

POLITECNICO DI TORINO

Department of Control and Computer Engineering

Master of Science in Mechatronic Engineering



Simulative analysis of the applicability of the Inverse Reinforcement Learning for Advanced Driver Assistance systems

Academic Supervisor:

Prof. Stefano Alberto Malan

Candidate:

Francesco Allegra

Addfor S.p.A. supervisors:

Ing. Marco Fainello

Ing. Stefano Ballesio

Academic Year 2019/2020

Contents

Summary of notation	IV
Acronyms	VI
Introduction	1
1 Introduction to Artificial Intelligence	4
1.1 Supervised Learning	5
1.1.1 Linear Regression and Gradient Descent	8
1.1.2 Classification and Logistic Regression	12
1.1.3 Support Vector Machine.....	16
1.1.4 Artificial Neural Network.....	20
1.2 Unsupervised Learning.....	25
1.2.1 K-means Clustering	26
2 Reinforcement Learning	28
2.1 Markov Decision Process	31
2.2 Dynamic Programming.....	38
2.3 Reinforcement Learning for Prediction.....	43
2.3.1 Monte Carlo for Prediction.....	43
2.3.2 Temporal Difference 0.....	45
2.3.3 Temporal Difference lambda & Eligibility Trace	47
2.4 Reinforcement Learning for Control	50
2.4.1 Monte Carlo for Control	50
2.4.2 SARSA.....	52
2.4.3 Q-learning	53
2.4.4 Actor-Critic Methods.....	54
2.5 Function Approximation.....	57
2.5.1 Linear Function Approximation	59
2.5.2 Non-Linear Function Approximation and deep Q-network	60
3 Inverse Reinforcement Learning	62
3.1 The Inverse Reinforcement Learning Problem	62
3.2 Max-Margin IRL	66
3.2.1 Projection based method.....	71
3.3 Maximum Entropy IRL	73

4	ADAS and Vehicle Dynamics	76
4.1	Introduction to ADAS	76
4.1.1	Generalities of ACC	78
4.1.2	Generalities of LKC.....	82
4.2	Vehicle Model	85
4.2.1	Longitudinal vehicle dynamics.....	85
4.2.2	Lateral Vehicle dynamics	89
4.2.3	Simulink models	93
5	IRL strategy for ACC and LKC.....	95
5.1	Problem Presentation.....	95
5.2	Data collection and processing.....	96
5.2.1	ACC features extraction	98
5.2.2	LCK features extraction.....	100
5.3	Main Algorithm	101
5.3.1	Simulink environments.....	103
5.3.2	Diving Scenario Designer.....	106
5.3.3	Agent Design	108
6	Simulations and Results	110
6.1	ACC simulations.....	110
6.2	LKC simulations.....	118
7	Conclusions and future works	129
8	List of figures.....	133
9	References.....	136

Summary of Notation

Mathematical operators

Real-valued vectors are written in bold and in lower case while matrices are bold capitals.

\approx	approximately equal
\neq	not equal
$\mathbb{P}[X = x]$	probability that a random variable X takes on the value x
$X \sim p$ $x]$	the random variable X is selected with distribution $p(x) = \mathbb{P}[X =$
$\sum_{i=j}^k f_i$	summation of all the elements f_i , for i from j to k
$E[X]$	expectation of a random variable X
\forall	for each, e.g. $\forall x$
\in	is an element of, e.g. $x \in X$
\exists	exists
\mathbb{R}	set of real numbers
$\max_x f(x)$	maximum value of f with respect to x
$\min_x f(x)$	minimum value of f with respect to x
$\arg \max_x f(x)$	a value of x at which f takes its maximum value
$\log(x)$	natural logarithm of x
α	step size parameter
γ	discount factor
ε	probability of taking a random action in ε – <i>greedy</i> (Chapter 2)
δ	temporal difference error (Chapter 2)
δ_j	backpropagation error of j -th layer (Chapter 1)

λ	decay rate parameter for eligibility trace
i.i.d	independent and identically distributed
μ	features vector expectation
ϕ	features vector
ω	parameters vector of the reward function (from Chapter 2)

In a Markov Decision Process:

s, s'	states
a	action
t	discrete time step
T	final time step of an episode
s_t, a_t	state and action at time step t
\mathcal{S}	set of all non terminal states
\mathcal{T}	transition matrix
\mathcal{A}	set of all available actions
r	immediate reward
\mathcal{R}	reward function
$\pi(s)$	action taken in state s under a deterministic policy π
$\pi(a s)$	probability of taking action a in state s under stochastic policy π
G_t	return (cumulative discounted reward) following time t
G_t^n	n -step return
G_t^λ	λ -return
$p(s', r s, a)$	probability of going in state s' with reward r taking action a in state s
$v_\pi(s)$	value of state s under policy π (expected return)
$q_\pi(s)$	value of taking action a in state s under policy π
π_*	optimal policy
$v_*(s)$	value of state s under the optimal policy π_*
$q_*(s)$	value of taking action a in state s under the optimal policy π_*

Acronyms

ML	Machine Learning
SL	Supervised Learning
ANN	Artificial Neural Network
DNN	Deep Neural Network
MLP	Multilayer Perceptron
SVM	Support Vector Machine
UL	Unsupervised Learning
RL	Reinforcement Learning
IRL	Inverse Reinforcement Learning
MDP	Markov Decision Process
MRP	Markov Reward Process
DP	Dynamic Programming
PI	Policy Iteration
MC	Monte Carlo
TD	Temporal Difference
GPI	General Policy Iteration
MSE	Mean Square Error
DQN	Deep Q-Network
QP	Quadratic Programming
ACC	Adaptive Cruise Control
ADAS	Advanced Driver Assistance Systems
LKC	Lane Keeping Control
EV	Ego Vehicle
PV	Preceding Vehicle
CoG	Centre of Gravity
DOF	Degrees of Freedom

Introduction

This project has been developed in collaboration with my team co-worker Stefano D’Aiuto, a mechatronic engineering student, at Addfor S.p.A. and following the guidelines of our academic supervisor, Stefano Alberto Malan. The writing of the thesis has been divided as follows: I wrote chapters 1-3-6 while Stefano D’Aiuto wrote chapters 2-4-5, and the results achieved so far have been summarized in “Conclusions and future works”.

In this thesis the analysis of the application of Inverse Reinforcement learning algorithms for the realization of Advanced Driver Assistance Systems (ADAS) has been performed.

Automotive industries and research groups are spending a lot of financial and human resources to solve the complex problem of self-driving cars which is having a strong impact in the whole world during the last decades, but just recently, thanks to the big steps forward done in the artificial intelligence domain, it seemed possible to find a concrete solution.

Nowadays the aim of scientists and engineers is to simplify the human daily routine by automatizing first the simplest everyday tasks and then also the complex ones, through robotics and Artificial Intelligence, two different fields who apparently must grow in parallel. Their effort is even devoted to the improvement of the human-machine interaction so that robots start to behave in every possible way to make human life much more comfortable and stop to regard human beings just as obstacles to avoid.

As it has been already highlighted, among all the possible goals, “autonomous driving” is one of the most challenging, since it is a task performed by a huge amount of people everyday, and that is becoming more and more complex as the number of vehicles grows up during the years. The complexity of driving lies in the fact that a driving scenario is strongly unpredictable and unstructured, either in highways and especially in a city: pedestrians across the roads, overtaking maneuvers, traffic lights and road signals that must be respected, traffic congestion and so on. Furthermore, it has been shown that most of car accident causes can mostly be attributed to human-driver’s misjudgments or distractions. These are some of the reasons which led to the introduction of advanced driving-assistance systems (ADAS) in

modern urban cars, that assist the driver in some of the fundamental tasks that can be executed while driving (regulate the acceleration, parking, maintain the lane), but according to the Society of Automotive engineers (SAE) we are still far from the highest level of automation (Level 5) that indicates the ability of the car of driving by itself without any human input. At the moment, ADAS are implemented through classical control techniques, e.g. P.I.D, P.I or P.D., or using some advanced control theories like Model predictive control (MPC).

Simultaneously, Artificial Intelligence (AI) has acquired a lot of importance, spreading in a wide range of engineering sectors since it provides solutions in a very efficient and clever manner; for example Computer Vision is a key point of mobile robotics and even of ADAS, as these systems have to receive in input every kind of information (2D-images, 3-D maps, presence of a leading vehicle etc.) coming from the surrounding environment. Anyway, Machine Learning (ML) is not only used to solve classification, regression or clustering problems as in supervised and unsupervised learning; indeed Reinforcement Learning (RL), another branch of ML, can effectively be involved as a decision-making problem solver, and it is just here that the control problem in autonomous driving meets machine learning.

The focal point of RL is the Reward, a feedback signal able to show how good or bad are the taken actions, so RL is the right choice only if we are dealing with tasks that can be accomplished in a small environment requiring discrete state and action spaces, instead if the attention is focused on complex scenarios with continuous spaces, it leads to wrong result, since the correctness of the chosen reward cannot be guaranteed. Those limits opened the door to a new theory, introduced by Andre Ng and Russel in 2000s as inverse reinforcement learning, which aim is to find the unknown reward in the cases cited before.

The objective of this thesis is to highlight the advantages and the limits of using the projection-based method, an inverse reinforcement learning algorithm, to design the high level control of an adaptive cruise control and a lane keeping control, given a set of data provided by Addfor S.p.A., which describes the driving style of two different expert human drivers that our agent has to emulate.

Eventually, the two controls are able to accomplish those tasks in a way as close as possible to a human manner.

The entire code has been written in MATLAB while the scheme containing the agent block, the vehicle dynamic block and the sensors blocks, has been realized in Simulink. Moreover, driving scenario Toolbox has been used to build a driving scenario ad hoc for the problem. In “Conclusions and future works” the obtained results have been discussed and possible solutions have been provided to improve the proposed method.

Chapter 1

1 Introduction to Artificial Intelligence

In this chapter a general idea of Artificial Intelligence (AI) and Machine Learning (ML) is presented. It is important to note that many of explanations and demonstrations described in this first chapter are a reinterpretation and inspired mainly by Andrew Ng Stanford course [3].

Machine Learning is a field focused on the construction of algorithms that make predictions based on data. A ML task aims to identify and learn a function $f: X \rightarrow Y$ that maps the input domain X (of data) onto output domain Y (of possible predictions) [1]. Functions f are chosen from different function classes, dependent on the type of learning algorithm that is being used. Different definitions of Machine Learning exist in bibliography. An older and informal definition is provided by Arthur Samuel that described it as:

"the field of study that gives computers the ability to learn without being explicitly programmed."

Tom Mitchell provides a more modern and practical definition:

"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P , if its performance at tasks in T , as measured by P , improves with experience E " [2].

In this definition the performance measure P provides quantitatively how well a certain ML algorithm is performing. For a classification task, the accuracy of the system is usually chosen

as the performance measure, where accuracy is defined as the proportion for which the system correctly produces the output. E is the experience that Machine Learning algorithms learn through datasets containing a set of examples that are used to train and test these algorithms.

Furthermore, ML is subdivided in two main fields, each one focusing on a specific task: Supervised and Unsupervised Learning. In Supervised Learning (SL) model, the algorithm learns and trains on a labeled dataset, the so called ground truth, that gives a priori knowledge of what the output values for the samples must be and the main goal is to build a mapping function from the input to the output that is able to predict well the output variable (i.e. assigning the correct label) when new input data are given. Whereas in Unsupervised Learning (UL) the goal is to infer the underlying or hidden structure within a set of input unlabeled data in order to learn more about data themselves.

Moreover, another branch of ML is Reinforcement Learning (RL). The Reinforcement Learning algorithm strategy is based on a reward system, providing feedback from the environment in which the agent is dropped into, and proceeds based on trials and error logic. This RL topic is investigated and analyzed more in depth and further ahead in Chapter 2.

1.1 Supervised Learning

The majority of applications related to Machine Learning falls into the Supervised Learning problem that is indeed the most common subbranch of ML today. SL algorithms are designed to learn by example and indeed the name “supervised” learning itself originates from the idea that training these algorithms is like to have an entity that behaves like a teacher that tells if an output is correct or not. Hence a labelled dataset is provided to the algorithm in which a “right answer” (ground truth) is given a priori and then the algorithm learns from this training data. Then once the algorithm is trained, it provides a “right” answers based on new data, where the output is unknown, finding a relationship between the input and the output.

Moreover, supervised learning problems are categorized into "regression" and "classification" problems.

In a regression problem the aim is to predict results within a continuous output, this means mapping input variables to some continuous function.

Instead in a classification problem, the aim is to predict results in a discrete output, and this means to classify data into one of two or more discrete number (classes or categories) mapping input variables into discrete categories. Some examples extracted from Andrew Ng's course [3] are shown below

- Classification examples
 - A classification problem is when the output variable is a category, such as “red” or “blue” or “disease” and “no disease”.
 - Given a patient with a tumor, predict whether the tumor is malignant or benign.

- Regression examples
 - A regression problem is when the output variable is a real value, such as “value in dollars” or “weight”
 - On the basis of a given picture of a person, predict his/her age

In order to going further into more technical explanations it is better to establish a common. The “input” variables are denoted with $x^{(i)}$ that is generally called input features while for the predicted value is used $y^{(i)}$ in order to indicate the target value or the “output”. The training example is the pair $(x^{(i)}, y^{(i)})$ with $i = 1, \dots, m$ is the training set. Furthermore, X indicates

the input values space and Y the output values space. The aim of supervised learning is to learn a function $h: X \rightarrow Y$ given a training set $(x^{(i)}, y^{(i)})$ so that the learned function $h(x)$ is a “good” predictor for the corresponding value of y [3].

Then, the learning process is a regression problem when the predicted target variables y is a continuous variable. When the target y can take on only a small number of discrete values it is called classification problem.

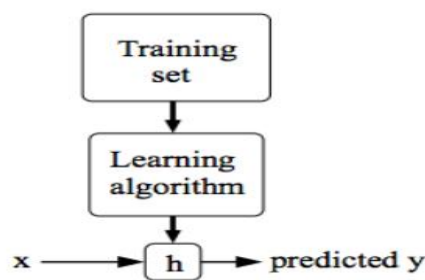


Figure 1.1: learning problem process

Some popular examples of supervised ML algorithms are *Linear Regression*, *Logistic Regression*, *Random Forest*, *Support Vector Machine* and *Neural Networks*.

It is also of primary importance to highlight that all these Machine Learning algorithms completely depend on optimization techniques, that are the process by which the optimal set of parameters are found. Here Gradient Descent will be discussed that is the simplest and used optimization technique.

Other two important concepts are *model complexity*, and *the bias-variance tradeoff* that are related quantities. Model complexity refers to the complexity of the function that the algorithms is attempting to learn (similar to the degree of a polynomial). The proper level of model complexity is generally determined by the nature of the training data. If only a small amount of data is provided, or if the data are not uniformly spread throughout different possible scenarios, then the best choice is to opt for a low-complexity model. This is because a high-complexity model will overfit if used on a small number of data points. Overfitting refers to learning a function that fits training data very well, but does not generalize to new and unseen

data points — basically in this case the algorithm is learning to produce the training data without learning the real underlying structure in the data that leads to this output.

Therefore *bias-variance tradeoff* relates to model generalization and any model must be well-balanced between bias, which is the constant error term, and variance, which is the amount by which the error may vary between different training sets. Note that bias and variance typically move in opposite directions from each other; increasing bias will usually lead to lower variance, and vice versa, so it is necessary to find the best trade-off suitable for the specific cases.

1.1.1 Linear Regression and Gradient Descent

In linear regression a linear relationship is assumed between the input variables x and the single output variable y and can be represented by the equation:

$$y = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n \quad (1.1)$$

- y is the predicted value
- θ_0 is the bias term
- $\theta_1 \dots \theta_n$ are the model parameters
- $x_1 \dots x_n$ are the features values
- n is the number of features

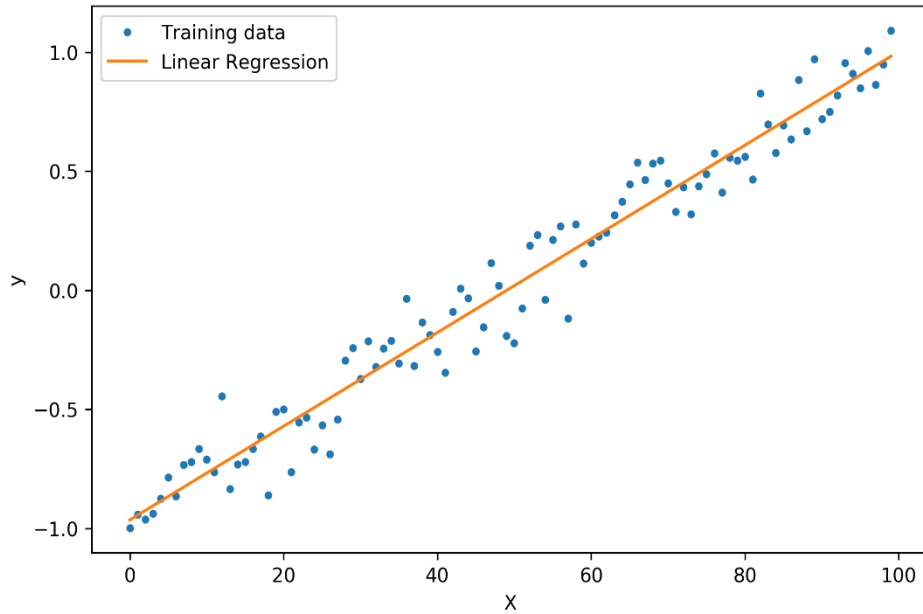


Figure 1.2: Linear Regression

Eq. (1.1) can be also re-written as $y = \boldsymbol{\theta}^T \boldsymbol{x}$ in which the $\boldsymbol{\theta}$ is the vector parameter term including the bias and \boldsymbol{x} is the input features vector.

The primary goal of ML algorithms is always to build a model, which is basically a hypothesis function which can be used to find an estimation for y based on \boldsymbol{x} the input features:

$$y = \boldsymbol{\theta}^T \boldsymbol{x} \quad (1.2)$$

The hypothesis function is: $h(\boldsymbol{x}) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n \quad (1.3)$

The $h(\boldsymbol{x})$ takes in input the input features \boldsymbol{x} (the number of features is indicated with n) and gives as output the estimated value y . In order to measure the goodness and the accuracy of this hypothesis function a cost function is used. The cost function is defined as:

“a function that maps an event or values of one or more variables onto a real number intuitively representing some “cost” associated with the event.” [4].

The cost function is defined as the average difference between all the results of the hypothesis and the actual predicted output y :

$$J(\theta_0, \theta_1, \dots, \theta_n) = \frac{1}{2m} \sum_{i=1}^m (\hat{y}^{(i)} - y^{(i)})^2 = \frac{1}{2m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2. \quad (1.4)$$

In the formula above m is the number of training example $(x^{(i)}, y^{(i)})$, x are the input features and y are the output “target” variable. This specific cost function is the Mean Square Error loss function (MSE (1.4)) or also called Quadratic loss or L2 loss, that is basically the sum of squared distances between the target variable y_i (ground truth) and predicted values \hat{y}_i and this loss function is a way of measuring how well the model $h(x)$ fits into the data. The difference between the predicted values and ground truth measures the error difference $e_i = \hat{y}_i - y_i$. Other loss functions can be used: the Mean Absolute Error (MAE or L1 loss), Huber loss (Smooth Mean Absolute Error) and the Cross-entropy loss (or log-loss) and all of these provide a different measure of how good a prediction model does in terms of being able to predict the expected outcome. There is not a single loss function that works for all kind of dataset but each one can perform at their best in different cases. Then, once a proper loss function has been selected a common optimization method to find the minimum $\min_{\theta} J(\theta)$ is the *Gradient Descent*.

The best fit line is considered to be the line for which the error between the predicted values and the observed values is minimum. It is also called the regression line and the errors are also known as residuals as shown in *figure 1.3*. Residuals can be visualized by the vertical lines from the observed data value to the regression line.

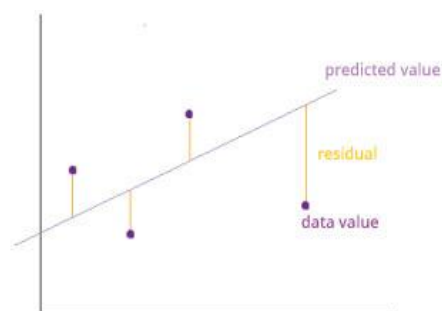


Figure 1.3: Regressor Line and residual

Optimization by definition is the process by which an optimum is achieved. The simplest and most popular optimization algorithm in Machine Learning is the Gradient Descent that is used

to update parameters in a model (parameters can vary according to the algorithms, such as *coefficients* in Linear Regression and weights in Neural Networks) that helps ML models to find out paths to a minimum value using repeated steps.

Parameters update through gradient descent is made by taking the derivative (the tangential line to a function) of the cost function. The slope of the tangent is the derivative at that point and therefore it provides information about the direction in which the gradient is moving at each step. The size of each step is determined by the parameter α , which is called the learning rate [3]. For example, as shown in *figure 1.4*, the distance between each 'star' in the graph represents a step determined by the parameter α . A smaller α would result in a smaller step and a larger α results in a larger step. The direction in which the step is taken is determined by the partial derivative of $J(\theta)$. *Figure 1.4* shows two different starting points that end up in two different places.

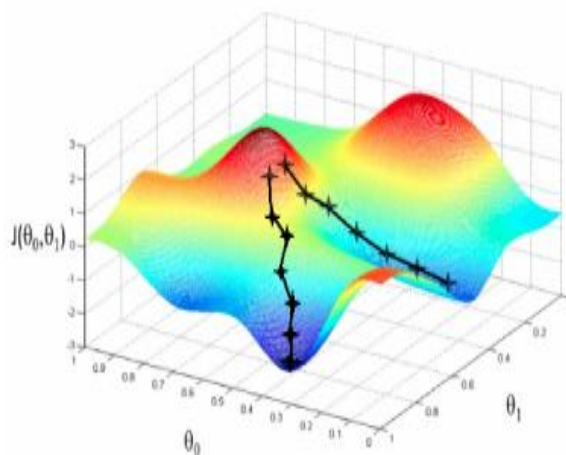


Figure 1.4: Gradient Descent moving on a cost function case of only two features 3D plot

The gradient descent algorithm is:

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \quad (1.5)$$

Where $j = 0, 1, \dots, n$ represents the feature index number and the parameters $\theta_0, \theta_1, \dots, \theta_n$ must be simultaneously updated at each iteration j . Then the process is repeated until convergence.

The repeated application of these gradient descent equations will lead the model to become more and more accurate. This method looks at every example in the entire training set on every step, and is called batch gradient descent [3]. Note that, while in general gradient descent can be susceptible to local minima, the optimization problem posed here for linear regression has only one global, and no other local, optimum. Thus, gradient descent always converges (assuming the learning rate α is not too large) to the global minimum since in this case J is a convex quadratic function.

1.1.2 Classification and Logistic Regression

Classification is the process of predicting the class of a given set of data points. Classes are sometimes called as targets, labels or categories. In a more formal way *classification* is the task of approximating a mapping function $h(x)$ from input variables x to discrete output variables y . The output variables are often called labels or categories and the mapping function predicts the class or category for a given observation. Intuitively the classification problem seems just like the regression problem, except that the predicted values take on only a small number of discrete values. In the first place the binary classification problem will be treated, in which y can take on only two values, 0 and 1. Then it will be also generalized to the multi-class classification case.

Classification can answer questions like:

- Find out which disease a patient is affected by, who has a number of symptoms. Then, attribute the set of symptoms from which the patient is affected to the correct disease
- Check transactions in order to classify fraudulent ones

In accordance with the previously established formalism, $x^{(i)}$ represents the input features and $y^{(i)}$ the predicted value. The aim in a classification problem is to learn a function $h: X \rightarrow Y$ given a training set $(x^{(i)}, y^{(i)})$ so that the learned function $h(x)$ is able to classify and assign new data to the correct classes. In binary classification $y \in \{0,1\}$. Notice that 0 is called the

negative class, and 1 the positive class, and they are sometimes also denoted by the symbols “-” and “+”.

Since the classification problem is very similar to the regression problem it may seem a sensible choice to classify using linear regression and map all predictions greater than 0.5 as a 1 and all less than 0.5 as a 0. However, a linear model does not output probabilities, but it treats the classes as numbers (0 and 1) and fits the best hyperplane (for a single feature, it is a line) that minimizes the distances between the points and the hyperplane. So, it simply interpolates between the points, and its output cannot be interpreted as probability. A linear model also extrapolates and gives values below zero and above one so this method does not work well because classification is not actually a linear function as shown in *figure 1.5*.

A solution for classification is Logistic Regression that is a statistical method for classifying dataset in which there are one or more independent variables determining an outcome (in case of binary classification it takes only two possible values). So Logistic regression models the probabilities for classification problems with two possible outcomes and it is an extension of the linear regression model for classification problems. Logistic regression is part of the so called “Discriminative Models” subgroup of ML methods like Support Vector Machines (SVM) and Perceptron in which linear equations are used as building blocks [3].

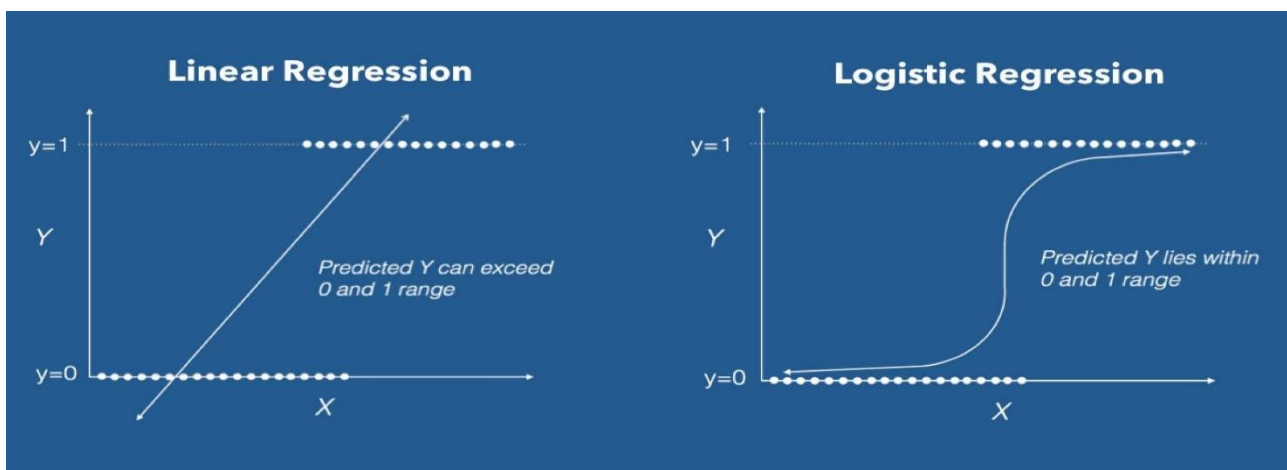


Figure 1.5: Linear Regression and Logistic Regression for classification

Instead of fitting a straight line or hyperplane, the logistic regression model uses the logistic function to compress the output of a linear equation between 0 and 1.

Intuitively, it also does not make sense for $h_{\theta}(x)$ to take values larger than 1 or smaller than 0 since the output $y \in \{0, 1\}$. Then the hypothesis function $h_{\theta}(x)$ has to be bounded between $0 \leq h_{\theta}(x) \leq 1$ and this can be easily done plugging $\theta^T x$ into the Logistic Function or the Sigmoid Function (figure 1.6):

$$h_{\theta}(x) = f(\theta^T x) \quad (1.6)$$

$$z = \theta^T x \quad (1.7)$$

$$f(z) = \frac{1}{1+e^{-z}} \quad (1.8)$$

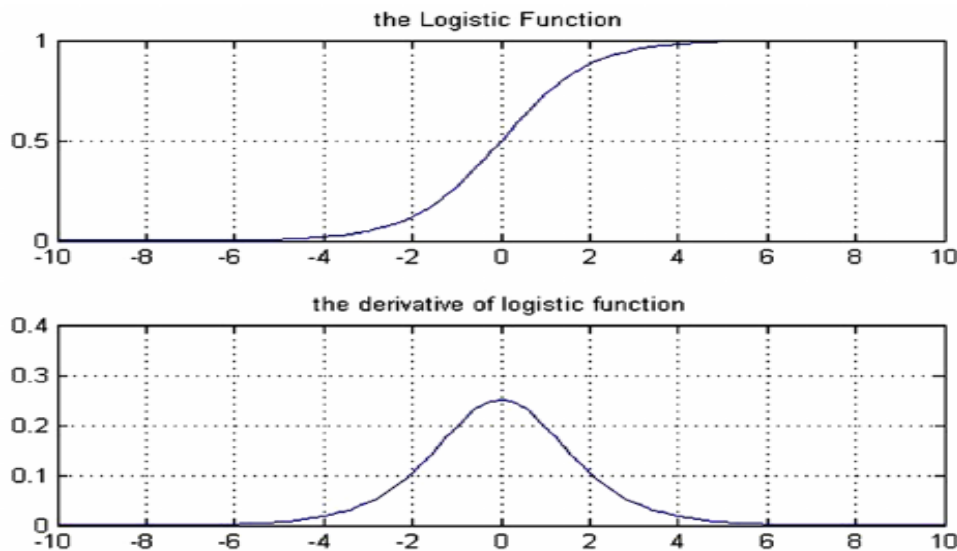


Figure 1.6: Logistic Function

The sigmoid function $f(z)$ maps any real number to the $(0, 1)$ interval, making it useful for transforming an arbitrary-valued function like $h_{\theta}(x) = \theta^T x$ into a function useful for classification purposes $f(z) = \frac{1}{1+e^{-\theta^T x}}$, $h_{\theta}(x)$ gives the probability that the output is 1.

In order to get discrete 0 or 1 classification, the output of the hypothesis function behave as follows:

$$h_{\theta}(x) \geq 0.5 \rightarrow y = 1 \quad h_{\theta}(x) < 0.5 \rightarrow y = 0$$

Therefore, the way the logistic function $h_{\theta}(x) = f(\theta^T x)$ behaves is that when its input is greater than or equal to zero, its output is greater than or equal to 0.5 that in this case is the selected threshold:

$$f(z) \geq 0.5 \text{ when } z \geq 0 \text{ where } z = \theta^T x$$

$$h_{\theta}(x) = f(\theta^T x) \geq 0.5 \quad \text{when } \theta^T x \geq 0 \Rightarrow y = 1 \quad (1.9)$$

$$h_{\theta}(x) = f(\theta^T x) < 0.5 \quad \text{when } \theta^T x < 0 \Rightarrow y = 0 \quad (1.10)$$

The decision boundary is created by the model hypothesis function $h_{\theta}(x)$ and it is the line that separates the area where $y = 0$ and where $y = 1$ (figure 1.7).

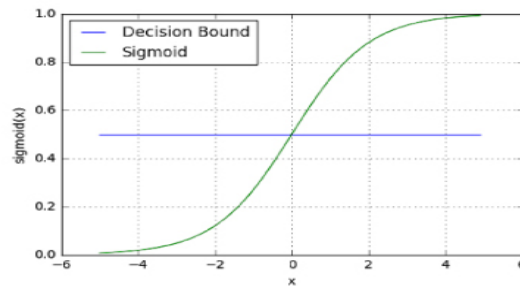


Figure 1.7: Sigmoid Function and Decision boundary threshold

As already seen in Linear Regression, the **Gradient Descent** optimization algorithm can be used in order to minimize the cost function and find the optimal parameters. However, the cost function used for linear regression cannot be reused in this case because Logistic Function will produce a wavy cost function with many local minima and therefore will not be a convex function.

The loss function for Logistic Regression is defined as

$$J(\theta) = \frac{1}{m} \sum_{i=1}^m \text{Cost}(h_{\theta}(x^{(i)}), y^{(i)}) \quad (1.11)$$

Where :

$$\text{Cost}_1(h_{\theta}(x), y) = -\log(h_{\theta}(x)) \quad \text{if } y = 1 \quad (1.12)$$

$$\text{Cost}_0(h_{\theta}(x), y) = -\log(1 - h_{\theta}(x)) \quad \text{if } y = 0 \quad (1.13)$$

Eq. (1.11) can be rewritten in a single and more compact cost function as follows:

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^m [y^{(i)} \log(h_{\theta}(x^{(i)})) - (1 - y^{(i)}) \log(1 - h_{\theta}(x^{(i)}))] \quad (1.14)$$

Based on the number of categories, Logistic regression can be classified as:

1. **Binary classification:** target variable can have only 2 possible types: “0” or “1” which may represent “win” vs “loss”, “pass” vs “fail, etc.
2. **Multiclass classification:** target variable can have 3 or more possible types which are not ordered (i.e. types have no quantitative significance) like “disease A” vs “disease B” vs “disease C”.

In multiclass classification the classical One-vs-All technique is used in which one class is chosen and all the others are grouped together into a single second class. Then binary logistic regression for each case is applied repeatedly. Finally, the hypothesis function which returns the highest value is selected as the prediction.

1.1.3 SVM - Support Vector Machine

Support Vector Machines (SVMs) are an extremely powerful and modern model implementation of a classification technique and can be used in classification and regression problems. SVMs aim at finding decision boundaries that separate observations belonging to different membership classes and therefore SVM is a Discriminative Classifier formally defined by a separating hyperplane.

This kind of classifier models are highly accurate and are able to compute and process the high-dimensional data and they are very flexible in modelling diverse types of data. Moreover, SVMs belong to the general category of kernel methods and thanks to the kernel

trick a non-linear classification boundary can be easily generated using a method designed for linear classifiers (Linear SVM).

In the binary classification case, let $(x^{(i)}, y^{(i)})$ with $i = 1, \dots, m$ be the training dataset where $x^{(i)}$ are the feature vectors representing the observation and $y^{(i)} \in \{0,1\}$ be the labels of the instances. Support vector learning is the problem of finding a separating hyperplane that separates the positive examples

(labeled +1) from the negative examples (labeled -1) with the largest margin. The

margin of the hyperplane is defined as the shortest distance between the positive

and negative instances (training data sample) that are closest to the hyperplane. The intuition behind searching for the hyperplane with a large margin is that it is more resistant to noise than a hyperplane with a smaller margin [5]. The maximal margin classifier is the hyperplane for which the margin is the largest [6].

Formally, suppose that all the data satisfy the constraints

$$\omega \cdot x_i + b \geq +1 \quad y_i = +1$$

$$\omega \cdot x_i + b \leq -1 \quad y_i = -1$$

ω is the normal to the hyperplane and $|b|/\|\omega\|$ is the perpendicular distance from the hyperplane to the origin, and $\|\omega\|$ is the Euclidean norm of ω .

The two constraints above can be conveniently combined into the following

$$y_i(\omega \cdot x_i + b) \geq 1 \quad \forall i \quad (1.15)$$

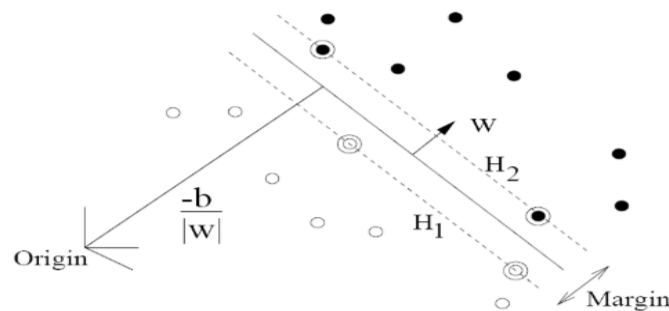


Figure 1.8: Hyperplane separating two classes with the maximum margin.

In *figure 1.8* the circled samples (black circles are negative samples while the white ones are the positive instances) lying on the canonical hyperplanes are called support vectors.

