END-TO-END SELF DRIVING VEHICLES USING TIME DEPENDECY MODELS, REINFORCEMENT LEARNING AND SENSOR FUSION

By

Abdallah Ahmed El Saeed

Mohamed Abdel El-Halim Bedier

Mohamed Sabry Abd El-Rady

Youssef Mostafa Ibrahim

A Thesis Submitted to the
Faculty of Engineering at Cairo University in
Partial Fulfillment of the
Requirements for the Degree of
BACHELOR OF ENGINEERING
in
Electronics and Communications Engineering

Under the supervision of

Prof. Hanan Ahmed **Dr. Hassan Mostafa**

Dr. Samah Tantawy

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
2020

END-TO-END SELF DRIVING VEHICLES USING TIME DEPENDECY MODELS, REINFORCEMENT LEARNING AND SENSOR FUSION

By

Abdallah Ahmed El Saeed

Mohamed Abdel El-Haleem

Mohamed Sabry

Youssef Mostafa

A Thesis Submitted to the
Faculty of Engineering at Cairo University in
Partial Fulfillment of the
Requirements for the Degree of
BACHELOR OF ENGINEERING
in
Electronics and Communications Engineering

Under the supervision of

Prof. Hanan Ahmed

Affiliation

Electronics and Communications Engineering

Faculty of Engineering, Cairo University

Dr. Hassan Mostafa	**Dr. Samah Tantawy**
Affiliation	Affiliation
Department	Department
Faculty, University	Faculty, University

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
2020

Acknowledgements

This dissertation does not only hold the results of the year's work, but also reflects the relationships with many generous and stimulating people. This is the time where we have the opportunity to present our appreciation to all of them.

First, to our advisors, **Prof. Dr. Hanan Kamal**, **Dr. Hassan Mustafa** and **Dr. Samah El-Tantawy**, we would like to express our sincere gratitude and appreciation for your excellent guidance, caring, patience, and immense help in planning and executing this project in a timely manner. Their great personality and creativity provided us with an excellent atmosphere for work, while their technical insight and experience helped us a lot in our research. Their support at the time of crisis will always be remembered.

Of course, we will never find words enough to express the gratitude and appreciation that we owe to our families. Their tender love and support have always been the cementing force for building what we achieved. The all-round support rendered by them provided the much-needed stimulant to sail through the phases of stress and strain.

Finally, we would also like to thank **Eng. Eslam Baker** who is an Algorithms Software Engineer and Deep Learning Researcher at **Valeo**, who was always there for any questions, providing precious advice and sharing his time and experience.

Abstract

The development of autonomous ground vehicles is a long-studied instantiation especially, navigation in densely populated urban environments. This setting is particularly challenging due to the complex multi-agent dynamics at traffic intersections; the necessity to track and respond to the motion of tens or hundreds of other actors in the surrounding environment that may be in view at any given time; prescriptive traffic rules that necessitate recognizing street signs, street lights, and road markings and distinguishing between multiple types of other vehicles, and recognizing long tail of rare events and road construction, for instance, a child running onto the road, an accident ahead, or a rogue driver barreling on the wrong side; and the necessity to rapidly reconcile conflicting objectives, such as applying appropriate deceleration when an absent-minded pedestrian strays onto the road ahead but another car is rapidly approaching from behind and may rear-end if one brake too hard.

In this work, supervised learning and reinforcement learning are used to efficiently train a simulated car to drive autonomously and navigate from a start point to an end point without collision or intersections with road lanes in an end-to-end fashion.

- **Time-dependent Neural Networks for End-to-End Autonomous Driving using 3D Convolution and Long Short-Term Memories (LSTM).**

We implemented and trained three time-dependent neural architectures as a modification on the branched neural network introduced by Intel, Imposing time dependency to NN which increased the accuracy and improved the generalization capability measured using Intel benchmarks in CARLA Simulator.

- **End-to-end Sensor Fusion between RGB Images and LiDAR Point Cloud.**

We are pioneers in testing the fusion of Camera and LiDAR in end-to-end driving using Voxel Net introduced in (Codevilla, Muller, Lopez, Koltun, & Dosovitskiy, 2017) and the branched neural network introduced in (Zhou & Tuzel, 2017). The fusion increased the accuracy even further due to the presence of depth information provided by the LiDAR point cloud.

- **Integration of Reinforcement Learning and Supervised Learning.**

We trained an agent using Reinforcement Learning in order to achieve a well generalized driving policy, but due to the high dimensional state space the model was not able to converge well even by trying state of the art RL algorithms like Rainbow algorithm, therefore we integrated both supervised learning and reinforcement learning to enhance agent convergence, and we achieved better results than that of using RL alone.

Contents

Table of Figures

List of Tables

Chapter 1 Introduction

Imitation learning and reinforcement learning are receiving renewed interest as promising approaches to build autonomous driving vehicles. Demonstration of human driving can be easily collected at scale, and can easily be simulated using graphics engines, both imitation learning and reinforcement learning can be used to train an agent to predict control commands; for example, mapping camera images to steering, acceleration and brake. However, these systems have characteristic limitations especially when it comes to fully autonomous urban driving. One limitation is the assumption that the optimal action can be predicted only from perceptual input alone, because making a turn at an intersection cannot be predicted from camera images only. The work in (Codevilla, Muller, Lopez, Koltun, & Dosovitskiy, 2017) addresses this issue using conditional imitation learning, where the GPS high level directions are fed to the function approximator along with the camera images to predict optimal actions, just as mapping applications and passengers provide turn-by-turn directions to human drivers. However, their work still faces some difficulties. Fitting a function approximator to imitate a driving policy requires massive amounts of demonstrations and should be prone to overfitting. In this report, we address this challenge by invoking different function approximator architectures that can benefit from time as well as spatial information. Also, we presented a novel end-to-end fusion solution that can benefit from both Camera spatial information and LiDAR depth information to enhance capability of prediction. Finally, we introduce an integration between reinforcement learning and imitation learning to achieve more generalization and allow efficient online learning. Experiments are done with realistic simulation of urban driving using CARLA simulator and can be easily deployed in the physical world. Recordings of both systems are provided in the supplementary video. In this work, supervised learning and reinforcement learning is used to efficiently train a simulated car to drive autonomously and navigate from start point to end point without collision with or intersections with road lanes in and end-to-end fashion.

1.1 History

Research in autonomous urban driving of requires a lot of infrastructure costs and the logistical difficulties of training and testing systems in the physical world.

In 1969, John McCarthy — a.k.a. one of the founding fathers of artificial intelligence — describes something similar to the modern autonomous vehicle in an essay titled “Computer-Controlled Cars.” McCarthy refers to an “automatic chauffeur,” capable of navigating a public road via a “television camera input that uses the same visual input available to the human driver.” He writes that users should be able to enter a destination using a keyboard, which would prompt the car to immediately drive them there. Additional commands allow users to change destination, stop at a rest room or restaurant, slow down, or speed up in the case of an emergency. No such vehicle is built, but McCarthy’s essay lays out the mission for other researchers to work toward.

In the early 1990s, Carnegie Mellon researcher Dean Pomerleau writes a PhD thesis, describing how neural networks could allow a self-driving vehicle to take in raw images from the road and output steering controls in real time. Pomerleau isn’t the only researcher working on self-driving cars, but his use of neural nets proves way more efficient than alternative attempts to manually divide images into “road” and “non-road” categories.

In 1995, Pomerleau and fellow researcher Todd Jochem take their Navlab self-driving car system on the road. Their bare bones autonomous minivan (they have to control speed and braking) travels 2,797 miles coast-to-coast from Pittsburgh, Pennsylvania to San Diego, California in a journey the pair dubs “No Hands Across America.”

In 2002, DARPA announces its Grand Challenge, offering researchers from top research institutions a one million prize if they can build an autonomous vehicle able to navigate 142 miles through the Mojave Desert. When the challenge kicks off in 2004, none of the 15 competitors are able to complete the course. The “winning” entry makes it less than eight miles in several hours, before catching fire. It’s a damaging blow to the goal of building real self-driving cars. While autonomous vehicles still seem way in the future in the decade of the 2000s, self-parking systems begin to emerge — demonstrating that sensors and autonomous road technologies are getting close to ready for real world scenarios. Toyota’s Japanese Prius hybrid vehicle offers automatic parallel parking assistance from 2003, while Lexus soon adds a similar system for its Lexus LS sedan, Ford incorporates Active Park Assist in 2009, and BMW follows one year later with its own parallel parking assistant.

Starting in 2009, Google begins developing its self-driving car project, now called Waymo, in secret. The project is initially led by Sebastian Thrun, the former director of the Stanford Artificial Intelligence Laboratory and co-inventor of Google Street View. Within a few years, Google announces that its autonomous cars have collectively driven 300,000 miles under computer control without one single accident occurring. In 2014, it reveals a prototype of a driverless car without any steering wheel, gas pedal or brake pedal, thereby being 100 percent autonomous. By the end of last year, more than 2 million miles had been driven by Google's autonomous car.

By 2013, major automotive companies including General Motors, Ford, Mercedes Benz, BMW, and others are all working on their own self-driving car technologies. Nissan commits to a launch date by announcing that it will release several driverless cars by the year 2020. Other cars, such as the 2014 Mercedes S-Class, add semiautonomous features such as self-steering, the ability to stay within lanes, accident avoidance, and more. The likes of Tesla and Uber also begin actively exploring self-driving technology, while Apple is rumored to be doing so. The first autonomous car fatality

At CES 2018, Nvidia unveiled a new self-driving car chip called Xavier that will incorporate artificial-intelligence capabilities. The company then announced that it was partnering with Volkswagen to develop AI for future self-driving cars. While not the first effort to imbue autonomous cars with AI (Toyota was already researching the concept with MIT and Stanford), the VW-Nvidia collaboration is the first to connect AI to production-ready hardware. It opens up the possibility for self-driving cars to perform better, as well as for new convenience features like digital assistants.

In recent years, AI becomes the major element towards designing a full self-driving car using different approaches that differ on how much they rely on AI to perform autonomous driving tasks.

1.2 Problem Definition

Instrumenting and operating even one robotic car require significant funds and manpower. And a single vehicle is far from sufficient for collecting the requisite data that cover the multitude of corner cases that must be processed for both training and validation. This is true for classic modular pipelines approaches and even more so for data hungry deep learning techniques. Training and validation of sensorimotor control models for urban driving in the physical world is beyond the reach of most research groups. An alternative is to train and validate driving strategies in simulation. Simulation can democratize research in autonomous urban driving. It is also necessary for system verification, since some scenarios are too dangerous to be staged in the physical world (e.g., a child running onto the road ahead of the car). Simulation has been used for training driving models since the early days of autonomous driving research. More recently, urban simulators have been used to evaluate new approaches to autonomous driving. CARLA simulator, for example, is one of the most used urban driving simulators to train and benchmark autonomous vehicles.

In our work, CARLA simulator is used to train and evaluate an end-to-end self-driving car using two different approach, supervised learning and reinforcement learning, aiming to achieve an efficient end-to-end solution that can complete driving missions safely in different conditions.

First, we address the problem as a controller that interacts with the environment over discrete time steps. At each time step t, the controller receives an observation O_t and takes an action a_t. The basic idea behind imitation learning is to train a controller that mimics an expert. The training data is a set of observation-action pairs $D = \{\langle o_i, a_i \rangle\}_{i=1}^{N}$ generated by the expert.

Regarding these settings; our problem in which the parameters θ of a function approximator $F(O_i; \theta)$ must be optimized to fit the mapping of observations to actions.

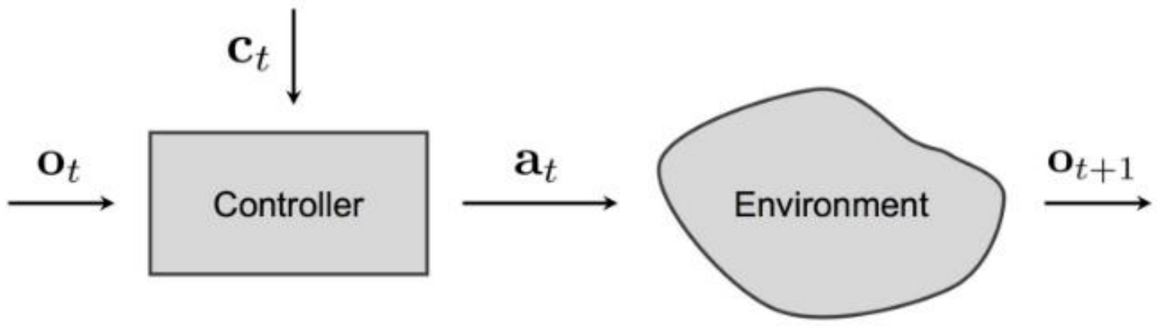


Figure 1-1 Controller basic settings

1.3 Objectives

Since safety is crucial in the self-driving car fields, the accuracy of the end-to-end AI model and its generalization capability is critical, our objective from this project is to test new ideas in designing neural networks for both imitation learning and deep reinforcement learning including, Firstly, sensor fusion between sensors used by autonomous cars, Camera and LIDAR, for example, secondly, to benefit from time-dependent neural network architectures like recurrent neural networks and 3D-Convolution networks to increase accuracy of prediction and achieve generalization without overfitting, Lastly, following the desire in covering large portion in state space and covering many driving scenarios without using huge datasets, reinforcement learning approach is used integrated with imitation learning to achieve stability and convergence.

Chapter 2 Supervised Learning

2.1 The Learning Problem

If you show a picture to a three-year-old and ask if there is a tree in it, you will likely get the correct answer. If you ask a thirty-year-old what the definition of a tree is, you will likely get an inconclusive answer. We didn't learn what a tree is by studying the mathematical definition of trees. We learned it by looking at trees.

In other words, we learned from "data". Learning from data is used in situations where we don't have an analytic solution, but we do have data that we can use to construct an empirical solution. This premise covers a lot of territory, and indeed learning from data is one of the most widely used techniques in science, engineering, and economics, among other fields. In this chapter, we present examples of learning from data and formalize the learning problem. We also discuss the main concepts associated with learning, and the different paradigms of learning that have been developed.

2.1.1 Problem Setup

What do financial forecasting, medical diagnosis, computer vision, and search engines have in common? They all have successfully utilized learning from data. The repertoire of such applications is quite impressive.

Suppose that a bank receives thousands of credit card applications every day, and it wants to automate the process of evaluating them. Just as in the case of movie ratings, the bank knows of no magical formula that can pinpoint when credit should be approved, but it has a lot of data. This calls for learning from data, so the bank uses historical records of previous customers to figure out a good formula for credit approval. Each customer record has personal information related to credit, such as annual salary, years in residence, outstanding loans, etc. The record also keeps track of whether approving credit for that customer was a good idea, i.e., did the bank make money on that customer. This data guides the construction of a successful formula for credit approval that can be used on future applicants.

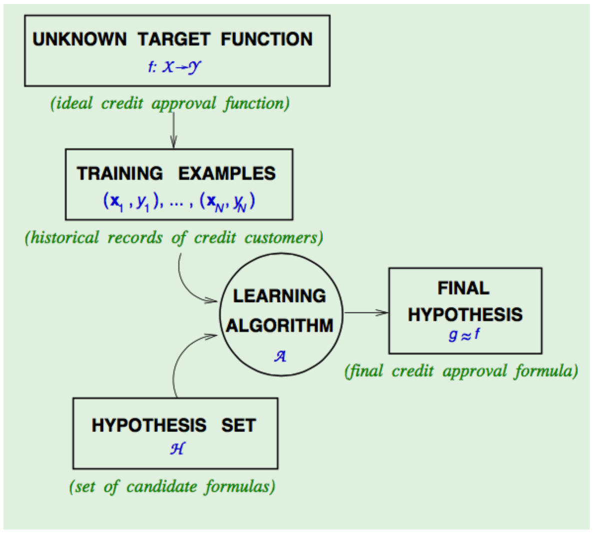

UNKNOWN TARGET FUNCTION
$f: \mathcal{X} \to \mathcal{Y}$
(ideal credit approval function)
TRAINING EXAMPLES
$(\mathbf{x}_1, y_1), \cdots, (\mathbf{x}_N, y_N)$
(historical records of credit customers)
LEARNING ALGORITHM
$\mathcal{A}$
FINAL HYPOTHESIS
$g \approx f$
(final credit approval formula)
HYPOTHESIS SET
$\mathcal{H}$
(set of candidate formulas)

Figure 2-1 Basic setup of the learning problem

Let us give names and symbols to the main components of this learning problem. There is the input X (customer information that is used to make a credit decision), the unknown target function f: X→Y (ideal formula for credit approval), where X is the input space (set of all possible inputs x), and Y is the output space (set of all possible outputs, in this case just a yes/no decision). There is a data set D of input-output examples $(x_1,y_1),......,(x_N,y_N)$, where $y_n = f(x_n)$ for $n = 1,...,N$, (inputs corresponding to previous customers and the correct credit decision for them in hindsight).

The examples are often referred to as data points. Finally, there is the learning algorithm that uses the data set D to pick a formula g: X→Y that approximates f. The algorithm chooses g from a set of candidate formulas under consideration, which we call the hypothesis set *H*. For instance, *H* could be the set of all linear formulas from which the algorithm would choose the best linear fit to the data, as we will introduce later in this section. When a new customer applies for credit, the bank will base its decision on g (the hypothesis that the learning algorithm produced), not on f (the ideal target function which remains unknown). The decision will be good only to the extent that g faithfully replicates f. To achieve that, the algorithm chooses g that best matches f on the training examples of previous customers, with the hope that it will continue to match f on new customers. Whether or not this hope is justified remains to be seen. Figure 2-1 illustrates the components of the learning problem.

We will use the setup in Figure 2-1

As our definition of the learning problem. Later on, we will consider a number of refinements and variations to this basic setup as needed. However, the essence of the problem will remain the same. There is a target to be learned. It is unknown to us. We have a set of examples generated by the target. The learning algorithm uses these examples to look for a hypothesis that approximates the target.

2.1.2 Types of Learning

The basic premise of learning from data is the use of a set of observations to uncover an underlying process. It is a very broad premise, and difficult to fit into a single framework. As a result, different learning paradigms have arisen to deal with different situations and different assumptions. In this section, we introduce some of these paradigms. The learning paradigm that we have discussed so far is called supervised learning. It is the most studied and most utilized type of learning, but it is not the only one. Some variations of supervised learning are simple enough to be accommodated within the same framework. Other variations are more profound and lead to new concepts and techniques that take on lives of their own. The most important variations have to do with the nature of the data set.

2.1.2.1 Supervised Learning

When the training data contains explicit examples of what the correct output should be for a given inputs, then we are within the supervised learning setting that we have covered so far. Consider the hand-written digit recognition problem. A reasonable data set for this problem is a collection of images of hand-written digits, and for each image, what the digit actually is. We thus have a set of examples of the form (image, digit). The learning is supervised in the sense that some ’supervisor’ has taken the trouble to look at each input, in this case an image, and determine the correct output, in this case one of the ten categories O, 1, 2, 3, 4, 5, 6, 7, 8, 9. While we are on the subject of variations, there is more than one way that a data set can be presented to the learning process. Data sets are typically created and presented to us in their entirety at the outset of the learning process. For instance, historical records of customers in the credit-card application, and previous movie ratings of customers in the movie rating application, are already there for us to use. This protocol of a “ready” data set is the most common in practice, and it is what we will focus on in this book. However, it is worth noting that two variations of this protocol have attracted a significant body of work. One is active learning, where the data set is acquired through queries that we make. Thus, we get to choose a point x in the input space, and the supervisor reports to us the target value for x. As you can see, this opens the possibility for strategic choice of the point x to maximize its information value, similar to asking a strategic question in a game of 20 questions. Another variation is called online learning, where the data set is given to the algorithm one example at a time. This happens when we have streaming data that the algorithm has to process “on the run”. Online learning is also useful when we have limitations on computing and storage that preclude us from processing the whole data as a batch. We should note that online learning can be used in different paradigms of learning, not just in supervised learning.

2.1.2.2 Unsupervised Learning

Unsupervised learning is where you only have input data (X) and no corresponding output variables. The goal for unsupervised learning is to model the underlying structure or distribution in the data in order to learn more about the data. These are called unsupervised learning because unlike supervised learning above there is no correct answers and there is no teacher. Algorithms are left to their own devises to discover and present the interesting structure in the data. Unsupervised learning problems can be further grouped into clustering and association problems. Clustering: A clustering problem is where you want to discover the inherent groupings in the data, such as grouping customers by purchasing behavior. Association: An association rule learning problem is where you want to discover rules that describe large portions of your data, such as people that buy X also tend to buy Y. Some popular examples of unsupervised learning algorithms are: k-means for clustering problems. Apriori algorithm for association rule learning problems.

Between supervised and unsupervised learning is semi-supervised learning, where the teacher gives an incomplete training signal: a training set with some (often many) of the target outputs missing.

2.1.2.3 Reinforcement Learning

When the training data does not explicitly contain the correct output for each input, we are no longer in a supervised learning setting. Consider a toddler learning not to touch a hot cup of tea. The experience of such a toddler would typically comprise a set of occasions when the toddler confronted a hot cup of tea and was faced with the decision of touching it or not touching it. Presumably, every time she touched it, the result was a high level of pain, and every time she didn’t touch it, a much lower level of pain resulted (that of an unsatisfied curiosity). Eventually, the toddler learns that she is better off not touching the hot cup. The training examples did not spell out what the toddler should have done, but they instead graded different actions that she has taken. Nevertheless, she uses the examples to reinforce the better actions, eventually learning what she should do in similar situations. This characterizes reinforcement learning, where the training example does not contain the target output, but instead contains some possible output together with a measure of how good that output is. In contrast to supervised learning where the training examples were of the form (input, correct output), the examples in reinforcement learning are of the form (input, some output, grade for this output). Importantly, the example does not say how good other outputs would have been for this particular input. Reinforcement learning is especially useful for learning how to play a game. Imagine a situation in backgammon where you have a choice between different actions and you want to identify the best action. It is not a trivial task to ascertain what the best action is at a given stage of the game, so we cannot easily create supervised learning examples. If you use reinforcement learning instead, all you need to do is to take some action and report how well things went, and you have a training example. The reinforcement learning algorithm is left with the task of sorting out the information coming from different examples to find the best line of play.

2.1.2.4 Other Views of Learning

The study of learning has evolved somewhat independently in a number of fields that started historically at different times and in different domains, and these fields have developed different emphases and even different jargons. As a result, learning from data is a diverse subject with many aliases in the scientific literature. The main field dedicated to the subject is called machine learning, a name that distinguishes it from human learning. We briefly mention two other important fields that approach learning from data in their own ways. Statistics shares the basic premise of learning from data, namely the use of a set of observations to uncover an underlying process. In this case, the process is a probability distribution and the observations are samples from that distribution. Because statistics is a mathematical field, emphasis is given to situations where most of the questions can be answered with rigorous proofs. As a result, statistics focuses on somewhat idealized models and analyzes them in great detail. This is the main difference between the statistical approach to learning and how we approach the subject here. We make less restrictive assumptions and deal with more general models than in statistics. Therefore, we end up with weaker results that are nonetheless broadly applicable. Data mining is a practical field that focuses on finding patterns, correlations, or anomalies in large relational databases. For example, we could be looking at medical records of patients and trying to detect a cause-effect relationship between a particular drug and long-term effects. We could also be looking at credit card spending patterns and trying to detect potential fraud. Technically, data mining is the same as learning from data, with more emphasis on data analysis than on prediction. Because databases are usually huge, computational issues are often critical in data mining.

2.1.3 Error and noise

We close this chapter by revisiting two notions in the learning problem in order to bring them closer to the real world. The first notion is what approximation means when we say that our hypothesis approximates the target function well. The second notion is about the nature of the target function. In many situations, there is noise that makes the output of f not uniquely determined by the input. What are the ramifications of having such a ’noisy’ target on the learning problem?

2.1.3.1 *Error Measures*

Learning is not expected to replicate the target function perfectly. The final hypothesis g is only an approximation of f. To quantify how well g approximates f, we need to define an error measure that quantifies how far we are from the target.

