

POLITECNICO DI TORINO

Master of Science in Automotive Engineering

Master Thesis

**Development of a Reinforcement Learning based
Energy Management System (EMS) for a plug-in
Hybrid Electric Vehicle**



**Politecnico
di Torino**

Supervisors:

Prof. Federico MILLO

Prof. Luciano ROLANDO

Luca PULVIRENTI

Luigi TRESCA

External supervisor:

Prof. Moein MEHRTASH

Candidate:

Nicola CAMPANELLI

October 2023

To my parents

Abstract

As the world strives in the pursuit of carbon neutrality, Hybrid Electric Vehicles (HEVs) raised as one of the most promising solutions to reduce greenhouse gas emissions. In this context, an effective and reliable Energy Management System (EMS) becomes essential to fully exploit the hybrid potential since the presence of two energy sources (i.e., the fuel tank and the chemical battery) introduces an additional degree of freedom which has to be properly handled. To address this control problem, in the wide portfolio of available techniques, Artificial Intelligence (AI) represents nowadays the cutting-edge solution. Reinforcement Learning (RL) seems to be one of the most promising thanks to its peculiar structure, where an agent interacts directly with an environment, making decisions and receiving feedback in the form of rewards. In this study, a new Soft Actor-Critic (SAC) agent, which exploits a stochastic policy, was implemented and trained on a digital twin of a state-of-the-art diesel Plug-in Hybrid Electric Vehicle (PHEV) available on the European market. The training procedure was performed over the Worldwide harmonized Light-duty vehicles Test Cycle (WLTC) in order to consider driving conditions representative of real vehicle usage. Finally, a sensitivity analysis was performed on different rewards by testing multiple penalty functions aimed at enhancing the fuel economy while guaranteeing the battery charge sustainability. The simulation platform was built in the Python environment thanks to its flexibility and robust support for implementing and assessing the SAC agent performance compared to the benchmark set by Dynamic Programming (DP). The SAC was able to achieve energy consumption levels close to the DP results, with differences lower than 10% but with the great additional benefit of being suitable for online applications.

Contents

List of Figures	xi
List of Tables	xvii
1 Introduction	1
2 HEVs at a glance	5
2.1 Introduction	5
2.2 HEVs architecture	6
2.2.1 EM position in parallel configuration	10
2.3 Current market trends	12
3 Reinforcement Learning fundamentals	15
3.1 Introduction	15
3.2 Markov Decision Process	19
3.2.1 Optimal policy evaluation	24
3.2.2 Sample-based models: Monte Carlo	27
3.2.3 Off-policy algorithms	29
3.2.4 Temporal Difference algorithm	31
3.3 Q-learning algorithms	34
3.3.1 Overview on models	35
3.3.2 Function approximators	36

3.4	Policy gradient methods	38
3.4.1	Stochastic gradient descent	40
3.5	Deep Reinforcement Learning	43
3.6	DDPG, TD3 and SAC	52
4	Energy Management System	57
4.1	Introduction	57
4.2	Conventional strategies	59
4.2.1	Rule-based strategies	59
4.2.2	Optimization-Based strategies	61
4.3	Current AI-based control strategies	67
5	Test case	71
5.1	Vehicle specification	71
5.2	Mission profiles	74
5.3	Modelling approach	75
5.4	Simulation tools	81
5.5	Model validation	85
6	EMS controller design	93
6.1	Agent selection	93
6.2	Implementation	94
6.3	Training	99
6.4	Neural Network manual tuning	102
6.5	Reward	111
7	Results	115
7.1	Comparative analysis of the rewards	115
7.2	Agent performance assessment	130
7.2.1	Agent testing on WLTC	131

7.2.2	Generalization over different mission profiles	134
8	Conclusions and Future Work	141
Appendix A	Python Code	147
A.1	Engine Class (extract)	147
A.2	Reset Method	149
A.3	Step Method	150
References		155

List of Figures

2.1	HEV series architecture [1].	6
2.2	Series hybridization ratio [2].	7
2.3	HEV parallel architecture [1].	8
2.4	Parallel hybridization ratio [2].	8
2.5	Spectrum of vehicle electrification levels [3].	9
2.6	HEV complex architecture [4].	10
2.7	Electric Motor positions in a HEV [5].	11
3.1	The agent–environment interaction in a Markov decision process [6].	19
3.2	The "dance" of policy and value is visualized as bouncing back and forth between one line, where the value function is accurate, and another where the policy is greedy [6].	27
3.3	Monte Carlo pseudo code [6]	28
3.4	Neural Network layout [6].	44
3.5	Neural Network notation[6].	45
3.6	Neural Network stacking[6].	46
3.7	Neural Network action generalization[6].	46
3.8	Exploration with action values[6].	47
3.9	Parametrized policy and function approximation[6].	48
3.10	Actor and critic.	51
3.11	Actor and critic loop[6].	52

3.12	How the stochastic policy is created [7].	55
4.1	The role of energy management system in a hybrid electric vehicle [1].	58
4.2	Control loop on board[2].	58
4.3	An example of rule-based control[3]	60
4.4	Evolution of EMSs from 1993 to 2018 [8].	68
5.1	Powertrain layout: a diesel engine is connected through an auxiliary clutch to an EM. Both the ICE and the EM are connected to the transmission by means of a torque converter.	71
5.2	Forces acting on a vehicle [3].	76
5.3	Information flow in a forward simulator [9].	78
5.4	Information flow in a backward kinematic approach [3].	79
5.5	Detailed information flow in a backward kinematic approach [1].	80
5.6	The main vehicle's components are modeled through Python classes.	86
5.7	Dynamic Programming is used as the algorithm to evaluate the optimal control strategy.	86
5.8	Power at the final drive plotted against the speed profile.	87
5.9	Power at the gearbox plotted against the speed profile.	87
5.10	Power at the gearbox where a discrepancy point is highlighted.	88
5.11	Focus on the power at the gearbox where it is clear the difference between the model built in Python and the results of the DP from the MATLAB model.	89
5.12	Focus on the gearbox efficiency.	89
5.13	Visual explanation of the inconsistency between the backward approach and the way the gearbox efficiency has been modeled.	90
5.14	Fuel rate plotted against the speed profile.	90
5.15	Cumulative fuel consumption plotted against the speed profile.	91
5.16	SoC plotted against the speed profile.	91
5.17	A look closer to the SoC trend over the cycle.	92

6.1	One of the logger displayed on Python console during training.	100
6.2	Mean reward trend of different agents visualized through Tensorboard against the number of steps performed.	101
6.3	Table with additional information regarding the parameter plotted for several agents.	101
6.4	Entropy coefficient adopted by the agent during training as a function of the steps performed.	102
6.5	Cumulative mean reward trend with default parameter in all its behavior plotted against the number of steps performed by the agent.	104
6.6	Cumulative mean reward trend over the number of steps with default parameter in a zoomed view to catch the behavior where it will be compared with other models.	104
6.7	Cumulative mean reward trends of an agent with default parameter, i.e., 0,0003, (green line) and another one with learning rate equal to 0,001 (orange line) plotted against the number of steps	106
6.8	Cumulative mean reward trends of an agent with default parameter, i.e., 0.99, (green line) and another one with gamma equal to 0.9 (blue line)	107
6.9	Cumulative mean reward trends of an agent with default parameter, i.e., 0.99, (green line) and another one with gamma equal to 0.999 (yellow line)	107
6.10	Cumulative mean reward trends of an agent with default parameter, i.e., 256, (green line) and another one with a batch size of 512 (cyan line)	108
6.11	Table with additional data regarding the comparison between an agent with default parameter (green circle) and another one with a batch size of 512 (cyan circle)	109
6.12	Cumulative mean reward trends of an agent with default parameter, i.e., 0.005 for τ and 1 for the target update frequency, (green line) and another one with the τ equal to 1 and the target update equal to 1801 (purple line)	110

