Institute for Integrated Signal Processing Systems (ISS)
RWTH Aachen University
Prof. Dr.-Ing. Gerd Ascheid

Master Thesis M107

Deep Learning for Autonomous Driving using End-to-End learning and Data Fusion

by

Deepak Ramani

Matr.-No. 314726

September, 2020

Supervisors:
Prof. Dr.-Ing. Gerd Ascheid
Shawan Taha Mohammed

This document is for internal use only. All copyrights are controlled by the supervising chair.
Publications of any kind are only authorized with permission of the chair.

I assure that this project was accomplished by me, without any foreign assistance except the official support of the chair. The used literature is fully indicated in the bibliography.

Aachen, Sep 30, 2020

(Deepak Ramani)

FIXME:

Placeholder

Please replace this sheet by the official thesis title/task document signed by the professor and you.

Contents

1 Introduction

The last decade has seen massive growth in the field of Autonomous Driving, primarily due to proliferation of graphical processing unit(GPU), and several projects like Google(Waymo) [1], Berkeley-DeepDrive [2], Apollo [3], making their datasets open-source which have made it easier for people to work on these data and achieve better performance gains.

Training a deep neural network(DNN) forms the core of making a car autonomous. By using supervised learning, one can achieve reliable results as it gives greater control at each stage of training. The data-driven approach collects data in advance and labels it appropriately. It can then be fed to the DNN using supervised learning algorithms to train the best model possible.

Ever since the discovery of Alexnet in 2012 [4], the convolutional neural network(CNN) and deep learning(DL) are preferred choices to analyse images. However, it is well known that the camera sensors are susceptible even to a slight change in weather conditions. Sensors like radar [5], LIDAR [6], ultrasonic[7], depth camera give additional depth information for obstacle detection. These features are then fused with the camera images and this process is called data fusion.

Even though there are some public data available, it is still not enough to reliably train a DNN. Then there is the cost of building an autonomous car. Fortunately, the last years have seen growth in reliable simulators which helped massively to collect data to help explore this field of research. To name a few simulators that are being actively used – LGSVL [8], Nvidia Drive [9], Carla [10], CarMaker [11]. In this thesis, the LGSVL simulator is used.

Figure 1.1: LGSVL[8] simulator active with all sensors

The LGSVL simulator allows the use of different sensors with minimal effort. The data from different sensors are published through websocket. So to capture these data, we need an interface/protocol

which can understand the sent data's message type and enable the receiving node to store them. However, the data from each sensor arrives at different rates. Hence it is necessary to collect and synchronise them in the order of their arrival before storing, so as to not lose their integrity and thereby prevent corrupting the dataset. Robotic operating system(ROS) [12] and its functionalities fulfil this purpose. It allows seamless transfer of simulator's data by subscribing to sensor nodes in the form of topics. Then the subscribing node with the help of ROS libraries, synchronises it as necessary for storage.

So, the data that resembles real-world is stored locally for later analysis and research.

1.1 Motivation

The motivation for this thesis is to use a simulator, do the required tests, and determine whether using a simulator does indeed help in perceiving the environment and accomplishing the goal of driving in the real-world.

One of the major obstacles in autonomous driving is the cost associated with integrating sensors in addition to manufacturing a vehicle. Representing the environment around the vehicle(ego vehicle) requires information from all in-car sensors. The resources demanded to make an optimal decision are also a challenge.

The high cost of associated sensors such as LIDAR[13], has put off many smaller research groups from using them in their work. Simulation allows conducting adequate tests(at a low cost) and quicker development of algorithms. Simulator provides a safer environment to test and debug these algorithms.

So with the help of a simulator it is observed how different constellation of sensors work, how different modalities interact with each other, and what impact these factors have on the overall performance of the deep neural networks.

Finally, an end-to-end system is implemented which simulates real-world behaviour and gives results which can then be applied to future research to make it more robust.

1.2 Goal

The desired goals of this thesis are listed below:

1. Building a basic autonomous driving framework which comprises of the following three components:
    - ROS - use ROS2 to synchronise the data received from the simulator and its plugin through a rosbridge, use functionalities such as slop and cache, to sort the data according to their received time in order not to scramble the information. During the evaluation, use the same functionalities to send command controls back to the simulator.
    - Rosbridge - use a bridge transport protocol that connects the ros to the simulator.
    - Docker - set up a work environment that is independent of hardware or operating system which allows easy running of the commands for data collection and evaluation.
2. Implement a system that can efficiently collect and label data.
3. Implement an end-to-end neural network architecture which applies state of the art deep learning techniques to learn driving by predicting the steering and other control commands from image pixels.
4. Implement and analyse different constellation of sensors with different data fusion techniques.

1.3 Related Work

In 2012, Alexnet [4] used CNNs to do object classification which, then in Computer Vision became the dominated approach for classification. Both Chen *et al.* [14] and Bojarski *et al.* [15] extended [4]'s approach of using CNN and showed that in addition to classification, CNN can extract features from images. Then they went on to demonstrate through an end-to-end network(which self-optimises itself based on its inputs), that steering angles can be predicted to keep the car in the lane of a road.

In a different field, but using CNN, Sergey Levine *et al.* [16] in 2016 corroborated that it was indeed possible to extract features with CNN and predict motor control actions in *object picking robots*.

Then, Xu *et al.* [17] in the same year with CNN-LSTM architecture showed that using the previous ego-motion events helped predict future ego-motion events. Using CNNs in an end-to-end architecture raised some questions on how it reached its decisions. So in 2017, both [18], [19] did visual analysis after the CNN layers to better understand the module's functionality. Vehicle control is more than just steering control. For smoother control, acceleration and braking are necessary besides steering. Both acceleration and deacceleration are dependent on the user's driving style, lane speed limit and traffic etc. Yand *et al.* [20] used CNN-LSTM architecture and provided the LSTM with feedback speed to determine the velocity of the ego vehicle.

Besides vehicle control, perceiving the environment is necessary for collision avoidance. The RGB colour camera sensors don't provide the depth information which is critical for collision avoidance. Hence, it is essential to fuse other sensors with diverse modalities with RGB to predict an optimal output. Liu *et al.* [21] provided rules in fusing data. They said that it was essential to pick out only vital information and discard other noisy data. They also described the techniques involved in data fusion – early/late fusion, hybrid fusion, model ensemble and joint training. Park *et al.* [22] gave us methods to enhance the features by using feature amplification or multiplicative fusion. Zhou *et al.* [23] detailed how fusing data into CNN affects the overall performance.

Even though the fused dataset gives a performance boost, it performs worse compared to individual modality. The combined fused model overfits more than its counterparts. The fundamental drawback of *gradient descent* in backpropagation causes the networks to overfit. This paper [24] introduced a technique called *gradient blending* to counteract this problem.

Xiao *et al.*[25] applied all the fusion techniques mentioned above with an imitation based end-to-end network[26]. They concluded that RGB images with depth information(obtained through a different modality) could indeed result in better performing end-to-end network model.

1.4 Contribution

In this thesis, it is shown that an end-to-end neural network model for autonomous driving using LGSVL simulator can be implemented. In addition, famous data fusion techniques can be applied and improvement in performance is observed.

2 Fundamentals

2.1 Machine learning: What and why?

Machine learning is all about learning from data and gaining knowledge from it. Machine learning was initially thought of as automating redundant human tasks and later developed into something that allowed solving complex mathematical problems. It was seen just an addition to humans than extension of them. Machine learning these days are required to perform tasks that are quite obvious and natural to humans such as recognising faces in images or perceiving the road environment around the vehicle and making decisions instinctively.

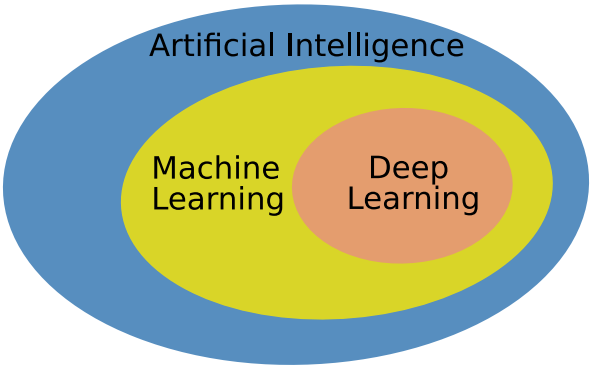

Figure 2.1: Schema of AI, ML and DL

All these attributes require to extend the field of machine learning.The figure 2.1 shows how artificial intelligence(AI) which was just a robot with simple if-else conditions, paved way for a subset in Machine learning(ML) and ML in turn getting narrower focus to result in another subset in Deep learning(DL).

So, in this chapter, a brief overview is given on the concepts that are implemented in the later chapters.

2.1.1 Learning algorithms

Machine learning provides a means to tackle tasks that are complex to solve through fixed programmes and designed by human beings [27]. A learning algorithm is an algorithm which gains the ability to learn from data. A ML algorithm is one that gains the ability to learn from an experience E with respect to some class of tasks T and performance measure P [28]. With experience, the algorithm can improve its performance.

Tasks T

The two major tasks in ML are *classification* and *regression*.

In classification related tasks, the system identifies which of k categories an input belongs to.

A function $f : \mathbb{R}^n \rightarrow \{1, \ldots, k\}$ is used by the learning algorithm to solve this task. When $y = f(x)$, the model assigns an input described by vector $\mathbf{x}$ to a category identified by numeric code y. There are other variants of the classification task, for example, where f outputs a probability distribution over classes [29]. Alexnet [4] is one of the examples of classification task that used it to do object recognition.

Regression predicts continuous value output and at any given time for an appropriate input to the neural network, regression will output a value corresponding to it.

A function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ predicts a numerical value for some input. Predicting the steering control value is a prime example for a regression task.

There are of course other tasks but only classification and regression are used in this thesis. Hence the narrow focus.

Performance measure P

To evaluate the performance of a ML algorithm, it is a must to design quantitative measure of its performance. Usually this performance measure P is specific to a task T. There are two distinct types of measurements – accuracy and error rate.

If the goal is to learn a mapping from inputs x to outputs y, where $y \in \{1, \ldots, C\}$, with C being the number of classes. If $C = 2$, this is called binary classification (in which case we often assume $y \in \{0, 1\}$); if $C > 2$, this is called multi class classification. If the class labels are not mutually exclusive (e.g., somebody may be classified as tall and strong), we call it multi-label classification [30].

Accuracy is just the proportion of examples for which the model produces the correct output. So in the case of binary classification, if the function f predicts a probability densities $\hat{y} \in \{0.3, 0.7\}$, for a ground truth y of value 1, then P is 70% accurate or the error rate is 30%.

It is essential that the model is evaluated with a data that it has not seen before. This data *testing set*, gives a good judgement on the performance of the trained model.

Experience E

The ML algorithms can be classified into *supervised*, *unsupervised* and *reinforcement* learning based on the kind of experience they are allowed to have. A learning algorithm is allowed to gain experience by going through the *dataset*. A dataset is collection of all the examples for a given task. For example, to classify which category a shown image belongs to has collection of images as dataset [31]. Sometimes datasets are also called as *data points*.

The focus will be on supervised learning in our case. A random vector $\mathbf{x}$ explicitly attempts to learn the probability distribution $p(\mathbf{x})$ and predicts $\mathbf{y}$ from $\mathbf{x}$, usually estimating $p(\mathbf{y} \mid \mathbf{x})$. The CIFAR dataset [31], for example, contain images as features which inturn have *targets* or *labels* associated with it. Here supervised learning(SL), the target functionality(labels) is known. So it uses the images and predicts the probability distribution to classify the images in the corresponding label.

2.2 Deep Learning

Deep learning is a subset of machine learning. It takes all the algorithms, concepts from machine learning, and narrows the focus to enable a model to learn from data such that tasks involve less human involvement, huge amount of data, and parameters.

2.2.1 Simple neural network

Linear regression is one of the common SL algorithms. It solves the regression problem. For example, if there is vector $\mathbf{x} \in \mathbb{R}^n$ as input and predict a scalar value $y \in \mathbb{R}$ as its output, then in linear

regression, output is a linear function of the input. We can define it as

$$\hat{y} = \mathbf{w}^T \mathbf{x} \tag{2.1}$$

where $\mathbf{w} \in \mathbb{R}^n$ is a vector of parameters.

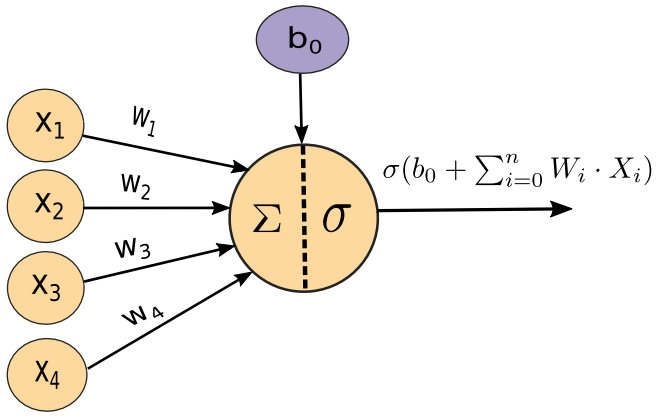

Figure 2.2: A simple neutral network

$\mathbf{w}$ is usually referred to as a set of weights that determine how each feature affects the prediction. A $\mathbf{w}_i$ is simply multiplied with a feature x_i to predict $\hat{y}$. By manipulating the $\mathbf{w}_i$ value, the corresponding feature x_i has an effect on the prediction $\hat{y}$.

A learning algorithm, in this case linear regression, is implemented as a perceptron. It is a single-layer neural network as first suggested by Rosenblatt in 1958. They generally consist of four main parts – input nodes x_i, weights w_i, bias b_0(if necessary), net sum Σ and an activation function σ. This is shown in the figure 2.2.

2.2.2 Activation function

The common activation functions used are Rectified Linear unit(ReLu), Sigmoid, tanh and softmax function. For each type of activation, σ then decides if the input received is relevant or not relevant. To convert linear inputs to non-linear, all that has to be done is to use a non-linear activation function. The figure 2.3, shows the characteristics of some of the activation functions.

For classification tasks, usually the last layer of the networks is equipped with softmax activation layer. This function normalises the output to a probability distribution over predicted output classes.

2.2.3 Multilayer feedforward networks

Deep feedforward networks or multilayer perceptrons are the quintessential deep learning models. Its goal is to approximate function f^*. In the below figure 2.4, information flows from inputs $\mathbf{x}$ to output y using a mapping function $\mathbf{y} = f(\mathbf{x}; \theta)$ where θ are the parameters values which the MLP learns for optimal approximation.

They are called feedforward as there are no feedback connections in which outputs of the model are fed back into itself. Feedforward networks with feedbacks are called *recurrent neutral networks*.

Feedforward networks form the core for many commercial applications. For example, the convolutional neural networks used for object detection are a special kind of feedforward networks.

More the hidden layers, more the depth of the feedforward networks. The width is given by the dimensionality of the hidden layer.