It can be shown that the maximum margin separating hyperplane can be constructed by solving the Primal optimization problem (1.16):

$$\min_{\omega \in H} \frac{1}{2} \|\omega\|^2 \quad \text{subject to} \quad y_i(\omega \cdot x_i + b) \geq 1 \quad \forall i \quad (1.16)$$

Using the same formalism as in the other chapters, the Support Vector Machine cost function can be expressed like this:

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^m [y^{(i)} \text{Cost}_1(h_\theta(x), y) - (1 - y^{(i)}) \text{Cost}_0(h_\theta(x), y)] \quad (1.17)$$

Notice that the newly θ parameters coincide with the previously defined parameters ω .

The SVM cost function is basically the same identical formula for logistic regression (1.14) with the only difference that now the $\text{Cost}_1(h_\theta(x), y)$ and $\text{Cost}_0(h_\theta(x), y)$ are Rectifier functions (**ReLU**) as shown in Figure (1.9) so that:

$$\theta^T x > 1 \quad \text{for} \quad y = 1 \quad (1.18)$$

$$\theta^T x < -1 \quad \text{for} \quad y = 0 \quad (1.19)$$

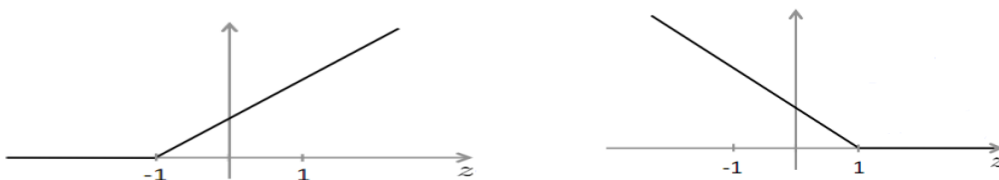


Figure 1.9: ReLu, $\text{Cost}_1(\theta^T x)$ on the right and $\text{Cost}_0(\theta^T x)$ on the left

Now the model must learn parameters θ such that $\theta^T \mathbf{x} > 1$ for $y = 1$ and $\theta^T \mathbf{x} < -1$ for $y = 0$. This is related to the large-margin intuition. The overall objective for a SVM looks like:

$$\min_{\theta} C \sum_{i=1}^m [y^{(i)} Cost_1(\theta^T \mathbf{x}^{(i)}) - (1 - y^{(i)}) Cost_0(\theta^T \mathbf{x}^{(i)})] + \frac{1}{2} \sum_{j=1}^n \theta_j^2 \quad (1.20)$$

Notice how λ (the regularization parameter which governs the trade-off between overfitting and the complexity of the model) has been replaced by another constant that plays the same role $C = \frac{1}{\lambda}$.

Examining equation (1.20) it is clear how a Support Vector Machine chooses a decision boundary which maximizes the margins from all the classes. For the purposes of this discussion, it is assumed that the SVM constant C is set to a very large value, $C \rightarrow \infty$. A large value of C acts like a small value of λ , this is like having SVM with no regularization, and this means that the model will be subject to overfitting.

As C goes up, the penalty associated to the term with $Cost_1$ and $Cost_0$ goes up, so the SVM is highly incentivized to make both the costs equal to 0 (i.e. make $\|\theta^T \mathbf{x}\| > 1$). Then assuming a large value of C , Equation (1.20) can be reformulated as:

$$\min_{\theta} \frac{1}{2} \sum_{j=1}^n \theta_j^2 \quad \text{s.t.} \quad \|\theta^T \mathbf{x}^{(i)}\| > 1. \quad (1.21)$$

Moreover, SVM is a very powerful tool thanks to the kernel trick that makes also possible to classify non linearly separable dataset. What the kernel trick does is to use some mapping function $K(x)$, called kernels, that projects the data onto a higher-dimensional space where it is possible to find a linear decision boundary.

Indeed, the basic idea with nonlinear SVMs is to map training data into a higher dimensional feature space via the kernel function $K(x)$ and construct a separating hyperplane with maximum margin in the input space. It can be shown that a linear decision function in the feature space corresponds to a non-linear decision boundary in the original input space.

1.1.4 Artificial Neural Networks

Artificial Neural Networks (ANN) are a powerful and universal tool, inspired by biological nervous system and the human brain, used mainly for pattern recognition, computer vision and function approximation. Indeed, when dealing with a complex, partially unknown and non-linear system ANN are the “perfect” tool for fulfilling the task of non-linear function approximator.

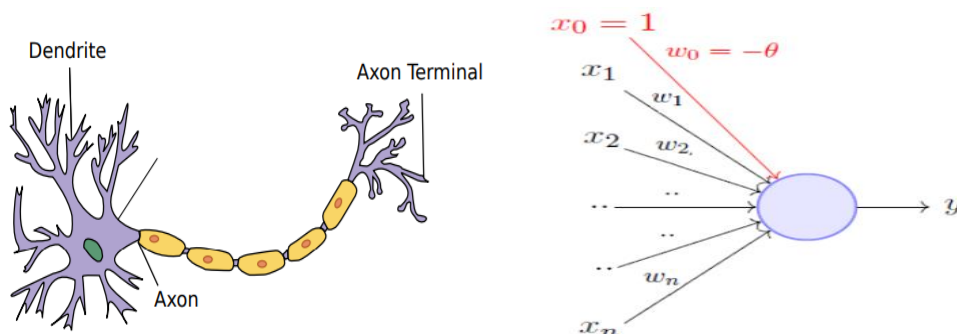


Figure 1.10: Biological neuron model (Wikipedia image) in nature (left) and mathematical model (right)

Non-linear decision boundary can be done mainly in two different ways:

- Adding non-linear features (i.e. x_1x_2, x_1^2)
- Using ANN modeling a non-linear decision boundary by internalizing the non-linearity

Neurons are computational units that take inputs (dendrites) as electrical inputs (called "spikes") that are connected to outputs (axons) as shown in *figure 1.10*. Dendrites are like the input features (x_0, x_1, \dots, x_n) and the output is the result of the hypothesis function $h_\theta(x)$ [3].

In neural networks a logistic or sigmoid activation function is often used: $g = \frac{1}{1+e^{-\omega^T x}}$. The ω parameters are "weights" of the network.

The simplest structure for a neural network is the Perceptron that is a simple binary classification algorithm proposed by Frank Rosenblatt in 1957. Its basic unit component is the so called artificial neuron. The perceptron structure is formed by an input layer, that stored the components of the input vector \mathbf{x} , the channels of weights ω that connect the input layer to the neuron and finally the body of the neuron (soma) as shown in the *figure 1.11*.

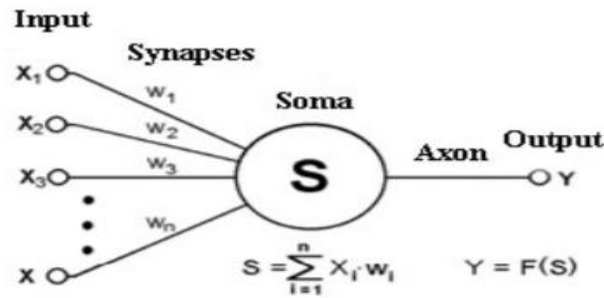


Figure 1.11: Perceptron structure

As already said (x_0, x_1, \dots, x_n) are the components of the input vector x , while $(\omega_0, \omega_1, \dots, \omega_n)$ are the ones of the weight vector. Notice that in this model the x_0 input node is the “bias unit” whose corresponding weights ω_0 is always 1. The neuron works by confronting the input and weight vector via the scalar product

$z = \sum_{i=1}^n x_i \omega_i$ (that is the weighted sum where n is the number of connections to the perceptron) outputting the answer via the activation function $g = \frac{1}{1+e^{-\omega^T x}}$. The output is $y = g(z)$ (or $a_j^{(l)} = g(z)$ if it is not the final output layer, indicating the activation neuron unit j in layer l) generating an output decision of “0” or “1”.

The Perceptron learning process can be described as in the following steps:

1. Perceptron takes the input, multiplies them by the weights and compute their sum. The weights allow the perceptron to evaluate the relative importance of each input.
2. The bias factor is added. This procedure is needed in order to move the activation function and makes possible to fine-tune the numeric output of the perceptron.
3. The neuron feeds the previously computed sum through the activation function (in this case a logistic or sigmoid function) that maps the input values to the required output values. Activation function help the learning process especially in case of a multi-layer perceptron (MLP). The non-linear properties of the activation function make possible to train complex neural networks.
4. The perceptron output is a classification decision. In a feedforward neural networks the information always moves on one direction, from the input layer to the output. In

case of an MLP the output of one layer perceptron is the input of the next layer. The final layer is the output layer and its output is the final prediction.

As anticipated before a multilayer perceptron (MLP) is a structure made by many perceptrons, stacked in several layers where each layer takes input from the previous and then signals to the next layer. The diagram in *figure 1.12* shows an MLP with three layers. Each perceptron in the first layer on the left (the input layer), sends outputs to all the perceptrons in the second layer (the hidden layer), and all perceptrons in the second layer send outputs to the final layer on the right (the output layer).

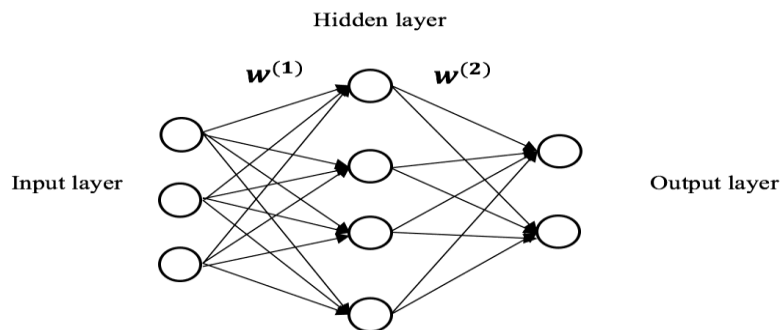


Figure 1.12: Network diagram of a two-layer neural network

Neural Networks algorithms learn by discovering better weights at every step that result in a more accurate prediction [7]. There are several algorithms for the fine tuning of the weights and the most common is the backpropagation that is a basic concept in modern neural network training and it belongs to supervised learning (meaning that the network is learning from pairs - input, expected output - of the training set). The Backpropagation training method for MLP is based on the gradient descent approach that iteratively adjust the network weights and gradually decreasing the error on the training set in an iterative procedure. To give a more in-depth description on backpropagation, it is necessary to fix the formalism and architecture of a neural networks:

- $a_j^{(l)}$ is the “activation” of the unit j in layer l
- $\Omega^{(l)}$ is the matrix of weights ω that controls function mapping from layer l to layer $l+1$

The value of each “activation” nodes is obtained for each unit j in a layer l as follows:

$$a_j^{(l+1)} = g(\Omega_{j0}^{(l)} x_0 + \Omega_{j1}^{(l)} x_1 + \dots + \Omega_{jn}^{(l)} x_n) = g\left(\sum_{i=0}^n \Omega_{ji}^{(l)} x_i\right) \quad (1.22)$$

- $j=1, \dots, n$ with n number on input features
- $l=1, \dots, L$ with L total number of layers in a network

For example, for the activation of the first node in the second layer:

$$a_1^{(2)} = g(\Omega_{10}^{(1)} x_0 + \Omega_{11}^{(1)} x_1 + \dots + \Omega_{1n}^{(1)} x_n)$$

Therefore, the computation of the activation nodes is done by using a

$s_{l+1} \times (s_l + 1)$ matrix of parameters (where s_l is the number of unit nodes in the layer l). This phase is called forward propagation in which all the activation value for every node in the network are computed. The hypothesis function output related to the output layer is the logistic function applied to the sum of the values of the activation nodes computed in the previous layer:

$$h_\Omega(x) = a_j^{(L)} = g(\Omega_{j0}^{(L-1)} x_0 + \Omega_{j1}^{(L-1)} x_1 + \dots + \Omega_{jn}^{(L-1)} x_n) \quad (1.23)$$

for each unit j in the output layer ($j_{out} > l$ in multiclass classification case)

The cost function $J(\Omega)$ for neural network is:

$$-\frac{1}{m} \sum_{i=0}^n \sum_{k=1}^m [y^{(i)} \log(h_\Omega(x^{(i)})_k) - (1 - y^{(i)}) \log(1 - h_\Omega(x^{(i)})_k)] + \frac{\lambda}{2} \sum_{l=1}^{L-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} (\Omega_{ji}^{(l)})^2 \quad (1.23)$$

The goal is to compute the minimum of this loss function that means finding an optimal set of parameters Ω that minimize the cost function $J(\Omega)$:

$$\min_{\Omega} J(\Omega) \quad (1.24)$$

Backpropagation is used in order to update the parameters Ω and this will bring - after a certain number of back and forward iteration - to the optimal set of parameters. In this backpropagation phase the term $\delta_j^{(l)}$ is computed, for each node and in each layer, and it represents the error of node j in layer l . In other words $\delta_j^{(l)}$ term captures the error in the activation $a_j^{(l)}$ value computed in the feedforward phase, and is computed starting from the last layer. For example with reference to Figure (1.12) starting from the third and last layer, for each node j the error can be expressed as: $\delta_j^{(3)} = a_j^{(1)} - y_j$ where y_j is the actual value observed in the training example. So, for the output layer (L):

$$\delta_j^{(L)} = a_j^{(L)} - y_j = h_{\Omega}(x)_j - y_j \quad (1.25)$$

And backpropagating the error in the previous layer:

$$\delta_j^{(L)} = (\Omega^{(L-1)})^T \delta^{(L)} \times g'(z^{(L-1)}) \quad (1.26)$$

Where $z^{(L-1)} = \Omega_{j0}^{(L-1)} x_0 + \Omega_{j1}^{(L-1)} x_1 + \dots + \Omega_{jn}^{(L-1)} x_n$ and g' is the first derivative of the activation function g .

And it is possible to prove, ignoring regularization term λ that:

$$\frac{\partial}{\partial \Omega_{ji}^{(l)}} J(\Omega) = a_j^{(l)} \delta_i^{(l+1)} \quad (1.26)$$

So, thanks to backpropagation the partial derivative needed for the minimization process of the cost function can be easily computed as show in the formula (1.26).

Therefore, by iteratively executing the forward and backward propagation phases the set of optimal parameters Ω minimizing the cost function is reached. Once the Artificial Neural Network is trained, it provides the “correct” predictions. It is also important to note that one of the most important hyperparameters for ANN is the learning rate α – mentioned in sub-Chapter (1.1.1) for the gradient descent technique – that specifies how fast the learning process is performed. The learning rate parameter value lies between 0 and 1 and its correct setting is crucial to the success of the learning process and finding the optimal values

1.2 Unsupervised Learning

The second major problem type is Unsupervised Learning (UL) that allows to find and infer the underlying structure of the data by clustering the data based on relationships among the variables in the dataset. In unsupervised learning unlabeled data are provided and the aim is to find the underlying causes and identify the intrinsic structure and hidden pattern behind the data. Since in unsupervised learning there is no prior knowledge on what the “clusters” look like, in contrast with supervised learning, there is no feedback based on the prediction results since labelled data are not provided. So, the training dataset is made up as follows: $\{x^{(i)}\}$ with $i = 1, \dots, m$ where there is no presence of the target variable $y^{(i)}$ (label). Therefore, unsupervised learning technique turns out to be very useful in all those cases where there is no clue of what the desired target output should be, such as – for example – the determination of a reference market for a new product that has to be put on the market by a company.

Some other applications of Unsupervised Machine Learning techniques are:

- **Clustering:** automatically split the dataset into groups according to similarity. However, can happen that cluster analysis overestimates the similarity between groups and does not treat data points as individuals. For this reason, cluster analysis is a poor choice for applications like customer segmentation and targeting.
- **Anomaly detection:** automatically discover unusual data points in the dataset. This is useful in pinpointing fraudulent transactions, discovering faulty pieces of hardware, or identifying an outlier caused by a human error during data entry.
- **Latent variable models:** are commonly used for data preprocessing, such as reducing the number of features in a dataset (dimensionality reduction and PCA- Principal Component Analysis).

In this chapter will be analyzed only the clustering methods and specifically only the K-means algorithm as an example of these kind of algorithms.

1.2.1 K-means Clustering

The fundamental idea behind Clustering is to group and categorize a set of objects into many subsets, called clusters, in such a way that all the items inside one subset are most "similar" to each other. The way to distinguish between "similar" and "dissimilar" items is done through the use of some metrics to calculate the similarity. Several metrics are present and the research about which among the different measurements is the best is one of the challenges. The most used metric is the Euclidean distance between points \mathbf{x}_1 and \mathbf{x}_2 , that basically is the length of the line segment connecting them. In Cartesian coordinates, for two points $\mathbf{x}_1 = (x_1^{(1)}, x_1^{(2)}, \dots, x_1^{(n)})$ and $\mathbf{x}_2 = (x_2^{(1)}, x_2^{(2)}, \dots, x_2^{(n)})$ their Euclidean distance is given by:

$$d(\mathbf{x}_1, \mathbf{x}_2) = \sqrt{\sum_{i=1}^n |x_1^i - x_2^i|^2} \quad (1.27)$$

where n is the dimension of a data point that is the number of features that characterize each data.

K-means is one of the simplest unsupervised learning algorithms that solve the clustering problem. The main idea of K-means is to define "K" number of centroids that represents the number of clusters. Each element in the data set is assigned to a cluster center (centroid) to which it is closest, for example assigning a given sample of the dataset to the centroid with respect to which it has the minimum Euclidean distance. Provided that there is a suitable distance definition, then the algorithm is composed of the following steps:

1. *Place K points into the space represented by the objects that are being clustered. These points represent initial group centroids*
2. *Assign each object to the group that has the closest centroid*
3. *When all objects have been assigned, recalculate the positions of the K centroids*
4. *Repeat Steps 2 and 3 until the centroids no longer move. This produces a separation of the objects into groups from which the metric to be minimized can be calculated.*

These steps are iterated until the algorithm converges to one optimum value (most likely a local optimum). The choice of the K value has to be taken in accordance to the number of classes in which the data must be clustered, as shown in *figure 1.13*.

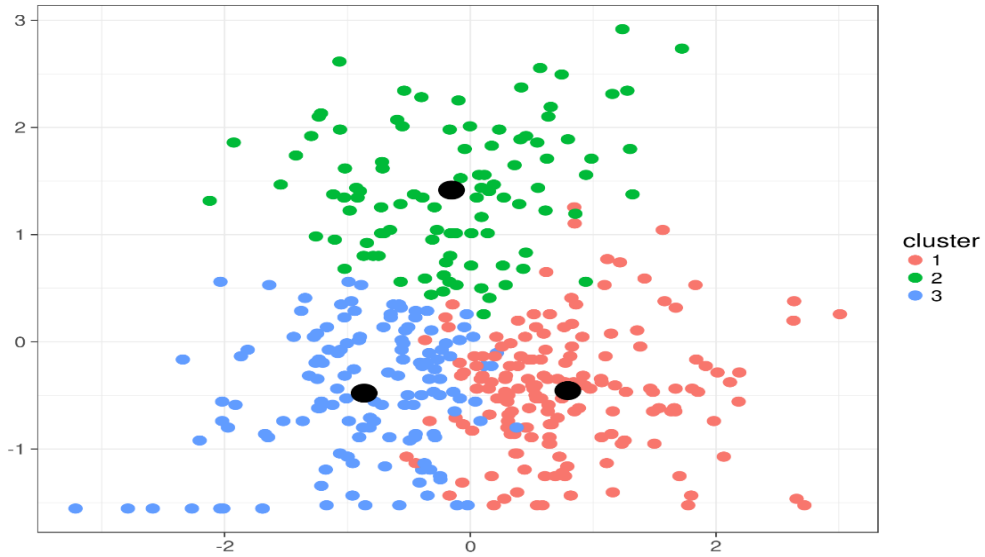


Figure 1.13: K-means clustering example with 3 centroids

Chapter 2

2 Reinforcement Learning

In order to ensure a deep comprehension of what Reinforcement Learning is, it is necessary to think how a human being learns to accomplish a task during his life. Indeed, since birth a child starts to understand how to behave by interacting with the environment and waiting for a response, so that after a given number of attempts, he is able to choose the correct action to achieve the desired goal. This trial-and-error logic is exactly the one on which the Reinforcement Learning is built, in fact it is the only paradigm of ML that reflects the human and animal way of learning that can lead robots and/or cars to act as a real person would do. It is already easy to notice that RL is very far from Supervised Learning (SL) since it has no need of supervisors and so of a label dataset; all the information and all the correct answers are collected from the interaction with the external world. At first glance it could be classified as an Unsupervised Learning (UL) problem, but the difference between the two stands in the fact that, as discussed in chapter one, UL refers to clustering problems while RL is used to solve decision-making problems.

It is possible to distinguish four fundamental elements needed to describe a Reinforcement Learning (RL) problem:

- *Reward*: it is a scalar feedback signal sent from the environment that can tell how good is doing the agent at step t . As claimed by Silver in his course, « all goals can be described by the maximization of expected cumulative reward » [8], in this sense in RL, feedback is delayed because an action can influence not only the next move but also the next n -moves and so the next collected rewards. Indeed, in many cases it is better to sacrifice a high reward at the current state rather than all the future states

rewards. Resuming the comparison with a human being, the reward is a biological stimulus that inflicts pain or pleasure based on the goodness/badness of the taken behavior.

- *Policy*: it represents the agent behavior and can be deterministic, $a = \pi(s)$ or stochastic, $\pi(a|s) = \mathbb{P}[\mathcal{A}_t = a' | \mathcal{S}_t = s']$. It is a map from state to actions.
- *Value function*: it returns the expected cumulative discounted (the meaning of “discounted” is going to be clarified in paragraph 2.1) reward at state s taking into account all the collected rewards during the long run, i.e. it gives an estimate of the total reward that the agent would collect by taking that specific action in the current state. In this sense, even if \mathcal{R} (reward) is the focal point in RL, a value function is what we are interested on, but unfortunately it is often very difficult to compute, so it is of paramount importance to choose the RL strategy that leads to the best estimate of the value function in each state.
- *Model*: it is the agent representation of the external environment but in the most challenging scenarios it is unknown.

The two main characters playing in a RL systems are the *agent* and the *environment*.

An *agent* has the aim of learning which is the best action to take in order to maximize the cumulative reward so, after receiving the immediate reward and observing the state inside the environment, it samples an action among all the possible ones giving much more importance to the future rather than the present. Of course, in order to pick the correct action at every time step, the agent must have in memory all the past states so that, when it reaches again one of them, it is sure of what move is the optimal one. Nevertheless, an agent cannot be stuck in this exploitation phase, because to find the best strategy it also has to do an exploration of the environment, by visiting unknown states. The same way of acting can be observed in people daily routines, for example when a driver wants to find the optimal path to reach a predefined location, he has to try all the different paths leading to that location and select the best one on the basis of his experience.

How to handle the balance between *exploration* and *exploitation* is a problem that mathematicians are still trying to solve, since it affects in a very deep manner the

convergence of a RL algorithm.

It is possible to classify three kinds of agents and their differences will be explained later:

- Value based or critic → the policy is correlated to the value function
- Policy based or actor → there is no value function
- Actor-Critic → there is either a policy and/or a value function

Even more in general, it can be distinguished between *model-based* agents, when there is a model representation of the environment, and *model-free* agents, if the dynamics of the environment is totally unknown.

The *environment* is everything that surrounds an agent and the world in which it has to navigate; for example, in a robot the agent can be seen as the brain of the robot itself and it will perceive as environment not only the external world but also the entire robot structure.

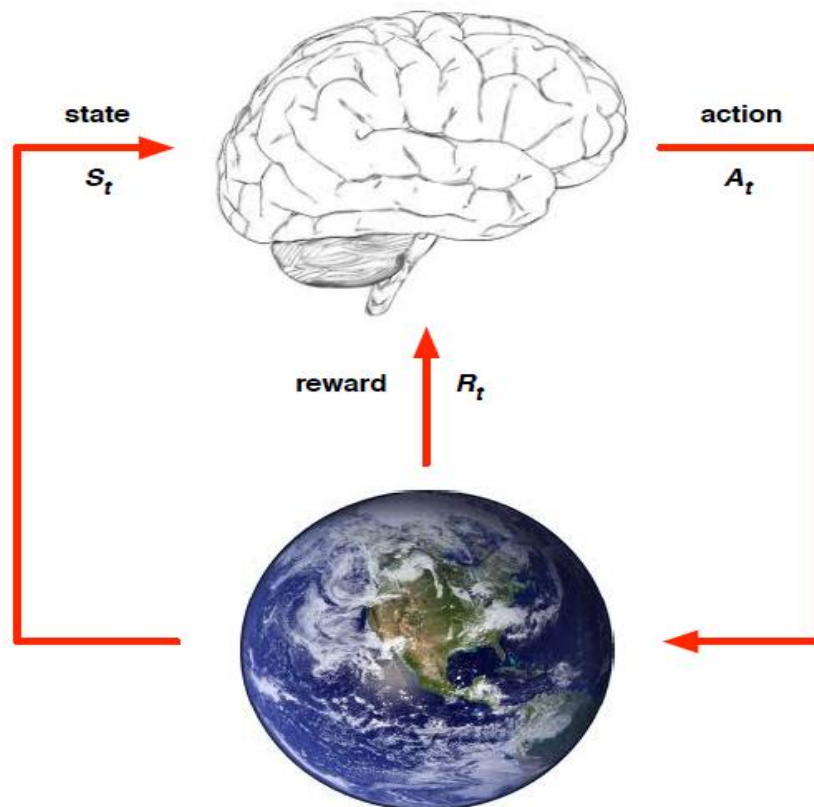


Figure 2.1: Schematic representation of the system/agent interaction

2.1 Markov Decision Process

After a general introduction of RL and its characterizing features, it has to be described from a purely mathematical side.

The classical formalism used to represent most of the RL problems and whatever decision-making problem is the Markov Decision Problem (MDP). In every MDP the learner is called agent and, like in RL, it has the job of collecting the highest reward by performing the right moves in each state. By referring to *figure 2.1*, at every discrete time step t_1, t_2, \dots, t_n the agent gets an observation of the environment s_t , takes action a_t and in the next time step it receives a scalar signal \mathcal{R}_{t+1} provided by the environment. It arrives in the new state s_{t+1} , giving birth to a trajectory that is called *history* of the MDP, i.e. the sequence of the observations, actions and rewards, $H_t = s_0, a_0, \mathcal{R}_1, s_1, a_1, \dots, \mathcal{R}_n, s_n, a_n$.

A main role is played by the state, a great source of information, that can be written as a function of the history $s_t = f(H_t)$ [8].

Anyway, care must be taken for not making confusion between the environment state S_t^e and the agent state S_t^a .

S_t^e is the exact environment representation, often invisible to the agent or containing a lot of useless information, whereas S_t^a is the internal agent representation of the environment and contains just the meaningful feature the agent uses to choose an action. Clearly, talking about Markov Process it is implicitly said that the environment state must be a Markovian state, and so it must synthesize all the relevant history information.

By definition [8], a state \mathcal{S}_t is called information state (or Markov state) if and only if

$$\mathbb{P}[\mathcal{S}_{t+1}|\mathcal{S}_t] = \mathbb{P}[\mathcal{S}_{t+1}|\mathcal{S}_1, \dots, \mathcal{S}_n] \quad , \quad (2.1)$$

(\mathbb{P} indicates a probability) meaning that the state \mathcal{S}_t is a sufficient statistic of the future since future does not depends on the past, given the present. When environment state and agent state coincide, the problem is called Markov Decision Process ($S_t^e = S_t^a = O_t$, where O_t is

the observation), while if the two states are different a Partially Observable Markov Decision Process (POMDP) is defined and the agent must construct its own representation of the state. RL problems deal only with full observable MDP.

From a mathematical point of view, it is called Markov Process (MP) a memoryless random process in which all the states are Markov states, and it is formalized as a tuple $\mathcal{M}=\langle \mathcal{S}, \mathcal{T} \rangle$ with

- $\mathcal{S} = \{s_1, \dots, s_n\}$ as the space of all possible states describing the environment
- $\mathcal{T}: \mathcal{S} \times \mathcal{S}' \rightarrow \mathbb{R}$ as the transition matrix, where each element defines the probability of moving to the state s' starting from state s . In this sense, \mathcal{T} contains the dynamic of the environment.

Another example of MP is the Markov Reward Process (MRP), a tuple $\mathcal{M}=\langle \mathcal{S}, \mathcal{T}, \mathcal{R}, \gamma \rangle$, with

- $\mathcal{R}: \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ as reward function, $r(s, a)$ as the immediate reward when action a is taken in state s
- $\gamma \in [0,1)$ as the discount factor used in the expected cumulative reward computation

in which the concept of return G_t must be introduced to justify the fundamental role played by the value function in a RL algorithm.

The return G_t can be whatever function that the agent tries to maximize during its training, for example the sum of all the collected reward inside an episode

$$G_t = \mathcal{R}_{t+1} + \mathcal{R}_{t+2} + \dots + \mathcal{R}_{t+T}, \quad (2.2)$$

where T is the final time step. Nevertheless, this return construction can work well only when the interaction agent-environment is split in experiments lasting for a finite number of time steps, which are called *episodes*. A task constituted by episodes is said episodic task. An *episode* always terminates in the same state, namely the terminal state, and no matter what happens in the end, it will always restart from the same initial state. The only outcome changing is the reward earned by the agent at terminal state on the basis of the chosen action,

as it happens when playing a videogame the match stops and you win or you lose depending on all the previous moves.

As T approaches to ∞ , the computation of G_t using (2.2) gets more and more heavier; this implies that for continuing tasks, i.e. tasks that last for an infinite time (the experiment is not divide into episodes), a new definition of the return G_t must be formalized

$$G_t = \mathcal{R}_{t+1} + \gamma\mathcal{R}_{t+2} + \gamma^2\mathcal{R}_{t+3} + \dots = \sum_{k=0}^{\infty} \gamma^k \mathcal{R}_{t+k+1}, \quad (2.3)$$

where the discount factor γ is a parameter used to tell how much confidence there is on future predictions. If γ tends to zero the agent is said to be ‘myopic’ and in case of $\gamma = 0$ the return collapses on the immediate reward, meaning that the action a_t does not affect the future decisions. On the contrary, if it approaches to 1 is said ‘far sighted’ since it is giving a strong importance to the future rewards. It seems trivial to notice that the new return shape increases the complexity of the problem from a purely conceptual side while simplifying the mathematical computation.

In order to adapt (2.3) to a reinforcement learning problem it makes sense to re-shape it by separating the immediate reward from the future time steps returns

$$\begin{aligned} G_t &= \mathcal{R}_{t+1} + \gamma\mathcal{R}_{t+2} + \gamma^2\mathcal{R}_{t+3} + \dots = \\ &= \mathcal{R}_{t+1} + \gamma(\mathcal{R}_{t+2} + \gamma\mathcal{R}_{t+3} + \dots) = \\ &= \mathcal{R}_{t+1} + \gamma G_{t+1} \end{aligned} \quad (2.4)$$

that holds for all $t < T$.

As it has been already mentioned, both (2.2) and (2.4) are needed since it has to deal with two different problems: episodic task and continuing task. For the first cited one, it has to be used a notation that make us able to discern among quantities related to different episodes, for example by adding a subscript as identifier of that particular episode ($\mathcal{R}_{t,i}$, $\mathcal{S}_{t,i}$).

However, since the reference is usually to a single episode this kind of notation

is not needed, but it is worth to write a single expression of the return that can be applied to both problems, to do so, the episodic task is schematized in *figure 2.2* [9]. States are drawn as circles while the terminal state is a square with the function of *absorbing state*, since once it has been reached, the agent remains stuck in that state for all the next time steps until a new episode starts, and the reward collected will be all equal to 0.

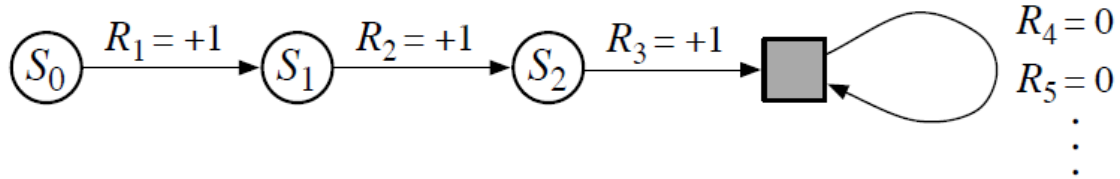


Figure 2.2: episodic tasks [9]

It is trivial to understand that, giving a reward equal to +1 to all the intermediate states, the return G_t will be 3 even if T is not finite and $\gamma = 1$.

Thanks to this example, a new formulation of the G_t can easily be derived,

$$G_t = \sum_{k=t+1}^T \gamma^{k-t-1} \mathcal{R}_k , \quad (2.5)$$

containing both parameters T and γ , paying attention to the fact that the two conditions $T = \infty$ and $\gamma = 1$ cannot be true at the same time.

At this point the Markov Decision Process (MDP) must be included [8], that is a tuple $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma \rangle$, with $\mathcal{A} = \{a_1, \dots, a_p\}$ as the space of all p actions that could be executed by the agent. Recalling the concept of policy, which tells the probability of choosing action a in state s_t , the state sequence $\mathcal{S}_1, \dots, \mathcal{S}_n$ is a MP $\langle \mathcal{S}, \mathcal{T}^\pi \rangle$ and the state-reward sequence $\mathcal{R}_1, \mathcal{S}_1, \mathcal{R}_2, \mathcal{S}_2, \dots$, is a MRP $\langle \mathcal{S}, \mathcal{T}^\pi, \mathcal{R}^\pi, \gamma \rangle$.

All these considerations opened up the road to the comprehension of the elements that characterize almost all RL algorithms, the *value function* and the *policy*, given that a value function is always defined in according to a particular policy.

By definition [9] «The value function of a state s under a policy π , denoted $v_\pi(s)$, is the expected return when starting in s and following π thereafter», in other words it says how good is for the agent to be in state s_t or more precisely how good is for the agent to execute a given action in that state.

Mathematically, considering (2.5), the value function is expressed as

$$v_\pi(s) = E_\pi[G_t | S_t = s] = E_\pi[\sum_{k=0}^{\infty} \gamma^k \mathcal{R}_{t+k+1} | S_t = s], \forall s \in \mathcal{S}, \quad (2.6)$$

that is an expected value of the returns computed starting from state s . In the terminal state the value function is always zero. Actually, $v_\pi(s)$ must be called state value function to avoid confusion with the state-action value function, which tells how convenient is for the agent to take action a in state s , under the policy π ,

$$q_\pi(s, a) = E_\pi[G_t | S_t = s, A_t = a] = E_\pi[\sum_{k=0}^{\infty} \gamma^k \mathcal{R}_{t+k+1} | S_t = s, A_t = a], \forall s \in \mathcal{S}. \quad (2.7)$$

As it has been done for the return G_t , in the same way a value function can be written as the sum of the immediate reward and the discounted value functions of the successor states,

$$\begin{aligned} v_\pi(s) &= E_\pi[\sum_{k=0}^{\infty} \gamma^k \mathcal{R}_{t+k+1} | S_t = s] = E_\pi[\sum_{k=0}^{\infty} \gamma^k \mathcal{R}_{t+k+1} | S_t = s] = \\ &= E_\pi[\mathcal{R}_{t+1} + \gamma \mathcal{R}_{t+2} + \gamma^2 \mathcal{R}_{t+3} + \dots | S_t = s] = E_\pi[\mathcal{R}_{t+1} + \gamma G_{t+1} | S_t = s] = \\ &= E_\pi[\mathcal{R}_{t+1} + \gamma v(S_{t+1}) | S_t = s], \end{aligned} \quad (2.8)$$

and this holds for all the states.

Now the *Bellman equation* can be introduced, showing how to compute a value function in every kind of RL or dynamic programming problems, which are going to be explained starting from paragraph 2.2.