6.13	Cumulative mean reward trends of an agent with default parameter, i.e., 1, (green line) and another one with the target update equal to 10 (violet line)	110
6.14	Cumulative mean reward trends of two agents with default parameters but one trained in series (green line) while the other one in parallel (cyan line)	111
6.15	SoC boundary lines for Case2 reward.	114
7.1	SoC trend of the SAC agent trained with reward 1 with default parameters: $k_1 = 0$, $k_2 = 200$ and $k_3 = 400$	116
7.2	FC trend of the SAC agent trained with reward 1 with default parameters: $k_1 = 0$, $k_2 = 200$ and $k_3 = 400$	117
7.3	SoC trend of the SAC agent trained with reward 1 with default parameters: $k_1 = 0$, $k_2 = 200$ and $k_3 = 400$; plotted against the speed profile of WLTC.	118
7.4	SoC trend of the SAC agent trained with reward 1 with modified parameters: $k_1 = 0$, $k_2 = 200$ and $k_3 = 200$	119
7.5	FC trend of the SAC agent trained with reward 1 with modified parameters: $k_1 = 0$, $k_2 = 200$ and $k_3 = 200$	120
7.6	SoC trend of the SAC agent trained with reward 1 with modified parameters: $k_1 = 0$, $k_2 = 200$ and $k_3 = 300$	121
7.7	FC trend of the SAC agent trained with reward 1 with modified parameters: $k_1 = 0$, $k_2 = 200$ and $k_3 = 300$	121
7.8	SoC trend of the SAC agent trained with reward 1 with modified parameters: $k_1 = 0$, $k_2 = 200$ and $k_3 = 250$	122
7.9	SoC trend of the SAC agent trained with reward 1 with modified parameters: $k_1 = 0$, $k_2 = 200$ and $k_3 = 275$	123
7.10	FC trend of the SAC agent trained with reward 1 with modified parameters: $k_1 = 0$, $k_2 = 200$ and $k_3 = 275$	123
7.11	SoC trend of the SAC agent trained with reward 2 with default parameters: $k_1 = 0$, $k_2 = 200$ and $k_3 = 400$; compared with the SoC trend of reward 1 with equal parameters.	124

7.12	FC trend of the SAC agent trained with reward 2 with default parameters: $k_1 = 0$, $k_2 = 200$ and $k_3 = 400$; compared with the FC trend of reward 1 with equal parameters.	125
7.13	SoC trend of the SAC agent trained with reward 2 with modified parameters: $k_1 = 0$, $k_2 = 200$ and $k_3 = 200$; compared with the SoC trend of reward 1 with equal parameters.	126
7.14	SoC trend of the SAC agent trained with reward 2 with modified parameters: $k_1 = 0$, $k_2 = 200$ and $k_3 = 300$; compared with the SoC trend of reward 1 with equal parameters.	127
7.15	FC trend of the SAC agent trained with reward 2 with modified parameters: $k_1 = 0$, $k_2 = 200$ and $k_3 = 300$; compared with the FC trend of reward 1 with equal parameters.	127
7.16	SoC trend of the SAC agent trained with reward 2 with modified parameters: $k_1 = 0$, $k_2 = 200$ and $k_3 = 250$; compared with the SoC trend of reward 1 with equal parameters.	128
7.17	SoC trend of the SAC agent trained with reward 2 with modified parameters: $k_1 = 0$, $k_2 = 200$ and $k_3 = 275$; compared with the SoC trend of reward 1 with equal parameters.	129
7.18	Comparison between SAC agent with 2 different rewards with 5 configurations each and DP optimization: trade-off between fuel consumption and final SoC.	130
7.19	SoC behavior during SAC agent training over the WLTC with reward 2 with modified parameters: $k_1 = 0$, $k_2 = 200$ and $k_3 = 300$	131
7.20	Trend of the mean value of the selected best performing reward in all its behavior.	132
7.21	SoC trend performed by the SAC agent trained with the selected reward configuration.	133
7.22	FC trend performed by the SAC agent trained with the selected reward configuration.	133
7.23	SoC trend performed by the SAC agent over the FTP75 standardized cycle plotted against the speed profile.	135
7.24	SoC trend performed by the SAC agent over the FTP75 standardized cycle.	136

7.25 FC trend performed by the SAC agent over the FTP75 standardized cycle. 137

7.26 SoC trend performed by the SAC agent over the HFET standardized cycle, plotted against the speed profile. 138

7.27 SoC trend performed by the SAC agent over the HFET standardized cycle. 138

7.28 FC trend performed by the SAC agent over the HFET standardized cycle. 139

List of Tables

5.1	Vehicle and powertrain main specifications.	72
5.2	Characteristic values of the cycles performed during the experimental campaign.	75
7.1	Dynamic Programming optimal control strategy results over WLTC.	117
7.2	Results comparison between Dynamic Programming optimal control strategy and SAC control strategy on WLTC.	134
7.3	Dynamic Programming optimal control strategy results for the two additional cycles.	135
7.4	Results comparison between Dynamic Programming optimal control strategy and SAC control strategy on FTP75.	137
7.5	Results comparison between Dynamic Programming optimal control strategy and SAC control strategy on HFET.	139

List of Symbols

Acronyms

AECMS	Adaptive Equivalent Consumption Minimization Strategy
AI	Artificial Intelligence
AT	Automatic Transmission
AWD	All-Wheel Drive
BEV	Battery Electric Vehicle
BMEP	Brake Mean Effective Pressure
BMS	Battery Management Systems
BSFC	Brake Specific Fuel Consumption
BSG	Belt Starter Generator
CD	Charge Depleting
CFD	Computational Fluid Dynamics
CNN	Convolution Neural Network
CPU	Central Processing Unit
CS	Charge Sustaining
DDPG	Deep Deterministic Policy Gradient
DNN	Deep Neural Network
DoF	Degree of Freedom
DP	Dynamic Programming
ECMS	Equivalent Consumption Minimization Strategy
ECU	Electronic Control Unit
EM	Electric Machine
EMS	Energy Management System
FEAD	Front-End Accessory Drive
FWD	Front Wheel Drive
GHG	Greenhouse Gas

GPI	General Policy Iteration
HEV	Hybrid Electric Vehicle
HV	High Voltage
ICE	Internal Combustion Engine
Li-NMC	Li-Ion Nickel Manganese Cobalt
LV	Low Voltage
MC	Monte Carlo
MDP	Markov Decision Process
ML	Machine Learning
MPC	Model Predictive Control
MSVE	Mean Squared Value Error
NEDC	New European Driving Cycle
NN	Neural Network
OEM	Original Equipment Manufacturer
PID	Proportional Integral Derivative
PM	Permanent Magnet
PMP	Pontryagin's Minimum Principle
RB	Rule Based
RDE	Real Driving Emissions
ReLU	Rectified Linear Unit
RL	Reinforcement Learning
RNN	Recurrent Neural Network
RT	Real-Time
SAC	Soft Actor-Critic
SoC	State of Charge
SoE	State of Energy
SoH	State of Health
SUV	Sport Utility Vehicle
TC	Torque Converter
TD	Temporal Difference
TD3	Twin-delayed Deep Deterministic policy gradient
TTR	Through The Road
WLTP	Worldwide harmonized Light-duty vehicles Test Procedures
xEV	Electrified Vehicle

Definitions

Chapter 1

CO_2 Carbon Dioxide

Chapter 2

$R_{h,series}$ Series HEV hybridization ratio
 $P_{EL,GEN}$ Electrical power request by generator unit in a series HEV
 P_{EM} Electrical power requested by motor generator
 $R_{h,parallel}$ Parallel HEV hybridization ratio
 P_{ICE} Power request to the internal combustion engine

Chapter 3

q Puntual Action-Value
 a Action taken by the agent in the environment
 r Reward obtained from the action taken in a certain instant
 R_t Cumulative reward at timestep t
 α Step size parameter
 p Joint probability describing the dynamic of the environment
 G_t Return at timestep t
 γ Discount factor
 π Policy followed by the agent
 $v_\pi(s)$ State-value function of state s following policy π
 $q_\pi(s, a)$ Action-value function of state s and action a following policy π
 s State in the environment
 s' Next state in the environment
 x Random variable sampled by a certain policy
 ρ Importance Sampling Ratio
 W Weight used to correct the return
 δ_t Temporal Difference error at timestep t
 w_i Weight used in a linear approximation function

\hat{v}	State-value approximation
$\bar{V}E$	Mean squared value error
$\mu(s)$	Probability distribution to assign a relevance to the weights
f	Example function
U_t	Any estimate of the value can be used
s_i	Neural Network inputs
x_j	Neural Network features
\hat{y}_k	Neural Network outputs
A_{ij}, B_{jk}	Neural Network weights
\hat{q}	Action-value approximation
θ	Parameters of the parametrized policy
h	Action preference function
\mathcal{H}	Entropy function
β	Entropy trade-off factor

