1 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.

1.1 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.

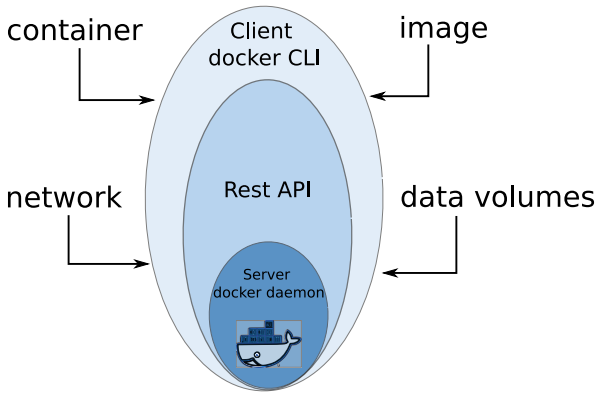


Figure 1.1: Docker Engine and its functions

A docker architecture, as shown in **??**, 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 1.2, 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.

1.2 LGSVL simulator

The LGSVL simulator is developed using Unity engine which is written in C# language. The LGSVL team organises their code base[1] 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.

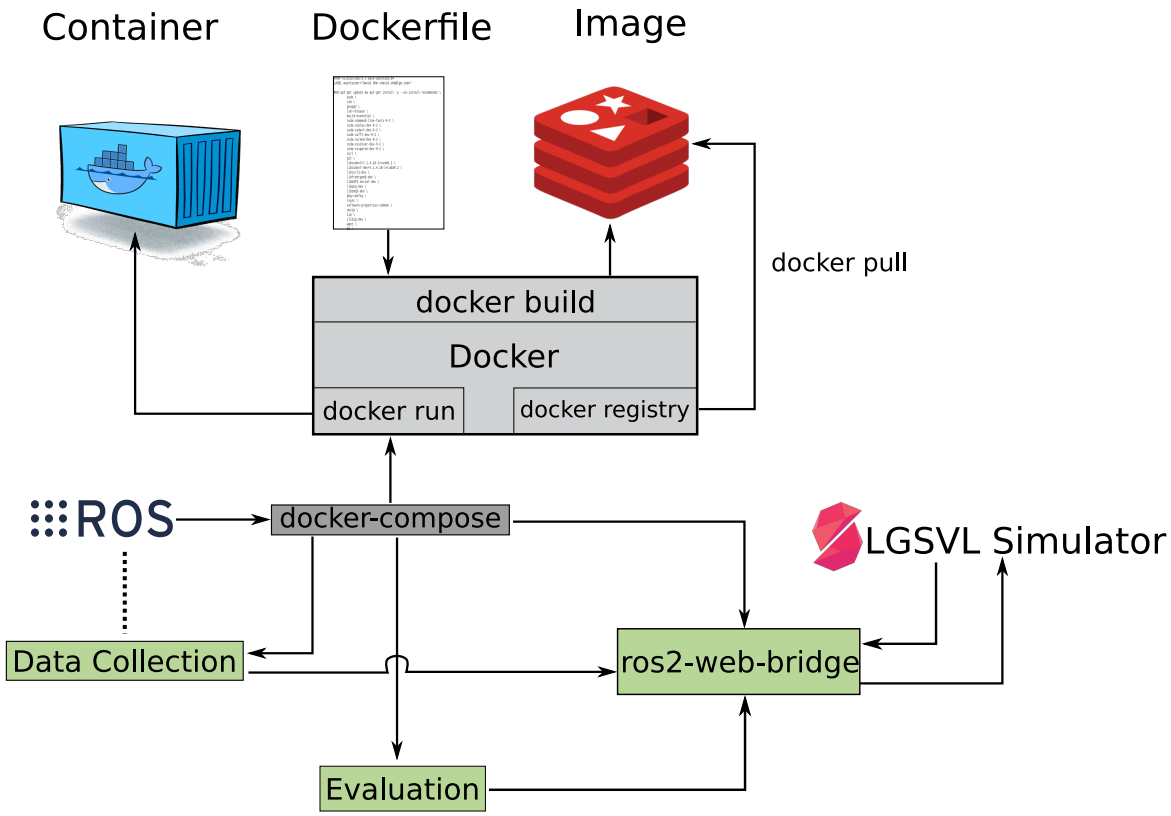


Figure 1.2: Docker and its various functions

1.2.1 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.

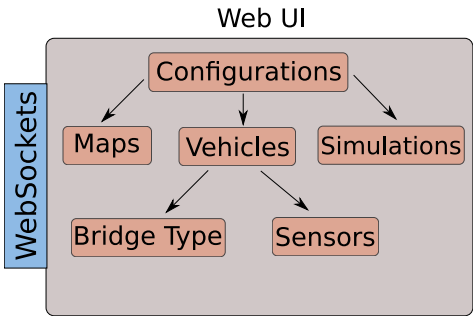


Figure 1.3: LGSVL Simulator - WebUI

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.

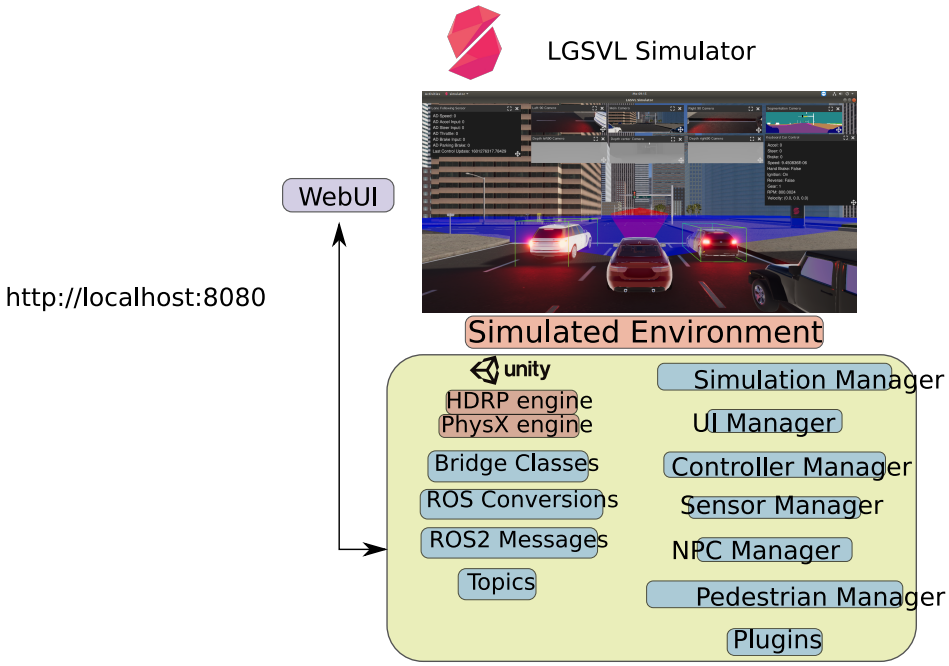


Figure 1.4: LGSVL software architecture

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.

1.2.2 Inside LGSVL simulator framework

The figure 1.4 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.

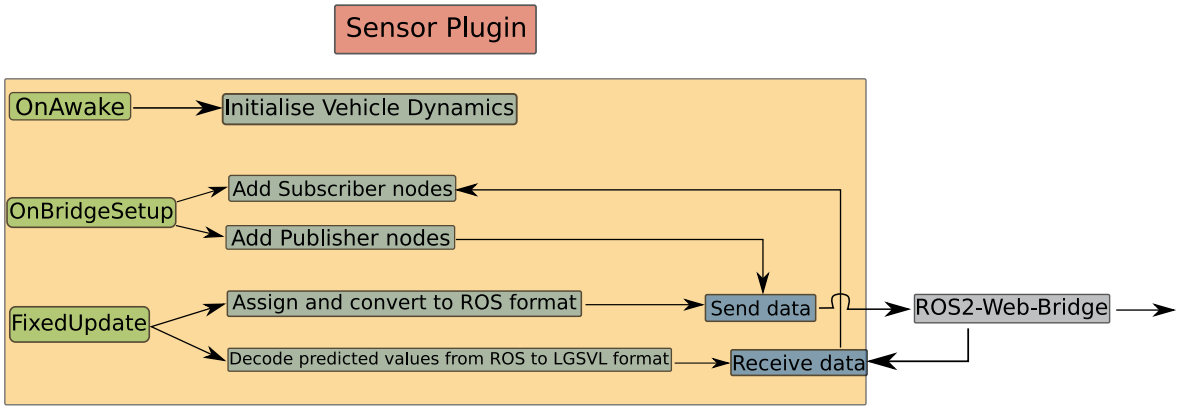


Figure 1.5: Inside sensor plugin

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.

1.2.3 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 1.5 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.

1.3 Data Collection Module

The figure 1.6 gives an overview of the data collection task of sensor module.

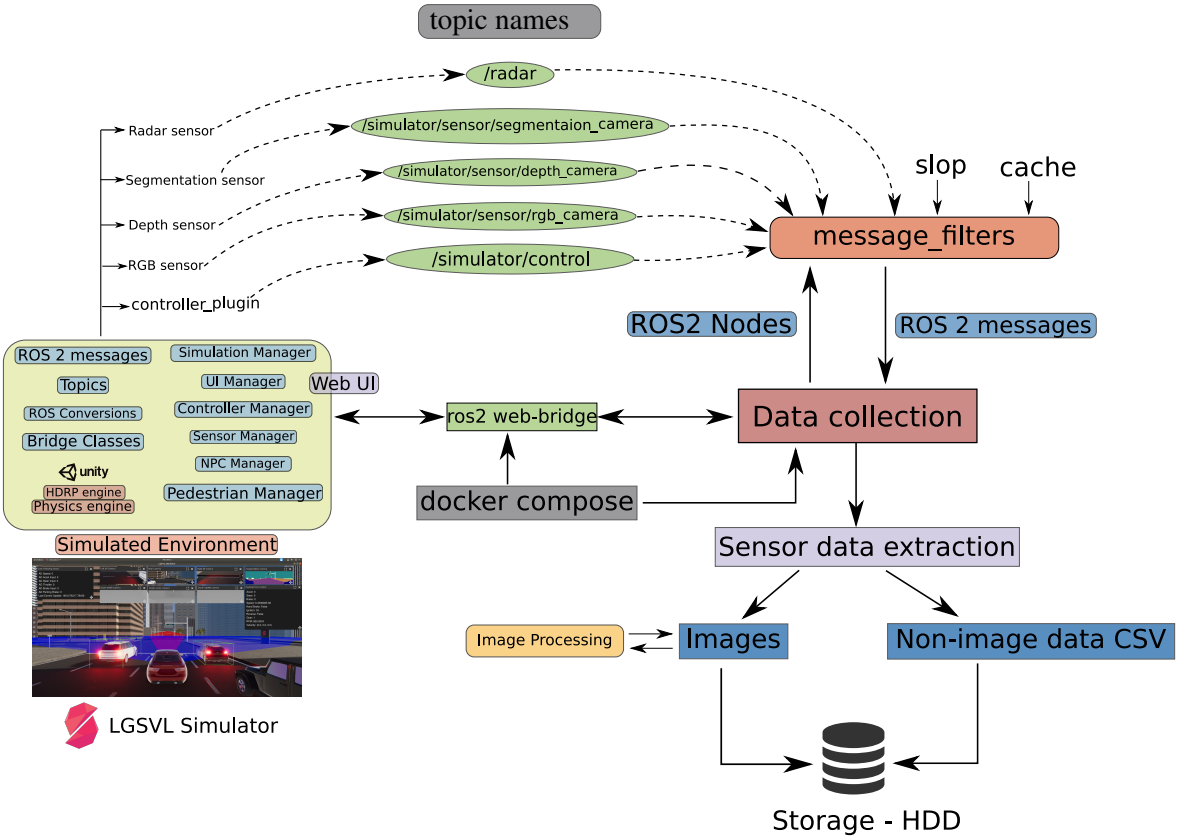


Figure 1.6: A detailed summary of data collection module

1.3.1 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.

