in different degrees of illumination.

The testing occurs in two cases (Sun Light and Room Light).

Figure 4.9: Sunlight

Figure 4.10: Room light

The maximum distance that the camera can measured is about 20 meters. with max error $\pm$1meter error. The camera can reach 30 metersif we decrease the matching threshold, but the max error will increase.

Figure 4.11: Maximum distance

3.2. Accuracy

In room light, the error between the measured values of depth estimation and the real values of depth estimation is very small.

Tested on a demo video,where error is almost equal $\pm 0.1m$.

Table 4.3: Real values and measured value

Object	Real value(meter)	Measured value(meter)
Person 1	4.6	4.6
Person 2	7.1	7
Board	8.6	8.7
Bottle	2.6	2.7
Screen 1	6.44	6.4
Screen 2	5.8	5.7
Screen 3	10.3	10.3

In the sun light “street”, the error between the measured values of depth estimation and the real values of depth estimation is0.5m to 1m almost.

CHAPTER 5

Vision-based lane detection System

1. Introduction

1.1. Overview

Among the various accidents which are happening nowadays, unintended lane departure is the leading cause that is risking lives of people. A fatal collision can occur within a split of second due to driver's inattention to the surrounding cars and drowsiness. To reduce the number of traffic accidents and to improve safety Lane Keeping assist systems are becoming more common in passenger vehicles. The system continuously monitors the position of the vehicle on the lane markers on either side. If the vehicle comes within a certain distance of a marker the system provide alert to the driver.

1.2. Goal

The purpose of the Vision-based lane detection System is monitoring the position of thevehicle on the lane markers on either side and warn the driver in case the vehicle comes within a certain distance of a marker.

2. Lane Detection Algorithm

The Algorithm divided into eight stages as showing in the below block diagram.

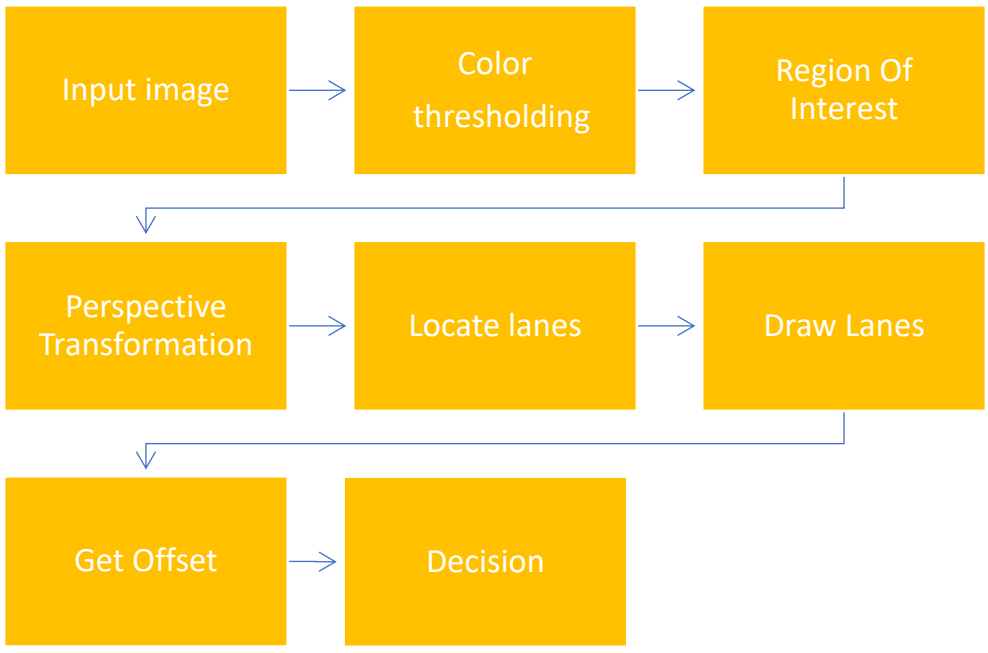


Figure 5.1: Lane detection algorithm stages

So, the goal is:

a b

Figure 5.2: Lane detection result such that (a) input image and (b) output image

- **Input Image**
  The algorithm takes an input image as first step from a camera which is usually located in the vehicle center (we may need to crop the vehicle hood).