Chapter 4

$f(\cdot)$	Function
$p(\cdot)$	Penalty Function
J	Cost Function
J_n^{opt}	Optimal Cost in State n
$H(\cdot)$	Hamiltonian Function
k	Discrete Time Index
\dot{m}_{vl}	Virtual Instantaneous Fuel Consumption
$\dot{m}_{f,eq}$	Equivalent Instantaneous Fuel Consumption
\dot{m}_f	Engine Instantaneous Fuel Consumption
P_{batt}	Battery Power
Q_{LHV}	Fuel Lower Heating Value
s	Equivalence Factor
t	Time
t_0	Initial Time of the Optimization Horizon
t_f	Final Time of the Optimization Horizon
u	Control Variable
g	Cost of a state
u_k	Discrete Control Variable
g_k	Discrete Cost of a state

x	State Variable
x_k	Discrete State Variable
p	Co-state vector
$\phi(\cdot)$	Terminal Cost
*	Relative to the Optimal Solution

Chapter 5

A_f	Vehicle Frontal Area
a	Vehicle Acceleration
C_d	Aerodynamic Drag Coefficient
C_i	Coast-Down Coefficient
c_i	Rolling Resistance Coefficient
c_{roll}	Rolling Resistance Coefficient
F_{aero}	Aerodynamic Resistance
F_{brakes}	Braking Force
F_{grade}	Resistant Force due to Road Slope
$F_{inertia}$	Inertial Force
F_{pwt}	Tractive Force
F_{roll}	Rolling Resistance
g	Gravitational Acceleration
$I_{wh,f}$	Inertia Front Wheels
$I_{wh,r}$	Inertia Rear Wheels
I_{eng}	Inertia Engine
I_{EM}	Inertia Electric Motor
i	Revolutions per Power Stroke
i_{fd}	Final Drive Ratio
i_{gb}	Gearbox Ratio
M_{veh}	Vehicle Mass
M_{eq}	Equivalent Mass
n	Engine Rotational Speed
P_{eng}	Power Requested to the Engine
P_{gb}	Inlet Gearbox Power
r_{wh}	Wheel Radius
v	Vehicle Speed
V	Engine Displacement

w_{wh}	Wheel Speed
w_{gb}	Gearbox Speed
α	Road Slope Angle
η_{gb}	Gearbox Efficiency
ρ_{air}	Air Density

Chapter 1

Introduction

In recent decades, the world has witnessed a growing awareness of the upcoming challenges posed by climate change and the accelerating depletion of natural resources. This heightened consciousness has been propelled by the harsh realities of our changing environment. As a response to these challenges, there has been a remarkable surge in public interest and concern regarding environmental issues. Policymakers across the globe have been engaged in extensive deliberations to establish comprehensive frameworks and international agreements aimed at reversing the alarming trend of increasing Greenhouse Gas (GHG) emissions and mitigating the effects of global warming.

In the European context, the European Commission (EC) has made a strong and unwavering dedication to address these critical issues through the adoption of the European Green Deal. This proposal seeks to maintain the European status of a competitive economy while efficiently handling the available resources in the pursuit of achieving zero net emissions of GHGs by the year 2050 [10]. Within this framework, the European Commission unveiled the 'Fit for 55' regulatory proposals on July 14, 2021, designed to ensure an EU economy-wide reduction of at least 55% in GHG emissions by 2030, compared to 1990 levels. Being the transport sector accountable for a substantial 35% of worldwide energy consumption [11], these proposals include significant measures about vehicle emissions. Among them, the regulatory framework intends to lower the CO_2 emission targets for new passenger cars from -37.5% to an even more ambitious -55%, and for new vans, from -31% to -50% by 2030, again relative to 1990 levels. Furthermore, an exceptional and controversial goal is set for new cars and vans from 2035 onward: a remarkable 100% reduction in CO_2 emissions.

Within the light-duty vehicle sector, electrified mobility solutions featuring Lithium-ion (Li-ion) batteries have emerged as one of the most promising pathways for automakers to align with the evolving legislation. The synergy of an expanding market offering, coupled with incentives and benefits for the purchase of an Electrified Vehicle (xEV) [12], has led to an unprecedented peak in xEV sales across the European market. Remarkably, the number of new xEV registrations witnessed a gradual but steady rise from 2010 to 2020, comprising at the end of that decade 11% of all new registrations. This momentum intensified in 2021, surging to an impressive 18% of newly registered passenger cars [13], and the trend has persevered into 2022 [14]. Similar trends are observable in the two other major global automotive markets, the United States and China. In these regions, the number of new xEV registrations has more than doubled and nearly tripled in 2021, respectively [14].

Even if, among the xEVs, a trendy wave has involved Battery Electric Vehicles (BEVs) whose sales are always rising, there's still plenty of room for improvements in terms of reliability, performance, and cost to be fairly comparable with conventional ICE-based vehicles. The limited range of BEVs, an issue worsened by the long recharging times and by the poor infrastructure, currently makes Hybrid Electric Vehicles (HEVs) and plug-in Hybrid Electric Vehicles (pHEVs) the most suitable solution in the upcoming scenario [15]. Their greatest advantage comes from the on-board combination of an electric and a conventional powertrain. The additional degree of freedom introduced by the presence of two energy sources increases the complexity of energy management and makes the fuel economy highly dependent on the cooperation between the Internal Combustion Engine (ICE) and the Electric Machines (EMs) installed on board. Responsible for skillful control of power actuators and the optimization of their power distribution [16], the Energy Management System (EMS) plays a pivotal role in the exploitation of the high-efficiency potential characterizing HEVs. To this end, simulating the powertrain's behavior thanks to computer-assisted software is largely embraced, and, in these terms, extensive research has been conducted on EMS in recent decades [17]. Several approaches have been investigated, and significant advancements have been made. However, there are still various aspects to be improved and in particular, the current emphasis is directed toward the development of real-time implementable and optimal strategies [8].

Amidst the so-called "Intelligence Revolution" that is reshaping our society [18], Artificial Intelligence (AI) has been proven to be a game changer also in the

HEV's field [8]. The substantial growth in Central Processing Unit (CPU) processing capabilities, coupled with the capacity to leverage cloud-based computing, has fueled a growing interest in harnessing Machine Learning (ML) techniques to formulate energy management strategies for HEVs. Among the main ML branches, the one denoted as Reinforcement Learning (RL) seems to represent a promising solution to address high-complexity tasks such as the energy management of a HEV.

The motivation for this thesis project arises in the framework just described to model a computationally inexpensive virtual test rig and implement a new RL agent as the algorithm in charge of controlling the power allocation. Of particular interest will be the assessment of the performance of the selected agent both in terms of training and testing. The proposed methodology does not claim to provide a ready-to-go solution for the EMS of a vehicle but rather to carry out an analysis of the effectiveness of a new cutting-edge tool with a simple but reliable simulation platform.

The remainder of the thesis is structured as follows:

- Chapter 2 provides a glance at the most spread HEV layouts and technologies, pointing out their main features which characterize advantages and disadvantages;
- Chapter 3 provides a comprehensive overview of the theoretical background, intended to clarify the main concepts of RL through equations and examples, to gain insights about the algorithm selected as the EMS core;
- Chapter 4 introduces the role and main features of energy management in HEVs. It formalizes the problem and presents an overview of the main classifications and their operating principles. Finally, the current and future trends are described;
- Chapter 5 examines the modeling techniques of a HEV in a simulation environment and presents the case study, listing the main vehicle specifications and the mission profiles involved in the project. Finally, two paragraphs are devoted to the explanation of the functioning of the used simulation tool and the validation of the virtual model;
- Chapter 6 goes through the selection, implementation, and training of the RL agent from both the theoretical and technical points of view. Relevant

focus is spent on the calibration of some RL features and the manual tuning of the algorithm hyperparameters;

- Chapter 7 presents the results of the tests performed with the trained RL agent and a performance assessment of the proposed controller against the benchmark set by a global optimization method known in literature;
- Finally, Chapter 8 summarizes the conclusions and future work.

Chapter 2

HEVs at a glance

2.1 Introduction

With growing concerns over environmental pollution and finite fossil fuel resources, several alternatives to comply with the current transportation needs have been explored by the global automotive industry. One of the solutions that gained more popularity thanks to their promising features aimed at improving fuel economy are Hybrid Electric Vehicles (HEVs). Combining a conventional thermal energy source with an electrical one, they can achieve higher efficiency values due to the increased number of degrees of freedom of operating conditions [1] [9]. This kind of vehicle is able to exploit the high energy density of a petroleum fuel through the well-known Internal Combustion Engine (ICE), which usually guarantees a long-range, with the high performance of one or more electric machines relying on a battery which is meant to shift the engine operating points toward higher efficiency regions. This goal is achieved in two different ways: the engine working conditions can be moved by either making the electric motor provide a motor or a brake torque or downsizing the engine, i.e. decreasing the displacement, to make it run at higher loads covering the peak power with the electrical motor. Furthermore, a key advantage of an HEV lies in the capability of recovering the kinetic energy dissipated during braking simply making the electric motor work like a generator converting the mechanical energy into electrical one and storing it in the battery.