1.3.2 ROS web bridge

In the figure 1.7, 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 [2], 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 [3]. Though this bridge has additional features, for our tasks with LGSVL it merely acts as transport.

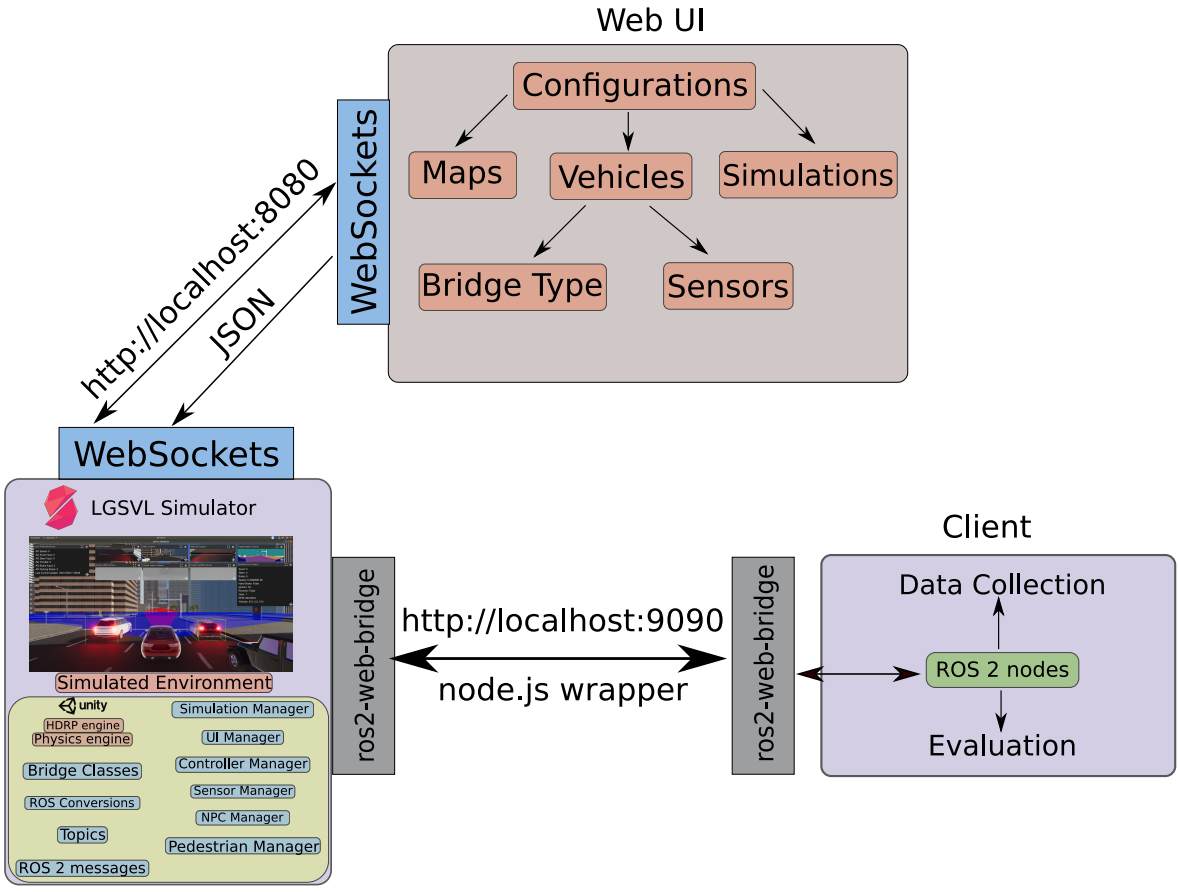


Figure 1.7: ROS2 web bridge implementation

1.4 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.

1.4.1 Preprocessing

As shown in figure 1.8, 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 [4]. 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.

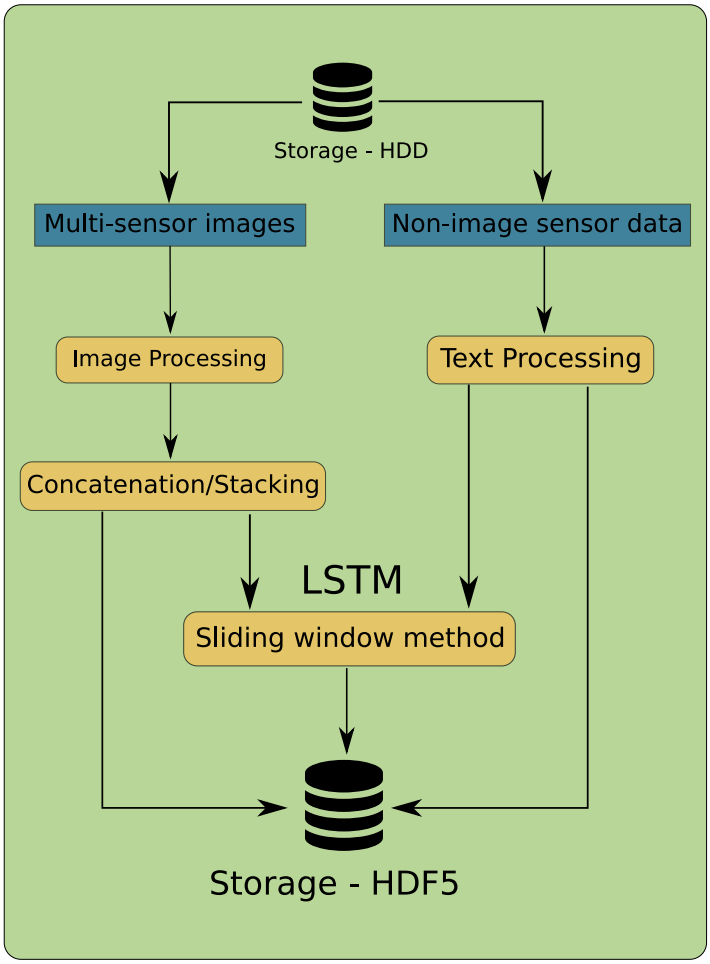


Figure 1.8: Preprocessing module

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(1.4.3).

1.4.2 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 1.9, 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.

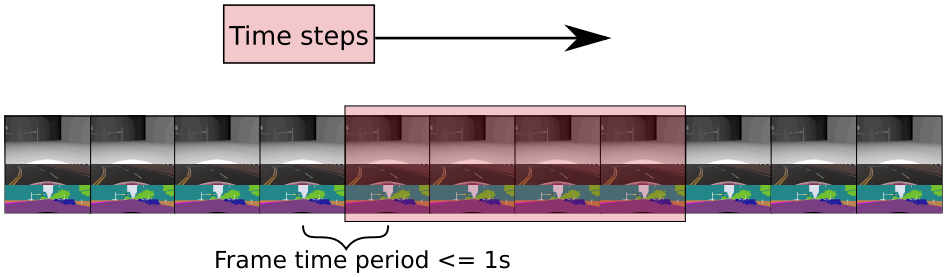


Figure 1.9: Sliding frame window implementation module

1.4.3 Datafusion

Data fusion is one of the primary goals of this thesis. As discussed in fundamentals chapter(**??**), 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 **??** 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 1.10 illustrates one of the late fusion processes.

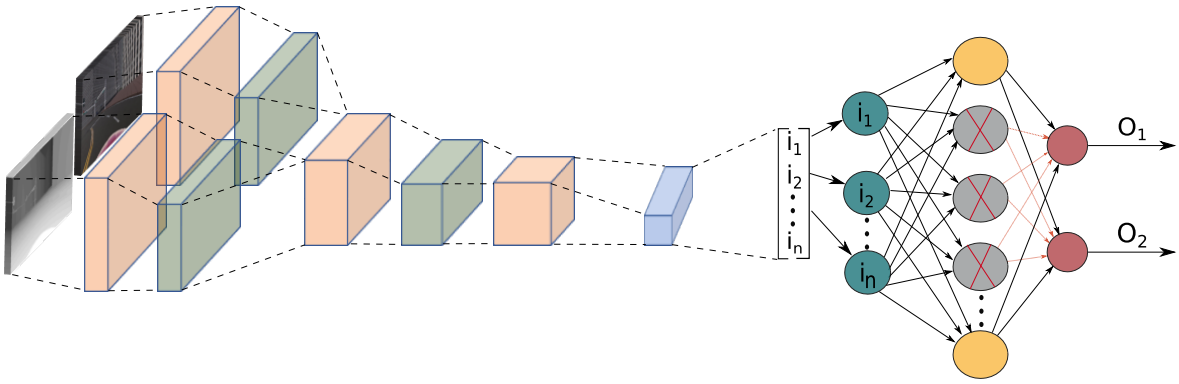


Figure 1.10: Late Fusion

1.4.4 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.