  Cropping:

a b

Figure 5.3: Cropping result (a) After cropping and (b) Before cropping

```
cap=cv2.VideoCapture(0)                 // take input images from camera
while cap.isOpened() == True:
ret,img=cap.read()                      // read the image
img=img[100:640,200:1210,:]     // crop with that range
```

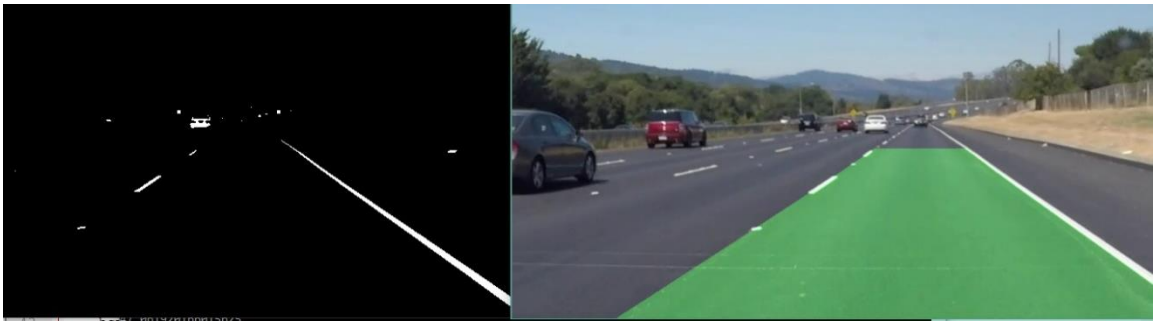
Code 5.1: Image cropping

- **Color Thresholding**
  In this step, we extract yellow and white colors which are the lanes color and set other colors to zero. We take an input RGB image and making two masks using HSV transformation for Yellow color and HSL for White color.

a b

Figure 5.4: Thresholding (a) the binary image and (b) the output image

We got the mask values from experiments on Udacity self-driving Nano degree course's videos.



a b

Figure 5.5: White lane thresholding (a) the binary image and (b) the output image

```
def white_thresh(frame,show=False):
h,w,c=frame.shape
      mask=np.zeros((h,w),np.uint8)
hsv=cv2.cvtColor(frame,cv2.COLOR_BGR2HLS)
low_white=np.array([0,210,0])
high_white=np.array([255,255,255])
ymask=cv2.inRange(hsv,low_white,high_white)
      output=cv2.bitwise_or(mask,ymask,mask=ymask)
      return output
```

Code 5.2: White thresholding

```
def binary(img,show=False):
    output1=white_thresh(img,False)                 // Get the white binary image
    output2=yellow_thresh(img,False)                // Get the yellow binary image
    output=cv2.bitwise_or(output1,output2)          //  make Oring operation to get final binary
    return output
```

Code 5.3: Binary function

```
def yellow_thresh(frame,show=False):
h,w,c=frame.shape// Get the image shape
mask=np.zeros((h,w),np.uint8)                       // Create the mask
hsv=cv2.cvtColor(frame,cv2.COLOR_BGR2HSV)  // convert to HSV plane
low_yellow=np.array([0,150,150])                    // create the low yellow thresholding
high_yellow=np.array([50,255,255])                  //create the high yellow thresholding
ymask=cv2.inRange(hsv,low_yellow,high_yellow)  // create the mask
output=cv2.bitwise_or(mask,ymask,mask=ymask) // create yellow binary output image
return output
```

Code 5.4: Yellow thresholding

- **Region of Interest**
  We take a region that it will probably contain the lanes marks. Which has a width greater than the lane width.
  This region will be the source for upcoming stage (Perspective transformation).

Figure 5.6: Region of interest

- **Perspective Transformation:**
  The perspective transform is used to obtain a bird's-eye view of lanes that lets us view a lane from above. This will help in locating lanes as we will see in the next stage (locate lanes).

a b

Figure 5.7: (a) Binary image (b) Perspective image

```
def perspective_view(img):
   h, w = img.shape[:2]
src=np.float32([[100,img.shape[0]-1],[400,340],[600,340], [1000,img.shape[0]-1]])
   dst=np.float32([[100,img.shape[0]-1],[100,0],[1100,0],[1000,img.shape[0]-1]])
   M = cv2.getPerspectiveTransform(src, dst)                    //Get the transformation matrix
Minv = cv2.getPerspectiveTransform(dst, src)                    //Get the inverse transformation
   warped = cv2.warpPerspective(img, M, (w, h), flags=cv2.INTER_LINEAR) //transform the img
   return warped,M,Minv
```

Code 5.5: Perspective view

- **Locate the lanes:**
  To detect the lanes from its points, we get two equations in the form of second polynomials (ax^2+bx+c) by sum each pixel in every column and the greater two columns will be the two centers of the two lines.