2.2.4 Loss function

As mentioned before, a mapping function f noisily approximates the input x to output y. So, the noise or the deviation from the true value(ground truth) must be kept at minimum. The function that

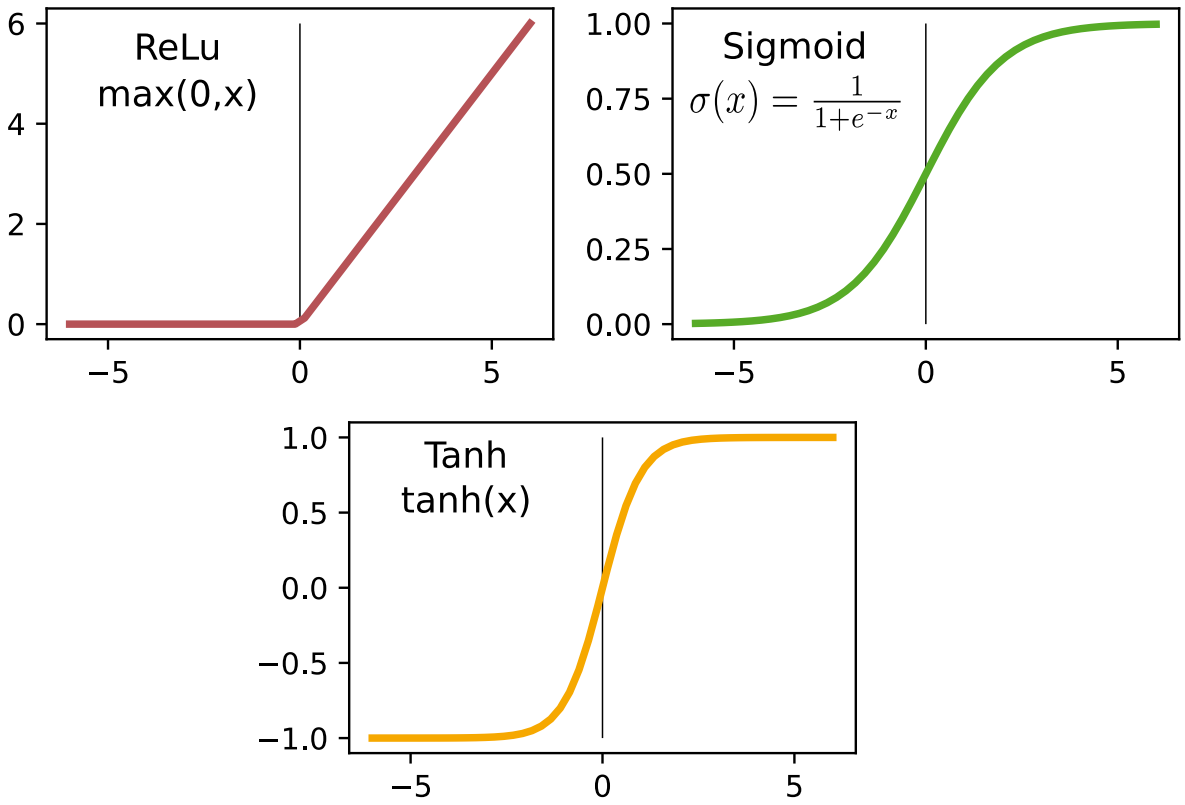

Figure 2.3: Activation functions

calculates the deviation is called *cost* or *loss* function. It is important to choose the right loss function for a model.

For multi-label classification tasks, *categorical cross-entropy* function is used. For each category, cross-entropy is calculated. The difference between the cross-entropy of training data and the model's predictions is the cost function.

$$CCE = -\frac{1}{N}\sum_{i=1}^{N}[\hat{y}_i \log(y_i) + (1-\hat{y}_i)\log(1-y_i)] \tag{2.2}$$

For regression tasks, the models are subjected to loss functions such as *mean absolute error*(MAE), *mean squared error*(MSE) and *mean squared logarithmic error*(MSLE). In MAE, the mean of absolute differences among predictions and expected results are calculated.

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{2.3}$$

In MSE, the mean of squared differences among predictions and true outputs are calculated.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{2.4}$$

In MSLE, the mean of relative distances between predictions and true outputs are calculated.

$$MSLE = \frac{1}{n}\sum_{i=1}^{n}(log(y_i+1) - log(\hat{y}_i+1))^2 \tag{2.5}$$

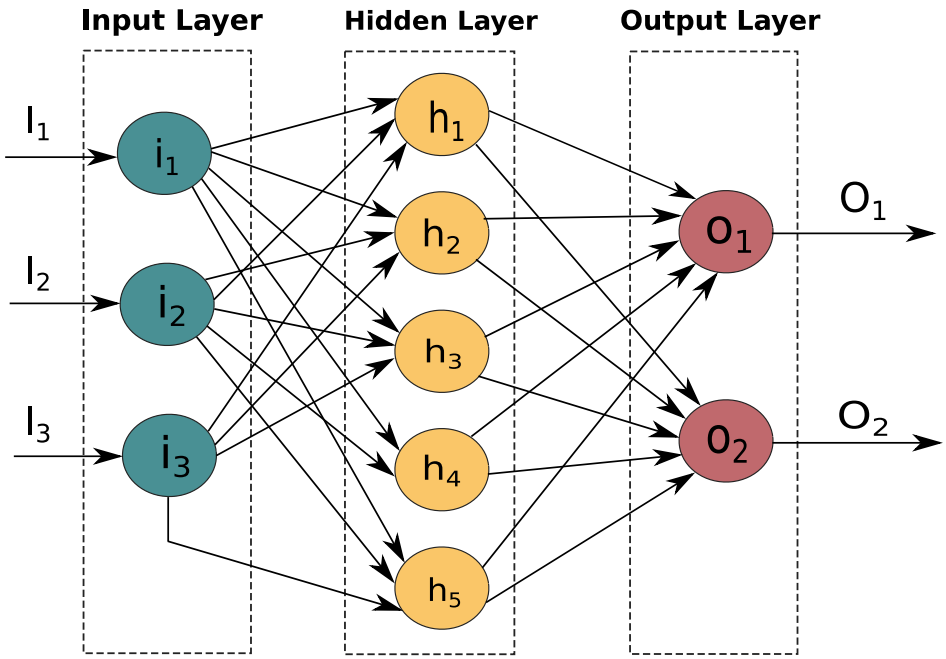

Figure 2.4: Multi layer perceptrons

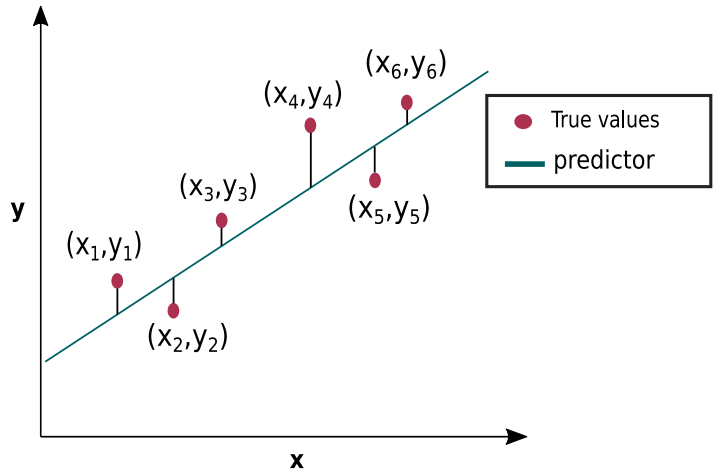

Figure 2.5: Mapping from x to y. The predictor is shown as linear line. The distance between the true values and predictor gives the loss. The sum of all the distances gives the loss function.

2.2.5 Gradient descent

Gradient descent is an optimization algorithm to minimise the cost function parameterised by a model parameter $\mathbf{w}$ in a function f. The first derivative(or gradient) gives the slope of the cost function. Hence, to minimise it, direction opposite to the gradient is chosen.

The rate at the which the gradient step reduces is given by the *learning rate*. It is one of the important parameters in training a model. It is also easily controlled by the user. Higher the learning rate, greater the step size of each gradient; possibly causing the step to miss the global minima. Lower the learning rate, more the number of steps or training cycles needed to reach the global minima. Greater care must be taken in choosing the learning rate when training a model.

2.2.6 Backpropagation

Backpropagation is the practice of fine-tuning the weights of a neural net based on the error rate (i.e. loss) obtained in the previous epoch (i.e. iteration). Proper tuning of the weights ensures lower error rates, making the model reliable by increasing its generalization. This practice is a part of model training these days.

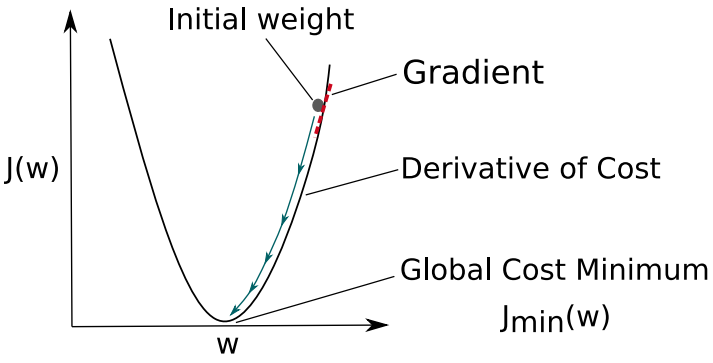

Figure 2.6: Finding the stochastic gradient descent

At the heart of backpropagation is an expression for the partial derivative $\frac{\partial C}{\partial(w)}$ of the cost function C with respect to any weight w (or bias b) in the network. The expression tells us how quickly the cost changes when we change the weights and biases. Since this method requires computation of the gradient of the error function at each iteration step, we must guarantee the continuity and differentiability of the error function. This can be achieved by using appropriate activation function such as tanh, sigmoid etc.

2.2.7 Optimizer

The loss function explains how far the predictions are compared to the true outputs in a mathematical way. During training process, certain parameters can be tweaked to help the loss function predict correct and optimised results. However, there are question such as how to change them, by how much and when?

This is exactly optmizer's function. As explained in 2.2.5, gradient descent and learning rate form the core of optimizer's functionality. *Stocastic gradient descent*(SGD) is one of the oldest techniques in which gradients for all of training examples are calculated on every pass. Hence, they are slow and require much computation power. Some of the other popular optimizers are Adam [32], Adagrad [33], RMSprop [1]. In this work, Adam is used. Adam stands for adaptive moment estimation. It is a combination of all the advantages of two other extensions of SGD – Adagrad and RMSprop. Adam is computationally efficient, straight forward to implement, invariant to diagonal rescale of the gradients, and less effort need to hyperparameters tuning.

2.2.8 Challenges in Machine learning algorithms

1. insufficient labelled data
2. poor quality data and irrelevant features
3. overfitting/underfitting a model

The first two issues can be solved if the user is careful during data collection and does preprocessing before feeding the data into the training model. However, if the training or the test data is too small, the model is subject to underfitting or overfitting. Though our aim is to reduce the error in the training

[1] RMSprop is an unpublished, adaptive learning rate method proposed by Geoff Hinton in Lecture 6e of his Coursera Class [34]

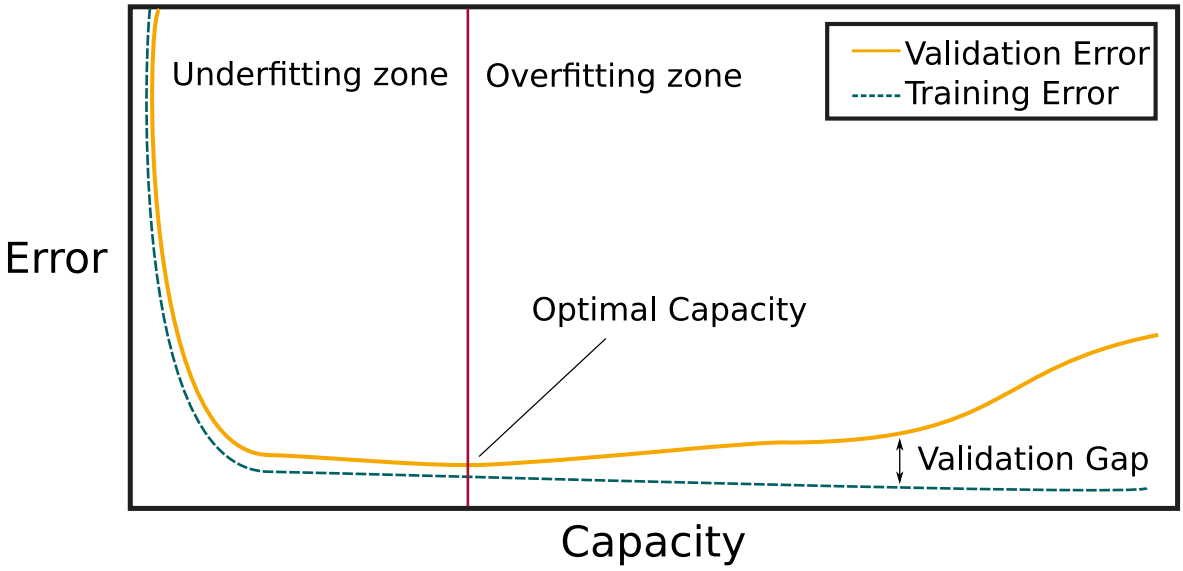

Figure 2.7: Relationship between capacity and error. Inspired from [27]

set, we also need to reduce the error in the test set. The gap between training and testing error is also an important parameter.

Underfitting occurs when the model is not able to obtain sufficiently low error value for the training set. And if the gap between training and testing error is too large, overfitting happens. The sweet spot is to stop training the model when the testing error increases while the training error decreases Left of the optimal point, the model underfits. Right of it, the model overfits. The figure 2.7 shows how the relationship between capacity and error. Validation error is the error calculated for the test set.

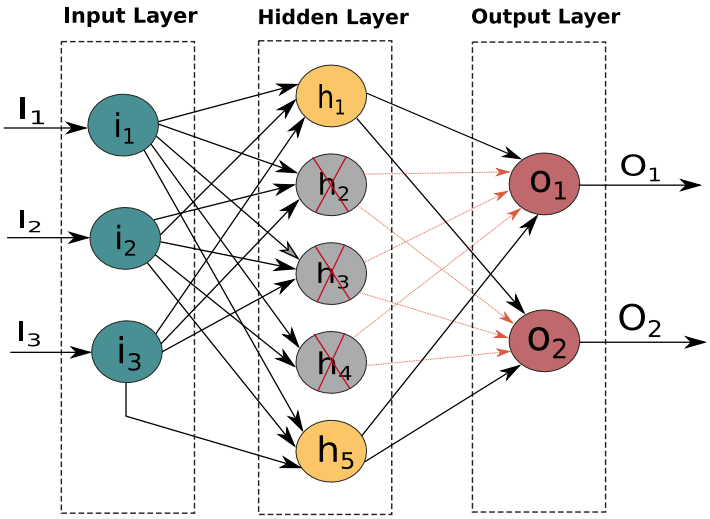

Figure 2.8: Illustrating dropout functionality

2.2.9 Regularization techniques

DNNs contain multiple non-linear hidden layers which make them easily learn complex relationships between their inputs and outputs. With a small training set, this relationship adds sampling noise that won't exist in the real-world data even if drawn from the same distribution. This leads to overfitting

and several methods have been developed to reduce its effect.

1. early stopping as soon as the validation error gets worse than the training error.
2. L1 and L2 regularisation which penalises the weights [35].
3. Randomly drop units(along with their connection) from the neutral network during training [36]. Figure 2.8 illustrates how to do the random dropping of units.