In *figure 2.3* a tree-like scheme is reported [9], where the root represents the current state s , each of all the other circles indicates the successor state that the agent can occupy, according

to the action chosen through the policy π and the dynamic of the system, instead the black dots represents the state-action pairs.

By re-shaping (2.8) and by referring to *figure 2.3a*,

$$\begin{aligned}
 v_{\pi}(s) &= E_{\pi}[\mathcal{R}_{t+1} + \gamma G_{t+1} | S_t = s] = \\
 &= \sum_a \pi(a|s) \sum_{s', r} p(s', r|s, a) [r + \gamma E_{\pi}[G_{t+1} | S_{t+1} = s']] = \\
 &= \sum_a \pi(a|s) \sum_{s', r} p(s', r|s, a) [r + \gamma v_{\pi}(s')] . \tag{2.9}
 \end{aligned}$$

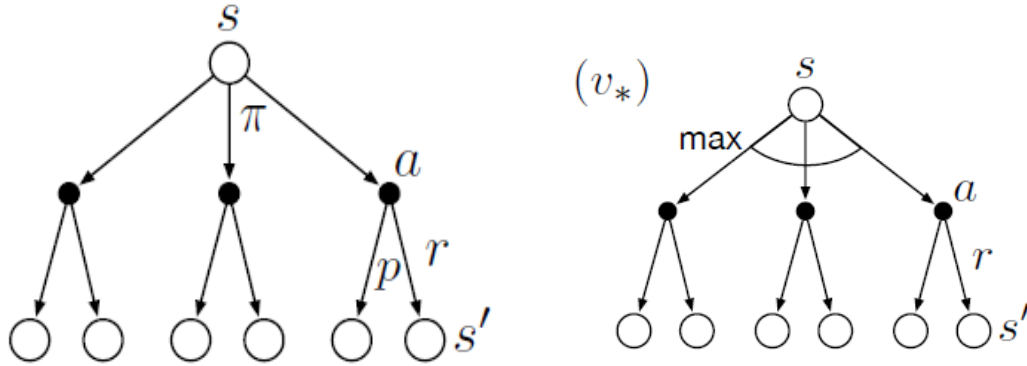


Figure 2.3: (a) bellman equation scheme, (b) bellman optimality equation scheme

Although it has been discussed about policy only in the singular, in a decision-making problem the number of policies that the agent could follow is potentially infinite. For this reason, the optimal policy must be found in order to accomplish the task in the best way possible.

By definition, the optimal policy, denoted by π_* , is the policy that leads to a value function $v_*(s) > v_{\pi}(s)$ for all $\pi \neq \pi_*$ and $\forall s \in \mathcal{S}$, with $v_*(s)$ as the optimal value function

$$v_*(s) = \max_{\pi} v_{\pi}(s) ; \tag{2.10}$$

simultaneously it must be also defined the optimal state-action value function $q_*(s, a)$ as

$$q_*(s, a) = \max_{\pi} q_{\pi}(s, a), \quad (2.11)$$

that can be also re-written as function of $v_*(s)$,

$$q_*(s, a) = E_{\pi}[\mathcal{R}_{t+1} + \gamma v_*(S_t) | S_t = s, A_t = a]. \quad (2.12)$$

Note that a deterministic optimal policy can be found for any MDP, and knowing the $q_*(s, a)$ the optimal policy can easily be deduced.

Being $v_*(s)$ a value function, it must satisfy the self-consistency condition expressed by the Bellman equation, leading to the formulation of the Bellman optimality equation,

$$\begin{aligned} v_*(s) &= \max_{a \in A} q_{\pi_*}(s, a) = \max_a E_{\pi} [G_{t+1} | S_t = s, A_t = a] = \\ &= \max_a E_{\pi} [\mathcal{R}_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a] = \\ &= \max_a \sum_{s', r} p(s', r | s, a) [r + \gamma v_*(s')] \end{aligned} \quad (2.13)$$

understandable thanks to the tree-like scheme in *figure 2.3b*.

Finally, there are all the mathematical tools needed to afford the first decision-making problem, that is called dynamic programming.

2.2 Dynamic Programming

The two terms Dynamic and Programming (DP) define a particular class of algorithms that can be used to solve decision-making problems in which the environment is modelled as a perfect MDP. In this perspective, they have not a strong impact in a practical sense, since most of the time it is not possible to have a perfect model of the environment. Nevertheless, it is worth to talk about DP, given that it is the theoretical basis on which all RL algorithms rely. As a matter of fact, in RL the aim is to find the optimal value function and state-action value function that satisfy the Bellman optimality equations and that lead to the optimal policy, as it happens in DP, but with a smaller computational complexity and neglecting the complete knowledge of the environment.

Usually DP deals with discrete problems where the environment is a finite MDP and the sets \mathcal{S} , \mathcal{A} , \mathcal{R} contain a finite number of states, actions and rewards respectively. For continuous systems DP ensures exact solutions only in a few cases, so it is convenient to discretize state, action and reward spaces to go back to the discrete case.

The great advantage of DP is that it is a very general solution method for problems characterized by sub-optimal substructure, e.g. the principle of optimality applies, and that can be divided into overlapping sub-problems, so once one of them is solved, the solution can be stored and later reused, since they recur many times. The same two proprieties are completely satisfied by a MDP.

To be specific, a DP algorithm is used to solve a problem in which the agent can do the computation and improve the policy with no kind of interactions, having the exact

representation of the external world. This problem is known as *Planning* and it can be divided in two parts:

1. **Prediction:** given a MDP or MRP and a policy π as input, the objective is to evaluate π , i.e. to demonstrate how good is the input policy.
2. **Control:** given a MDP as input, the aim is to find the optimal state value function $v_*(s)$ or the optimal state-action value function $q_*(s, a)$ and the optimal policy π_* .

The proposed solution for Prediction is the *Iterative Policy Evaluation*, an algorithm that applies and solves the Bellman expectation equation iteratively. Given a policy π the first idea is to use (2.9) for every state s , but if the number of states is denoted by S , this would mean to solve a system of S equations in S unknowns, requiring too much effort in the computation. A better way is to use an iterative process in which first a random value function $v_\pi(s)$ is set for each state, except for the terminal states that must have value function equal to zero, and then the Bellman equation is used to update $v_k(s)$ at each new time step $k+1$, knowing the successor states $v_k(s')$,

$$v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) [r + \gamma v_k(s')]. \quad (2.14)$$

Basically, (2.14) shows how for each state s , at time step $k+1$, the new value function is computed starting from the old value functions of the future states. It can be demonstrated that the algorithm goes to convergence, as k tends to ∞ .

The iterative policy iteration logic can be also explained graphically by *figure 2.4* [8].

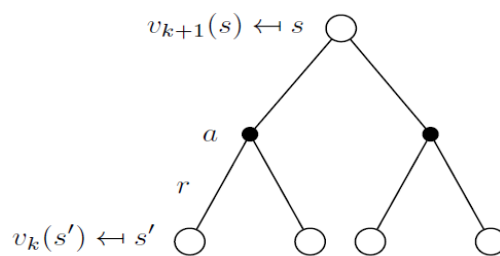


Figure 2.4: Iterative Policy Evaluation

In a control problem, starting from a deterministic policy, the optimal policy can be obtained only by following two steps: the policy evaluation needed to evaluate how good is π through the value function computation, and the policy improvement to enhance the policy by performing a greedy action on $v_\pi(s)$. The plan is to apply in state s an action a not selected by π so that $q_{\pi'}(s, a) > v_\pi(s)$, then $v_{\pi'}(s) > v_\pi(s)$ which means the new greedy policy is better than policy π .

π can be upgraded substituting $\pi(s)$ with $\pi'(s)$, where [9]

$$\begin{aligned}\pi'(s) &= \arg \max_a q_\pi(s, a) = \\ &= \arg \max_a E_\pi[\mathcal{R}_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a].\end{aligned}\quad (2.15)$$

When the condition $v_{\pi'}(s) = v_\pi(s)$ is achieved, it is sure that the new policy is as good as the previous one, $\pi' = \pi$ and by (2.15) it can be written

$$\begin{aligned}v_\pi(s) &= \max_a E_\pi[\mathcal{R}_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a] = \\ &= \max_a \sum_{s', r} p(s', r | s, a) [r + \gamma v_{\pi'}(s')].\end{aligned}\quad (2.16)$$

(2.16) is again the Bellman optimality equation, therefore it has been proved that π' is the optimal policy.

The *policy iteration* (PI) is an algorithm that applies iteratively the two steps in order to get π_* . As soon as the policy π is improved, the new policy π' is improved again to obtain a new

better policy π'' . This procedure perpetuates until the optimal policy π_* and the optimal value function v_* are generated, giving birth to the chain in *figure 2.5* [9].

$$\pi_0 \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} v_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \dots \xrightarrow{I} \pi_* \xrightarrow{E} v_*$$

Figure 2.5:

E stands for Policy Evaluation performed by value function computation and I stands for Policy Improvement

Actually, in many examples the number K of policy evaluation iterations, required to obtain the optimal policy, is smaller than the number of iterations needed to get the exact convergence, therefore the policy iteration algorithm can be improved by adding a stopping criteria to reduce k . Among all the possible choices it is smart to select just one iteration for PI, as in *value iteration algorithm*, by applying directly (2.13) like an update rule,

$$v_{k+1}(s) = \max_a \sum_{s',r} p(s',r|s,a) [r + \gamma v_k(s')]. \quad (2.17)$$

The cyclic nature of policy evaluation and policy improvement is denoted as *General policy iteration* (GPI) in the most general sense, characterizing not only DP, but also RL methods.

The two processes interact with each other (*figure 2.6* [9]), but at the same time they operate in opposite directions, indeed in policy improvement π must be consistent with respect to the value function, while in policy evaluation v_π must be coherent with the new greedy policy. The problem is that when the new greedy policy is obtained, the value function becomes inconsistent and when the value function is evaluated, the policy is no more greedy.

Nevertheless, both algorithms converge to the same outcomes, v_* and π_* , so that the policy is stable if the value function is stable, and vice versa.

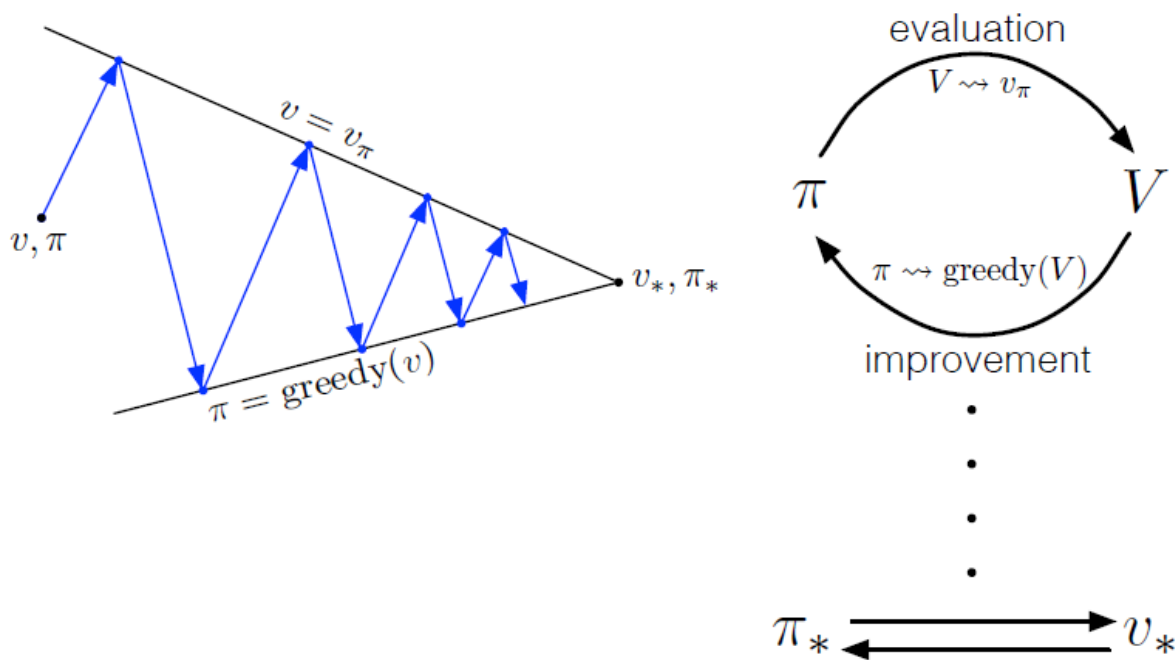


Figure 2.6: Generalized Policy Iteration

The discussed problems and all their relative DP solutions can be adequately summarized through the table 2.1, showed by Sutton and Barto in [9].

Problem	Bellman Equation	Solution
Prediction	Bellman expectation Equation	Iterative Policy Evaluation
Control	Bellman expectation equation + Greedy Policy Improvement	Policy Iteration
Control	Bellman optimality equation	Value Iteration

Table 2.1

2.3 Reinforcement Learning for prediction

In the previous section of this chapter it has been proved as DP could be a very powerful tool for the resolution of prediction and control problems, but its biggest limit lays in the fact that a fully knowledge of the environment is needed. Since most of the times it is impossible to acquire a model, e.g. the transition matrix, of the MDP, RL algorithms made their ways to provide a solution for the same two typologies of problem, but completely model-free, satisfying the information demand just through a real or simulated experience. The cornerstones of all RL algorithms are Monte Carlo (MC) and Temporal difference (TD) in which the concept of episode, intended as the sequence from starting state to terminal state, is crucial.

2.3.1 Monte Carlo for prediction

When MC is used for prediction, again the aim is to evaluate a given policy by computing the value function, also called utility function $U_{\pi}(s)$. In according to (2.6), the utility is expressed as the expectation of the returns, but the first peculiarity of MC methods is that it is computed as the empirical mean over the number of visited states. Each occurrence of a state during the episode is called **visit**. The concept of visit is important because it permits to categorize two different MC approaches:

1. **First-Visit MC:** $U_{\pi}(s)$ (from now on Patacchiola notation will be used) is defined as the average of the returns considering only the *first visit* to s , within a given episode.
2. **Every-Visit MC:** $U_{\pi}(s)$ is defined as the average of the returns considering all the *visits* to s , within a given episode.

Basically, in first-visit MC the utility relative to each state is updated just once for an episode, neglecting all the other visits to the same state during the current episode,

$$\begin{aligned}
 N(s) &\leftarrow N(s) + 1 \\
 S(s) &\leftarrow S(s) + G_t \\
 U(s) &\leftarrow \frac{S(s)}{N(s)} \quad \forall s \in \mathcal{S}, \quad (2.18)
 \end{aligned}$$

where $N(s)$ indicates the number of first-visit in state s during all the episodes, $S(s)$ is the sum of returns G_t computed at each first-visit, and $U(s)$ is the utility function in s .

In every-visit MC the utility $U(s)$ is updated at each visit of the same state during one episode, therefore the steps of the algorithm are the same of (2.18), but $N(s)$ can be updated more than once during one epoch.

Notice that MC is a good solution also for fully observable MDP, when DP would require too much effort in calculations.

Another way to perform the updates in MC algorithms is the usage of incremental mean,

$$\begin{aligned}
 N(s) &\leftarrow N(s) + 1 \\
 U(s) &\leftarrow U(s) + \frac{1}{N(s)} (G_t - U(s)),
 \end{aligned}$$

even if in non-stationary problem old states cannot be considered,

$$U(s) \leftarrow U(s) + \alpha(G_t - U(s)), \quad (2.19)$$

with α as a step-size parameter called learning rate, that gives the name to this method, the $\alpha - constant MC$.

The rule of large number proves that $U(s) \rightarrow U_{\pi}(s)$ as $N(s) \rightarrow \infty$ in both cases.

The greatest problem of MC is that it is necessary to wait for the ending of an episode to enable the agent to learn something and most of the times the experiments are very long, so delaying learning until the end can get too slow. Furthermore, the termination of the episode is not always guaranteed, in fact in MC there is no **bootstrapping**, meaning that the estimates for a state are independent on the other states. Temporal difference methods must be applied to have the possibility of performing an online update of the policy and consequently speeding up the process towards the convergence.

2.3.2 Temporal difference

Temporal Difference (TD) methods are constituted by a mixture of elements belonging to either MC and DP. Indeed, as in MC a TD method needs to learn from experience but like DP it has the bootstrapping propriety, so the estimate in state s depends on old states and it has not to wait for arriving at the terminal state to perform the updates.

The general rule of TD is expressed by [10]

$$new\ estimate \leftarrow new\ estimate + \alpha * [Target - oldestimate], \quad (2.20)$$

and the quantity $[Target - oldestimate]$ is called estimation error δ .

In (2.20) the parameter α (step size), usually chosen constant, gives an idea on how much the action value is going in the direction of the estimate, in fact if α is 0 the agent does not learn at all while if it is set to 1 the agent consider only the most recent information, neglecting the previous ones. By (2.19) it can be recognized that the MC target is equal to the return G_t ,

computed starting from state s until the terminal state. In other words, all the states from s to terminal state must be involved for the target estimate. In TD the basic idea is to perform all the utilities update every time a new state is visited, hence the ending of the episode is not required to compute the new $U(s)$.

In a sense, this makes TD a more powerful tool than MC, indeed it can be applied in continuous tasks as well (MC is applied just in episodic tasks) and the agent is able to learn online at every time step.

The TD target is expressed as

$$Target = \mathcal{R}_{t+1} + \gamma U(S_{t+1}) \quad (2.21)$$

giving the idea of the *bootstrap* concept as a guess from a guess.

(2.20) can be re-written by substituting (2.21),

$$U(s) \leftarrow U(s) + \alpha(\mathcal{R}_{t+1} + \gamma U(s_{t+1}) - U(s)), \quad (2.22)$$

obtaining the update rule for the simplest TD algorithm known as TD(0) or one-step TD, a special case of TD(λ) that will be pointed out later. By summarizing, TD mixes the bootstrapping of DP and the sampling of MC and as result it shows a low variance with some bias, in contrast with MC that has no bias but high variance. Despite of all these differences, the bound between them can be clarified by introducing n-step return theory.

2.3.3 n-step return, TD(λ) and eligibility trace

Until now only one-step TD has been analysed, where the target depends just on the estimate of the utility function at the next step, but nothing prevents us to look ahead for the n next steps to perform the update. In n -step TD, the target is computed as the n -step return,

$$G_t^n = \mathcal{R}_{t+1} + \gamma \mathcal{R}_{t+2} + \gamma^2 \mathcal{R}_{t+3} + \dots + \gamma^{n-1} V(s_{t+n}),$$

leading to a new update rule

$$U(s) \leftarrow U(s) + \alpha(G_t^n - U(s)). \quad (2.23)$$

After that, it is trivial to observe how for $n \rightarrow \infty$ the G_t^n return becomes the MC return and (2.23) is the same of (2.19) proving the link between the two methods (*figure 2.7*), already anticipated.

n -step return theory gives the chance to introduce the other TD method, denominated as TD(λ), where the λ parameter is a weight used to combine all the G_t^n returns giving as outcome the λ - *return*

$$G_t^\lambda = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_t^n.$$

Actually, G_t^λ represents the target in **forward-view TD(λ)**, since the utility is computed looking into the future, and as it happens in MC, it requires the ending of the episode.

The other type of TD(λ) is called **backward-view TD(λ)** and it exploits the concept of eligibility trace.

The idea of TD(λ) with eligibility trace is to update the utility function by considering not only the next step, but also the previous states visited during the current episode.

In general, the eligibility function is defined as follow:

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s) & \text{if } s \neq s_t \\ \gamma\lambda e_{t-1}(s) + 1 & \text{if } s = s_t \end{cases} \quad (2.24)$$

Employed to have a short memory of all the visited states until the terminal state. Here γ is the discount factor, a value between 0 and 1, and it is called **trace-decay**, while λ is the weight factor. With $0 < \lambda < 1$ the traces decrease in time by a factor $\gamma\lambda$ including all the previous state in the utility estimation, $\lambda = 0$ implies $e_t(s) = 1$ that brings back to the TD(0) case where only the next state is involved in the update, while for $\lambda = 1$ TD(1) is defined and all the preceding predictions are equally updated. Notice that the estimation error accumulated in TD(1) is exactly the MC error, then if the updates are performed off-line TD(1) coincides with MC.

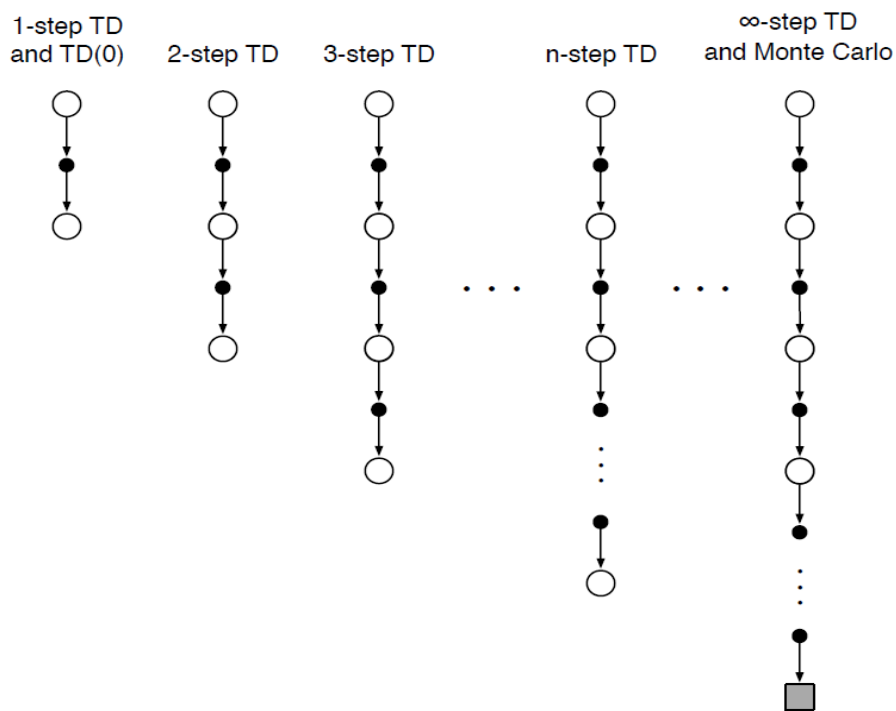


Figure 2.7: from one-step TD to MC [9]

The updated rule in backward TD(λ) is the following,

$$U(s) \leftarrow U(s) + \alpha \delta e_t(s), \quad (2.25)$$

with estimation error

$$\delta = \mathcal{R}_{t+1} + \gamma U(s_{t+1}) - U(s).$$

The advantage of TD(λ) over TD(0) is that the convergence is achieved faster.

All these concepts can be well resumed with table 2.2.

<i>Off-line updates</i>	$\lambda = 0$	$0 < \lambda < 1$	$\lambda = 1$
<i>Backward view</i>	TD(0)	TD(λ)	TD(1)
<i>Forward view</i>	TD(0)	Forward TD(λ)	MC
<i>On-line updates</i>	$\lambda = 0$	$0 < \lambda < 1$	$\lambda = 1$
<i>Backward view</i>	TD(0)	TD(λ)	TD(1)
		⊥	⊥
<i>Forward view</i>	TD(0)	Forward TD(λ)	MC

Table 2.2

2.4 Reinforcement Learning for control

It has been already disclosed that in control problems the aim is to find the optimal policy and RL takes over DP when the MDP is unknown or is too big to be used.

This implies that from now on, the concept of value function must be substituted with the state-action value function, which does not need the MDP to be defined, thus the policy improvement is performed [8],

$$\pi(s|a) = \mathit{arg} \max_{a \in \mathcal{A}} Q(s, a) . \quad (2.26)$$

All the algorithms defined in RL for control are based on the GPI concept.

Moreover, two kinds of learning have been recognized

1. On-policy learning: find optimal policy π_* from experience sampled by π
2. Off-policy learning: find optimal policy π_* from experience sampled by μ , another policy that is never updated.

2.4.1 Monte Carlo for control

As GPI claims, in control MC a part of the algorithms has to perform the policy evaluation process while the other part performs policy improvement. Here, the first problem shows up, since it is not sufficient to adopt a greedy policy improvement because of the trade-off between exploration and exploitation. To be sure that the provided policy is optimal,

exploration cannot be avoided since all the states of the environment must be visited, even if they do not return the highest immediate reward.

The only way to ensure exploration is to apply an $\epsilon - greedy$ policy improvement through the definition of a policy π that assigns probability $1 - \epsilon$ to a greedy action, and a probability ϵ to a non-greedy random action,

$$\pi(s|a) = \begin{cases} \frac{\epsilon}{m} + 1 - \epsilon & \text{if } a_* = \arg \max_{a \in \mathcal{A}} Q(s, a) \\ \frac{\epsilon}{m} & \text{otherwise} \end{cases} \quad (2.27)$$

with m equal to the number of possible actions.

This strategy is implemented in GLIE (**G**reedy in the **L**imit with **I**nfinite **E**xploration) Monte Carlo that by definition [8] converges to $Q(s, a) = q_*(s, a)$.

Within one episode, for every state-action pair (s_t, a_t)

$$\begin{aligned} N(s_t, a_t) &\leftarrow N(s_t, a_t) + 1 \\ Q(s_t, a_t) &\leftarrow Q(s_t, a_t) + \frac{1}{N(s_t, a_t)} (G_t - Q(s_t, a_t)), \end{aligned}$$

and the policy is improved with respect to the action-value function,

$$\begin{aligned} \epsilon &\leftarrow \frac{1}{k} \\ \pi &\leftarrow \epsilon\text{-greedy}(Q), \end{aligned}$$

k defines the k -th sampled episode.

However, MC has many disadvantages if compared with RL algorithms for control derived by TD.

2.4.2 Sarsa

To extend the TD methods to the control case (or active learning), since the new goal is to estimate the optimal policy starting by a random policy, as it has been done in MC for control, the action-value function must be invoked.

In TD control the estimation is based on the tuple **State-Action-Reward-State-Action** which gives the name to the algorithm, **SARSA**. For each state there is an associated action and now, state-action transitions take the state transitions place since Q receives as input state-action pairs (*figure 2.8*). The update rule is obtained by (2.22) just replacing U with Q,

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(\mathcal{R}_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)) . \quad (2.28)$$

In the standard implementation of SARSA the previous states are ignored, while moving forward, the algorithm takes into account only the current state and the state at time instant t+1. Thanks to GLIE assumption, SARSA convergence is guaranteed as long as exploration is implemented by visiting all the state-action pairs an infinite number of times.

The pseudo-code of this algorithm can be resumed in 4 steps [10]:

1. Perform an action provided by policy $\pi(s_t)$
2. Observe s_{t+1} , a_{t+1} and r_{t+1}
3. Update state-action value function $Q(s_t, a_t)$
4. Improve policy $\pi(s_t)$ as $\pi(s_t) = \mathop{\text{arg max}}_a Q(s_t, a_t)$

Here, from the practical point of view, Q is the state-action value function matrix that substitutes the utility matrix and contains the value of all the possible state-action pairs.

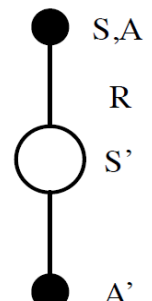


Figure 2.8:
SARSA

2.4.3 Q-learning

SARSA gives an idea of what on-policy learning means, since the agent learns about policy π through experience sampled by policy π itself. On the contrary, in off-policy learning (learning from observation) optimal policy is obtained by the observation of another policy μ that is never updated.

One of the biggest advantages that makes off-policy learning a very powerful tool, lies in the fact that it gives to the agent the ability of learning an optimal policy looking to the actions taken by another agent, for instance a robot, that follows a sub-optimal policy, or learning multiple policies. Furthermore, it opens the doors to the “experience replay” of deep learning strategies, where a part of information collected in the past is re-used to enhance the current policy.

Q-learning is probably the most important off-policy learning TD algorithm.

Its update rule is derived by (2.28),

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(\mathcal{R}_{t+1} + \gamma \max_a Q(s_t, a_t) - Q(s_t, a_t)), \quad (2.29)$$

where the only difference is the computed target.

Q-learning procedure is synthesized by 4 steps, as it happens in SARSA,

1. Perform an action provide by policy $\mu(s_t)$
2. Observe s_{t+1} and r_{t+1}
3. Update state-action value function $Q(s_t, a_t)$
4. Improve policy $\pi(s_t)$ as $\pi(s_t) = \arg \max_a Q(s_t, a_t)$

By step 1 and 4 it can be observed that, unlike SARSA, the updated policy is always $\pi(s_t)$, but actions are selected by a different policy, $\mu(s_t)$ and in step 2 the next action is no more taken in consideration. Leaving aside these aspects, the two algorithms are almost the same.

2.4.4 Actor-critic

Reinforcement learning and in general machine learning are strictly connected to neuroscience, by definition “*a branch of the life sciences that deals with the anatomy, physiology, biochemistry, or molecular biology of nerves and nervous tissue and especially with their relation to behavior and learning*”, so that a lot of algorithms are inspired from the idea of how the nervous systems works.

By undergoing different species of mammals to experiments in which a go/no-go task had to be accomplished, receiving a particular stimulus as input and a negative or positive reward as output, it has been found out that in this kind of situations the brain response of the animal is activated through the dopamine, a neuromodulator produced by neurons inside two different brain areas, substantia nigra pars compacta (SNpc) and ventral tegmental area (VTA). Dopamine is involved in some of the most important processes in both human and animal brains.

These two areas have direct projections to another area of the brain, the striatum. The striatum is divided in two parts, ventral striatum and dorsal striatum, whereas the output of the striatum is directed to motor areas and prefrontal cortex, and it is implicated in motor control and planning. It is possible to cluster all these brain sections in two different groups [10],

- **Group 1:** Ventral striatum, substantia nigra, ventral tegmental area
- **Group 2:** Dorsal striatum and motor areas

Group 1 can evaluate the relevance of a stimulus on the basis of the reward coming from the external environment. Meanwhile, it can estimate the error measure between the result of the action and its direct consequences, and use this value to calibrate an executor. Group 1 is

labelled as *critic*. Group 2 can only provide an action, independently on the utility of the stimulus, it is the *actor*.

All these concepts, very similar to the ones that has been studied and developed in RL, represent the invisible wire connecting neuroscience and ML, two science fields that just apparently have nothing in common.

Group 1 and group 2 help defining two macro sets of RL algorithms,

- Critic only, such as MC TD or DP based on the GPI mechanism → policy update and utility update are strictly connected,
- Actor only → no need to update a utility function and so the policy, such as Genetic algorithms in which different policies are followed and the best among them is chosen at the end of all the experiments.

The intersection between the sets gives the actor-critic algorithm (*figure 2.9*).

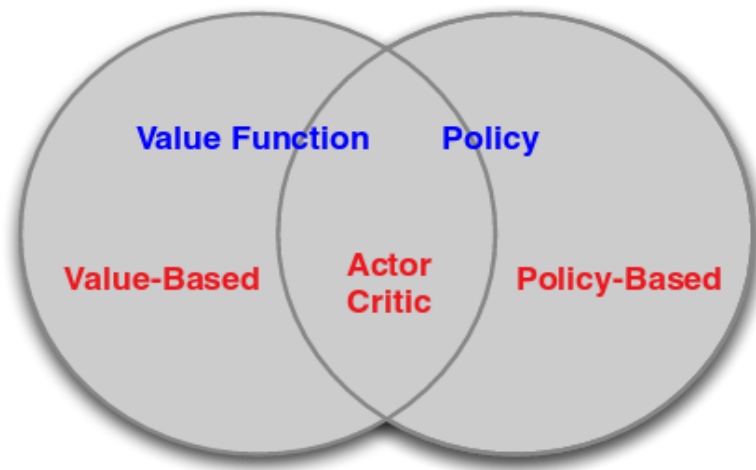


Figure 2.9: RL algorithms sets [8]

In actor-critic (figure 2.10) the utility is computed using a TD(0) as the *critic* and the action-value function is updated through the estimation error δ , a positive δ comes with a “good” action because it brings the utility towards a higher value, whereas a negative δ gives the opposite result and helps to define a “bad” action.

On the opposite, the action is chosen by the *actor* that can be realized in many ways, such as by a probability function (the softmax function, for example) or by an Artificial Neural Network (ANN) [9]. Anyway, there is no real update of a given policy.

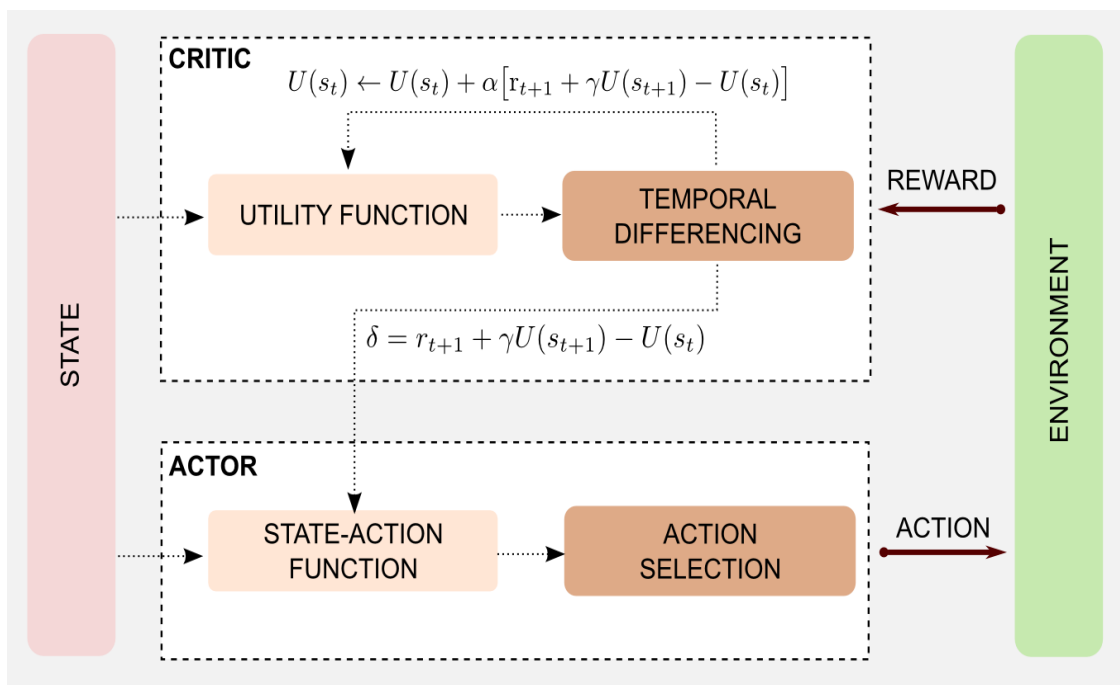


Figure 2.10: Actor-Critic scheme [10]

2.4 Function approximation

All the algorithms introduced so far constitute the simplest and the most popular methods used to solve RL problems, that generally are applied in small environments with reduced states and action spaces. In the implementation either the state value function or the state action value function can be built as look-up tables or matrices. More precisely $U(s)$ must be intended as a matrix of N elements, with N equal to the number of states within the environment, while $Q(s, a)$ is a $N * m$ matrix where m represents the number of possible actions for each state. The problem is that it is not possible to store all the values of $U(s)$ and $Q(s, a)$ when the laying MDP is too large (too many states and actions) or continuous. Therefore, it is necessary to approximate both through a function parametrized by a vector of weights $\omega \in \mathbb{R}^d$, such that $\hat{U}(s, \omega) \approx U_\pi(s)$ is the approximate value of state s , depending on ω . This is beneficial since the number of ω components is smaller than the number of states, $d \ll N$.

The two most accredited kinds of approximations are the linear function approximation, through which the function is written as a linear combination of the states, weighted by the components of vector ω , and the non-linear function approximation performed with an artificial neural network.