The choice of an error measure affects the outcome of the learning process. Different error measures may lead to different choices of the final hypothesis, even if the target and the data are the same, since the value of a particular error measure may be small while the value of another error measure in the same situation is large. Therefore, which error measure we use has consequences for what we learn. What are the criteria for choosing one error measure over another? We address this question here. First, let's formalize this notion a bit. An error measure quantifies how well each hypothesis h in the model approximates the target function f, *Error* = $E(h,f)$. While $E(h,f)$ is based on the entirety of *h* and *f*, it is almost universally defined based on the errors on individual input points x. If we define a pointwise error measure $e(h(x),f(x))$, the overall error will be the average value of this pointwise error. So far, we have been working with the classification error $e(h(x),f(x)) = [h(x)f -J(x)]$. In an ideal world, $E(h,J)$ should be user-specified. The same learning task in different contexts may warrant the use of different error measures. One may view $E(h,J)$ as the 'cost' of using *h* when you should use f. This cost depends on what his used for, and cannot be dictated just by our learning techniques. Here is a case in point.

2.1.3.2 *Noisy Targets*

In many practical applications, the data we learn from are not generated by a deterministic target function. Instead, they are generated in a noisy way such that the output is not uniquely determined by the input. For instance, in the credit-card example we presented in Section 1.1, two customers may have identical salaries, outstanding loans, etc., but end up with different credit behavior. Therefore, the credit 'function' is not really a deterministic function, but a noisy one. This situation can be readily modeled within the same framework that we have. Instead of $y = f(x)$, we can take the output *y* to be a random variable that is affected by, rather than determined by, the input *x*. Formally, we have a target distribution $P(yIx)$ instead of a target function $y = f(x)$. A data point (x,y) is now generated by the joint distribution $P(x,y) = P(x)P(y|x)$. One can think of a noisy target as a deterministic target plus added noise. If *y* is real-valued for example, one can take the expected value of *y* given *x* to be the deterministic $f(x)$, and consider $y-f(x)$ as pure noise that is added to *f*. This view suggests that a deterministic target function can be considered a special case of a noisy target, just with zero noise. Indeed, we can formally express any function f as a distribution $P(y|x)$ by choosing $P(y|x)$ to be zero for all y except $y = f(x)$. Therefore, there is no loss of generality if we consider the target to be a distribution rather than a function. Figure 2.2 modifies the previous Figures 2.1 to illustrate the general learning problem, covering both deterministic and noisy targets.

2.2 Machine Learning Models

In this chapter, we present examples of learning from data and formalize the learning problem. We also discuss the main concepts associated with learning,

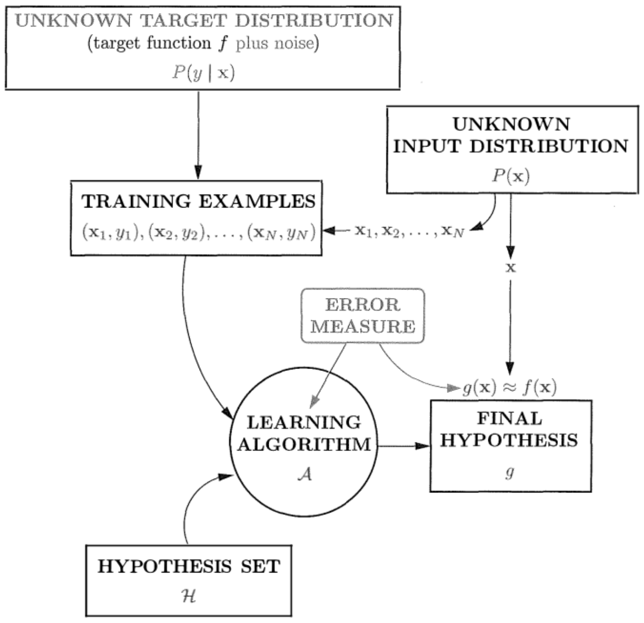

UNKNOWN TARGET DISTRIBUTION
(target function f plus noise)
$P(y \mid \mathbf{x})$
UNKNOWN INPUT DISTRIBUTION
$P(\mathbf{x})$
TRAINING EXAMPLES
$(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_N, y_N)$
$\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N$
$\mathbf{x}$
ERROR MEASURE
$g(\mathbf{x}) \approx f(\mathbf{x})$
LEARNING ALGORITHM
$\mathcal{A}$
FINAL HYPOTHESIS
g
HYPOTHESIS SET
$\mathcal{H}$

Figure 2-2 The general supervised learning problem

and the different paradigms of learning that have been developed.

2.2.1. Linear Models

We often wonder how to draw a line between two categories; right versus wrong, personal versus professional life, useful email versus spam, to name a few. A line is intuitively our first choice for a decision boundary. In learning, as in life, a line is also a good first choice.

In Chapter 2, we (and the machine) learned a procedure to 'draw a line' between two categories based on data (the perceptron learning algorithm). We started by taking the hypothesis set H that included all possible lines (actually hyperplanes). The algorithm then searched for a good line in H by iteratively correcting the errors made by the current candidate line, in an attempt to improveE_{in}. As we saw in Chapter 2, the linear model set of lines has a small VC dimension and so is able to generalize well from $E_{in}$to E_{out}.

The aim of this chapter is to further develop the basic linear model into a powerful tool for learning from data. We branch into three important problems: the classification problem that we have seen and two other important problems called regression and probability estimation. The three problems come with different but related algorithms, and cover a lot of territory in learning from data. As a rule of thumb, when faced with learning problems, it is generally a winning strategy to try a linear model first.

2.2.1.1. Linear Regression

Linear regression is another useful linear model that applies to real-valued target functions. It has a long history in statistics, where it has been studied in great detail, and has various applications in social and behavioral sciences. Here, we discuss linear regression from a learning perspective, where we derive the main results with minimal assumptions.

The Algorithm:

Linear regression is used for finding linear relationship between target and one or more predictors. There are two types of linear regression- Simple and Multiple.

2.2.1.1.1. Simple Linear Regression

Simple linear regression is useful for finding relationship between two continuous variables. One is predictor or independent variable and other is response or dependent variable. It looks for statistical relationship but not deterministic relationship. Relationship between two variables is said to be deterministic if one variable can be accurately expressed by the other. For example, using temperature in degree Celsius it is possible to accurately predict Fahrenheit. Statistical relationship is not accurate in determining relationship between two variables. For example, relationship between height and weight.

The core idea is to obtain a line that best fits the data. The best fit line is the one for which total prediction error (all data points) are as small as possible. Error is the distance between the point to the regression line.

2.2.1.1.2. Logistic Regression

Logistic Regression is all about predictions. Logistic Regression was used in the biological sciences in early twentieth century. It was then used in many social science applications. Logistic Regression is used when the dependent variable(target) is categorical.

Consider a scenario where we need to classify whether an email is spam or not. If we use linear regression for this problem, there is a need for setting up a threshold based on which classification can be done. Say if the actual class is malignant, predicted continuous value 0.4 and the threshold value is 0.5, the data point will be classified as not malignant which can lead to serious consequence in real time.

From this example, it can be inferred that linear regression is not suitable for classification problem. Linear regression is unbounded, and this brings logistic regression into picture. Their value strictly ranges from 0 to 1.

2.2.1.1.3. SVM

Now you have accustomed yourself with linear regression and logistic regression algorithms. Support vector machine is another simple algorithm that every machine learning expert should have in his/her arsenal. Support vector machine is highly preferred by many as it produces significant accuracy with less computation power. Support Vector Machine, abbreviated as SVM can be used for both regression and classification tasks. But it is widely used in classification objectives.

The objective of the support vector machine algorithm is to find a hyperplane in an N-dimensional space (N — the number of features) that distinctly classifies the data points.

To separate the two classes of data points, there are many possible hyperplanes that could be chosen. Our objective is to find a plane that has the maximum margin, i.e. the maximum distance between data points of both classes. Maximizing the margin distance provides some reinforcement so that future data points can be classified with more confidence.

Hyperplanes are decision boundaries that help classify the data points. Data points falling on either side of the hyperplane can be attributed to different classes. Also, the dimension of the hyperplane depends upon the number of features. If the number of input features is 2, then the hyperplane is just a line. If the number of input features is 3, then the hyperplane becomes a two-dimensional plane. It becomes difficult to imagine when the number of features exceeds 3.

In logistic regression, we take the output of the linear function and squash the value within the range of [0,1] using the sigmoid function. If the squashed value is greater than a threshold value (0.5) we assign it a label 1, else we assign it a label 0. In SVM, we take the output of the linear function and if that output is greater than 1, we identify it with one class and if the output is -1, we identify is with another class. Since the threshold values are changed to 1 and -1 in SVM, we obtain this reinforcement range of values ([-1,1]) which acts as margin.

2.2.2. Non-Linear Models

2.2.2.1. Artificial Neural Networks

An artificial neural network (ANN) is the piece of a computing system designed to simulate the way the human brain analyzes and processes information. It is the foundation of artificial intelligence (AI) and solves problems that would prove impossible or difficult by human or statistical standards. ANNs have self-learning capabilities that enable them to produce better results as more data becomes available.

Artificial neural networks are built like the human brain, with neuron nodes interconnected like a web. The human brain has hundreds of billions of cells called neurons. Each neuron is made up of a cell body that is responsible for processing information by carrying information towards (inputs) and away (outputs) from the brain.

An ANN has hundreds or thousands of artificial neurons called processing units, which are interconnected by nodes. These processing units are made up of input and output units. The input units receive various forms and structures of information based on an internal weighting system, and the neural network attempts to learn about the information presented to produce one output report. Just like humans need rules and guidelines to come up with a result or output, ANNs also use a set of learning rules called backpropagation, an abbreviation for backward propagation of error, to perfect their output results.

An ANN initially goes through a training phase where it learns to recognize patterns in data, whether visually, aurally, or textually. During this supervised phase, the network compares its actual output produced with what it was meant to produce—the desired output. The difference between both outcomes is adjusted using backpropagation. This means that the network works backward, going from the output unit to the input units to adjust the weight of its connections between the units until the difference between the actual and desired outcome produces the lowest possible error.

During the training and supervisory stage, the ANN is taught what to look for and what its output should be, using yes/no question types with binary numbers. For example, a bank that wants to detect credit card fraud on time may have four input units fed with these questions: (1) Is the transaction in a different country from the user's resident country? (2) Is the website the card is being used at affiliated with companies or countries on the bank's watch list? (3) Is the transaction amount larger than $2,000? (4) Is the name on the transaction bill the same as the name of the cardholder?

The bank wants the "fraud detected" responses to be Yes Yes Yes No, which in binary format would be 1 1 1 0. If the network's actual output is 1 0 1 0, it adjusts its results until it delivers an output that coincides with 1 1 1 0. After training, the computer system can alert the bank of pending fraudulent transactions, saving the bank lots of money.

2.2.2.2. Convolutional Neural Network

A Convolutional Neural Network (ConvNet/CNN) is a Deep Learning algorithm which can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image and be able to differentiate one from the other. The pre-processing required in a ConvNet is much lower as compared to other classification algorithms.

While in primitive methods filters are hand-engineered, with enough training, ConvNets have the ability to learn these filters/characteristics.

Convolutional layers are the major building blocks used in convolutional neural networks.

A convolution is the simple application of a filter to an input that results in an activation. Repeated application of the same filter to an input results in a map of activations called a feature map, indicating the locations and strength of a detected feature in an input, such as an image.

The innovation of convolutional neural networks is the ability to automatically learn a large number of filters in parallel specific to a training dataset under the constraints of a specific predictive modeling problem, such as image classification. The result is highly specific features that can be detected anywhere on input images.

2.2.2.3. Recurrent Neural Networks

Recurrent Neural Networks (RNNs) add an interesting twist to basic neural networks. A vanilla neural network takes in a fixed size vector as input which limits its usage in situations that involve a 'series' type input with no predetermined size.

RNNs are designed to take a series of input with no predetermined limit on size. One could ask what the big deal is, I can call a regular NN repeatedly too? Sure can, but the 'series' part of the input means something. A single input item from the series is related to others and likely has an influence on its neighbors. Otherwise it's just "many" inputs, not a "series" input (duh!).

Recurrent Neural Network remembers the past and its decisions are influenced by what it has learnt from the past. Note: Basic feed forward networks "remember" things too, but they remember things they learnt during training. For example, an image classifier learns what a "1" looks like during training and then uses that knowledge to classify things in production.

While RNNs learn similarly while training, in addition, they remember things learnt from prior input(s) while generating output(s). It's part of the network. RNNs can take one or more input vectors and produce one or more output vectors and the output(s) are influenced not just by weights applied on inputs like a regular NN, but also by a "hidden" state vector representing the context based on prior input(s)/output(s). So, the same input could produce a different output depending on previous inputs in the series.

2.3 Training Problems and Tuning Methods

2.3.1 Train/dev/test sets

2.3.1.1 Training Vs Testing

Before a final exam, a professor may hand out some practice problems and solutions to the class. Although these problems are not the exact ones that will appear on the exam, studying them will help you do better. They are the 'training set' in your learning.

If the professor's goal is to help you do better in the exam, why not give out the exam problems themselves? Well, nice try. Doing well in the exam is not the goal in and of itself. The goal is for you to learn the course material. The exam is merely a way to gauge how well you have learned the material. If the exam problems are known ahead of time, your performance on them will no longer accurately gauge how well you have learned.

The same distinction between training and testing happens in learning from data. In this chapter, we will develop a mathematical theory that characterizes this distinction. We will also discuss the conceptual and practical implications of the contrast between training and testing.

2.3.1.2 Training Dataset

Training Dataset: The sample of data used to fit the model.

The actual dataset that we use to train the model (weights and biases in the case of Neural Network). The model sees and learns from this data.

2.3.1.3 Validation Dataset

Validation Dataset: The sample of data used to provide an unbiased evaluation of a model fit on the training dataset while tuning model hyperparameters. The evaluation becomes more biased as skill on the validation dataset is incorporated into the model configuration.

The validation set is used to evaluate a given model, but this is for frequent evaluation. We as machine learning engineers use this data to fine-tune the model hyperparameters. Hence the model occasionally *sees* this data, but never does it "*Learn*" from this. We (mostly humans, at-least as of 2017) use the validation set results and update higher level hyperparameters. So, the validation set in a way affects a model, but indirectly.

2.3.1.4 Test Dataset

Test Dataset: The sample of data used to provide an unbiased evaluation of a final model fit on the training dataset.

The Test dataset provides the gold standard used to evaluate the model. It is only used once a model is completely trained (using the train and validation sets). The test set is generally what is used to evaluate competing models (For example on many Kaggle competitions, the validation set is released initially along with the training set and the actual test set is only released when the competition is about to close, and it is the result of the model on the Test set that decides the winner). Many a times the validation set is used as the test set, but it is not good practice. The test set is generally well curated. It contains carefully sampled data that spans the various classes that the model would face, when used in the real world.

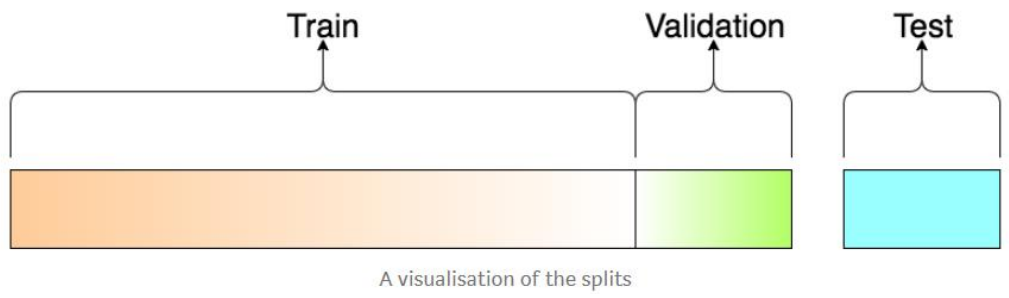


Figure 2-3 Dataset split

About the dataset split ratio

Now that you know what these datasets do, you might be looking for recommendations on how to split your dataset into Train, Validation and Test sets…

This mainly depends on 2 things. First, the total number of samples in your data and second, on the actual model you are training.

Some models need substantial data to train upon, so in this case you would optimize for the larger training sets. Models with very few hyperparameters will be easy to validate and tune, so you can probably reduce the size of your validation set, but if your model has many hyperparameters, you would want to have a large validation set as well (although you should also consider cross validation). Also, if you happen to have a model with no hyperparameters or ones that cannot be easily tuned, you probably don't need a validation set too!

All in all, like many other things in machine learning, the train-test-validation split ratio is also quite specific to your use case and it gets easier to make judge as you train and build more and more models.

Note on Cross Validation: Many a times, people first split their dataset into 2 — Train and Test. After this, they keep aside the Test set, and randomly choose X% of their Train dataset to be the actual ***Train*** *set and the remaining (100-X)% to be the* ***Validation*** *set, where X is a fixed number(say 80%), the model is then iteratively trained and validated on these different sets. There are multiple ways to do this, and is commonly known as Cross Validation. Basically, you use your training set to generate multiple splits of the Train and Validation sets. Cross validation avoids over fitting and is getting more and more popular, with K-fold Cross Validation being the most popular method of cross validation*

2.3.2 Training Procedure

First, we gather and prepare our data. This step is very important because the quality and quantity of data that you gather will directly determine how good your predictive model can be. Data preparation, where we load our data into a suitable place and prepare it for use in our machine learning training. We'll first put all our data together, and then randomize the ordering. We don't want the order of our data to affect what we learn, since that's not part of determining whether a drink is beer or wine. In other words, we make a determination of what a drink is, independent of what drink came before or after it.

Next, the most important step is to choose our model. There are many models that researchers and data scientists have created over the years. Some are very well suited for image data, others for sequences (like text, or music), some for numerical data, others for text-based data. Then we dive into training our models on our machines

The last step is to evaluate our model performance, we use the test set which we split earlier and the model hasn't seen before to evaluate our model and test how it can generalize and predict well.

After those steps, we have our model ready, but there are a few things which we can do to enhance our model performance. So, we will discuss parameter tuning. A lot of parameters can be tuned by us and not learnable parameters by the model such as learning rate, split ratio of the given data, number of neighbors in case of KNNs classifier, maximum depth in case of using Decision Trees. So, many hyperparameters which we need to tune before submit the last phase of our model

2.3.3 Bias and Variance Trade-off

Whenever we discuss model prediction, it's important to understand prediction errors (bias and variance). There is a tradeoff between a model's ability to minimize bias and variance. Gaining a proper understanding of these errors would help us not only to build accurate models but also to avoid the mistake of overfitting and underfitting.

Bias is the difference between the average prediction of our model and the correct value which we are trying to predict. Model with high bias pays very little attention to the training data and oversimplifies the model. It always leads to high error on training and test data.

Variance is the variability of model prediction for a given data point or a value which tells us spread of our data. Model with high variance pays a lot of attention to training data and does not generalize on the data which it hasn't seen before. As a result, such models perform very well on training data but has high error rates on test data.

If our model is too simple and has very few parameters then it may have high bias and low variance. On the other hand, if our model has large number of parameters then it's going to have high variance and low bias. So, we need to find the right/good balance without overfitting and underfitting the data.

This tradeoff in complexity is why there is a tradeoff between bias and variance. An algorithm can't be more complex and less complex at the same time.

To build a good model, we need to find a good balance between bias and variance such that it minimizes the total error. An optimal balance of bias and variance would never overfit or underfit the model. Therefore, understanding bias and variance is critical for understanding the behavior of prediction models.

2.4 Conclusion

In this chapter, we present examples of learning from data and formalize the learning problem. We also discuss the main concepts associated with learning, and the different paradigms of learning that have been developed.

Chapter 3 Reinforcement Learning

Reinforcement learning (RL) is the area of machine learning that deals with sequential decision-making. In this chapter, we describe how the RL problem can be formalized as an agent that has to make decisions in an environment to optimize a given notion of cumulative rewards. It will become clear that this formalization applies to a wide variety of tasks and captures many essential features of artificial intelligence such as a sense of cause and effect as well as a sense of uncertainty and nondeterminism. This chapter also introduces the different approaches
to learning sequential decision-making tasks and how deep RL can be useful.

A key aspect of RL is that an agent *learns* a good behavior. This means that it modifies or acquires new behaviors and skills incrementally. Another important aspect of RL is that it uses trial-and-error *experience* (as opposed to e.g., dynamic programming that assumes full knowledge of the environment a priori). Thus, the RL agent does not require complete knowledge or control of the environment; it only needs to be able to interact with the environment and collect information. In an *offline* setting, the experience is acquired a priori, then it is used as a batch for learning (hence the offline setting is also called batch RL).

This is in contrast to the *online* setting where data becomes available in a sequential order and is used to progressively update the behavior of the agent. In both cases, the core learning algorithms are essentially the same but the main difference is that in an online setting, the agent can influence how it gathers experience so that it is the most useful for learning. This is an additional challenge mainly because the agent has to deal with the *exploration/exploitation* dilemma while learning. But learning in the online setting can also be an advantage since the agent is able to gather information specifically on the most interesting part of the environment. For that reason, even when the environment is fully known, RL approaches may provide the most computationally efficient approach in practice as compared to some dynamic programming methods that would be inefficient due to this lack of specificity.

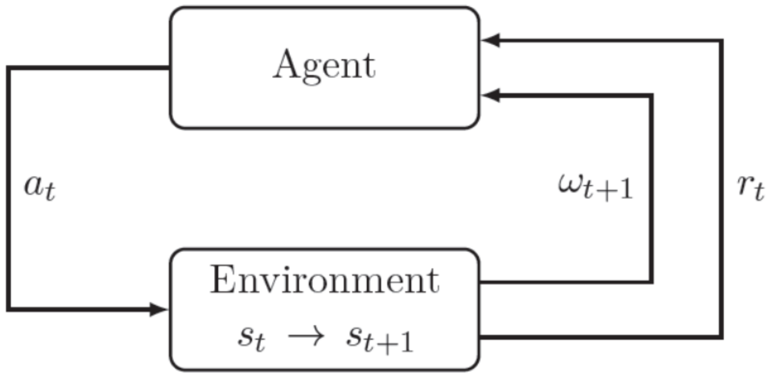


Figure 3-1 RL basic setting

3.1 Formal framework

3.1.1 The reinforcement learning setting

The general RL problem is formalized as a discrete time stochastic control process where an agent interacts with its environment in the following way: the agent starts, in a given state within its environment $s_o \in S$, by gathering an initial observation $o_o \in O$. At each time step t, the agent has to take an action $a_t \in A$. As illustrated in Figure 3-1, it follows three consequences: (i) the agent obtains a reward $r_t \in R$, (ii) the state transitions to s_{t+1}, and (iii) the agent obtains an observation o_{t+1}. This control setting was first proposed by (Bellman R. , 1957b). Here, we review the main elements of RL before delving into deep RL in the following chapters.

3.1.2 The Markov property

The Markov property means that the future of the process only depends on the current observation, and the agent has no interest in looking at the full history. A Markov Decision Process (MDP) (Bellman R. a., 1957a) is a discrete time stochastic control process defined as follows:

Definition 3.1. An MDP is a 5-tuple (S, A, T, R, γ) where:

- S is the state space,
- A is the action space,
- $T : S \times A \times S \to [0,1]$ is the transition function (set of conditional transition probabilities between states),
- $R : S \times A \times S \to R$ is the reward function, where R is a continuous set of possible rewards in range $R_{max} \in R^{+} (e.g., [0, R_{max}])$,
- $\gamma \in [0,1)$ is the discount factor.

The system is fully observable in an MDP, which means that the observation is the same as the state of the environment: $o_t = s_t$. At each time step t, the probability of moving to s_{t+1} is given by the state transition function $T(s_t, a_t, s_{t+1})$ and the reward is given by a bounded reward function $R(s_t, a_t, s_{t+1}) \in \boldsymbol{R}$.

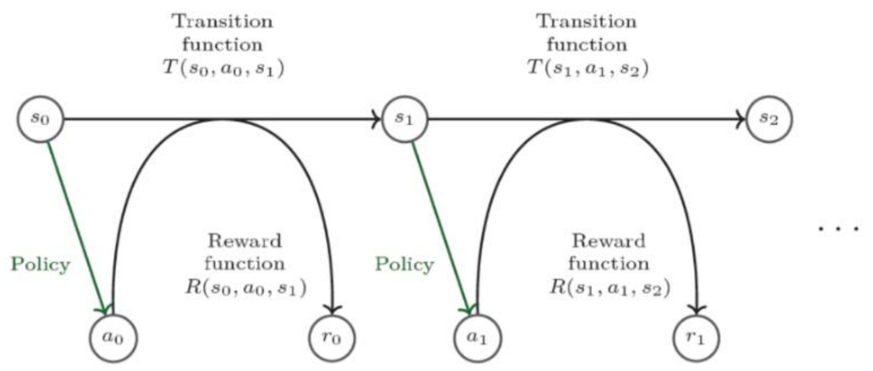

Transition function $T(s_0, a_0, s_1)$
Transition function $T(s_1, a_1, s_2)$
s_0
s_1
s_2
Policy
Reward function $R(s_0, a_0, s_1)$
Policy
Reward function $R(s_1, a_1, s_2)$
a_0
r_0
a_1
r_1
...