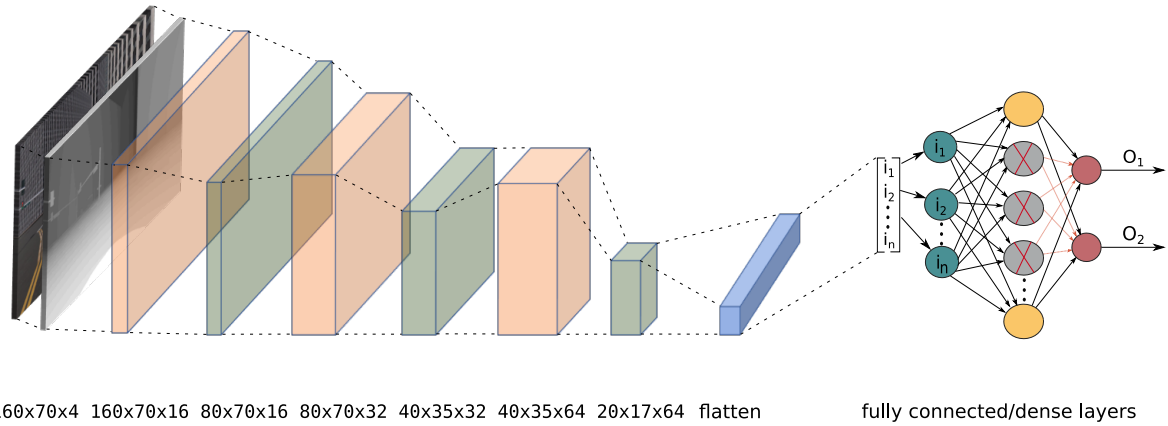

Figure 2.9: CNN architecture

2.2.10 Convolution Neutral Network - CNN

Convolutional neural network(CNN) or in short convnets is a deep learning algorithm for object recognition tasks. What makes CNN stand out for image analysis? The network takes images as inputs, reduces them into a form easier to process, without losing features which are critical for a good prediction. So not only this is important to consider while designing an architecture but also while scaling massive dataset.

Convolution Layer

The images which are taken as inputs are just n-dimensional matrix with pixel values. So a convolution operation can be easily carried on it using a filter or *kernel*. A kernel matrix is pre-defined according to the task. Usually the size of the kernel is tiny compared to that of the images which facilitates easy convolution. A *stride* is the value of a step taken by the kernel after each convolution. If stride = 1, then it is called *non-strided*. Convolution remarkably extracts the high-level features such as edges. Normally there are many Convnets in an architecture. Each layer extracts a different feature or expands on last layer's task.

If the dimensionality of the convolved feature stays the same or increased compared to the input, then it is called *same padding*. If the dimensionality is reduced, its *valid padding*. Padding is extremely useful for solving boundary conditions.

Pooling Layer

This layer is similar to convolutional layer. Its task is to decrease the computational power required to process data, usually done through reducing the dimensionality. It is, furthermore, useful to extract

dominant features that are rotational and positional invariant, thus maintaining the goal of training the model.

There are two types of pooling – *max pooling* and *average pooling*. Max Pooling returns the maximum value from the portion of the image covered by the Kernel. On the other hand, average Pooling returns the average of all the values from the portion of the image covered by the Kernel. Pooling also helps in reducing the noisy pixels which sometimes skew feature extraction.

Flatten and Fully Connected Layer

The main goal for extracting features from the images is to do some task; for example - classification. So the extracted features must be converted into a form understandable for the MLP (2.2.3), which happens to be 1-dimensional vector. This is exactly the task of *flatten layer*.

MLP gets a vector as input and feeds it to a feedforward network in *fully connected layer*. Fully connected layer then outputs the necessary values depending on the task.

It is important to remember that each layer employs an activation function(2.2.2) to introduce linearity or non-linearity to the inputs.

2.2.11 Recurrent neural networks - RNN

One of the drawbacks of neural networks is that they always start from scratch; with no memory of the previous state. If a neural network has to be used for word prediction, knowledge of previous letter and word is necessary. Recurrent neural networks addresses this issue.

RNN provide the temporal dynamic behaviour. A typical RNN looks like in the figure 2.10. The left hand side shows it folded and right hand side unfolded in time. RNN, however, suffers from *long term*

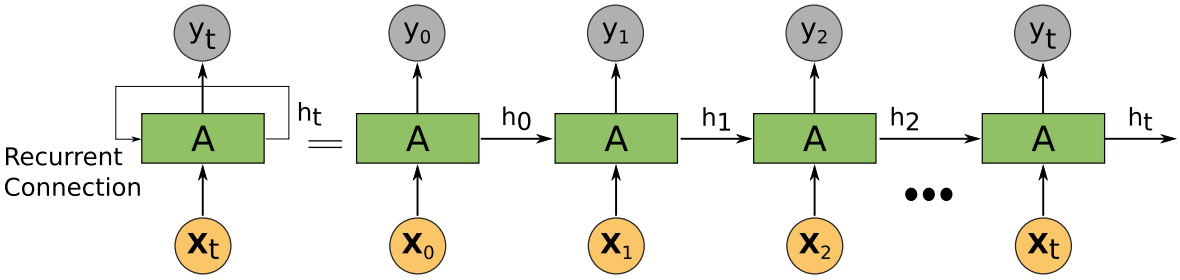

Figure 2.10: A Simple RNN

dependencies. This is well explored in [37] and [38].

2.2.12 LSTM

The shortcomings of RNN are overcome by LSTM - *Long Short Term Memory*. They are a special kind of RNN which was first introduced by Hochreiter *et al.* [39]. They remember information for long periods as their default behaviour with ease. The figure 2.11 shows how the structure of a LSTM differs from simple RNN. The LSTM employs gates and activation functions to add or delete information from the previous state.

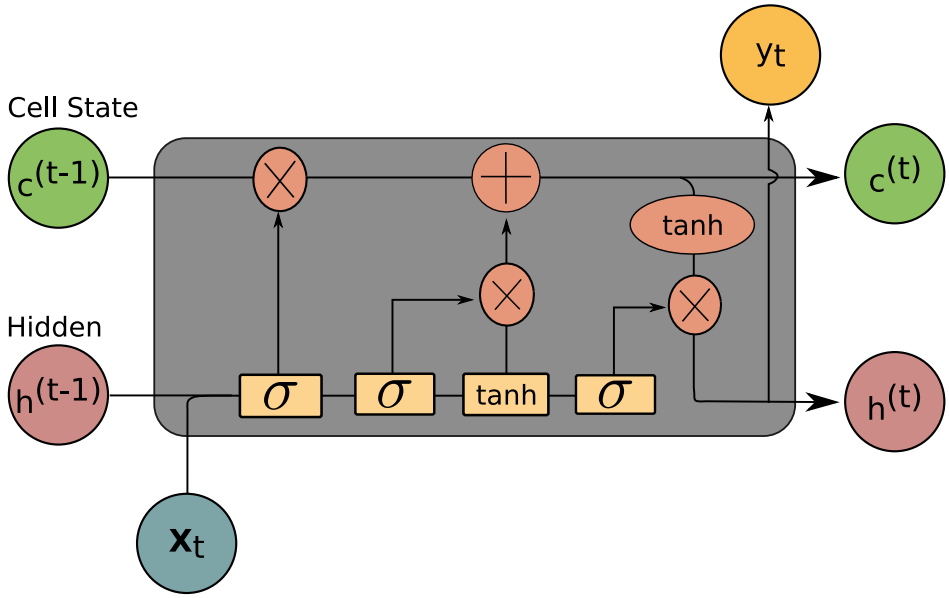

Figure 2.11: LSTM Architecture - Rolled

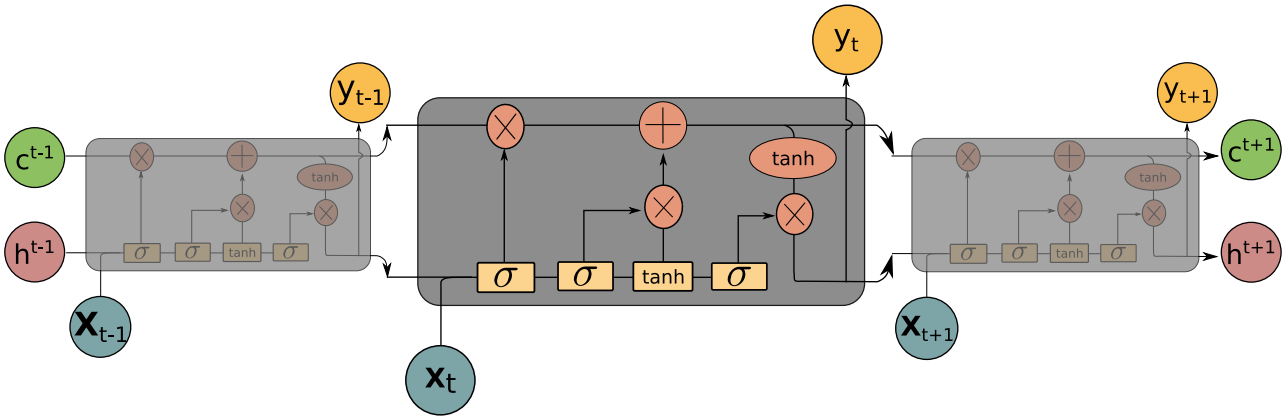

Figure 2.12: LSTM Architecture - Unrolled

2.3 Sensors

Deep neural networks need data – images or measurements to perform necessary tasks. These information/data are captured using sensors.

2.3.1 Visual Sensors

Visual sensors are one of the commonly used sensors for image capture of the environment. Usually cameras are used.

RGB Colour Camera

A camera uses the lens to focus and captures objects. The information travels in the form of electromagnetic waves such as light. The sensor that is present behind the lens are made of photodetectors, is exposed to allow incoming light. An variable electric charge is produced depending on the intensity

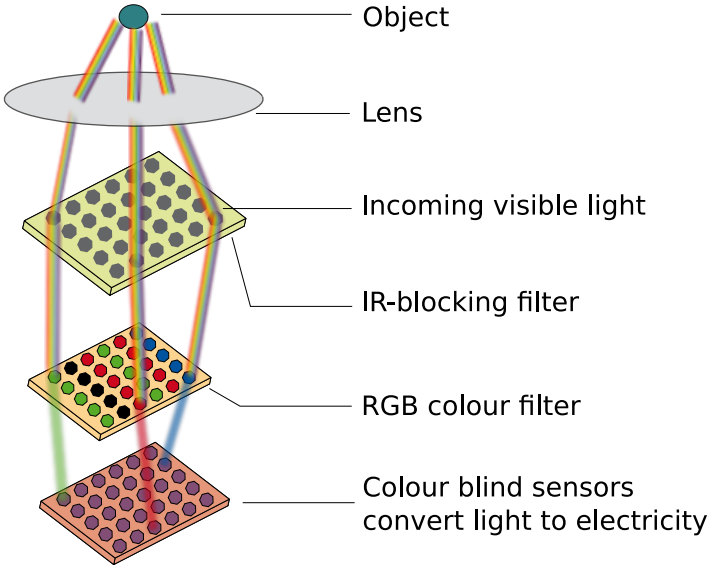

Figure 2.13: Inside RGB camera

of the light waves. The intensity of the light waves changes with object's exposure. These charges are then quantified and stored as numerical values called *pixels*.

A pixel is generally the smallest single component of an image. Each pixel are arranged one after another in the form of matrices. So, for a resolution of "640 x 480" display, they are 640 pixels side to side and 480 from top to bottom. A colour image is captured by using a colour filter such as Bayer filter to filter out only light waves that are of RGB colour spectrum wavelength. Each pixel are then coded in *bits*. In our case we use eight bits to represent an image.

Depth Camera

A depth camera is usually a stereo camera with two cameras. They are displaced horizontally from one another. They are used to obtain two differing views on a scene. Images are captured from these points and comparing the pixel values gives the relative position of the objects. A Depth camera

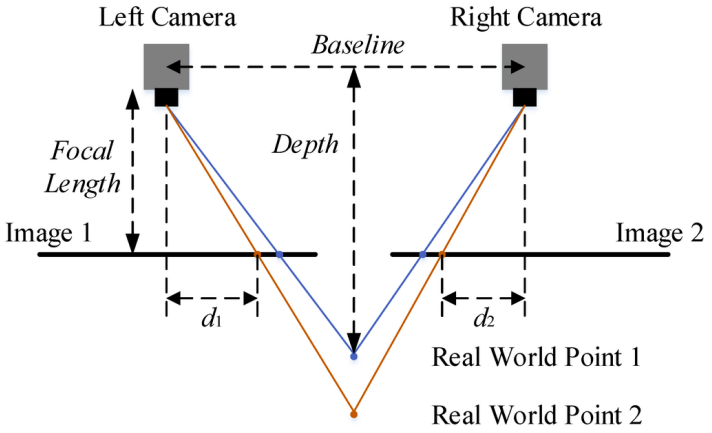

Figure 2.14: How depth sensor works. Figure redrawn using a website [40] as reference.

sensor, in our case, captures images just like a colour camera but only as grayscale images – black and white pixels. These pixels are then stored with eight bits per pixel(shades of grey). The shades on the grey-scale correspond to the depth of objects. This paper [41] shows how to combine a low

resolution Time-Of-Flight (TOF) depth image camera based on Photonic Mixer Devices with two standard cameras in a stereo configuration and without accurate calibration. And this paper [42] gives the basics on stereo camera for object perception.

Segmentation Camera

RGB images are fed to CNN-DNN which groups the pixels of similar attributes using a process called image segmentation. The most common image segmentation method is thresholding. Pixels with certain threshold are grouped together. This allows to images to have segments that may be more meaningful to analyse those segments than the whole image for relevant information.

For autonomous driving, semantic segmentation method is used. RGB images are fed to a DNN to group pixels according to user-defined tag. For example, a car is blue, pedestrian red, road boundaries white etc. This paper by Poudel *et al* [43] uses encoder-decoder architecture to do offline semantic image segmentation.

2.3.2 Measurement Sensors

These sensors are required for providing information other than visual such as telemetry data.

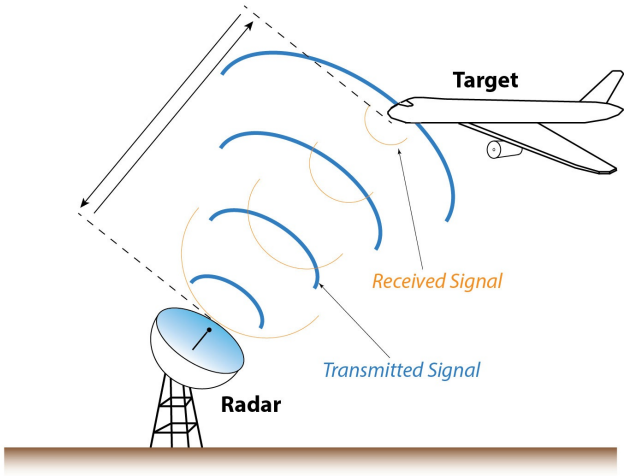

Figure 2.15: How radar works

Radar Sensor

Radar is a detection system that uses electro-magnetic waves(radio waves) to determine the range and velocity of objects. A common sensor that is used in weather forecast which is known for its long range sensing and resistance to adverse weather conditions. It consists of a transmitting antenna with a transmitter producing radio or microwave waves, a receiving antenna and receiver to process the information. Radio waves reflect off the object and return to the receiver carrying the information about object's velocity and position. If the object is moving either toward or away from the transmitter, there will be a slight change in the frequency of the radio waves due to the Doppler effect.

Three major classes of radar systems are typically employed in automotive active safety systems:

1. Short-range radar (SRR) for collision proximity warning and safety, and to support limited parking assist features.
2. Medium-range radar (MRR) (24GHz) to watch the corners of the vehicle, perform blind spot detection, observe other-vehicle lane crossover and avoid side/corner collisions.
3. Long-range radar (LRR) (77GHz) for forward-looking sensors, adaptive cruise control (ACC) and early collision detection functions.

However, in LGSVL simulator, only high-level implementation of radar is implemented. It does not use any waves, reflection or occlusion techniques. Since the simulator is aware of vehicles' position at any given moment in the map, the information is converted to depth information and stored.

Control Sensor

With this sensor, telemetry information can be collected. This is usually done by encoding the key presses in the keyboard.