2.2 HEVs architecture

A HEV can be realized through different connections between thermal and electrical sources. The chosen layout defines the HEV architecture and so its peculiar operating conditions [9].

- Series architecture:

This layout is characterized by an indirect connection between the ICE and the wheels. It has two electric motors, one is used as a generator and it is meant to transform the mechanical energy produced by the ICE into electrical energy to be stored in the battery, while the second one is designed to satisfy the peak power requested by the car and it is directly linked to the wheels. The advantage of such a type is the complete decoupling between the load required by the road and the one at which we make the ICE run achieving so much higher efficiency because the engine can operate in high-efficiency region in an almost stationary condition. Even if the propulsion is still the product of two different energy sources, the vehicle is in fact electrically driven, making the need for a gearbox less relevant if not even useless.

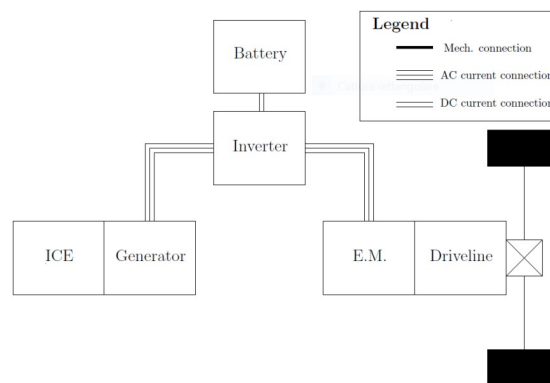


Fig. 2.1 HEV series architecture [1].

On the contrary, this solution suffers from lower overall efficiency due to the double energy conversion and it needs two electrical machines, thus increasing the complexity of the powertrain. Due to these reasons, the most common application of this configuration is in the so-called

range extender configuration, which is a sort of Battery Electric Vehicle (BEV) where the range guaranteed by the electrochemical battery is extended by burning fuel and using the ICE to recharge the battery when needed. The degree of electrification of these vehicles is defined by the so-called *hybridization ratio* [1] [9] reported in the following.

$$R_{h,series} = \frac{P_{EL,GEN}}{P_{EM}} \quad (2.1)$$

For a series architecture HEV, this ratio ranges from 0, which represents a pure electric vehicle, to 1 which stands for the so-called electric transmission meaning that there's no onboard energy-power source, aside from the fuel, and so the engine directly feeds the electric motor for traction.

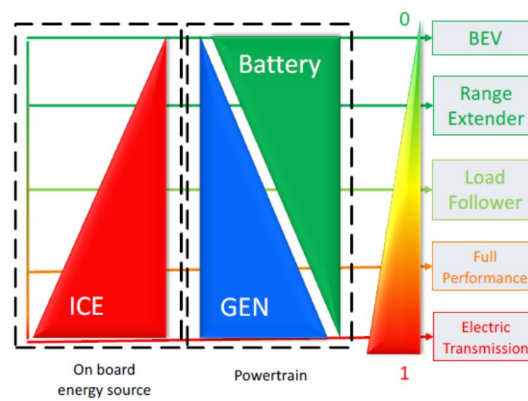


Fig. 2.2 Series hybridization ratio [2].

- Parallel architecture:

The main difference between this configuration and the previous one relies on the capability of a parallel powertrain to employ both the ICE and the EM to power the wheels simultaneously in the so-called *Seamless Power Transition*. This feature usually allows the downsizing of the two propulsion systems because they can add up their power to satisfy the peak road demand and work individually when lower energy is required.

The poor efficiency typical of a series hybrid due to the double energy conversion is avoided and one electric motor is enough to work both as a motor and as a generator reducing the complexity of the architecture.

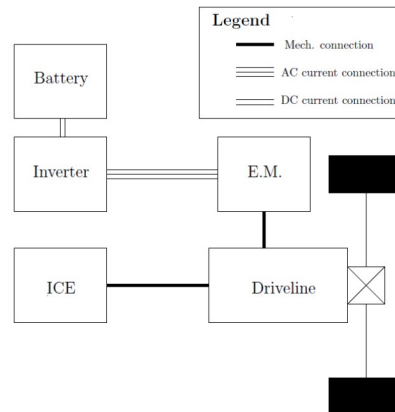


Fig. 2.3 HEV parallel architecture [1].

Even the performance and acceleration are positively affected by this architecture where the two power sources combine their best working conditions to create a very efficient vehicle. On the other hand, a sophisticated control system is required to enable different operating conditions. A *hybridization ratio* [1] [9] can be defined also for a parallel architecture with a slightly different meaning with respect to the one defined for a series architecture.

$$R_{h,parallel} = \frac{P_{EM}}{P_{ICE} + P_{EM}} \quad (2.2)$$

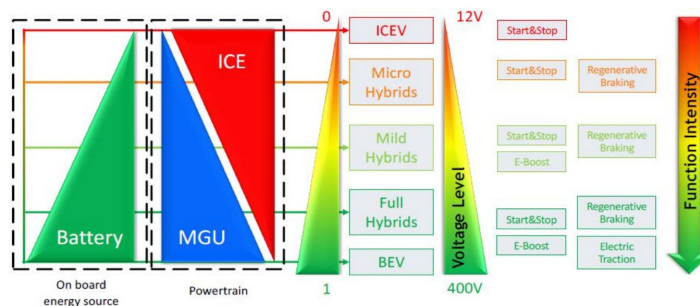


Fig. 2.4 Parallel hybridization ratio [2].

For a parallel configuration, the meaning of the hybridization ratio is rather intuitive since a 0 value represents a conventional ICE-based vehicle while a value equal to 1 is a full electric vehicle. All the intermediate values that the ratio can assume are related to a certain hybridization level, each of which enables different modes and functionalities from the simplest start and stop, to a pure electric range where the vehicle is able to cover a certain distance relying solely on the Electric Motor and the battery.

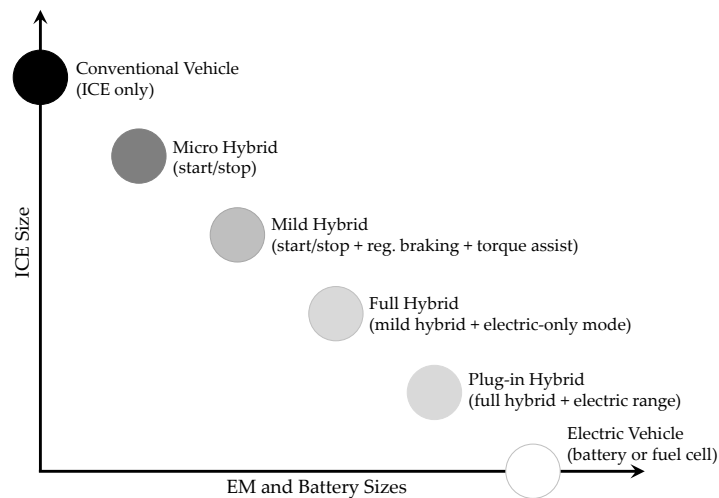


Fig. 2.5 Spectrum of vehicle electrification levels [3].

- Complex architecture:

This category represents the attempt to combine the elements of both series and parallel hybrid systems through an advanced powertrain architecture which allows to enhance the exploitation of all the energy sources and actuators available. The way through which the several machines are linked identifies the complex architecture to be a *series/parallel architecture* if a system of clutches is used or to be a *power – split architecture* if a planetary gearbox is employed. As it is possible to easily notice, the mechanical complexity increases a lot as well as the control system complexity.

Through a proper design of the powertrain control system, it is possible to efficiently exploit all the components enabling different modes which could not be possible with a simple series or parallel configuration. The number of degrees of freedom characterizing this configuration is the key feature of a very efficient energy management system. Nonetheless, the weight and the cost of such a powertrain are heavily affected.

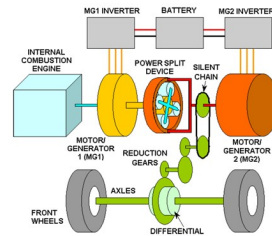


Fig. 2.6 HEV complex architecture [4].