Figure 3-2 Illustration of a MDP. At each step, the agent takes an action that changes its state in the environment and provides a reward

3.1.3 Policies

A policy defines how an agent selects actions. Policies can be categorized under the criterion of being either stationary or non-stationary. A non- stationary policy depends on the time-step and is useful for the finite- horizon context where the cumulative rewards that the agent seeks to optimize are limited to a finite number of future time steps (Bertsekas, 1995). In this introduction to deep RL, infinite horizons are considered and the policies are stationary.

Policies can also be categorized under a second criterion of being either deterministic or stochastic:

In the deterministic case, the policy is described by

$$\pi(s) \ : S \rightarrow \ A$$

In the stochastic case, the policy is described by $\pi(s, a) \ : S \ \times \ A \rightarrow [0,1]$ where $\pi(s, a)$ denotes the probability that action *a* may be chosen in state s.

3.1.4 The expected return

Throughout this survey, we consider the case of an RL agent whose goal is to find a policy $\pi(s, a) \in \Pi$, so as to optimize an expected return

$$V^{\pi}(s) : S \rightarrow R$$

(also called V-value function) such that

$$\boldsymbol{V^{\pi}(s) = E\left[\sum_{0}^{\infty} \gamma^{k}\ r_{t+k}\ | s_t = s, , \pi\right]} \quad \textbf{(3.1)}$$

Where,

$$r_t = \underset{a\sim\pi(s_t,.)}{\mathrm{E}}[R(s_t, a, s_{t+1})]$$

$$P(s_{t+1}|s_t, a_t) = T(s_t, a_t, s_{t+1})\ with\ a_t \sim \pi(s_t, .)$$

From the definition of the expected return, the optimal expected return can be defined as:

$$\boldsymbol{V^{*}(s) = \max_{\pi\, \epsilon\, \Pi} V^{\pi}(s)} \quad \textbf{(3.2)}$$

In addition to the V-value function, a few other functions of interest can be introduced.

The Q-value function $Q^{\pi}(s, a) : S \times A \to R$ is defined as follows:

$$\boldsymbol{Q^{\pi}(s) = E\left[\sum_{0}^{\infty} \gamma^{k}\ r_{t+k}\ | s_t = s, a_t = a, \pi\right]} \quad \textbf{(3.3)}$$

This equation can be rewritten recursively in the case of an MDP using Bellman’s equation:

$$\boldsymbol{Q^{\pi}(s, a) = \sum_{s'\, \epsilon\, S} T(s, a, s')\Big(R(s, a, s') + \gamma Q^{\pi}\big(s', a = \pi(s')\big)\Big)} \quad \textbf{(3.4)}$$

Similarly, to the V-value function, the optimal Q-value function $Q^{*}(s, a)$ can also be defined as

$$\boldsymbol{Q^{*}(s, a) = \max_{\pi\, \epsilon\, \Pi} Q^{\pi}(s, a)} \quad \textbf{(3.5)}$$

The particularity of the Q-value function as compared to the V-value function is that the optimal policy can be obtained directly from $Q^*(s, a)$:

$$\boldsymbol{\pi^*(s) = \max_{a \in A} Q^*(s, a)} \qquad \textbf{(3.6)}$$

The optimal V-value function $V^*(s)$ is the expected discounted reward when in a given state s while following the policy π^* thereafter.

The optimal Q-value $Q^*(s, a)$ is the expected discounted return when in a given state *s* and for a given action *a* while following the policy π^* thereafter.

It is also possible to define the advantage function

$$\boldsymbol{A^{\pi}(s, a) = Q^{\pi}(s, a) - V^{\pi}(s)} \qquad \textbf{(3.7)}$$

This quantity describes how good the action a is, as compared to the expected return when following directly policy π.

Note that one straightforward way to obtain estimates of either $V^{\pi}(s), Q^{\pi}(s, a)$ *and* $A^{\pi}(s, a)$ is to use Monte Carlo methods, i.e. defining an estimate by performing several simulations from *s* while following policy π. In practice, we will see that this may not be possible in the case of limited data. In addition, even when it is possible, we will see that other methods should usually be preferred for computational efficiency.

3.2 Different components to learn a policy

An RL agent includes one or more of the following components:

- a representation of a *value function* that provides a prediction of how good each state or each state/action pair is,
- a direct representation of the *policy* $\pi(s)$ or $\pi(s, a)$

- a *model* of the environment (the estimated transition function and the estimated reward function) in conjunction with a planning algorithm.

The first two components are related to what is called *model-free* RL. When the latter component is used, the algorithm is referred to as *model-based* RL.

For most problems approaching real-world complexity, the state space is high-dimensional (and possibly continuous). In order to learn an estimate of the model, the value function or the policy, there are two main advantages for RL algorithms to rely on deep learning:

- Neural networks are well suited for dealing with high-dimensional sensory inputs (such as times series, frames, etc.) and, in practice, they do not require an exponential increase of data when adding extra dimensions to the state or action space.

- In addition, they can be trained incrementally and make use of additional samples obtained as learning happen

3.3 Different settings to learn a policy from data

3.3.1 Offline and online learning

Learning a sequential decision-making task appears in two cases: (i) in the offline learning case where only limited data on a given environment is available and (ii) in an online learning case where, in parallel to learning, the agent gradually gathers experience in the environment. The specificity of the batch setting is that the agent has to learn from limited data without the possibility of interacting further with the environment. In the online setting, the learning problem is more intricate and learning without requiring a large amount of data (sample efficiency) is not only influenced by the capability of the learning algorithm to generalize well from the limited experience. Indeed, the agent has the possibility to gather experience via an *exploration/exploitation strategy*. In addition, it can use a *replay memory* to store its experience so that it can be reprocessed at a later time. In both the batch and the online settings, a supplementary consideration is also the computational efficiency, which, among other things, depends on the efficiency of a given gradient descent step. All these elements will be introduced with more details in the following chapters.

3.3.2 Off-policy and on-policy learning

According to (Sutton R. S., 2017), « on-policy methods attempt to evaluate or improve the policy that is used to make decisions, whereas off-policy methods evaluate or improve a policy different from that used to generate the data ». In off-policy based methods, learning is straightforward when using trajectories that are not necessarily obtained under the current policy, but from a different behavior policy *β* (*s, a*). In those cases, experience replay allows reusing samples from a different behavior policy. On the contrary, on-policy based methods usually introduce a bias when used with a replay buffer as the trajectories are usually not obtained solely under the current policy π. As will be discussed in the following chapters, this makes off-policy methods sample efficient as they are able to make use of any experience; in contrast, on-policy methods would, if specific care is not taken, introduce a bias when using off-policy trajectories.

3.4 Value-based methods for deep RL

The value-based class of algorithms aims to build a value function, which subsequently lets us define a policy. We discuss hereafter one of the simplest and most popular value-based algorithms, the Q-learning algorithm and its variant, the fitted Q-learning, that uses parameterized function approximators. We also specifically discuss the main elements of the deep Q-network (DQN) algorithm (Mnih, et al., 2015) which has achieved superhuman- level control when playing ATARI games from the pixels by using neural networks as function approximators. We then review various improvements of the DQN algorithm and provide resources for further details. At the end of this chapter and in the next chapter, we discuss the intimate link between value-based methods and policy-based methods.

3.4.1 Q-learning

The basic version of Q-learning keeps a lookup table of values $Q(s, a)$ with one entry for every state-action pair. In order to learn the optimal Q-value function, the Q-learning algorithm makes use of the Bellman equation for the Q-value function whose unique solution is $Q*(s, a)$:

$$\boldsymbol{Q*(s,a) = (BQ*)(s,a)} \qquad \textbf{(3.8)}$$

where B is the Bellman operator mapping any function $K : S \times A \rightarrow R$ into another function $S \times A \rightarrow R$ and is defined as follows:

$$(\boldsymbol{BK})(\boldsymbol{s},\boldsymbol{a}) = \sum_{\boldsymbol{s}' \in \boldsymbol{S}} \boldsymbol{T}(\boldsymbol{s},\boldsymbol{a},\boldsymbol{s}')(\boldsymbol{R}(\boldsymbol{s},\boldsymbol{a},\boldsymbol{s}') + \boldsymbol{\gamma} \max_{\boldsymbol{a}' \in \boldsymbol{A}} \boldsymbol{K}(\boldsymbol{s}',\boldsymbol{a}')) \qquad \textbf{(3.9)}$$

By Banach's theorem, the fixed point of the Bellman operator exists since it is a contraction mapping. In practice, one general proof of convergence to the optimal value function is available (Watkins, 1992) under the conditions that:

- the state-action pairs are represented discretely, and
- all actions are repeatedly sampled in all states (which ensures sufficient exploration, hence not requiring access to the transition model). (Hasselt, H., Guez, & Silver., 2016)

This simple setting is often inapplicable due to the high-dimensional (possibly continuous) state-action space. In that context, a parameterized value function $Q(s,a;\ \theta)$ is needed, where ϑ refers to some parameters that define the Q-values.

3.4.2 Fitted Q-learning

Experiences are gathered in a given dataset *D* in the form of tuples $< s,a,r,s' >$ where the state at the next time-step s' is drawn from $T\ (s,a,.)$ and the reward *r* is given by $R(s,a,s')$. In fitted Q-learning, the algorithm starts with some random initialization of the Q-values $Q(s,a;\theta_o)$ where θ_o refers to the initial parameters (usually such that the initial Q-values should be relatively close to 0 so as to avoid slow learning). Then, an approximation of the Q-values at the *k*th iteration $Q(s,a;\theta_k)$ is updated towards the target value

$$\boldsymbol{Y_k^Q} = \boldsymbol{r} + \boldsymbol{\gamma} \max_{\boldsymbol{a}' \in \boldsymbol{A}} \boldsymbol{Q}(\boldsymbol{s}',\boldsymbol{a}';\boldsymbol{\theta_k}) \qquad \textbf{(3.10)}$$

where ϑ_k refers to some parameters that define the Q-values at the *k*th iteration.

In neural fitted Q-learning (NFQ), the state can be provided as an input to the Q-network and a different output is given for each of the possible actions. This provides an efficient structure that has the advantage of obtaining the computation of $\max_{a' \in A} Q(s',a';\theta_k)$ in a single forward pass in the neural network for a given s'.

The Q-values are parameterized with a neural network $Q(s, a; \theta_k)$ where the parameters θ_k are updated by stochastic gradient descent (or a variant) by minimizing the square loss:

$$\boldsymbol{L_{DQN} = (Q(s, a; \theta_k) - Y_k^Q)^2} \tag{3.11}$$

Thus, the Q-learning update amounts in updating the parameters:

$$\boldsymbol{\theta_{k+1} = \theta_k + \alpha(Y_k^Q - Q(s, a; \theta_k))\nabla_{\theta_k} Q(s, a; \theta_k)} \tag{3.12}$$

where α is a scalar step size called the learning rate. Note that using the square loss is not arbitrary. Indeed, it ensures that $Q(s, a; \theta_k)$ should tend without bias to the expected value of the random variable Y^Q. Hence, it ensures that $Q(s, a; \theta_k)$ should tend to $Q^*(s, a)$ after many iterations in the hypothesis that the neural network is well-suited for the task and that the experience gathered in the dataset *D* is sufficient.

When updating the weights, one also changes the target. Due to the generalization and extrapolation abilities of neural networks, this approach can build large errors at different places in the state-action space. Therefore, the contraction mapping property of the Bellman operator in Equation is not enough to guarantee convergence. It is verified experimentally that these errors may propagate with this update rule and, as a consequence, convergence may be slow or even unstable. Another related damaging side-effect of using function approximators is the fact that Q-values tend to be overestimated due to the max operator (Hasselt, H., Guez, & Silver., 2016) Because of the instabilities and the risk of overestimation, specific care has be taken to ensure proper learning

3.4.3 Deep Q-networks

Leveraging ideas from NFQ, the deep Q-network (DQN) algorithm introduced by (Mnih, et al., 2015) is able to obtain strong performance in an online setting for a variety of ATARI games, directly by learning from the pixels. It uses two heuristics to limit the instabilities:

- The target Q-network in Equation is replaced by $Q(s', a'; \theta_k^-)$ where its parameters θ_k^- are updated only every *N* iterations with the following assignment $\theta_k^- = \theta_k$ this prevents the instabilities and to propagate quickly and it reduces the risk of divergence as the target values Y_k^Q are kept fixed for N iterations. The idea of target networks can be seen as instantiation of fitted Q-learning, where each period between target network updates corresponds to single fitted Q-iteration.

- In an online setting, the replay memory keeps all information for the last $N_{replay} \in N$ time steps, where the experience is collected by following an ∈-greedy policy. The updates are then made on a set of tuples $< s, a, r, s' >$ (called mini-batch) selected randomly within the replay memory. This technique allows for updates that cover a wide range of the state- action space. In addition, one mini-batch update has less variance compared to a single tuple update. Consequently, it provides the possibility to make a larger update of the parameters, while having an efficient parallelization of the algorithm.

In addition to the target Q-network and the replay memory, DQN uses other important heuristics. To keep the target values in a reasonable scale and to ensure proper learning in practice, rewards are clipped between -1 and +1. Clipping the rewards limits the scale of the error derivatives and makes it easier to use the same learning rate across multiple games (however, it introduces a bias). In games where the player has multiple lives, one trick is also to associate a terminal state to the loss of a life such that the agent avoids these terminal states (in a terminal state the discount factor is set to 0).

In DQN, many deep learning specific techniques are also used. In particular, a preprocessing step of the inputs is used to reduce the input dimensionality, to normalize inputs (it scales pixels value into [-1,1]) and to deal with some specificities of the task. In addition, convolutional layers are used for the first layers of the neural network function approximator and the optimization is performed using a variant of stochastic gradient descent called RMSprop.

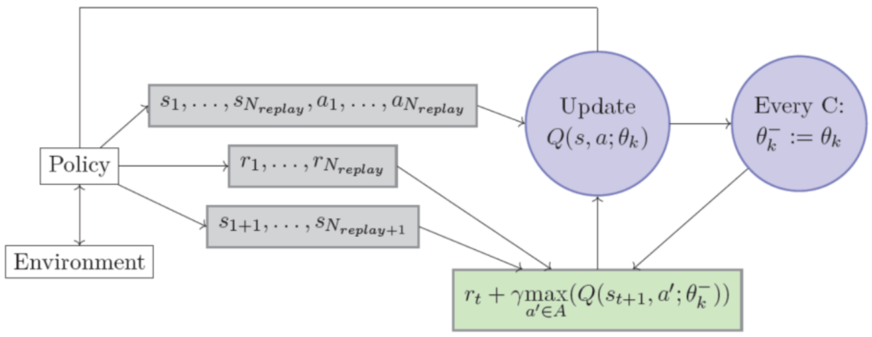

$s_1, \ldots, s_{N_{replay}}, a_1, \ldots, a_{N_{replay}}$
Update $Q(s, a; \theta_k)$
Every C: $\theta_k^- := \theta_k$
Policy
$r_1, \ldots, r_{N_{replay}}$
$s_{1+1}, \ldots, s_{N_{replay+1}}$
Environment
$r_t + \gamma \max_{a' \in A}(Q(s_{t+1}, a'; \theta_k^-))$

Figure 3-3 DDQN learning architecture

3.4.4 Double DQN

The max operation in Q-learning uses the same values both to select and to evaluate an action. This makes it more likely to select overestimated values in case of inaccuracies or noise, resulting in overoptimistic value estimates. Therefore, the DQN algorithm induces an upward bias. The double estimator method uses two estimates for each variable, which allows for the selection of an estimator and its value to be uncoupled. Thus, regardless of whether errors in the estimated Q-values are due to stochasticity in the environment, function approximation, non-stationarity, or any other source, this allows for the removal of the positive bias in estimating the action values. In Double DQN, or DDQN (Hasselt, H., Guez, & Silver., 2016), the target value Y_k^Q is replaced by

$$\boldsymbol{Y_k^{DDQN} = r + \gamma\, Q(s', \underset{a \in A}{\operatorname{argmax}} Q(s', a;\ \theta_k);\ \theta_k^-)} \qquad \textbf{(3.13)}$$

which leads to less overestimation of the Q-learning values, as well as improved stability, hence improved performance. As compared to DQN, the target network with weights ϑ_t^- are used for the evaluation of the current greedy action. Note that the policy is still chosen according to the values obtained by the current weights θ.

3.5 Policy gradient methods for deep RL

This section focuses on a particular family of reinforcement learning algorithms that use policy gradient methods. These methods optimize a performance objective (typically the expected cumulative reward) by finding a good policy (e.g. a neural network parameterized policy) thanks to variants of stochastic gradient ascent with respect to the policy parameters. Note that policy gradient methods belong to a broader class of policy-based methods that includes, among others, evolution strategies. These methods use a learning signal derived from sampling instantiations of policy parameters and the set of policies is developed towards policies that achieve better returns. In this chapter, we introduce the stochastic and deterministic gradient theorems that provide gradients on the policy parameters in order to optimize the performance objective. Then, we present different RL algorithms that make use of these theorems.

3.5.1 Stochastic Policy Gradient

The expected return of a stochastic policy π starting from a given state s_o can be written as (Sutton, A. McAllester, P. Singh, & Mansour, 2000):

$$\boldsymbol{V^{\pi}(s_o) = \int_S \rho^{\pi}(s) \int_A \pi(s,a)\, R'(s,a)\, dads,} \tag{3.14}$$

Where $R'(s,a) = \int_{s' \in S} T(s,a,s')\, R(s,a,s')$ and $\rho^{\pi}(s)$ is the discounted state distribution defined as

$$\boldsymbol{\rho^{\pi}(s) = \sum_{t=0}^{\infty} \gamma^t \, \mathrm{Pr}\{s_t = s \mid s_o\, , \pi\}} \tag{3.15}$$

For a differentiable policy π_w, the fundamental result underlying these algorithms is the policy gradient theorem (Sutton, A. McAllester, P. Singh, & Mansour, 2000):

$$\boldsymbol{\nabla_w\, V^{\pi_w}\, (s_o) = \int_S \rho^{\pi}(s) \int_A \nabla_w\, \pi_w(s,a)\, Q^{\pi_w}(s,a)\, dads,} \tag{3.16}$$

This result allows us to adapt the policy parameters w: $\Delta w \;\propto\; \nabla_w\, V^{\pi_w}$ from experience. This result is particularly interesting since the policy gradient does not depend on the gradient of the state distribution (even though one might have expected it to). The simplest way to derive the policy gradient estimator (i.e., estimating $\nabla_w\, V^{\pi_w}(s_o)$ from experience) is to use a *score function gradient estimator*, commonly known as the REINFORCE algorithm (Williams & J., 1992).The likelihood ratio trick can be exploited as follows to derive a general method of estimating gradients from expectations:

$$\boldsymbol{\begin{aligned} \nabla_w\, \pi_w(s,a) &= \pi_w(s,a) \frac{\nabla_w\, \pi_w(s,a)}{\pi_w(s,a)} \\ &= \pi_w(s,a)\, \nabla_w \log(\pi_w(s,a)) \end{aligned}} \tag{3.17}$$

It follows that

$$\boldsymbol{\nabla_w V^{\pi_w}(s_o) = E_{s\sim\rho^{\pi_w},a\sim\pi_w}[\nabla_w \log(\pi_w(s,a))\, Q^{\pi_w}(s,a)\,]} \qquad \textbf{(3.18)}$$

Note that, in practice, most policy gradient methods effectively use undiscounted state distributions, without hurting their performance (Thomas & P., 2014). So far, we have shown that policy gradient methods should include a policy evaluation followed by a policy improvement. On the one hand, the policy evaluation estimates Q^{π_w}. On the other hand, the policy improvement takes a gradient step to optimize the policy $\pi_w(s,a)$ with respect to the value function estimation. Intuitively, the policy improvement step increases the probability of the actions proportionally to their expected return.

The question that remains is how the agent can perform the policy evaluation step, i.e., how to obtain an estimate of $Q^{\pi_w}(s,a)$. The simplest approach to estimating gradients is to replace the Q function estimator with a cumulative return from entire trajectories. In the Monte- Carlo policy gradient, we estimate the $Q^{\pi_w}(s,a)$ from rollouts on the environment while following policy π_w. The Monte-Carlo estimator is an unbiased well-behaved estimate when used in conjunction with the backpropagation of a neural network policy, as it estimates returns until the end of the trajectories (without instabilities induced by bootstrapping). However, the main drawback is that the estimate requires on-policy rollouts and can exhibit high variance. Several rollouts are typically needed to obtain a good estimate of the return. A more efficient approach is to instead use an estimate of the return given by a value-based approach, as in actor-critic methods. We make two additional remarks. First, to prevent the policy from becoming deterministic, it is common to add an entropy regularizer to the gradient. With this regularizer, the learnt policy can remain stochastic. This ensures that the policy keeps exploring.

Second, instead of using the value function Q^{π_w}, an advantage value function A^{π_w} can also be used. While $Q^{\pi_w}(s,a)$ summarizes the performance of each action for a given state under policy π_w, the advantage function $A^{\pi_w}(s,a)$ provides a measure of comparison for each action to the expected return at the state (*s)*, given by $V^{\pi_w}(s)$. $A^{\pi_w}(s,a) = Q^{\pi_w}(s,a) - V^{\pi_w}(s)$ has usually lower magnitudes than $Q^{\pi_w}(s,a)$. This helps reduce the variance of the gradient estimator $\nabla_w V^{\pi_w}(s_o)$ in the policy improvement step, while not modifying the expectation. In other words, the value function $V^{\pi_w}(s)$ can be seen as a *baseline* or *control variate* for the gradient estimator. When updating the neural network that fits the policy, using such a baseline allows for improved numerical efficiency – i.e. reaching a given performance with fewer updates – because the learning rate can be bigger.

3.5.2 Deterministic Policy Gradient

The policy gradient methods may be extended to deterministic policies. The Neural Fitted Q Iteration with Continuous Actions (NFQCA) (Hafner, R., & Riedmiller, 2011) and the Deep Deterministic Policy Gradient (DDPG) (Silver, et al., 2014) algorithms introduce the direct representation of a policy in such a way that it can extend the NFQ and DQN algorithms to overcome the restriction of discrete actions. Let us denote by $\pi(s)$ the deterministic policy $\pi(s) : S \rightarrow A$. In discrete action spaces, a direct approach is to build the policy iteratively with:

$$\boldsymbol{\pi_{k+1}(s) = \underset{a \in A}{\mathbf{argmax}}\, Q^{\pi_k}(s, a),} \qquad \textbf{(3.19)}$$

where π_k is the policy at the k^{th} iteration. In continuous action spaces, a greedy policy improvement becomes problematic, requiring a global maximization at every step. Instead, let us denote by $\pi_\omega(s)$ a differentiable deterministic policy. In that case, a simple and computationally attractive alternative is to move the policy in the direction of the gradient of Q, which leads to the Deep Deterministic Policy Gradient (DDPG) algorithm:

$$\boldsymbol{\nabla_w V^{\pi_w}(s_o) = E_{s\sim\rho^{\pi_w}}[\nabla_w \pi_w \nabla_a(Q^{\pi_w}(s, a))\, |_{a=\pi_w(s)}]} \qquad \textbf{(3.20)}$$

This equation implies relying on $\nabla_a Q^{\pi_w}(s, a)$ (in addition to $\nabla_w \pi_w$), which usually requires using actor-critic methods.

3.5.3 Actor-Critic Methods

As we have seen, a policy represented by a neural network can be updated by gradient ascent for both the deterministic and the stochastic case. In both cases, the policy gradient typically requires an estimate of a value function for the current policy. One common approach is to use an actor-critic architecture that consists of two parts: an actor and a critic. The actor refers to the policy and the critic to the estimate of a value function (e.g., the Q-value function). In deep RL, both the actor and the critic can be represented by non-linear neural network function approximators (Mnih, et al., 2016) . The actor uses gradients derived from the policy gradient theorem and adjusts the policy parameters w. The critic, parameterized by θ, estimates the approximate value function for the current policy π: $Q(s, a; \theta) \approx Q^\pi(s, a)$.