Steps:

1. Sum each pixel in every column.
2. Find max value before and after the mid. Column.
3. The max values will be the center of the two lines.
4. Determined number of boxes you will draw and its width.
5. Get nonzero pixel in the image.
6. Check if the nonzero pixel located in the boxes and append them in a list.
7. Get the two equations from the appended pixels.
8. Update the current values.

Figure 5.8: Perspective image

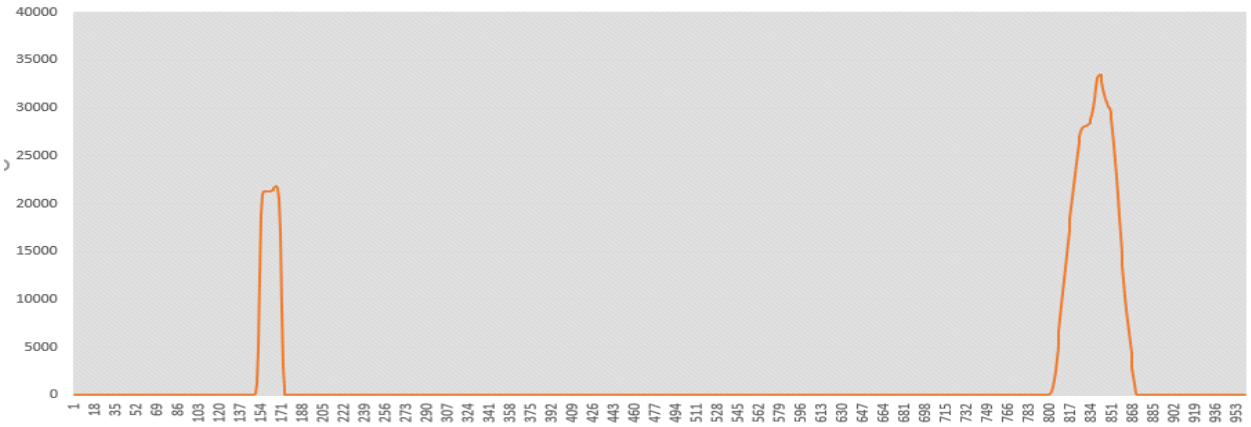


Figure 5.9: Columns summation

Figure 5.10: Find lane

```
def find_lane(warped,im,frame):
h,w=warped.shape
nonzero=warped.nonzero()
nonzero_y = np.array(nonzero[0])
nonzero_x = np.array(nonzero[1])
margin=100
if  dep.first_time==1:
num_windows=9
sumation=np.sum(warped[int(h/2):,:],axis=0)
midpoint=int(h/2)
win_height=int(h/num_windows)
left_x_base=np.argmax(sumation[:midpoint])
right_x_base=np.argmax(sumation[midpoint:])+midpoint
current_left_x=left_x_base
current_right_x=right_x_base
```

Code 5.6: Find lane function (1)

```
minpix=50
left_point_ind=[]
right_point_ind=[]
color_warp = np.dstack((warped,warped,warped))
for window in range(num_windows):
win_y_low=h-(window+1)*win_height
win_y_high=win_y_low+win_height
x_left_low=current_left_x-margin
x_left_high=current_left_x+margin
x_right_low=current_right_x-margin
x_right_high=current_right_x+margin
good_left_inds=((win_y_low<nonzero_y)
&(win_y_high>nonzero_y)&(x_left_low<nonzero_x)&(x_left_high>nonzero_x)).nonzero()[0]
good_right_inds = ((nonzero_y>= win_y_low) & (nonzero_y<win_y_high) & (nonzero_x>=
x_right_low) & (nonzero_x<x_right_high)).nonzero()[0]
left_point_ind.append(good_left_inds)
right_point_ind.append(good_right_inds)
if len(good_left_inds) >minpix:
current_left_x = np.int(np.mean(nonzero_x[good_left_inds]))
if len(good_right_inds) >minpix:
current_right_x = np.int(np.mean(nonzero_x[good_right_inds]))
left_point_ind = np.concatenate(left_point_ind)
right_point_ind = np.concatenate(right_point_ind)
leftx = nonzero_x[left_point_ind]
lefty = nonzero_y[left_point_ind]
rightx = nonzero_x[right_point_ind]
righty = nonzero_y[right_point_ind]
```