2.4 Sensor/Data Fusion

To allow DNNs to make the best perception of the environment, it is necessary to fuse data from several sensors and feed that combined data into the DNN. This technique of fusing information exists for decades [44]. Often used data fusion technique is *Kalman filtering* and its variant [45] [46]. [47], [48] give a comparison in performance between using Kalman filter and LSTMs. [49] uses recurrent YOLO(LSTMs) to track objects through space and time.

For autonomous driving, RGB and depth information(RGB-D) is vital for obstacle avoidance. [25] uses data fusion to get better results for their experiment.

2.4.1 Types of Data Fusion

There are two traditional approaches to data fusion – *early fusion* and *late fusion*. In early fusion, all the sensor inputs are concatenated before being fed to the CNN. Whereas in late fusion, each sensor inputs are fed to separate convolutional layer and down the line, they are concatenated together.

These techniques can be seen in action in this [24] recently published paper from Facebook research team.

2.5 Machine Learning Library

To train the neural networks, we need a ML library framework to programme it. *Tensorflow* [51], *Keras* [52], *Pytorch* [53] are some of the popular ML frameworks used today. For this thesis, we use Keras, a python high-level wrapper for tensorflow. With keras, one can easily design DNN architectures with minimal effort. All the DL techniques we discussed above are reduced to a bunch of human understandable commands. Keras has several application programming interfaces(APIs) – Models, Layers and Callbacks.

2.5.1 Models API

They are of two types – Sequential and Functional. Sequential is just stack of layers with one input and one output. Functional can handle models with non-linear topology, shared layers, and multiple

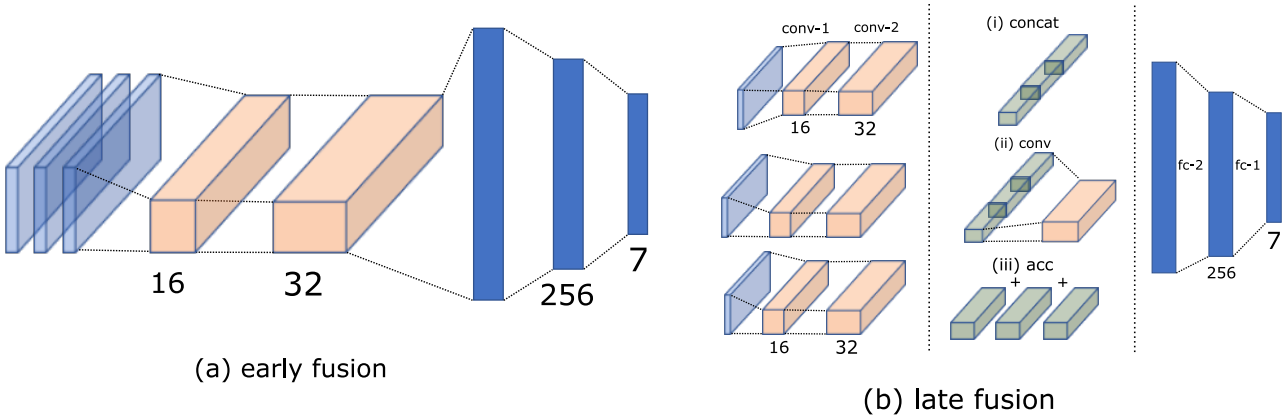

Figure 2.16: This figure is taken from this [50] paper where they describe early and late fusion architectures and also present three types of late fusion.

inputs or outputs. Since functional API offers flexibility, it is used here. In addition to offering the overall functionality, this API has the power to implement optimizers (2.2.7), loss functions (2.2.4).

2.5.2 Layers API

The layers needed for CNN – convolution (2.2.10), pooling (2.2.10), normalization, regularization, activation (2.2.2) and time series operations with LSTMs (2.2.12) are easily implemented with this API.

2.5.3 Callbacks API

With this API, some of the overfitting challenges can be automatically avoided. Some functionalities available are early stopping (1) and ModelCheckpoint (2.2.8).

Early stopping sets an epoch parameter *n*. If the gap between training and validation loss don't improve/reduce for the next *n* defined epochs, the training is automatically stopped.

With ModelCheckpoint, the gap is monitored w.r.t a monitoring parameter; usually minimum validation loss or maximum validation accuracy. Then automatically the best model gets saved.

In order to visualise the performance of the training, *TensorBoard* class is used.

2.6 Robotic Operating System - ROS

The Robotic Operating System(ROS) [54] is a set of software libraries and tools created to help developers build robot applications. From drivers to state-of-the-art algorithms, and with powerful developer tools, ROS is a necessary set of tools for any robotics project. And its all open source.

ROS environment was first developed by Willow Garage for the PR2 robot [55]. PR2 is a humanoid robot that can navigate autonomously in a known environment. Since then, ROS is now used in all kinds of robots in various fields. With its popularity, many companies manufacture ROS compatible robots. This massively helps in integrating multiple components to communicate with each other.

ROS(ROS1), since its launch, was considered as a middleware/interface between components. There was a parent to which all the children were connected. Every child node had to go through the parent every time to discover another node. In today's expanding robotics market, this approach is outdated.

This led to the development of ROS2 [56].

2.6.1 ROS2

ROS2 uses a data distribution service(DDS) for publishing and subscribing instead of custom message handler. With DDS, the transmission performance is also improved. Each node is *peer-to-peer* and can contact other nodes efficiently. ROS2 is not simply an extension of ROS1; although some of the functionalities have been ported.

2.6.2 ROS2 concepts

In this section we will study different concepts used in the thesis.

Nodes

A node is an entity that uses ROS protocol to communicate with each other. In a ROS graph, there are networks of nodes and connections between them.

Messages

Messages are ROS data type that are used when subscribing or publishing to a topic.

Topics

A topic is named information *bus* over which nodes exchange messages. A topic usually begins with *"/"* followed by the topic name. For example, "/radar" is topic associated with radar bus. Each topic carries information of a particular message type. This message type can either be a standard or custom type.

Subscriber and Publisher

If a node subscribes to a topic, then the node is called a *subscriber*. If it publishes to a topic, then it is a *publisher*.

Both publisher and subscriber when they are initialised over a topic, a *queue size* is defined. Depending on the queue size, a topic's messages can be queued and processed as needed. The figure 2.17 shows a subscriber and publisher node exchanging data with each other.

Spins and Callbacks

In computer programming, spinning is a technique in which a process repeatedly checks to see if the condition is true. In ROS, a node is set to spin with or without a condition. This enables it to do its tasks as programmed.

Also from computer programming, callback is a function that executes at a given time. There are two types of callbacks – *blocking* and *deferred*. In ROS, deferred callback is used. It means that the callback function is invoked after a node returns something. It can be a subscriber receiving a message from its subscribed topic.

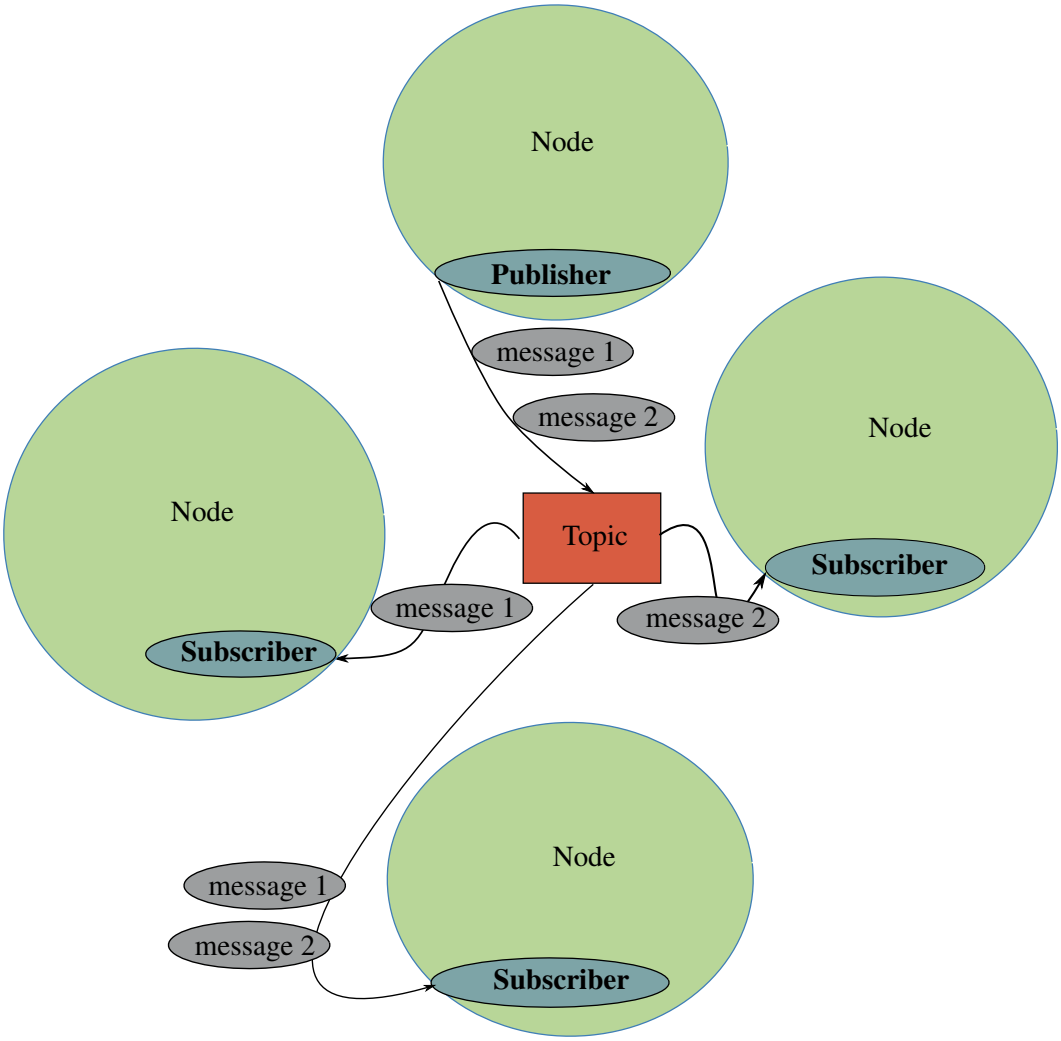

Figure 2.17: A graph showing how a publisher or subscriber node interact and exchange messages with each other through a topic.

Rosbridge

We are aware that there are some non-ROS robots which would need to communicate with ROS ones. So a rosbridge [57] acts as communication API between these two. The rosbridge specification is programming language and transport agnostic.

Message Filters

Since the main goal of this thesis is to do data fusion, we need to use ROS to communicate with different sensor nodes. These sensor nodes receive and transmit data at different rates. So we need a filter that can trap the received or transmitted messages, serialize them(so as to not lose data's integrity) and make it possible for storage.

With the help of a filter, all the nodes can be made to wait till every node receives the message and then invoke the callback function only once or multiple times as per design. Inside the callback, further operations can be carried out before saving.

Message_filter [58] is one such filter. It has the functionalities we are looking for, such as TimeSynchronizer, cache(a buffer to store messages while waiting for others), and slop(an extra delay pa-

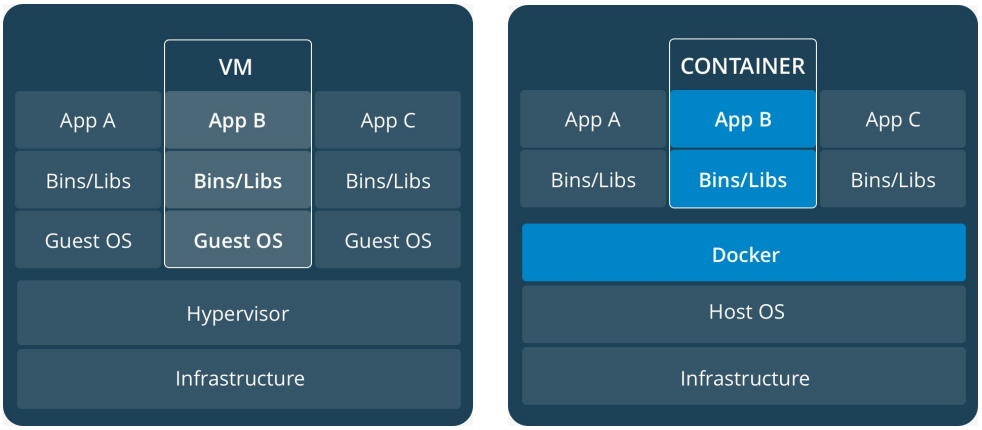

Figure 2.18: Difference between VM and Docker

rameter to TimeSynchronizer modules which defines the delay(seconds) with which the incoming messages can be synchronized.) Caution must be kept when choosing the slop value. Otherwise, the data will lose its integrity.

In the next chapter, we will see how the LGSVL [8] simulator is used.

2.7 Docker

Docker [59] is an open-source tool designed to make it easier to create, deploy, and run applications by using containers. These software containers allow developers to package an application with all the parts it needs, such as libraries and other dependencies, and deploy it as image based packages. By doing so, developers can be sure that their application will run smoothly irrespective of the client's environment. This also allows for easy debugging and development.

Docker can also be loosely considered as virtual machine(VM). But unlike a VM, rather than using a whole operating system, a docker shares the kernel of the system and ships only the applications that are not in the host machine. Also a docker container is independent of host machine's applications. This greatly improves performance.

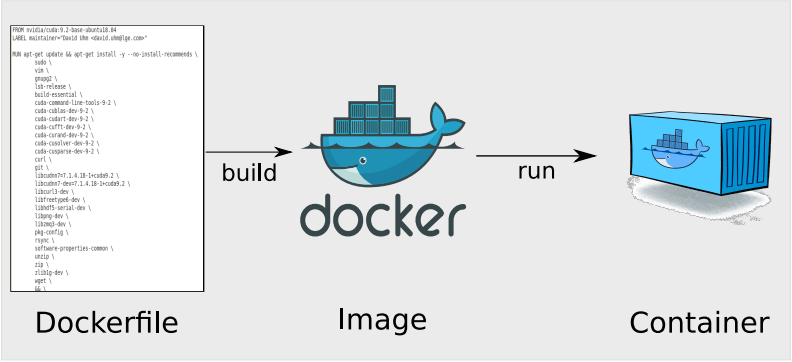

Figure 2.19: How a docker image is created

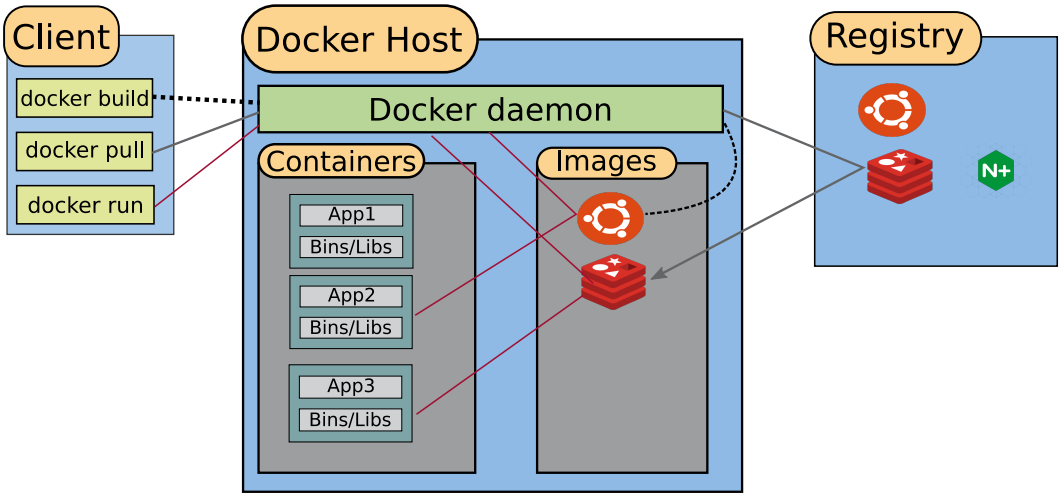

Figure 2.20: Docker Architecture

However, docker has its drawbacks. For example, building a docker image sometimes takes long to compile and consume a lot of resources. So not everyone can build an image as regularly as they wish.