In both approaches, ω is the output of an iterative process, in which at each step the weights are updated on the basis of a target value and a cost function chosen properly.

Function approximation is an instance of supervised learning and this becomes clear underlining the presence of an update rule and an error measure.

As it has been pointed in Chapter 1, the most common way to define an error measure is the **MSE**. For this specific problem, it can be computed giving the optimal utility function $U_*(s)$ and its estimate $\hat{U}(s, \omega)$ and introducing function $\mu(s)$, with $\mu(s) \geq 0$, that tells how much relevance is attributed to state s .

The new loss function is denoted as **mean square value error** [10],

$$MSVE(\boldsymbol{\omega}) = \frac{1}{N} \sum_{s \in \mathcal{S}} \mu(s) [U_*(s) - \widehat{U}(s, \boldsymbol{\omega})]^2. \quad (2.30)$$

For what concern the update rule, gradient descent is applied, so by (1.5),

$$\boldsymbol{\omega}_{t+1} = \boldsymbol{\omega}_t - \frac{1}{2} \alpha \nabla_{\boldsymbol{\omega}} MSVE(\boldsymbol{\omega}). \quad (2.31)$$

The real issue of function approximation is that $U_*(s)$ is unknown. Nonetheless, the obstacle can be avoided by constructing an approximated utility function $\widetilde{U}(s)$ through MC or bootstrapping methods, and using it in (2.30) in place of $U_*(s)$.

2.5.1 Linear function approximation

In linear approximation the state is expressed as a vector of features $x(s_t)$, where the features are the most relevant information about that state and of course they change depending on the problem.

A linear approximator, that is the simplest case of linear combination, allows to write the utility function as a first order polynomial, so that the features are linearly combined through weights,

$$\begin{aligned}\hat{U}(s, \boldsymbol{\omega}) &= \boldsymbol{\omega}^T \mathbf{x}(s) \\ &= \omega_1 x_1 + \omega_2 x_2 + \dots + \omega_N x_N,\end{aligned}\tag{2.32}$$

with N equal to the number of features.

By substituting (2.31) in (2.30),

$$MSVE(\boldsymbol{\omega}) = \frac{1}{N} \sum_{s \in \mathcal{S}} \mu(s) [\tilde{U}(s) - \boldsymbol{\omega}^T \mathbf{x}(s)]^2,$$

and defining $\tilde{U}(s)$ as the TD(0) target, the update rule is

$$\boldsymbol{\omega}_{t+1} = \boldsymbol{\omega}_t - \alpha [r_{t+1} + \gamma \boldsymbol{\omega}^T \mathbf{x}(s_{t+1}) - \boldsymbol{\omega}^T \mathbf{x}(s)] \nabla_{\boldsymbol{\omega}} \hat{U}(s, \boldsymbol{\omega}),\tag{2.33}$$

where it is very easy to prove that the gradient is

$$\nabla_{\boldsymbol{\omega}} \hat{U}(s, \boldsymbol{\omega}) = x_1 + x_2 + \dots + x_N.$$

(2.33) can be re-written as follows,

$$\boldsymbol{\omega}_{t+1} = \boldsymbol{\omega}_t - \alpha [r_{t+1} + \gamma \boldsymbol{\omega}^T \mathbf{x}(s_{t+1}) - \boldsymbol{\omega}^T \mathbf{x}(s)] \mathbf{x}(s),\tag{2.34}$$

showing the final update rule in the linear function approximation case.

Though, a linear approximator is limited by the fact that it does not take into consideration the dependencies that might be among the features, hence more complex approximators must be involved. For instance, a quadratic approximator lets the possibility to model the utility as a second order polynomial,

$$\hat{U}(s, \boldsymbol{\omega}) = \omega_1 x_1 + \omega_2 x_1^2 + \omega_3 x_2 + \omega_4 x_2^2 \dots + \omega_{M-1} x_N + \omega_M x_N^2.$$

However, as it happens in supervised learning, attention must be taken in the bias-variance trade-off, since a too simple approximator might miss relevant information whereas approximators with a raised grade could not be general enough for the estimation of the utility in unvisited states.

2.5.1 Non-linear function approximation and Deep Q-network

Beside linear function approximation, non-linear function approximation is another instrument used in RL to estimate $U(s)$ and $Q(s, a)$, when a look-up table would require a too large space in memory.

The most powerful non-linear function approximator is a ANN, since its structure can be easily modified by adding multiple layers or by choosing different sigmoid functions. It follows that a ANN can shape any kind of utility from the simplest (it can coincide with the linear approximator) to the more complex one.

A practical example of non-linear function approximation in the RL domain is the deep Q-network (DQN) which overcomes the limits of Q-learning.

As suggested by the name, in DQN the goal is to build the $Q(s, a)$ as a Deep Neural Network (DNN), but the main difference with respect to supervised learning is that here the inputs of the net are not a i.i.d. (e.g. independent and identically distributed) set and during the training the target value $\hat{Q}(s, a; \boldsymbol{\omega})$ is continuously variable.

In addition, once the weights of the DNN are updated at the end of one iteration, the estimate of $Q(s, a)$ tries to move closer to the target, but it drags with it all the Q values in the neighborhood.

All these factors make the training unstable.

As a consequence, in order to stabilize the process, two concepts come into play, the *experience buffer replay* and the *target network*.

Experience buffer replay consists of storing in memory a given number of past experiences and then sampling randomly a small sized mini-batch of those experiences, that will be used as stable input for the DNN.

The target-network is a DNN used to create the target and make it constant for a definite number of iterations. Indeed, in a DQN algorithm there are always two DNN denoted by weights ω and ω_- , to create $\hat{Q}(s, a; \omega)$ and the target $\tilde{Q}(s, a; \omega_-)$ respectively.

At the beginning of the algorithm ω and ω_- are set equal with a random initialization and thereafter ω is continuously updated while ω_- is kept constant. Just after C iterations a reset is executed, and again $\omega_- = \omega$.

Here, the loss function that must be minimized is $\mathcal{L}_i(\omega_i)$,

$$\mathcal{L}_i(\omega_i) = E_{s,a,s',r \sim D} [(r + \gamma \max_{a'} Q(s'_t, a'_t; \omega_i^-) - \hat{Q}(s, a; \omega_i))^2], \quad (2.35)$$

where D is the dimension of the experience buffer.

Chapter 3

3 Inverse Reinforcement Learning

In this Chapter the problem of Inverse Reinforcement Learning (IRL) has been discussed, which is the inverse problem of the “forward” Reinforcement Learning (RL) examined in Chapter 2. Then, the causes that bring the need to face this type of problem will be explained, focusing on the importance of the reward function due to its great generalizability since it intrinsically defines the task to be performed. Moreover, different IRL methods and algorithms will be shown, highlighting the so called feature expectation based methods.

3.1 The Inverse Reinforcement Learning Problem

In MDP formalism and in Reinforcement learning process the implicit notion of “task” (or goal) is assumed to be encoded in a Markov decision process by means of the reward function. Therefore, assuming that the environment is equipped with a reward function, which is the feedback signal with which the agent actions are evaluated, then the reinforcement learning frameworks proposes a wide range of algorithms to estimate the optimal policy when the dynamics of the environment is unknown, as fully explained in Chapter 2. Indeed the MDP

formalism can be effectively applied in all the problems in which the reward can be specified more easily rather than directly indicate the optimal policy for a task. Unfortunately, many complex problems exist in which the design of a proper reward function is a difficult and time-consuming task, and this happens for example when dealing with unpredictable and dynamic scenarios or in not well-defined environments as it happens for autonomous driving. Indeed, for the driving task case, the human driver typically makes a trade-off between different objectives such as maintaining a desired speed and a certain safety distance from the leading vehicle while following correctly the lane and staying far from the pedestrians. So the reward function is frequently difficult to specify manually, in fact usually the starting point is to construct an hypothesized reward based on the information of the task to be performed that then is manually adjusted till the desired behavior is obtained.

Therefore, the entire field of RL is founded on the presupposition that the reward function is [11]:

“rather than the policy or the value function, is the most succinct, robust, and transferable definition of the task”

indeed, the reward function provides a description and the underlying logic of the task, that can be used then in RL and optimal control algorithms.

However, as it has been already said it is often an hard problem to define formally the task in a reward function form, in fact it is much more easier to define good performance on a specific task, for example using expert demonstration to extract a definition of the task.

In this regards RL community has provided several contributions to solve the problem of learning from demonstrations, giving rise to the Imitation Learning (IL) framework.

The main approaches solving IL problems are Behavioral Cloning (BC) [15] and **Inverse Reinforcement Learning (IRL)** [16]. BC recovers the demonstrated policy by learning the state-action mapping in a supervised learning way, while IRL aims to learn the reward function that makes the expert optimal. Behavioral cloning is a simple approach aiming at

imitate a movement from demonstration without learning a reward function, and it is usually achieved by learning a parametrized target policy or target trajectory using regression, but its main limitation is that it simply learns to replicate the observed policy. Moreover, BC method has several limitations: it requires a huge amount of data when the environment (or the expert) is stochastic [17]; it does not provide good generalization or a description of the expert goal. In IRL approach instead, the focus is on the direct learning and extraction of the unknown reward function for the purpose of generalization and transferability.

Indeed, the reward function can be transferred to any new scenario or environment in which the features are well defined. As a consequence, IRL allows recovering the optimal policy a posteriori, even under variations of the environment. It is crucial to stress that the reward function is a much more powerful and compact information rather than the optimal policy. The transition model can change over time, due to external factors, as a consequence also the optimal policy might change, whereas the reward will remain the same. Knowing the reward, the optimal policy associated to the new MDP can be recomputed. Thus, it can be understood why the reward function is a transferable information (it can be plugged into new problems allowing learning the optimal policy).

The problem of deriving a reward function from observed behavior, namely the Inverse Reinforcement Learning (IRL, Ng and Russell, 2000), seems one of the most promising method.

The purpose of IRL is to develop a method to recover the reward function that led to the observed sequences of choices made by an expert agent acting in a high-dimensional, stochastic environments. Then, utility functions found through IRL process can be used to simulate decision-making policies in the original scenario but also in unseen and new scenario in which the logic to be adopted is the same.

In the RL “forward” problem, the focus is in finding the optimal policy π^* given the MDP. In the inverse problem (IRL) the notation is $MDP \setminus R$, and the aim is to find this unknown reward, that has generated the policy π_E , through the data given by the expert demonstration μ_E .

This is equivalent to find the intent behind the demonstration rather than locally approximating the demonstrated trajectories as in behavioral cloning [15]. The main advantages of learning the reward functions, rather than imitating the optimal policy, is that the agent can solve the forward problem with the learned reward function not only to imitate but also to predict the expert motion, in the sense that in unknown scenarios the trained agent will behave like the expert managing to generalize the behavior learned even in cases that are different w.r.t the ones seen from the expert.

Among all IRL algorithms, there are the feature expectation-based methods which share a common structure presented as follows:

1. Parametrize linearly the reward function as: $R(s) = \omega^T \phi(s)$ (3.1)
2. Initialize ω randomly
3. Solve the forward problem (reinforcement learning) problem for the current reward parameters to get the current optimal policy π_i
4. Compute the difference between the expert policy π_E and the current optimal policy π_i
5. If the “error” is smaller than a threshold, the algorithms stops. Otherwise ω is update in a direction that reduces “error” then the algorithm iterates from step 3 onwards.

IRL assumes a parametric approximation of the reward function $R(\omega): \mathbb{R}^k \rightarrow \mathbb{R}$ with parameter values $\omega \in \mathbb{R}^k$. The set of states and action space are associated to with domain-specific feature vectors $\phi = \{\phi_{s,a}: S \times A \rightarrow \mathbb{R}\}$, where k is the number of features. For convenience the feature matrix Φ over state-action space with dimensions $|S||A| \times |k|$ is defined.

The agent in IRL solve an optimization problem in order to find parameters for U_θ that induce behavior in the environment that matches expected expert feature expectation, $\mu_E(\tau)$ computed as an average over m trajectories:

$$\mu_E(\tau^{(i)}) = \frac{1}{m} \sum_{\tau^{(i)} \in D} \sum_{(s_t, a_t) \in \tau^{(i)}} \Phi(s_t^{(i)}, a_t^{(i)}) \quad (3.2)$$

Where $D = \{\tau^{(i)}\}_{i=1:m}$ is the set of trajectories (training data) in which $\tau^{(i)}$ represent the i^{th} trajectory, composed by a sequence of state and actions made by an expert agent $\tau^{(i)} = ((s_0^{(i)}, a_0^{(i)}), (s_1^{(i)}, a_1^{(i)}), \dots, (s_{N-1}^{(i)}, a_{N-1}^{(i)}))$.

The structure presented helps to better understand the algorithms that will be shown in this Chapter such as Max-Margin, Projection based and Maximum Entropy IRL. In paragraph 3.2 the IRL Max Margin method proposed by Pieter Abbeel and Andrew Y. Ng [11] will be presented, and key concept such as feature expectation, policy mixing and the shortcomings of this method are discussed. Then in section 3.2.1 a simplification of the Max Margin method, called Projection-Based method, is shown. In conclusion in paragraph 3.3 is introduced the Maximum Entropy IRL which attempts to fix the problem raised in the Max Margin.

3.2 Max-Margin IRL

The Inverse Reinforcement Learning (IRL) is based on the same mathematical formalism discussed in Chapter 2 when examined the “forward” RL, but in the IRL case a complete Markov Decision Process (MDP) is not provided, since the Reward function is unknown. We denote MDP \setminus R and MDP without a reward function.

In [12] it is assumed some vector of features $\phi : S \rightarrow [0,1]^k$ over states, and that there is some “true” reward $R^*(s) = \omega^{*T} \phi(s)$, where $\omega^* \in \mathbb{R}^k$ with k number of features. It is also assumed $\|\omega^*\|_1 \leq 1$ in order to have the rewards bounded by 1. In the driving domain as will be explained in more detail in Chapter 5, $\phi(s)$ is a vector of features indicating the key variable for the specific ADAS behavior implemented.

A policy maps states over actions and the value function of a policy π is

$$E_{s_0}[V^\pi(s_0)] = E[\sum_{t=0}^{\infty} \gamma^t R(s_t) | \pi] \quad (3.3)$$

$$= E[\sum_{t=0}^{\infty} \gamma^t \omega^T \phi(s) | \pi] \quad (3.4)$$

$$= \omega^T E\left[\sum_{t=0}^{\infty} \gamma^t \phi(s) | \pi\right] \quad (3.5)$$

The expected discounted accumulated feature value vector $\mu(\pi)$ called also feature expectations is defined as:

$$\mu(\pi) = E\left[\sum_{t=0}^{\infty} \gamma^t \phi(s) | \pi\right] \in \mathbb{R}^k. \quad (3.6)$$

Making the assumption that the reward R is expressible as a linear combination of the features $\phi(s)$, the feature expectation previously defined basically is the expected sum of the discounted rewards following the policy π .

Is assumed to have access to demonstrations by some expert agent π_E , which can be thought as the optimal policy under the unknown “true” reward $R^*(s) = \omega^{*T} \phi(s)$. The algorithm has to estimate the expert’s feature expectations $\mu_E = \mu(\pi_E)$ and more specifically given a set of m trajectories $\{\tau_0^{(i)}, \tau_1^{(i)}, \dots\}_{i=1:m}$ generated by the expert.

Is it possible to defined now the empirical estimate for μ_E by

$$\hat{\mu}_E = \frac{1}{m} \sum_{i=1}^m \sum_{t=0}^{\infty} \gamma^t \phi(s_t^{(i)}). \quad (3.7)$$

So the MDP\(R) problem is considered in which given a feature mapping ϕ and the expert’s feature expectation μ_E the aim is to find a policy whose performance is close to the expert policy on the unknown reward function $R^*(s) = \omega^{*T} \phi(s)$. In order to do this, the policy $\tilde{\pi}$ has to be found such $\|\mu(\tilde{\pi}) - \mu_E\|_2 \leq \varepsilon$ and for such a $\tilde{\pi}$ we have that for any $\omega \in \mathbb{R}^k$ with $\|\omega\|_1 \leq 1$:

$$|E[\sum_{t=0}^{\infty} \gamma^t R(s_t) | \pi_E] - E[\sum_{t=0}^{\infty} \gamma^t R(s_t) | \tilde{\pi}]| \quad (3.8)$$

$$= |\omega^T \mu(\tilde{\pi}) - \omega^T \mu_E|$$

$$\leq \|\omega\|_2 \|\mu(\tilde{\pi}) - \mu_E\|_2$$

$$\leq 1 \cdot \varepsilon = \varepsilon \quad (3.9)$$

Notice that the first inequality follows from the fact that $|x^T y| \leq \|x\|_2 \|y\|_2$, and the second simply from $\|\omega\|_2 < \|\omega\|_1 \leq 1$. Therefore, the problem is simply reduced to find a policy $\mu(\tilde{\pi})$ close to μ_E . So, the Abbeel and Andrew Y. Ng [11] proposed ‘‘Apprenticeship Algorithm’’ is the following:

1. A policy $\pi^{(0)}$ is randomly picked and compute (or approximate via Monte Carlo) $\mu_0 = \mu(\pi^{(0)})$, set $i=1$.
2. Computing a new ‘‘guess’’ of the reward function by solving the following convex quadratic programming problem:

$$\begin{aligned} \min_{\lambda, \mu} & \|\mu_E - \mu\|_2 & (3.10) \\ \text{s.t.} & \sum_{j=0}^{i-1} \lambda_j \mu^{(j)} = \mu \\ & \lambda \geq 0 \\ & \sum_{j=0}^{i-1} \lambda_j = 1 \end{aligned}$$

$$\text{Set } t^{(i)} = \|\mu_E - \mu\|_2, \omega^{(i)} = \frac{\mu_E - \mu}{\|\mu_E - \mu\|_2}.$$

3. If $t^{(i)} \leq \varepsilon$, then terminate.
4. Using an RL algorithm in order to compute the optimal policy $\pi^{(i)}$ for the MDP using the rewards $R = (\omega^{(i)})^T \phi$.
5. Compute (or estimate) $\mu_i = \mu(\pi^{(i)})$
6. Set $i=i+1$, and go back to step 2.

When the algorithm terminates the policy $\tilde{\pi}$ is obtained as a mixture of the policies found at each iteration $\{\pi^{(i)}: i = 0 \dots n\}$ (n number of iteration) combined with weights λ_i that attains

features count μ , which are within ε from the expert's feature counts μ_E . Indeed it is proven in [11] that given a set of policy $\pi_1, \pi_2, \dots, \pi_d$ a new policy can be found whose feature expectation vector is a convex combination of these policy thus creating a policy mixture: $\sum_1^n \lambda_i \mu(\pi_i)$ in which $\lambda_i \geq 0, \sum_i \lambda_i = 1$. Therefore, the initial set of policy can be mixed and the probability of picking π_i is λ_i . So fundamentally the algorithm is composed by two main phases: solving a convex optimization problem that provides the expected features closest to the expert's policy in reward feature space amongst features counts achievable by using a mixture policy of the previously found policies $\pi^{(0)}, \dots, \pi^{(i-1)}$ and setting the guessed reward weights $\omega^{(i)}$. Then the new updated reward function $R = (\omega^{(i)})^T \phi$ is then used in a RL phase for the computation of the successive optimal policy $\pi^{(i)}$.

It is shown in [12] that the same weights ω can be computed by solving the following quadratic programming problem:

$$\max_{t, \omega} t \quad (3.11)$$

$$\text{s.t.} \quad j = 1, \dots, i - 1 \quad (3.12)$$

$$\omega^T \mu_E \geq \omega^T \mu^{(i)} + t$$

$$\|\omega\|_2 \leq 1 \quad (3.13)$$

From the constrain equation (2.9) it is evident that the algorithm is trying to find a reward function $R = (\omega^{(i)})^T \phi$ on which the expert does better, by a "margin" of t , than any of the $\pi^{(i)}$ policies found previously. As can be noticed the equations (3.10), (3.11) and (3.12) are equivalent to the SVM problem in which the aim is to find the maximum margin hyperplane separating two set of sample points [12]. Due to this, this problem is called **Max Margin IRL**. The equivalence is obtained by associating label 1 with the expert's feature expectation μ_E , and label -1 with the feature expectations μ_i computed at each step.

Following the parallelism with SVM the vector $\omega^{(i)}$ is the unit vector orthogonal to the maximum margin separating hyperplanes.

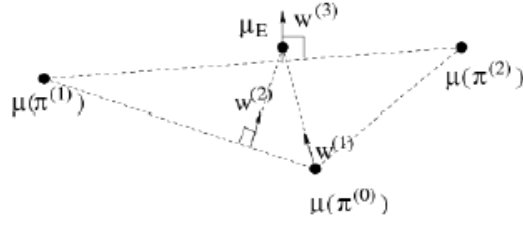


Figure 3.1: Graphic representing the Initial iteration of the max margin method

So, besides a generic Quadratic Program (QP) solver, an SVM solver can be used in order to find $\omega^{(i)}$. In *figure 3.1* a geometrical intuition is given of what the first iterations of the algorithm looks like, showing the first three $\mu(\pi_i)$ computed and the relative $\omega^{(i)}$.

Now, suppose the algorithm terminates, with $t^{(n+1)} \leq \varepsilon$, then following the optimization formulation in Step 2 of the algorithm:

$$\forall \omega \text{ with } \|\omega\|_2 \leq 1 \exists i \text{ s.t. } \omega^T \mu^{(i)} \geq \omega^T \mu_E + \varepsilon$$

Since $\|\omega^*\|_2 < \|\omega^*\|_1 \leq 1$, this means that there is at least one policy from the set returned by the algorithm, whose performance under R^* is at least as good as the expert's performance minus ε . Thus, it is up to the agent designer to manually examine the policies found by the algorithm in order to pick the one with “acceptable” performance in the sense of desired behavior.

On the other hand, the building of the policy $\tilde{\pi}$ can be automatized just solving the QP in step 2 (Formula 3.8) of the original algorithm presented, with no need for human help. In this latter case the policy $\tilde{\pi}$ is obtained as a mixture of the policies $\{\pi^{(i)} : i = 0 \dots n\}$ with weights λ_i taking into account the feature expectations which are within ε of the expert's feature expectations μ_E .

Note that the algorithm presented here does not necessarily recover the underlying reward function correctly. The performance guarantees of the algorithm only depend on (approximately) matching the feature expectations, not on recovering the true underlying reward function [12].

The most important thing to note in SVM IRL is that we do not, in fact, learn the underlying reward function but rather the **policies** that allows us to match the **feature expectation** of the expert probabilistically. Moreover, there exist many policies that give the same return. It is not clear which among these policies should be preferred since we can only access the feature expectation of the expert through the structure of the IRL problem and not the exact trajectory itself [13]. Unfortunately, this IRL concept proposed by [12] suffers from this ambiguity linked to the fact that it is based on the match of the feature expectation. Indeed, each policy can be optimal for many reward functions and many policies lead to the same feature expectation. When sub-optimal behavior is demonstrated, that means if the expert agent does not show an optimal behavior but a sub-optimal one, then a mixture of policies is required to match the desired expert feature expectation. Therefore, similarly to what was said for policy and reward correspondence, many different mixtures of policies satisfy feature matching. So, in summary, while [11] shows that matching feature expectations is both necessary and sufficient in order to derive policies that emulate expert behavior in the MDP, the IRL problem of recovering reward functions under this constraint is underdetermined. That is, optimal policies where the utility function is all zeros may be recovered. Maximum entropy IRL was developed by [14] to address this problem.

3.2.1 Projection based method

The algorithm described in the previous section requires to solve in step 2 a Quadratic Programming (QP) problem in the case of the original posed algorithm (Formula 3.9) or a SVM problem in the case described in the equations (3.10), (3.11) and (3.12). It is possible to simplify the algorithm so that no QP solver is needed. Therefore, it is possible to replace the Max-Margin IRL method described in Section 3.2, with a new algorithm: the **Projection method**.

With the Projection method at each step 2 no QP or SVM solver is needed, but the weights are updated just using a geometrical approach to the problem. So the maximization problem (3.9) in the Max-margin case is replaced, and now at each iteration i the orthogonal projection

$\bar{\mu}$ of $\mu(\pi_E)$ on the line passing through the last two $\mu(\pi_i)$ found previously is computed instead. Then the distance $\mu(\pi_E) - \bar{\mu}_{(i-1)}$ is taken as the new weights ω . So, the Projection method replaces step 2 of the algorithm with the following equation:

- Set $\tilde{\mu} = \bar{\mu}^{(i-1)} = \bar{\mu}^{(i-2)} + \frac{(\mu^{(i-1)} - \bar{\mu}^{(i-2)})^T (\mu_E - \bar{\mu}^{(i-2)})}{(\mu^{(i-1)} - \bar{\mu}^{(i-2)})^T (\mu^{(i-1)} - \bar{\mu}^{(i-2)})} (\mu^{(i-1)} - \bar{\mu}^{(i-2)})$ (3.14)
- Set $\omega^{(i)} = \mu_E - \bar{\mu}_{(i-1)}$
- Set $t' = \|\mu_E - \bar{\mu}^{(i-1)}\|_2$

Notice that equation (3.14) computes the orthogonal projection of μ_E onto the line through $\bar{\mu}^{(i-2)}$ and $\mu^{(i-1)}$. In the first iteration, we also set $\omega^{(1)} = \mu_E - \mu^{(0)}$ and $\bar{\mu}^{(0)} = \mu^{(0)}$.

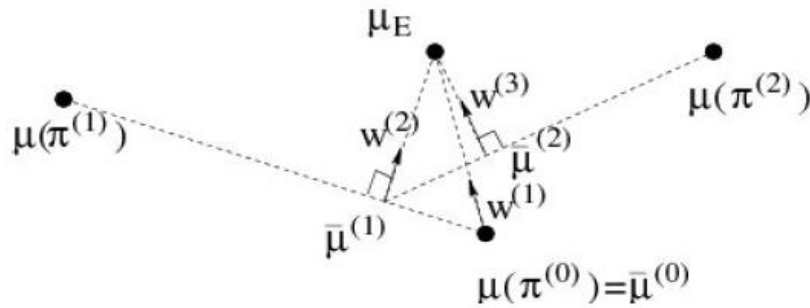


Figure 3.2: First three iterations of the projection version of the algorithm

Convergence results for both the Max margin and the Projection methods are provided in [12]. This method shows the same criticalities of the original Max-Margin method discussed in the end of the previous section. A solution can be found in the Maximum Entropy IRL method.

3.3 Maximum Entropy IRL

As we have already pointed out, the IRL problem is in general ill-posed, since there exist an infinite number of reward functions for the same MDP that make the expert's policy optimal. The [14] shows how to solve this ambiguity by leveraging on the maximum entropy principle [18]. The maximum entropy principle applies to a constrained probability estimation problem from data (like estimating the expert's trajectory distribution subject to matching expert's feature expectations). It states that the probability distribution that best represents the available data is the one with the largest entropy. In other words, with no constraint a uniform distribution can be selected, that is the distribution with maximum possible entropy, whereas under some constraints it can be proved [18] that it is necessary to resort to the maximum likelihood Boltzmann distribution. Like [11], it is assumed that the true reward function can be expressed as a linear combination of state features $\phi(s)$. Given a trajectory $\tau \in D$ the feature expectation [13] can be define as in formula (3.1):

$$\mu(\tau) = \sum_{t=0}^{\infty} \gamma^t \phi(s) \quad \forall \tau \in D \quad (3.15)$$

The return of trajectory τ can be computed, given a vector of the reward weights ω , as $R(\tau) = \omega^T \mu(\tau)$. Given a set of N expert's trajectories the aim is to estimate the probability distribution of the demonstrated trajectories, named p_ω constrained to the fact that the feature expectation matches the average feature expectation:

$$E_{\tau \sim p_\omega} [\mu(s)] = \bar{\mu}, \quad (3.16)$$

where $\bar{\mu} = \frac{1}{N} \sum_{t=0}^N \mu(\tau)$. The premises made so far are identical to those made for both the two algorithms previously seen, but now according to the maximum entropy principle [14] this is equivalent to finding the maximum likelihood Boltzmann distribution from the sampled trajectories that matches the average feature expectation $\bar{\mu}$:

$$p_{\omega}(\tau^{(i)}) = \frac{e^{\omega^T \mu(\tau^{(i)})}}{\sum_{j=1}^N e^{\omega^T \mu(\tau^{(j)})}}, \quad i=1,2,\dots, N \quad (3.17)$$

The equation (3.17) tells that the trajectories with the same return have the same probability while trajectories with larger return are exponentially preferred. One important benefit of this probabilistic approach is that we implicitly handle the uncertainty and noise in the observed trajectories, potentially leading to obtaining clearer or more robust reward functions. The equation (3.17) holds only for deterministic MDPs. In case of stochastic environments there is the need of more complex distribution and the whole treatment becomes much more complex and is reported in [13],[14] and [19].

So, following the equation (3.17) the paths with higher returns are exponentially preferred to those with lower total rewards. As described in [14] the maximum entropy objective function maximizes the likelihood of demonstrations according to:

$$\omega^* = \underset{\omega}{\operatorname{argmax}} \sum_{\tau^{(i)} \in D} \log p_{\omega}(\tau^{(i)}) \quad (3.18)$$

The maximum likelihood can be found using gradient descent, where the gradient of the likelihood function $L(\omega) = \log p_{\omega}(\tau^{(i)})$ (this is the objective function in this case) is given by the difference between the expert feature expectation μ_E and the feature expectation for the current estimate of the reward function [13] (please note: the reward function is linear in the features):

$$\nabla L(\omega) = \mu_E - \sum_{s \in S} \sum_{a \in A} E[\mu(s)] \pi_{\omega}(a | s) \phi(s, a) = \mu_E - \mu(\pi_{\text{Max-Entropy}}) \quad (3.19)$$

As seen and demonstrated in [13] and [14] the criticalities raised in the Max-Margin and Projection based IRL are solved. Indeed, the Max-Entropy IRL essentially initializes the parameters ω randomly and uses gradient descent to maximize the objective $L(\omega)$. Intuitively, maximizing the likelihood in equation (3.18) means that the goal is to find the parameters ω of the reward function which maximizes the probability of seeing the demonstrations D while minimizing the probability of observing any other trajectory.

Since the Reward $R(s) = \omega^T \phi(s)$ is linear the gradient in equation (3.19) can be interpreted as the difference between the expert feature expectation μ_E and that of the current maximum entropy policy $\mu(\pi_{Max-Entropy})$. Indeed, it can be noticed that the objective function $L(\omega)$ is just the inner product of the parameters ω and $\mu_E - \mu(\pi_{Max-Entropy})$:

$$L(\omega) = \omega^T (\mu_E - \mu(\pi_{Max-Entropy})).$$

Clearly this objective function is increased when the length of projection of ω along the difference $\mu_E - \mu(\pi_{Max-Entropy})$ is increased. In Projection based methods (section 3.2.1), it is used a new weight equal to this difference, setting $\omega^{(i)} = \mu_E - \bar{\mu}_{(i-1)}$ at each step, and try to reach the expert's feature expectation μ_E by mixing all the policies. In Maximum Entropy IRL, the difference to the current weight is added to push it towards the missing components in order to get closer to the expert's feature expectations [13],[14].

Chapter 4

4 ADAS and vehicle dynamics

4.1 Introduction to ADAS

Studies in the automotive fields proved that the majority of car accidents must be attributed to human driver misjudgments or distractions. On the basis of these results engineers introduced systems, that have been called Driving Assistance Systems (DAS), which aim is to improve vehicle safety assisting the driver during some critical events that can happen while driving.

Among the most important DAS that nowadays can be found on any car, there are the Anti-lock Braking System (ABS), the Electronic Stability Control (ESC) and Traction Control System (TCS). DAS are strictly linked to the use of proprioceptive sensors, which are able to measure most of the vehicle quantities, such as velocity, acceleration, wheel rotational speed and so on.


Thereafter, in order to enhance even more the level of assistance given to the human driver, *exteroceptive* sensors, i.e. sensors that acquire measurements from the external environment (road, other vehicles, traffic etc.), were introduced in a new kind of systems, the Advanced Driver Assistant Systems (ADAS).

ADAS represent an evolution of DAS since they can help the human driver in performing either some critical tasks, such as an emergency braking, or the most common tasks, for instance maintaining a constant speed, parking and so on.

Indeed, the most known ADAS are:

- Adaptive Cruise Control (ACC),
- Lane keeping Control (LKC),
- Park Assist (PA),
- Autonomous Emergency Braking (AEB).

ADAS constitute the first step towards the autonomous driving, in fact according to the SAE (Society of Automotive Engineers) standard (*table 4.1*), each one of these systems is an example of Level 1 automation, called “driver assistance”. In the case in which two or more ADAS work simultaneously on the car, the level 2 “partial automation” is achieved. Therefore, the idea is to update more and more these functionalities to reach Level 5, namely “full automation”, so that the car will be able to drive by itself without any human input.



SAE J3016™ LEVELS OF DRIVING AUTOMATION

	SAE LEVEL 0	SAE LEVEL 1	SAE LEVEL 2	SAE LEVEL 3	SAE LEVEL 4	SAE LEVEL 5
What does the human in the driver's seat have to do?	You are driving whenever these driver support features are engaged – even if your feet are off the pedals and you are not steering			You are not driving when these automated driving features are engaged – even if you are seated in “the driver’s seat”		
	You must constantly supervise these support features; you must steer, brake or accelerate as needed to maintain safety			When the feature requests, you must drive	These automated driving features will not require you to take over driving	
What do these features do?	These are driver support features			These are automated driving features		
	These features are limited to providing warnings and momentary assistance	These features provide steering OR brake/acceleration support to the driver	These features provide steering AND brake/acceleration support to the driver	These features can drive the vehicle under limited conditions and will not operate unless all required conditions are met	This feature can drive the vehicle under all conditions	
Example Features	<ul style="list-style-type: none"> • automatic emergency braking • blind spot warning • lane departure warning 	<ul style="list-style-type: none"> • lane centering OR • adaptive cruise control 	<ul style="list-style-type: none"> • lane centering AND • adaptive cruise control at the same time 	<ul style="list-style-type: none"> • traffic jam chauffeur 	<ul style="list-style-type: none"> • local driverless taxi • pedals/steering wheel may or may not be installed 	<ul style="list-style-type: none"> • same as level 4, but feature can drive everywhere in all conditions

Table 4.1: [20]

4.1.1 Generalities of ACC

The ACC has been designed extending the CC functionalities. Indeed, CC is an automatic control, introduced by Mitsubishi in 1995, able to keep a constant speed, even in case of external disturbances such as wind and road slope. To be more specific, a CC controls the throttle valve in order to maintain the speed constant at the reference value set by the driver. One important issue is that the driver must take back the control of the car if another ahead vehicle is being approached. Furthermore, CC can work adequately only in highways scenarios (high speed) but it cannot still be used inside a city due to the presence of a very high number of vehicles.

In case of free highway, the ACC behaves exactly as the CC, but in addition it can act either on the throttle valve and on the brakes to keep a desired safety distance from a preceding vehicle.

Furthermore, the limits of the CC can be overcome with the ACC by adding the Stop and go feature, which gives the possibility to exploit all the functionalities also in the urban scenario, at low speed ($< 30 \text{ km/h}$). Indeed, ACC with Stop&Go ensures that the vehicle is able to stop when a leading vehicle performs sudden brakes and to restart from rest in traffic jam situations.

The ACC can be made even more efficient introducing the V2V (vehicle to vehicle) communication, as in the cooperative adaptive cruise control (CACC), and the V2I (vehicle to infrastructure) communication.

Radar and Lidar are the sensors adopted in the ACC to acquire information from the ego vehicle (EV, i.e. the reference vehicle used in the experiments to evaluate the controller performances) and the preceding vehicle (PV).