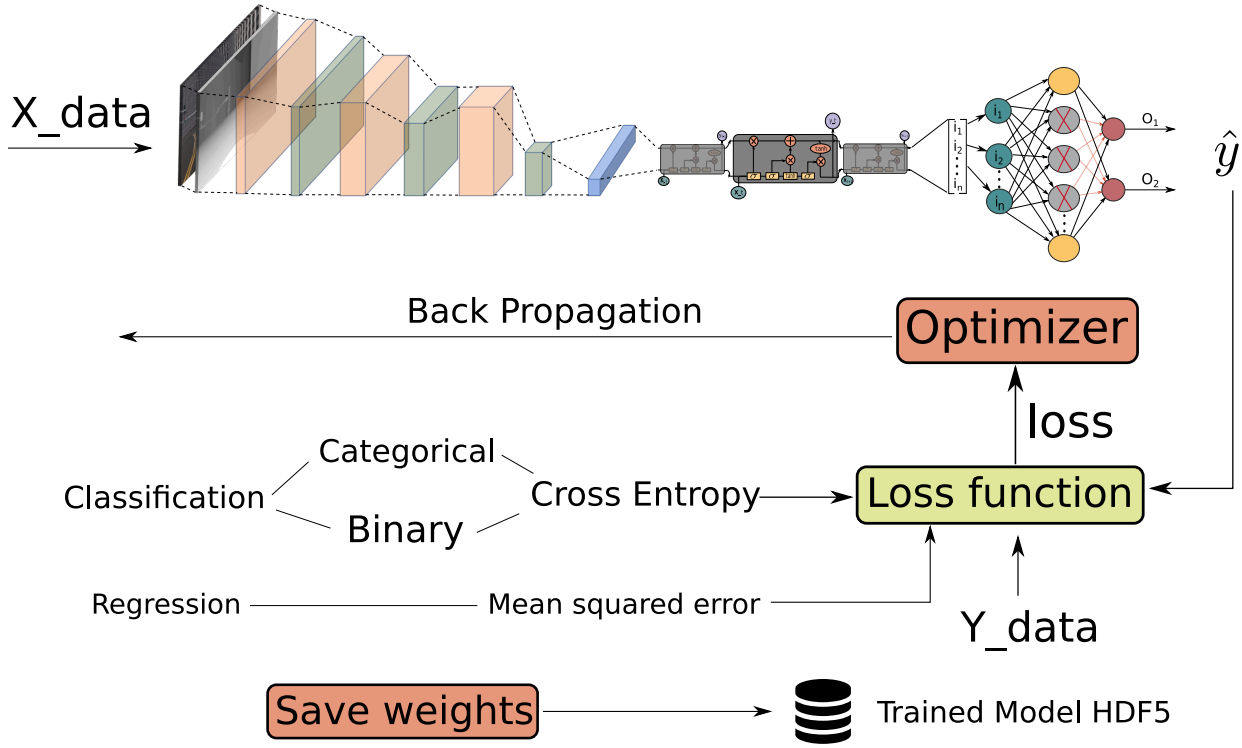


Figure 1.12: Implementation of Training module.

1.4.5 CNN and fully connected layers

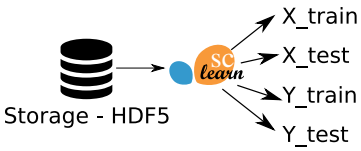


Figure 1.11: Splitting the dataset into train and test data using Sci-kit learn module.

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 1.12, 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.

1.5 Evaluation Module

The figure 1.13 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.

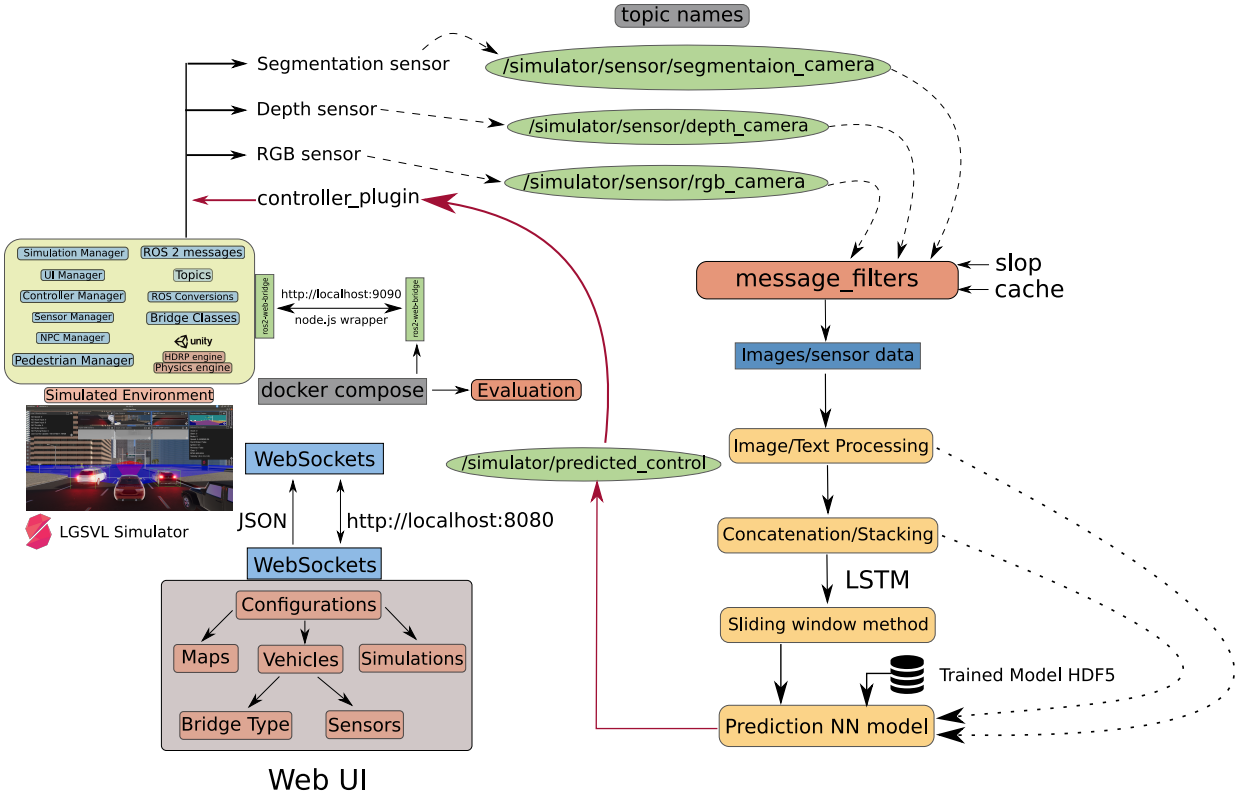


Figure 1.13: Control Module

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.

2 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.

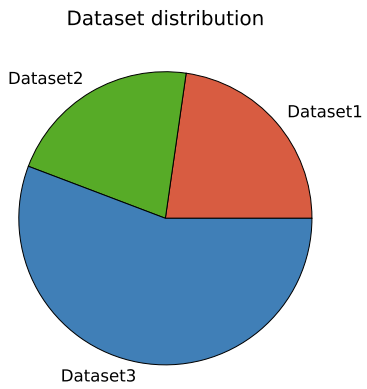


Figure 2.1: Datasets distribution

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

1. Dataset 1 - Contains 100,000 raw data as seen in figure 2.1. 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, collected 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 to centre RGB camera facing forward, and a radar sensor just in front of the car near the hood also facing forward.

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

Table 2.1: Time of the day

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.

As supervised learning is used, the models have to be tested/validated with unknown data to determine its capability. Hence the datasets are split 80-20. Meaning 80% is train data and 20% validation data. The optimizer *Adam* takes the 20% data to test the trained model. Training data leads to training loss and test data to validation loss.

Also till a single dataset is chosen, all datasets are of equal data entries.

2.1 Determine which datasets and best lighting conditions to test the model

All three datasets are used. The test is conducted in San Francisco 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(see table 2.1).

The steering angle is a continuous value ranging between -1 and 1(negative values to turn left and positive values to turn right), a continuous loss function has to be used. Because of that *mean square error*(MSE) as loss function is chosen. 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 number of collisions are noted down.

It is seen from figure 2.2(a) 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 with traffic toggled ON, as shown in figure 2.2(c), is calculated, dataset 3 performs the best among the datasets for morning and afternoon part of the day.

2.1.1 Datasets performance 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 2.2(b), 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.

2.1.2 Observations

1. All 3 datasets do well at afternoon time of the day.
2. Predicting only steering angle with MSE as loss function works as seen from 2.2.
3. Dataset 2 shows higher number of collisions. So it is better to avoid for further analysis.

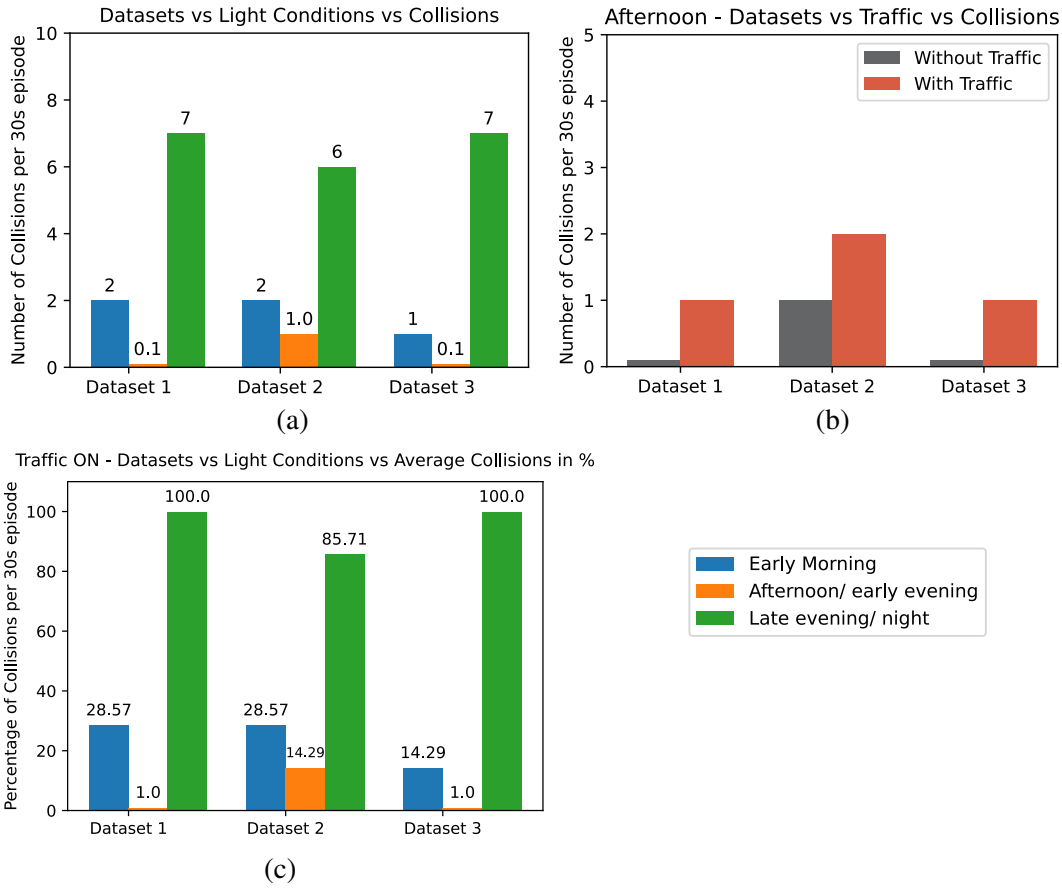


Figure 2.2: a) Datasets vs Light Conditions vs Collisions. b) Afternoon - Datasets vs Traffic vs Number of Collisions. c) Average number of collisions in percentage

2.2 Acceleration - Determine which activation and loss functions to use

2.2.1 Tanh as activation and MSE as loss functions

Since acceleration and steering values in LGSVL range from *-1* to *1*, *tanh* activation function is selected as the output dense layer activation function.