2.2.1 EM position in parallel configuration

In a parallel configuration, a further classification is introduced to specify the electric motor position along the powertrain. The following configurations can be identified:

- P0 (or P1f):
The electric machine is connected to the engine on the Front-End Accessory Drive (FEAD) through a belt, realizing the so-called Belt Starter Generator (BSG). The role of the electric motor is limited due to its allocation and dimension that only allows to guarantee the start and stop functionality and a little energy recovery during braking.
- P1 (or P1r):
The electric motor/generator is directly connected to the crankshaft so that the same functionalities of a P0 configuration are achieved while avoiding the losses related to the belt connection. The capabilities of this configuration are again limited, and the pure electric mode is still not as feasible as in the previous case.
- P2:
The electric machine is positioned between the engine and the transmission and, thanks to the interposition of a clutch in between the two elements, the pure electric mode is enabled: disconnecting the ICE from the EM, and so from the transmission, eliminates all the drag resistance due to the engine

inertia. If, on the other hand, the clutch is engaged, the engine and the motor have the same speed. For these reasons, the size of the electric motor is larger than the above-mentioned configurations making possible more hybrid functionalities.

- P3:

The electric machine is integrated into the transmission shaft and, being positioned downstream of the gearbox, it does not benefit from the gear reduction which usually means a higher exploitation of the electric power available which results in a faster battery aging. However, thanks to this positioning, the electric motor can work at higher speed with respect to the ICE leading to a reduction in the torque provided by the EM for the same power output, allowing in this way a size reduction.

- P4:

This is the only parallel configuration where the two machines are not mechanically connected, in fact, the electric motor is located on the opposite axle realizing the so-called Through-The-Road (TTR) connection. In this way, it is possible to design a four-wheel driving vehicle without the need for a central differential. Easy to be implemented in pre-existing Front Wheel Drive (FWD) vehicles, it suffers from poor regenerative braking.

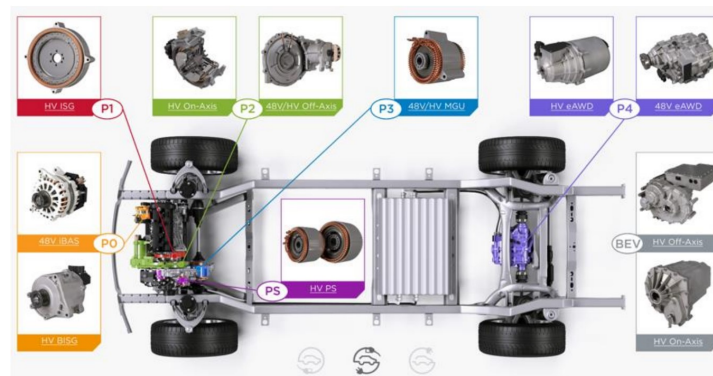


Fig. 2.7 Electric Motor positions in a HEV [5].

2.3 Current market trends

It is now in the public domain the relevance of the environmental issues the world is facing, that's why HEVs have gained a substantial growth of interest in the latest years representing for the customer the perfect balance between the need for a greener vehicle with respect to the traditional ICE vehicles and the need of a vehicle as close as possible to the conventional one in terms of performance, range and usage [15]. The governments and the Original Equipment Manufacturers (OEMs) themselves have also been incentivizing the purchase of HEVs delivering tax incentives, subsidies, and tightening the emission regulations [12]. Another key factor that led, and is still leading, to the diffusion of HEVs is the huge amount of investment of the OEMs in favor of the research towards always more efficient and functional vehicles. All the features and modes enabled by the above-mentioned architecture represent a growing appeal for these vehicles which are experiencing an increasing share of the market. Initially limited to small and medium-sized passenger cars, the electrification of traditional thermal engine cars is expanding towards various vehicle segments like Sport Utility Vehicles (SUVs), crossovers, pick-up trucks and even luxury vehicles where the hybridization is promoted as a high-performance feature, no more only related to efficiency and fuel savings. This broadening of hybrid technology has affected also the sector of commercial vehicles such as buses and delivery trucks, contributing to the spreading of environmentally friendly vehicles. One special mention has to be made with regard to the Plug-In HEVs (pHEVs). Unlike conventional hybrids, they feature a larger battery pack that can be charged using an external power source. This characteristic allows an extended pure electric range without getting rid of the fuel-based ICE as a backup propulsion system in case the vehicle runs out of charge. These kinds of vehicles are the closest type of hybrid to the traditional vehicle in terms of potentialities and so they represent the missing link between simple hybrids and fully electric vehicles. Of course, with the advantages also come a bunch of disadvantages like the need for more structured and branched charging infrastructure to charge these kinds of vehicles. In fact, if not recharged, a pHEV is still able to move relying mainly on the ICE that is downsized because it is supposed to work seamlessly with the electric motor. Only thanks to regenerative braking, the vehicle is not able to recharge the battery and sometimes using the ICE to recharge it is not feasible. For this reason, a pHEV can become even worse than a conventional vehicle in terms of pollutant emissions. The last challenge hybrid vehicles are asked to solve if they want to

have a higher share of the market is the higher price of the car, usually moving the car to a high-hand or premium section of the market. The purchase price makes a great difference to the average customer who is not able to take into account the long-term benefits.

Chapter 3

Reinforcement Learning fundamentals

3.1 Introduction

Any kind of ML has been applied to solve the energy management problem, from Supervised Learning, Unsupervised Learning and Reinforcement Learning. The first category can predict numerical values relying on a huge amount of data properly used to train and test the Neural Network (NN) which is the core of the AI model. It is called *Supervised* because to train the model a pair of inputs and outputs is submitted to the NN so it can tune its parameters to create a function that relates input and output [19]. More simply, the learner has access to labeled examples giving the correct answer. The second category is typically used to identify patterns among the input data that a human being cannot see, that's why they are particularly promising in building effective prediction models. It is called *Unsupervised* because the algorithm is provided with just input data and it is then its role to create the output, not as a numerical value but as a classification result [19]. Unsupervised learning is about extracting underlying structure in data. The third and last category is based on how animals and humans learn that is through a trial-and-error process. The great benefit of this approach is that it does not require any dataset for the training procedure because it has an agent that learns by acting in an environment and receiving a reward for each action. The reward is a score given to every action to evaluate how good that action was based on its effect on the environment[6].

Reinforcement Learning (RL) is quite a recent research topic, in fact, in the mid-2000s, only a few labs around the world were focusing on this machine learning class. At the time it was only partially considered successful and it was not used in the industry at all. The idea that an agent could improve its behavior in the world just by trying things and seeing what happens was the bravest among all the AI algorithms. After all, that's a concept very familiar to everyone since it is the way human beings and animals use to learn. One of the greatest benefits of RL is that the agent is not supposed to only find a good behavior and then apply it in an open loop but the learning and the acting are both bound to a close loop behavior, where the reward is the judge of the action effectiveness. Learning online becomes a defining feature for RL. For this reason, nowadays, universities, companies, and manufacturers are pushing research in this field to ease their daily work, increase their income or efficiency, and so on and so forth [20] [21].

In RL, the agent generates its own training data by interacting with the world, it does not require to be told what the correct actions are. This means that the agent must face the so-called "decision-making problem" where it is asked to choose among different options to interact with the environment. In this context, it is possible to introduce the "K-armed bandit problem" [22] where there is a decision-maker or agent who chooses between k different actions, receiving a reward based on that action. Therefore, how does the agent pick one action rather than another? It chooses according to the "Action-Values" that is related to the reward obtained by taking that action. This concept is formalized by the following formula:

$$q_*(a) \doteq \mathbb{E}[R_t | A_t = a] \quad \forall a \in \{1, \dots, k\} \quad (3.1)$$

$$= \sum_r p(r|a)r \quad (3.2)$$

The Action-Value q of an action a is, using probability and stochastic concepts, the expected reward of selecting a certain action. With a discrete space of the actions is the sum over all the possible rewards multiplied by the probability of taking the action which outputs the same reward. Be aware that the goal of the agent is to maximize the total cumulative reward and so the following:

$$\arg \max_a q_*(a) \quad (3.3)$$

The agent will be prone to choose the action which maximizes the reward and so take the maximum of the argument of the Action-Value. But q_* is not known to the agent, that is why we need to estimate it. The simplest way to do it is called *Sample – Average* and it consists of the ratio between the sum of the rewards obtained by taking a certain action and the number of times that action has been taken before time t :

$$Q_t(a) = \frac{\sum_{i=1}^{t-1} R_i}{t-1} \quad (3.4)$$

At each time instant, the agent will select the action with the highest Action-Value to maximize the long-term reward, acting following the so-called *greedy* behavior. Alternatively, the agent could choose not to pick the greedy action, and so exploit its current knowledge of the world, in order to explore the state-action space and make more accurate estimates of the current Action-Values. The trade-off between exploration and exploitation is one of the biggest dilemmas of RL [6].