In *figure 4.1* the basic ACC situation is shown. The EV (blue car) is approaching the PV (grey car).

The yellow radius indicates the radar beam implicated in the measurements acquisition and the yellow arrows indicate the distance between the front edge position x_{ev} of the EV and

the rear edge position x_{pv} of the PV. The sensor measures the relative distance d_{rel} (or clearance) and the relative velocity v_{rel} between the two cars, as

$$d_{rel} = x_{pv} - x_{ev} , \quad (4.1)$$

$$v_{rel} = v_{pv} - v_{ev} . \quad (4.2)$$

It is also possible to derive other two important variables, the time headway t_h and the time to collision t_c .

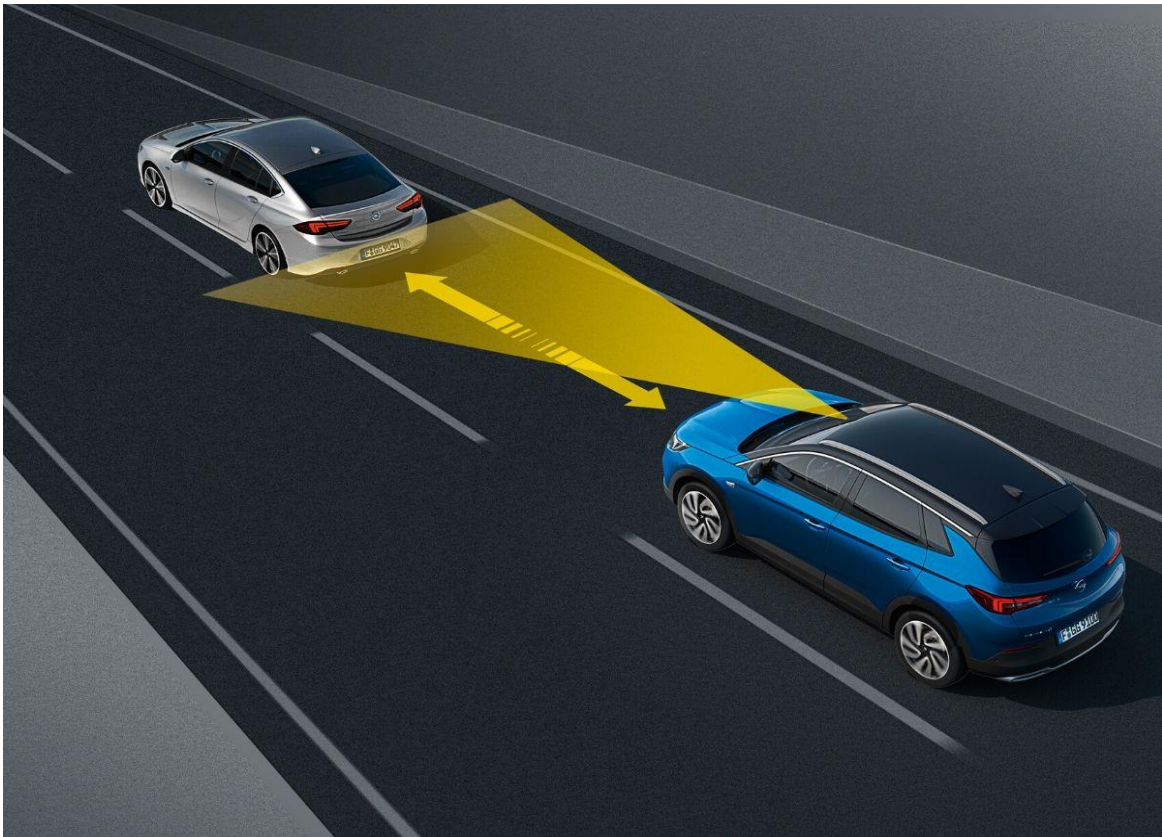


Figure 4.1: ACC standard situation

t_h is the time needed by the EV to collide with the PV when the EV keeps constant its velocity and the PV suddenly stops,

$$t_h = \frac{d_{rel}}{v_{ev}} \quad (4.3)$$

t_h is also called time gap since it indicates how the two vehicles are distant in terms of time.

t_c is the time required the EV to collide with the PV if they keep constant their velocities, proceeding on the same path,

$$t_c = \frac{d_{rel}}{v_{rel}} . \quad (4.4)$$

The ACC is implemented through a hierarchical architecture (figure 4.2) in which there are an upper level control and a lower level control in cascade.

The upper level control receives the reference speed, the safety distance, expressed in terms of time headway t_h , and all the measurements coming from the sensors, as inputs and provides the desired acceleration a_{des} computed through two different control strategies, the velocity control or the spacing control. Depending on the situations, a switching logic selects one modality or the other.

The lower level control receives a_{des} in input and computes the actuator commands, i.e. the throttle command or brakes command.

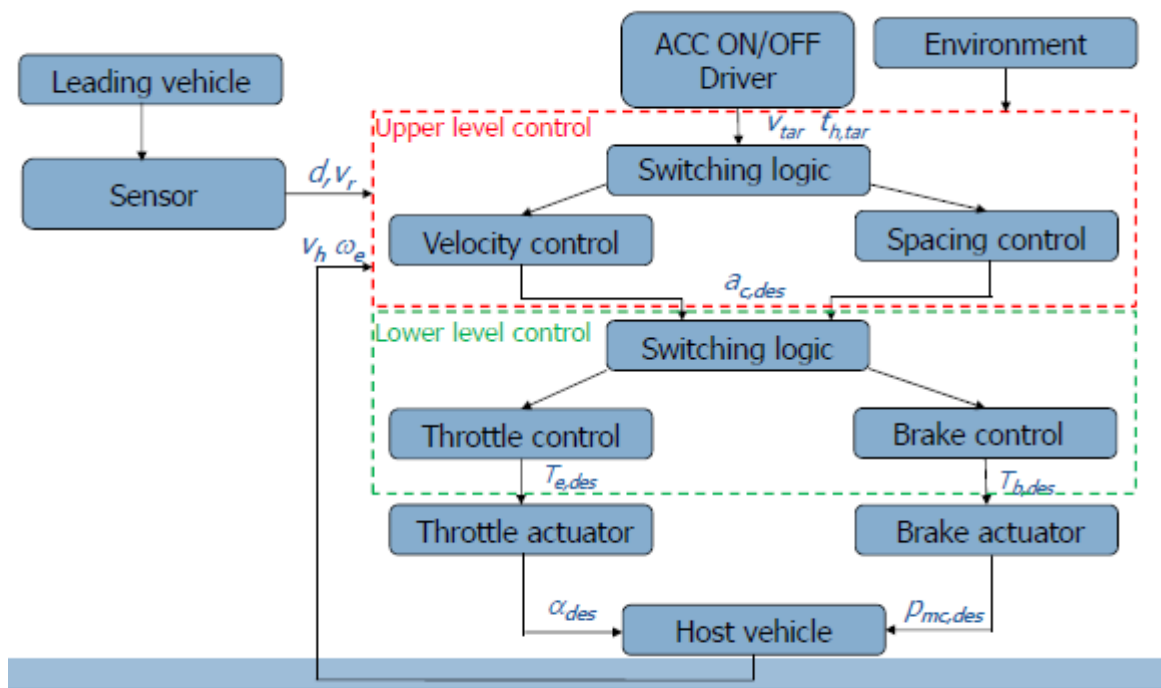


Figure 4.2: [21] ACC hierarchical control scheme

It is possible to notice that the ACC design implies a deep knowledge of the longitudinal vehicle dynamics that is going to be discussed in paragraph 4.2.

When the velocity control mode is selected, the aim of the ACC is to make v_{ev} equal to the target speed v_{tar} set by the driver, and keep it constant. This modality is activated when the radar does not detect any vehicle ahead in the same lane.

The control scheme is the one reported in *figure 4.3a*, where $C_{velocity}$ is the control transfer function and $e_v = v_{tar} - v_{ev}$ is the tracking error on the velocity.

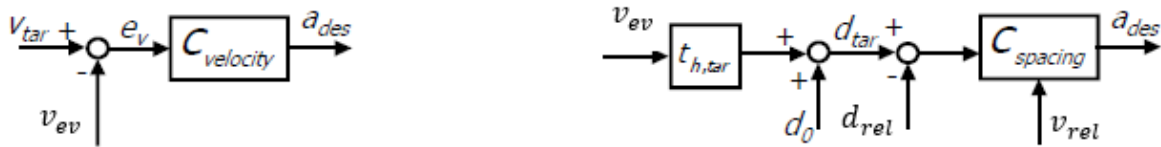


figure 4.3: (a) velocity control, (b) spacing control

The spacing control is chosen when the radar detects a PV inside the lane. The aim is to ensure that the EV maintains a safety distance from the PV defined by the driver setting $t_{h,tar}$.

The block scheme is in *figure 4.3b*. d_{tar} is the desired relative distance,

$$d_{tar} = d_0 + t_{h,tar}v_{ev} , \quad (4.5)$$

d_0 is the minimum relative distance that has to be guaranteed, and $C_{spacing}$ is the control transfer function.

4.1.2 Generalities of LKC

Different surveys proved that a large percentage of fatal car accidents must be attributed to unintended lane departures. To improve the safety of vehicles systems performing lane keeping support functions have been provided.

They are all based on the same working principle that consists of detecting the markings lane with a vision system, to establish the vehicle position inside the lane, intervening if the distance between the vehicle CoG (*centre of gravity*) and the lane centre exceeds a given threshold.

Lane keeping assistance systems can offer different functionalities, such as warning functions, intervention functions and control functions. For instance, the Lane Departure Warning (LDW) alerts the driver by emitting an acoustic signal during a dangerous situation, the Lane Keeping Assist applies a limited steering action to help the human driver when the car goes off the lane.

An important issue related to these systems is that they work well only if turns and overtaking are performed activating the turn indicator. In fact, if this not happens the systems perceives the manoeuvre as an unintentional behaviour and applies a counter torque with respect to the human driver action, generating a very risky situation.

Instead, the Lane Keeping Control (LKC) can completely control the steering wheel of the vehicle.

LKC can be used as an ADAS or it can be implemented in autonomous driving, making the car able to follow a predefined trajectory by itself, without any human control.

A LKC is designed by taking into account the lateral dynamics of the vehicle, for simplicity modelled through a single track model (it is going to be explained in the following paragraph of this chapter) and it is usually applied in highway scenarios respecting the following hypothesis: flat road, curvature radius $R \geq 500 \text{ m}$, small steering angles and almost constant longitudinal velocity v_x .

By a technical point of view, a camera mounted on the EV detects the lane markings up to a given look-ahead distance L so that a lane detection procedure gives a linear approximation of the lane centreline y (figure 4.4),

$$y = \tan(m)x + q, \quad (4.6)$$

where $\tan(m)$ is the tangent of the angle m between the vehicle longitudinal axis and the centreline approximation, x is the longitudinal distance, whereas q is the current lateral distance of the vehicle CoG from the centreline.

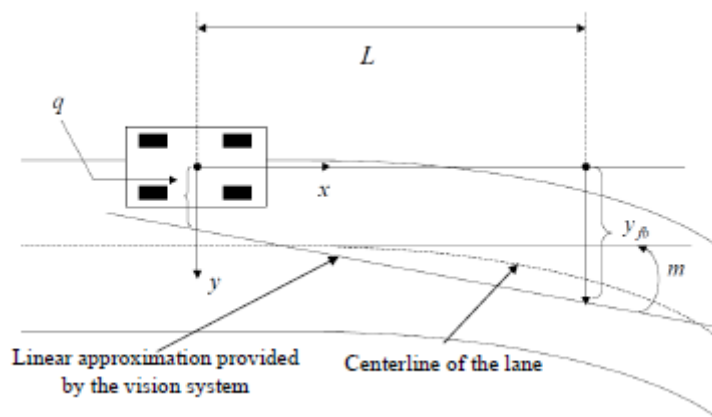


Figure 4.4: [22] centreline linear approximation provided by the vision system

Since m is very small (4.6) can be rewritten as

$$y \approx mx + q. \quad (4.7)$$

According to comfort requirements, the target of the LKC is to make equal zero not q , but the lateral distance y_{fb} at the look-ahead distance L ,

$$y_{fb} = mL + q. \quad (4.8)$$

Figure 4.5 shows the general control scheme of a LKC where the control action is computed in order to follow the reference signal $y_{fb,ref} = 0$.

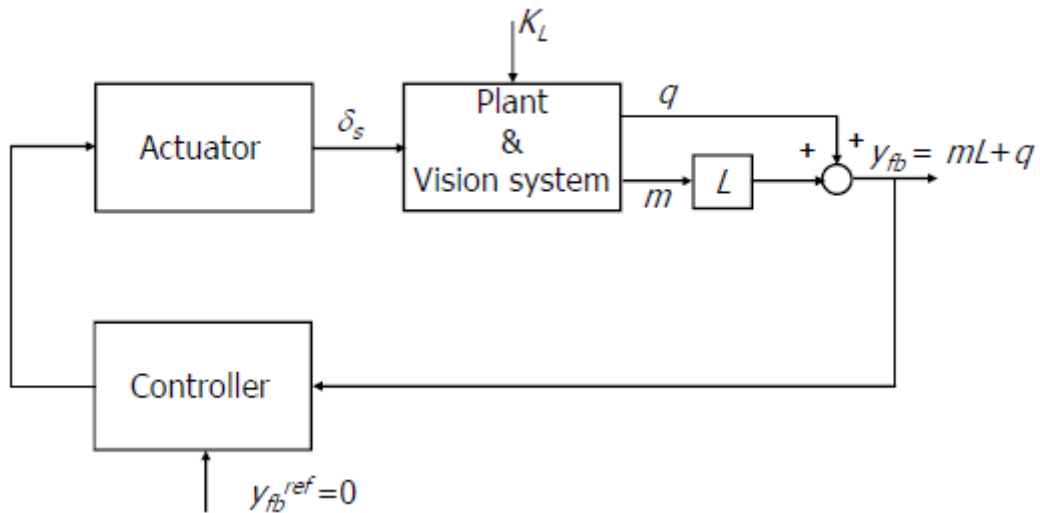


Figure 4.5 [21] Conceptual block scheme of the LKC with vision system

δ_s is the steering angle provided by the actuator and K_L is the road curvature that acts as a disturbance.

The vision systems can be incorporated inside the plant state-space representation or for example in Matlab by adding the sensor fusion blocks in the scheme, as explained in chapter 5.

However, a LKC works well only if some requirements are not violated:

- $|q| \leq q_{max}$, with $q_{max} \in [20 \div 40]cm$,
- lateral acceleration $|a_y| \leq a_{y,max}$, with $a_{y,max} \in [0.3 \div 0.4]g$.

Note that g is a unity of measure adopted in automotive to indicate acceleration values in terms of the gravity acceleration g.

4.2 Vehicle Model

In this section a brief explanation of the vehicle dynamics will be given because for the design of an ADAS control system and for autonomous vehicles it is necessary to have clear idea of the system under examination since autonomous driving car need to know the car model, i.e Kinematic and Dynamic models, in order to accomplish the function of path planning and control. Several mathematical models are available in the literature with different levels of complexity and accuracy according to the physical phenomena captured [23, 24, 25]. Models can consider different types of dynamics such as vertical, longitudinal, or lateral dynamics, or a combination of these. Depending on the purpose, a model must include representations of the appropriate vehicle systems. It may be necessary to model the effects of the suspension system, steering system, brake system or tires. The representations of these systems can be linear or non-linear depending on the accuracy required. Of course, the vehicle model used must be suitable for the maneuvers it will describe. Since in this thesis the aim is to implement an ACC and an LKC with high level (*figure 4.2*) control based on inverse reinforcement learning techniques, there is no need for an extremely detailed model since the focus is not on the stability controls realization. Specifically, the longitudinal dynamics will be presented in the case of the ACC and the lateral dynamics as regards the LKC as in this case it is important to verify the behavior of the vehicle when cornering. For the purpose of this thesis, a simple vehicle model will be examined, namely the two degrees of freedom or popularly known in the literature as a bicycle model. In the final section the model used in MATLAB/Simulink for the realization of this thesis project will be specifically presented.

4.2.1 Longitudinal Vehicle Dynamics

Vehicle motion is generally described in terms of the velocities: forward, lateral, vertical, roll, pitch and yaw in the vehicle-fixed coordinate system as referenced to an earth-fixed (inertial) reference frame, respectively u (*longitudinal velocity*), v (*side velocity*), w (*normal*

velocity), p (roll velocity), q (pitch velocity) and r (yaw velocity). A vehicle is modelled in two main parts: the unsprung mass that takes into account for the mass of the wheels, part of the suspension and other component directly connected to them, and the *sprung mass* that is basically the vehicle total mass supported by the suspension. The Society of Automotive Engineers (SAE) has introduced standard coordinates and a notation to describe vehicle dynamics that are widely used [23, 24] (figure 4.6). Here the earth-fixed reference frame's axes are referred as X , Y and Z and the vehicle reference frame's axes as x , y and z . The angles are roll (θ), pitch (ϕ) and yaw (ψ).

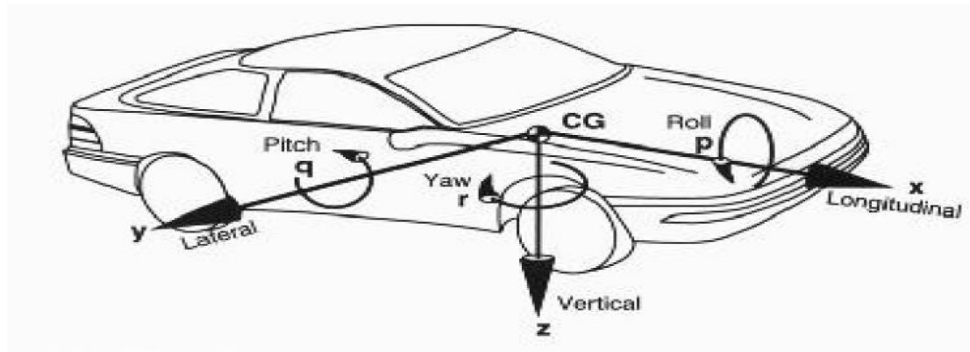


Figure 4.6: Vehicle Axis System (SAE)

More specifically, referring to the bicycle model in figure 4.7 the key variables are:

- u is the forward velocity
- v is the lateral velocity
- ψ is the yaw angle
- β is the sideslip angle
- $\rho = 1/R$ is the curvature radius where $R = \overline{OO'}$

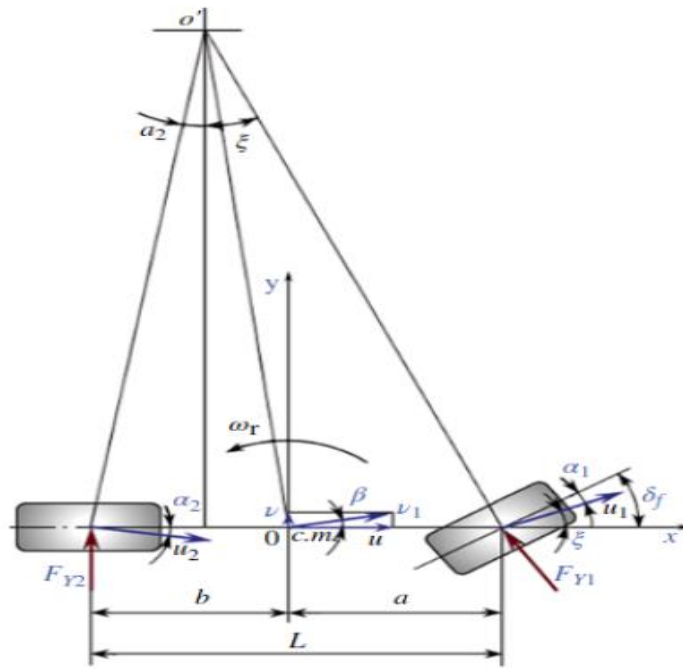


Figure 4.7: Bicycle model

The two major elements of the longitudinal vehicle model are the vehicle dynamics and the powertrain dynamics. The vehicle dynamics are influenced by longitudinal tire forces, aerodynamic drag forces, rolling resistance forces and gravitational forces. The study of this chapter is focused only in the vehicle dynamics. The longitudinal powertrain system of the vehicle consists of the internal combustion engine, the torque converter, the transmission and the wheels but this contribution will not be discussed here.

Consider a vehicle moving on an inclined road as shown in *figure 4.8*. The external longitudinal forces acting on the vehicle include aerodynamic drag forces, gravitational forces, longitudinal tire forces and rolling resistance forces.

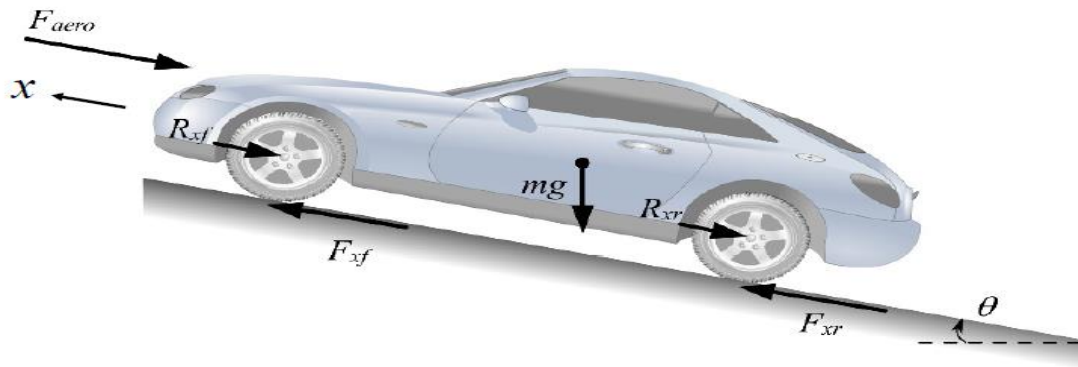


Figure 4.8: Longitudinal forces acting on a vehicle moving on an inclined road [24]

The force balance along the vehicle longitudinal axis is:

$$m\ddot{x} = F_{xf} + F_{xr} - F_{aero} - R_{xf} - R_{xr} - mg \sin(\theta) \quad (4.9)$$

Where

- F_{xf} is the longitudinal tire force at the front tires
- F_{xr} is the longitudinal tire force at the rear tires
- $F_{aero} = \frac{1}{2} \rho C_d A_F (V_x + V_{wind})^2$ is the equivalent longitudinal aerodynamic drag force
- ρ is the mass density of the air
- C_d is the aerodynamic drag coefficient
- A_F is the frontal area of the vehicle
- V_x is the longitudinal velocity
- V_{wind} is the wind velocity
- R_{xf} is the force due to rolling resistance at the front tires
- R_{xr} is the force due to rolling resistance at the rear tires
- m is the mass of the vehicle
- g is the acceleration due to gravity
- θ is the angle of inclination of the road on which the vehicle is traveling

Ignoring the road gradient and wind speed, the longitudinal dynamics can be represented as:

$$ma_x = F_{xf} + F_{xr} - R_x - D_a V_x^2 \quad (4.10)$$

Since there is neither braking nor throttle angle inputs, the longitudinal tire force under these conditions is small and can be assumed to be zero. The road is assumed to be level with $\theta = 0$ and the wind velocity *wind* is assumed to be zero. Under these conditions, the longitudinal dynamics equation can be re-written as:

$$-m \frac{dV_x}{dt} = D_a V_x^2 + R_x$$

Notice that $D_a = \frac{1}{2} \rho C_d A_F$ is the aerodynamics drag coefficient.

4.2.2 Lateral Vehicle Dynamics

A “bicycle” model of the vehicle with two degrees of freedom is considered, as shown in *figure 4.9*. The two degrees of freedom are represented by the vehicle lateral position y and the vehicle yaw angle ψ .

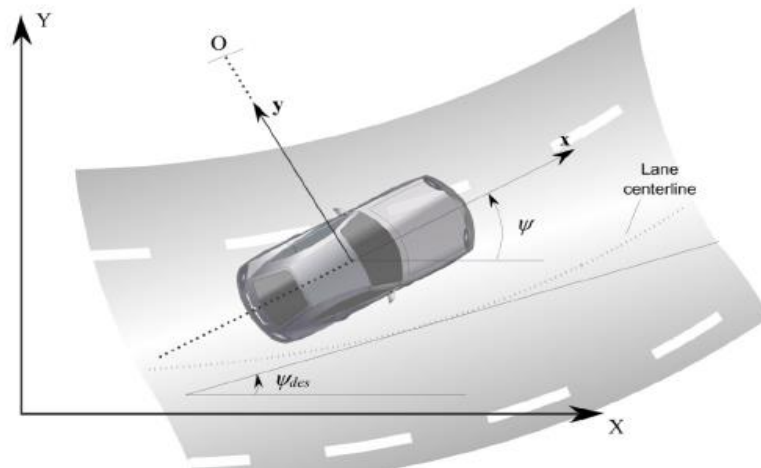


Figure 4.9: Lateral Vehicle Dynamics [24]

Applying Newton's second law for motion along the axis:

$$ma_y = F_{yf} + F_{yr} \quad (4.11)$$

Where $a_y = \frac{d^2y}{dt^2}$ is the inertial acceleration of the vehicle at the Center of Gravity (CoG) in the direction of the y axis, F_{yf} and F_{yr} are the lateral tire forces of the front and rear wheels respectively. Two terms contribute to the a_y that are the acceleration \ddot{y} which is due to motion along the y axis and the centripetal acceleration $V_x\dot{\psi}$, hence:

$$a_y = \ddot{y} + V_x\dot{\psi} \quad (4.12)$$

Substituting the eq. (4.12) into eq. (4.11), the equation for the lateral translational motions of the vehicle is obtained:

$$m(\ddot{y} + \dot{\psi}V_x) = F_{yf} + F_{yr} \quad (4.13)$$

Considering now the moment balance about the z axis, the equation for the yaw dynamics follows:

$$I_z\ddot{\psi} = l_f F_{yf} - l_r F_{yr} \quad (4.14)$$

Where l_f and l_r are the distances of the front tire and the rear tire respectively from the CoG of the vehicle.

From experimental results it can be seen that lateral tire force of a tire is proportional to the “slip-angle” for small slip-angle, where the slip-angle of a tire is defined as the angle between the orientation of the tire and the orientation of the wheel (*figure 4.10*). The slip angles of the front and rear wheels are denoted as:

$$\alpha_f = \delta - \theta_{vf} \quad (4.15)$$

$$\alpha_r = -\theta_{vr} \quad (4.16)$$



Figure 4.10: Slip angle

The lateral tire force for the front wheels can therefore be written as:

$$F_{yf} = 2C_{\alpha f}(\delta - \theta_{vf}) \quad (4.17)$$

Where the constant $C_{\alpha f}$ is the cornering stiffness of each front tire, δ is the front wheel steering angle and θ_{vf} is the front tire velocity angle (factor 2 accounts for the fact that there are two front wheels). Similarly, for the rear wheels it can be written:

$$F_{yr} = 2C_{\alpha r}(-\theta_{vr}) \quad (4.18)$$

Where the $C_{\alpha r}$ is the cornering stiffness of each rear tire and θ_{vr} is the rear tire velocity angle.

The front and rear tire velocity angle can be computed as follow:

$$\tan(\theta_{Vf}) = \frac{V_y + l_f \dot{\psi}}{V_x} \quad (4.19)$$

$$\tan(\theta_{Vr}) = \frac{V_y - l_r \dot{\psi}}{V_x} \quad (4.20)$$

For small angles the eqs. (4.19), (4.20) can be simplified using small angle approximation (notice that $V_y = \dot{y}$):

$$\theta_{Vf} = \frac{\dot{y} + l_f \dot{\psi}}{V_x} \quad (4.21)$$

$$\theta_{Vr} = \frac{\dot{y} - l_r \dot{\psi}}{V_x} \quad (4.22)$$

Then substituting from eqs. (4.15), (4.16), (4.21), (4.22) into eqs. (4.13) and (4.14), the state space model can be found:

$$\frac{d}{dt} \begin{bmatrix} y \\ \dot{y} \\ \psi \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & -\frac{2C_{\alpha f} + 2C_{\alpha r}}{mV_x} & 0 & -V_x - \frac{2l_f C_{\alpha f} - 2l_r C_{\alpha r}}{mV_x} \\ 0 & 0 & 0 & 1 \\ 0 & -\frac{2l_f C_{\alpha f} - 2l_r C_{\alpha r}}{I_z V_x} & 0 & -\frac{2l_f^2 C_{\alpha f} - 2l_r^2 C_{\alpha r}}{I_z V_x} \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{2C_{\alpha f}}{m} \\ 0 \\ \frac{2l_f C_{\alpha f}}{I_z} \end{bmatrix} \delta$$

4.2.3 Simulink models

In this project the vehicle dynamics has been realized in Simulink using the bicycle model inside the Vehicle Dynamics Blockset™.

It implements a 3DOF (Degrees of Freedom) rigid single track vehicle model that approximates the longitudinal, lateral and yaw motion.

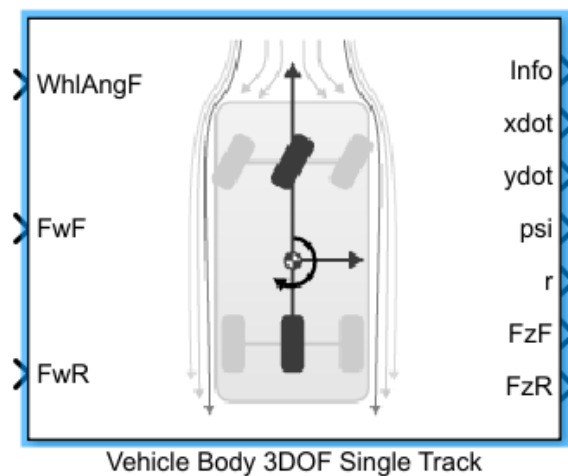


Figure 4.11: Simulink vehicle body 3 degrees of freedom with external longitudinal force inputs

In particular, for the ACC the bicycle model with force input has been chosen (*figure 4.11*), where the block receives as inputs the external longitudinal forces that are used to accelerate and brake the car and gives in output the lateral forces through the tire slip angles and the cornering stiffness.

The *info* output is a bus containing all the information related to the vehicle dynamics, such as longitudinal and lateral position, longitudinal and lateral acceleration and so on. *r* stands for the yaw angle whereas *psi* indicates the yaw rate.

Instead, for the LKC the choice fell on the bicycle model with velocity input (*figure 4.12*) that works with longitudinal acceleration approximately equal to 0 since, for simplicity, the

longitudinal velocity has been imposed constant. As output the block provides only the lateral forces.

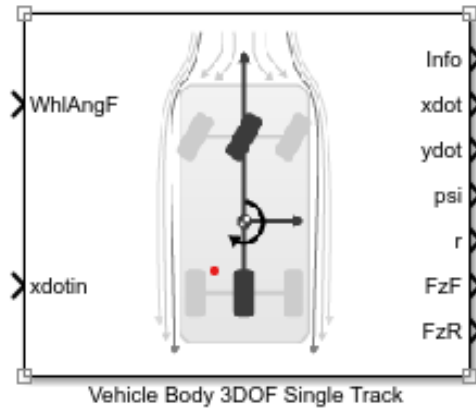


Figure 4.12: Simulink vehicle body 3 degrees of freedom with velocity input

The equations to which the Matlab single track model refers can be written in the state-space representation as,

$$\frac{d}{dt} \begin{bmatrix} v_y \\ \Psi \\ \dot{\Psi} \\ v_x \\ \dot{v}_x \end{bmatrix} = \begin{bmatrix} -\frac{2C_f + 2C_r}{mv_x} & 0 & -v_x - \frac{2C_f l_f - 2C_r l_r}{mv_x} & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ -\frac{2C_f l_f - 2C_r l_r}{I_z v_x} & 0 & -\frac{2C_f l_f^2 + 2C_r l_r^2}{I_z v_x} & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & -\frac{1}{\tau} \end{bmatrix} \begin{bmatrix} v_y \\ \Psi \\ \dot{\Psi} \\ v_x \\ \dot{v}_x \end{bmatrix} + \begin{bmatrix} \frac{2C_f}{m} \\ 0 \\ 2C_f l_f \\ \frac{I_z}{0} \\ 0 \end{bmatrix} \delta + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ \frac{1}{\tau} \end{bmatrix} u + \begin{bmatrix} 0 \\ 0 \\ 0 \\ v_y \dot{\Psi} \\ 0 \end{bmatrix},$$

where v_x and v_y are the longitudinal and the lateral acceleration respectively, $\dot{\Psi}$ is the yaw rate, δ is the front steering angle and u is the longitudinal acceleration input. All the other parameters have been set in the Matlab setup script.

Chapter 5

5 IRL strategy for ACC and LKC

This chapter, intended as the core of the thesis, contains all the clarifications and justifications for the procedures that have been followed to get an IRL agents able to replace the high level control of an Adaptive Cruise Control (ACC) and of a Lane Keeping Control (LKC).

5.1 Problem presentation

The primary goal of the project is to solve the control problem addressed by an ACC and a LKC, through RL algorithms. Nonetheless, as it has been claimed in chapter 3, some of the tasks that one can think to fulfil by RL techniques, are actually performed in too complex and strongly dynamical environments, setting the stage for all the limits that cannot be overcome by a pure RL theory. The discussed problem falls exactly inside that class.

Indeed, the design of an ACC or a LKC requires to work with continuous physical quantities (velocities, distances, accelerations, steering angle and so on) changing inside a scenario deeply unstructured and very unpredictable. Thus, it seems truly unlikely to construct a handcrafted reward function embedding all the combinations of events that might occur.

This is the reason that leads to choose the IRL strategy.

Among all the approaches that have been studied through researches on the state-of-the-art, the choice fell on the projection-based method since, on the basis of the collected papers, it seems to be the most applied in real world experiments and moreover the simplest, from the implementation point of view.

All data needed for the application of the strategy has been provided by Addfor S.p.A.

The entire code has been written in MATLAB 2019b and the agent design has been performed using the reinforcement learning toolbox. Simulink has been used to build all the environments in which the agent has been trained. Finally, Driving Scenario Designer has been employed for the construction of driving scenarios that are able to conglomerate the meaningful events occurring when the ACC or LKC functionality is activated.

5.2 Data collection and processing

The first part of the project has been devoted to the collection and reorganization of the data, so that they have been reformatted in order to avoid possible issues linked to the dimensionality and to neglect all the values made incorrect by the simulator.

This was a crucial procedure since data have been used for the computation of the expert driver's feature expectation μ_E , that plays a key role in the applied IRL algorithm, as extensively clarified in chapter 3.

Data were extracted by a simulation in which two human expert drivers drove the ego vehicle (EV) in a predefined map, adopting two driving styles, one more relaxed than the other.

The map was divided in regions where the drivers must deal with different situations.

For the LKC data, referring to regions in which straight road stretches and stretches with road curvature not equal to 0 are alternated, have been considered. Whereas, for the ACC the attention has been focused on the part of the simulation in which the EV was preceded by a preceding vehicle (PV).

All the trajectories have been sampled following these guidelines, valid either for the ACC and the LKC, and have been truncated, so that they are all constituted by the same number of samples, thus satisfying one of the algorithm requirements.

In *figure 5.1* a part of the code is shown, written to perform the described operations. **N_samples** indicates the number of samples by which every trajectory must be constituted, and it has been set equal to 300 in order to avoid of performing simulations that requires too much computational effort during the training phase. **traj_truncated_list** is a list that contains the truncated trajectories.

```
for i=1:num_traj
    traj_full=cell2mat(features_Pascal(i));
    samples=size(cell2mat(features_Pascal(i)),1);
    a=randi([1 samples-N_samples-1]);
    for j=1:N_features
        traj_truncated(:,j)=traj_full(a:a+N_samples-1,j);
    end
    traj_truncated_list=[traj_truncated_list,trajectories];
end
```

Figure 5.1: code to perform trajectories truncation

After the first step of data processing, features extraction has been performed separately for the two applications.