In this thesis, docker containers are used for data collection from the simulator and later for evaluation of the model with the simulator.

3 Simulation and Simulator

From the beginning of autonomous driving research, simulators have played a key role in development and testing new algorithms. Simulators allow developers to quickly test their algorithms without driving real vehicles. In this chapter, we will the conditions a simulator must satisfy, and go in detail about LGSVL simulator and its development.

3.1 Need for a simulator

One of the important questions to ask before explaining about simulation is to understand why should one need a simulator to do simulation. As explained in the previous chapter (2), deep neural network(DNN) using supervised learning algorithm needs huge amount of labelled data. Since the cost of collecting that amount of data in real road vehicle is too expensive, researchers have sought the help of simulators. A simulator is an application which simulates a real-world environment, virtually. With the help of a simulator, one can collect any amount of data they wish for their project.

3.2 Conditions for a simulator

Data collection is one of the most important phases in supervised learning. So caution must be taken in choosing a simulator. A simulator must fulfil certain conditions to be qualified as a good one.

- It must have a vehicle that can move around in a virtual map.
- The vehicle must be equipped with appropriate sensors for perceiving the environment properly.
- The virtual map must try to mirror the real-world to an extent. That mean it should have proper terrain to drive around, lane marking for lane detection, other cars to mirror the real world traffic, pedestrians, and real world weather conditions.
- It must provide a medium to collect data and allow interfaces to transfer the data. It should also be able to receive data in case the user needs to validate the data collected.
- Finally and most importantly, support end-to-end, full stack simulation.

3.3 LG SVL simulator

A simulator chosen for this thesis is from LG research centre in Silicon Valley, California called LGSVL simulator. It is an open source project where the code is regularly published at Github [60]. This simulator satisfies all the conditions listed above. They provide an out-of-the-box solution which can meet the needs of developers wishing to focus on testing their autonomous vehicle algorithms. It also supports Apollo [61] and Autoware [62].

3.3.1 LGSVL simulator development

LGSVL simulator's core simulation engine is developed using the Unity game engine [63]. Unity game engine is written in C# programming language. Since a game engine inherently supports animation, the simulator is able to extend that functionality easily. In addition to Unity, also supports several libraries necessary to compute complex mathematical operations. With Unity's latest High Definition Render Pipeline(HDRP), LGSVL is able to simulate photo-realistic virtual environments that match the real world.

3.3.2 Overview of LG SVL simulator

User AD Stack

It supports user autonomous driving(AD) stack. That means a user can develop, test and verify through simulation. The user AD stack connects to LGSVL Simulator through a communication bridge interface; a bridge is selected based on the user AD stacks runtime framework. This bridge interface can use a standard protocol such ROS, ROS2 or custom one like CyberRT [61].

In addition LGSVL supports plug-in component which a user can develop and attach it to the simulator. The simulator during runtime picks up this plug-in.

Simulation Engine

As mentioned above, LGSVL uses Unity's latest HDRP game engine.

Sensor and vehicle models

It supports sensor arrangement and importantly they are customisable. The sensors are added and removed through JSON formatted text along with its parameters. These parameters include sensor type, position of the sensor, topic name, publishing rate, and in some sensors reference frame of measurement. Some of the popular sensors like camera sensors, radar and LIDAR are supported. In addition, users can add their own custom sensors as plug-in. Fig.3.1 gives a good overview of some of the sensors in action.

Vehicles provide a medium to travel the environment. Hence, vehicle dynamics is also important.

Environment and maps

An environment, in this case, virtual, is a primary component in autonomous driving simulation to provide many input to AD system. An environment affects almost all the functionalities in a AD system such as perception, prediction and tracking modules. It also affects the vehicle dynamics which is the key factor in vehicle control mechanism. Through changes in the HD map, the environment affects localization and planning modules. Finally, weather conditions such as rain, fog, night driving naturally affect the environment. So caution must be taken while design the environment.

LGSVL supports creating, editing and exporting HD maps of existing 3d virtual environment. 3D environment also defines the rules about how agents must behave such as stopping at traffic lights, giving way to priority traffic, respect lane boundaries etc.

As of writing, LGSVL supports virtual Sanfranciso city HD map. They also support smaller maps like Shalun and Cubetown.

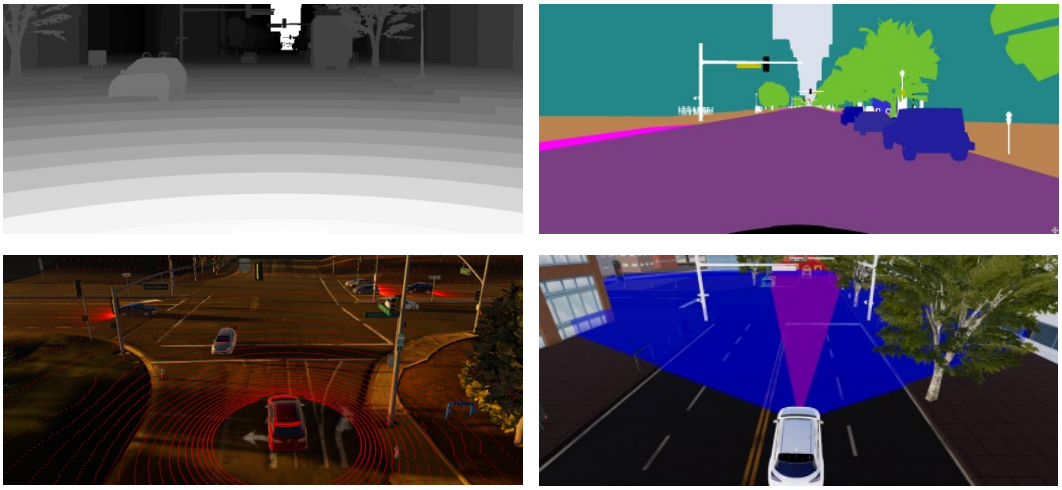

Figure 3.1: Different types of sensors in LGSVL simulator. Anticlockwise(from top): Depth camera, LiDAR, Radar(also 3D bounding boxes),and Segmentation camera

Figure 3.2: LGSVL simulator in different weather conditions

Test scenarios

Test scenarios enable users to test their AD stack by simulating in an environment and comparing and contrasting correct and expected behaviours. A lot of variables like HD maps, traffic movement behaviour and their density, time of the day, weather conditions etc. also play a role while testing. It is also possible to write scripts with the help of Python API where scenarios can be created and tested.

Thus LGSVL simulator [8] provides the best virtual environment to conduct our experiments for autonomous driving.

4 Implementation

This chapter will present the implementation of end-to-end network with its extensions. First we start with docker to set the environment. Then move on to LGSVL and ROS. From there a closed loop is achieved to collect data, preprocess, introduce neural network, implement the models, and evaluate the trained model. After achieving the basic results for the preliminary architecture, sensor fusion techniques are implemented.

4.1 Docker

Docker is an open-source platform for developing, shipping and running applications. Because docker makes installing applications hardware independent, we use docker for our tasks.

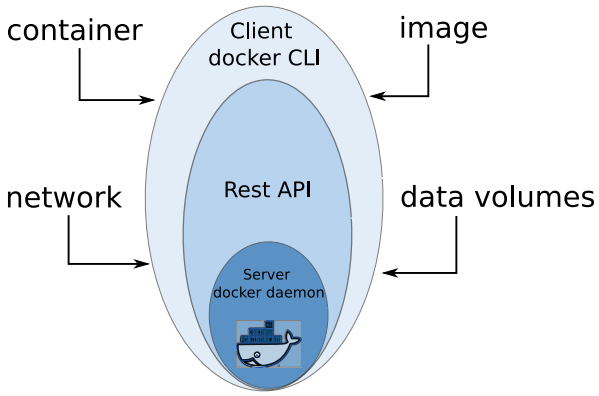

Figure 4.1: Docker Engine and its functions

A docker architecture, as shown in 2.20, consists of client, host and registry. To make all these components work, docker daemon is necessary. A daemon is a type of long-running background process. The LGSVL docker image is either pulled from the registry using *docker pull* command or built using a *dockerfile*. An image is a read-only template with instructions for creating a docker container. In our case since the image is readily available, so we pull it.

A docker container for each task can be defined. Along with a task, certain other services may need to be run along with it. *Docker compose* gives a perfect solution to manage docker applications. As seen in figure 4.2, docker-compose helps execute multiple commands. Upon execution, ROS environment are set then, a websocket with a port exposed. In our case, it is `http://localhost:9090`. The LGSVL on the other side through its web user interface(WebUI), listens on this port. So a bridge is established to flow of data.

4.2 LGSVL simulator

The LGSVL simulator is developed using Unity engine which is written in C# language. The LGSVL team organises their code base[60] in such a way that it makes it easy for a beginner to learn the structure and either implement new features or change the existing ones.

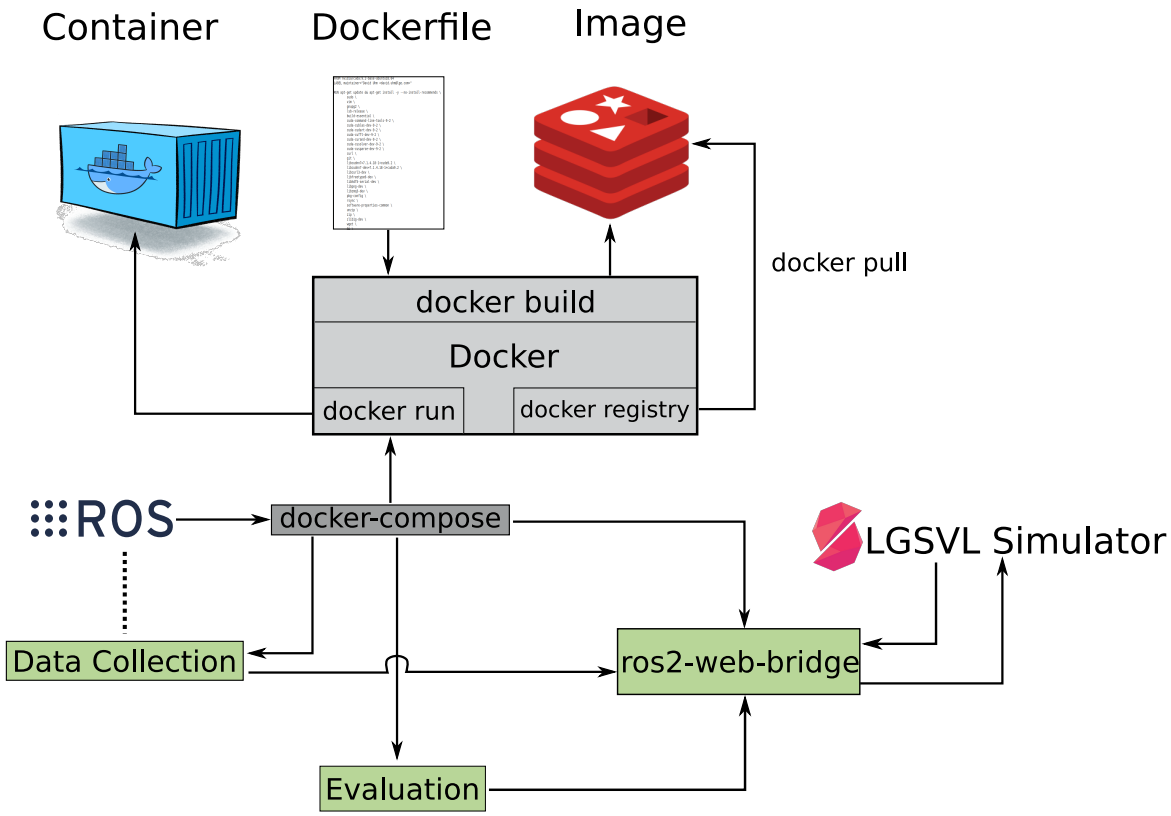

Figure 4.2: Docker and its various functions

4.2.1 Simulator configurations

The LGSVL team has developed a WebUI to help users to configure maps, vehicles and simulations. The LGSVL application connects with the WebUI through a web-socket(`http://localhost:8080`). The users are allowed to configure the simulation settings using the UI. These configurations are stored in JSON format. Some of the JSON parameters are parsed in the UI itself and some are transferred to the application using *http* protocol.

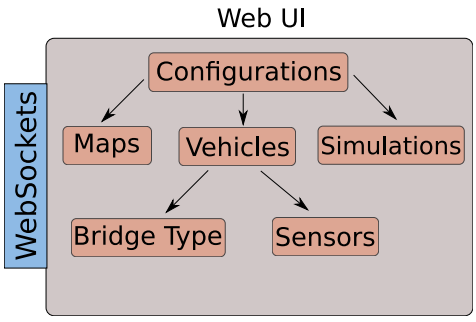

Figure 4.3: LGSVL Simulator - WebUI

Sensor parameters are defined in vehicles *tab* of the WebUI in JSON format. If a user wishes to use a colour camera sensor, then they need to use the JSON format appropriate for this sensor to the vehicle configuration. Each sensor has a *topic name*. The LGSVL application parses the JSON parameters and stores them appropriately. In order for the ROS 2 nodes to subscribe to the sensors, they would need to use the exact topic name as defined in the vehicles tab.

In our case, we use a variety of sensors:

1. RGB colour camera,
2. Depth camera,
3. Segmentation camera(uses the output of a RGB camera sensor and internally segments them according to the tags defined by the user in the sensor parameter),
4. Radar sensor.

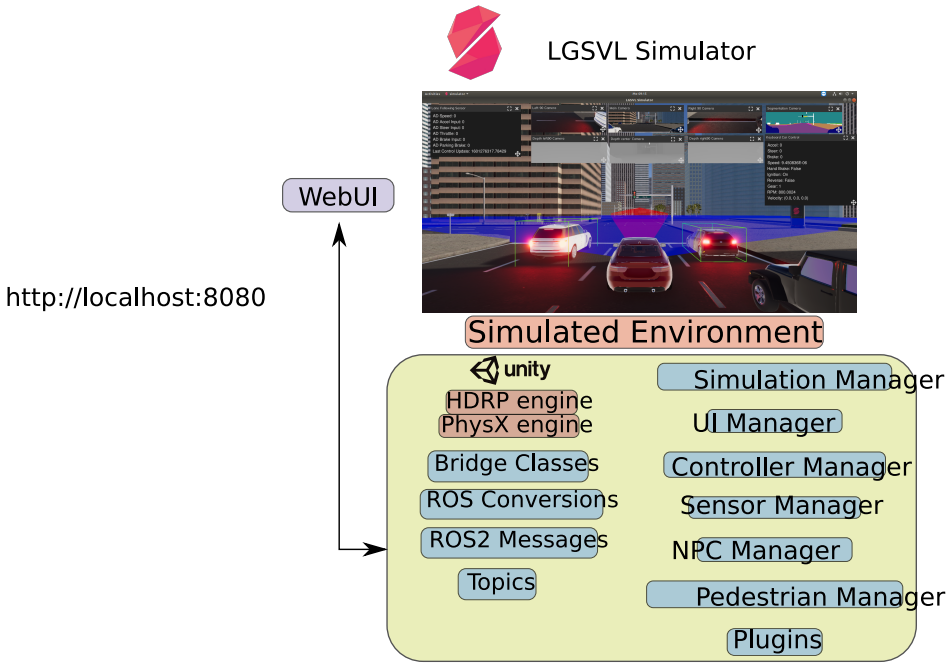

Figure 4.4: LGSVL software architecture

The user is also given the flexibility to arrange/align these sensors in different constellations according to requirements using a *transform* parameter.