3.5.4 The critic

From a (set of) tuples < *s, a, r, s*0 >, possibly taken from a replay memory, the simplest off-policy approach to estimating the critic is to use a pure bootstrapping algorithm *TD*(0) where, at every iteration, the current value $Q(s, a; \theta)$ is updated towards a target value:

This approach has the advantage of being simple, yet it is not computationally efficient as it uses a pure bootstrapping technique that is prone to instabilities and has a slow reward propagation backwards in time. This is similar to the elements discussed in the value-based methods. The ideal is to have an architecture that is

- sample-efficient such that it should be able to make use of both off-policy and on-policy trajectories (i.e., it should be able to use a replay memory), and
- computationally efficient: it should be able to profit from the stability and the fast reward propagation of on-policy methods for samples collected from near on-policy behavior policies.

3.5.5 The actor

The off-policy gradient in the policy improvement phase for the stochastic case is given as:

$$\nabla_w V^{\pi_w}(s_o) = E_{s\sim\rho^{\pi_\beta}, a\sim\pi_\beta}[\nabla_\theta \log(\pi_w(s,a))\,(Q^{\pi_w}(s,a))\,] \qquad \textbf{(3.21)}$$

where β is a behavior policy generally different than, which makes the gradient generally biased. This approach usually behaves properly in practice but the use of a biased policy gradient estimator makes difficult the analysis of its convergence without the GLIE assumption (Munos, R., Stepleton, Harutyunyan, & Bellemare., 2016).

In the case of actor-critic methods, an approach to perform the policy gradient on-policy without experience replay has been investigated with the use of asynchronous methods, where multiple agents are executed in parallel and the actor-learners are trained asynchronously (Mnih, et al., 2016). The parallelization of agents also ensures that each agent experiences different parts of the environment at a given time step. In that case, n-step returns can be used without introducing a bias. This simple idea can be applied to any learning algorithm that requires on-policy data and it removes the need to maintain a replay buffer. However, this asynchronous trick is not sample efficient. An alternative is to combine off-policy and on-policy samples to trade-off both the sample efficiency of off-policy methods and the stability of on-policy gradient estimates.

Chapter 4 Simulator

3.1 CARLA Simulator

Since Training in the physical world is not feasible. We can't learn the car learn in the physical world, this will cause many of accidents and problems. So, we need to have a virtual urban driving simulator. So, we go to CARLA Simulator, One of the best simulators available. It has an online support from the team and there is papers published based on it from the top universities and companies. CARLA has been developed from the ground up to support development, training, and validation of autonomous driving systems. In addition to open-source code and protocols, CARLA provides open digital assets (urban layouts, buildings, vehicles) that were created for this purpose and can be used freely. The simulation platform supports flexible specification of sensor suites, environmental conditions, full control of all static and dynamic actors, maps generation and much more.

3.2 Highlighted features

- **Scalability via a server multi-client architecture:** multiple clients in the same or in different nodes can control different actors.
- **Flexible API:** CARLA exposes a powerful API that allows users to control all aspects related to the simulation, including traffic generation, pedestrian behaviors, weathers, sensors, and much more.
- **Autonomous Driving sensor suite:** users can configure diverse sensor suites including LIDARs, multiple cameras, depth sensors and GPS among others.
- **Fast simulation for planning and control**: this mode disables rendering to offer a fast execution of traffic simulation and road behaviors for which graphics are not required.

- **Maps generation:** users can easily create their own maps following the Open Drive standard via tools like Roadrunner.
- **Traffic scenarios simulation:** our engine Scenario Runner allows users to define and execute different traffic situations based on modular behaviors.
- **ROS integration:** CARLA is provided with integration with ROS via our ROS-bridge

Autonomous Driving baselines: we provide Autonomous Driving baselines as runnable agents in CARLA, including an Auto Ware agent and a Conditional Imitation Learning agent.

Chapter 5 Previous Imitation Learning Implementations (Intel's and NVidia's work)

In this chapter we will explain the previous methods of implementing the supervised learning approach (Imitation learning) and explain the architecture layers and the training setup used and all details related to the implementation and tuning.

5.1 Model architecture

The previous work of imitation learning was mainly related to two works: NVidia architecture and Intel architecture. In fact, the most recent work was a combination of both of them. The very first implementation of the imitation learning model is separated into two modules as inputs: photo-processing module, measurements processing module and the command module as shown in figure 5-1 on the left. The photo-processing module is the NVidia's architecture. This architecture takes the command input directly as a feature vector, or in Carla it's a single value represents the desired high-level command e.g. 2 means turn left (High Level Command is a CARLA auto generated signal that provides the agent with the optimal direction related to the optimal path). The high-level command in reality is obtained from GPS sensor, which guides the car with commands like: turn left and etc. This implementation is separated into one module as an output: control module. The second improvement provided by Intel shown in figure 5-1 on the right. The control module instead of using 1 control module the model uses control module consists of 4 identical parallel branches each branch has a specific weights and identifies which branch will be activated for this frame which mean that when the signal generated by CARLA is "turn right" for example the control module first branch will be activated and when the signal generated by CARLA is "turn left" for example the control module second branch will be activated.

The imitation learning module details is shown in table 1.

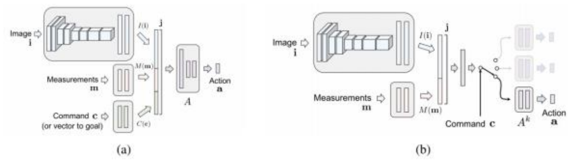


Figure 5-1 Unbranched and Branched architectures

Table 5-1: CIL Network Implementation details

Module	Layer no.	Layer	No. Filters/ Neurons	Filter Size	Stride	Dropout
Image Module	1	Conv	3	5x5	2	0.2
	2	Conv	24	5x5	1	0.2
	3	Conv	36	5x5	2	0.2

	4	Conv	48	3x3	1	0.2
	5	Conv	64	3x3	2	0.2
	6	FC	512	-	-	0.5
	7	FC	512	-	-	0.5
Speed Module	1	FC	128	-	-	0.5
	2	FC	128	-	-	0.5
	3	FC	128	-	-	0.5
Decision Module	1	FC	256	-	-	0.5
	2	FC	256	-	-	0.5
	3	FC	128	-	-	0.5
	4	FC	3	-	-	0.5

5.2 Branched vs. Unbranched architecture

Previous work used the branched architecture instead of the unbranched architecture shown in figure 5-1 on left because it has the advantage of taking the planner command into consideration in all cases, unlike the unbranched which not forced to obey the planner command in all cases which leads to a suboptimal performance The branched architecture is a conditional imitation learning because the network parts are activated by a condition (in our case the planner command is the condition). This new approach causes a complex training setup and complexity in the data preprocessing, the new approach prevents us from using many of the API built-in functions due to the branching complexity so we implemented some training functions from scratch.

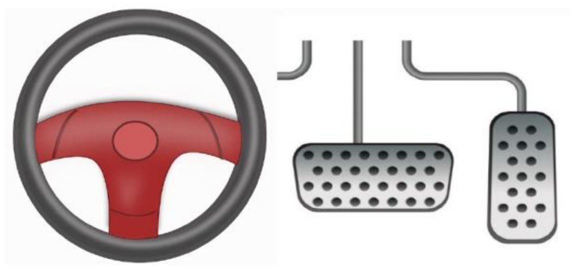


Figure 5-2 Control Signals

5.3 Signals range

The outputs of the network are normalized and mapped to the values shown in table 2.

Table 5-2: Labels information

Control Signal	Range	Description
Steering Range	[-1,1]	Positive: Rotation to the right Negative: Rotation to the left
Gas	[0,1]	Zero: no pressing One: full pressing
Brake	[0,1]	Zero: no pressing One: full pressing

5.4 Speed branch

It’s clear that the three control signals (steering angle – throttle – brake) are enough to control the car and there was no need to predict the speed in controlling the car. However, the speed was predicted to ensure that the car operates correctly, each frame we compare the current speed provided by the simulator with the predicted speed and if the difference between them are above a certain threshold we modify the control signals by adding offset to the gas otherwise we don’t change anything. This problem is called the fake stopping problem, when the car stuck and stays in this situation for a long time without logical reason; therefore, the speed is predicted with separated branch to solve this problem not to pass it as a control signal.

5.5 Training Setup

We can consider the network as two parts: first is the feature extraction part which includes the image module, speed module and concatenation layers, second

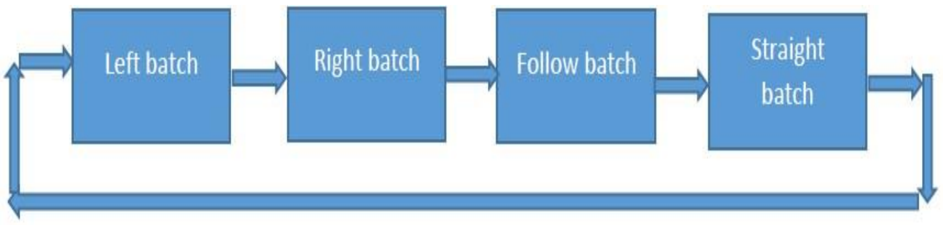


Figure 5-3 Training loop

is the decision module which includes the four branches (left branch- right branch -follow branch-straight branch)? The training process will be iterative process as shown in figure 5-3.

The network is trained using firstly, left batch and mask the 3 other branches then we will train it using right batch and mask the 3 other branches and the same thing with the follow and straight branches, this process will be repeated over the whole dataset to make one epoch be biased towards one task than another, to force the feature extraction part to extract features generally for all tasks This training setup helped us to achieve great results and amazing generalization in all scenes, although the training setup causes a slow train and high CPU dependency (due to the iterative method) ,it's the best setup to enhance the performance. Note that the speed branch will be trained in all batches because it's independent on the planner command.

5.6 Weighted sum loss function

The loss function is an important part that we must choose it carefully. The absolute error between the network predictions and the true labels was chosen, but this wasn't enough to enhance the performance. Previous work tried to give each control signal a specific weight expresses its importance in the optimization process Equation 1 shows the weighted sum loss function used.

$$Loss = 0.95(0.45\ Steer + 0.45\ Gas + 0.45\ Brake) + 0.05\ Speed$$

They notice that the steering angle and gas has higher weight than the brake or the speed branch because the main control signals that affect the performance are the steering angle and gas. After applying this loss function the performance improved and the model collisions and mistakes decreased as expected.

5.7 Model Convergence

To indicate that the model converged or not they apply the fitting test. They used some training sample to test the model if the loss with high then the model needs more time to train and learn, if the loss was small value then they indicate that the model fits the data so we need to test the overfitting to prevent the model

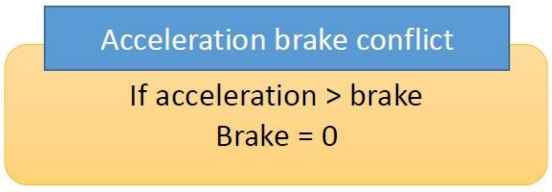


Figure 5-4 Acceleration brake conflict

from the bad generalization capability, they tested the overfitting using real frames don't included in the dataset and check the performance, if the performance was bad the training must be stopped and try to improve the performance by taking earlier epoch result or by the common techniques like regularization and dropout. Leaving the model in training stage for a long time causes overfitting, therefore the model should be tested after almost each epoch to prevent the resources waste and to speed up the train process. The overfitting problem is the main supervised learning algorithms challenge, the more data we train using it the less overfitting problem occurs but there are some other effective parameters that affect the overfitting like training setup, dropout and data balancing and augmentation.

5.8 Soft processing on the control signal

In some cases, the network outputs maybe not logical or maybe not suitable to the physical problems like the gas brake conflict and other cases so we sometimes modify these outputs to prevent the network output from being incompatible.

5.8.1 Acceleration brake conflict

In this case the acceleration value and the brake value are not compatible, therefore in case of the acceleration is higher than the brake it was forced that the brake to be zero before providing it to the simulator.

5.8.2 Speed Limitation

The speed is limited to certain values to satisfy the town rules, therefore they prevented the network from break this rule be deactivating the acceleration signal in case of the car passed the speed limit.

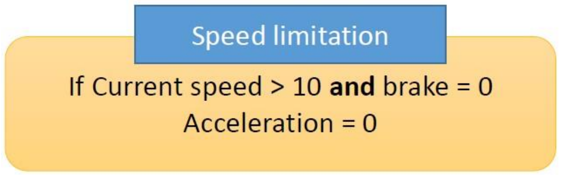


Figure 5-5 Speed limitation

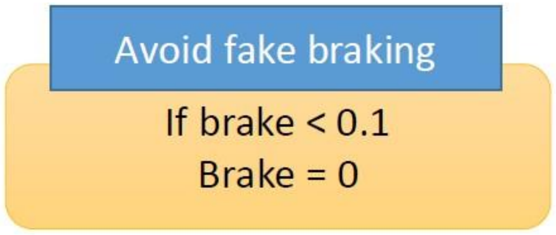


Figure 5-6 Avoid Fake Braking

5.8.3 Avoid Fake Braking

When the brake is below a certain value, they consider it as an error and force it to be equal to zero.

5.9 Conclusion

In this part we discussed the previous work in imitation learning architecture and training setup which lead to the amazing results reached. In the next chapter, we will discuss out implementations as well as other details we added in the imitation learning.

Chapter 6 Data Description and Pre-processing

In imitation networks we used **two datasets:**

- The first one collected by CARLA developers and used for the time dependency approach networks
- The second one we collected which include LiDAR point cloud for each frame

As mentioned in the previous chapter, the IL technique mainly depends on the training data, thus having a consistent, balanced and clean data is a game changer for the overall model performance. This chapter handles the description of the training data before and after preprocessing.

6.1 CARLA developers published Dataset Description

The dataset has the following characteristics before any processing:

- It was collected with the simulator running on 10 fps.
- It's compressed in HDF (.H5 format) files, each file containing 200 frames. And each frame has 28 measurements that are taken out of the simulator.
- It contains about 3290 files i.e. approximately 658,000 frames (Training points).
- Each frame is represented as an RGB image (3 channels).
- Each channel has a resolution of 200 x 88.

6.2 Visualizing a single data point

Here we are visualizing a single data point. Again, it consists of an RGB formatted image with dimensions 200 pixels x 88 pixels. And each RGB image has 28 measurements. It can easily be noticed that we are interested in five measurements only, thus the rest are left for future development.

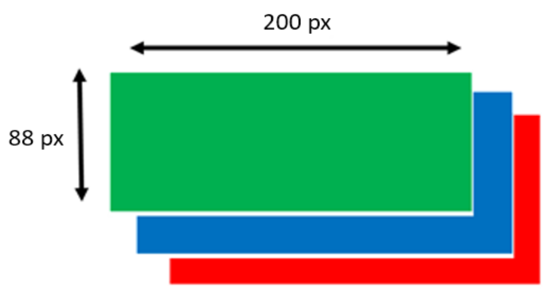


Figure 6-1 RGB Image of a single data file

Table 6-1: Measurements Vector

index	1	2	3				11				25		
Label	Steer	Gas	Brake				Speed				High level command		
Data type	Float	Float	Float				Float				Integer		
Range	[-1,1]	[-1,1]	[0,1]				[-19, 83]				{2,3,4,5}		

6.3 Visualizing s single data file

Here we are visualizing a single data file, figure 6-2 shows that each file has 200 data points as mentioned before.

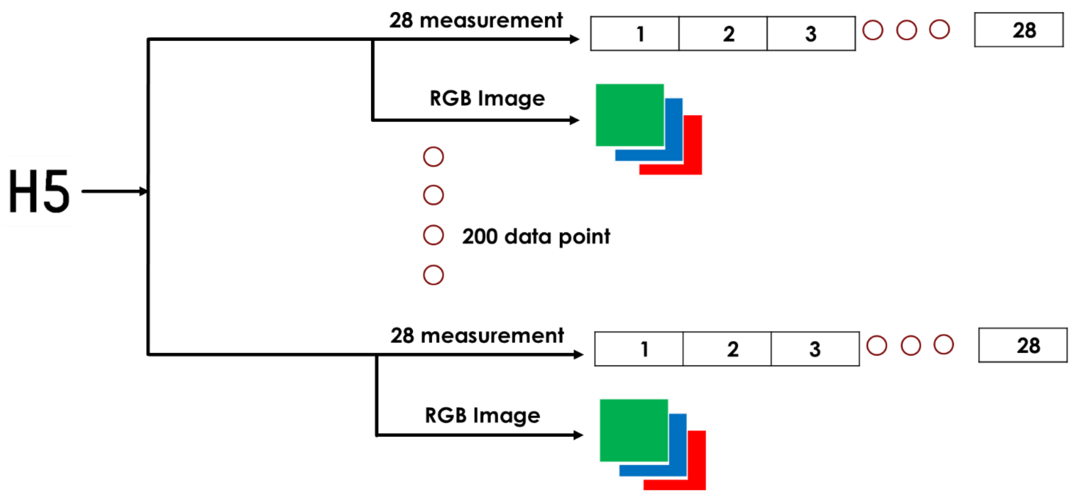


Figure 6-2 Visualization of a single data file

6.4 Notes regarding the data

6.4.1 HDF (H5) Format

H5 file is a data file saved in the Hierarchical Data Format (HDF). It contains multidimensional arrays of scientific data. The following figure 6-3 shows its architecture:

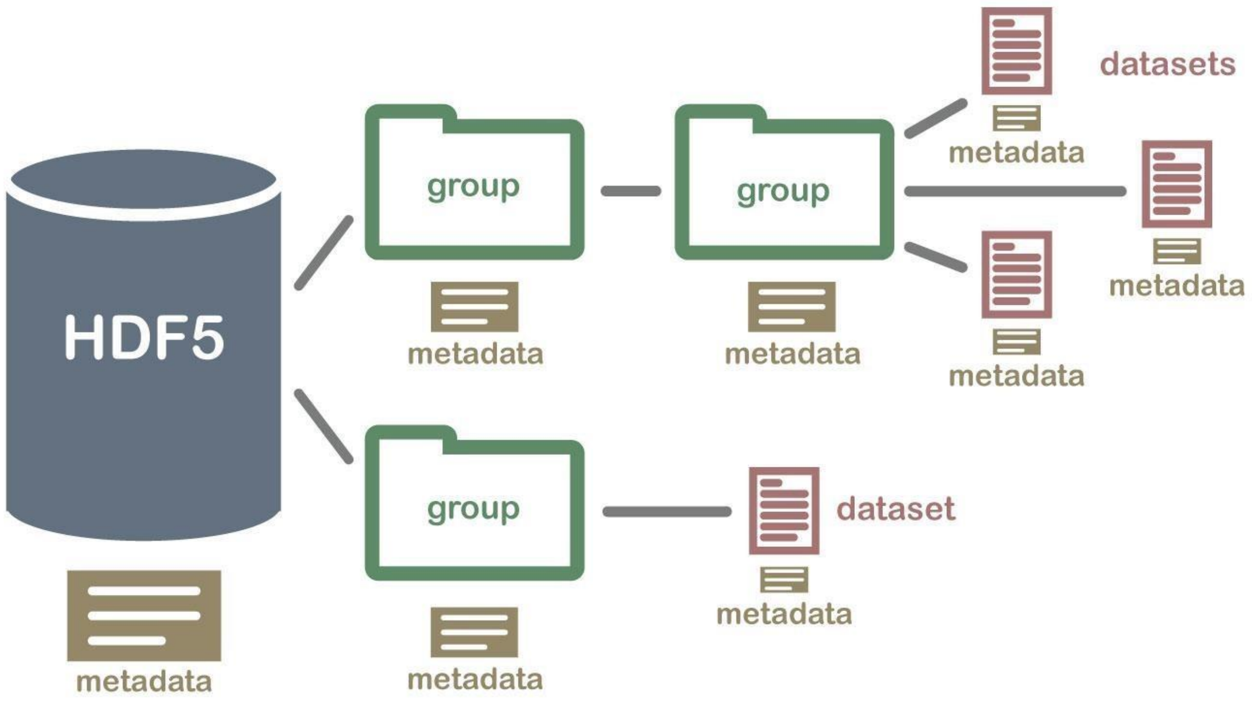


Figure 6-3 HDF Files Architecture

which is very suitable since the data is in hierarchical format.

6.4.2 RGB Images

It’s accurate that the RGB images raise the necessity of a more complex model and more memory cost, however they offer more features to extract.

6.4.3 Measurements Vector

Not all the 28 measurement is used, only five of them, the rest of them is intentionally left for future development

6.4 High Level Command

For the self-driving car at point A to reach point B, there has to be a 3rd party supporting software independent of the car itself to accomplish two tasks in sequence:

- Routing: Choosing the path.
- Navigation: Navigating through the path i.e. generating one of four signals: (Follow the lane, turn right, turn left, and keep straight).

This is what the planner does. The planner basically routes and generates the High command signals. This can be easily shown in the following table.

Table 6-2: High Level Command

Value	Command
2	Follow
3	Left
4	Right
5	Straight

6.4.5 Statistics

The below table shows some statistics about the measurements.

Table 6-3: Data Statistics

Description	Steer	Gas	Brake	Speed
Mean	0.00163	0.54	0.175	18
Standard Deviation	0.1559	0.33	0.38	14.463
Minimum	-1	0	0	-18.73
Maximum	1	1	1	83

6.4.6 Cropping

The images are cropped to 200 x 88. This step crops the sky and the Engine hood which are not effective in the decision. This also slightly helps saving.

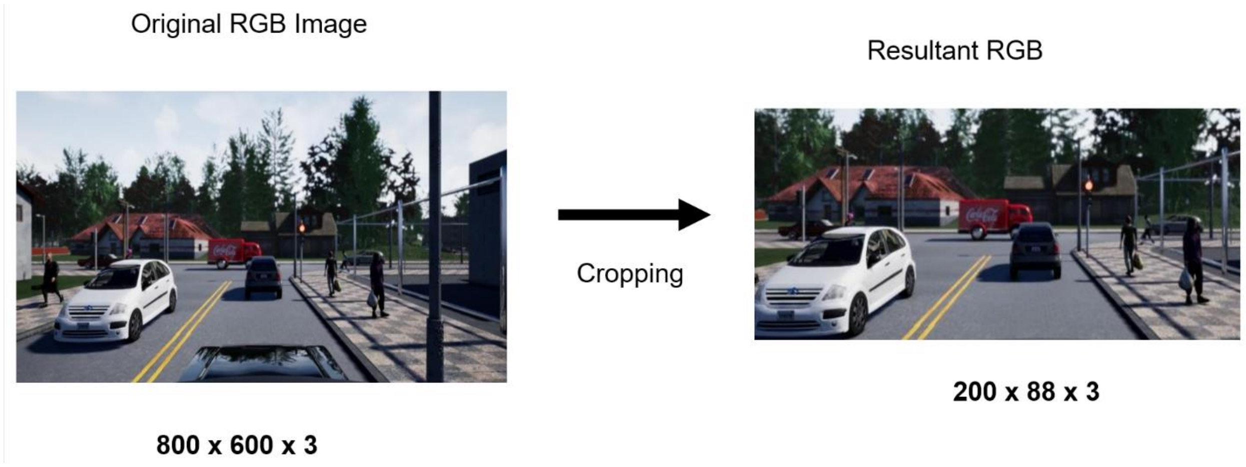


Figure 6-4 Cropping frame from 800x600 to 200x88

6.5 Pros and Cons of the data

6.5.1 Pros

- Relatively large amount of data which means many scenes for the network to be trained on.
- The measurement vector contains more than what is needed which opens the doors for future development.
- Almost totally clean.

6.5.2 Cons

- Doesn't take traffic lights into consideration.
- Almost completely ignores pedestrians.
- Which effectively limits the performance of the supervised learning algorithms.
- 10 fps produces many duplicated frames which must be taken care of in the preprocessing step.

6.6 Data processing prior to our training

In this section, we will discuss the processing done on the data before the training.

6.6.1 Filtering data with unaccepted speeds

Some speeds are negative which means that the car is moving backward. Thus, all frames with speeds less than -1 were filtered out and those with speeds less than zero and greater than -1 were left and considered as noise. Noise is something we usually add to increase robustness to over fitting.