Code 5.7: Find lane function (2)

```
if (lefty.size!=0)&(leftx.size!=0):
dep.left_fit = np.polyfit(lefty, leftx, 2)
else:
dep.left_fit=dep.prev_left
if (righty.size!=0)&(rightx.size!=0):
dep.right_fit = np.polyfit(righty, rightx, 2)
else:
dep.right_fit=dep.prev_right
if dep.first_time==1:
dep.first_time=0
dep.prev_left=dep.left_fit
dep.prev_right=dep.right_fit
```

Code 5.8: Find lane function (3)

We donot need to calculate those calculation in every frame as the lane position will not change a lot in the two frames, so we can use the equation which we got from previous frame to locate the new lane equation.

```
else:
good_left_inds = ((nonzero_x> (dep.left_fit[0]*(nonzero_y**2) + dep.left_fit[1]*nonzero_y + dep.left_fit[2] - margin)) & (nonzero_x< (dep.left_fit[0]*(nonzero_y**2) + dep.left_fit[1]*nonzero_y + dep.left_fit[2] + margin)))
good_right_inds = ((nonzero_x> (dep.right_fit[0]*(nonzero_y**2) + dep.right_fit[1]*nonzero_y + dep.right_fit[2] - margin)) & (nonzero_x< (dep.right_fit[0]*(nonzero_y**2) + dep.right_fit[1]*nonzero_y + dep.right_fit[2] + margin)))
leftx = nonzero_x[good_left_inds]
lefty = nonzero_y[good_left_inds]
rightx = nonzero_x[good_right_inds]
righty = nonzero_y[good_right_inds]
```

Code 5.9: Find lane function (4)

```
# Fit a second order polynomial to each
if (lefty.size!=0)&(leftx.size!=0):
dep.left_fit = np.polyfit(lefty, leftx, 2)
else:
dep.left_fit=dep.prev_left
if (righty.size!=0)&(rightx.size!=0):
dep.right_fit = np.polyfit(righty, rightx, 2)
else:
dep.right_fit=dep.prev_right
dep.left_fit= (dep.left_fit+dep.prev_left+dep.prev_prev_left)/3
dep.right_fit=(dep.right_fit+dep.prev_right+dep.prev_prev_right)/3
dep.prev_prev_right=dep.prev_right
dep.prev_right=dep.right_fit
dep.prev_prev_left=dep.prev_left
dep.prev_left=dep.left_fit
```

Code 5.10: Find lane function (5)

- **Draw Lanes:**
  Now we have the left & right fitting coefficients from the last step, so we substitute with numbers from 0 to image's height - 1 and gets its x value. Then we draw the detected lanes back to the input image but before we must do an inverse perspective transform to return to the original image not the bird eye view.

```
def draw(warped,frame,im):
h,w=warped.shape
ploty = np.linspace(0, warped.shape[0]-1, warped.shape[0] )
left_fitx = dep.left_fit[0]*ploty**2 + dep.left_fit[1]*ploty + dep.left_fit[2]
right_fitx = dep.right_fit[0]*ploty**2 + dep.right_fit[1]*ploty + dep.right_fit[2]
warp_zero = np.zeros_like(warped).astype(np.uint8)
color_warp = np.dstack((warp_zero, warp_zero, warp_zero))
pts_left = np.array([np.transpose(np.vstack([left_fitx, ploty]))])
pts_right = np.array([np.flipud(np.transpose(np.vstack([right_fitx, ploty])))])
 pts = np.hstack((pts_left, pts_right))
 cv2.fillPoly(color_warp, np.int_([pts]), (0,255, 0))
newwarp = cv2.warpPerspective(color_warp, im, (w, h))
 result = cv2.addWeighted(frame ,1, newwarp, 0.3, 0)
```

Code 5.11: Draw lanes.

Figure 5.11: Draw lanes

- **Offset:**
  Now we can calculate the offset between the center of the image (assume it will be also the car center) and the lane width center.

```
distance_px=(float(np.max(np.where(newwarp[-1,:,1]==255))+np.min(np.where(newwarp[-1,:,1]==255)))/2)-(frame.shape[1]/2)
if distance_px>0:
   direction='left'
elifdistance_px<0:
   direction='right'
 else:
   direction='center'
font = cv2.FONT_HERSHEY_SIMPLEX
result=cv2.putText(result,'Vehicle is {:.2f}pixels '.format(abs(distance_px))+direction+' of center',(10,150), font, 2,(255,255,255),2,cv2.LINE_AA)
return result
```

Code 5.12: compute the offset.