To keep the value estimates of the actions up-to-date, an incremental update rule should be defined. Starting from the sample-average method, it can be rewritten in a recurrent form and rearranged as follows:

$$Q_{n+1} = Q_n + \frac{1}{n} (R_n - Q_n) \quad (3.5)$$

where the new estimate is equal to the old estimate plus the difference between the new target, here the new reward, and the old estimate multiplied by the so-called step size parameter α_n [6]. This formula can be generalized as follows:

$$Q_{n+1} = Q_n + \alpha_n (R_n - Q_n) \quad (3.6)$$

where the step-size parameter is usually a function of n .

One key aspect to point out about this formulation is that the most recent reward affects the estimate more than the older ones. This means that the weight of a previously collected reward decreases more and more with the passing of time.

It has been mentioned the exploration-exploitation dilemma. *Exploration* is what the agent should go for to improve its current knowledge about each action,

making its estimates more accurate. Hopefully, this approach allows the agent to make more conscious decisions in the future, leading to long-term benefits. On the other hand, *Exploitation* takes advantage of the current estimates value of each action choosing the “greedy” action and maximizing the reward it can get in that instant. In this case, it is preferred to maximize the short-term benefit rather than trying to increase the reward in the long run. One very simple method developed to solve this tricky trade-off is called $\epsilon - greedy$. It simply consists of picking the greedy action all the time apart from ϵ percent of the time where exploration is pursued by taking a random action.

$$A_t \leftarrow \begin{cases} \arg \max_a Q_t(a) & \text{with probability } 1 - \epsilon \\ a \sim \text{Uniform}(a_1 \dots a_k) & \text{with probability } \epsilon \end{cases} \quad (3.7)$$

Another way to encourage exploration is using *OptimisticInitialValues* which means overestimating the Action-Values on purpose. In this way, once the first action is selected, the observed reward from taking that action will likely be smaller than the optimistic estimate making other actions more appealing. It can be shown how this method really increases exploration from the beginning of the interactions of the agent with the world. Anyway, some limitations belong to this method:

- Optimistic initial Values only encourage early exploration: at some point, the agent will have defined an estimate for each action and will behave always in a greedy manner;
- In non-stationary problems, the actual Action-Value might change while the agent can be already settled around a value and so prone not to take that action anymore even if it has changed and the estimate is no longer up to date. This limitation derives from the previous one, if exploration is in some way encouraged throughout the learning process, this issue would not arise;
- Especially in real-world problems, knowing what an optimistic initial value for the case could be a not trivial problem

3.2 Markov Decision Process

All the definitions and principles shown up to now were thought to be related to a world where our agent is asked to act always in the same situation or where the best action is always the same. In order to translate these concepts toward cases closer to reality, the Markov Decision Process (MDP) formalism is introduced. This concept is based on the understanding that various situations require distinct actions. Current actions should not only aim to maximize immediate rewards but also take into account the future outcomes that may emerge as a result of the action, thus impacting future rewards. It is necessary now to explain better the concept of state and how it fits in this field.

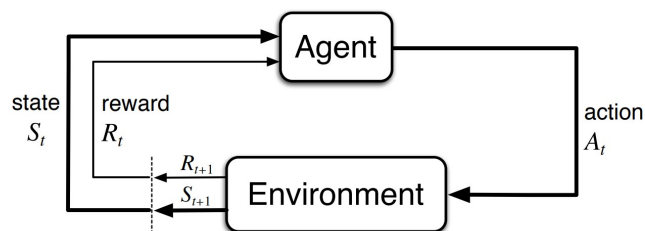


Fig. 3.1 The agent–environment interaction in a Markov decision process [6].

The figure 3.1 shows a framework where the agent and the environment interact at discrete time steps. At each instant t , the agent reads the state that describes the environment at that stage (S_t), and based on that state, it takes an action A_t from a set of valid action at state S_t . When an action is taken, it will affect the reward as well as the following state as shown by the following diagram.

The dynamics involved in this iterative process can be described by using some stochastic concepts where the density function p represents the joint probability of obtaining a reward R and future state s' if a precise action a is taken in a certain state s . The formula is here reported:

$$p(s', r | s, a) \quad (3.8)$$

It is worth noting that the future state and reward only depend on the current state and action. This important relation is called *Markovproperty* and it means that the present state is sufficient to represent the environment and allow the agent to act consequently, so any kind of memory about past states would not improve the accuracy of the probability distribution as a prediction of the future.

Thanks to this simple and complete formulation, MDP formalism can be adopted to formalize a wide variety of sequential decision-making problems.

The goal of RL is to maximize the long-term reward and so at each instant, an action must be taken considering also the possible affection of that on the environment and so on the future rewards. It is now useful to introduce a new definition: the Return at timestep t is the sum of rewards obtained after time step t . It is usually denoted through the letter G . Due to the randomness involved in the possible reward obtained by the agent and the state transitions of the environment, it is usually more meaningful to consider the expected reward that can be defined as follows:

$$\mathbb{E}[G_t] = \mathbb{E}[R_{t+1} + R_{t+2} + R_{t+3} + \dots] \quad (3.9)$$

To give a proper definition of the return and of the expected return it is necessary to consider a finite sum which means that there will be a final step.

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \dots + R_T \quad (3.10)$$

At that time step, the interaction between the agent and the environment ends and this is also called the end of an *Episode*. Each episode starts at the same initial condition and it is not affected by the *Terminalstate* which is how the previous episode ended. When the mission that the agent is required to carry out can be split into episodes, it is called *Episodic – Task*.

What if the task cannot be broken up into independent episodes? In that case, there are no terminal states and the agent environment interaction continues without end. This is the case of a so-called *ContinuingTask*. For this task, the previous definition of return is no longer suitable because it would mean obtaining an infinite sum. For this reason, the introduction of a *DiscountFactor* is necessary in order to let the return be finite. This factor is well known in the literature with the symbol γ and it ranges from 0 to 1, including the lower bound but excluding one as a legit value otherwise it will lead to the infinite sum issue discussed above.

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots + \gamma^{k-1} R_{t+k} + \dots \quad (3.11)$$

The effect of discounting on the return formulation is striking: it makes more accountable the immediate reward while decreasing the contribution of rewards far in the future. It can be proven that this sum is finite. Now, to better understand how the discount factor affects the decision of the agent at each instant let's consider the two extreme cases:

- When gamma equals zero, the agent is said *short – sighted* because, being the return just equal to the very next reward, it only cares about immediate reward;
- When gamma approaches one, the weight of the immediate and future reward is almost the same and in this case, the agent is defined as *far – sighted*.

Last but not least, let's rewrite the return formula in a recursive way: it turns out that the return simply is the immediate reward plus the discounted future return.

$$G_t = R_{t+1} + \gamma G_{t+1} \quad (3.12)$$

Having understood that the action selected by the agent affects the immediate reward as well as the following state, the reader may wonder how an action is selected from all the feasible ones in a certain state. It has been mentioned the concept of ϵ – *greedy* approach where the agent selects an action aiming to maximize the reward 1- ϵ percent of the time and a random action the remaining ϵ percent of the time. But how are the actions and the states related? How does the agent actually pick an action? The answer is through a function called policy that can be deterministic or stochastic. In the first case the policy links to each possible state one and one action only while, in the second case, it assigns a non-zero probability to each action. For a stochastic policy is possible and meaningful to define a policy π in the following way:

$$\pi(a|s) \quad (3.13)$$

where π is the probability of taking action a when in state s and being the policy a probability over a set of actions it must fulfill the conditions reported in [6] and below:

$$\begin{cases} \sum_{a \in \mathcal{A}(s)} \pi(a|s) = 1 \\ \pi(a|s) \geq 0 \end{cases} \quad (3.14)$$

As a defining feature of a policy, it is important that it does not depend on anything but the current state. The probability of picking an action in a certain state must not be affected in any way by previous actions or states. This should not be seen as a boundary on the agent but as a requirement on the state instead. Remember that we are dealing with MDP where the state is supposed to hold all the required information for decision-making.