6.6.2 Creating Scenarios as a sequence of frames

The time dependent networks should be fed by a sequence of frames instead of single frame to take the time feature into consideration so in the preprocessing stage we stack the current frame with the frame came before it as its past associated with the current frame (last frame in the sequence) target labels

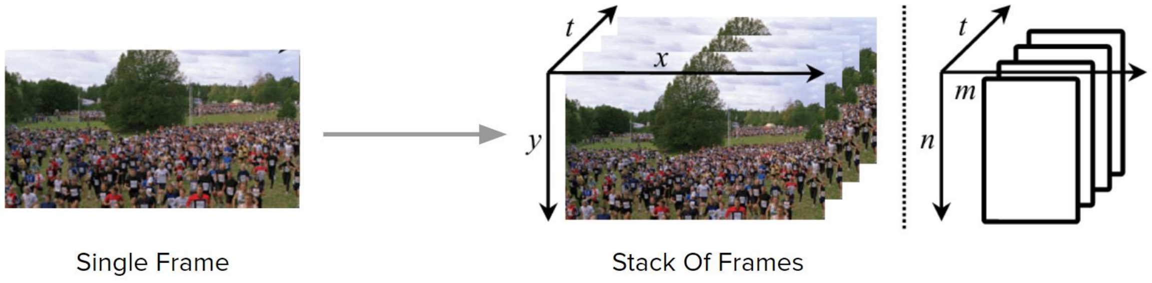


Figure 6-5 From single frame to stack of frames

Creating scenarios statically needs a huge amount of storage, so we use the shit in scenarios concept to reduce the number of samples which is illustrated in figure 6-6.

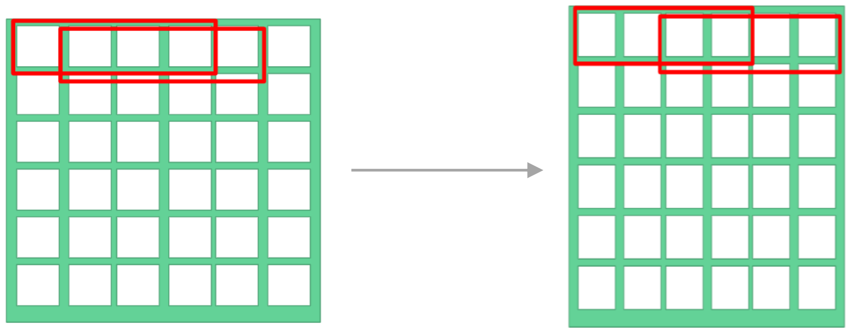


Figure 6-6 Creating scenarios

6.6.3 Up sample for the right and left branches

As mentioned before, each data point is associated with a high-level command (Turn right, turn left, follow the lane, continue straight).

It might be intuitive that each of these commands doesn't represent 25% of the data which is a problematic situation especially with the branched architecture. Biasing the model towards one of the commands is something we need to avoid.

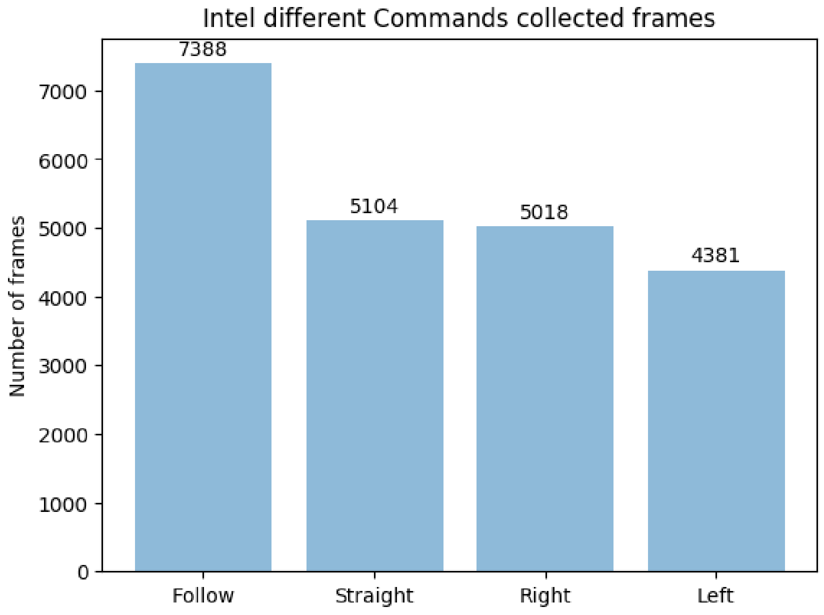


Figure 6-7 Data distribution over the four classes

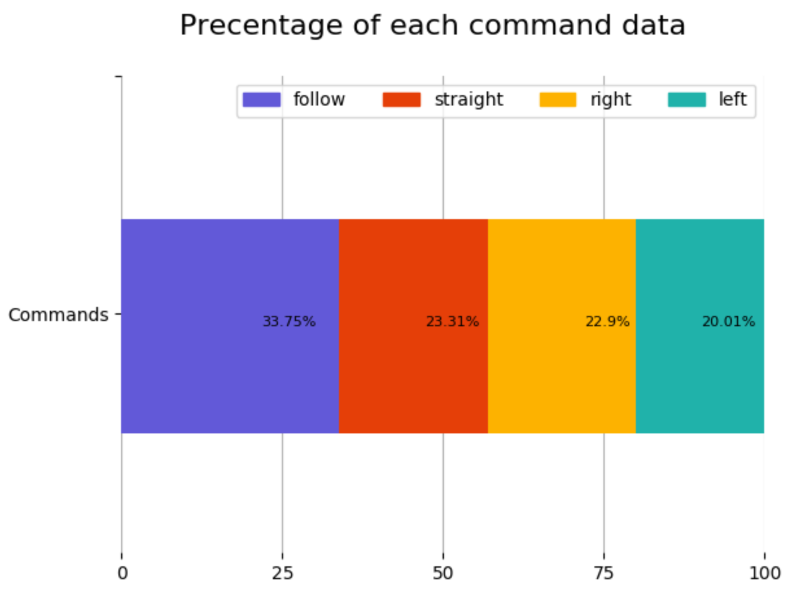


Figure 6-8 Data percentage for each class

By analyzing the data, we found that frames associated with the right and left high level command have the interesting property that high-Level command usually comes early before the curve which should create correlation between follow and the other high-level commands.

So, we create a dictionary for each high-level command containing the start and end for each High-level command sequence and use the last third form the right and the left sequences to balance the dataset. This will help us also to improve the performance of the right and left branches in testing.

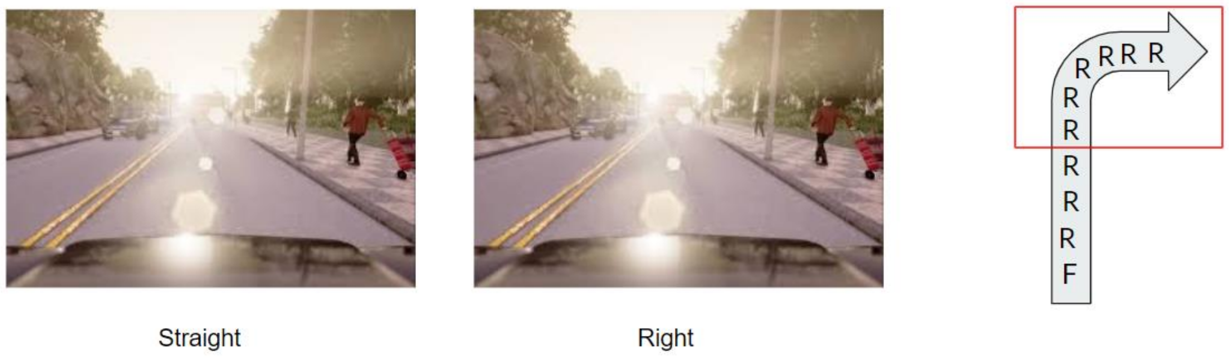


Figure 6-9 Right and straight looks like follow and up sampling

6.6.4 Data augmentation

Data augmentation is a strategy that enables practitioners to significantly increase the diversity of data available for training models, without actually collecting new data. There are two data augmentation techniques:

- Position Augmentation, i.e. Crop, resize, horizontal flip and vertical flip.
- Color Augmentation, i.e. Brightness, contrast, saturation and Grayscale transformation.

Not all the augmentation techniques work for a certain problem, for instance and in our specific case, horizontal flip and vertical flip will be a complete mess.

Below is the original scenario taken from the simulator alongside with some examples to augmented scenarios.

1. Original scenario with length of 4 frames

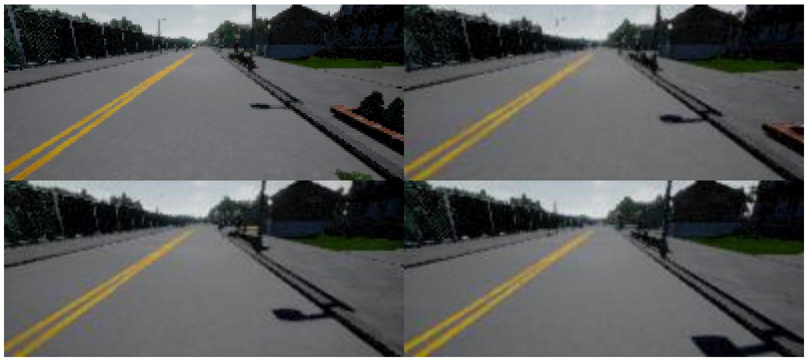


Figure 6-10 Original scenario

2. Augmented scenario with Change Color Temperature, Multiply and Add to Brightness and Rain

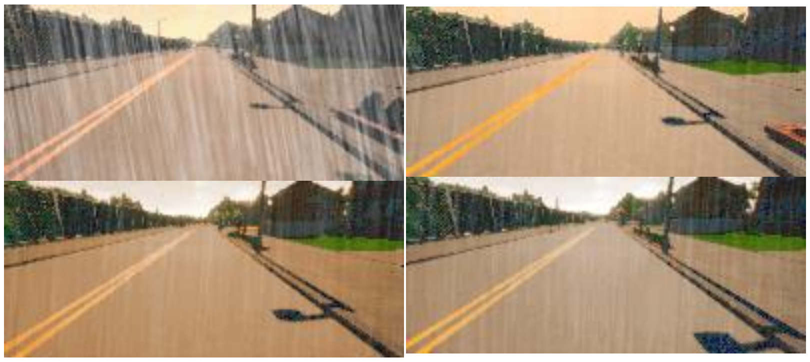


Figure 6-11 Augmented scenario 1

3. Augmented scenario with Change Color Temperature, Multiply and Add to Brightness and Gaussian Blur

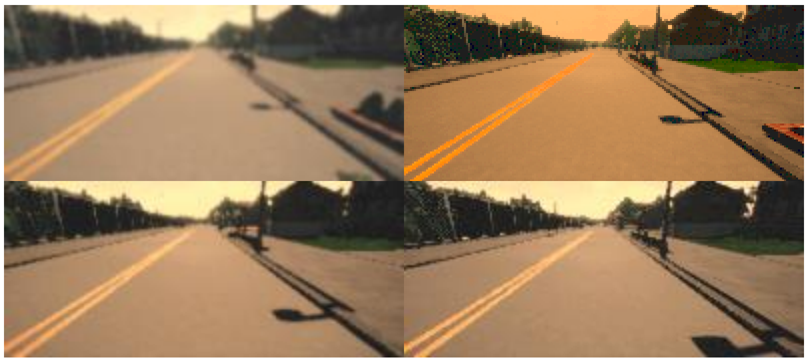


Figure 6-12 Augmented scenario 2

4. Augmented scenario with Change Color Temperature, Multiply and Add to Brightness and Additive Gaussian Noise

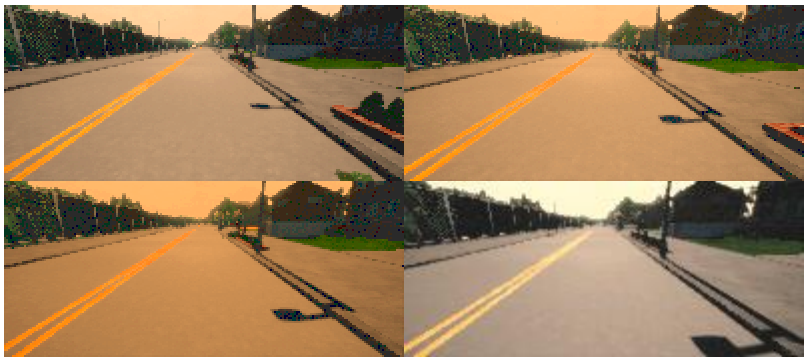


Figure 6-13 Augmented scenario 3

5. Augmented scenario with Change Color Temperature, Multiply and Add to Brightness and Coarse Salt and Pepper

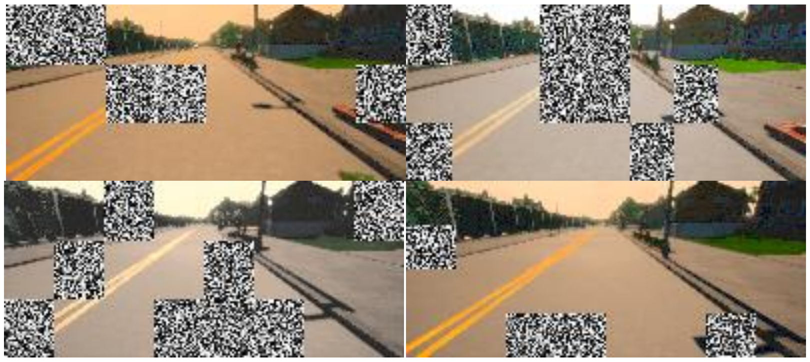


Figure 6-14 Augmented scenario 4

6. Augmented scenario with Change Color Temperature, Multiply and Add to Brightness and Coarse Dropout

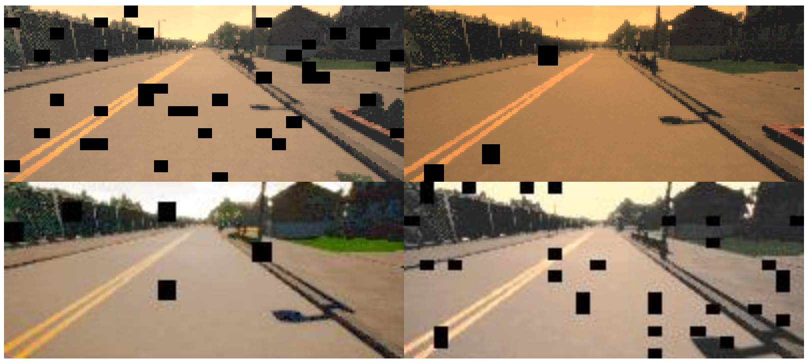


Figure 6-15 Augmented scenario 5

7. Augmented scenario with Contrast Normalization

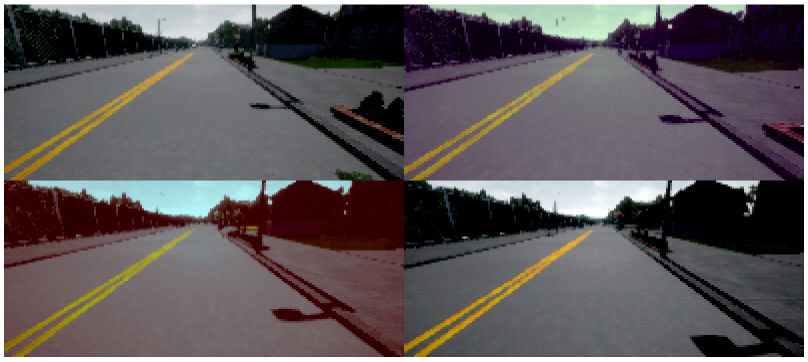


Figure 6-16 Augmented scenario 6

6.6.7 Processing sequence

The processing sequence of the data was as follows:

- First, we have 3290 files of data, each containing 200 frames.
- Secondly, we filter these data with unaccepted speeds, then we create scenarios while classifying them into four files, each containing data related to a certain class.
- Up sampling is used to balance and improve the dataset
- Data is written as 32 data points per file. This is because, experimentally, it's the highest batch size applicable for the available GPUs.
- The last thing to do is to augment the data, this is done through reading the data, shuffling it, then randomly augment a portion of it, and add the augmented array to the original array and finally shuffling again or augment on the fly while training.

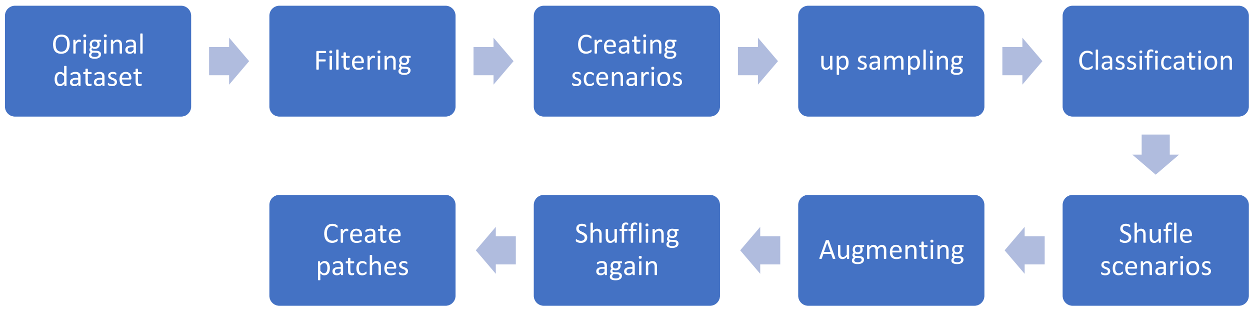


Figure 6-17 Data processing sequence

Chapter 7 Time Dependent Neural Networks

In this chapter we address our first approach in designing deep learning networks, we describe several approaches in the time dependent neural networks.

7.1 Supervised Camera Based Architectures

The first approach is to use the camera sensor. Camera image is the most scenery enrich sensor that acquires a high-quality representation of the front scene.

Overall, the camera input passes through:

a) A preprocessing stage: where images get filtered and augmented
b) Feature Extraction Network: CNN (convolutional neural network) which locates significant features of the input image
c) Merging module: Weights each feature
d) Prediction Network: predicts the optimal action regarded the input observation

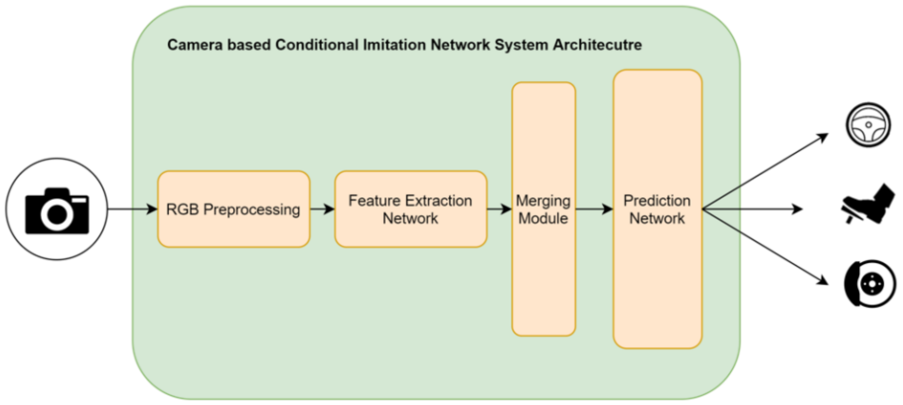

Camera based Conditional Imitation Network System Architecutre
RGB Preprocessing
Feature Extraction Network
Merging Module
Prediction Network

Figure 7-1 Imitation learning Camera based overall system architecture

Our contribution in the **feature extraction module** is that we wanted our controller to mimic real-life driving policy. For instance, during car driving we don't only rely on the current observation O_t, but also previous scenes or observations $O = \{\dots, O_{t-2}, O_{t-1}, O_t\}$. In fact, the RGB Preprocessing module in figure 7-2 converts between the old single image representation to the new stack of frame representation as shown in figure 7-2.

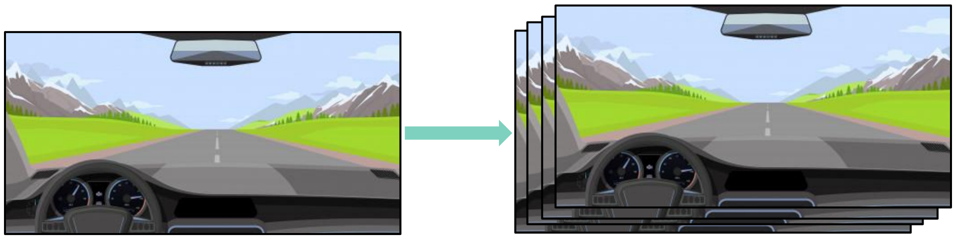


Figure 7-2 Multi-frames generation

In order to match the new data representation, we used three two main ideas in the feature extraction part

1. 3D Convolution (convolution across spatial and time information)
2. Reusing 2D Convolution across different time steps to feed an LSTM

Our enhancement in the **Merging module** regards how we'll be combining the outputs from the feature extraction module. Choices are as follows:

1. Ordinary FC in case of 3D Convolution
2. LSTM and LSTM with Attention in case of 2D Convolution

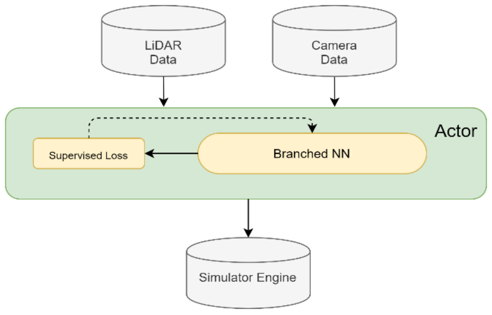
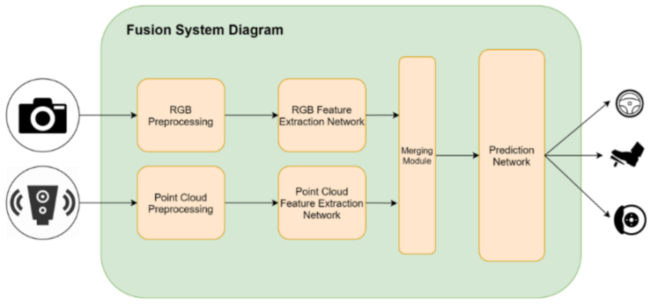


Figure 7-3 Fusion Imitation learning software architecture


Fusion System Diagram
RGB Preprocessing
RGB Feature Extraction Network
Point Cloud Preprocessing
Point Cloud Feature Extraction Network
Merging Module
Prediction Network
Figure 7-4 Sensor fusion overall system architecture

7.2 3D-Convloution SuperCam Architecture

Observed image is at 200x88 pixel resolution with an input shape of scenario_lengthx200x88x3. Feature extraction network consists of 8 3D Convolution layers with a stride of 2. The number of filters increased from 32 to 256 at the final layer. The output feature map is passed through 2 FC layers each with 512 neurons and then concatenated with the speed embedding. Finally, the concatenated features are passed through a FC layer with 512 neurons and then to all the four branches. Each branch consists of 2 FC layers of 256 neurons then 3 neurons output.

Batch normalization is added after each convolution and fully connected layers. RELU is used after all layers.

The current is speed is predicted using images feature map only to avoid the car from stopping during testing.

We used an MSE loss function to train our branched network along with Adam optimizer and an initial learning rate of 0.001, where a 120 batches size was used.

It was better to use online augmentation due to memory limits and also to improve the generalization capability. We decided to fix sum augmentation techniques while changing others during each training step. Coarse drop out, Gaussian noise and salt and pepper were used as fixed augmentation to force the network to focus on the road features, while a changing in colors and brightness where used dynamically changing to force the network to be immune to any weather conditions.

$$min_{\theta} \sum l(F(o_i, \theta), a_i)$$

Where $F(o_i, \theta)$ the network is output and a_i are actions for given observation o_i

7.3 LSTM SuperCam Architecture

The only change in this architecture is that we used 2D convolution and we reuse the feature extraction part according to the number of the input multiple frames.

We used LSTM mechanism. Observed image is at 200x88 pixel resolution with an input shape of 200x88x3. Feature extraction network consists of 8 2D-Convolution layers with a stride of 2. We reused filters to match the scenario length e.g. each 2D Filter is used scenario length times on each input frame and the number of filters increased from 32 to 256 at the final layer. The output feature map is passed through 2 FC layers each with 512 neurons, before we concatenate the output with the speed embeddings, we first apply LSTM between all outputs of the Reused 2D-Conv layers and obtain the output from the last time sample frame (As shown in figure 7-5). Finally, the LSTM output is then concatenated with the speed embedding and passed through a FC layer with 512 neurons and then to all the four branches. Each branch consists of 2 FC layers of 256 neurons then 3 neurons output.