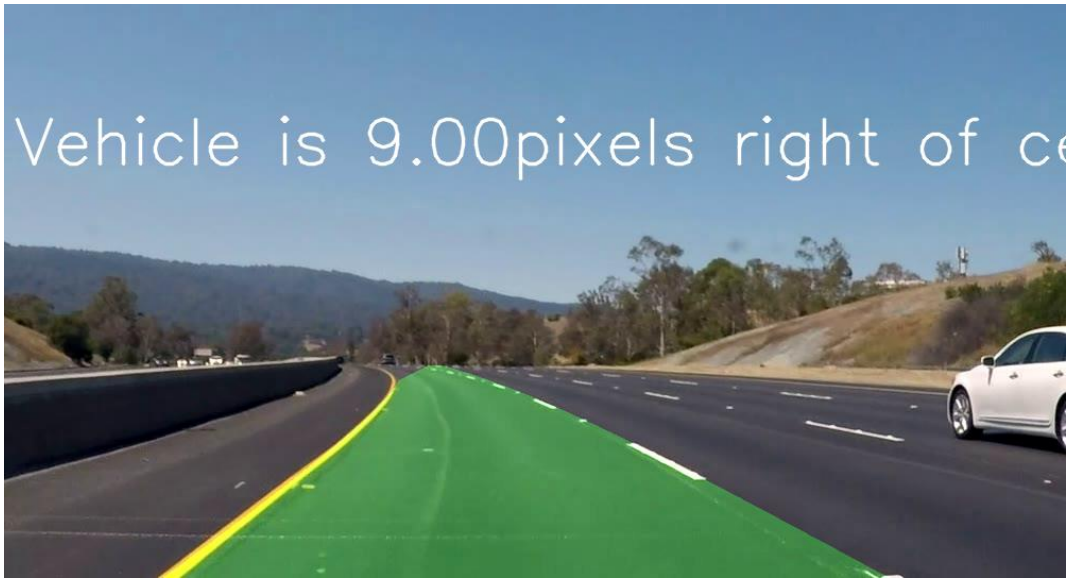


Figure 5.12: Output image

You can get the offset in meter by measure the lane width and divided it with the lane width in pixels then you will get the value of meter per pixel.

- **Decision:**
  Once the calculated car position is attempts to cross one of the lanes system alerts the driver.

CHAPTER 6

Modules Integration

1. Introduction

1.1. Goal

The purpose of the integration is monitoring the position of the objects relative to our vehicle and the lane marks.

If there is an object inside the lane, the lane color becomes red and warn the driver and the same action will be taken in case any object comes within a certain distance.

1.2. Flow of Integration Algorithms Modules

1. Initiate the zed camera parameters.
2. Initiate the object detection parameters.
3. Initiate the lane parameters.
4. Take the image and make a copy.
5. Create the disparity map.
6. Detect the objects from the image and store the objects location in 2 lists.
7. Get the object distance.
8. Set the object location with zero (remove) in the copy image.
9. Get the lane marks from the copy image.
10. Find if any point from the lane marks exist in the detected box.
11. Set the lane color.
12. Warn the driver if the object distance in the range you specify orin lane.

a b

Figure 6.1: Object and lane integration (a) input image and (b) output image

```
if len(founds)>=5:
result=drawc.draw(warped,im,frame,(0,0,255))
else:
result=drawc.draw(warped,im,frame,(0,255,0))
if len(tls) == 0:
result=drawc.draw(warped,im,frame,(0,255,0))
tock=time.time()
cv2.imshow("1",result)
if cv2.waitKey(30)==ord('q'):
cv2.imwrite("h.png",frame)
cv2.destroyAllWindows()
break
t=(tock-teck)*1000
print(t)
cv2.destroyAllWindows()

while (cap.isOpened()):
stime = time.time()
ret, frame = cap.read()
newframe = frame.copy()
if ret:
results = tfnet.return_predict(frame)
for color, result in zip(colors, results):
tl = (result['topleft']['x'], result['topleft']['y'])
br = (result['bottomright']['x'], result['bottomright']['y'])
label = result['label']
frame = cv2.rectangle(frame, tl, br, color, 7)
frame = cv2.putText(frame, label, tl, cv2.FONT_HERSHEY_COMPLEX, 1, (0, 0, 0), 2)
newframe[tl[1]:br[1],tl[0]:br[0],:] = 0
tls.append(tl)
brs.append(br)
```

```
teck=time.time()
binary=b.binary(newframe)
warped,m,im=p.perspective_view(binary)
fl.find_lane(warped,im,frame)
newwarped=draw.drawbin(warped,frame,im)
nonzero=newwarped.nonzero()
nonzero_y = np.array(nonzero[0])
nonzero_x = np.array(nonzero[1])
for i in range(len(tls)):
founds=((tls[i][1]<nonzero_y)
&(brs[i][1]>nonzero_y)&(tls[i][0]<nonzero_x)&(brs[i][0]>nonzero_x)).nonzero()[0]
```

Code 6.1: lane and object integration.