So, we have

1. a RGB camera placed facing ahead parallel to the ground, another on the left and right side of the car pointed an angle towards the ground,
2. a depth camera following same configuration as RGB,
3. a segmentation camera placed adjacent to the RGB front facing camera,
4. a radar sensor placed front of the car near to the hood pointing ahead.

4.2.2 Inside LGSVL simulator framework

The figure 4.4 shows some of the major functionalities inside simulator framework. When the application is first started, the WebUI is called. Natively, the app connects to UI via a websocket. Maps, vehicles, and simulation settings are communicated and initialised.

As these settings get transferred, internally, a *simulator manager* program is triggered. This is a main manager responsible for the overall functionality of the simulator. From this manager program, other managers are invoked when necessary. The *sensor manager* is responsible for coordinating with all sensors, the *NPC manager* manages traffic, signal intersections, speed limits, etc., the *controller manager* manages controls for vehicle movement. Natively, LGSVL uses Unity's *PhysX* engine for vehicle dynamics. Using Unity's *HDRP* graphics engine, the simulated environment is visualised. The *UI and Camera managers* are then responsible for the display of the environment in the application.

So, the sensor manager that oversees the sensor JSON parameters are initialised and assigned correctly, and the information from the environment using respective sensors. These sensors' data are then passed on to *bridge classes* for ROS data type conversion and consequently publish to the subscribing ROS nodes via a bridge. The topics defined in JSON are used to publish respective sensor data.

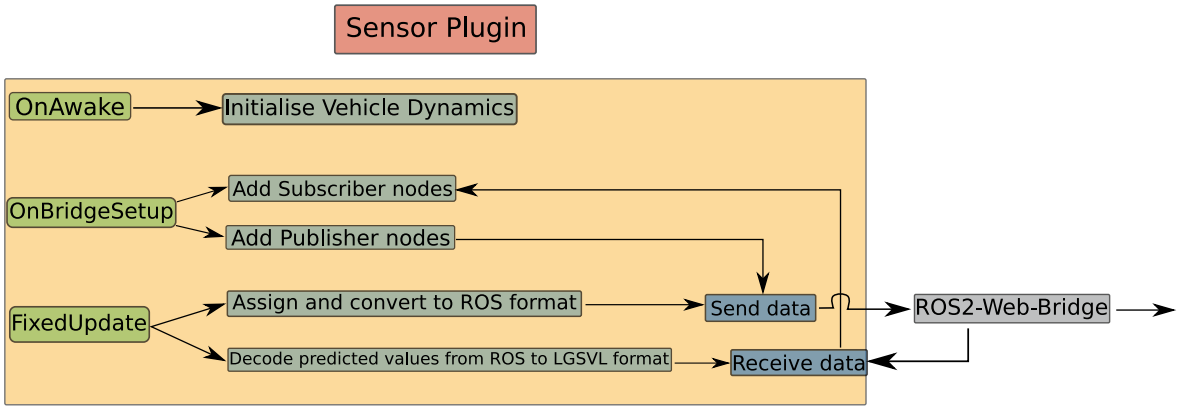

Figure 4.5: Inside sensor plugin

Control Commands

The control manager is responsible for vehicle control. The steering control is defined using Unity math function as continuous float values between -1 and 1. The velocity is determined by distance travelled in the map over time. Its unit is *meter per second*. The acceleration, throttle and braking are defined as absolute values and are interconnected to one another. Acceleration ranges from -1 to 1. While accelerating, it is set as 1. Throttle is also assigned this value. While braking, the acceleration is set -1 while braking 1.

4.2.3 Sensor plugin

When a user doesn't want to disturb the current setup of the simulator but rather wants to add some custom sensor to the vehicle configuration, sensor plugin can be used. A custom sensor is useful when there is a need to combine multiple standard ROS message type or define custom message types. LGSVL allows this functionality. A set of guidelines must be followed while developing the plugin. In our case, it is necessary to have a sensor plugin that would create a sensor and topics. This sensor and these topics would then be used to fetch data from the simulator, and transfer it through rosbridge.

This custom plugin as shown in figure 4.5 extends the unity engine libraries just like any other sensors to read the values from the JSON definition. Upon *OnAwake*, the vehicle dynamics is initialised. Then a check whether a bridge is available is done. In *OnBridgeSetup* method, publisher and subscriber w.r.t to the LGSVL simulator is created with topic names. At a fixed time interval, *FixedUpdate* method is invoked. Here the values obtained from vehicle dynamics are assigned appropriately. In addition if the task is to publish, the values such as steering, throttle, braking, velocity are converted to ROS defined format. If the task is subscribing, the data received through the topic such as predicted control values from the neural network model, are assigned to LGSVL variables to be reflected in the application.

LGSVL simulator is now configured to send data towards the client. In order to reach the client, as mentioned before, a rosbridge is needed. In the next section we will talk about ROS and its uses.

4.3 Data Collection Module

The figure 4.6 gives an overview of the data collection task of sensor module.

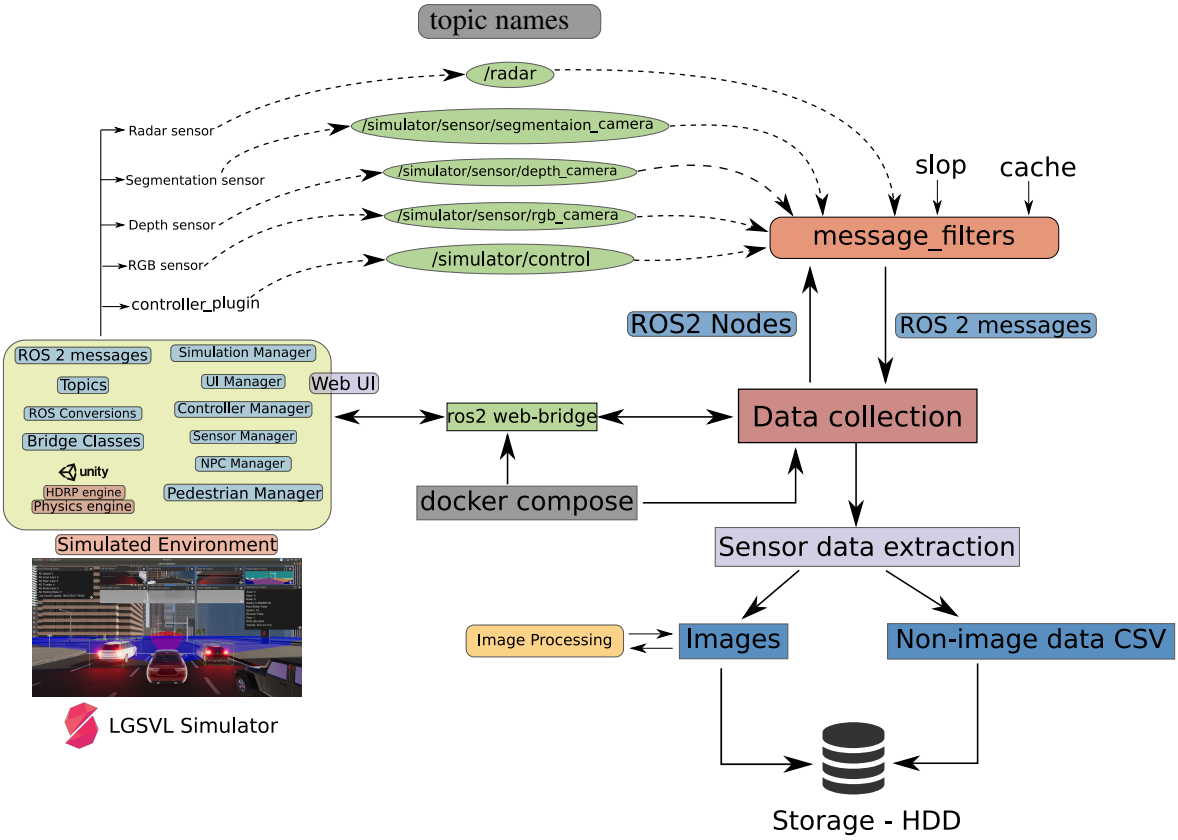

Figure 4.6: A detailed summary of data collection module

4.3.1 An overview of data gathering task

ROS, in our case, acts as an interface between simulator(server) and scripts(client). We use ROS 2 and in particular *dashing* iteration. The script's ROS 2 subscribing nodes listen to the sensors' topics(defined using JSON sensor parameters) and invoke a callback whenever they receive data. Since each sensor receives at different rates, a filter called message filters is used. With message filters, the queue size is set to a higher value for example, 1000 and a delay(in seconds) through a *slop* parameter of value 0.1 are used. This filter gathers all the subscribing nodes as one, synchronises approximately to the delay parameter and invokes just one callback. This assures that data from each listening node is present.

Inside the callback, the ROS 2 sensor data received have a header and data parts.The header part consists of the time at which the message is created and data part contains the real data. The next step would be to extract the real data. If the data are images then image processing is done. If not, the scalar values are stored in a CSV file. Then, using numpy libraries the real data is stored. The images are stored as image files. The non-image/scalar data along with corresponding image files(with filenames) are stored in a CSV file.

4.3.2 ROS web bridge

In the figure 4.7, we can see a ROS web bridge is a virtual bridge between scripts using ROS and LGSVL simulator. In this case, a ROS 2 web bridge [64], written in nodejs, is established. It basically starts an instance that listens to an IP address and its port(`http://localhost:9090`). The LGSVL on the other side(defined inside WebUI), listens to this IP address and port. Hence a bridge

is created to allow flow of data.

The sensor data are sent or received through this bridge. The *node.js* bridge creates a nodejs wrapper around the ROS2 messages and follows rosbridge v2.0 protocol [65]. Though this bridge has additional features, for our tasks with LGSVL it merely acts as transport.

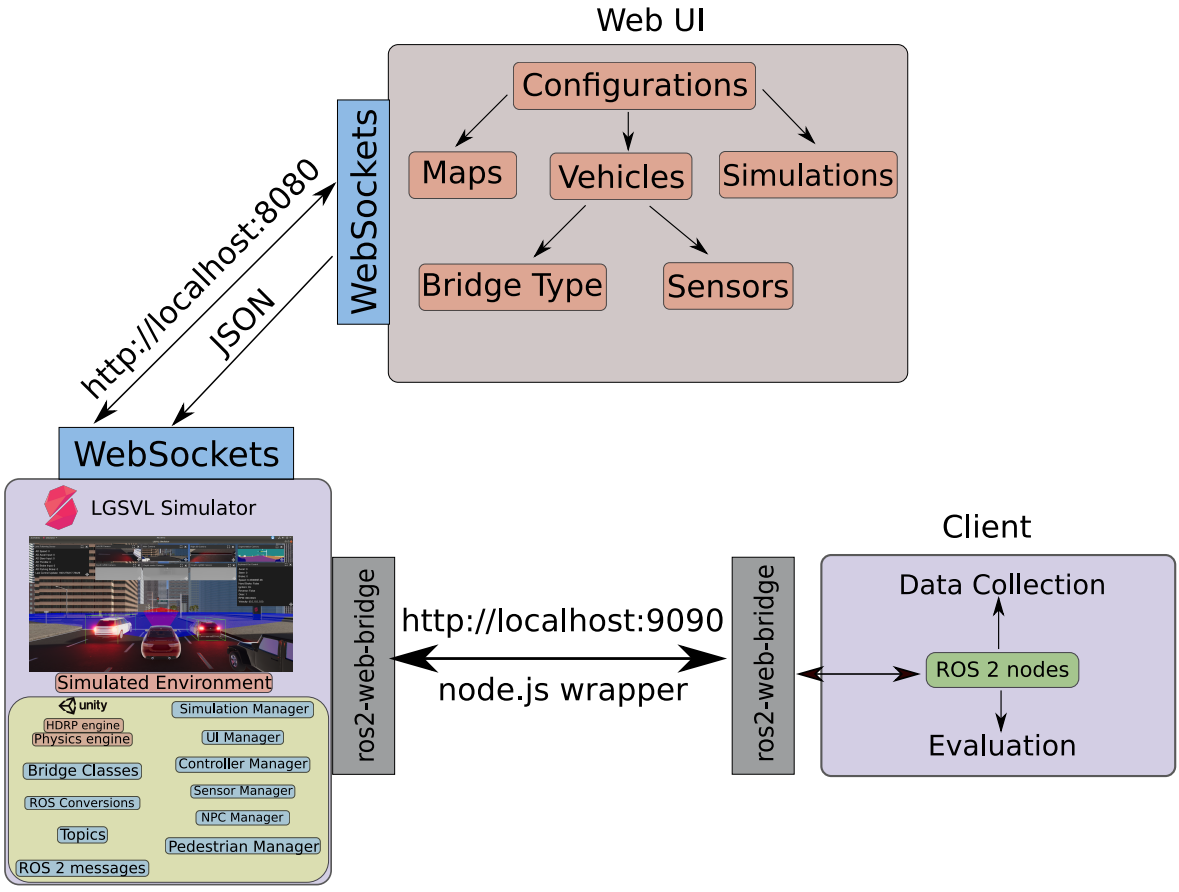

Figure 4.7: ROS2 web bridge implementation

4.4 Training Module

The stored data can't be always used directly for training. Most times it must be preprocessed to user's needs and goals. Input is *X_data* and output *Y_data*. Since we use supervised learning algorithm, the output is known and labelled.

4.4.1 Preprocessing

As shown in figure 4.8, the first task in preprocessing is to select which non-image sensor *Y_data* is necessary for prediction and separate them out into a small text file. Using this file, the images which contain timestamp in the filename are fetched, manipulated using CV2 libraries, stored in arrays and saved in the form of HDF5 files [66]. The Hierarchical Data Format version 5 (HDF5), is an open source file format that supports large, complex, heterogeneous data. Within one HDF5 file, you can store a similar set of data organized in the same way that you might organize files and folders on your computer. It is a compressed format and supports *data slicing* which allows only a part of the dataset to be read and not load all of them in the RAM memory.

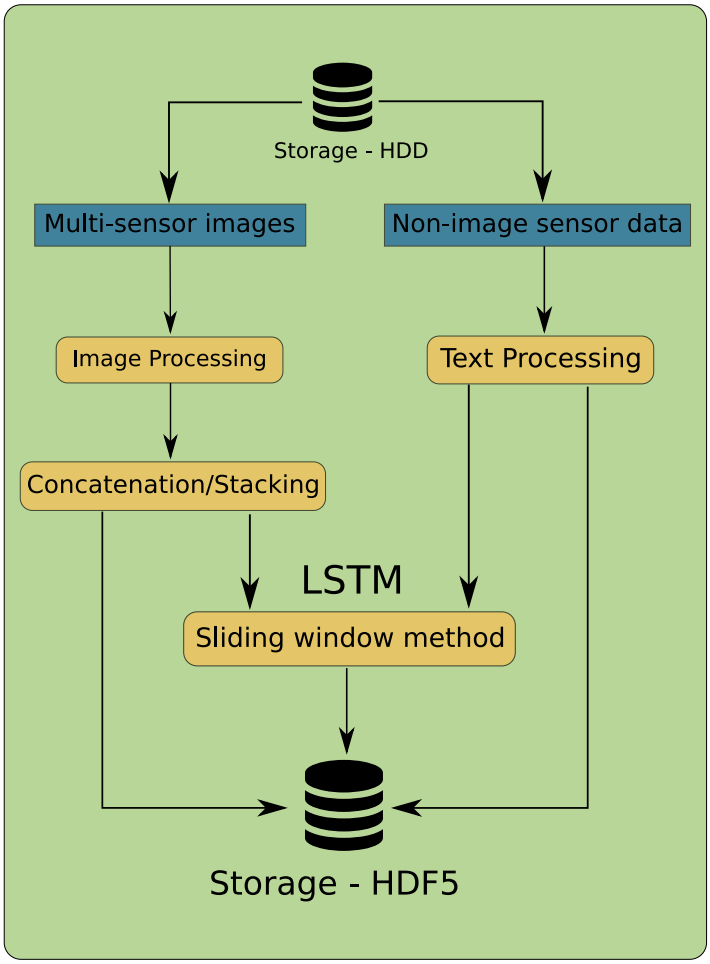

Figure 4.8: Preprocessing module

The images in our case are read either as grayscale or RGB colour images. Then are cropped and resized to a smaller resolution such as 160x70. For grayscale image there is one channel. So the image's dimensions resemble 160x70x1 and for RGB image it has 3 channels which means the dimensions are 160x70x3.