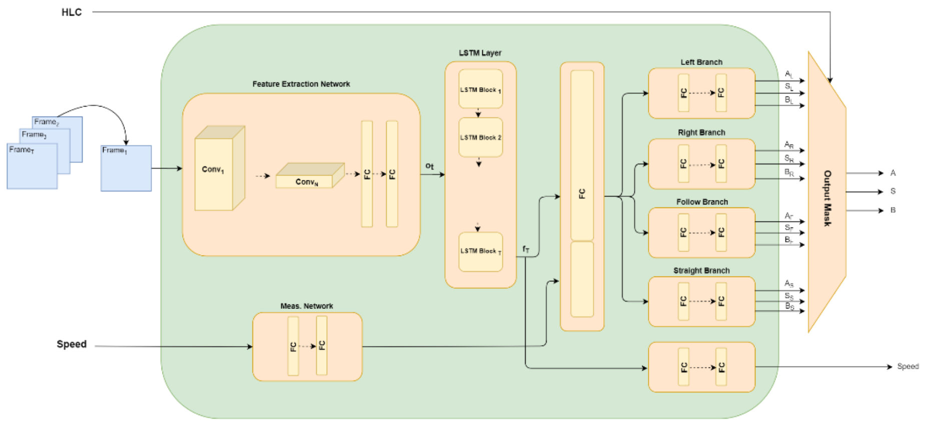


Figure 7-5: LSTM SuperCam Architecture

7.4 LSTM SuperCam and LSTM + Attention SuperCam Architectures

The same blocks from previous normal LSTM, we used 2D convolution and we reuse the feature extraction part according to the number of the input multiple frames. The difference between the LSTM SuperCam architecture and the LSTM + Attention SuperCam architecture is that the latter uses attention from (Bahdanau, Cho and Bengio 2015) while the former uses the output from the last time step LSTM cell. Finally, the concatenated features are passed through a FC layer with 512 neurons and then to all the four branches. Each branch consists of 2 FC layers of 256 neurons then 3 neurons output.

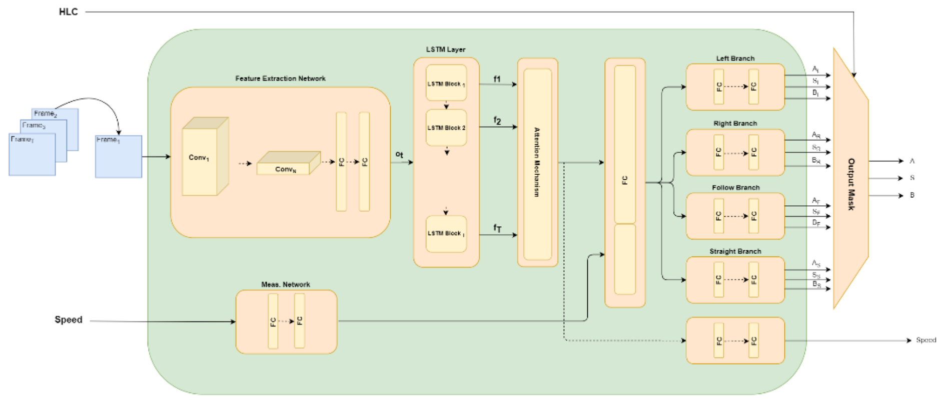


Figure 7-6 LSTM + Attention SuperCam Architecture

Chapter 8 Fusion Networks

This chapter address the fusion part in our project, we'll mainly talk about three parts: New collected data, SuperFusion, SuperVoxelNet

7.1 Dataset

We've collected new data with the same configurations of the original datasets as it doesn't have point clouds. The collected dataset has the following characteristics before any processing:

- It was collected with the simulator running on 10 fps.
- It consists of 89 episodes
- It contains approximately 300, 000 frames (Training points).
- Each frame is represented as an RGB image (3 channels).
- Each channel has a resolution of 200 x 88.
- Each frame has a point cloud.
- Each point cloud has a maximum of 10000 points.

7.2 SuperFusion Net

7.2.1 Point Cloud representations

The network has two inputs from the point cloud

1. Range Image:

The LiDAR produces a cylindrical range image as it sweeps over the environment with a set of lasers. The horizontal resolution of the image is determined by the rotation speed and laser pulse rate, and the vertical resolution is determined by the number of lasers. The LiDAR contains a set of 64 lasers with Elevation Field of view = (-10, 30). For each point in the sweep, the sensor provides a range r, reflectance e, azimuth θ, and laser id m, which corresponds to a known elevation angle. Using the range, azimuth, and elevation, we can compute the corresponding 3D point (x, y, z)in the sensor frame. We build an input image by directly mapping the laser id m to rows and discretizing azimuth θ into columns. If more than one point occupies the same cell in the image, we keep the closest point.

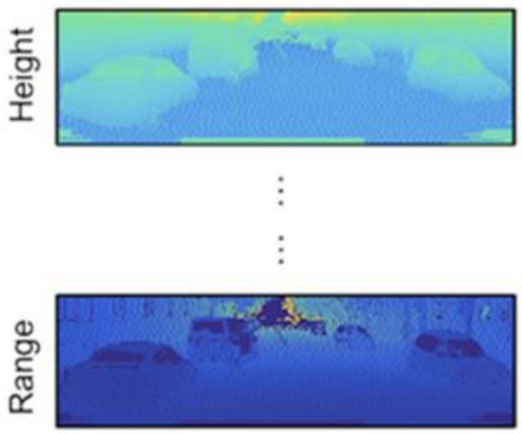


Figure 8-1 Range image channels in the front camera field of view

For each cell coordinate in the image, we collect a set of input channels from its corresponding point: range r, height z, azimuth angle θ. The result is a three-channel image that forms the input to our network

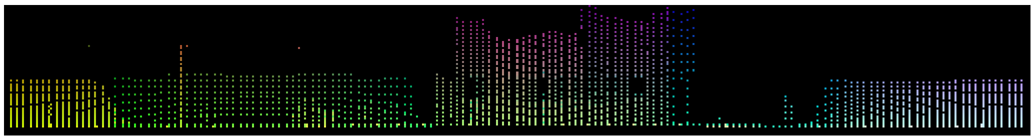


Figure 8-2 360-degree wide view Range image

2. Bird's Eye View

A view from a high angle as if seen by a bird in flight that we got by projecting the points from the 3D-point cloud into 2D-grayscale image representing the x-y plane or the road geometry as each pixel contains the height of any obstacle there.

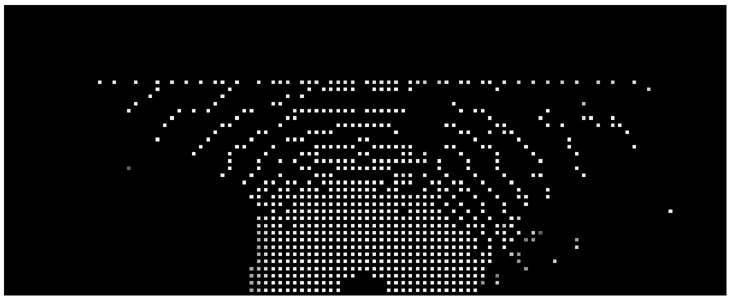


Figure 8-3 180-degree bird's eye view

7.2.2 Point Cloud Networks

We maintained the settings for the classification part and for the images feature extraction part, but we added a new point cloud feature extraction network. Feature extraction network for point cloud is based on the VoxelNet in (Zhou & Tuzel, 2017). The output feature map from for both Camera and LiDAR are concatenated with the speed embedding. Finally, the concatenated features are passed through a FC layer with 512 neurons and then to all the four branches.

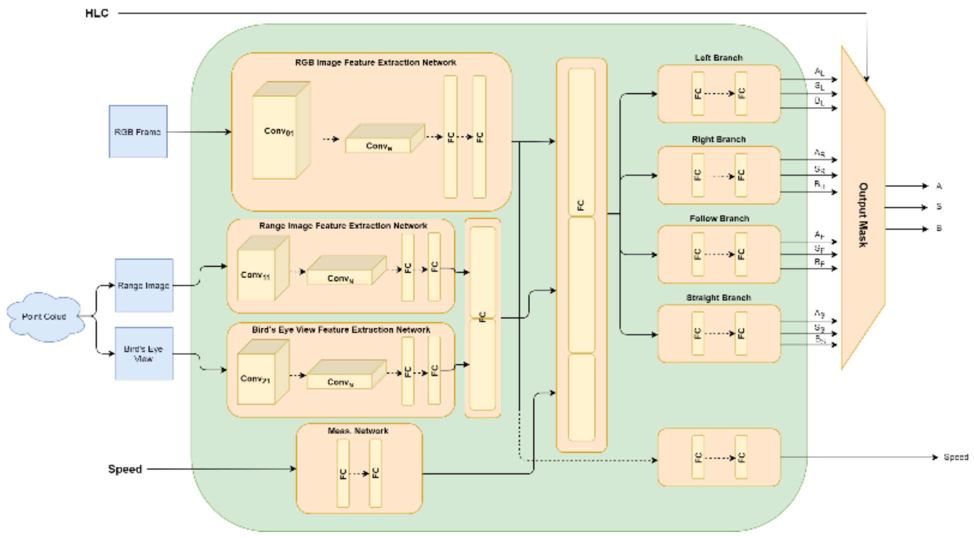


Figure 8-4 RGB, Range Image and Bird's Eye View SuperFusion Architecture

7.3 SuperVoxel Net

In this approach we need to divide the space into small voxels each voxel contains some points. After choosing the voxel size and the max number of points in each voxel, we can create sequences of voxel, then we capture dependencies between points within the same voxel (Using Conv Operation). The output of this is a filtered, correlated and cleaner voxels each voxel contains features, 3D-Conv is used to capture correlation between voxels.

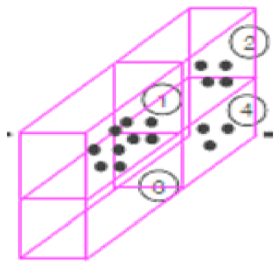


Figure 8-5 Voxels in space

7.3.1 VoxelNet Intuition

A work [] proposed VoxelNet that "a generic 3D detection network that unifies feature extraction and bounding box prediction into a single stage, end-to-end trainable deep network". As mentioned in the paper, the three networks are:

1. Feature Learning Network
2. Convolutional Middle Layers
3. Region Proposal Network

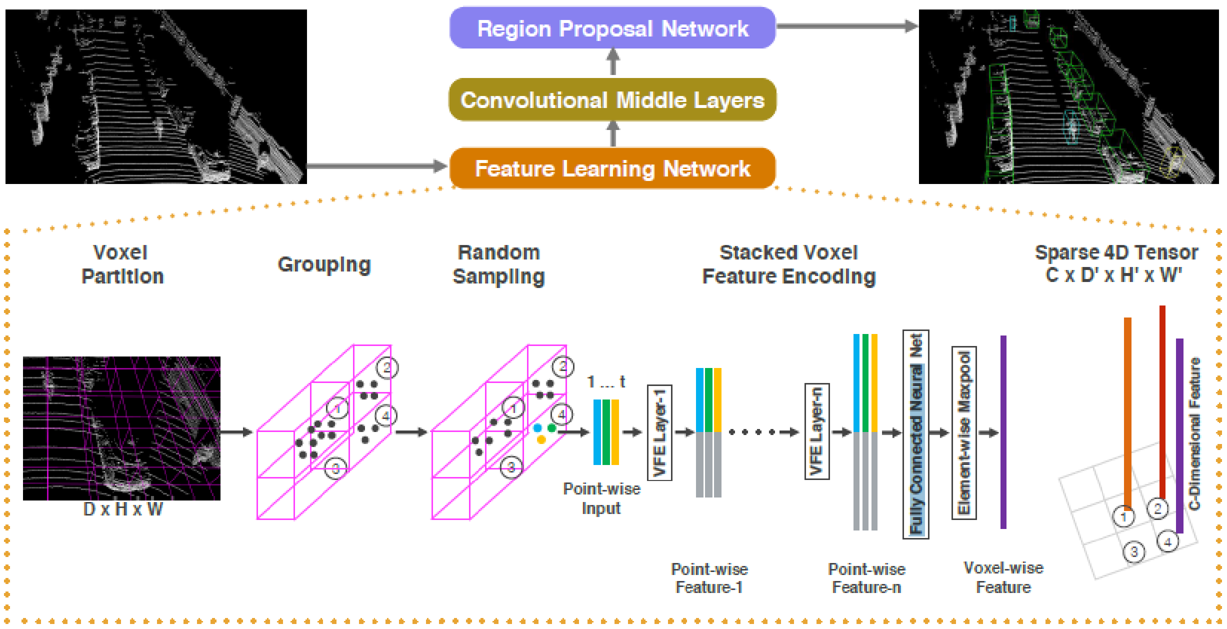


Figure 8-6 VoxelNet Overview

And the steps are follows:

1. Preprocessing:
   a. Divide the input point cloud into voxels (Boxes in space) with a picked width, height, depth
   b. Some voxels might contain more or less points than other voxels, so a maximum number of points T is chosen, and if the number of points is larger than T then random T point are picked. If the number of points is smaller than T then we zero-pad the rest of points
2. Feature Learning Network: All coming operations are per voxel:
   a. VFE layer
   b. Element Max Pooling
   c. Concatenation
   d. Reallocate each voxel with a tensor representing the output feature vector
3. Convolutional Middle Layers
   a. 3D Convolution layers
4. Region Proposal Network

7.3.2 SuperVoxel

In general, our task wasn't region proposal so we decided to discard the final RBN module mentioned in the paper. Our architecture is divided into three modules:

1. RGB Module: Done with Single frame input or as per our enhances of Time dependencies
2. VoxelNet module: Same Feature Learning Network and Convolutional Middle mentioned in []
3. Speed moduel

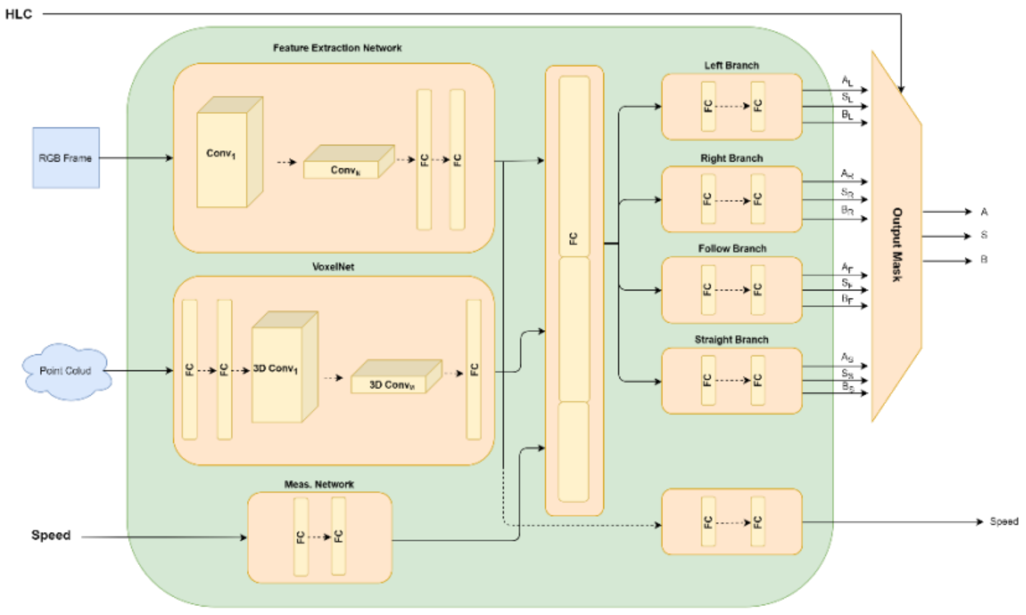


Figure 8-7 RGB and Point CLoud SuperFusion

7.3 Conclusion

In this chapter we discussed the details of our data and the preprocessing sequence we used. Now that we are done with the used algorithm, the implementation, the data, and before we dig into the results let's talk a little bit about the simulator and the hardware resources we used, which are the topics of the following chapters respectively.

Chapter 9 Hardware

In order to start deploying our product we wanted to create a POC (Proof of Concept) prototype on a scale 1/5 of a real truck. This section explains two parts: Physical System Specs and Fine Tuning.

8.1 Physical System Specs

The setup of the physical system is shown in Figure 9-1,9-2, we were supposed to follow this work [] by Intel. The main components of the hardware systems are as follows:

The system overall is: NVidia TX23 takes input image from the front camera, process the image using out deployed model, send signal to the flight controller to control the speed as well as the steering of the car.

Table 9-1: Hardware Main Components

Component	Description
Traxxas Maxx	The body of the vehicle
Nvidia TX2	Embedded processing GPU unit that will process incoming images (observations from the environment)
Camera	Front-camera to get the scene information
Holybro-Pixhawk	Flight Controller contains ARM processor, on-board sensors like GPS which is used to obtain the route

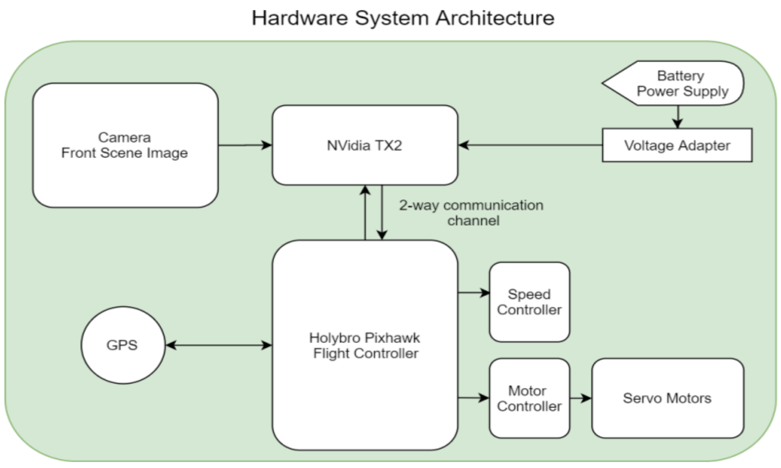


Figure 9-1 General Overview of the system

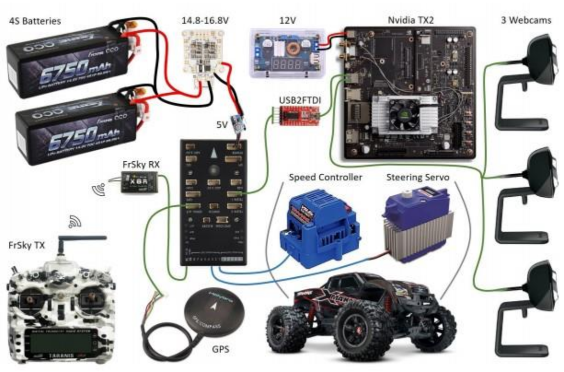


Figure 9-2 Hardware Component

8.2 Fine Tuning

We've tried as much as possible to enrich our model with augmented images that simulates the reality, e.g. Coarse Dropout was intended to provide a mimic the real shadow on the ground. However, since safety is our priority, we wanted to increase our model accuracy in reality. In order to reach that:

- We collected 700 frames from the real environment where our hardware will be tested, samples of the frames are shown in figure 9-3.
- The 700 frames were manually labeled with high level command.
- We fine-tuned the model and re-trained a few more epochs with the new frames.

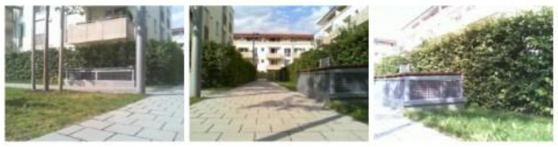


Figure 9-3 Collected frames

Chapter 10 Imitation and Fusion Results

9.1. Experimental setup

Carla simulator makes us able to run the evaluation mechanism in an episodic setup. In each episode, the agent is initialized at a new given point and asked to drive to a given destination point, given high-level turn commands from a topological planner. An episode is considered successful if the agent reaches the goal within a fixed time interval. CARLA has 2 towns. Town 1 is used for training, while town 2 is used exclusively for testing. There are 4 kinds of weather for training and 2 for testing.

For evaluation, we followed CoRL 2017 experiment [9.1], which consists of 4 tasks (straight, single turn, navigation without dynamics, and navigation with dynamics) each has 25 pairs of start and goal locations as in fig [9.2]. So, we sum up with 600 test cases (6 weathers * (4 tasks * 25 test cases per each task)) in each town. More details about the driving benchmark structure are shown in fig [9.3].

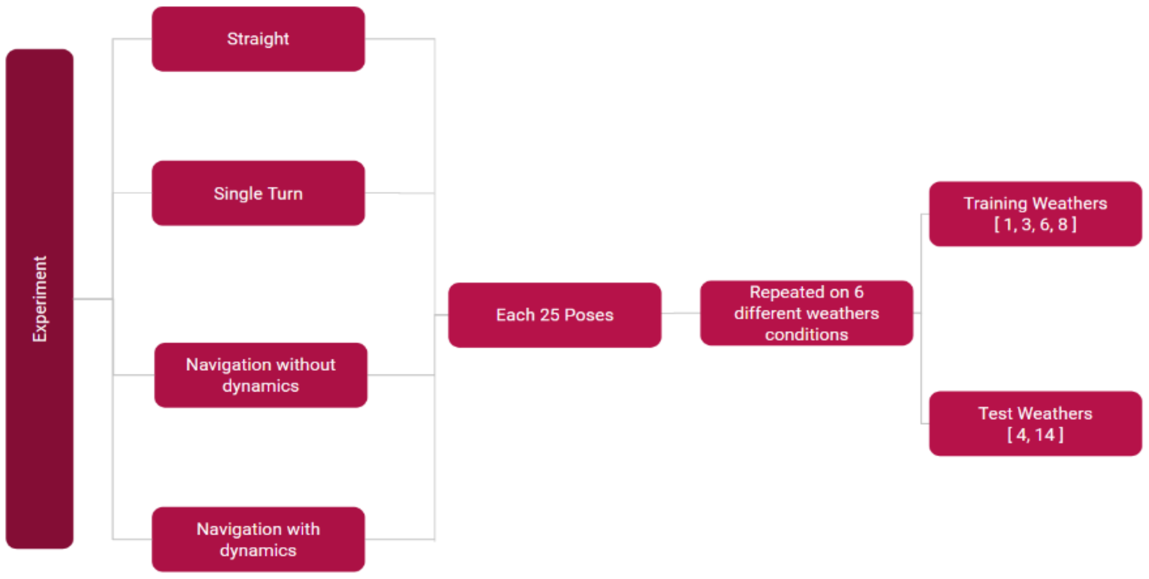

Experiment
Straight
Single Turn
Navigation without dynamics
Navigation with dynamics
Each 25 Poses
Repeated on 6 different weathers conditions
Training Weathers [1, 3, 6, 8]
Test Weathers [4, 14]