Keeping in mind that the goal of RL is to maximize the long-term reward, it is now possible to introduce two new parameters used by the agent to orient itself throughout its attempt to define an optimal policy, avoiding waiting for the end of the episode or for the next reward in the case of a continuing task. The *State – value function* is the expectation of a certain return at timestep t , starting from state s and following policy π . Here it is now reported the analytical formula:

$$v_{\pi}(s) \doteq \mathbb{E}_{\pi}[G_t | S_t = s] \quad (3.15)$$

Right after the definition of state-value, it is worth describing the *Action – value function* that is, as exemplified by equation 3.16, the expected return for an agent that selects action a in state s and follows policy π :

$$q_{\pi}(s, a) \doteq \mathbb{E}_{\pi}[G_t | S_t = s, A_t = a] \quad (3.16)$$

These two functions turn out to be very effective in enhancing agent learning because they summarize all the possible future paths by averaging over returns. Even in those tasks where the reward is unlikely to be obtained if not at the end of a successful episode, value functions are able to give the agent a hint about the quality of its behavior and condition based also on predictions about possible upcoming rewards and system dynamics.

Being the value function of a state or an action at a timestep t so important for the agent, it becomes much more relevant also relating the current value of a state (or an action) to the value of future states (or actions). In this way, it could be possible for the agent to predict possible bad or good actions without actually experiencing them. To formalize the relation between values and their possible successors the theory proposes the Bellman equation for both actions and states.

Starting with the state-value function, first, the definition of return is recalled:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \quad (3.17)$$

The return definition is substituted in the equation 3.16 of the state-value function. It is now possible to expand the expected return over possible actions, possible rewards, and next states.

$$v_{\pi}(s) = \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) [r + \gamma \mathbb{E}_{\pi}[G_{t+1} | S_{t+1} = s']] \quad (3.18)$$

It could now be possible to recursively expand equation 3.18 but it makes more sense to notice that the expected return is the definition of value function for the next state and so the following equation is obtained:

$$v_{\pi}(s) = \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) [r + \gamma v_{\pi}(s')] \quad (3.19)$$

Similarly, the Bellman equation for action-value can be derived. Its formulation is here reported:

$$q_{\pi}(s, a) \doteq \mathbb{E}_{\pi}[G_t | S_t = s, A_t = a] \quad (3.20)$$

Equation 3.20 can be rewritten considering the expectation as a summation over possible next states and rewards. Furthermore the return formulation is made explicit using the action value of the next state and action under policy π . The result is reported in equation 3.21.

$$q_{\pi}(s, a) \doteq \sum_{s'} \sum_r p(s', r|s, a) \left[r + \gamma \sum_{a'} \pi(a'|s') q_{\pi}(s', a') \right] \quad (3.21)$$

3.2.1 Optimal policy evaluation

It has now been repeated several times that the ultimate goal of reinforcement learning is maximizing the reward in the long run and the agent aims to that by searching for the optimal policy. Let's explain what it means optimal policy and how to find it. First of all, a policy is said to be better, or as good as, than another policy if and only if the value under it is greater than or equal to the value under the other policy for each possible state. At this point, an optimal policy is simply a policy in which state values are always greater or equal to every other policy. In other words, an optimal policy will always have the highest possible value in each state. Even if the definition of optimality is itself relatively simple, computing it could be quite long and complex: also limiting the search to discrete policies, the number of possible policies is the number of possible actions to the power of the number of states. Manually evaluating all the policies could cause the computational cost to explode, that is why the solution to this issue comes again with another set of Bellman equations.

As just stated, the value of the state of an optimal policy, defined with π_* , is always the highest among all the possible policies, thus it is possible to write:

$$v_{\pi_*}(s) \doteq \mathbb{E}_{\pi_*}[G_t | S_t = s] = \max_{\pi} v_{\pi}(s) \quad \forall s \in \mathcal{S} \quad (3.22)$$

And the same can be done for optimal action-values:

$$q_{\pi_*}(s, a) = \max_{\pi} q_{\pi}(s, a) \quad \forall s \in \mathcal{S} \text{ and } a \in \mathcal{A} \quad (3.23)$$

Recalling the equation of the state-value function, its optimal value can be defined by using an optimal policy in its definition as follows:

$$v_*(s) = \sum_a \pi_*(a|s) \sum_{s'} \sum_r p(s', r|s, a) [r + \gamma v_{\pi_*}(s')] \quad (3.24)$$

since there is always at least one optimal policy that selects an optimal action in each state. Such a deterministic optimal policy will assign Probability 1, for an action that achieves the highest value and Probability 0, for all other actions. We can express this another way by replacing the sum over π_* with a max over a:

$$v_*(s) = \max_a \sum_{s'} \sum_r p(s', r | s, a) [r + \gamma v_{\pi_*}(s')] \quad (3.25)$$

The same can be done for the action-value function, obtaining the following as the final result:

$$q_*(s, a) = \sum_{s'} \sum_r p(s', r | s, a) [r + \gamma \max_{a'} q_{\pi_*}(s', a')] \quad (3.26)$$

This set of equations, called Bellman's optimality equations, cannot be solved linearly as it was possible to do for the Bellman equations related to the value functions because the maximum is not a linear operation.

By solving the Bellman's equations for v_* (state-value) or q_* (action-value) it will be possible to find π_* , so the optimal policy. This is in fact fairly easy, especially if dealing with a discrete policy where an action taken in a state leads to only one state (and its relative reward). Once the state-value function is known, recalling equation 3.25, it is sufficient to look one step ahead between the available actions to choose in order to identify the one that gives the highest return value.

$$\pi_*(s) = \arg \max_a \sum_{s'} \sum_r p(s', r | s, a) [r + \gamma v_{\pi_*}(s')] \quad (3.27)$$

Using action-value functions instead (equation 3.26) it is even easier because it is not necessary to look forward since the value function itself considers the value of taking a certain action and the expected reward from that.

$$\pi_*(s) = \arg \max_a q_{\pi_*}(s, a) \quad (3.28)$$

It has been explained how Bellman's equations can be solved through linear algebra systems but this often turns out to be not feasible. One effective tool developed to carry out policy evaluation, that is evaluating the state-value function for a certain policy, and policy control, that is the task of improving a policy until the optimal policy is obtained, is Dynamic Programming (DP). DP uses Bellman's equations along with the knowledge of the dynamics of the environment, described by the probability function p , to do policy evaluation and control without interacting with the environment. Exploiting Bellman's equations with a recursive approach is how Dynamic Programming solves the state-value

function estimation [23]. Using those equations as an update rule, rather than as an equation, gives as a result the exact state-value. Applying this procedure, it is possible to iteratively refine the value function estimation, obtaining better and better approximation until the difference between two subsequent updates is lower than a preset threshold.

$$v_{\pi}(s) = \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) [r + \gamma v_{\pi}(s')] \quad (3.29)$$

Equation 3.29 turns into equation 3.30 in an iterative update procedure.

$$v_{k+1}(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) [r + \gamma v_k(s')] \quad (3.30)$$

The policy improvement theorem, that exploits the just-described evaluation of a policy to find a better one, is now analyzed.

$$q_{\pi}(s, \pi'(s)) \geq q_{\pi}(s, \pi(s)) \quad \forall s \in \mathcal{S} \quad (3.31)$$

If in each state the value of the action according to π' is higher or at least equal to the one selected by π , then it is possible to say that π' is at least as good as π . More practically, this can be done by acting greedily knowing the state-value function. In this way, it is sure that the new policy is an improvement with respect to the previous one. The value function should then be updated according to the new policy and then iteratively repeat the process. This process alternates between policy evaluation and improvement unless convergence is reached when a further policy improvement leaves unchanged the policy.

The term *Generalized Policy Iteration* is used to identify all the ways policy evaluation and improvement can be combined. One of these approaches called *Value Iteration* still sweeps among all the states to evaluate them and then greedify the policy, while the policy evaluation step is not carried out to completion. There's no reference to any policy, it is simply the state evaluation using the action that maximizes the current value. Then the optimal policy can be identified by computing the arg max. A threshold is used to understand when the change in the value is low enough to terminate the iteration. Carrying out a systematic sweep over states like this is called synchronous method. Asynchronous methods sweep among states in any order, usually promoting the update of the states closer to

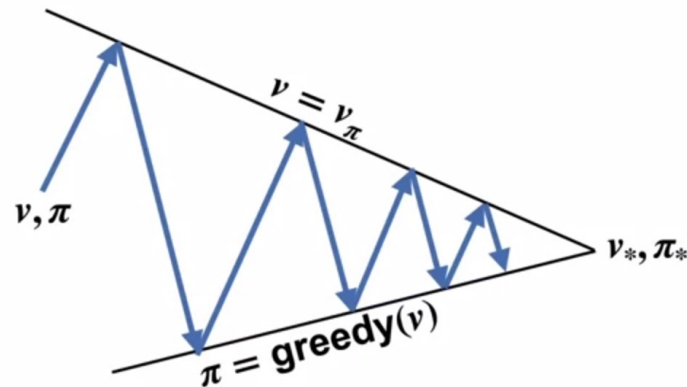


Fig. 3.2 The "dance" of policy and value is visualized as bouncing back and forth between one line, where the value function is accurate, and another where the policy is greedy [6].

the states where the value changes the most or that changed recently. Anyway, to guarantee convergence, the update must be run in every state.