A set of criteria are listed and the trained model is evaluated based on these conditions. From table 2.2 it can be observed that dataset 1 outperforms dataset 3 in most of the conditions. Dataset 1 retains good steering control at high speeds and turning, but acceleration skews steering angle when traffic is switched ON. Dataset 3 when evaluated stops completely after moving a few metres. This causes difficulty in evaluating according to the criteria. Hence it is assumed that this dataset fails to meet the criteria.

One of the reasons as to why dataset 3 fails could be because the losses are much higher than dataset 1 and the validation loss starts to overfit too quickly in the training(see figure 2.3).

2.2.2 Sigmoid as activation and MSE as loss functions

The acceleration values are split into positive and negative values. Instead of negative values an another variable we will call as *braking* is introduced. Negative acceleration values mean braking is active. Using this knowledge, sigmoid as activation function and mean square error as loss function, a training is conducted for both datasets 1 and 3.

Criteria(Tanh/MSE)	Dataset 1	Dataset 3
Lane keeping/Drive straight	Yes	No
Gradual acceleration increase	Yes	No
Smooth braking behaviour observed	Yes	No
Smooth steering control at high speed(10m/s)	Yes	No
Smooth steering control at turnings	Yes	No
Detects traffic as dynamic objects	Yes	No
Navigates traffic smoothly	No	No
Doesn't stop at random places	No	No

Table 2.2: Tanh/MSE - How the model evaluates to different criteria

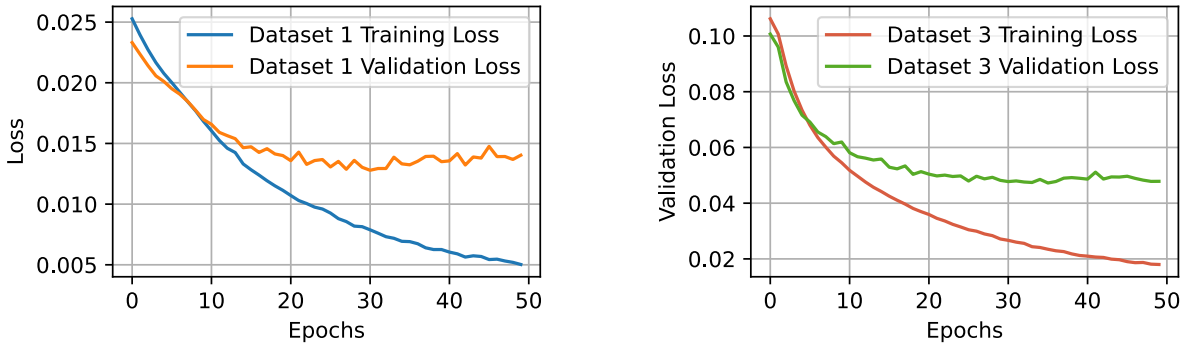


Figure 2.3: Datasets 1 vs 3 - Acceleration and Steering using Tanh activation and MSE loss functions.

Criteria similar to table 2.2 are put to test with this activation and loss function. As seen in table 2.3, both datasets fail to meet the conditions. During evaluation, both datasets trained models, accelerate to a huge velocity such as 40m/s and the steering cannot keep up, resulting in collisions.

Looking into their losses (see figure 2.4), don't really explain much. Though this needs more investigation, for now we assume this activation or loss function is not suitable. 2.3.

Criteria(Sigmoid/MSE)	Dataset 1	Dataset 3
Lane keeping/Drive straight	No	No
Gradual acceleration increase	No	No
Smooth braking behaviour observed	No	No
Smooth steering control at high speed(10m/s)	No	No
Smooth steering control at turnings	No	No
Detects traffic as dynamic objects	No	No
Navigates traffic smoothly	No	No
Doesn't stop at random places	No	No

Table 2.3: Sigmoid/MSE - How the model evaluates to different criteria

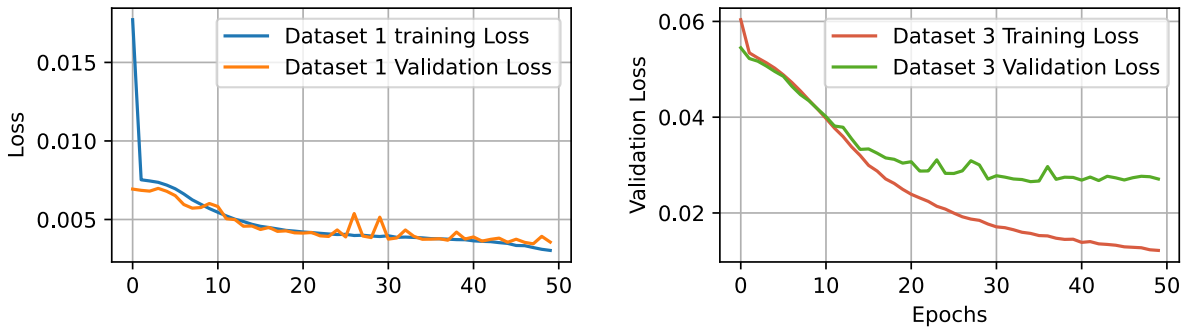


Figure 2.4: Datasets 1 vs 3 - Acceleration and Steering using Sigmoid activation and MSE loss functions.

2.2.3 Softmax as activation and Binary crossentropy as loss functions

Both tanh and sigmoid activation functions couldn't give stable results to continue pursuing with those parameters. Hence it is necessary to consider other functions which may suit our needs.

Since acceleration are basically two discrete values, it would be worthy to try the training as classification task. The goal of a classification task model is to classify to which category the prediction belongs to. In our case, acceleration or braking. Hence *softmax* activation function is needed. As loss function *binary crossentropy* is used to classify as binary classes. Of course for steering angle, being continuous, MSE is preferred.

From table 2.4, both datasets don't do well. Dataset 1 doesn't start at all as braking class dominates the evaluation whereas dataset 3 starts to move the car but resorts to brake indefinitely after a few metres of driving. This behaviour necessitates further analysis into the datasets.

Criteria(Softmax/Binary crossentropy)	Dataset 1	Dataset 3
Lane keeping/Drive straight	No	Yes
Gradual acceleration increase	No	No
Smooth braking behaviour observed	No	No
Smooth steering control at high speed(10m/s)	No	No
Smooth steering control at turnings	No	No
Detects traffic as dynamic objects	No	No
Navigates traffic smoothly	No	No
Doesn't stop at random places	No	No

Table 2.4: Softmax/Binary crossentropy - How the model evaluates to different criteria

Control commands distribution

When the datasets are analysed for patterns of different states – acceleration and braking, distribution chart(figure 2.5) reveals that in addition to acceleration and braking states, there is a third state called *no action* where the vehicle does absolutely nothing. In fact this state dominates in both datasets. Because of this, the binary crossentropy obviously fails to meet the conditions.

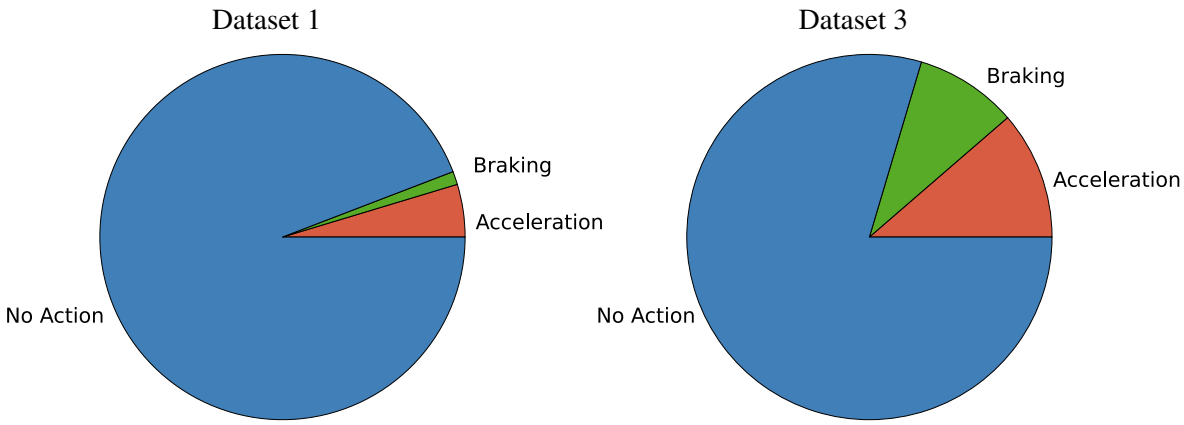


Figure 2.5: Datasets 1 vs 3 control commands distribution

2.2.4 Softmax as activation and Categorical crossentropy loss functions

So now we know that this dominant state *no action* needs a separate label if classification task has to be continued. After creating the label, we would then have three labels – acceleration, braking and no action. Hence, we use a new classification loss function called *categorical crossentropy*. This loss function classifies model into each category.

Criteria(Softmax/Categorical crossentropy)	Dataset 1	Dataset 3
Lane keeping/Drive straight	No	Yes
Gradual acceleration increase	Yes	Yes
Smooth braking behaviour observed	No	Yes
Smooth steering control at high speed(10m/s)	No	No
Smooth steering control at turnings	No	No
Avoids colliding into static objects	No	No
Detects traffic as dynamic objects	Yes	Yes
Navigates traffic smoothly	No	No
Doesn't stop at random places	No	No

Table 2.5: Softmax/Categorical crossentropy - How the model evaluates to different criteria

From table 2.5, dataset 1 fails in most conditions and dataset 3 though performs well in 4 out of 9 conditions, shows bad steering behaviour at traffic or high speeds. If the acceleration is controlled manually, the model reacts better and adapts itself.

Since no action state dominates, it is necessary to continue as a classification task. Also dataset 1 has only a small portion for acceleration and even smaller for braking. Accordingly, dataset 3 is collected with an attempt to increase acceleration and braking values share. However, since the vehicle most times has to drive straight, *no action* state even dominates in dataset 3.

The important condition in a classification task is to have balanced classes. Unfortunately in our case, this balance is not achieved. With this limitation, the evaluation is carried forward.

2.2.5 Observations

1. Steering angle uses tanh activation and MSE loss functions

2. Classify acceleration prediction as classification task which means softmax as activation.
3. Since acceleration carries three states, categorical crossentropy as loss function.
4. Dataset 3 has more of acceleration and braking states than dataset 1. So dataset 3 is preferred.

2.3 Predicting acceleration - categorical crossentropy

Now that the basic criteria for training is fixed such as which dataset, activation, and loss functions, we can move ahead and optimise the predicted classes by tuning the neutral network.