The images from multiple viewpoints or sensors can be fused together making multi-channels. This task will be explained more in data fusion section(4.4.3).

4.4.2 LSTM

LSTM comprises of serially lined up LSTM cells which allow prediction using previous data. Since previous data require data from past, each frame image must be backtracked to a certain, defined time period. This is called *time steps*. According to the time step, the images(frames) are gathered as one and stored. So for a $time\_step = 15$, the dimensions will look like `15x70x160x1` for grayscale images and `15x70x160x3` for RGB images. In the figure 4.9, the time_step acts as a frame window. This window is moved one step to the right for every image file(frame).

So for every frame(image file), its previous 14 frames are stored along with it. However, for *Y_data* only the current frame's output is stored as we should match the current frame with its output.

Another feature we use is restricting how big is the gap between two frames. This is essential because if the consecutive frames are separated by a bigger margin, combining them for previous may lead to unknown problems. The time period we use is 1s.

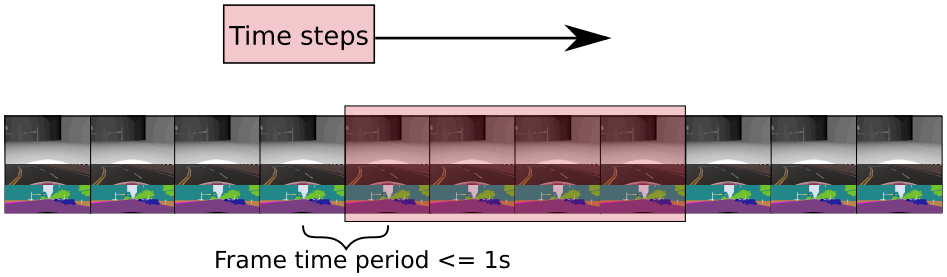

Figure 4.9: Sliding frame window implementation module

4.4.3 Datafusion

Data fusion is one of the primary goals of this thesis. As discussed in fundamentals chapter(2.4), data fusion techniques can be broadly classified into two techniques – early and late fusion. For early fusion, the images from multiple viewpoints or sensors are fused in the preprocessing stage. This fusion is accomplished either by stacking the images or concatenating them. So for example, if a grayscale and RGB images are fused/overlayed together using concatenation, then the dimensions would like `70x160x4` where 4 represents number of channels. These images are usually referred to as *multispectral images*. The figure 2.9 illustrates this approach.

Late fusion on the other hand is done during the training stage of the end-to-end work flow. Usual process involves combining(concatenating) two sources of information after one or two layers of convolution and then using the combined block to do further feature extraction and eventually prediction. Or if the source is of a different modality such as distance or velocity or number of vehicles in front of the ego car, they don't need to be feature extracted like imaged based pixel values as they are scalar values and easy of understand. Hence, it is added after the CNN is completed and the CNN outputs are made into a vector through flattening process. However, it must be remembered that late fusion increases the trainable parameters and costs on resources. The figure 4.10 illustrates one of the late fusion processes.

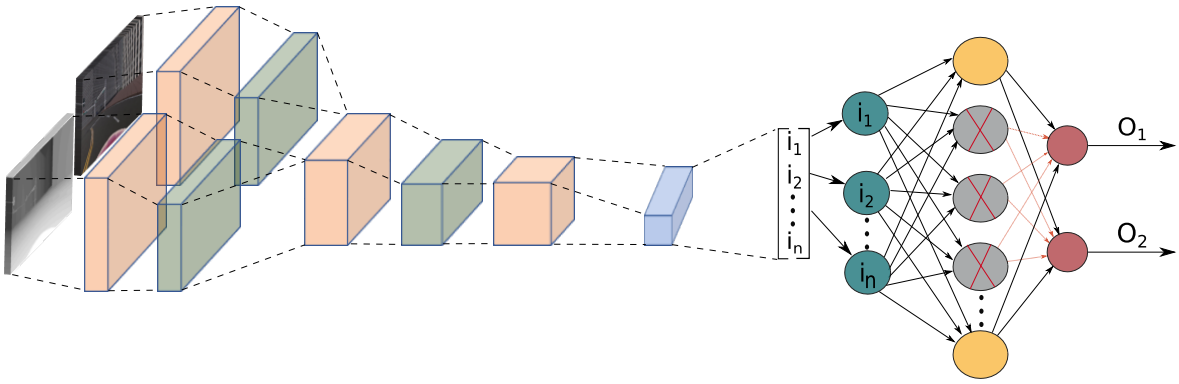

Figure 4.10: Late Fusion

4.4.4 Loading from HDF5 and splitting the dataset

The data stored in HDF5 files in preprocessing are loaded into memory as *X_data* and *Y_data* respectively. Then using scikit-learn module, the *X_data* is then split 80-20 as *X_train* and *X_test* respectively. Similarly *Y_data* as *Y_train* and *Y_test* respectively.

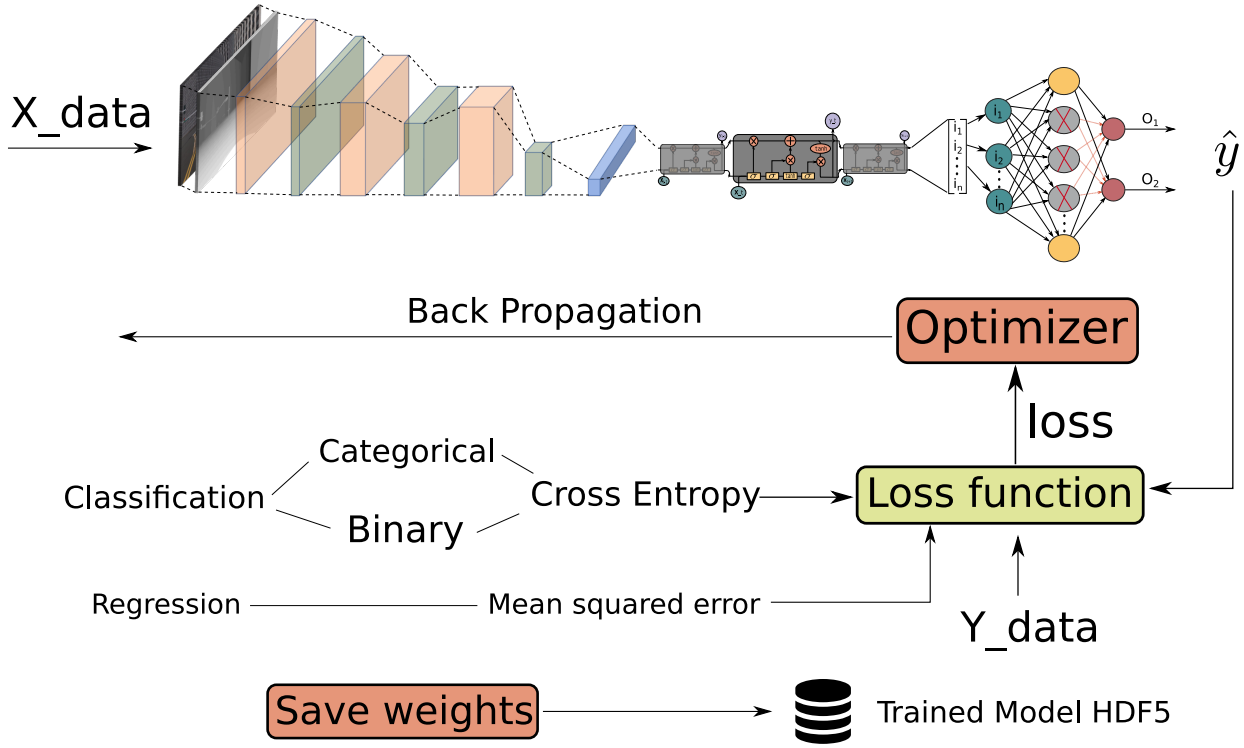

Figure 4.12: Implementation of Training module.

4.4.5 CNN and fully connected layers

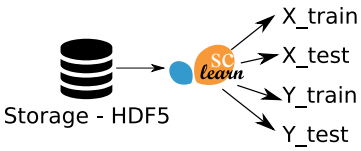

Figure 4.11: Splitting the dataset into train and test data using Sci-kit learn module.

Training a model involves designing a neural network architecture and deciding on its hyperparameters. In this thesis, CNN and dense layers are designed with appropriate activation functions, learning rate, epochs, batch size, CNN specific stride and kernel lengths, optimizer etc.

As shown in the figure 4.12, the CNN layers extract features from images through convolution. The features extracted can be stored inside feature maps. Depending on the input image dimensions, the feature maps' values are adjusted to store as many features as possible.

After extraction, these multi-dimensional data are transformed to 1-D vector through *flatten* process. Scalar, non-image sensor data can be fused at this stage. The vector 1-D data are then fed to fully connected layers which are gradually reduced till the output layer units matches the needed outputs.

The predicted output $\hat{y}$ is compared with the true *Y_train* output. The difference, *loss* is then minimised using an optimizer which does the backpropagation to adjust weights at each layer and node. The best, optimised output model is then stored in a HDF5 file.

4.5 Evaluation Module

The figure 4.13 gives an overview of the control module where the predicted control commands are evaluated. Using docker-compose the ros2-web-bridge and the evaluation are executed. Then at the LGSVL end, the same IP address and port are entered and listened in. A rosbridge is achieved.

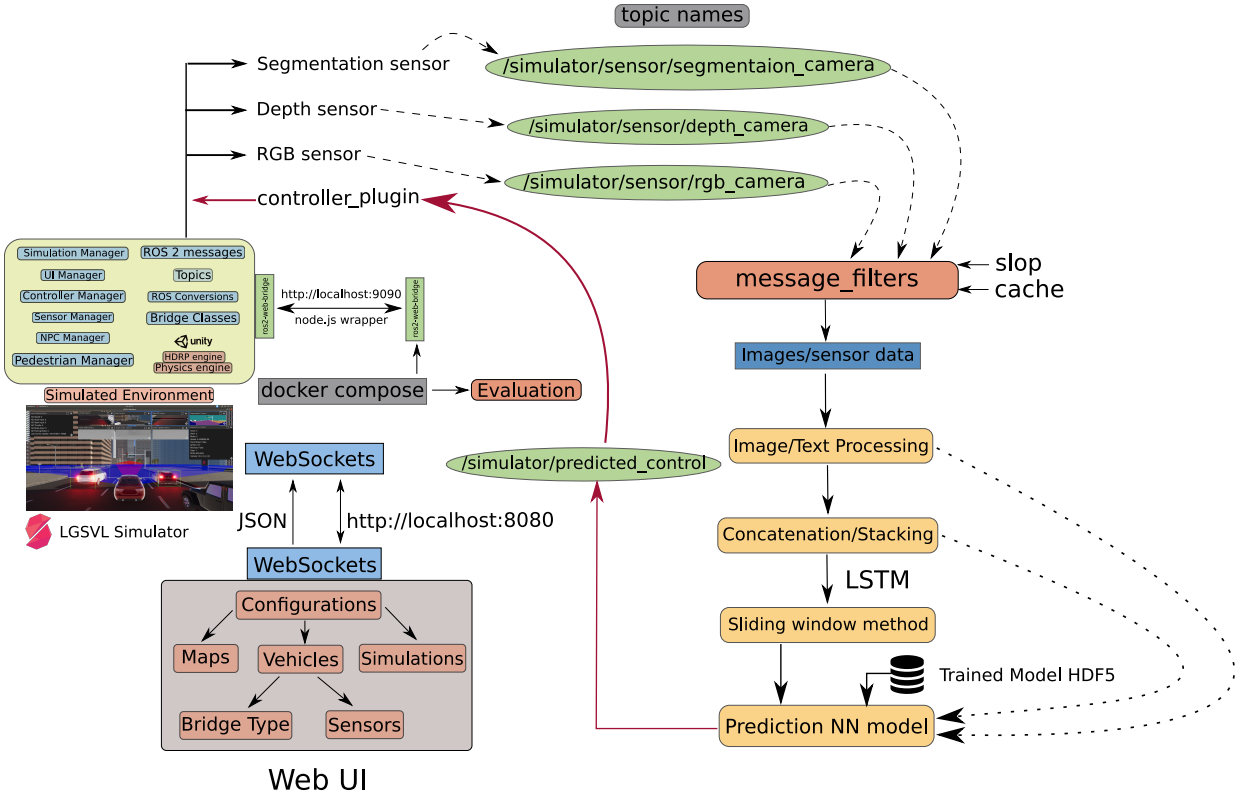

Figure 4.13: Control Module

Evaluation is basically completes the loop of end-to-end training architecture. The LGSVL simulator data are received through ROS bridge and subscriber nodes. With the help of message filters, the messages are collected. Inside the callback, the sensor data is extracted and image manipulation carried out in preprocessing phase, is repeated. The preprocessed image is then fed to the trained model. The model predicts the output which in our case, is control commands. These commands are then assigned and published/sent back to the simulator(sensor plugin) through rosbridge. The custom sensor plugin has a subscribing topic on the LGSVL side. The data sent through rosbridge, is picked up by appropriate data type. The predicted command behaviour is observed and evaluated using appropriate metrics. It is important to remember that, the exact steps followed in preprocessing must be repeated while evaluating. Otherwise, it will lead to inconsistent performance.

5 Evaluation

In this chapter, the workflow explained in last chapter is evaluated and results are presented.

Before showing the evaluation, it is necessary to define training and testing conditions that can be easily used by others to verify the results.

Each evaluation has a setup and its corresponding result.

We have three datasets that can be used for training and evaluation.

1. Dataset 1 - Contains 100,000 raw data. It is collected in no traffic environment, doing straight driving without any sudden turning. The data is using San Francisco map and driven during afternoon. This dataset has only data representing centre camera pointed ahead, parallel to the ground and right camera pointed to the ground at an angle $20°$. The control commands include acceleration, throttle, braking, and steering angle values.

2. Dataset 2 - Also contains 100,000 raw data. It is, however, collect with traffic where the cars stop at signal intersections for a longer time than dataset 3. This dataset is also collect in San Francisco map and during afternoon. It contains a centre camera, right camera like dataset 1, left camera similar to right camera by pointing at an angle $20°$ to the ground, depth camera sensors placed at centre, left and right just like RGB cameras. The control commands are same as dataset 1.

3. Dataset 3 - Contains 270,000 raw data. It is collected while driving around San Francisco. About 200,000 data is collected while driving in the afternoon. About 20,000 in different weather and light conditions. About 50,000 entries are collected in a different circular circuit map called CubeTown. In addition to RGB and depth cameras distributed just as dataset 2, a segmentation camera is kept next centre RGB camera facing forward, and a radar sensor just in front of the car near the hood also facing forward.

5.1 Evaluation setup

While evaluating, a testing parameter *episode* is used. Each episode lasts 30 seconds. A timer is started for 30 seconds and the model is tested for collisions. If a collision happens, the time at which collision happened is noted.