1.3. Output Scenarios

When the object isnot inside The Lane, the lane color becomes green.

Figure 6.2: No object on the lane

When the object is inside The Lane the lane color become red.

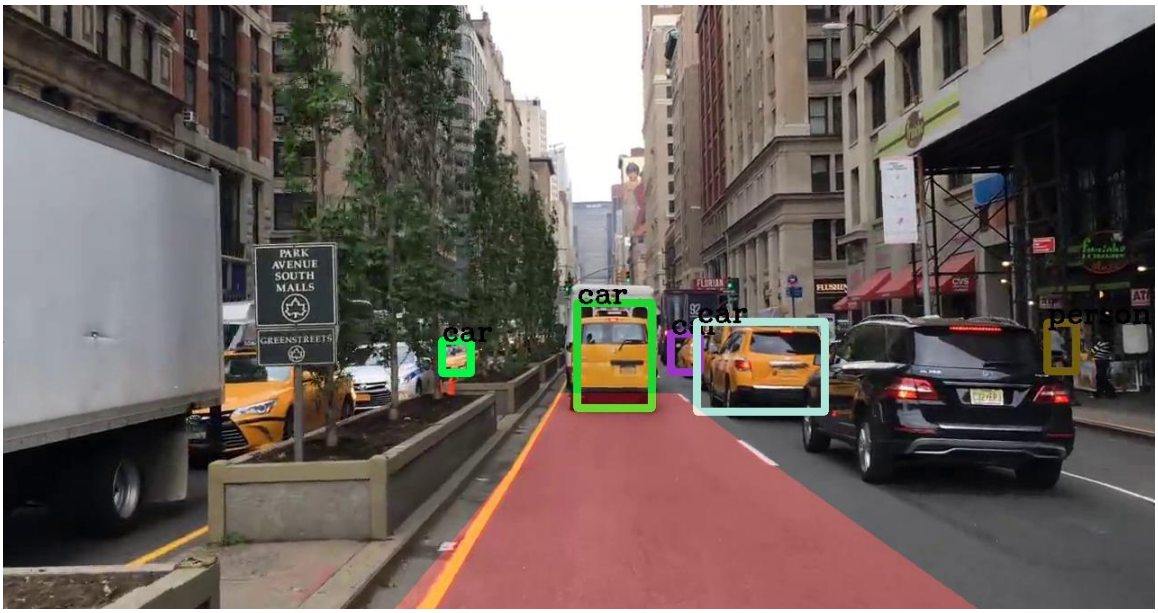


Figure 6.3: Object in the lane

1.4. Linking Integration Algorithm with TIVA Board

Our target is making a system to alert the driver when there is any danger.so, we must link the integration algorithm with an embedded board and make sure that the board receives the correct data so, we created a simple system according to the two cases above (object in lane or out the lane).

- **Case1: object is not in lane:**
  An RGB led colors take the same lane color which is green.

Figure 6.4: No object in the lane (TIVA implementation)

- **Case2: object is in lane**
  The led color turns to red and the buzzers start to make its sound.

Figure 6.5: Object in the lane (TIVA implementation)

1.5. Number of Frames per Second on Jetson TX2 for integration modules:

- Yolo lane 6 frame/s
- Yolo depth 5 frames/s
- Yolo depth lane 4.1 frames/s

CHAPTER 7

Behavioral Cloning
End-to-End Deep Learning for Self-Driving Cars

1. Introduction

In a new automotive application, we have used convolutional neural networks (CNNs) to map the raw pixels from a front-facing camera to the steering commands for a self-driving car. This powerful end-to-end approach means that with minimum training data from humans, the system learns to steer, with or without lane markings, on both local roads and highways. The system can also operate in areas with unclear visual guidance such as parking lots or unpaved roads.