LSTM vs Non-LSTM

Before going into tuning the neural network, it is necessary to tell that acceleration prediction needs temporal information; meaning decision to drive slower or faster depends on the previous, historical frames. LSTM is used for this purpose. When non-LSTM model is used to predict acceleration, it only predicts for the current frame(doesn't provide past frames information) which often results in vehicle being stationary. For our setup, we choose a $timestep = 15$. That means acceleration of current time

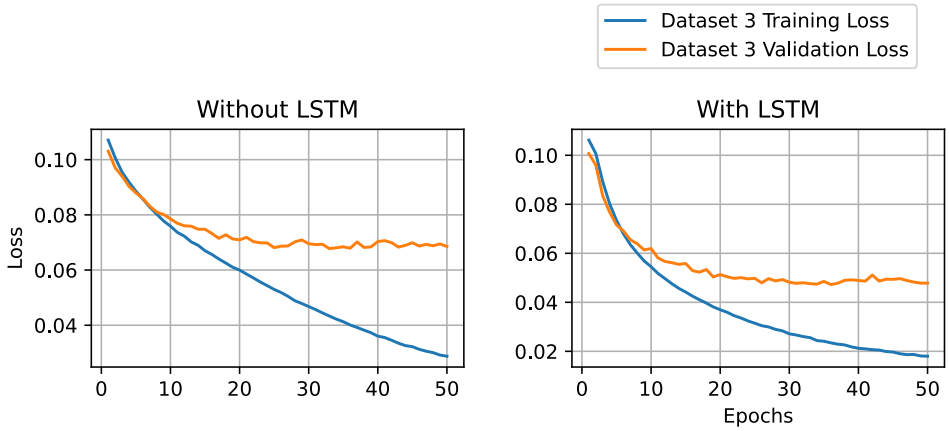


Figure 2.6: Datasets 3 - No LSTM vs LSTM comparison

frame is predicted using previous 14 time frames.

Determining the optimal LSTM output units

In Keras, the LSTM units refer to the dimension of hidden state vector *h* that is the state output from RNN cell. The table 2.6, compares different unit values and the number of trainable parameters(weight that can be trained during backpropagation) at this LSTM layer. In our case, it means that for a time series of 15, there will be 15 *cell states*, 15 *hidden states*, and 15 *outputs* each of vector size defined by the units in the table such as 20, 60 or 100.

Upon evaluation with these different units, 100 output units though has the highest trainable parameters for this layer alone, retains more information needed for training the model. Hence, a LSTM unit of 100 is chosen.

LSTM Output Units	Trainable Parameters(ca.)	Processing time needed
20	20000	1hr 44m
60	61000	1hr 42m
100	434000	1hr 40m

Table 2.6: LSTM Output Units vs Trainable Parameters vs Training time

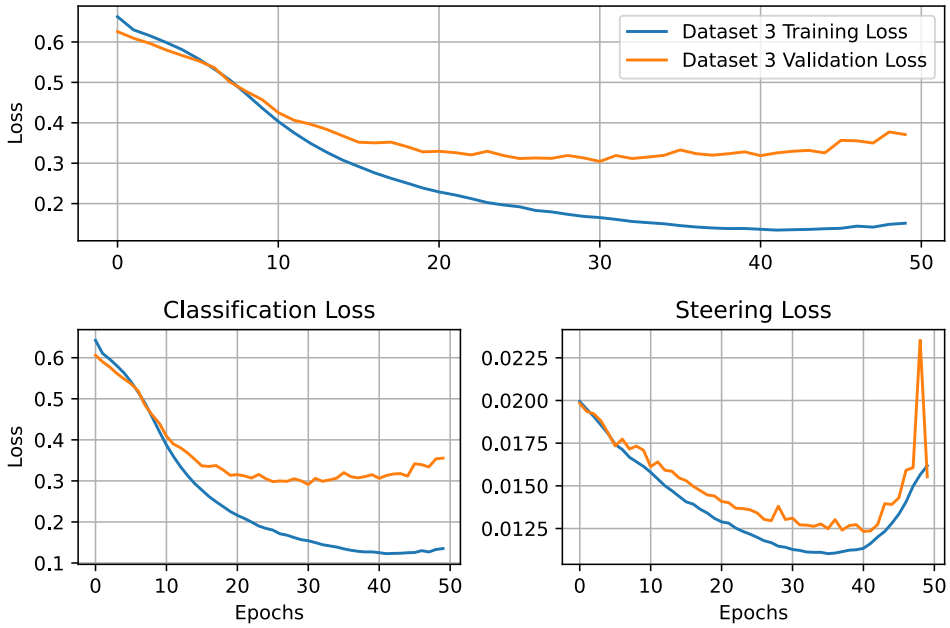


Figure 2.8: Basic model

2.3.1 Basic Model

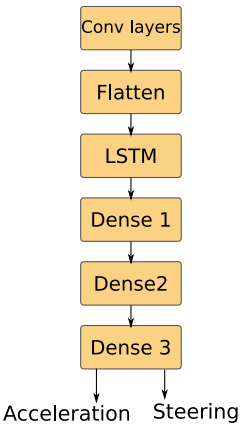


Figure 2.7: Basic model

For training, a model as shown in figure 2.7, is designed and its result is seen in figure 2.8. Interestingly, the training loss curve *follows* the classification loss as it dominates the model. Steering loss however, after epoch 32 starts to increase. Sure enough, upon evaluation, the steering was all over the place and acceleration was not stable at all, resulting in many collisions as shown in table 2.5 (dataset 3 column).

The cause for this behaviour is investigated and it is found that different losses have different magnitudes. By forcing the neural network to learn/train both classification and regression losses from a same dense layer causes instability in learning.

2.3.2 Splitting at the dense layers

To alleviate some of the burden the second dense layer(dense 2) is split into two separate dense layers; one for classification outputs and other for steering as show in figure 2.9. The result 2.10, stops the strange steering loss increase. Upon evaluation, this model shows better steering control but still the acceleration is not stable or consistent as shown in table 2.7.

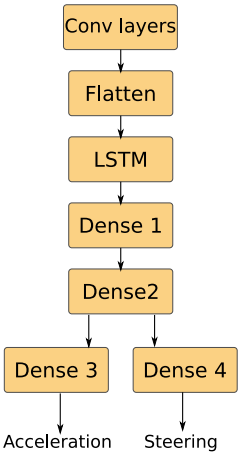


Figure 2.9: Split at the second dense layer

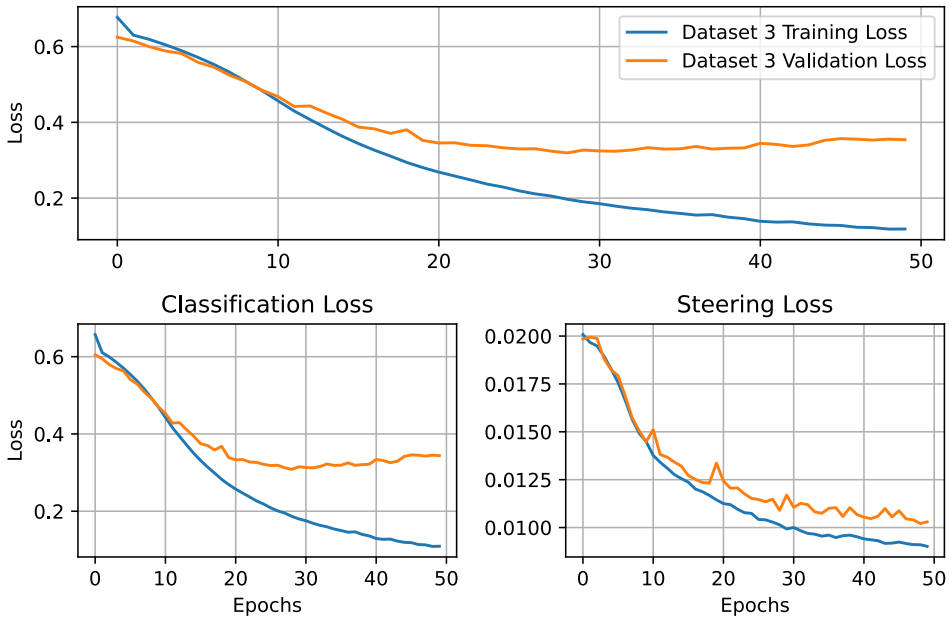


Figure 2.10: Separate dense layers for classification and steering

2.3.3 Splitting at the LSTM layers

Continuing the theme of tuning the network, the model is split further at LSTM layer as shown in figure 2.11. The main aim here is to see if steering control improves and is stable for considerable acceleration prediction. From figure 2.12, the steering loss gets a marginal gain. Still at evaluation, the trained models stops at random places, and steering control is not stable at higher acceleration predicted values. Hence the predicted acceleration value is reduced by 50-70% and fed to the controller. Sure enough the vehicle exhibits stable movements as seen from table 2.8.

2.3.4 Using two different NN for acceleration and Steering

It can be deduced that optimised weights play an important role on how it influences steering and acceleration prediction. The model is now split as two different neural networks(NN) as seen in figure 2.13. Though the result 2.14 looks similar to 2.12, the model predicts stable, consistent acceleration values.

Criteria(Softmax/Categorical crossentropy)	Dataset 3
Lane keeping/Drive straight	Yes
Gradual acceleration increase	Yes
Smooth braking behaviour observed	Yes
Smooth steering control at high speed(10m/s)	No
Smooth steering control at turnings	No
Avoids colliding into static objects	No
Detects vehicles as dynamic objects	Yes
Navigates traffic smoothly	No
Doesn't stop at random places	No

Table 2.7: Separate dense layers - How the model evaluates to different criteria

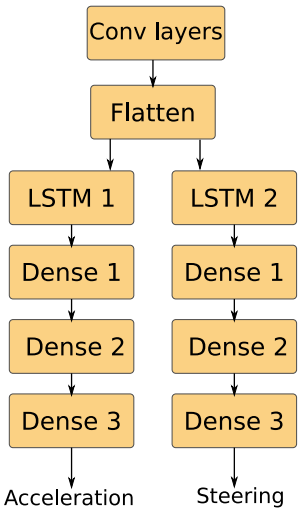


Figure 2.11: Split at the LSTM layer

Criteria(Softmax/Categorical crossentropy)	Dataset 3
Lane keeping/Drive straight	Yes
Gradual acceleration increase	Yes
Smooth braking behaviour observed	Yes
Smooth steering control at high speed(10m/s)	Yes
Smooth steering control at turnings	No
Avoids colliding with static objects	No
Detects vehicles as dynamic objects	Yes
Navigates traffic smoothly	Yes
Doesn't stop at random places(negative case)	Yes
Smooth evaluation experience	No

Table 2.8: Split at the LSTM layer - How the model evaluates to different criteria

From table 2.9, it even exhibits occasional turning behaviours at junctions. When it is exposed to traffic, the model does well to navigate, brake, and accelerate.

A quick overview of steering losses across different NN changes is shown in figure 2.15.