5.2.1 ACC features extraction

In the data file there are many values related to quantities (velocities and accelerations, yaw rate, lane gap, curvature radius and so on) that characterize the EV, the PV and all the other actors involved in the simulation. Nevertheless, a great number of those quantities are not very significant for abstracting the ACC control problem and so they have been neglected.

Taking into account the main physical quantities used to produce a classical ACC logic and to define the comfort requirements, the ACC features vector has been extracted,

$$\Phi_{ACC} = \begin{bmatrix} v_x \\ d_{rel} \\ v_{rel} \\ \varphi_{collision} \\ \varphi_{reverse} \\ \varphi_{safety_distance} \end{bmatrix}, \quad (5.1)$$

with v_x as the longitudinal velocity of the EV, d_{rel} and v_{rel} as the relative distance and the relative velocity between the EV and the PV, respectively.

The fourth component $\varphi_{collision}$ has been inserted to take into consideration potential crashes, so that if the crash happens $\varphi_{collision}$ is set to 1, otherwise is equal to 0. The fifth component $\varphi_{reverse}$ was added a posteriori after that, at the end of the first simulations, the agent was not able to discern the real goal, i.e. to behave as if the ACC were activated, from the wrong target of maximizing the relative distance, fulfilled by inverting the direction of travel. $\varphi_{reverse}$ associates a big penalty to this behaviour. The last component $\varphi_{safety_distance}$ tells if the agent overcame the imposed safety distance. Like $\varphi_{collision}$, also these last two features assume a value that can be 0, if the agent behaves correctly, or 1 if it performs the wrong action. Thereafter, all the components have been normalized (*figure 5.2*) in values between 0 and 1 (with 0 and 1 included) so that $\|\omega\|_1 \leq 1$ is verified, where ω is the vector of weights used to build the reward function.

The normalization has been carried out choosing the minimum and the maximum value of each Φ_{ACC} component as the extremes of the intervals to which the features belong.

Then, every negative feature value has been turned to positive by adding the minimum of the related interval.

As last step, each one of the new values has been divided by the corresponding interval amplitude.

```
%% features normalization between 0 and 1

range_radar          = 175;

min_ego_veh_vel      = -43.2;      % [km/h]
max_ego_veh_vel      = 129.6;      % [km/h]

rel_speed_range      = 65;         % [m/s]

ego_veh_vel_range=max_ego_veh_vel-min_ego_veh_vel;

traj_normalized_list={};
for i=1:num_traj
traj=cell2mat(traj_truncated_list(i));

for j=1:N_features

    if j==1
        traj(:,j)=(traj(:,j)-min_ego_veh_vel)./ego_veh_vel_range;
    end

    if j==2
        traj(:,j)=traj(:,j)./range_radar;
    end

    if j==3
        traj(:,j)=(traj(:,j)+36)./rel_speed_range;
    end

end
traj_normalized_list=[traj_normalized_list,traaj];
end
```

Figure 5.2: code to perform features normalization

5.2.2 LKC features extraction

The same reasoning has been carried forward for the LKC features extraction.

Thus, the LKC features vector is Φ_{LKC} ,

$$\Phi_{LKC} = \begin{bmatrix} K_R \\ \dot{\Psi} \\ a_y \\ \dot{a}_y \\ q_{lane} \\ \varphi_{lane_departure} \end{bmatrix}, \quad (5.2)$$

where K_R is the road curvature, $\dot{\Psi}$ is the yaw-rate, a_y is the chassis lateral acceleration, \dot{a}_y is the jerk used for satisfying comfort requirements and q_{lane} is the lateral deviation of the vehicle CoG (centre of gravity) from the centre of the lane.

The sixth component $\varphi_{lane_departure}$ has a job very similar to the $\varphi_{collision}$ in (5.1), because it can be either equal to 0, if the car remains inside the lane, or to 1 if there is a departure and the vehicle goes off-road.

The structure of the code, and so the logic, developed to perform the features normalization is identical to that of the ACC. It has been sufficient to change consistently the parameters for the computation.

For the LKC data processing a few more steps have been necessary, since among the q_{lane} values some of them were corrupted by simulation events which cannot occur in real life, such as the presence of road discontinuities between two different map regions.

Therefore, all those values have been neglected to reduce the probability that the agent perceived as “good” some misbehaviour.

5.3 Main algorithm

Inside the main script of the algorithm, three phases can be distinguished:

1. Initialization phase,
2. IRL phase,
3. RL phase.

In the initialization phase μ_E , μ_0 and ω_0 , three parameters used as inputs in the IRL phase, are computed.

$\mu_E \in \mathbb{R}^n$ is the driver features expectation, computed as suggested by (3.2). Actually, in both the applications two μ_E have been derived, one for each driving style. The dimension n is equal to the number of selected features.

```
%calcolo mu_E normalizzata
mu_E_normalized_vec=zeros(3,1);
N_SAMPLES_NEW=510;
for i=1:num_traj
traj=cell2mat(traj_normalized_list(i));

    for j=1:N_features
mu_E_normalized_traj(j,1)=sum(traj(1:10:N_samples,j));
mu_E_normalized_traj(j,1)=mu_E_normalized_traj(j,1)/N_SAMPLES_NEW;
    end

mu_E_normalized_vec=mu_E_normalized_vec+mu_E_normalized_traj;
end

mu_E_normalized=mu_E_normalized_vec/num_traj;

mu_E_collision    = 0;      % features to indicate a collision
mu_E_reverse      = 0;
mu_E_safe_dist    = 0;
mu_E_scaling=[mu_E_normalized;mu_E_collision;mu_E_reverse;mu_E_safe_dist];

save('mu_E_scaling')
```

Figure 5.3: μ_E computation

Paying attention, in *figure 5.3* it can be noticed that from the truncated trajectories not all samples have been involved; this because data have a sampling time equal to 0.01 s then, in order to be coherent with the simulations running at 10 Hz ($T_s = 0.1$ s), values have been collected after each tenth of second. The choice to select for the simulation a frequency $f = 10$ Hz rather than $f = 100$ Hz has been forced by the fact that it would have been required a too long simulation time and too much space in memory.

From theory, it has been learned that the projection-based method demands vector $\boldsymbol{\mu}_0$ to start the first iteration, that is the first feature expectation computed on-line during the agent simulation.

Speaking of which, a first Simulink environment (it is going to be described in the following paragraph) has been constructed just for $\boldsymbol{\mu}_0$. Here, the agent receives as input a totally random policy.

At this point $\boldsymbol{\omega}_0$, namely the implicit definition of the reward ($\mathcal{R} = \boldsymbol{\omega}^T \boldsymbol{\Phi}$) for the first IRL iteration, is easy to get,

$$\boldsymbol{\omega}_0 = \boldsymbol{\mu}_E - \boldsymbol{\mu}_0.$$

During the IRL phase a new $\bar{\boldsymbol{\mu}}_{sim}$ is computed at every iteration as the orthogonal projection of $\boldsymbol{\mu}_E$ on $\boldsymbol{\mu}_{sim}$ in according to (3.14), where $\boldsymbol{\mu}_{sim}$ is the features expectation vector extracted from the agent simulation at the previous iteration.

Afterwards, weights are updated through the comparison between $\boldsymbol{\mu}_E$ and $\boldsymbol{\mu}_{sim}$ and then the condition $t(i) \leq \varepsilon$, $t(i) = \|\boldsymbol{\omega}\|_2$ is checked, as illustrated in *figure 5.4*. If it is respected, the algorithm stops, meaning that the desired reward function has been found.

```

w(:, NUM_IRL_ITERATION) = mu_E_scaling - mu_mean(:, NUM_IRL_ITERATION - 1);

t(NUM_IRL_ITERATION) = norm(w(:, NUM_IRL_ITERATION), 2);

% check

if t(NUM_IRL_ITERATION) <= epsilon
    break
end

```

Figure 5.4: algorithm stop condition

For what concern the RL phase, it is fundamental to train the agent and so to find the optimal policy with respect to the current reward function. In this way, an iterative process is built, where the reward function of the IRL phase and the optimal policy coming from the RL phase improve each other until convergence.

Here, the agent has been designed and all the training options have been set.

5.3.1 Simulink environments

It is well known that in RL the agent must gain information from the interaction with the surrounding environment. For this reason, it is compulsory to create a virtual environment in which the agent has to navigate, enhancing its optimal policy.

In this experiment all the environments have been constructed in Simulink, since the code has been developed in Matlab, by modifying the already existing Simulink block schemes realized by Mathworks for the implementation of a classical ACC (*figure 5.5*) and a LKC with a MPC (*figure 5.7*).

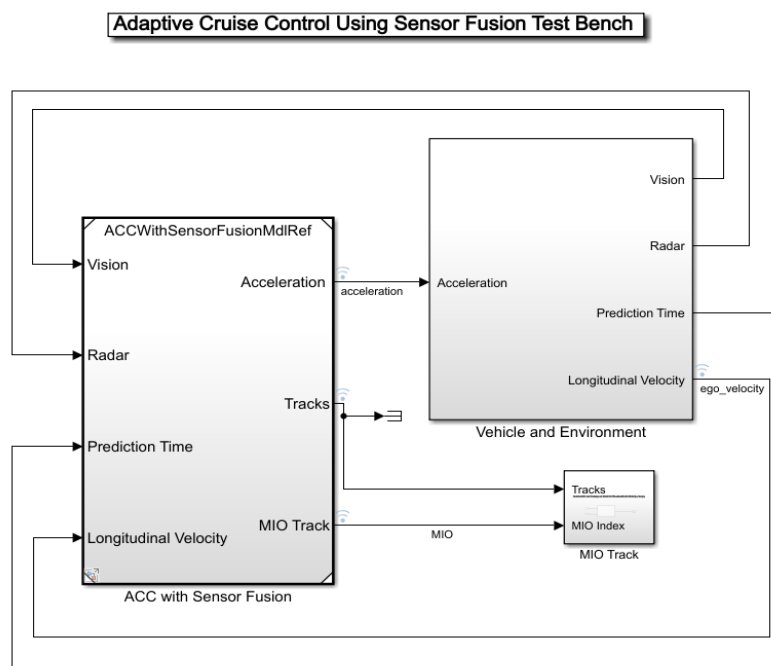


Figure 5.5: Adaptive cruise control block scheme

Inside the block “*vehicle and environment*” (figure 5.5) there is the classical ACC control (figure 5.6) that has been designed through a simple PD.

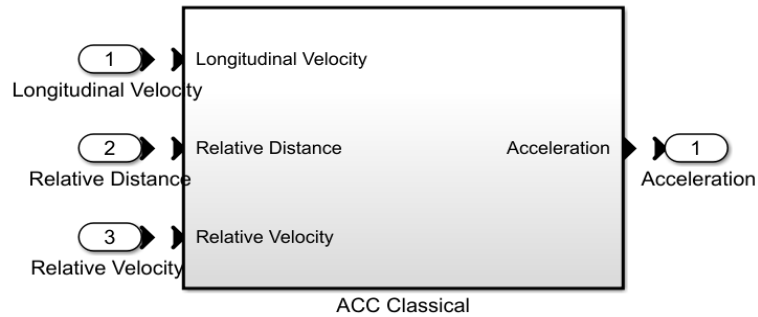


Figure 5.6: ACC control block

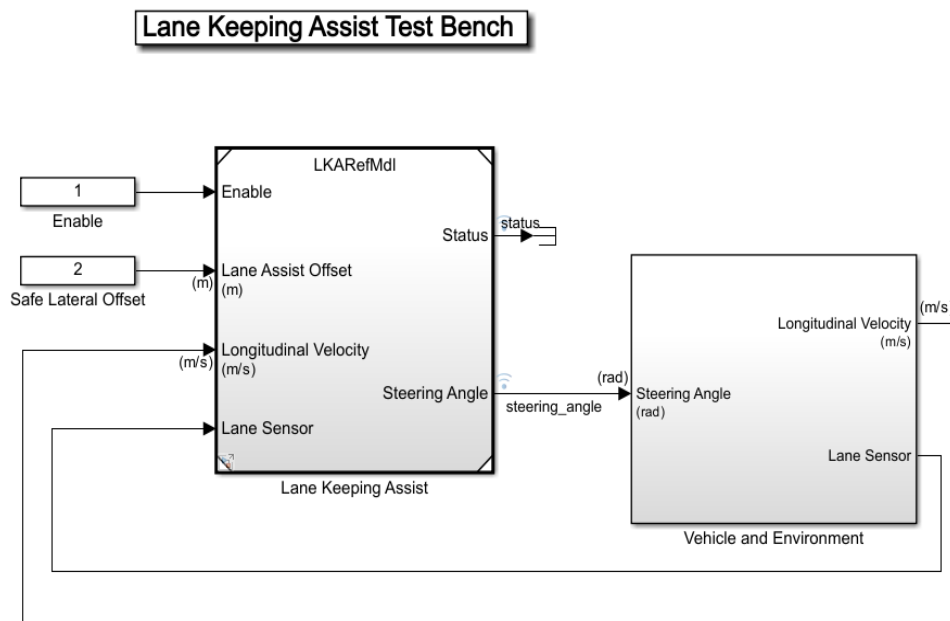


Figure 5.7: Lane Keeping Control with MPC

For both applications three environments have been created, the first for μ_0 , the second for training and the third for agent testing.

Inside *the* μ_0 block scheme no control and so no feedback signal is present since a selected random policy is directly provided as control action. At each time step, Φ is sent to workspace, where μ_0 is computed.

The training environment has been obtained by replacing the ACC and LKC control blocks with the Matlab RL block that represents the agent.

It receives 3 inputs, called *observation*, *reward* and *isdone*.

Observation is the features vector that is acquired extracting at every time step the six features, coming from the vehicle dynamics and sensor fusion. As one can notice from *figure 5.8*, the features normalization has been done directly in Simulink.

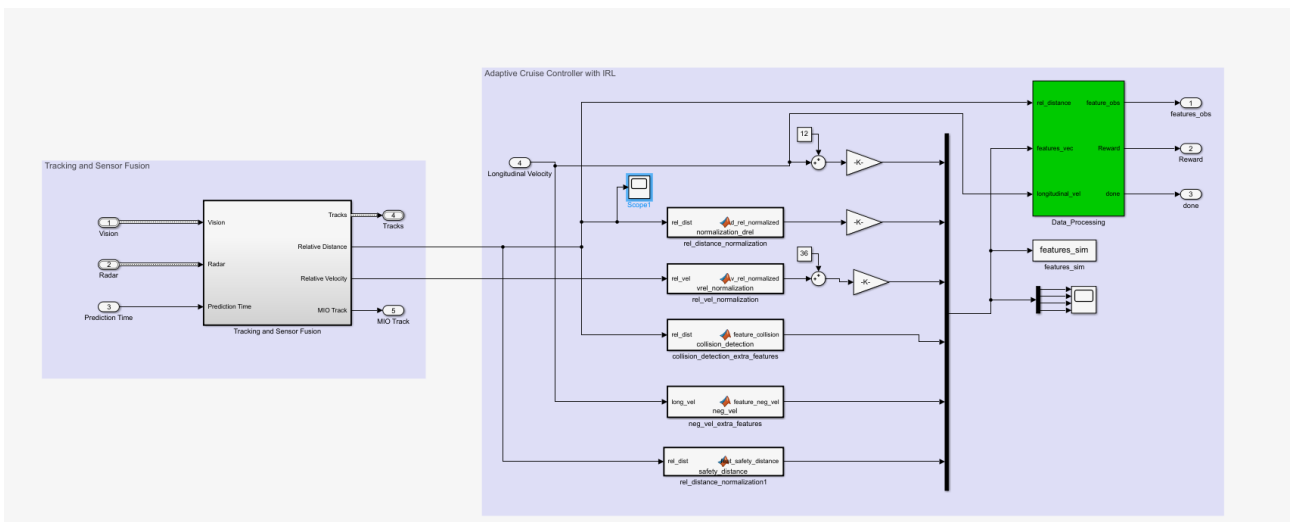


Figure 5.8:

Simulink scheme realized for the features normalization in the ACC. Nevertheless, for the LKC the scheme has the same structure.

The reward is provided by a Matlab function that implements equation (3.1), while *isdone* generally indicates a terminal condition but here it is not used, so this input has been connected to “ground”. Actually, since after some simulations of the ACC it has been observed that the agent learned to perceive a crash and a change of direction as undesired behaviours too slowly, the reward function has been further modified by assigning directly a big penalty in conjunction with those situations, as shown in *figure 5.9*.


```

function R = reward_function(features_vec,w,vel_long,rel_dist)
W_tras=w';

if vel_long<=0
    R=-20
elseif rel_dist==inf
    R=-20
else
    R = W_tras*features_vec;
end
R;

```

Figure 5.9: modified Reward function

At the end of the RL phase the agent is deployed on the simulation environment in order to test the learned policy. The only thing changing with respect to the training environment is that the features are collected and sent to the workspace for the μ_{sim} computation, as it happens for μ_0 .

5.3.2 Driving Scenario Designer

It can be immediately understood how, for the addressed problem, sensors play a fundamental role since they collect data that are essential in this kind of application and perform computer vision. Driving Scenario Designer (DSD) is a Matlab application that permits to build a customized driving scenario, in which the EV can be tested, whereas the sensor fusion block allows to mount sensors on the EV, such as cameras and radars.

In this project either for the ACC and for the LKC one driving scenario has been created using DSD.

In *figure 5.10* the ACC driving scenario is showed, where there are only two actors, the EV, controlled by the IRL agent, and the PV that follows a predefined trajectory, modifiable by adding or erasing waypoints, with a variable velocity. In *figure 5.10*, on the right it can be seen a list containing the coordinates x, y and z (in m) and the speed (in m/s) of the PV related

to each waypoint of the trajectory. The road has a curvature radius equal to infinite. On the EV a camera and a radar are mounted. For the whole duration of the simulation EV and PV follow always the same lane, since the aim is to verify that the agent learns to behave correctly in the presence of a leading vehicle.

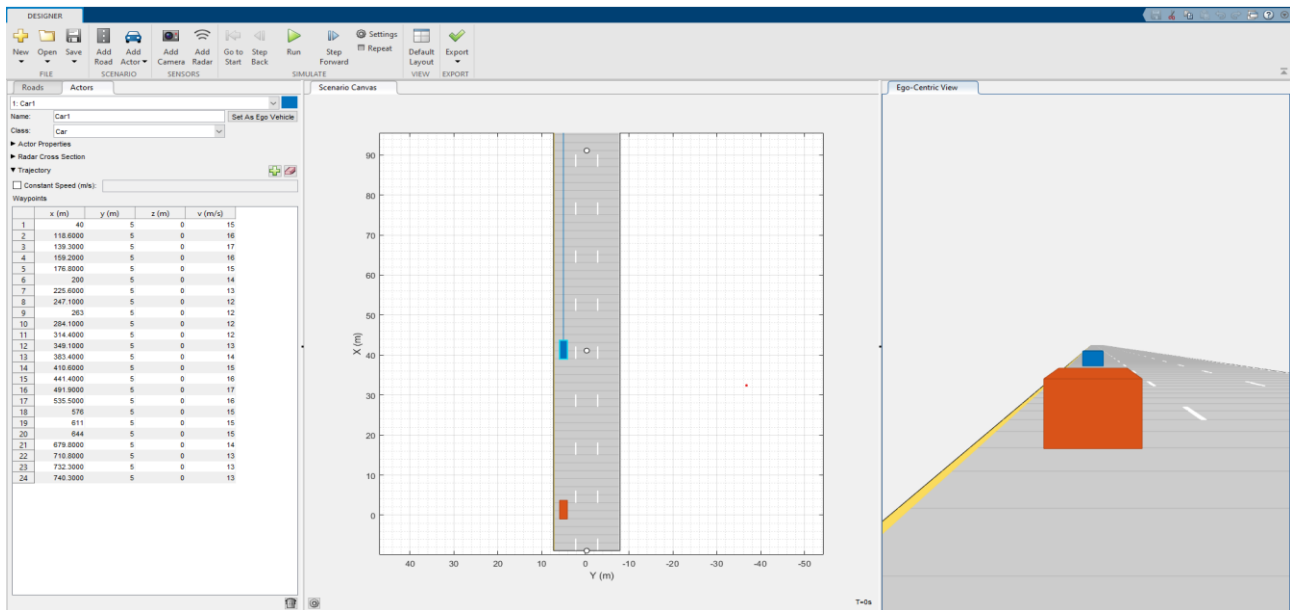


Figure 5.10: ACC driving scenario in Driving Scenario Designer

In figure 5.11 the scenario for the LKC is presented. Here there is only one actor, the EV, whose speed has been set constant for simplicity. The vision system for the lane detection is realized by a camera from which all the lane information are extracted to estimate the vehicle position inside the lane. The road has been designed with an “S” shape so that the agent can be tested during either a right and a left turn. Again, the EV is controlled by the IRL agent.

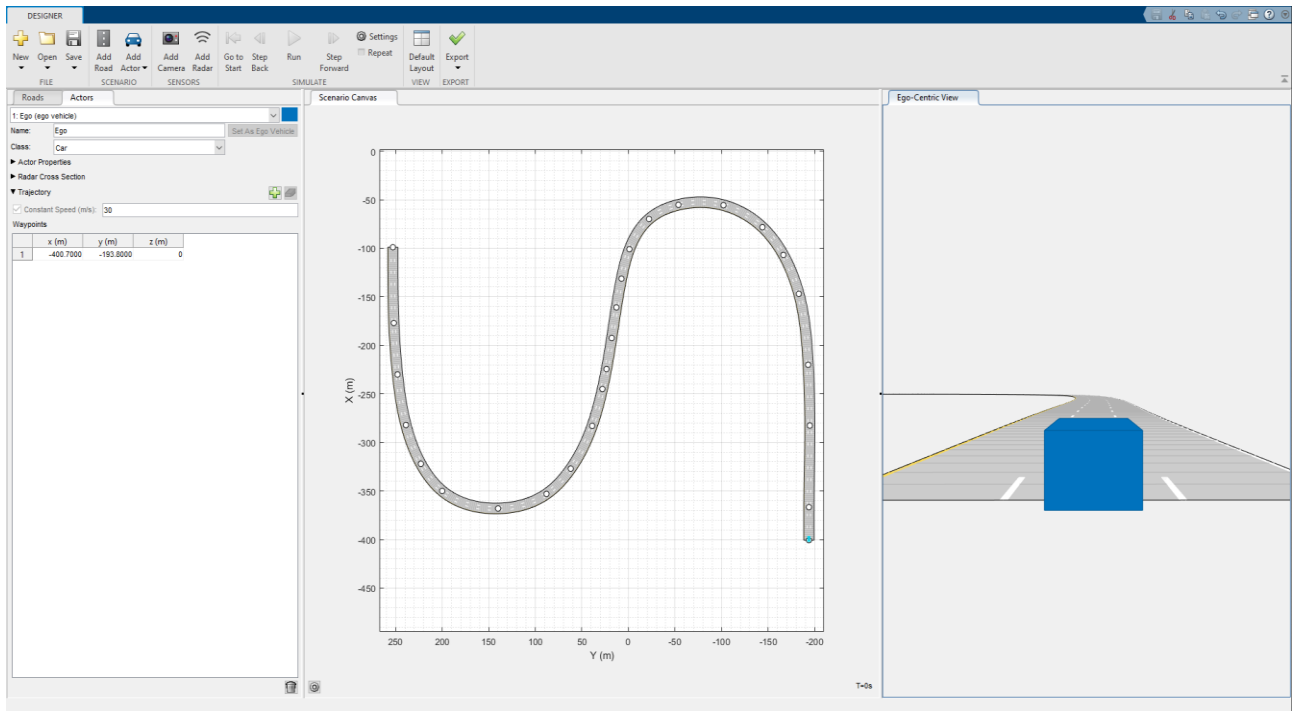


Figure 5.11: LKC driving scenario

5.3.3 Agent design

The IRL agent has been designed in Matlab as a deep Q-network that receives the features and the action vectors as inputs and gives the control action as output.

Despite acceleration (ACC) and steering angle (LKC) assume continuous values, in order to use a DQN it has been necessary to discretize them, obtaining two sets of actions among which the agent can choose.

In particular, $+2 \text{ m/s}^2$ and -3 m/s^2 have been taken as maximum and minimum acceleration values, while $+15^\circ$ and -15° are the maximum and minimum steering angle values, respectively. The acceleration interval has been divided into 51 segments with amplitude equal to 0.1 m/s^2 and the steering angle interval into 31 segments with amplitude equal to 1° . Note that such stringent comfort requirements on the longitudinal acceleration

have been chosen just to be coherent with Data, in which those values have never been overcome.

For both applications the critic network Q_{critic} has been designed with 2 input layers, i.e. the features layer with 6 neurons and the action layer (51 or 31 neurons), a hidden layer with 90 neurons and an output layer with just one neuron that provides the optimal control action.

For what concern the agent options, a learning rate $\alpha = 0.01$, a discount factor $\gamma = 0.99$, an experience buffer with a 10000 length have been set. To guarantee a suitable exploitation-exploration trade off, policy is updated through $\epsilon - greedy$, where at the beginning ϵ is equal to 0.9 and then it is reduced by a decay rate of 0.05 at each new epoch, until the minimum value $\epsilon_{min} = 0.1$ is achieved.

The target network Q_{target} is updated ($Q_{target} = Q_{critic}$) with update frequency equal to 4.

At the end of the RL phase the agent is always saved, so that it can be trained again in the next IRL iteration.

6 Simulations and results

In this chapter all the simulation results will be shown either for the Adaptive Cruise Control (ACC) and the Lane keeping Control (LKC), with the aid of some meaningful graphs acquired during the project development. All the quantities have been plotted with respect to time t in s (x -axis).

6.1 ACC simulations

In this section the simulation results of the ACC, implemented through the projection-based algorithm, during different training phases are shown and discussed, analysing the possible reasons that brought to undesired or expected performances.

As it has been extensively clarified in chapter 5, the agent always performs actions with frequency $f = 10 \text{ Hz}$ in according to the sampling time $T_s = 0.1s$. The simulations finishing time T_f has been set equal to $30 s$ in order to guarantee that the number of samples collected by the sensors inside the driving scenario were equal to the number of samples composing the human driver trajectories. Thus, it has been possible to make a coherent comparison between the two datasets.

All the graphs and results proposed are obtained from simulations in which the agent was tested inside the driving scenario constructed in the Driver Scenario Designer, as explained in chapter 5.

The ACC scenario is made of a straight lane with the Ego Vehicle (EV) posed at a distance of 40 meters from the Leading Vehicle (LV), this latter travels at variable speed between 12 and $17 \text{ m} / s^2$.

In *figure 6.1* the waveform representing the actions selected by the agent after the first IRL iteration of the algorithm has been reported. Here, the agent has been designed starting from

$\mu_{E,1_ACC}$, namely the μ_E vector computed through data related to the first driver that showed a sport driving style, and with a reward function R that implements eq. (3.1).

The agent training has been performed during the Reinforcement Learning (RL) phase, selecting a number of episodes (*maxepisodes*) equal to 25.

From the graph it can be noticed that a single IRL iteration is not enough to let the agent behaves correctly, in fact the provided acceleration values are either $+2 \text{ m/s}^2$ or -3 m/s^2 .

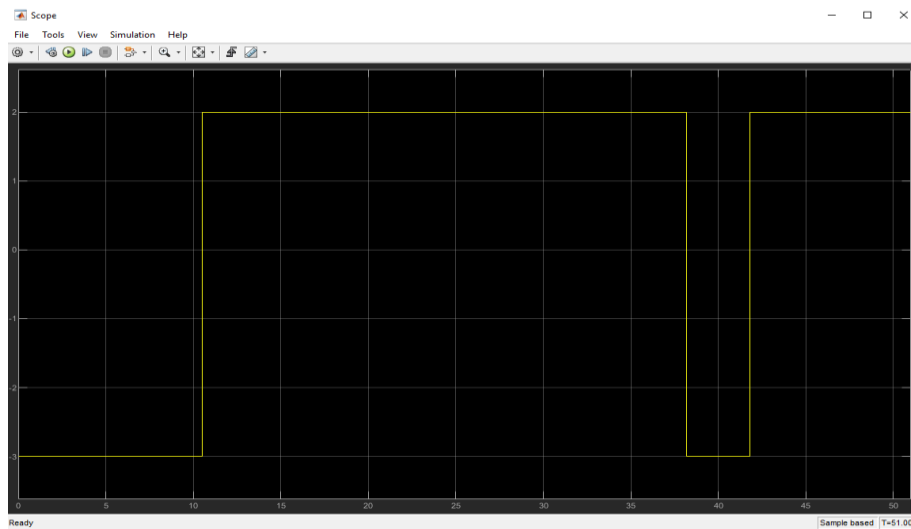


Figure 6.1: agent acceleration profile. On the y-axis acceleration values in m/s^2 are reported, on the x-axis time in s.

In table 6.1 the values assumed by the vector of weights ω and the feature expectation vector μ_{sim} have been reported for each feature at different IRL iteration.

IRL iteration		v_x	d_{rel}	v_{rel}	$\Phi_{collision}$	$\Phi_{reverse}$	$\Phi_{safety_distance}$
1	μ_{sim}	0.5677	0.1589	0.5473	0	0	1
	ω	-0.4078	-0.7629	0.015	-1	0	-1
	$\mu_{E,1}$	0.2890	0.0418	0.5527	0	0	0
2	μ_{sim}	0.5552	0.2700	0.5605	0	0	0
	ω	-0.2930	0.0001	0.0130	0.0326	0	-1
	$\mu_{E,1}$	0.2890	0.0418	0.5527	0	0	0
5	μ_{sim}	0.5841	0.0865	0.5403	0	1	1
	ω	-0.2896	-0.0657	0.0084	-0.0003	-0.0007	-0.016
	$\mu_{E,1}$	0.2890	0.0418	0.5527	0	0	0

Table 6.1: Features expectation vector and weights related to three different iteration during a simulation of 25 RL episodes

By analysing the values in the table, it is possible to observe that at the end of the 1st iteration the agent does neither collide with the preceding vehicle nor go in reverse (negative speed), given that the two features $\varphi_{collision}$ and $\varphi_{reverse}$ are equal to 0. On the contrary, $\varphi_{safety_distance}$ equal to 0 indicates that the ego vehicle overcame the imposed safety distance. Weights ω related to the 4th and 6th features are negative as expected, since weights must guarantee that misbehaviours represented by those features are adequately penalized, whereas at the end of the 2nd iteration $\omega_{collision}$ assumes wrongly a negative value. After the 5th IRL iteration it is easy to understand that the agent learned a bad policy since the last two features are equal to 1, even if the weights are still negative. The action waveform is reported in *figure 6.2* showing that $\varphi_{collision} = 0$ does not necessarily define a good policy, indeed in this case no collision happened just because the agent decelerated during all the simulation inverting the direction of travel.



Figure 6.2: bad acceleration profile [m / s^2], the agent learns to go in reverse

Similar results have been obtained by using $\mu_{E,2_ACC}$ as input of the algorithm.

In order to avoid these situations, the reward function has been modified in both cases, by associating additional penalties (see chapter 5) to that provided by the corresponding features.

Graph in *figure 6.3* shows the acceleration profile of the agent after one IRL iteration.

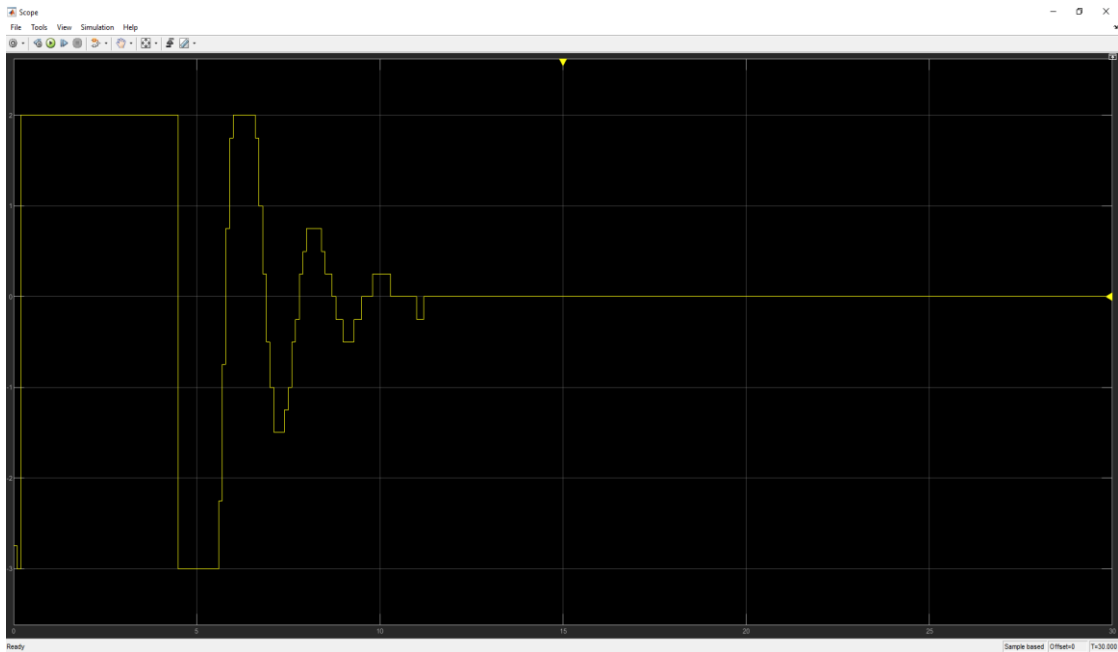


Figure 6.3: acceleration profile $[m / s^2]$ after the 1st IRL iteration within a simulation of 100 RL episodes

The waveform is clearly better than the waveforms obtained from the previous simulations. For the first 10 s of simulation the acceleration oscillates, assuming values between the maximum and the minimum, and then the agent chooses to go at constant speed since the acceleration goes to 0.

Figure 6.4 shows the evolution of the reward during a RL training with 100 episodes.

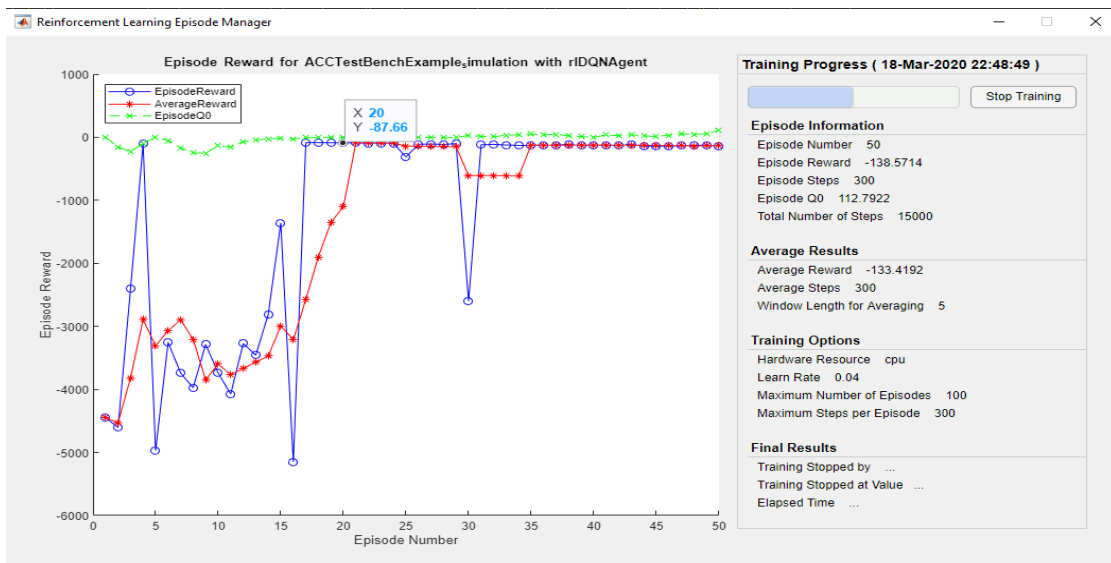


Figure 6.4: Reward profile after 50 episodes during the 1st IRL iteration

Actually, it can be observed that after 50 iterations the reward reaches the convergence, but obviously the constructed reward is not good enough to guarantee acceptable behaviours.