5.1.1 Setup 1 - Determine the best lighting conditions to test the model

All three datasets are used. The test is conducted in San Franciso map without traffic option switched ON. By varying the light conditions to morning, afternoon and evening, we observe how light influences the prediction of output. Only steering angle is predicted and a steady velocity of 3 meter per second is used. An episode length of 30s is used. When a collision is observed, the time of collision and the count are noted down.

time(in 24 hrs standard)	Morning	Afternoon	Evening
	7:35	15:30	18:30

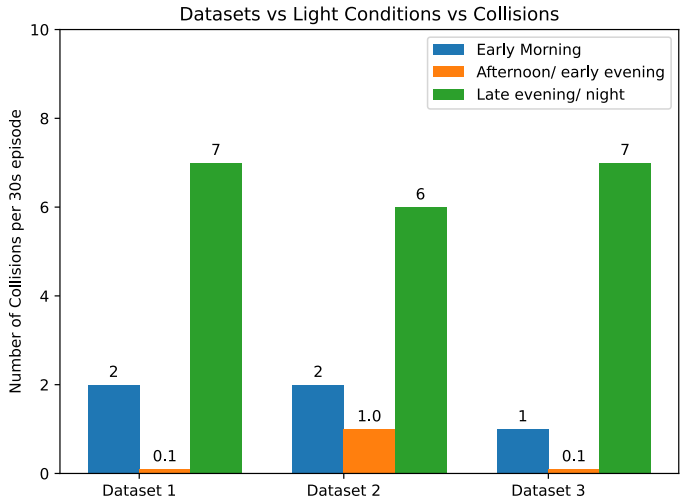

Figure 5.1: Datasets vs Light Conditions vs Number of Collisions

It is seen from figure 5.1 that afternoon time provides the best light conditions for all the three datasets. Dataset 1 and 3 perform equally across the three lighting conditions.

If the percentage of number of collisions, as shown in figure 5.2, is calculated, dataset 3 performs the best among the datasets for morning and afternoon part of the day.

5.1.2 Setup 2 - How the datasets perform during afternoon if traffic is enabled?

All three datasets are again used. The time is fixed at 15:30. The traffic is toggled ON.

From figure 5.3, we can observe that all three datasets do well even in traffic. However, it is surprising to see dataset 1 which had no traffic while the dataset was collected, performs remarkably well when driven in traffic.

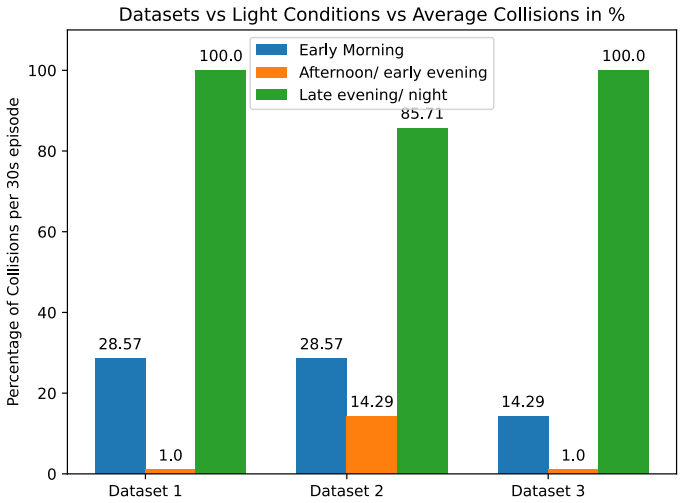

Figure 5.2: Datasets vs Afternoon vs Traffic vs Average number of Collisions

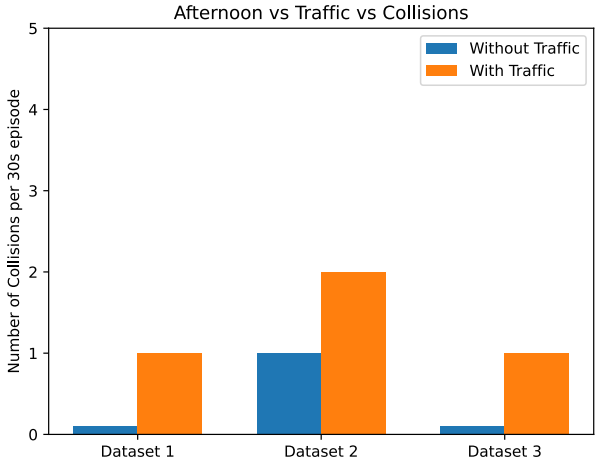

Figure 5.3: Datasets vs Afternoon vs Traffic vs Number of Collisions

6 Conclusion

In this thesis, an end-to-end neural network for autonomous driving with basic framework is implemented. Upon evaluating, it is observed that light conditions play an important role on how well the model predicts a correct decision. In the 5.1.1, it is seen that afternoon time is favourable to get optimal output.

7 Future Work

One of the motivations of this thesis is to see whether it is possible to use LGSVL simulator to implement an end-to-end network to predict control commands by using supervised learning. The results prove that the controlling the car using predicted commands just by using images is possible. However, once the velocity of the car goes beyond 3 meters per second, the steering control becomes erratic. Though the acceleration can be clipped by a percentage, it does not provide an optimal solution. So in future, it could be possible to understand why the acceleration behaves unevenly.

When doing classification based training model, it is observed that the dataset are biased to drive straight without doing anything. Acceleration and braking populate only about 10% of the whole dataset. Hence, a more balanced dataset would help train more classification based loss functions.

As it is seen from the evaluation chapter, the largest dataset has only 270,000 entries and during training the images are resized to smaller dimensions to avoid memory issues. So for the next phase, more computing power would help to train more robust models.

List of Figures

References

[1] Google, “Waymo.” https://waymo.com/

[2] Berkeley University, “Berkeley-deepdrive. [online].” https://deepdrive.berkeley.edu/

[3] Apollo, “Apolloscape dataset.” http://apolloscape.auto/

[4] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105. http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[5] H. Rohling and E. Lissel, “77 ghz radar sensor for car application,” in *Proceedings International Radar Conference*, 1995, pp. 373–379.

[6] D. Lv, X. Ying, Y. Cui, J. Song, K. Qian, and M. Li, “Research on the technology of lidar data processing,” in *2017 First International Conference on Electronics Instrumentation Information Systems (EIIS)*, 2017, pp. 1–5.

[7] A. Carullo and M. Parvis, “An ultrasonic sensor for distance measurement in automotive applications,” *IEEE Sensors Journal*, vol. 1, no. 2, pp. 143–, 2001.

[8] G. Rong, B. H. Shin, H. Tabatabaee, Q. Lu, S. Lemke, M. Možeiko, E. Boise, G. Uhm, M. Gerow, S. Mehta *et al.*, “Lgsvl simulator: A high fidelity simulator for autonomous driving,” *arXiv preprint arXiv:2005.03778*, 2020. https://www.lgsvlsimulator.com/

[9] Nvidia, “Nvidia simulator.” https://www.nvidia.com/en-us/self-driving-cars/drive-constellation/

[10] Carla, “Carla simulator.” https://carla.org/

[11] IPG-Automotive, “Carmaker- simulator.” https://ipg-automotive.com/products-services/simulation-software/carmaker/

[12] Open Robotics, “Robotic operating system.” https://index.ros.org/doc/ros2/

[13] Seth Lambert and Erika Granath, “Lidar systems: costs, integration, and major manufacturers.” https://www.mes-insights.com/lidar-systems-costs-integration-and-major-manufacturers-a-908358/

[14] Z. Chen and X. Huang, “End-to-end learning for lane keeping of self-driving cars,” in *2017 IEEE Intelligent Vehicles Symposium (IV)*, 2017, pp. 1856–1860.

[15] M. Bojarski, D. D. Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba, “End to end learning for self-driving cars,” 2016.

[16] S. Levine, P. Pastor, A. Krizhevsky, and D. Quillen, “Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection,” *The International Journal of Robotics Research*, 03 2016.

[17] H. Xu, Y. Gao, F. Yu, and T. Darrell, "End-to-end learning of driving models from large-scale video datasets," *CoRR*, vol. abs/1612.01079, 2016. http://arxiv.org/abs/1612.01079

[18] J. Kim and J. F. Canny, "Interpretable learning for self-driving cars by visualizing causal attention," *CoRR*, vol. abs/1703.10631, 2017. http://arxiv.org/abs/1703.10631

[19] M. Bojarski, P. Yeres, A. Choromanska, K. Choromanski, B. Firner, L. D. Jackel, and U. Muller, "Explaining how a deep neural network trained with end-to-end learning steers a car," *CoRR*, vol. abs/1704.07911, 2017. http://arxiv.org/abs/1704.07911

[20] Z. Yang, Y. Zhang, J. Yu, J. Cai, and J. Luo, "End-to-end multi-modal multi-task vehicle control for self-driving cars with visual perceptions," pp. 2289–2294, 08 2018.

[21] K. Liu, Y. Li, N. Xu, and P. Natarajan, "Learn to combine modalities in multimodal deep learning," 2018.

[22] E. Park, X. Han, T. L. Berg, and A. C. Berg, "Combining multiple sources of knowledge in deep cnns for action recognition." in *WACV*. IEEE Computer Society, 2016, pp. 1–8. http://dblp.uni-trier.de/db/conf/wacv/wacv2016.html#ParkHBB16

[23] Y. Zhou and K. Hauser, "Incorporating side-channel information into convolutional neural networks for robotic tasks," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, 2017, pp. 2177–2183.

[24] W. Wang, D. Tran, and M. Feiszli, "What makes training multi-modal classification networks hard?" in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 12 695–12 705.

[25] Y. Xiao, F. Codevilla, A. Gurram, O. Urfalioglu, and A. M. López, "Multimodal end-to-end autonomous driving," *CoRR*, vol. abs/1906.03199, 2019. http://arxiv.org/abs/1906.03199

[26] F. Codevilla, M. Mller, A. Lopez, V. Koltun, and A. Dosovitskiy, "End-to-end driving via conditional imitation learning," in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, 2018, pp. 4693–4700.

[27] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[28] T. Mitchell, "Machine learning," *McCraw Hill*, 1996.

[29] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, p. 100, http://www.deeplearningbook.org.

[30] K. P. Murphy, *Machine learning: a probabilistic perspective*, Cambridge, MA, 2012, p. 3.

[31] Canadian Institute for Advanced Research, "Cifar 10 image dataset." https://www.cs.toronto.edu/~kriz/cifar.html

[32] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014.

[33] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *J. Mach. Learn. Res.*, vol. 12, no. null, p. 21212159, jul 2011.

[34] G. Hilton, "RMSProp - Unpublished method from coursera lecture," 2012. http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

[35] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural Networks*, vol. 61, p. 85117, Jan 2015. http://dx.doi.org/10.1016/j.neunet.2014.09.003

[36] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 06 2014.

[37] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE Transactions on Neural Networks*, vol. 5, no. 2, pp. 157–166, 1994.

[38] S. Hochheiter and J. Schmidhuber, "Untersuchungen zu dynamischen neuronalen netzen. masters thesis, technische universitaet muenchen," *Diplomarbeit*, June 1991. http://people.idsia.ch/~juergen/SeppHochreiter1991ThesisAdvisorSchmidhuber.pdf

[39] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997. https://doi.org/10.1162/neco.1997.9.8.1735

[40] B. Cain, "Passive Stereo Camera." https://brenn10.github.io/MIT_experience_paper/index.html

[41] U. Hahne and M. Alexa, "Combining time-of-flight depth and stereo images without accurate extrinsic calibration," *Int. J. Intell. Syst. Technol. Appl.*, vol. 5, no. 3/4, p. 325333, nov 2008. https://doi.org/10.1504/IJISTA.2008.021295

[42] A. Lipnickas and A. Kny, "A stereovision system for 3-d perception," *Elektronika ir Elektrotechnika*, pp. 1392–1215, 04 2009.

[43] R. P. K. Poudel, S. Liwicki, and R. Cipolla, "Fast-scnn: Fast semantic segmentation network," *CoRR*, vol. abs/1902.04502, 2019. http://arxiv.org/abs/1902.04502

[44] B. Khaleghi, A. Khamis, F. O. Karray, and S. N. Razavi, "Multisensor data fusion: A review of the state-of-the-art," *Inf. Fusion*, vol. 14, no. 1, p. 2844, jan 2013. https://doi.org/10.1016/j.inffus.2011.08.001

[45] V. Malyavej, W. Kumkeaw, and M. Aorpimai, "Indoor robot localization by rssi/imu sensor fusion," in *2013 10th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology*, 2013, pp. 1–6.

[46] Y. Dobrev, S. Flores, and M. Vossiek, "Multi-modal sensor fusion for indoor mobile robot pose estimation," in *2016 IEEE/ION Position, Location and Navigation Symposium (PLANS)*, 2016, pp. 553–556.

[47] D. Iter, J. Kuck, and P. Zhuang, "Target tracking with kalman filtering, knn and lstms," *proj2016*, 2016. http://cs229.stanford.edu/proj2016/report/IterKuckZhuang-TargetTrackingwithKalmanFilteringKNNandLSTMs-report.pdf

[48] H. Coskun, F. Achilles, R. DiPietro, N. Navab, and F. Tombari, "Long short-term memory kalman filters:recurrent neural estimators for pose regularization," 2017.

[49] G. Ning, Z. Zhang, C. Huang, X. Ren, H. Wang, C. Cai, and Z. He, "Spatially supervised recurrent convolutional neural networks for visual object tracking," in *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2017, pp. 1–4.

[50] S. Bohez, T. Verbelen, E. De Coninck, B. Vankeirsbilck, P. Simoens, and B. Dhoedt, "Sensor fusion for robot control through deep reinforcement learning," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2017, pp. 2365–2370.

[51] Google, "Tensorflow, a open source ml framwork." https://www.tensorflow.org/

[52] Google, "Keras, a high level python wrapper for tensorflow." https://keras.io/

[53] Facebook, "Pytorch, a open source ml framework." https://pytorch.org/

[54] Open Robotics, "About ros." https://www.ros.org/about-ros/

[55] E. Ackerman and E. Guizzo, "Wizards of ros: Willow garage and the making of the robot operating system." https://spectrum.ieee.org/automaton/robotics/robotics-software/wizards-of-ros-willow-garage-and-the-making-of-the-robot-operating-system

[56] B. Gerkey, "Why ros2?" http://design.ros2.org/articles/why_ros2.html

[57] Open Robotics, "Rosbridge suite." http://wiki.ros.org/rosbridge_suite

[58] Open Robotics, "Message filters." http://wiki.ros.org/message_filters

[59] Docker, Inc., "Docker - how to get started." https://docs.docker.com/get-started/

[60] LG SVL team, “Lgsvl simulator github page.” https://github.com/lgsvl/simulator

[61] Baidu, “Baidu-apollo.” https://github.com/ApolloAuto/apollo

[62] S. Kato, E. Takeuchi, Y. Ishiguro, Y. Ninomiya, K. Takeda, and T. Hamada, “An open approach to autonomous vehicles,” *IEEE Micro*, vol. 35, no. 6, pp. 60–68, 2015.

[63] Unity Technologies, “Unity.” https://unity.com/

[64] RobotWebTools Team, “ros2-web-bridge.” https://github.com/RobotWebTools/ros2-web-bridge

[65] RobotWebTools Team, “rosbridge v2.0 Protocol Specification.” https://github.com/RobotWebTools/rosbridge_suite/blob/develop/ROSBRIDGE_PROTOCOL.md

[66] HDF Group, “The HDF5 Library & File Format.” https://www.hdfgroup.org/solutions/hdf5