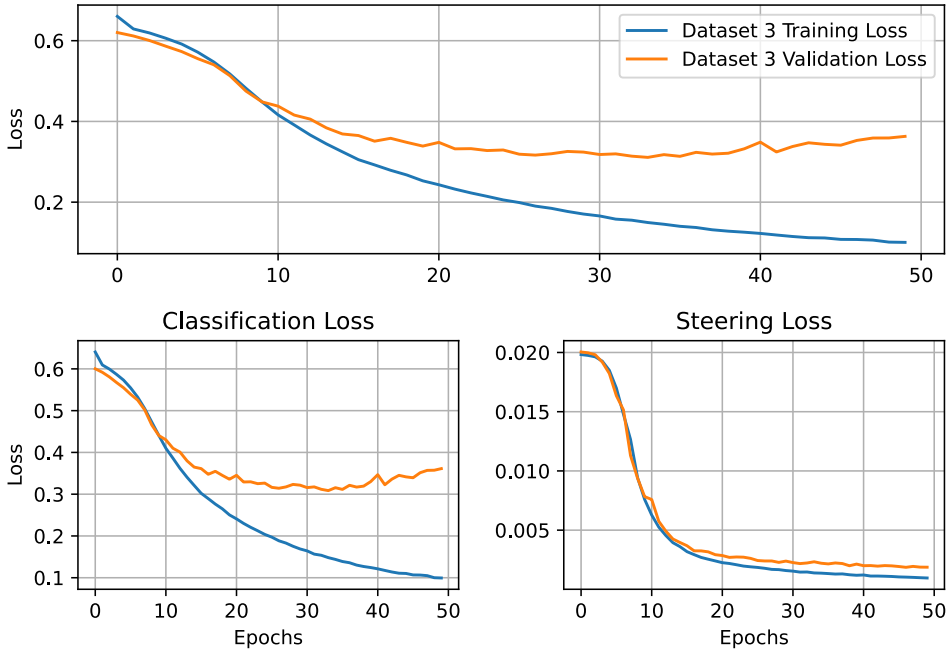


Figure 2.12: Separate LSTM layers for classification and steering

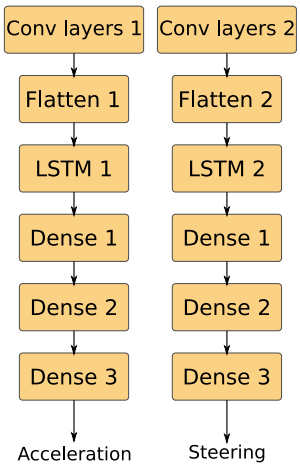


Figure 2.13: Separate NN training model

2.3.5 Observations

1. Tuning the neural network albeit only the fully connected layers, results in optimised prediction of steering control corresponding to the classification outputs.

2. The car exhibits random, unknown stops at random places. The actual reason behind it is unknown but it is suspected that since the dataset has imbalanced classes, it contributes to this random decisions.

3. Separating into two NNs allows better steering control at a higher speed.

4. Sunlight and shadows still play a major role. They do some random, strange things to models that eventually lead to crashes out-of-nowhere. It could be deduced that some buildings' shadows could be considered as static or dynamic objects.

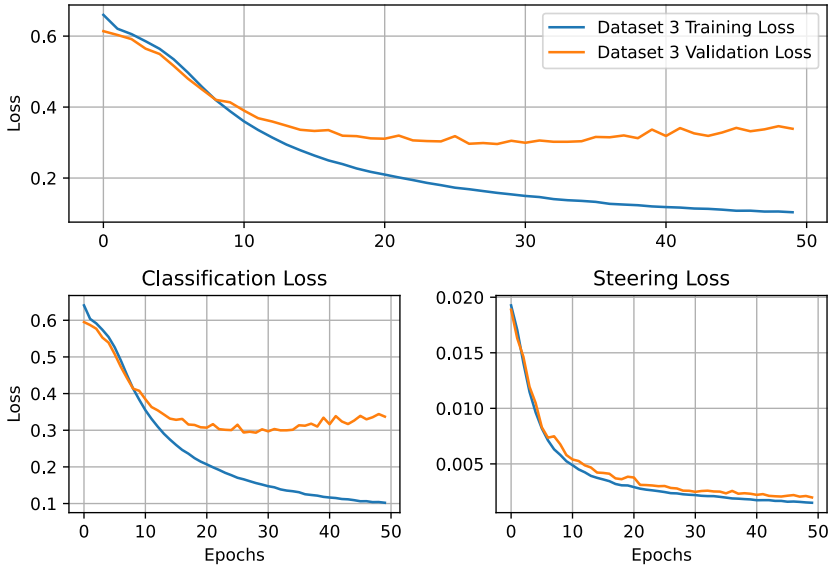


Figure 2.14: Separate neural network for Classification and Steering

Criteria(Softmax/Categorical crossentropy)	Dataset 3
Lane keeping/Drive straight	Yes
Gradual acceleration increase	Yes
Smooth braking behaviour observed	Yes
Smooth steering control at high speed(10m/s)	Yes
Smooth steering control at turnings	No
Avoids colliding with static objects	Yes
Detects vehicles as dynamic objects	Yes
Navigates traffic smoothly	Yes
Doesn't stop at random places(negative case)	Yes
Smooth evaluation experience	Yes

Table 2.9: Separate neural network for classification and steering outputs - How the model evaluates to different criteria

2.4 Velocity

Velocity is a scalar value, labelled output. It is considered as an *auxiliary task*. An auxiliary task usually consists of estimating quantities that are relevant to solving main supervised learning problem. That means that velocity is predicted as an auxiliary task using existing CNN-LSTM-Dense architectures without affecting the major tasks i.e., predicting acceleration and steering.

The images are fed into the models shown in figure 2.16 and results are compared as how predicting velocity affects acceleration and steering.

From the figure 2.17, looking at the loss-epoch graphs of model *a*, *b*, *c*, the validation losses follow velocity's validation loss. When compared to classification and steering loss, this loss is too high. Sure enough while evaluating steering and acceleration are worse. Since velocity is an auxiliary task, there is a need to rethink how cost function is calculated.

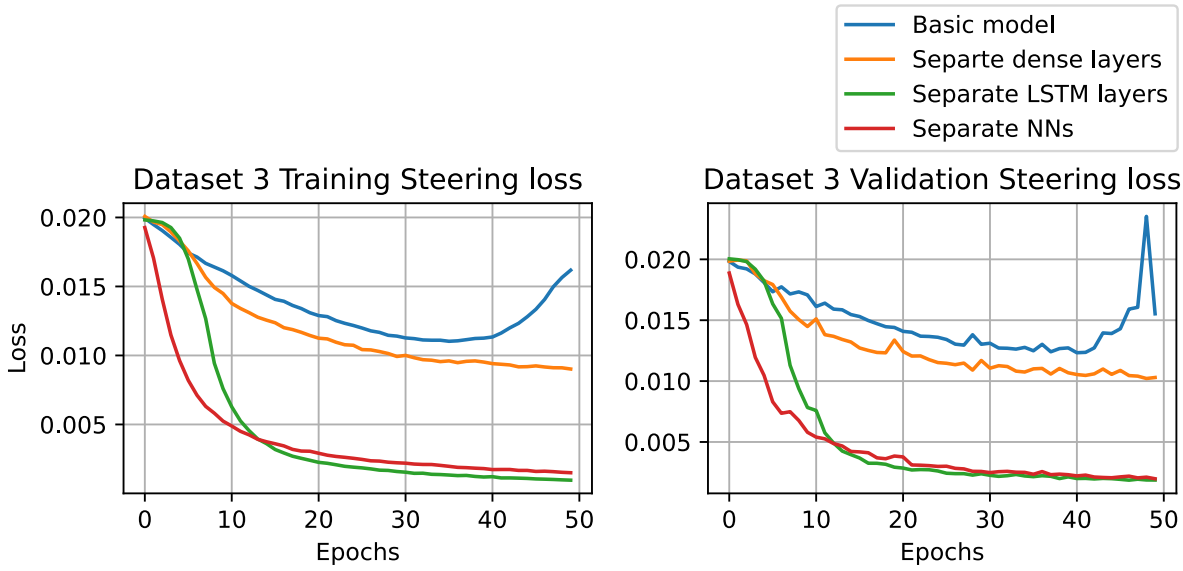


Figure 2.15: Steering command loss comparison

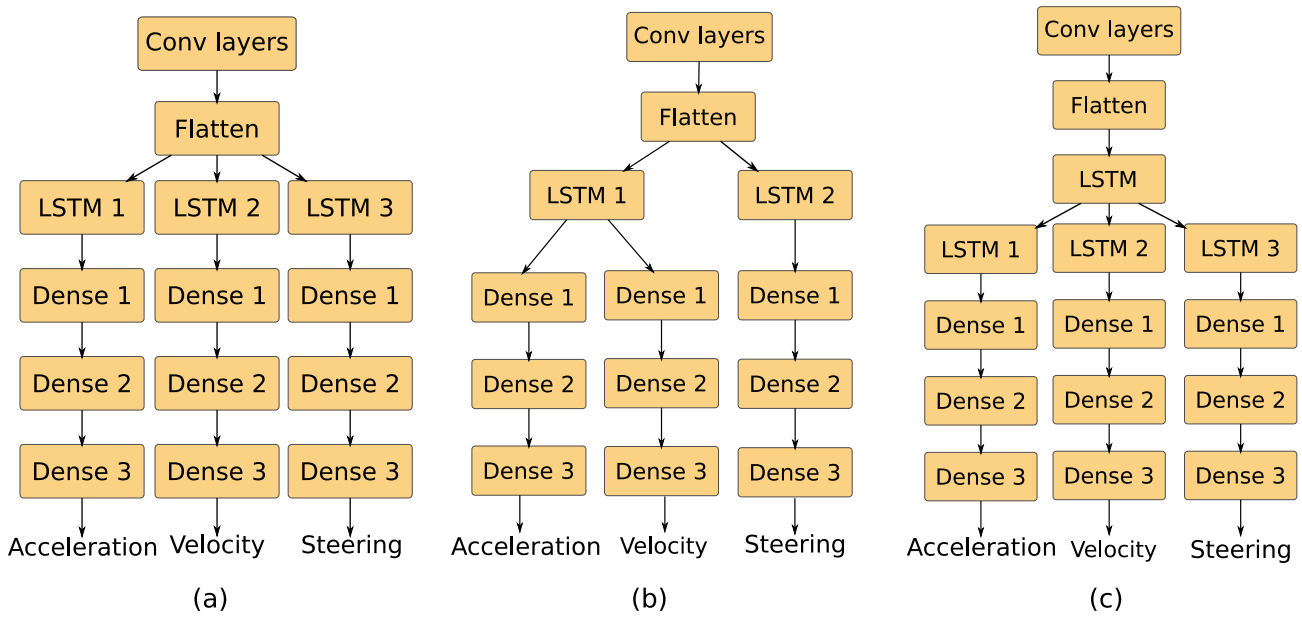


Figure 2.16: Different architectures used while predicting velocity

2.4.1 Weighted loss function

Since velocity is taking control of the neural network to provide feature extraction and decision making for its prediction, both acceleration and steering suffer bad consequences. In order to avoid this, weighted cost function is introduced to the three outputs. As velocity is an auxiliary task, it is suppressed more than others. From figures 2.16 and 2.17, model *a* gives nearly optimum losses. So that model is chosen to carry out this weighted cost function experiment.