2. Abstract:

We trained a convolutional neural network (CNN) to map raw pixels from a single front-facing camera directly to steering commands. This end-to-end approach proved surprisingly powerful. With minimum training data from humans the system learns to drive in traffic on local roads with or without lane markings and on highways. It also operates in areas with unclear visual guidance such as in parking lots and on unpaved roads. The system automatically learns internal representations of the necessary processing steps such as detecting useful road features with only the human steering angle as the training signal. We never explicitly trained it to detect, for example, the out-line of roads. Compared to explicit decomposition of the problem, such as lane marking detection, path planning, and control, our end-to-end system optimizes all processing steps simultaneously. We argue that this will eventually lead to better performance and smaller systems. Better performance will result because the internal components self-optimize to maximize overall system performance, instead of optimizing human-selected intermediate criteria, e. g., lane detection. Such criteria understandably are selected for ease of human interpretation which doesn’t automatically guarantee maximum system performance. Smaller networks are possible because the system learns to solve the problem with the minimal number of processing steps.

We used a Google Colab for training and a NVIDIA JetsonTX2 for determining where to drive. The system operates at 30 frames per second (FPS).

3. System Diagram

Figure 7.1 shows a simplified block diagram of the collection system for training data for Jetson. ZED camera is mounted behind the windshield of the data-acquisition car.

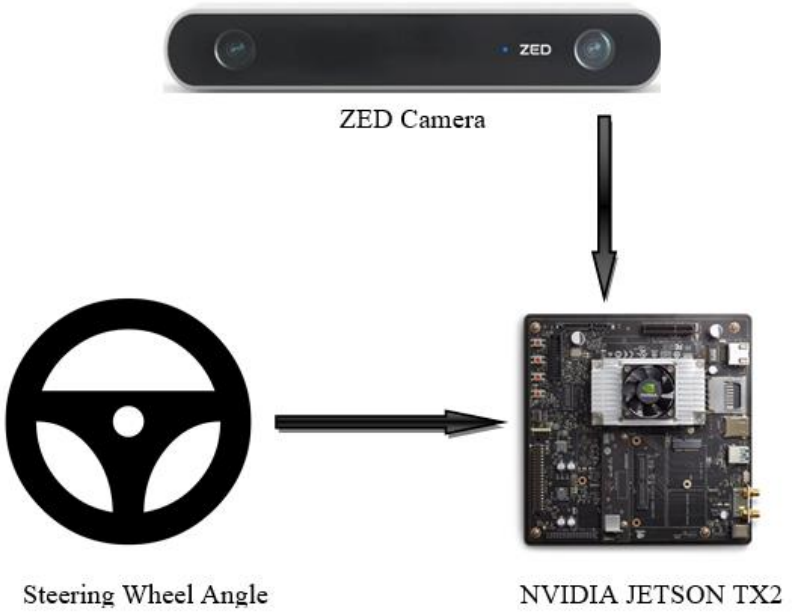


Figure 7.1: High-level view of the data collection system.

A block diagram of our training system is shown in Figure 7.2. Images are fed into a CNN which then computes a proposed steering command. The proposed command is compared to the desired command for that image and the weights of the CNN are adjusted to bring the CNN output closer to the desired output. The weight adjustment is accomplished using back propagation

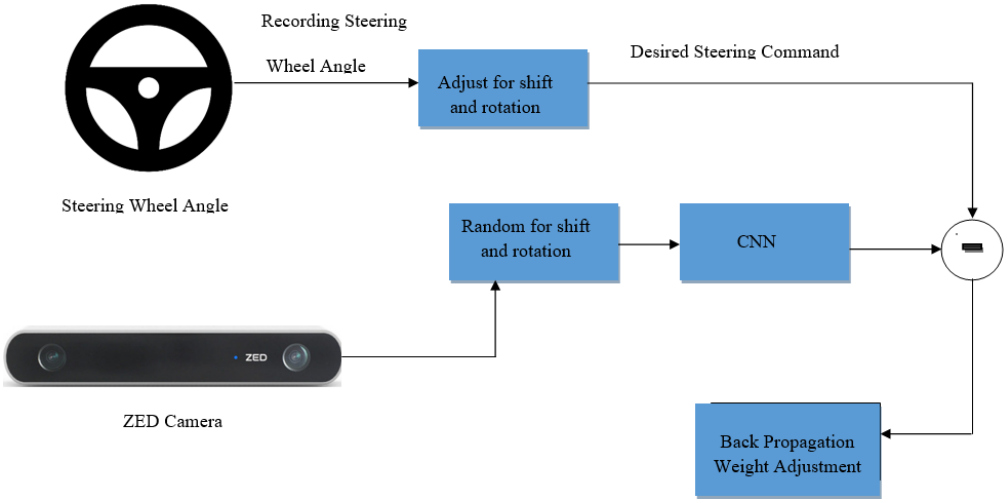


Figure7.2: Training the neural network

Once trained, the network can generate steering from the video images of a single center camera. This configuration is shown in Figure 7.3.