Figure 10-1 CoRL 2017 Experiment

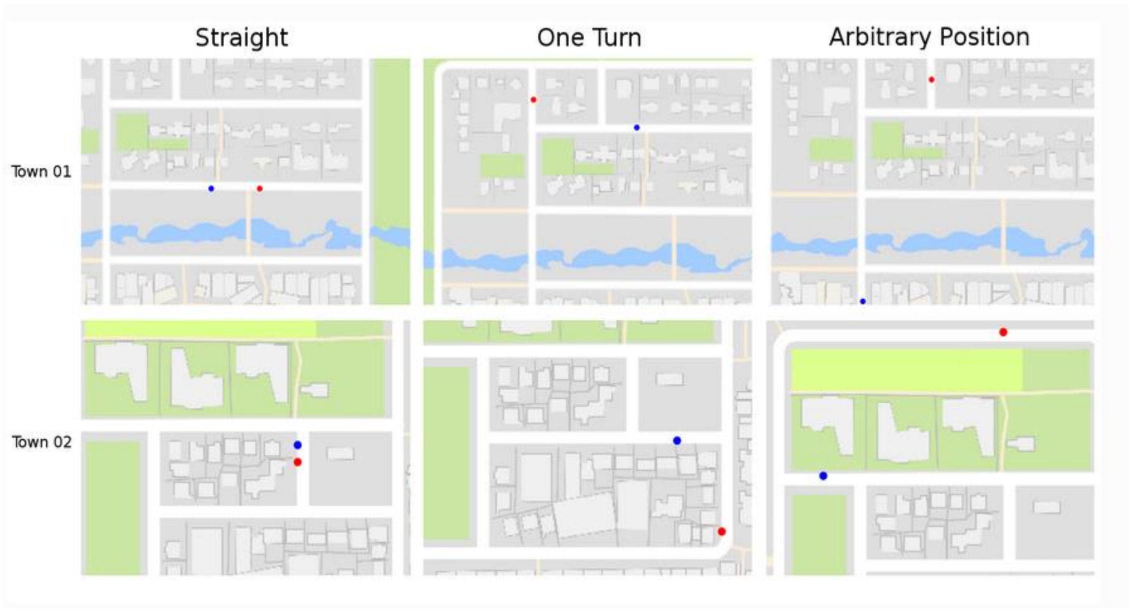


Figure 10-2 tasks Explanation in Carla towns

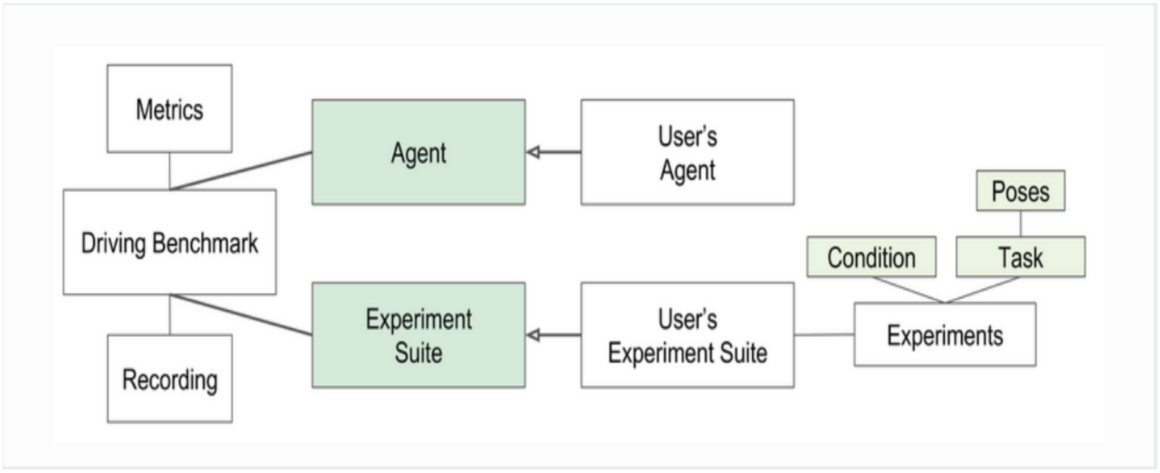


Figure 10-3 Benchmark Architecture

9.2. Trials and results

Trial iterations:

- Design Network architecture
- Hyper parameter tuning
    - Change dropout ratio
    - Add Augmentation in the fly instead of static and change augmentation ratio
    - Adding batch normalization
    - Change loss weights

    - Add Speed branches
    - Up sampling
- Deeside based on Tensor board whether the model efficiency is good enough to start benchmarking
- Benchmark and compare to previous iterations

Table 10-1: Results

Network	Task	T1W1	T1W2	T2W1	T2W2
Replication of intel network Intel conditional network	Straight	95	89	79	80
	Single Turn	89	90	59	48
	Navigation without dynamics	86	84	40	44
	Navigation with dynamics	83	82	38	42
3D convolution SuperCam	Straight	100	98	96	90
	Single Turn	97	96	74	70
	Navigation without dynamics	88	88	56	60
	Navigation with dynamics	80	84	48	54
LSTM SuperCam	Straight	98	96	95	90
	Single Turn	92	90	75	72
	Navigation without dynamics	84	84	60	62
	Navigation with dynamics	79	80	46	55
LSTM + Attention SuperCam	Straight	100	99	95	92
	Single Turn	94	92	80	74
	Navigation without dynamics	88	89	64	62
	Navigation with dynamics	80	82	48	57
Range Image SuperLidar	Straight	82	80	82	80
	Single Turn	54	56	54	56
	Navigation without dynamics	41	40	41	40
	Navigation with dynamics	36	37	36	37
Bird's eye view SuperLidar	Straight	90	88	86	86
	Single Turn	89	86	74	72
	Navigation without dynamics	78	70	70	56
	Navigation with dynamics	54	50	50	50
SuperLidar (Range + Bird's eye view)	Straight	96	96	89	88
	Single Turn	94	94	75	75
	Navigation without dynamics	82	82	64	64
	Navigation with dynamics	78	78	60	60
RGB, Ragne image, and Bird's eye view Super Fusion	Straight	100	100	98	96
	Single Turn	97	98	88	89
	Navigation without dynamics	93	93	76	75
	Navigation with dynamics	89	88	75	74
SuperVoxel	Straight	100	100	98	98
	Single Turn	98	98	90	88
	Navigation without dynamics	94	92	77	75
	Navigation with dynamics	91	90	74	73

Chapter 11 Deep Deterministic Policy Gradient in Autonomous Driving

As described in chapter 3, DDPG is a policy gradient algorithm that uses a stochastic behavior policy for good exploration but estimates a deterministic target policy, which is much easier to learn. Policy gradient algorithms utilize a form of policy iteration: they evaluate the policy, and then follow the policy gradient to maximize performance. Since DDPG is off-policy and uses a deterministic target policy, this allows for the use of the Deterministic Policy Gradient theorem. DDPG is an actor-critic algorithm as well; it primarily uses two neural networks, one for the actor and one for the critic. These networks compute action predictions for the current state and generate a temporal- difference (TD) error signal each time step. The input of the actor network is the current state, and the output is a single real value representing an action chosen from a continuous action space. The critic's output is simply the estimated Q-value of the current state and of the action given by the actor. The deterministic policy gradient theorem provides the update rule for the weights of the actor network. The critic network is updated from the gradients obtained from the TD error signal. In general, training and evaluating the policy and/or value function with thousands of temporally-correlated simulated trajectories leads to the introduction of enormous amounts of variance in your approximation of the true Q-function (the critic). The TD error signal is excellent at compounding the variance introduced by bad predictions over time. We used a replay buffer to store the experiences of the agent during training, and then randomly sample experiences to use for learning in order to break up the temporal correlations within different training episodes. This technique is known as experience replay. DDPG uses this. Directly updating the actor and critic neural network weights with the gradients obtained from the TD error signal that was computed from both your replay buffer and the output of the actor and critic networks causes your learning algorithm to diverge (or to not learn at all). It was recently discovered that using a set of target networks to generate the targets for your TD error computation regularizes your learning algorithm and increases stability. Figure 11-1 describes the DDPG in architecture in details.

Algorithms 1 is the DDPG algorithm.

Algorithm 1 DDPG algorithm

```
Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights $\theta^Q$ and $\theta^\mu$.
Initialize target network $Q'$ and $\mu'$ with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer $R$
for episode = 1, M do
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state $s_1$
  for t = 1, T do
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action $a_t$ and observe reward $r_t$ and observe new state $s_{t+1}$
    Store transition $(s_t, a_t, r_t, s_{t+1})$ in $R$
    Sample a random minibatch of $N$ transitions $(s_i, a_i, r_i, s_{i+1})$ from $R$
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:
        $$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$
    Update the target networks:
        $$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
        $$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$
  end for
end for
```

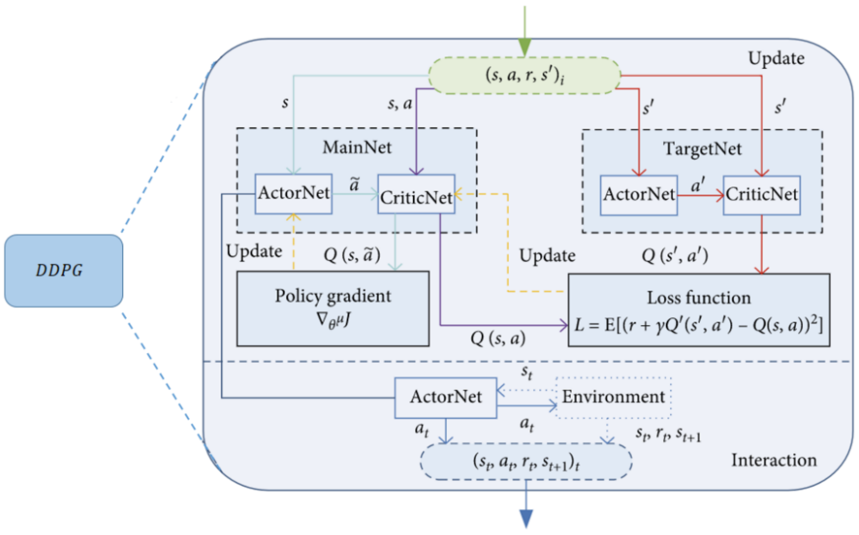


Figure 11-1 DDPG architecture

11.1 Actor

We used the same conditional network described in (Codevilla, Muller, Lopez, Koltun, & Dosovitskiy, 2017) as our actor. Figure 11.2 better describes the actor network in more details.

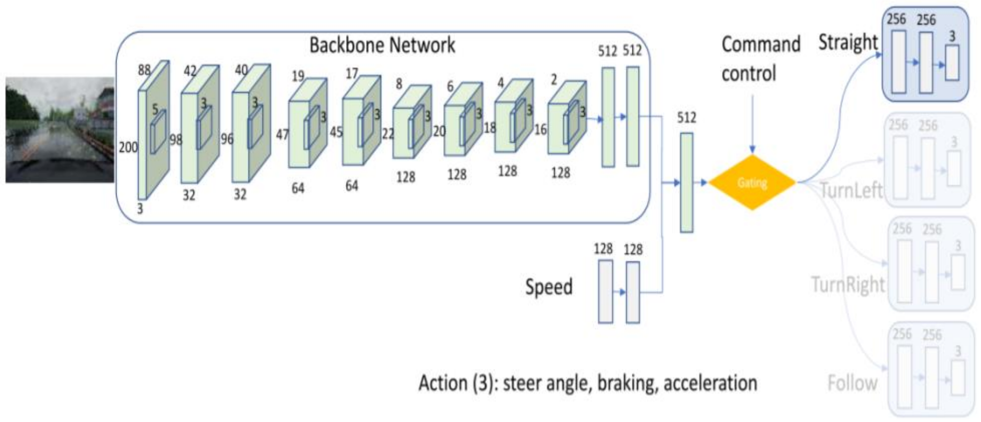


Figure 11-2 Actor Network detailed architecture

11.2 Critic

We used the same conditional network described in (Codevilla, Muller, Lopez, Koltun, & Dosovitskiy, 2017) as our critic that output the state value function for each branch. Figure 11.3 better describes the critic network in more details.

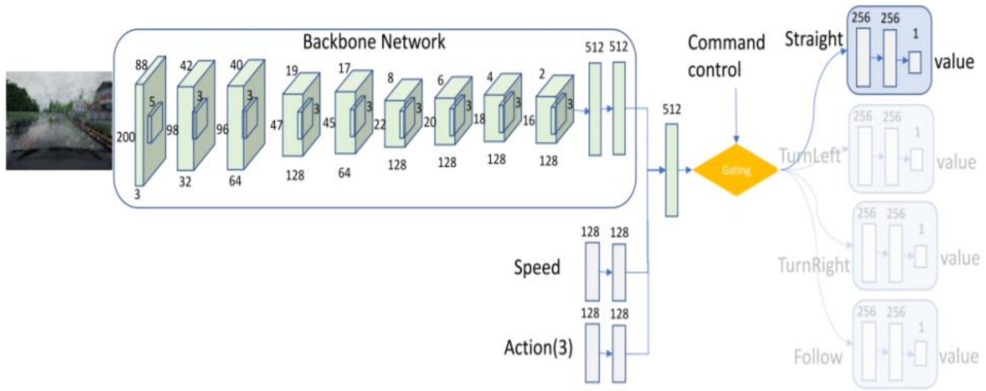


Figure 11-3 Critic architecture

11.3 Reward function

Reinforcement learning relies heavily on immediate rewards which needs to very representative to the task that needs to be solved. High dimensional problems might require complex reward shaping techniques to allow the agent to learn efficiently. The reward function that we used is described by

$$reward = max(current\ Speed - 1, 30) - 5 * lane\ intersection - 5 * offroad\ intersection - 100\ any\ collision - abs(steer) * 10$$

11.3.1 Speed reward

The car needs to move in most cases so there must be some positive reward that is proportional to its speed, but this is up to a limit which is 30 Km/h. Moreover, to prevent the car from stopping, a –ve reward is set along with the other rewards.

11.3.2 Lane and off-road intersection reward

The car needs to drive along its way without crossing any lanes or curbs, so we set a proportional –ve reward to accommodate for intersecting in lanes while driving which is given by

$$-5 * lane\ intersection - 5 * offroad\ intersection$$

11.3.3 Collision reward

To ensure safe driving, the car must learn to avoid hitting any static or dynamic object in the environment, so a very large –ve reward must be set if the car collide with anything which is given by

$$-100\ any\ collision$$

11.3.4 Steering reward

In most cases other than turning left of right, the car must drive in straight lines, so a need the car to learn not to output any steering while driving on its way, and this can be achieved by

$$-abs(steer) * 10$$

11.4 Exploration policy

We used epsilon-greedy (Sutton R. S., 2017) as our exploration policy starting with $\epsilon = 1$ and ending with $\epsilon = 0.01$ following a decay rate of 0.99999. This exploration schedule is set to ensure that the car can try as much exploration as it could for a sufficient amount of time before ending with a probability of 0.01 to choose a random action.

11.5 Conclusion

The reinforcement learning was expected to enhance the performance and achieve generalization, but the DDPG is sample inefficient especially in high dimensional state problems. It needs both large time and trying different random seeds, and due to huge networks used in Actor and Critic, our Memory usage affected the interaction process, and it was unfeasible to debug and monitor for tuning hyper-parameters. In the next chapter we will try some new ideas.

Chapter 12 Using Nervana coach framework and Rainbow algorithm

12.1 Nervana coach framework

Coach is a python framework which models the interaction between an agent and an environment in a modular way. With Coach, it is possible to model an agent by combining various building blocks, and training the agent on multiple environments. The available environments allow testing the agent in different fields such as robotics, autonomous driving, games and more. It exposes a set of easy-to-use APIs for experimenting with new RL algorithms, and allows simple integration of new environments to solve. Coach collects statistics from the training process and supports advanced visualization techniques for debugging the agent being trained.

12.2 Rainbow algorithm

The deep reinforcement learning community has made several independent improvements to the DQN algorithm. However, it is unclear which of these extensions are complementary and can be fruitfully combined. Rainbow combines six extensions to the DQN **Invalid source specified.**.

12.3 Extensions to DQN

DQN has been an important milestone, but several limitations of this algorithm are now known, and many extensions have been proposed. Six extensions have been proposed, and each have addressed a limitation and improved overall performance figure 13.1 shows how Rainbow outperforms all pervious modification in DQN.

Double Q-learning. As mentioned in chapter 3, Conventional Q-learning is affected by an overestimation bias, and this can harm learning. Double Q-learning (Hasselt, H., Guez, & Silver., 2016), addresses this overestimation by decoupling, in the maximization performed for the bootstrap target, the selection of the action from its evaluation. It is possible to effectively combine this with DQN using the target given by equation (3.13)

This change was shown to reduce harmful overestimations that were present for DQN, thereby improving performance.

Prioritized replay. DQN samples uniformly from the replay buffer. Ideally, we want to sample more frequently those transitions from which there is much to learn. As a proxy for learning potential, prioritized experience replay **Invalid source specified.** samples transitions with probability relative to the last encountered absolute TD error.

Dueling networks. The dueling network is a neural network architecture designed for value based RL. It features two streams of computation, the value and advantage streams, sharing a convolutional encoder, and merged by a special aggregator **Invalid source specified.**. This corresponds to the following factorization of action values:

$$\boldsymbol{q_\theta(s,a) = v_\eta\left(f_\xi(s)\right) + a_\psi(c(s),a) - \frac{\sum_{a'} a_\psi\left(f_\xi(s),a'\right)}{N_{actions}}} \qquad \textbf{(11.1)}$$

Where $\xi, \eta \ and \ \psi$ are, respectively, the parameters of the shared encoder f_ξ, of the value stream v_η, and of the advantage stream a_ψ; and $\theta = \{\xi, \eta, \psi\}$

Multi-step learning. Q-learning accumulates a single reward and then uses the greedy action at the next step to bootstrap. Alternatively, forward-view multi-step targets can be used (Sutton R. S., 2017). We define the truncated n-step return from a given state S_t as

$$R_t^{(n)} = \sum_{k=0}^{n-1} \gamma_t^{(k)} R_{t+k+1} \qquad (11.2)$$

A multi-step variant of DQN is then defined by minimizing the alternative loss,

$$(R_t^{(n)} + \gamma_t^{(n)} \max_{a'} q_{\bar{\theta}}(S_{t+n}, a') - q_\theta(S_t, A_t))^2 \qquad (11.3)$$

Multi-step targets with suitably tuned n often lead to faster learning (Sutton R. S., 2017).

Distributional RL. We can learn to approximate the distribution of returns instead of the expected return. Recently, **Invalid source specified.** proposed to model such distributions with probability masses placed on a discrete support z, where z is a vector with $N_{atoms} \in N^+$ atoms defoned by $z^i = v_{\min} + (i-1)\frac{v_{\max} - v_{\mathrm{m}}}{N_{atoms}-1}$. The approximating distribution $d_t = (z, p_\theta(S_t, A_t))$.

The goal is to update θ such that this distribution closely matches the actual distribution of returns.

Noisy Nets. The limitations of exploring using -greedy policies are clear in games such as Montezuma's Revenge, where many actions must be executed to collect the first reward. Noisy Nets (Fortunato et al. 2017) propose a noisy linear layer that combines a deterministic and noisy stream

$$y = (b + Wx) + (b_{noisy} \odot \in^b + (W_{noisy} \odot \in^w)x) \qquad (11.4)$$

where $\in^w$ and $\in^b$ are random variables, and $\odot$ denotes the element-wise product. This transformation can then be used in place of the standard linear $y = b + Wx$. Over time, the network can learn to ignore the noisy stream, but will do so at different rates in different parts of the state space, allowing state-conditional exploration with a form of self-annealing

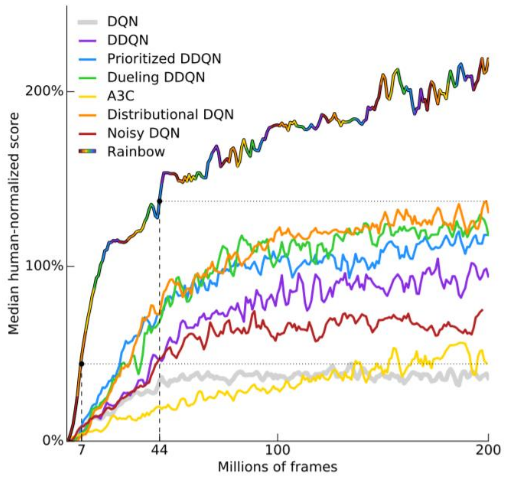


Figure 12-1 Median human-normalized performance across 57 Atari games

12.3.1 Integrated Agent training steps

1. Sample a batch of transitions from the replay buffer.
2. The Bellman update is projected to the set of atoms representing the Q values distribution
3. Network is trained with the cross-entropy loss between the resulting probability distribution and the target probability distribution. Only the target of the actions that were actually taken is updated.
4. Once in every few thousand steps, weights are copied from the online network to the target network.
5. After every training step, the priorities of the batch transitions are updated in the prioritized replay buffer using the KL divergence loss that is returned from the network.

12.4 Results

Rainbow algorithm is used from Nervana coach framework with CARLA 8.2 figure 12-2, the unconditional network described in (Codevilla, Muller, Lopez, Koltun, & Dosovitskiy, 2017) was used as our function approximator. The algorithm was learning very quickly to go straight but it had difficulties in learning to make turns so it needs a lot of reward shaping to make it work. Another problem when using Nervana is that the code was crashing after nearly 8 hours due to the lack of RAM. We used 16 GB is RAM that was not enough for the prioritized experience replay buffer, the solution to this problem is a new prioritized cashing method that will be discussed in the next chapter.

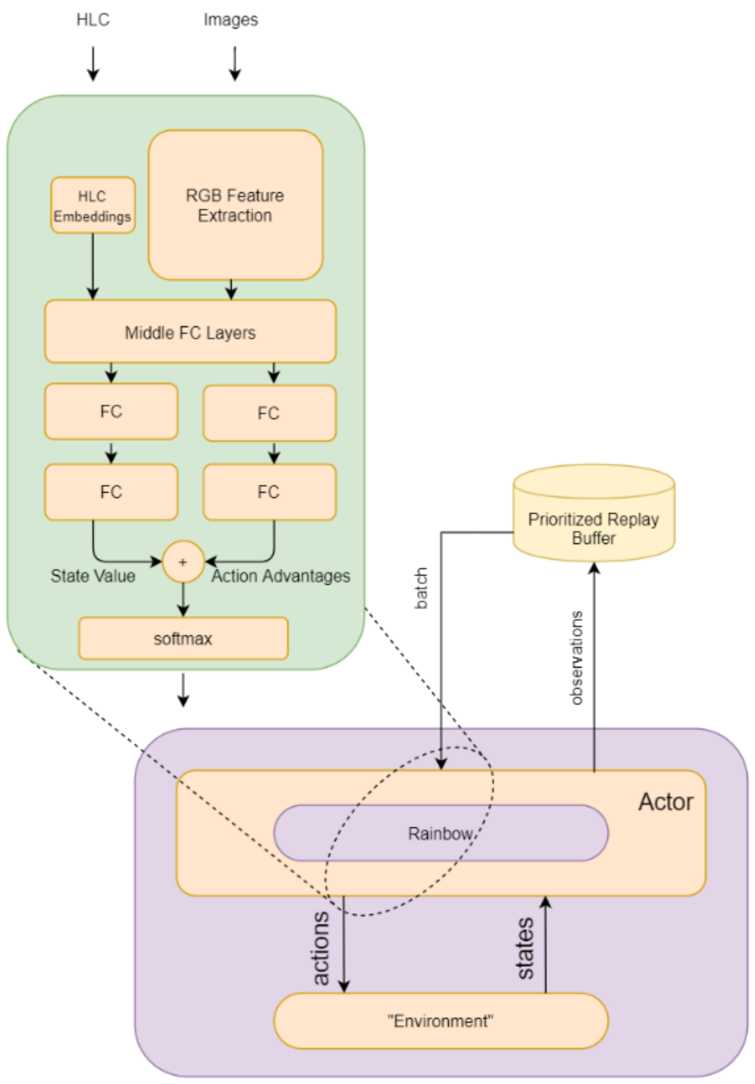


Figure 12-2 Rainbow architecture used with CARLA

Chapter 13 Deep Q-learning from demonstration

Deep reinforcement learning (RL) has achieved several high-profile successes in difficult decision-making problems. However, these algorithms typically require a huge amount of data before they reach reasonable performance. In fact, their performance during learning can be extremely poor. This may be acceptable for a simulator, but it severely limits the applicability of deep RL to many real-world tasks, where the agent must learn in the real environment. Deep Q-learning from Demonstrations (DQfD) is a setting where the agent may access data from previous control of the system. The algorithm leverages small sets of demonstration data to massively accelerate the learning process even from relatively small amounts of demonstration data and is able to automatically assess the necessary ratio of demonstration data while learning thanks to a prioritized replay mechanism. (DQfD) works by combining temporal difference updates with supervised classification of the demonstrator's actions.

13.1 Problem setup

We followed the work in **Invalid source specified.**. The optimal state-action value function $Q^*(s,a)$ provides maximal values in all states and is determined by solving the Bellman equation:

$$\boldsymbol{Q^*(s,a) = E[R(s,a) \ + \ \gamma \sum_{s'} P(s' \mid s,a) \max_{a'} Q^*(s',a') \,]} \qquad \textbf{(12.1)}$$

The optimal policy π is then $\pi(s) = \underset{a \in A}{argmax}\, Q^*(s,a)$. DQN approximates the value function $Q(s,a)$ with a deep neural network that outputs a set of action values $Q(s,\cdot\,;\theta)$ for a given state input s, where θ are the parameters of the network. There are two key components of DQN that make this work.