After 4 IRL iteration the results are better (figure 6.5), indeed the acceleration profile shows not too high jerks.

Furthermore, comparing values in table 6.2 other differences between the two simulations can be highlighted.

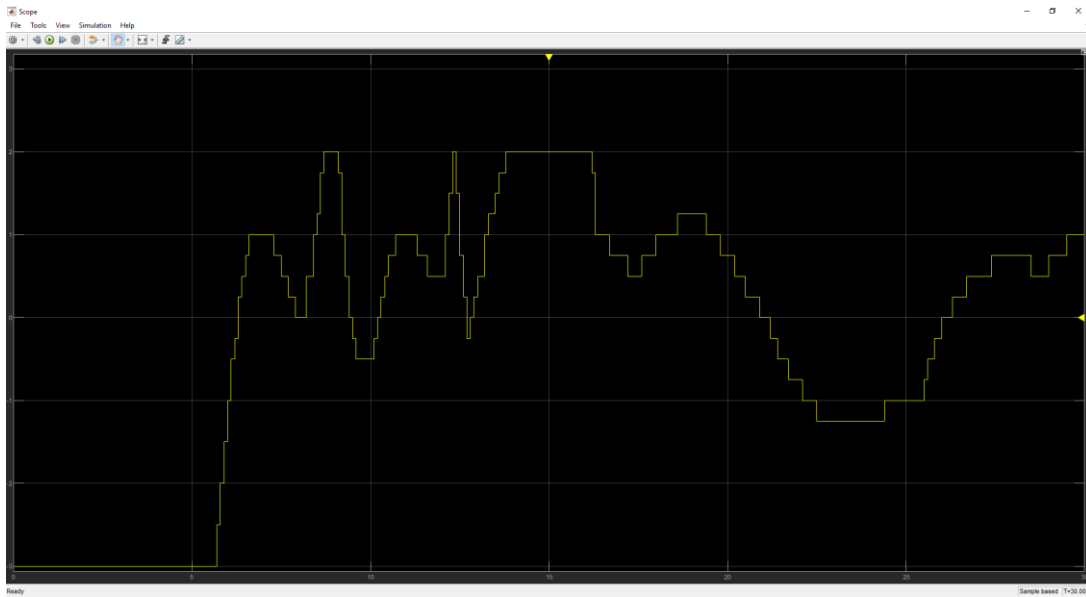


Figure 6.5: acceleration profile [m / s²] after the 4th IRL iteration within a simulation of 100 RL episodes

IRL iteration		v_x	d_{rel}	v_{rel}	$\varphi_{collision}$	$\varphi_{reverse}$	$\varphi_{safety_distance}$
1	μ_{sim}	0.7982	0.8746	0.5388	1	0	1
	ω	-0.2421	0.0127	0.0132	0.0174	0	-1
	$\mu_{E,2}$	0.3429	0.0815	0.5528	0	0	0
4	μ_{sim}	0.5869	0.0814	0.5396	0	0	1
	ω	-0.3539	-0.7232	0.0152	-1	0	-1
	$\mu_{E,2}$	0.3429	0.0815	0.5528	0	0	0

Table 6.2 Comparison between the feature expectation vectors and weights related to the acceleration profiles in figures 6.3, 6.5

In the 1st iteration $\varphi_{collision}$ and $\varphi_{safety_distance}$ are both at 1, while in the 4th iteration only $\varphi_{safety_distance}$ is equal to 1, meaning that crash did not happen but the EV did not respect

the safety distance. By comparing the d_{rel} (in m) obtained in both iterations it is evident that in the 1st iteration the value is very far from d_{rel} of the human driver ($\mu_{E,2}$, table 6.2).

As underlined by graph in figure 6.6, 50 iterations do not always guarantee that the algorithm converges.

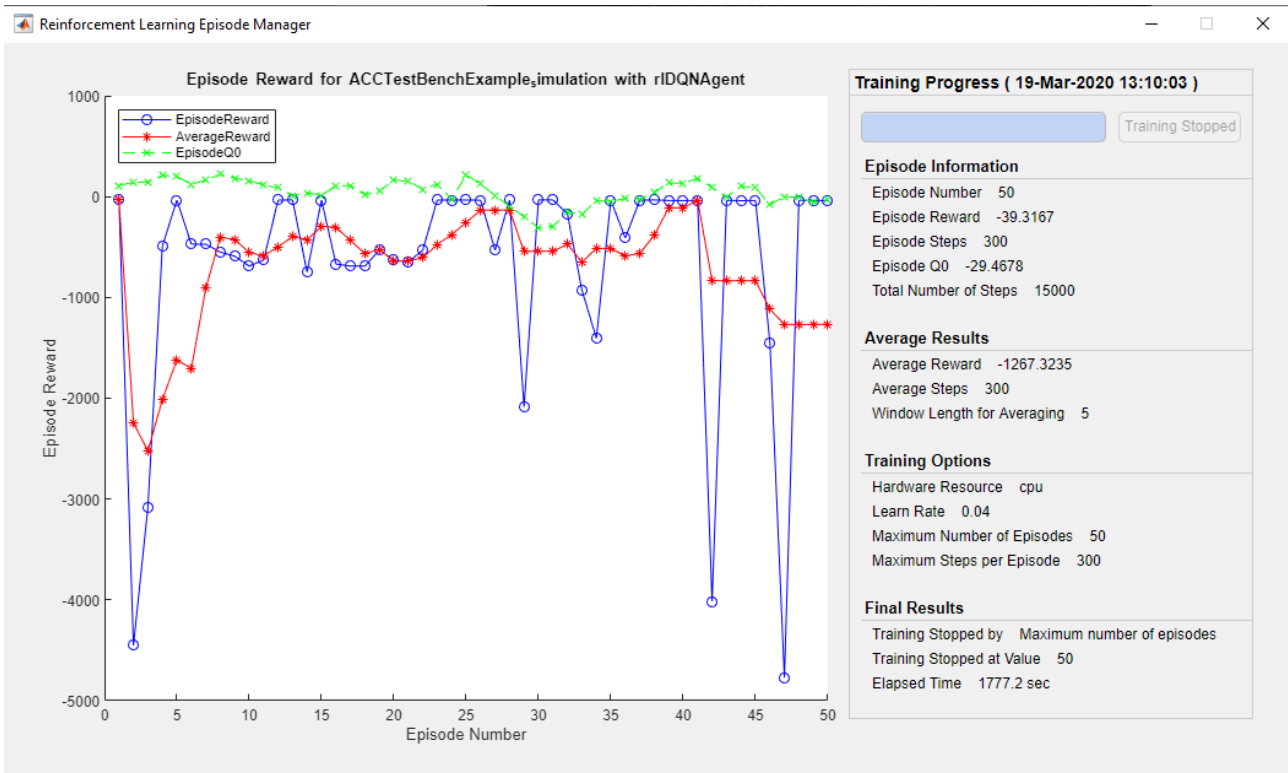


Figure 6.6: Reward profile after 50 episodes during the 4th IRL iteration

Only results relative to those two iterations have been reported because they are the most significant and at the 6th iteration Matlab session was killed due to the “out of memory” error. One of the biggest issues of the algorithm is the fact that a too large space in memory is required to perform a large IRL iterations number, since it is well known that each RL phase needs a very high number of episodes to guarantee that convergence is achieved and this is why *maxsteps* has been set to 50. Furthermore, the complexity of the Deep Q-Network (DQN) complicates even more the problem, since it receives as input an action vector with 21 components and the experience buffer replay needs very high memory space.

Nevertheless, some good policies have been found during the experiments, such as the one shown in *figure 6.7*. A simulation inside the Driving Scenario Designer pointed out that this time the agent was able to follow the leading vehicle without executing any risky situation.

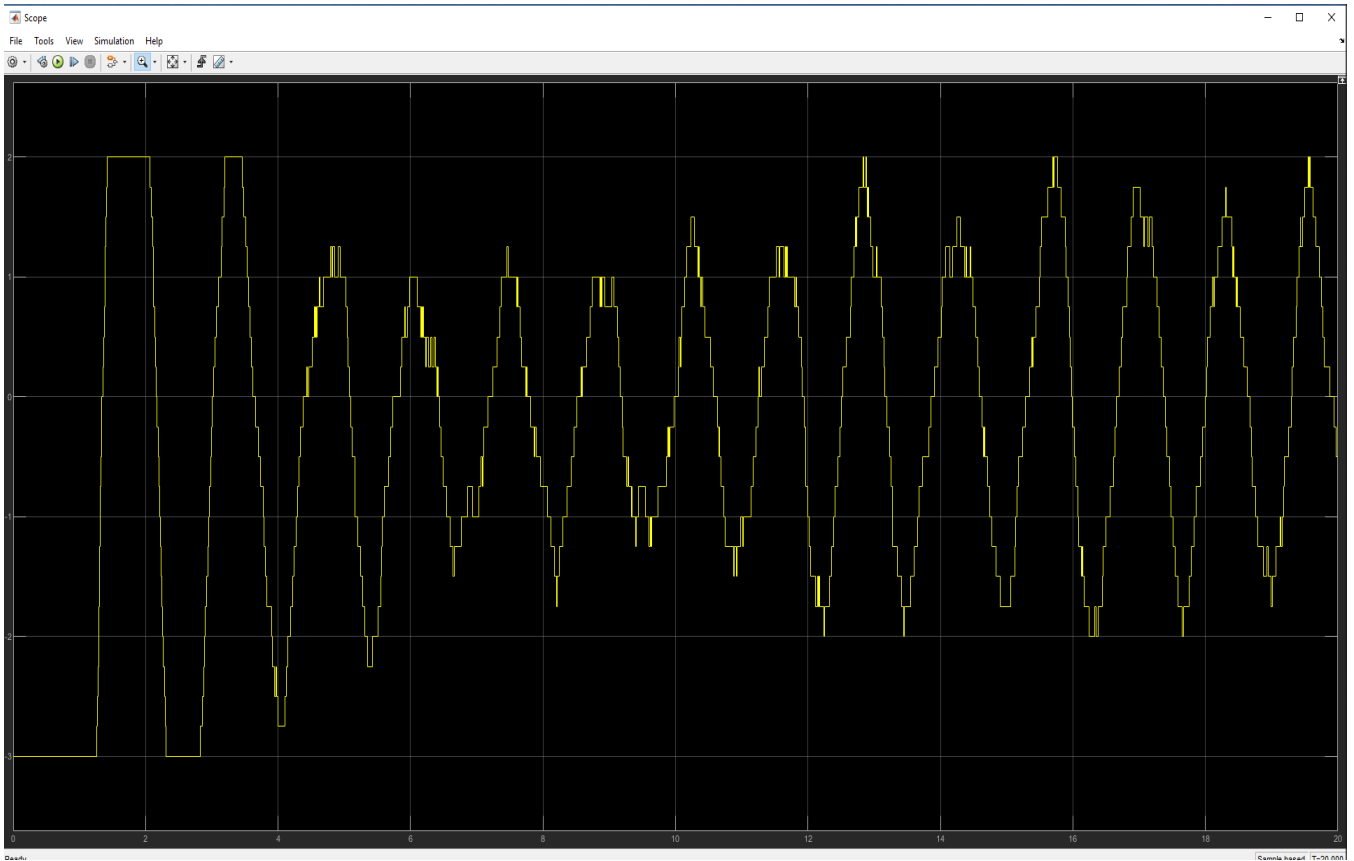


Figure 6.7:

agent acceleration profile [m / s²] after 3 IRL iterations, with weights $\omega = [-0.4078; -0.7629; 0.015; -1; 0; -1]$

Nonetheless, *figure 6.7* shows fast variations in the acceleration values that implicate high jerk, resulting in a very uncomfortable driving. In order to solve this issue, other two features have been added to (5.1), the longitudinal acceleration a_x and the jerk \dot{a}_x . As a consequence, also to the μ_E the same two features have been added. Having modified in this way the reward function, better results have been achieved (*figure 6.8*) indeed a smoother change in the acceleration is appreciable. μ_E and μ_{sim} of this simulation are reported:

$$\mu_E = [0.2886; 0.0428; 0.5521; 0.5986; 0.5002; 0; 0; 0]$$

$$\mu_{sim} = [0.5389; 0.5175; 0.5745; 0.5659; 0.4958; 0; 0; 0]$$

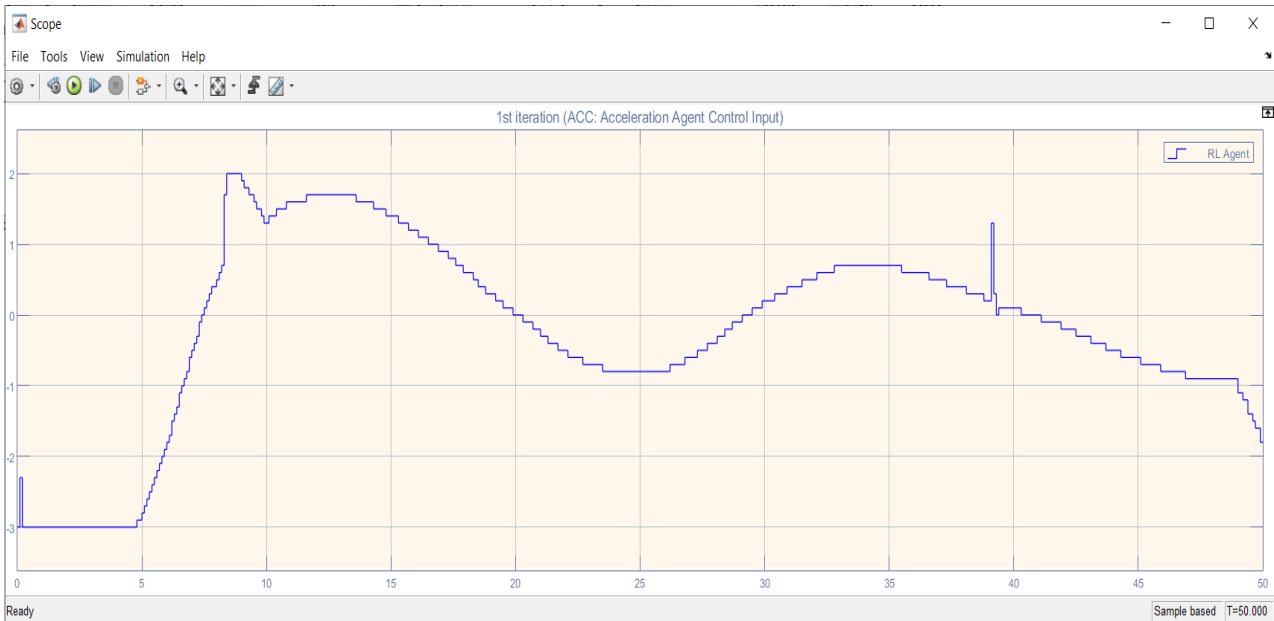


Figure 6.8:

agent acceleration profile $[\frac{m}{s^2}]$ after 3 IRL iterations, with weights $\omega = [-0.2369; -0.4469; -0.0261; -0.0041; 0.0116; -1; -1; -1]$

In addition, the trained agent has been also tested inside different scenarios with higher simulation times and where different initial conditions have been imposed for the preceding vehicle (initial position and initial velocity), proving that it can be able to adapt its behaviour not only to situations presented during the training.

As it is possible to grasp, the IRL problem is very challenging, especially when it is applied to complex environment such as those of autonomous driving. However, in the last chapter “Conclusion and future works” some possible strategies are going to be suggested, in order to improve the current procedure and obtain better results

6.2 LKC Simulation

In this section LKC results are presented, analysing the possible reasons that brought to undesired or expected performances as seen in the ACC case of study.

Initially the same procedure was used as in the case of the ACC but the driving scenario constructed with the Driving Scenario Designer does not allow the RL Agent to learn significantly since the sensors placed on the ego vehicle (EV), once the car exits from the simulation lane, provide insignificant data. One of the possible thought solutions was to stop an episode as soon as the car goes out to leave the lane, but also in this case the training would have taken too long. Due to this problem, a simpler system has been used for the simulations. The observations provided to the RL agent are different compared to the features that make up the Reward function. In fact, the observations passed to the agent are: lateral offset, integral lateral offset, derivate of the lateral offset, yaw error (difference between the estimated yaw $\dot{\psi} = v_x \rho$ and the yaw of the model), integral yaw error and the derivate of the yaw error (Figure 6.10). Furthermore, in the LKC Agent a prebuilt and already tested cost function has been used as the reward as explained in [26], in order to have a working reward function as example for extracting the IRL reward. This latter is always the linear combination $R = (\omega)^T \phi$ of the selected features as clarified in Chapter 5.

Like in the ACC, also in the LKC case the Agent performs actions with frequency $f = 10 \text{ Hz}$, in according to the sampling time $T_s = 0.1$ seconds, and each episode last $T_f = 45 \text{ s}$ as explained in the section 6.1.1 for consistency with the data collected from the expert driver in order to compare properly the μ_E of the human expert driver and the μ_{sim} of the agent simulation.

Then, for this reason a different path has been followed, simplifying the Simulink block scheme eliminating the driving scenario and using the model shown in *figure 6.9*. in which a bicycle model is used in the Ego Vehicle Dynamics block.

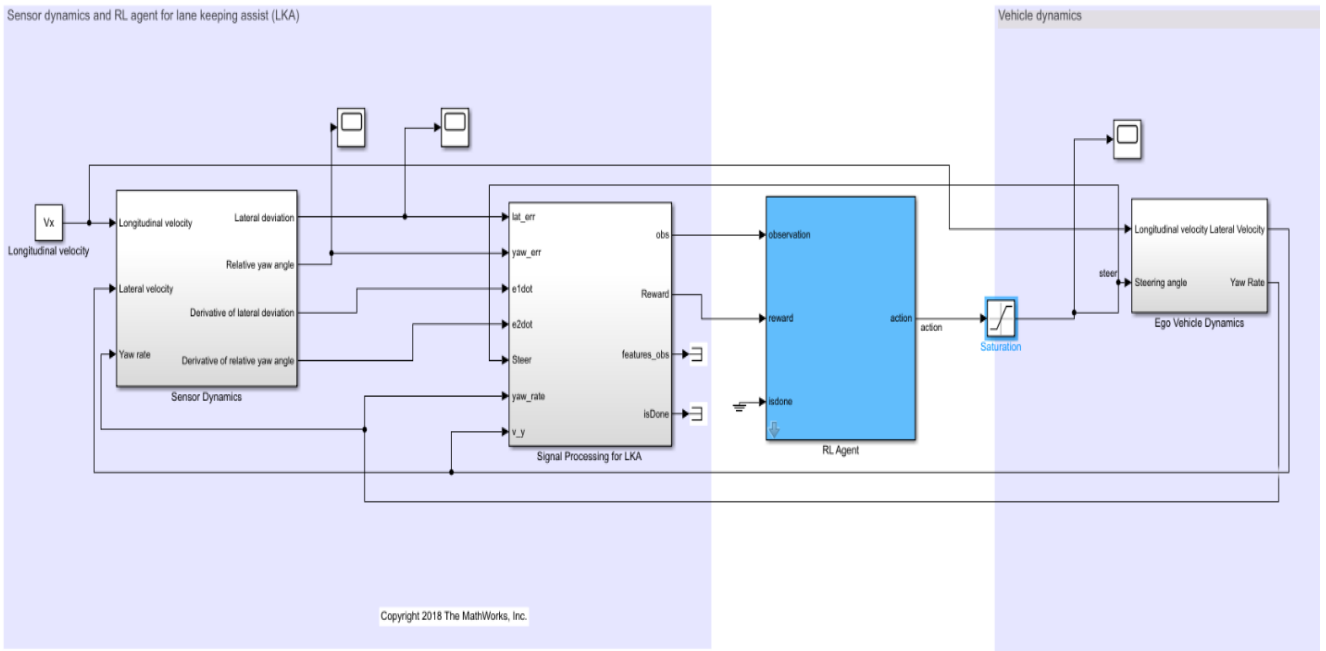


Figure 6.9 complete model with the RL LKC agent

As explained in [26] e_1 as the lateral deviation and e_2 as the relative yaw angle are defined, as show in the *figure 6.10*:

- The steering-angle action signal from the agent to the environment varies between -15 to +15 degree.
- The observations from the environment are the lateral deviation e_1 , relative yaw angle e_2 , their derivatives \dot{e}_1 and \dot{e}_2 , and their integrals $\int e_1$ and $\int e_2$ (*figure 6.11*).
- The simulation terminates when lateral deviation $|e_1| < 1$
- The reward r_t , provided at every time step t , is $r_t = -(10 e_1^2 + 5 e_2^2 + 2\delta_s + 5 \dot{e}_1^2 + 5 \dot{e}_2^2)$ where δ_s is the control input from the previous time step $t-1$.

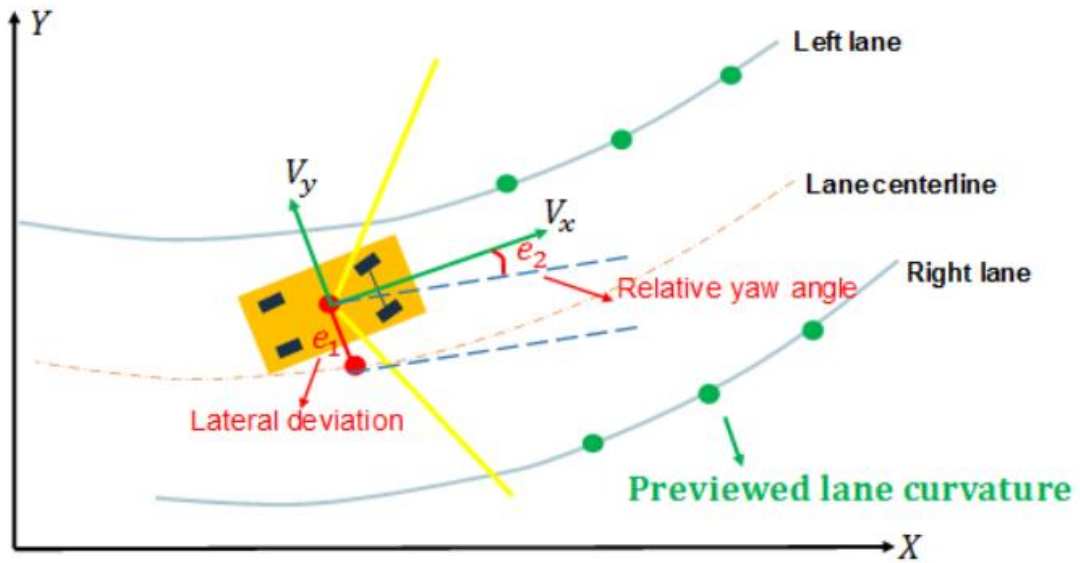


Figure 6.10: Graph representing the e_1 lateral deviation and e_2 the relative yaw angle

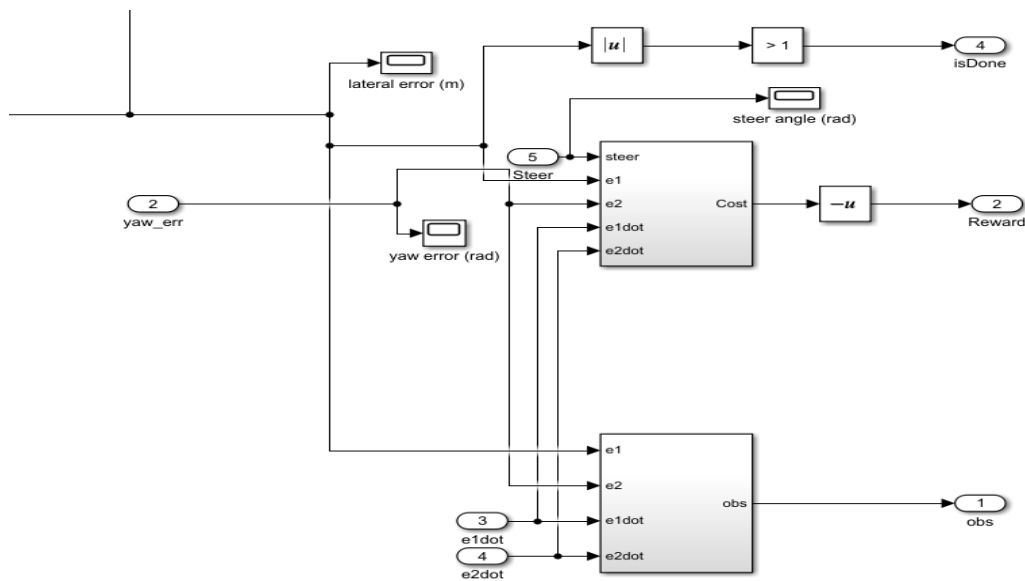


Figure 6.11: LKC agent observations in Signal Processing block

The longitudinal vehicle dynamics is separated from the lateral vehicle dynamics and in addition the longitudinal velocity is assumed to be constant. Moreover, the Sensor Dynamics

block outputs the lateral deviation and relative yaw angle. The dynamics for relative yaw angle are $\dot{e}_2 = \psi - V_x \rho$, where ρ denotes the curvature and ψ the yaw rate. The road curvature from the real expert driver data was used, showing firstly a left curve and then a right one (S curve). The dynamic for lateral deviation is $\dot{e}_1 = V_x e_2 + V_y$ as in [26].

So the idea was to train the RL agent using the observations defined previously, $e_1, e_2, \dot{e}_1, \dot{e}_2, \int e_1$ and $\int e_2$, but still updating the Reward function built as a linear combination of the features $\phi, R = (\omega)^T \phi$ that is indeed different from the observations of the RL Agent except for the lateral distance and the yaw. Thus, the objective is always to update the parameters ω in order to extract a better reward using, as always, the behavior of the driver μ_E as the target. It is worth to notice that in this case only the data relative to the second driver are used since the data of the other driver were corrupted due to road simulation discontinuity produced in the moment of the acquisition. Then the μ_{sim_LKC} is extracted from the simulation as shown in *figure 6.11*.

At the end of the simulation promising results have been found. The expected features vectors of the simulation μ_{sim_LKC} and of the expert human driver $\mu_{E_2_LKC}$ are:

$$\mu_{sim_LKC} = [0.5674; 0.8776; 0.5014; 0.5513; 0.5265; 0]$$

$$\mu_{E_2_LKC} = [0.5523; 0.5359; 0.4928; 0.5707; 0.7756; 0]$$

The components of the two vectors take values fairly close to each other and it is important to underline that the agent never goes out the land markings since the last component of μ_{sim_LKC} is equal to 0.

The parameters ω extracted at the end of the IRL process are:

$$\omega = [-0.4614; -0.4489; -0.4723; -0.5113; -0.6530; -1]$$

The last component is -1 since the reward must heavily penalize the agent when it exceeds the maximum limit on the lateral deviation set at 0.75m (i.e. for highway application).

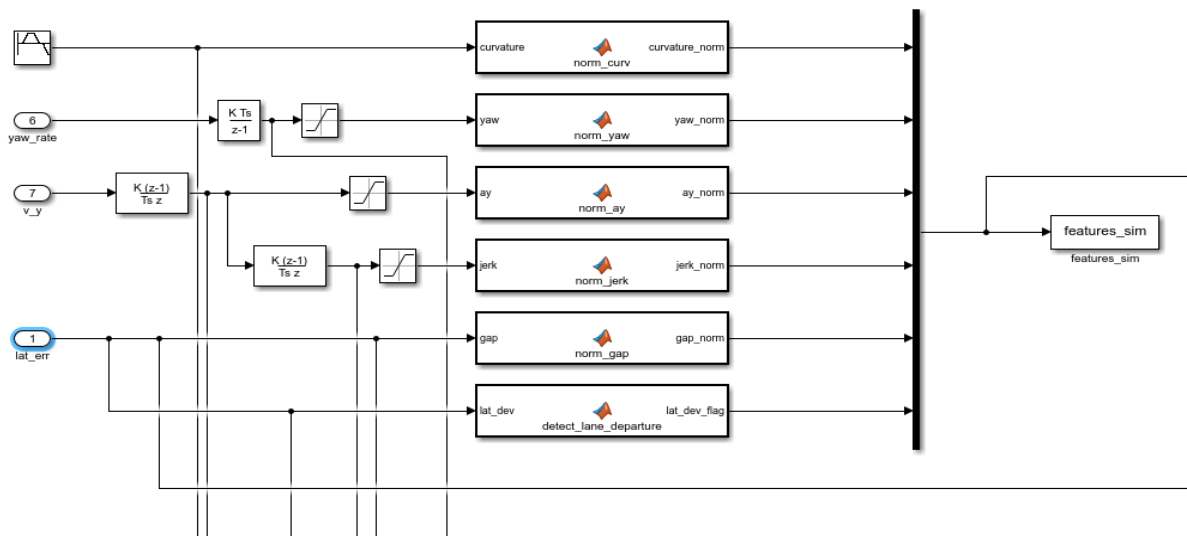


Figure 6.12: Features Extraction for the Reward function

In the *figure 6.13* the steering angle profile can be observed, highlighting that the agent control action follows the data road curvature.

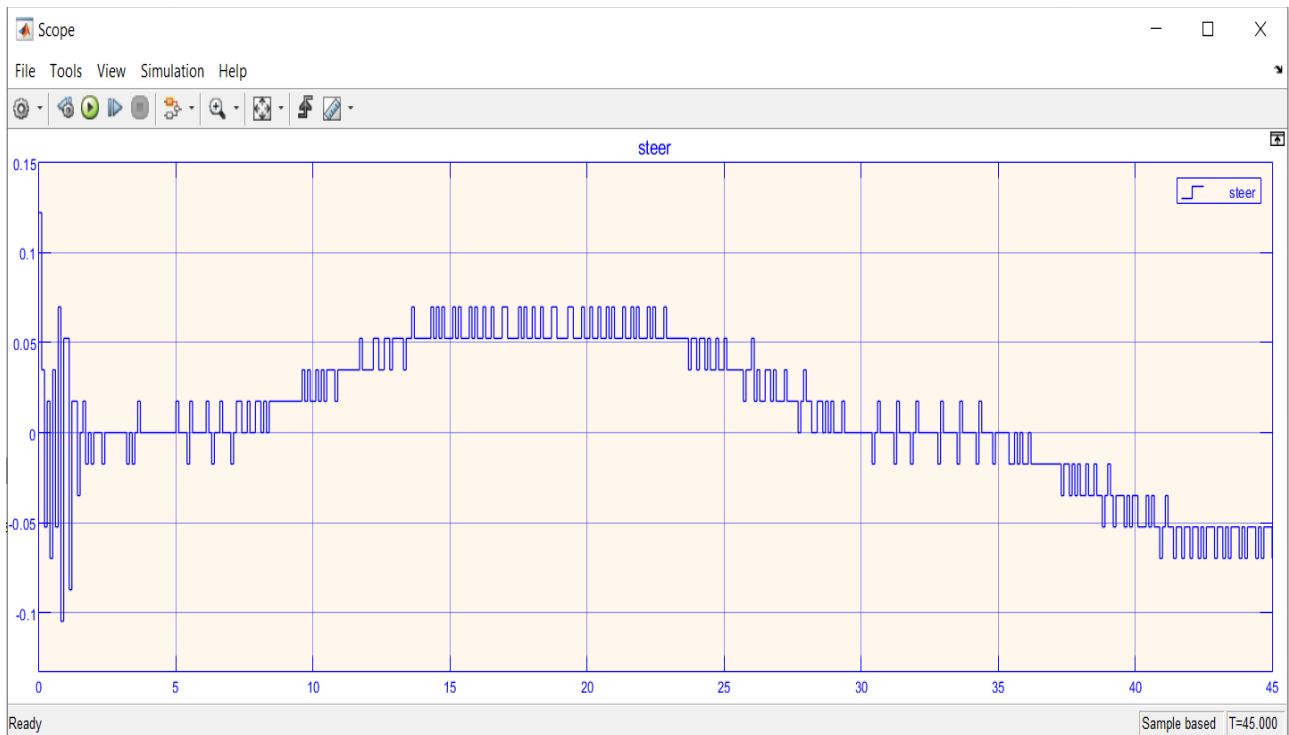


Figure 6.13: steering angle [rad] Agent control action

In figure 6.14 the lateral deviation obtained is shown. As it can be observed, except for the initial transitory, q_{lane} meets the requirements, which have been specified in chapter 4, such that the car never leaves the lane.

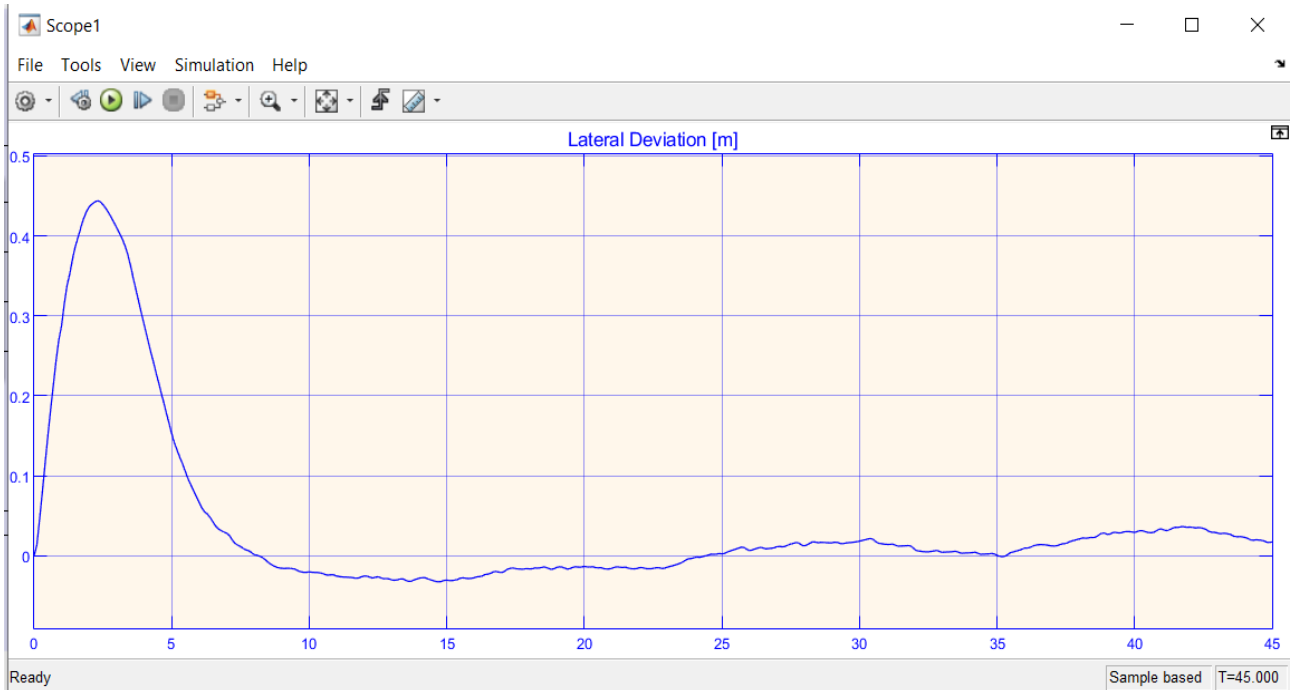


Figure 6.14: Lateral Deviation q_{lane} [m] related to the final simulation

The relative yaw angle is reported in figure 6.15.