Figure 7.3: The trained network is used to generate steering commands from camera.

4. Data Collection

Training data was collected by driving on a wide variety of roads and in a diverse set of lighting and weather conditions. Most road data were collected in home or Faculty. Data was collected in clear, cloudy, foggy, snowy, and rainy weather, both day and night. In some instances, the sun was low in the sky, resulting in glare reflecting from the road surface and scattering from the windshield. It takes about 12 hours of driving to collect data.

5. Network Architecture:

We train the weights of our network to minimize the mean squared error between the steering command output by the network and the command of either the human driver, or the adjusted steering command for off-centre and rotated images. Our network architecture is shown in Figure 4. The network consists of 9 layers, including a normalization layer, 5 convolutional layers and 3 fully connected layers. The input image is split into YUV planes and passed to the network.

The first layer of the network performs image normalization. The normalizer is hard-coded and is not adjusted in the learning process. Performing normalization in the network allows the normalization scheme to be altered with the network architecture and to be accelerated via GPU processing.

The convolutional layers were designed to perform feature extraction and were chosen empirically through a series of experiments that varied layer configurations. We use strided convolutions in the first three convolutional layers with a 2×2 stride and a 5×5 kernel and a non-strided convolution with a 3×3 kernel size in the last two convolutional layers.

We follow the five convolutional layers with three fully connected layers leading to an output control value which is the inverse turning radius. The fully connected layers are designed to function as a controller for steering, but we note that by training the system end-to-end, it is not possible to make a clean break between which parts of the network function primarily as feature extractor and which serve as controller.

6. Data Selection

The first step to training a neural network is selecting the frames to use. Our collected data is labelled with road type, weather condition, and the driver's activity (staying in a lane, switching lanes, turning, and so forth). To train a CNN to do lane following, we simply select data where the driver is staying in a lane and discard the rest. We then sample that video at 10 FPS because a higher sampling rate would include images that are highly similar, and thus not provide much additional useful information.

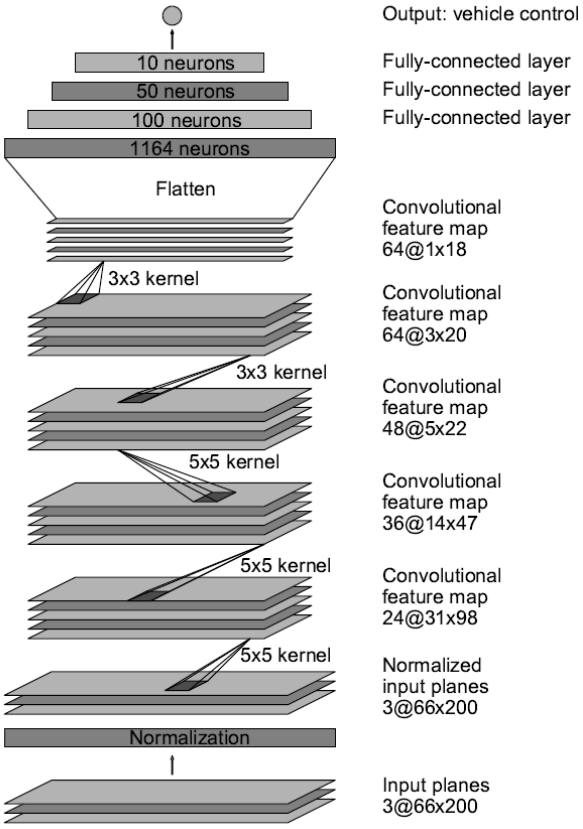


Figure 7.4: CNN architecture. The network has about 27 million connections and 250 thousand

To remove a bias towards driving straight the training data includes a higher proportion of frames that represent road curves.

7. Augmentation

After selecting the final set of frames, we augment the data by adding artificial shifts and rotations to teach the network how to recover from a poor position or orientation. The magnitude of these perturbations is chosen randomly from a normal distribution. The distribution has zero mean, and the standard deviation is twice the standard deviation that we measured with human drivers. Artificially augmenting the data does add undesirable artefacts as the magnitude increases (as mentioned previously).