The model is tweaked only to include custom, weighted cost function. The trained model exhibits now losses similar to classification losses than velocity(as seen in figure 2.17, *weighted* graph). Indeed during evaluation, acceleration and steering both get preference and control than velocity. Hence it is

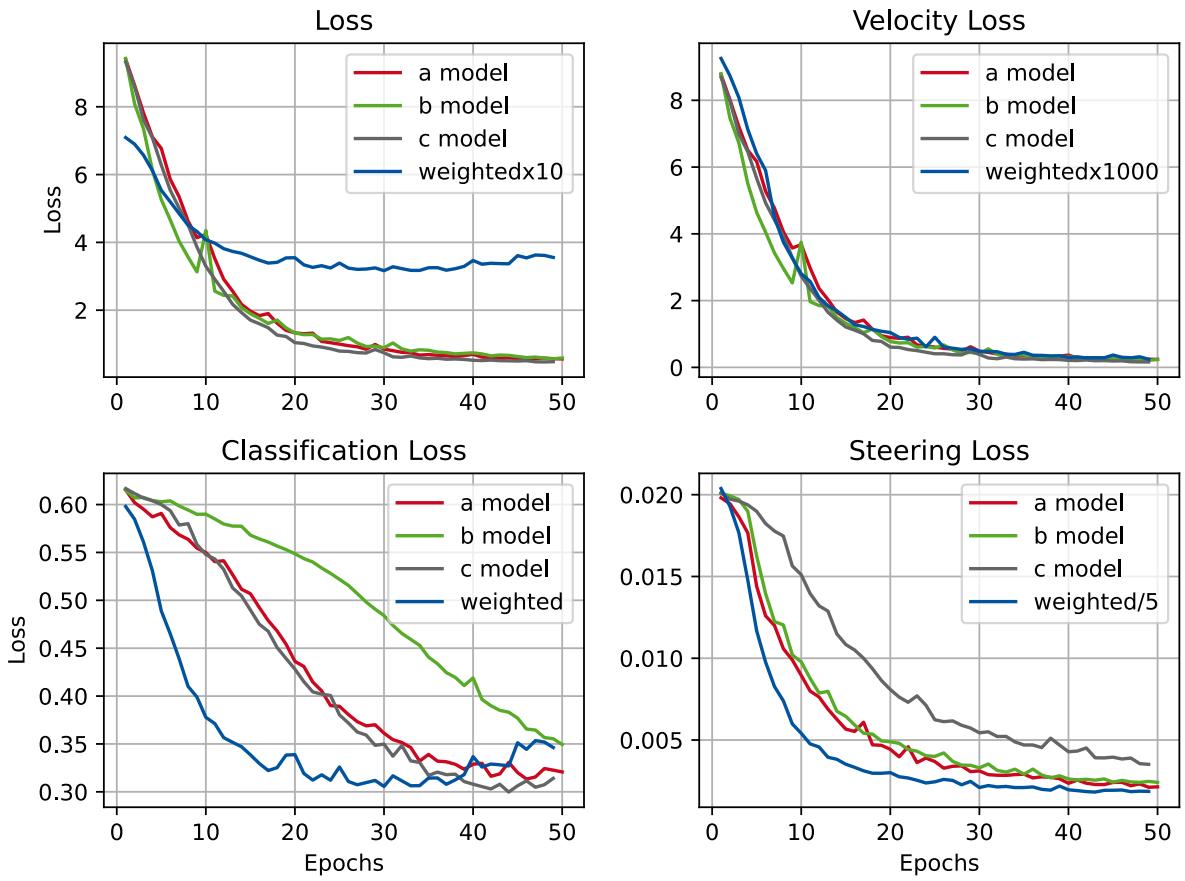


Figure 2.17: Comparison of losses for NN architectures shown in figure 2.16

possible to include auxiliary tasks to normal NNs without compromising the actual functionality of it.

2.5 Convolution layers manipulation

The fully connected/dense layers are tweaked to various designs as shown in figures 2.7, 2.9, 2.11, 2.13 and having convolutional layers as constant. Some interesting possibilities and observations are made possible. As a consequence, the convolutional layers are changed keeping dense layers as constant. As velocity prediction is also included, 2.16a model is considered.

The convolutional layers can be adjusted either by width(changing the feature maps or stride parameter) or depth(changing the number of layers).

2.5.1 Adjusting the width of the convolutional layers

Changing the feature maps channel depth

The convolutional layers consists of feature maps channels, kernel filter which convolves on the input using a specified stride. In our case, the last convolutional layer's feature map channel's depth is increased from 64 to 80 (as shown in figure 2.18a). This change, after *flatten* layer, increases the trainable parameters. This allows for more features to carried into the fully connected layer. Looking at the loss-epoch graph(fig.2.19) for this experiment, they give relatively similar results as 2.17a

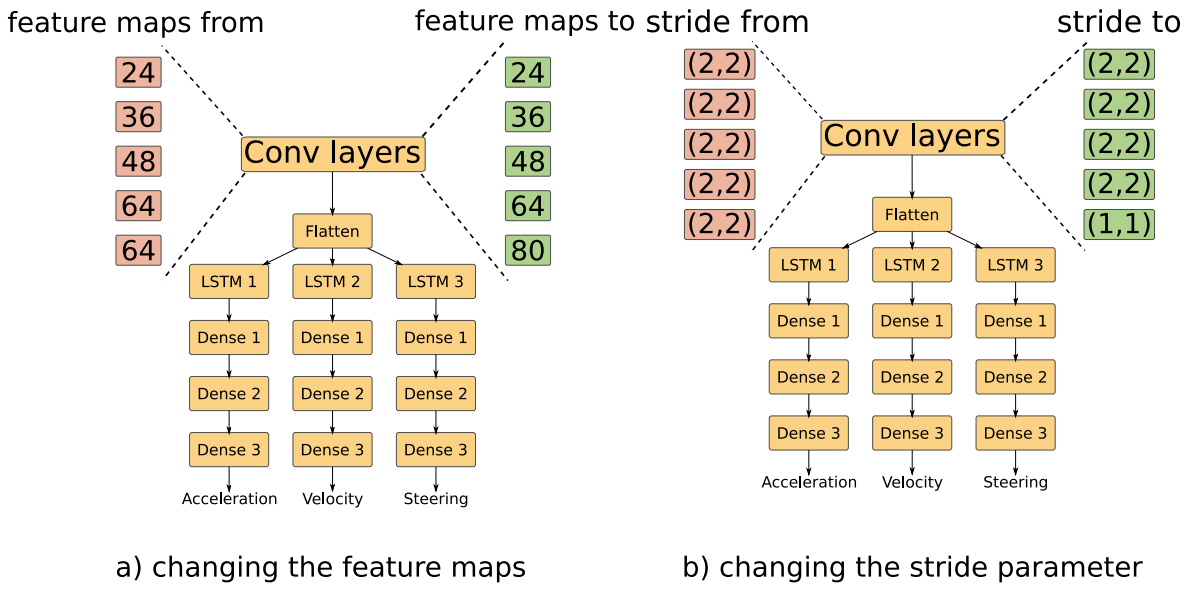


Figure 2.18: Convolutional layers width changes - Increasing feature maps channel depth

model.

Changing the stride

Stride is a component in CNN tuned for compressing the images data. This parameter specifies the movement step of the kernel/filter. In our case, the stride of the 5th convolutional layer is decreased from (2,2) to (1,1) (see fig. 2.18b). This ensures the information tensors are not compressed by half. The flatten layer gets uncompressed, more tensors which increases the trainable parameters of the features. From fig. 2.19, we see that losses are similar to the 2.17a model but the classification loss starts to overfit well ahead of the *a* model. Since the training model follows the classification loss, the best model is not stored with this change.

2.5.2 Adjusting the depth of the Convolutional layers

Increase the convolutional layers to eleven

Fig.2.20a, shows how the convolutional layers are increased to 11 layers and also the tensor shape before *flatten* layer is made greater than the previous case(a model). The change in losses because of these adjustments is instant. Fig.2.21 shows that the model overfits as soon as epoch 10. The model follows the classification loss. Though other losses continue to decrease, these events are not recorded.

Increase the convolutional layers to eight

If the convolutional layers are increased only to 8 and the tensor shape is kept as small as possible before flatten layer, the loss-epoch graph 2.21 shows better behaviour than increasing to 11 layers. However, this change also brings about overfitting behaviour. It could be seen that the velocity and steering losses converge well. However, like before, they are not recorded.