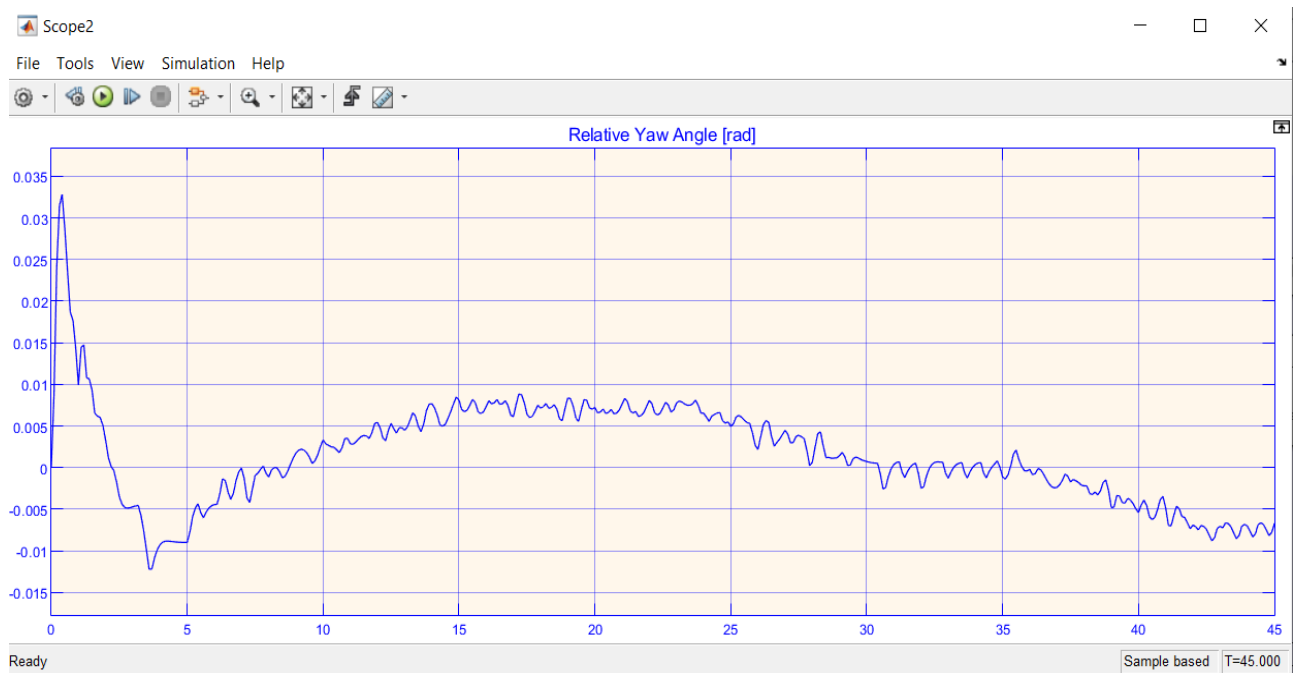


Figure 6.15: relative yaw angle [rad] related to the final simulation

The next step was to newly train the RL agent replacing the defined cost function with the reward function obtained from the previous training.

After only 80 episodes the agent is able to learn a good policy in which it never leaves the lane and never overcoming the lateral deviation threshold as can be seen in the *figures 6.16*.

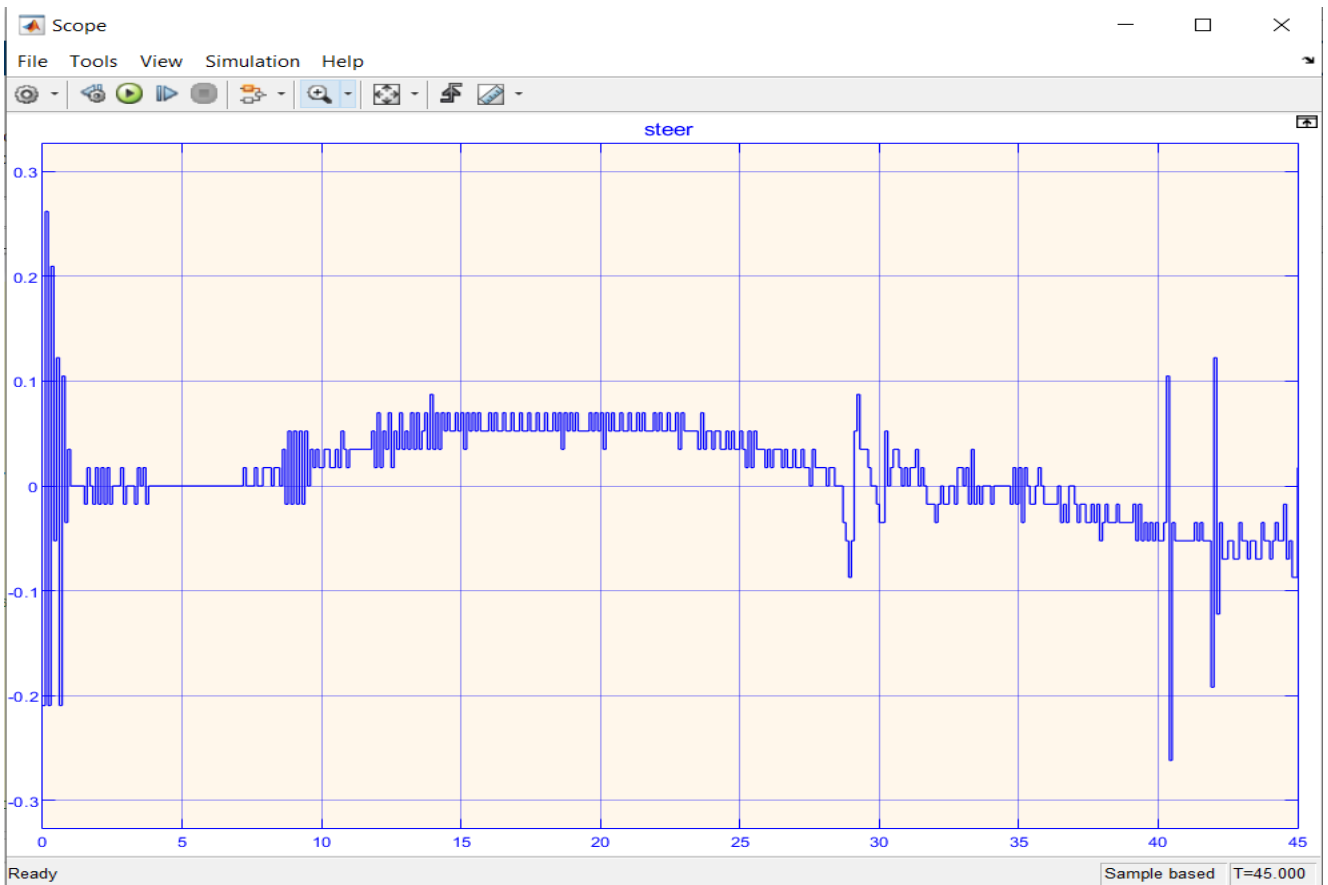


Figure 6.16: agent control steering angle [rad] using the extracted Reward function after 80 episodes

Figure 6.17 shows the lateral deviation. It can be noticed a peak around $T=30$ s due to a curvature road (*figure 6.18*) change that the agent is not able to face yet due to the policy that has found till now.

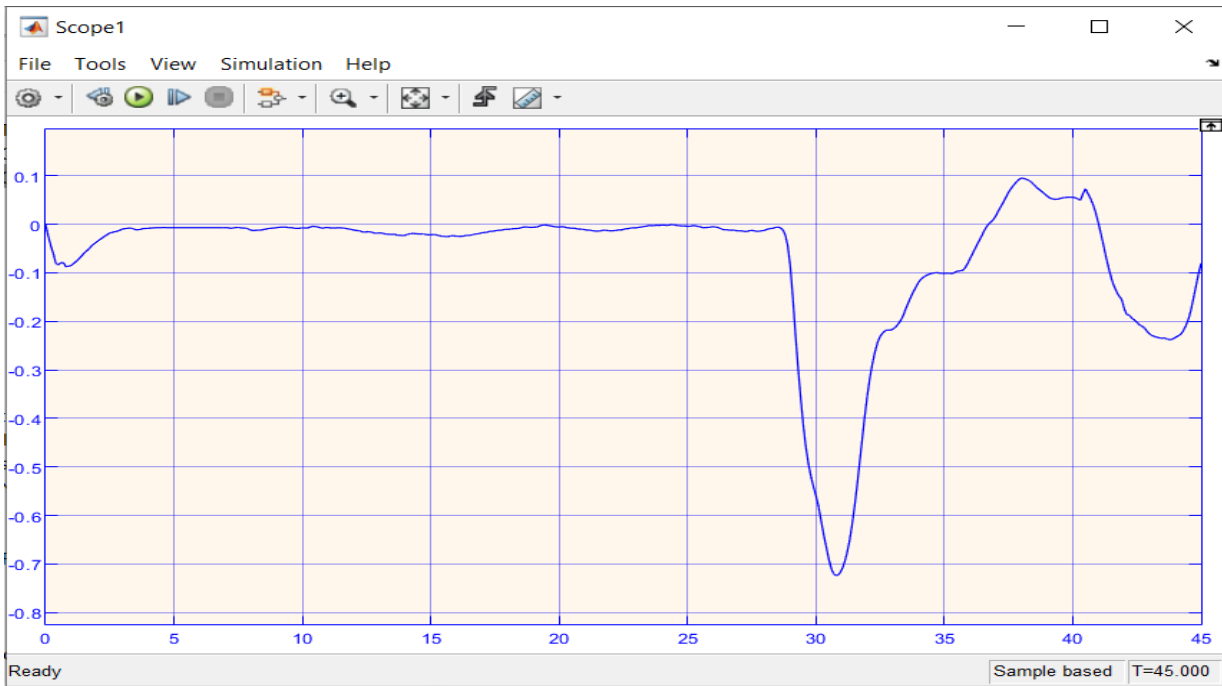


Figure 6.17: lateral deviation q_{lane} [m] using the extracted Reward Function after 80 episodes

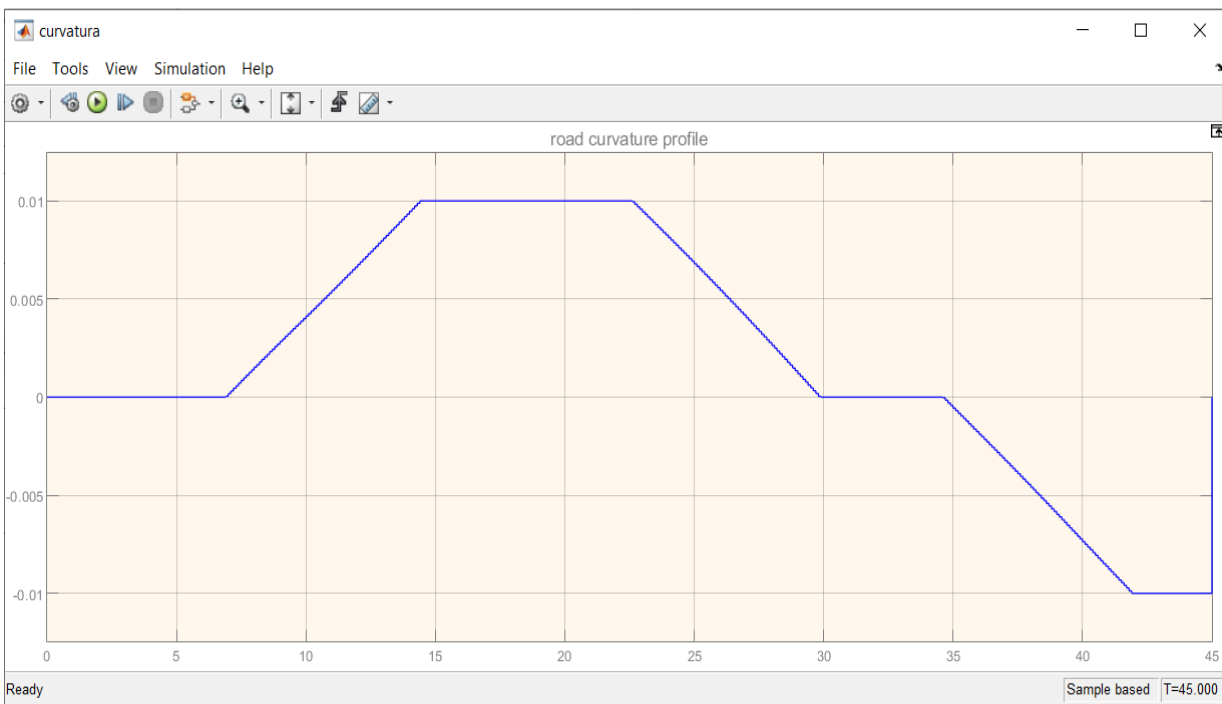


Figure 6.18: curvature road profile [1/m] extracted form data

Several changes have been done to the reward function in order to obtain a smoother agent control action profile (steer) and consequently obtaining a contained lateral acceleration, going to encourage – with the reward thus modified – a behavior as similar as possible to that of a human driver. In *figure 6.19* it can be observed that lateral acceleration, jerk, and also the derivate of the steer control action have been taken into account.

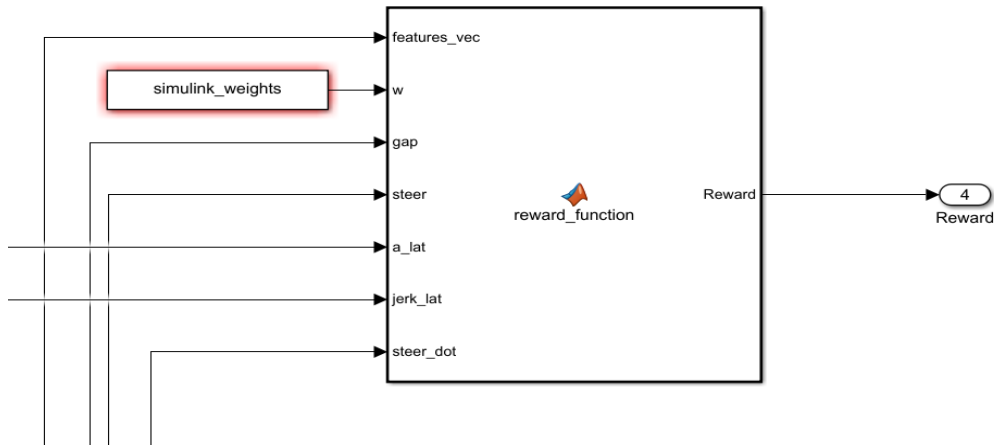


Figure 6.19: Reward function Simulink block

After these adjustments slightly better outcomes have been achieved as highlighted in *figures 6.20, 6.21, 6.22*. In fact, the steer profile in *figure 6.20* is characterized by smoother variations with respect to the control action showed in *figure 6.16* considering that the parameters of the two simulation are the same ($maxepisodes = 80$).

The new expected feature vector of the simulation μ_{sim_LKC} is

$$\mu_{sim_LKC} = [0.5674; 0.8776; 0.5014; 0.5610; 0.5032; 0],$$

while the parameters ω_{LKC} extracted are

$$\omega_{LKC} = [-0.4335; -0.2429; -0.4837; -0.5359; -0.7151; -1].$$

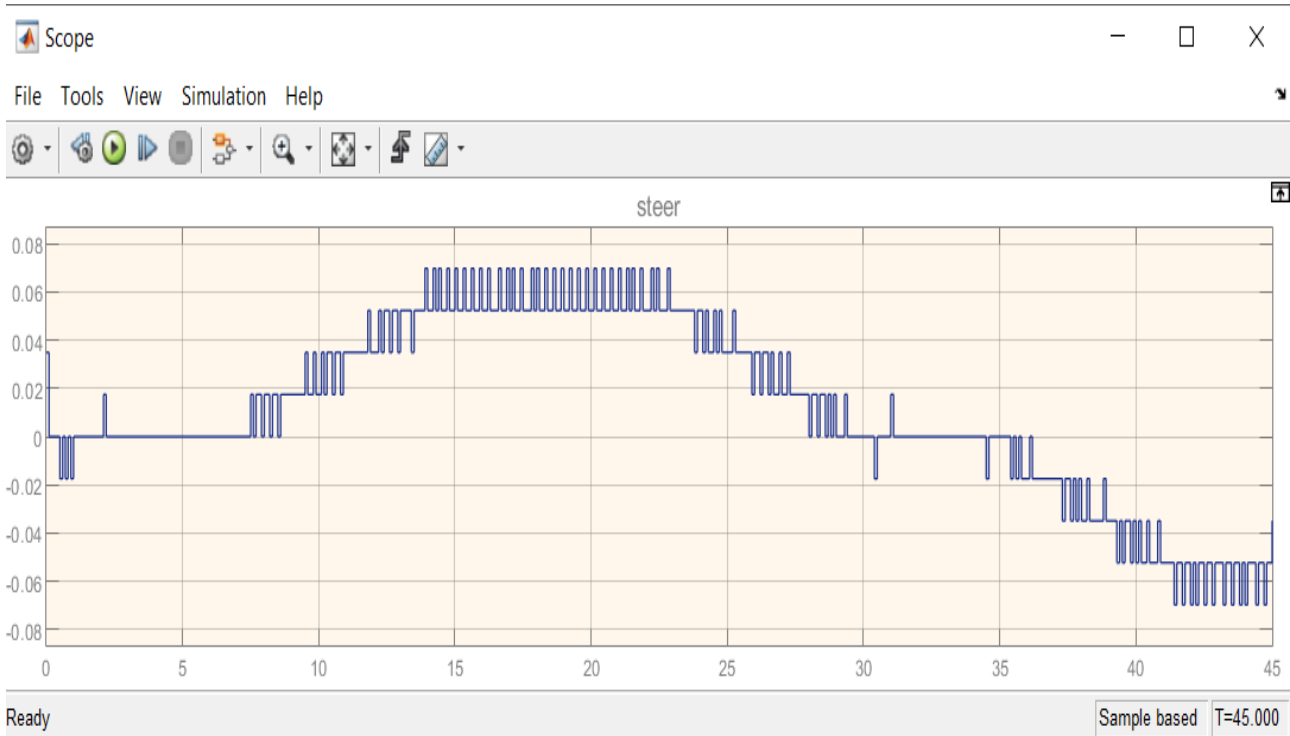


Figure 6.20: agent control steering angle [rad] using the modified Reward after 80 episodes

Figure 6.21 proves that the Ego Vehicle is able to maintain the centreline during the whole simulation with q_{lane} values between +0.08 and -0.036 meters.

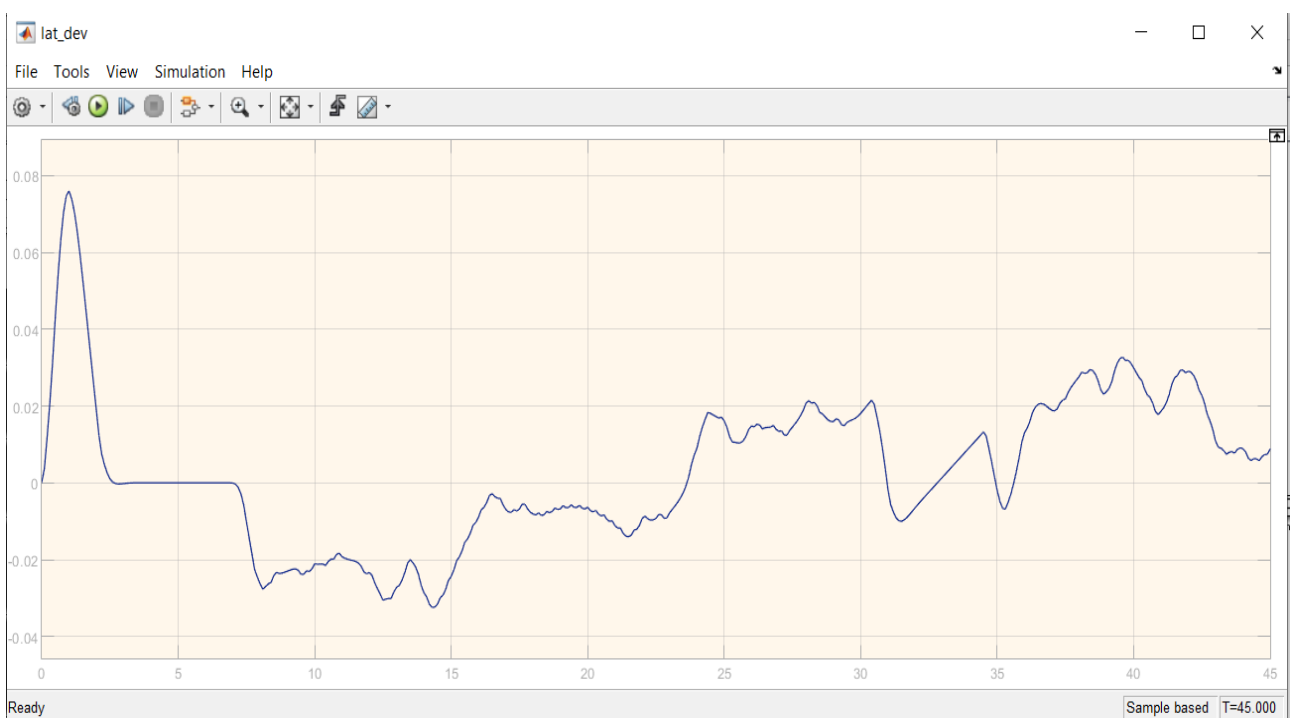


Figure 6.22: Lateral Deviation q_{lane} [m] using the modified Reward after 80 episodes

In the next chapter “Conclusion and future works”, as it has been done for the ACC, some expedients are going to be suggested in order to enhance the proposed methods.

7. Conclusion and future works

As it has been highlighted in the previous chapter through the final results, the approach used in this master thesis project to realize Advanced Driving Assistance Systems with Inverse Reinforcement Learning (IRL) techniques is very complex and requires more time to be corrected and then improved.

In synthesis, the objective of this Master Thesis project is to solve the control problem addressed by an ACC and a LKC through IRL. Since the task is highly complicated, it is not possible to build an a priori reward function which is able to englobe all the possible events that may occur and to give an appropriate feedback. For this reason, the projection-based method has been applied, in which the reward function is defined (hypothesis) as a linear combination of features $R = \omega^T \phi$, where the features vector ϕ contains the relevant parameters related to the task that has to be accomplished.

Initially, for the ACC ϕ_{ACC} is constituted by six components: longitudinal velocity v_x , relative distance d_{rel} , relative velocity v_{rel} and three flags, $\phi_{collision}$ to report collision events, $\phi_{reverse}$ to check the velocity sign (the agent must not go in reverse) and $\phi_{safety_distance}$ to check if the imposed safety distance has been overcome.

Also ϕ_{LKC} is made by six components: the road curvature K_R , the yaw rate $\dot{\Psi}$, the lateral acceleration a_y , the lateral jerk \dot{a}_y , the lateral deviation q_{lane} and $\phi_{lane_departure}$, a flag in order to check if lane departure events occur.

The expert driver behaviours that the agents must emulate have been extracted from DATA provided by Addfor S.p.A.

A large number of simulations has been performed in order to train the two agents, where a single simulation is composed by two phases: IRL phase and RL phase. In particular, at each IRL iteration an RL training is executed.

For what concern ACC, from the first simulations some good policies have been obtained since the agent learned to follow the preceding vehicle without any crashes and respecting the

imposed safety distance. Anyway, from the acceleration profile (agent control action), too fast acceleration variations have been observed, meaning that jerk values did not respect comfort requirements. After these results ϕ_{ACC} , and so the reward function, has been modified by adding two additional features: the longitudinal acceleration a_x and the longitudinal jerk \dot{a}_x .

The new simulations, performed in the same conditions (same number of IRL iterations and RL episodes) and same driving scenario, gave as outcome an acceleration profile characterized by smoother acceleration variation. Nevertheless, the agent behaviour is still far from that of a human driver.

In the LKC case, initially the same procedure has been used. The agent was not able to learn any kind of policy since the task turned out to be more complex than the ACC one. For this reason, the system has been simplified by deleting the driving scenario built in the Driving Scenario Designer and replacing the 3DOF single track model (used to model longitudinal, yaw and lateral dynamics in both applications) with the 2DOF one. The same road curvature extracted from data has been given as input to the model.

Even after these modifications the agent did not learn how to steer properly to stay inside the lane, so a different strategy has been followed. The RL agent has been trained on a predefined cost function, found in literature and used in a very similar problem. In parallel, during the simulation the initial weights to recover the IRL reward function have been extracted. Then, a further training has been performed, inserting the new reward in the IRL algorithm, giving promising results. As for the ACC, agents related to different IRL iterations have been saved and tested. Examining the steering angle profiles and the lateral deviation it has been possible to notice that some of the found policies were acceptable, since the vehicle was able to follow the road curvature, maintaining the centre of the lane.

From the analysis of the collected results it can be highlighted that, even if during some IRL iterations good policies have been found, the agent behaviour is still far from an expert human driver driving style.

Nonetheless, some positive outcomes have been obtained, suggesting that with appropriate modifications the algorithm could work.

For the ACC, first of all what can be done is to construct a more complex reward function including other features (square of the longitudinal velocity, square of the relative velocity and so on), as explained in chapter 6 where the longitudinal jerk has been added to avoid that the agent performs sudden acceleration (a) variations.

Furthermore, as it has been already clarified, one important issue is linked to the use of the Deep Q-Network (DQN) to train the IRL agent. A key point of the DQN is the experience buffer replay, which requires a very large space in memory and indeed it often was the reason for which the algorithm stopped and never reached the convergence [11,12]. In fact, by solving the memory issues, convergence can be achieved by increasing the number of episodes in the RL phase. From the implementation point of view, it has even been proved that when the action input vector is constituted by many components, the DQN could not converge. Therefore, another strategy is to build the agent with a Deep Deterministic Policy Gradient (DDPG), a Q-learning algorithm that is specifically adapted to environment characterized by continuous action space. In this way there would be no need to select a discrete number of actions (acceleration for the ACC and steering angle for the LKC) making the agent behaviour closer to the human driver policy. Nevertheless, as for others Q-learning algorithms, convergence can be reached only stabilizing the training through the experience buffer replay. Thus, to reduce the amount of memory occupied by the buffer a possible solution is to fill it only with the last experiences, which are for sure the most significant ones since they are related to the last weight updates and so contain the most recent information.

For what concerns the LKC, a possible path to follow in order to enhance this method is to make a more complex reward function inserting several different features, as explained for the ACC. In this specific case using for instance the yaw error, the yaw integral error, the yaw error derivative, the integral lane gap and its derivative similarly to the components used for the cost function. In this way, putting in the reward all the significative features that brings the meaningful information about the goals that has to be reached, through this IRL process the logic of the driver can be obtained. Thus, designing a control agent capable to generalize in every type of scenario, even in new and unseen ones. After these improvements the same IRL process implemented for the ACC can be followed. One of the main problems is that the complete steering wheel angle in real applications varies from -720 to $+720$ degrees, so assuming a resolution of 5 degrees the agent has to choose 1 among 289 available actions at

each step. Hence, since the actions number is too high, a possible solution can be again to create a DDPG agent in order to have no limits in the action space.

If in the future, actualizing these strategies, the desired results will be obtained, one way to evaluate the goodness/badness of the two IRL ADAS is to compare them with the same systems realized with already known techniques, i.e. Proportional Integral Derivative control (PID) or Model Predictive Control (MPC).

It can be interesting to make this comparison performing some statistical measurements.

For instance, an idea is to use the Gaussian Mixture Model (GMM) to create three different distribution, the first representing data coming from the IRL agent, the second related to the standard ACC and LKC and the third to describe data collected by expert human drivers.

Summarizing the obtained results are not optimal, anyway they prove that it is worth to examine more in depth and improve these strategies.

If the experiment is successful in the long run, the IRL agent distributions will be much more similar to the expert driver one rather than the classical ADAS distributions.

List of figures

Chapter 1

Figure 1.1: Learning problem process.....	7
Figure 1.2: Linear Regression	9
Figure 1.3: Regressor line and residual.....	10
Figure 1.4: Gradient Descent	11
Figure 1.5: Linear and Logistic Regression for classification	13
Figure 1.6: Logistic function.....	14
Figure 1.7: Sigmoid function and decision boundary threshold	15
Figure 1.8: Hyperplane separating two classes with the maximum margin	17
Figure 1.9: Rectifier function (ReLU)	18
Figure 1.10: Biological neuron model in nature and mathematical model.....	20
Figure 1.11: Perceptron structure	21
Figure 1.12: Network diagram of a two-layer neural network.....	22
Figure 1.13: K-means clustering example with 3 centroids.....	27

Chapter 2

Figure 2.1: Schematic representation of the system/agent interaction.....	30
Figure 2.2: episodic tasks	34
Figure 2.3: (a) Bellman equation scheme.....	36
Figure 2.3: (b) Bellman optimality equation scheme.....	36
Figure 2.4: Iterative Policy Evaluation	39
Figure 2.5: Policy Iteration.....	41
Figure 2.6: Generalized Policy Iteration	42
Figure 2.7: from one-step TD to MC	48

Figure 2.8: SARSA	52
Figure 2.9: Reinforcement Learning algorithms sets	55
Figure 2.10: Actor-Critic scheme.....	56

Chapter 3

Figure 3.1: Graphic representing the initial iteration of the max margin method	70
Figure 3.2: First three iterations of the projection version of the algorithm.....	72

Chapter 4

Figure 4.1: ACC standard situation.....	79
Figure 4.2: ACC hierarchical control scheme.....	80
Figure 4.3: (a) velocity control.....	81
Figure 4.3: (b) spacing control	81
Figure 4.4: centerline linear approximation provided by the vision system.....	83
Figure 4.5: Conceptual block scheme of the LKC with vision system.....	84
Figure 4.6: Vehicle axis system (SAE)	86
Figure 4.7: Bicycle model	87
Figure 4.8: Longitudinal forces acting on an inclined road	88
Figure 4.9: Lateral vehicle dynamics	89
Figure 4.10: slip angle.....	91
Figure 4.11: Simulink vehicle body 3DOF with external longitudinal force input	93
Figure 4.12: Simulink vehicle body 3DOF with velocity input.....	94

Chapter 5

Figure 5.1: code to perform trajectories truncation.....	97
Figure 5.2: code to perform features normalization.....	99
Figure 5.3: μ_E computation.....	101
Figure 5.4: algorithm stop condition.....	102

Figure 5.5: Adaptive Cruise Control block scheme	103
Figure 5.6: ACC control block.....	104
Figure 5.7: Lane Keeping Control with MPC	104
Figure 5.8: Simulink scheme realized for the feature normalization in the ACC.....	105
Figure 5.9: modified Reward function	106
Figure 5.10: ACC driving scenario in Driving Scenario Designer	107
Figure 5.11: LKC driving scenario	108

Chapter 6

Figure 6.1: agent acceleration profile.....	111
Figure 6.2: bad acceleration profile, the agent learns to go in reverse	112
Figure 6.3: acceleration profile after the 1 st IRL iteration with 100 RL episodes	113
Figure 6.4: Reward profile after 50 episodes during the 1 st IRL iteration	113
Figure 6.5: acceleration profile after the 4 th IRL iteration with 100 RL episodes	114
Figure 6.6: Reward profile after 50 episodes during the 4 th IRL iteration	115
Figure 6.7: agent acceleration profile after 3 IRL iterations.....	116
Figure 6.8: agent acceleration profile after 3 IRL iterations with the new Reward.....	117
Figure 6.9: complete model with the RL LKC agent.....	119
Figure 6.10: lateral deviation e_1 and the relative yaw angle e_2	120
Figure 6.11: LKC agent observations in Signal Processing block.....	120
Figure 6.12: features extraction for the Reward function	122
Figure 6.13: steering angle	122
Figure 6.14: lateral deviation q_{lane} related to the final simulation	123
Figure 6.15: relative yaw angle related to the final simulation	123
Figure 6.16: steering angle using the extracted Reward function after 80 episodes.....	124
Figure 6.17: lateral deviation using the extracted Reward function after 80 episodes	125
Figure 6.18: curvature road profile extracted from data	125
Figure 6.19: Reward function Simulink block.....	126
Figure 6.20: agent steering angle using the modified Reward after 80 episodes.....	127
Figure 6.21: lateral deviation using the modified Reward after 80 episodes.....	127

References

- [1] R. Bekkerman, M. Bilenko, J. Langford, *Scaling Up Machine Learning*, Cambridge University Press, January 2012.
- [2] Mitchell, T. (1997). *Machine Learning*. McGraw Hill.
- [3] Andrew Ng, Machine Learning (Coursera), Stanford University.
- [4] Hands-On Generative Adversarial Networks with Keras, Rafael Valle.
- [5] The Pennsylvania State University the Graduate School, Learning in extreme conditions: online and active learnine with massive imbalanced and noisy data (Master's Thesis), Seyda Ertekin.
- [6] An Introduction to Statistical Learning by Gareth James, Daniela Witten, Trevor Hastie and Rob Tibshirani.
- [7] Artificial neural networks and self-organization for knowledge extraction (Master's thesis), Larysa Aharkava.
- [8] UCL (University College London) Course on Reinforcement Learning, David Silver.
- [9] Reinforcement Learning: An Introduction Richard S. Sutton and Andrew G. Barto.
- [10] Dissecting Reinforcement Learning, Massimiliano Patacchiola, blog articles: <https://mpatacchiola.github.io/blog/>.
- [11] Abbeel, Pieter, Andrew Y Ng. 2004. Apprenticeship learning via inverse reinforcement learning. Proceedings of the twenty-first international conference on Machine learning ACM, 1.
- [12] Apprenticeship Learning and Reinforcement Learning with application to robotic control, Pieter Abbeel,(Doctor of Philosophy) Stanford University.
- [13] Brian D Ziebart. Modeling purposeful adaptive behavior with the principle of maximum causal entropy. Carnegie Mellon University, 2010.

- [14] Maximum Entropy Inverse Reinforcement Learning Brian D. Ziebart, Andrew Maas, J. Andrew Bagnell, and Anind K. Dey School of Computer Science Carnegie Mellon University Pittsburgh, PA 15213.
- [15] Brenna D. Argall, Sonia Chernova, Manuela Veloso, and Brett Browning. A survey of robot learning from demonstration. *Robotics and Autonomous Systems*.
- [16] Andrew Y. Ng, Stuart J. Russell, et al. Algorithms for inverse reinforcement learning.
- [17] Jonathan Ho and Stefano Ermon. Generative adversarial imitation learning. In *NIPS*, pages 4565–4573, 2016.
- [18] Edwin T. Jaynes. Information theory and statistical mechanics. *Physical review*, 106(4):620, 1957.
- [19] Brian D. Ziebart, Andrew L. Maas, J. Andrew Bagnell, and Anind K. Dey. Maximum entropy inverse reinforcement learning. In *AAAI*, volume 8, pages 1433–1438. Chicago, IL, USA, 2008.
- [20] “Levels-of-driving-automation” standard for self-driving vehicles: <https://www.sae.org>
- [21] Technologies for Autonomous Vehicle Course, Politecnico di Torino Massimo Canale.
- [22] Experimental results in vision-based lane keeping for highway vehicles, V. Cerone, A. Chinu, D. Regruto.
- [23] U. Kiencke and C. Nielsen. *Automotive Control Systems*. Springer-Verlag, 2000.
- [24] Rajesh Rajamani. *Vehicle Dynamics and Control*. Springer, 2006.
- [25] R. Marino, “A Nested PID Steering Control for Lane Keeping in Vision Based Autonomous Vehicles,” 2009 American Control Conference Hyatt Regency Riverfront, St. Louis, MO, USA June 10-12, 2009.
- [26] <https://it.mathworks.com/help/reinforcement-learning/ug/train-dqn-agent-for-lane-keeping-assist.html>.