First, it uses a separate target network that is copied every τ steps from the regular network so that the target Q-values are more stable. Second, the agent adds all of its experiences to a replay buffer $\mathrm{D}^{\mathrm{replay}}$, which is then sampled uniformly to perform updates on the network.

The double Q-learning update **Invalid source specified.** uses the current network to calculate the argmax over next state values and the target network for the value of that action. The double DQN loss is

$$\boldsymbol{J_{DQ}(Q) = (R(s,a) + \gamma Q(s_{t+1}, a_{t+1}^{max}, \theta') - Q(s,a;\theta))^2} \qquad \textbf{(12.2)}$$

where $\theta'$are the parameters of the target network, and $a_{t+1}^{max} = \underset{a}{argmax}\, Q(s_{t+1}, a, \theta)$
Separating the value functions used for these two variables reduces the upward bias that is created with regular Q-learning updates. Prioritized experience replay **Invalid source specified.**. modifies the DQN agent to sample more important transitions from its replay buffer more frequently. The probability of sampling a particular transition i is proportional to its priority, $P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$ where the priority $p_i = \delta_i + \varepsilon$ and δ_i is the last TD error calculated for this transition and ε is a small positive constant to ensure all transitions are sampled with some probability.

$$\boldsymbol{J_E(Q) = \max_{a \in A}[Q(s,a) + l(a_E, a)] - Q(s, a_E)} \qquad \textbf{(12.3)}$$

where $\mathrm{a_E}$ is the action the expert demonstrator took in states and $l(a_E, a)$ is a margin function that is 0 when $a = a_E$ and positive otherwise. This loss forces the values of the other actions to be at least a margin lower than the value of the demonstrator's action. Adding this loss grounds, the values of the unseen actions to reasonable values, and makes the greedy policy induced by the value function imitate the demonstrator. If the algorithm pre-trained with only this supervised loss, there would be nothing constraining the values between consecutive states and the Q-network would not satisfy the Bellman equation, which is required to improve the policy on-line with TD learning.

The overall loss used to update the network is a combination of two losses:

$$\boldsymbol{J(Q) = J_{DQ}(Q) + \lambda J_E(Q)} \qquad \textbf{(12.4)}$$

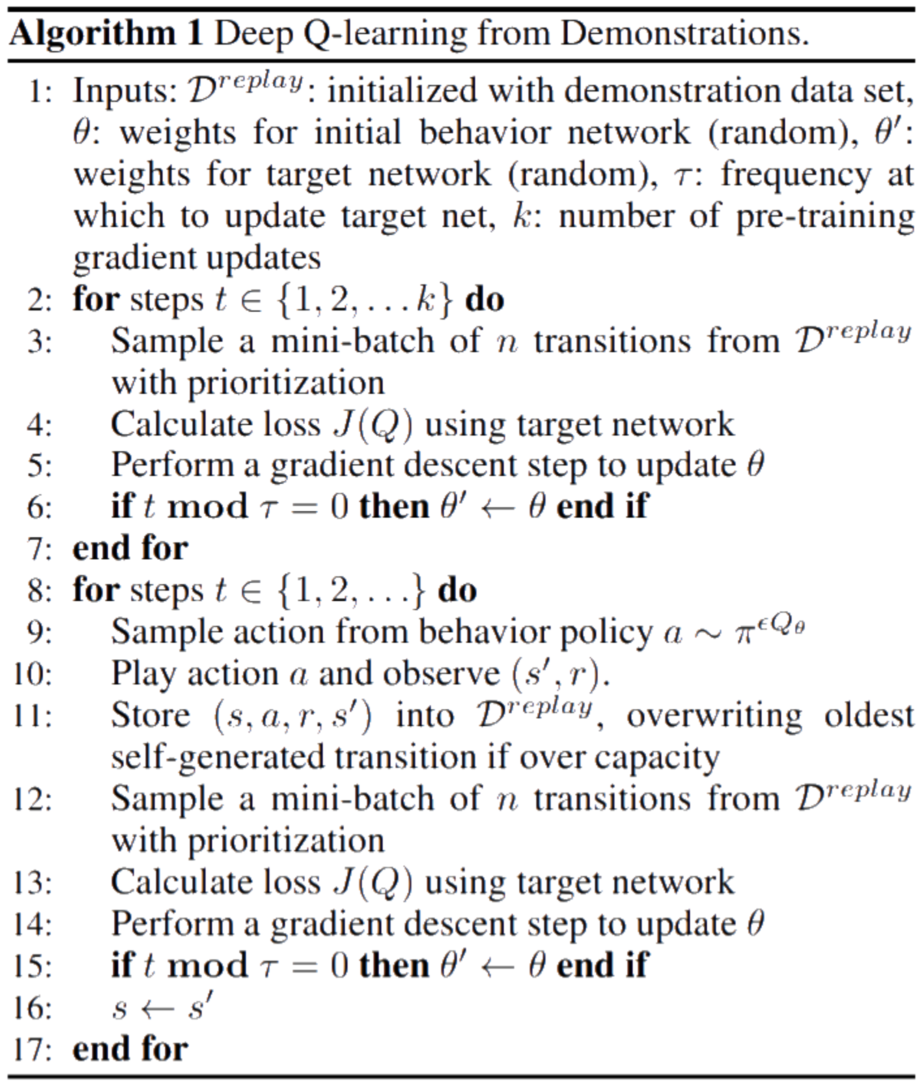
The λ parameters control the weighting between the losses.

We examine removing some of these losses in Section. Once the pre-training phase is complete, the agent starts acting on the system, collecting self-generated data, and adding it to its replay buffer D^{replay}. Data is added to the replay buffer until it is full, and then the agent starts overwriting old data in that buffer. However, the agent never over-writes the demonstration data. For proportional prioritized sampling, different small positive constants, $\epsilon_a\ and\ \epsilon_d$, are added to the priorities of the agent and demonstration transitions to control the relative sampling of demonstration versus agent data. All the losses are applied to the demonstration data in both phases, while the supervised loss is not applied to self-generated data.

Overall, Deep Q-learning from Demonstration (DQfD) differs from PDD DQN in six key ways:

- Demonstration data: DQfD is given a set of demonstration data, which it retains in its replay buffer permanently.
- Pre-training: DQfD initially trains solely on the demonstration data before starting any interaction with the environment.
- Supervised losses: In addition to TD losses, a large margin supervised loss is applied that pushes the value of the demonstrator's actions above the other action values.
- L2 Regularization losses: The algorithm also adds L2 regularization losses on the network weights to prevent over-fitting on the demonstration data.
- N-step TD losses: The agent updates its Q-network with targets from a mix of 1-step and n-step returns.
- Demonstration priority bonus: The priorities of demonstration transitions are given a bonus of ϵ_d, to boost the frequency that they are sampled.

Pseudo-code is sketched in **Algorithm 1**.



Algorithm 1 Deep Q-learning from Demonstrations.

```
 1: Inputs: $\mathcal{D}^{replay}$: initialized with demonstration data set,
    $\theta$: weights for initial behavior network (random), $\theta'$:
    weights for target network (random), $\tau$: frequency at
    which to update target net, $k$: number of pre-training
    gradient updates
 2: for steps $t \in \{1, 2, \dots k\}$ do
 3:     Sample a mini-batch of $n$ transitions from $\mathcal{D}^{replay}$
        with prioritization
 4:     Calculate loss $J(Q)$ using target network
 5:     Perform a gradient descent step to update $\theta$
 6:     if $t$ mod $\tau = 0$ then $\theta' \leftarrow \theta$ end if
 7: end for
 8: for steps $t \in \{1, 2, \dots\}$ do
 9:     Sample action from behavior policy $a \sim \pi^{\epsilon Q_\theta}$
10:     Play action $a$ and observe $(s', r)$.
11:     Store $(s, a, r, s')$ into $\mathcal{D}^{replay}$, overwriting oldest
        self-generated transition if over capacity
12:     Sample a mini-batch of $n$ transitions from $\mathcal{D}^{replay}$
        with prioritization
13:     Calculate loss $J(Q)$ using target network
14:     Perform a gradient descent step to update $\theta$
15:     if $t$ mod $\tau = 0$ then $\theta' \leftarrow \theta$ end if
16:     $s \leftarrow s'$
17: end for
```

13.2 Actor Network in DDQN

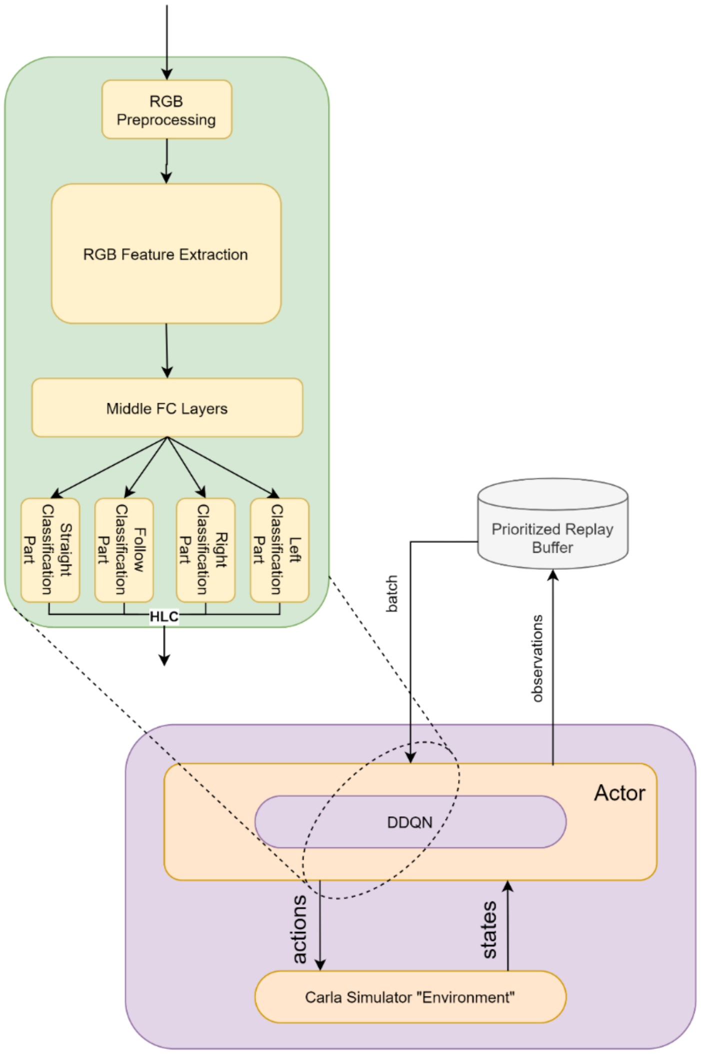

RGB Preprocessing
RGB Feature Extraction
Middle FC Layers
Straight Classification Part
Follow Classification Part
Right Classification Part
Left Classification Part
HLC
Prioritized Replay Buffer
batch
observations
Actor
DDQN
actions
states
Carla Simulator "Environment"

Figure 13-1 DDQN architecture diagram with conditional imitation network used inside the DDQN

The conditional network in (Codevilla, Muller, Lopez, Koltun, & Dosovitskiy, 2017) was used inside the DDQN, but the training was very sensitive to some hyper-parameters, table 12.1 describes the best hyper-parameters to be used

Table 13-1: summary of final-hyper-parameters' values

Parameter	Value	Reasons
Batch size	256 or bigger	Stability in learning
Losses ratio	1	Stability in learning from both demonstration and observation data
Discount factor	0.99	More stability
Exploration schedule	[1,0.01,0.9999]	Recommended experimentally by DeepMind
Learning rate and its schedule	0.01	Recommended experimentally by DeepMind
Number of pre training steps	750000 mini-batch updates	Recommended experimentally by DeepMind
Frequency of updating target network	10000 steps	Recommended experimentally by DeepMind
Frequency of updating online buffers	2000 steps	Recommended experimentally by us
Frequency of updating offline buffers	2000 steps	Recommended experimentally by us
Ratio of imitation and RL data in the interaction phase	0.1	Recommended experimentally by DeepMind
$l(a, a_E)$	0.8	Recommended experimentally by DeepMind
FC neurons	1024	Experimentally, more neurons are capable of capturing more information
Batch normalization	After all layers	Experimentally, removing batch normalization usually cause divergence
Dropout	0.2 after Convolution layers 0.5 after FC	Experimentally, dropout reduces multi-coolinearity and helps convergence on the long term
Reward at pre training	10	Experimentally, rewards above this value cause divergence

13.3 Action Space

Actions for controlling the car are continuous. Steering is in the range [-1,1], acceleration is in the range [-1,1] and brake in range [-1,1]. We chose to work with DDQN so firstly, we fused both acceleration and brake into one variable acc-brake so that brake falls in the range [-1,0] and acceleration is in the range [0,1]. Secondly, we discretized steering and acc-brake each to 20 actions, with a total 441 combination actions for each branch. Finally, we preprocessed the demonstration data by Intel (Codevilla, Muller, Lopez, Koltun, & Dosovitskiy, 2017) to match action space settings.

13.4 Prioritized Buffers Implementation

We first used an ordinary sorted array, but this dominated the run time. To scale to large memory sizes N and for proportional prioritization, we used 'sum-tree' data structure which is very similar in spirit to the array representation of a binary heap. However, instead of the usual heap property, the value of a parent node is the sum of its children. Leaf nodes store the transition priorities and the internal nodes are intermediate sums, with the parent node containing the sum over all priorities, p_{total}. This provides an efficient way of calculating the cumulative sum of priorities, allowing $O(log\ N)$ updates and sampling. To sample a mini batch of size k, the range *[0,* p_{total}*]* is divided equally into k ranges. Priorities are saved in the sum-tree and actual data are saved in an ordinary array **Invalid source specified.**. Software implementation is simply described in figure 13-2.

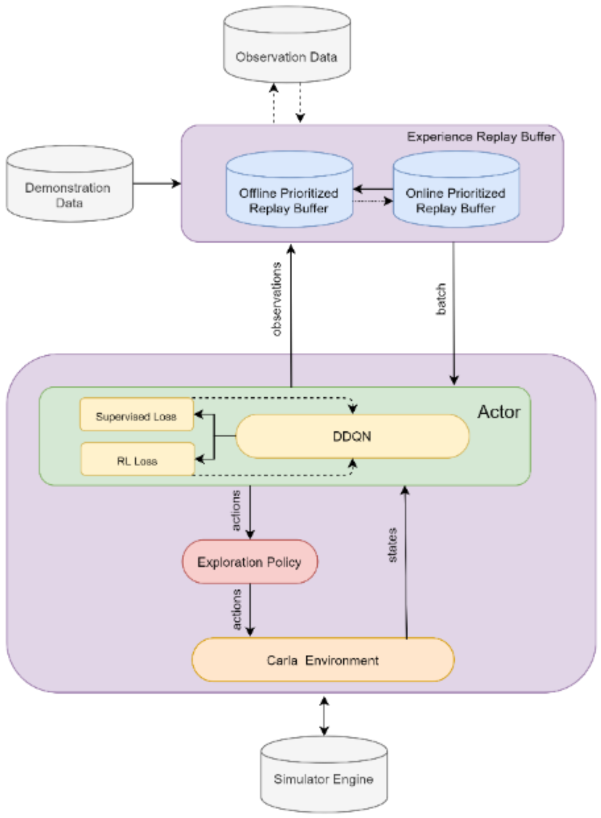


Figure 13-2 final DQFD architecture diagram

13.5 Prioritized Buffers Space Complexity and new prioritized caching method

To scale to large memory sizes N we have to use very large RAM sizes which is not practical. We adopted a new prioritized caching method as in figure 13-2 by using two kinds of buffers, an offline buffers that keeps track of the files' names on the disk and their current priorities and an online buffer that is a small buffer in the RAM which contains the actual data that is frequently sampled from the offline buffer. Using this idea is equivalent to using one large buffer and it is both time and space efficient.

We used 16 buffers in our method, 8 online buffers and 8 offline buffers, each of online buffers and offline buffers are subdivided to demonstration buffers and self-generated data buffers. There is a buffer for each head or output branch in our conditional imitation network structure.

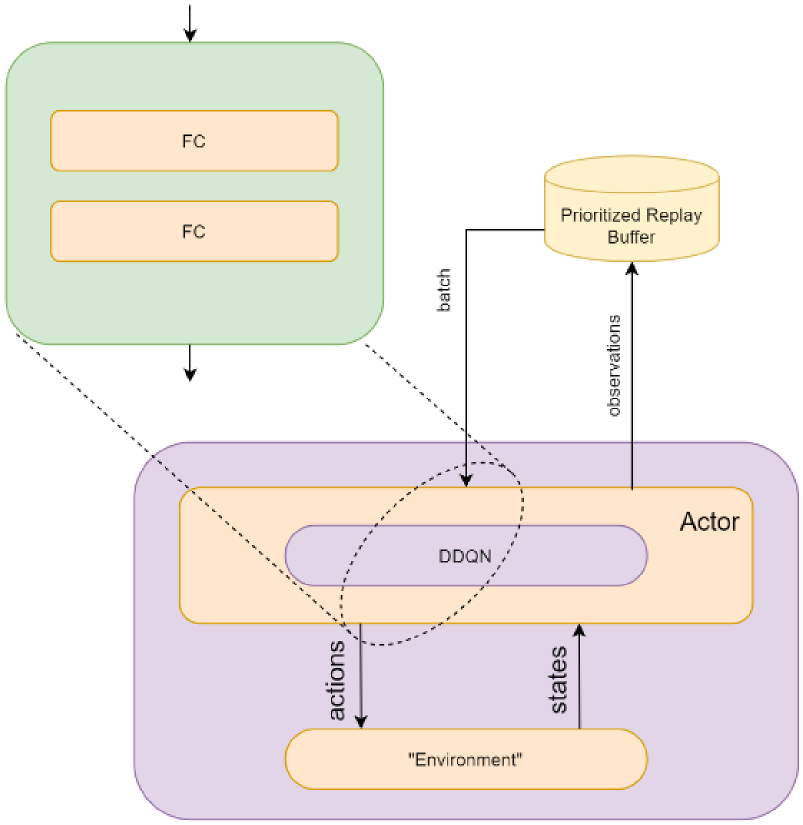


Figure 13-3 DQfD architecture diagram used with cartpole

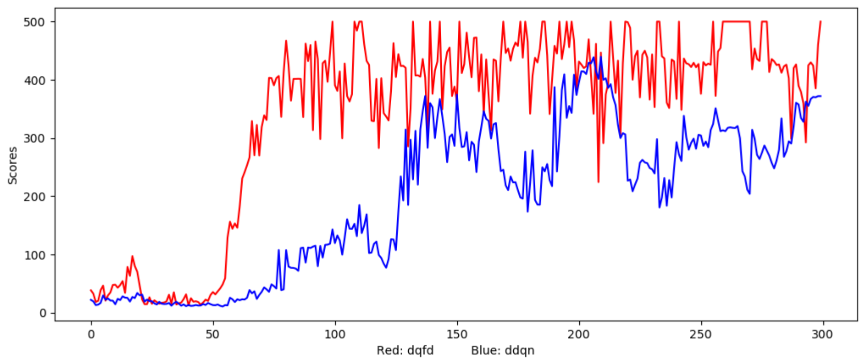


Figure 13-4 Returns of DQfD vs DDQN in the interaction phase with cartpole

13.5.1 Working with cartpole

As a proof of concept, we tested our code with a cartpole, we used a 2 FC layers in the DDQN to predict state-action value function and we ran the code twice, firstly we set the pretraining steps of the DQfD to 5000 minibatch updates. Secondly, we tried using DDQN alone. Figure 13-4 show the massive acceleration when using DQfD, and also it is capable of reaching higher returns.

13.6 Conclusion

Due to COVID-19 situation, we didn't manage to finish tuning the DQFD in the autonomous driving problem, but a proof of concept was done on cartpole problem. Working on DQfD in the autonomous driving problem is left as a future work.

Chapter 14 Conclusion, future work and recommendations

In this chapter, we will introduce two parts which are the conclusion and the future research works which could help in the extension of this project.

14.1 Conclusion

End to End Self Driving task is a very difficult problem to be solved, especially with algorithms that are still in a research area such as Reinforcement Learning algorithms. It might seem easy to apply them with a simulator, but in real life it is totally different. This is due to the numerous factors that aren't considered and which also can affect the vehicle, such as: Climate changes, pedestrians, congestion, traffic lights.

We applied three approaches, the supervised learning approaches using Imitation learning the Time dependency approach and the sensor fusion approach and the reinforcement learning approach using actor-critic algorithms.

We used the Intel self-driving model approach using conditional branches as Nvidia pilot-net as our base lines to enhance these model's performance by changing input from single image to a stack of frames represents time sequence or adding another input branch for the LiDAR point-cloud. Then, we trained and tested it with the help of the Urban Driving environment, e.g. CARLA Simulator.

We took the Intel model as our reference. It seemed fair to use the same data in camera only based architectures they used, and the collected data follows the same configurations for architectures that include LiDAR, also the exact same benchmark was used. Finally, we were able to outperform them in the same testing conditions and scenarios.

To go further with the project, we implemented the actor-critic network and carefully chose all the relevant parameters to enhance our model performance.

In conclusion our three contributions can be summarized as follows time dependent neural networks achieve high accuracy and more generalization in new driving scenarios, sensor fusion achieves more redundancy and higher accuracy, while reinforcement learning also achieves a higher accuracy but with online learning capability and a cost-effective deployment.

14.2 Future work

- Try Transformers instead of LSTM.
- Using GANs to predict future states instead of using the scenario of past frames in time-dependent neural network.
- Using self-supervised learning to predict future next states and using it in time-dependent settings.
- Try state of the art reinforcement learning algorithms integrated with supervised setting to achieve more stable and robust agents.
- Fine Tuning bigger models on real-life datasets.
- Enhancing fusion end-to-end networks with state-of-the-art LIDAR-based feature extraction methods like SA-SSD.

14.3 Recommendations

We highly recommend the following in any future work,

1. Tips and Tricks in Deep Learning Training

   https://github.com/Conchylicultor/Deep-Learning-Tricks#training

2. Tips and Tricks in Deep Reinforcement learning Training

   https://github.com/williamFalcon/DeepRLHacks

3. Never use Keras instead, use pure TF 1.15, TF 2.0 or Pytorch because it's very high level.

Chapter 15 Bibliography

Bahdanau, D., Cho, K., & Bengio, Y. (2015). Neural Machine Translation by Jointly Learning to Align and Translate .

Bellman, R. (1957b). “Dynamic Programming”.

Bellman, R. a. (1957a). “A Markovian decision process”.

Bertsekas, D. P. (1995). Dynamic programming and optimal control. Vol. 1. No. 2.

Codevilla, F., Muller, M., Lopez, A., Koltun, V., & Dosovitskiy, A. (2017). End-to-end Driving via Conditional Imitation Learning.

Hafner, R., & Riedmiller, M. (2011). “Reinforcement learning in feedback control”. Machine learning. 84(1-2): 137–169.

Hasselt, V., H., Guez, A., & Silver., D. (2016). “Deep Reinforcement Learning with Double Q-Learning.”.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., . . . Kavukcuoglu., K. (2016). “Asynchronous methods for deep reinforcement learning”. In: International Conference on Machine Learning.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., . . . Ostrovski, G. (2015). “Human-level control through deep reinforcement learning.

Munos, R., Stepleton, T., Harutyunyan, A., & Bellemare., M. (2016). “Safe and efficient off-policy reinforcement learning”. In: Advances in Neural Information Processing Systems. 1046–1054.

Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., & Riedmiller., M. (2014). “Deterministic Policy Gradient Algorithms”. In: ICML.

Sutton, R. S. (2017). Reinforcement Learning: An Introduction MIT Press.

Sutton, R. S., A. McAllester, D., P. Singh, S., & Mansour, Y. .. (2000). “Policy gradient methods for reinforcement learning with function approximation".

Thomas, & P. (2014). “Bias in natural actor-critic algorithms”. In: International Conference on Machine Learning. 441–448.

Watkins, C. J. (1992). “Q-learning”. Machine learning.

Williams, & J., R. (1992). “Simple statistical gradient-following algorithms for connectionist reinforcement learning”. Machine learning. 8(3-4):229–256.

Zhou, Y., & Tuzel, O. (2017). VoxelNet: End-to-End Learning for Point Cloud Based 3D Object Detection .