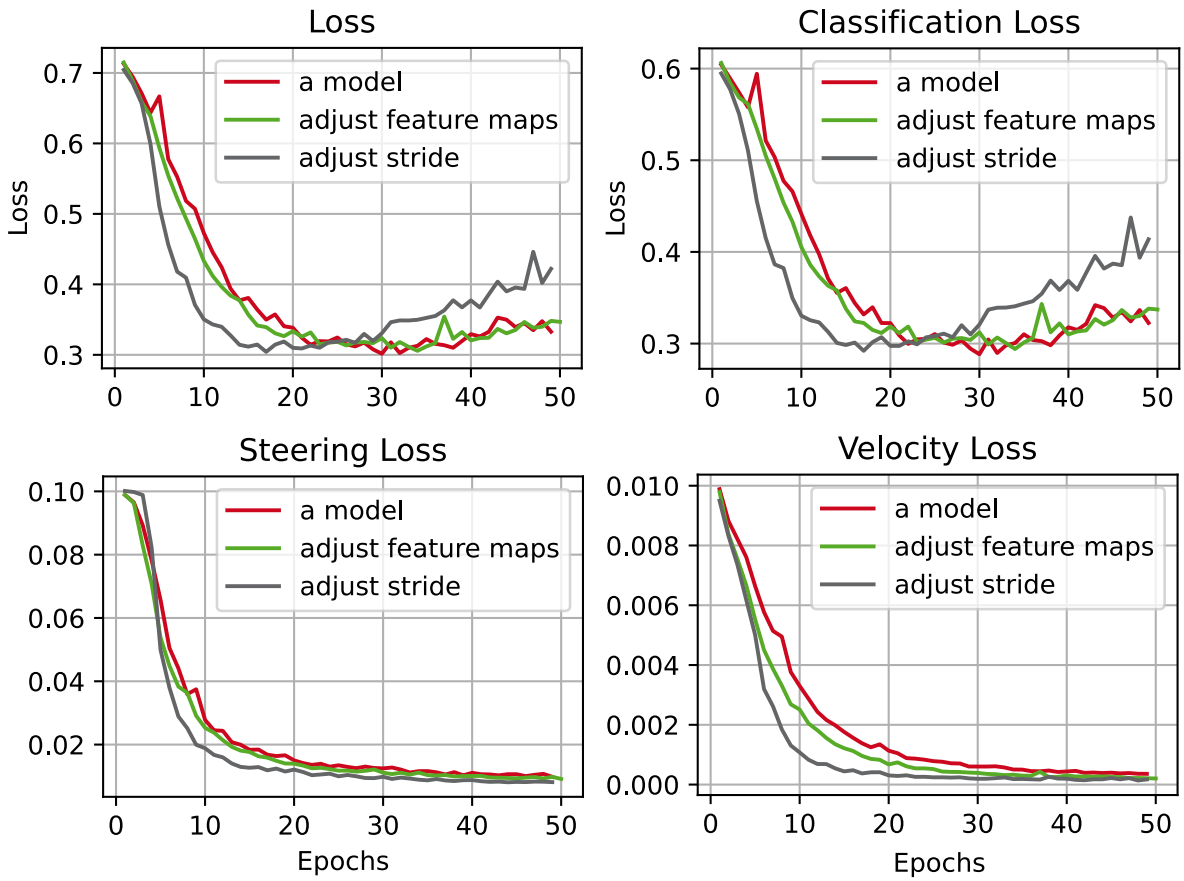
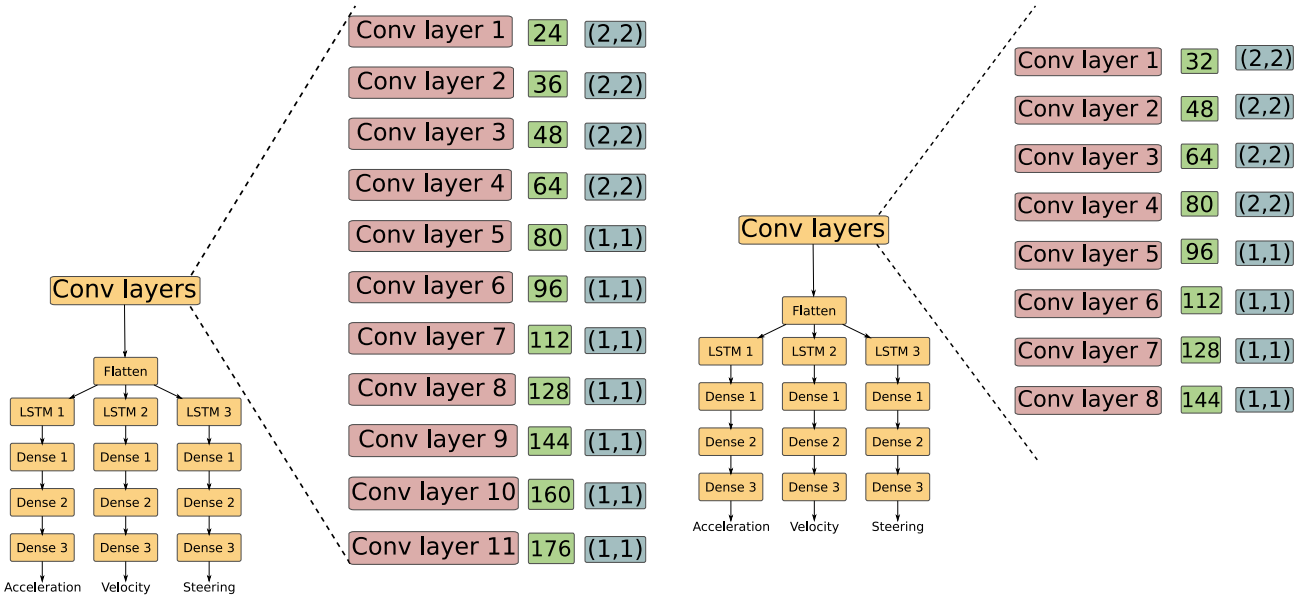


Figure 2.19: Convolutional layers width adjustments - Comparison of losses



a) Increasing to 11 layers

b) Increasing to 8 layers

Figure 2.20: Convolutional layers depth changes - Increasing the number of CNN layers

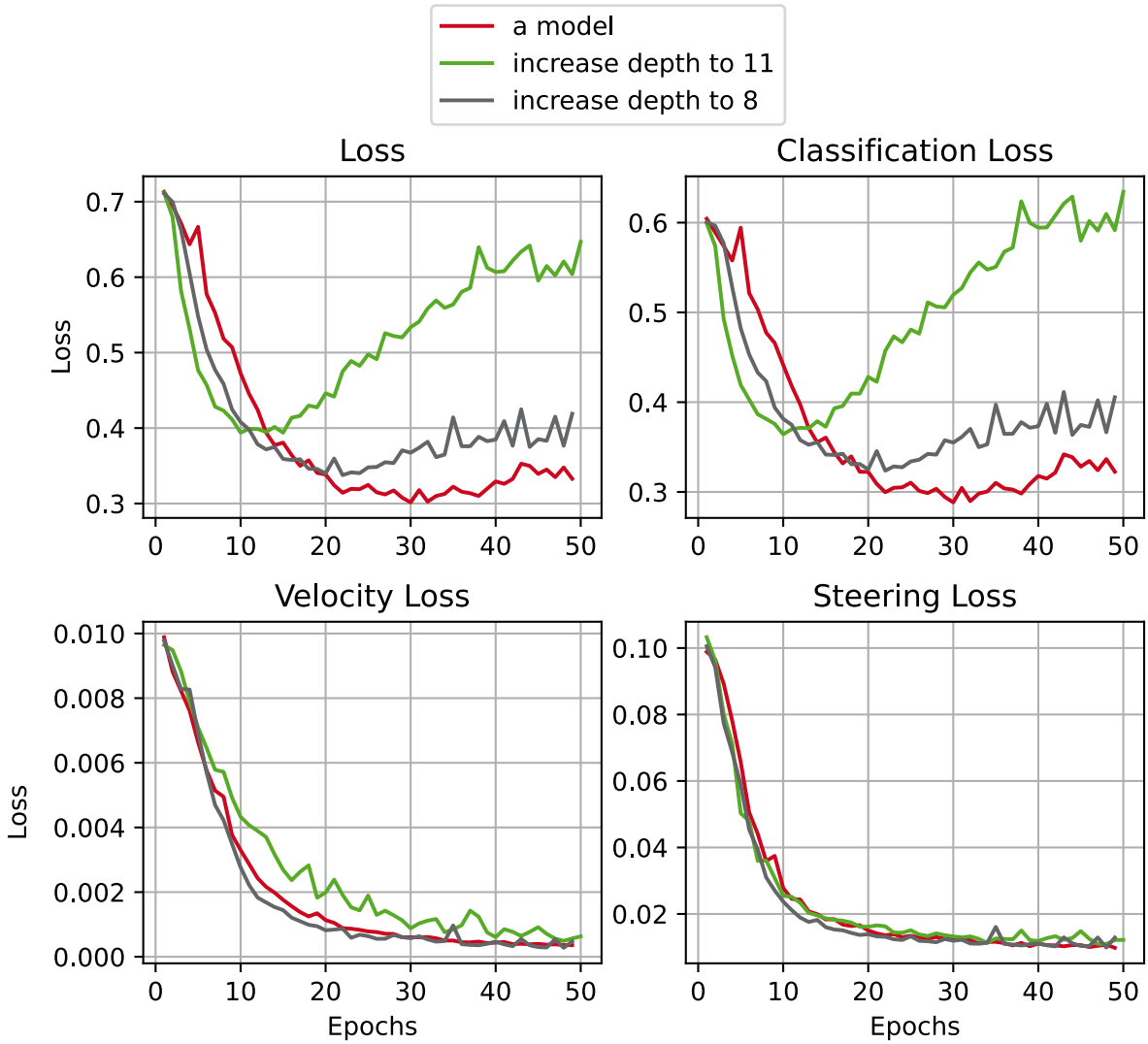


Figure 2.21: Convolutional layers depth adjustments - Comparison of losses

2.6 Depth Camera

The depth camera is typically used to measure the distance between objects. In our case, we use it to provide additional spatial information between vehicles and objects. We test how the neural networks adapts and predicts acceleration, braking and steering.

The figure 2.22 loss-epoch graph shows a little better performance compared to colour-RGB images trained model. For training the architecture 2.16a is used.

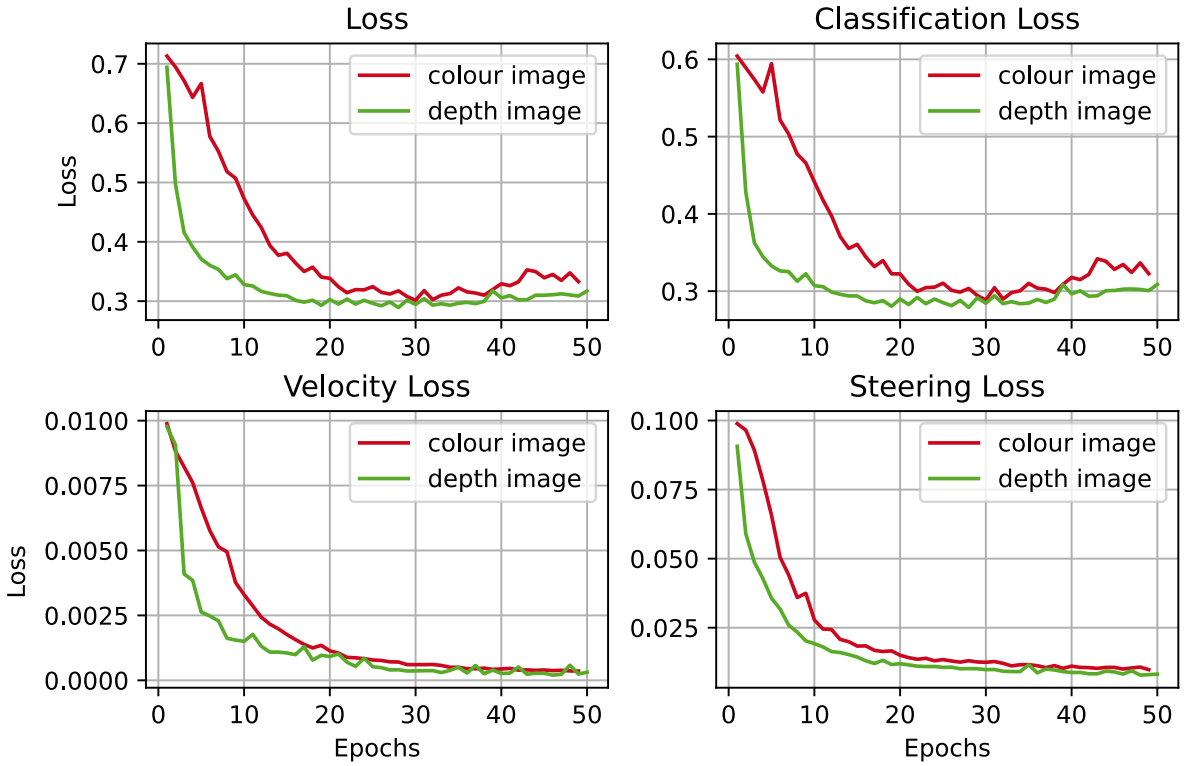


Figure 2.22: Comparison of losses between using colour and depth images

List of Figures

List of Tables

References

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

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

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

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