3.2.2 Sample-based models: Monte Carlo

So far, it has been explained how useful and efficient Dynamic Programming is to approach Reinforcement Learning problems. Unfortunately, very rarely the dynamics of the environment are known to the agent as DP requires, especially when dealing with real environment problems that are too complex to be described through probability functions. To solve this issue, sample-based models are the solution to make the agent able to learn just from its experience, without prior knowledge of the environment. One of these methods is called *Monte Carlo* and it estimates values by averaging over many random samples [24]. For instance, it can observe multiple returns from the same state, then it averages those samples to estimate the expected return. The returns can be observed only at the end of an episode. Learning from experience allows to avoid keeping a large model of the environment. The biggest advantage of Monte Carlo with respect to DP is that while the last one computes the value of a state relying on other state-values, in Monte Carlo each state estimate is independent. Furthermore, the computation does not depend on the dimension of the MDP problem but rather on the length of the episode. The same thing holds for state-action values where returns from

a state-action pair are collected, by following a policy, and averaged. Thanks to this approach it is possible to learn a policy since different action values are known for the same state and so it is possible to pick up the highest one. The most attentive reader may have noticed an issue in applying this approach to a deterministic policy: in some states, the agent will avoid taking some actions and so the Monte Carlo sample-average will not be able to properly estimate the value of unexplored state-action pairs. All the available actions at each state must be tried to fairly compare them and choose the best one. For this reason, exploration should be promoted and maintained throughout the training. In this context *Exploring Starts* represents a solution: they can be used to force the agent to take a random action at the beginning of the episode and then follow the given policy.

The pseudo-code reported in figure 3.3 explains the procedure to apply Monte Carlo to Generalized Policy Iteration.

```

Monte Carlo ES (Exploring Starts), for estimating  $\pi \approx \pi_*$ 
Initialize:
   $\pi(s) \in \mathcal{A}(s)$  (arbitrarily), for all  $s \in \mathcal{S}$ 
   $Q(s, a) \in \mathbb{R}$  (arbitrarily), for all  $s \in \mathcal{S}, a \in \mathcal{A}(s)$ 
   $Returns(s, a) \leftarrow$  empty list, for all  $s \in \mathcal{S}, a \in \mathcal{A}(s)$ 
Loop forever (for each episode):
  Choose  $S_0 \in \mathcal{S}, A_0 \in \mathcal{A}(S_0)$  randomly such that all pairs have probability  $> 0$ 
  Generate an episode from  $S_0, A_0$ , following  $\pi$ :  $S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T$ 
   $G \leftarrow 0$ 
  Loop for each step of episode,  $t = T-1, T-2, \dots, 0$ :
     $G \leftarrow \gamma G + R_{t+1}$ 
    Append  $G$  to  $Returns(S_t, A_t)$ 
     $Q(S_t, A_t) \leftarrow$  average( $Returns(S_t, A_t)$ )
     $\pi(S_t) \leftarrow \operatorname{argmax}_a Q(S_t, a)$ 

```

Fig. 3.3 Monte Carlo pseudo code [6]

As mentioned before, to enhance exploration, a random state-action pair is selected at the beginning of the episode, and then the selected policy is followed, storing state, actions, and rewards throughout the episode. Once the episode is completed, starting from the last one, all the returns are computed, stored, and averaged. Thanks to the computation of the average, all the action values are updated, completing this way the evaluation step. Then it is possible to carry out the policy improvement by choosing the greedy action for each state.

When dealing with real world problems, choosing a random state-action pair for the beginning of the episode might not be feasible. Adopting the so-called “ ϵ -soft policies” makes it possible to force the agent to continuously explore by

assigning a non-zero probability to each available action. ϵ -greedy policy is a stochastic policy belonging to the ϵ -soft policies. Being stochastic means that the agent continues to visit all possible state-action pairs indefinitely because each action can be selected with a probability equal to ϵ over the number of actions. Even when the optimal policy is reached, these stochastic policies perform worse than deterministic optimal policy because they continue to explore introducing so a sub-optimality, but they anyway turn out to be efficient, improving over and over the policy adopting an ϵ -greedy approach at the end of each episode.

3.2.3 Off-policy algorithms

So far, only *On-policy algorithms* have been presented where the agent behaves and learns following the same policy. In *Off-policy algorithms*, the agent generates data following a certain policy which is different from the off-policy that is the one the agent is learning, meant to become the optimal policy. The policy that is learned is called *Target Policy*. The policy used by the agent to act is called *Behavioral Policy* (or *Behavior Policy*) and it is usually a great source of randomness to make the agent explore as much as possible during learning. A very important aspect of the behavior policy is that it must contain the target policy otherwise it wouldn't be possible for the agent to properly estimate the value of a state-action pair because it would never experience the action suggested by the target policy. An on-policy is just a special case of off-policy where the target policy is equal to the behavior policy.

In order to do off-policy learning it must be possible to learn with one policy while following another. This is in fact possible thanks to the concept of *Importance Sampling*. As explained before, in off-policy algorithms the data, like the state, the action, the rewards and so the returns, are generated by the behavioral policy because it is the one through which the agent interacts with the environment, but it is required to perform the learning based on the target policy that is different from the behavioral. First of all, two policies are available, and sampling a random variable x is done from distribution b while learning according to distribution π of x . It is necessary to introduce a parameter called *Importance sampling ratio*, indicated with the letter ρ , that is simply the distribution of drawing a certain variable x from π over the probability according to b .

$$\mathbb{E}_\pi[X] = \sum_{x \in X} x \rho(x) b(x) \quad (3.32)$$

$$= \mathbb{E}_b[X \rho(X)] \quad (3.33)$$

In equation 3.33, the product between the random variable x and the Importance sampling ratio has been treated as a new variable and so the summation has been rewritten as an expectation over the distribution b , the one from which data are actually sampled. Recalling that an expectation over a value can be computed with a sample average:

$$\mathbb{E}[x] \approx \frac{1}{n} \sum_{i=1}^n x_i \quad (3.34)$$

Now, it is just necessary to compute a weighted sample average where the ratios are the weighting factors.

$$\mathbb{E}_\pi[X] = \sum_{x \in X} x \rho(x) b(x) \quad (3.35)$$

$$\approx \frac{1}{n} \sum_{i=1}^n x_i \rho(x_i) \quad (3.36)$$

Thanks to this immediate procedure it is possible to estimate expected values according to one distribution when data are sampled from another one. At this point, it might be asked how it is possible to compute the distribution π or b and so the trajectory followed by the agent during an episode. This can be computed as follows:

$$\mathbb{P}(\text{trajectory under } b) = \prod_{k=t}^{T-1} b(A_k | S_k) p(S_{k+1} | S_k, A_k) \quad (3.37)$$

where p stands for the probability of the environment to evolve into state S_{k+1} if action A_k is taken in state S_k . Substituting this formula in the definition of the Importance Sampling Ratio, it is obtained:

$$\rho_{t:T-1} \doteq \frac{\mathbb{P}(\text{trajectory under } \pi)}{\mathbb{P}(\text{trajectory under } b)} \quad (3.38)$$

$$\doteq \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)p(S_{k+1}|S_k, A_k)}{b(A_k|S_k)p(S_{k+1}|S_k, A_k)} \quad (3.39)$$

the dynamic response of the environment cancels out for each time step, making the equation look like:

$$\rho_{t:T-1} \doteq \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)} \quad (3.40)$$

When computing the returns at the end of the episode, starting from the last step, the ratio is also computed and then used to correct the return. The value is then stored and used in a recursive manner as it is done for the returns.

$$\rho_{t:T-1} \doteq \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)} \quad (3.41)$$

$$= \rho_t \rho_{t+1} \rho_{t+2} \dots \rho_{T-2} \rho_{T-1} \quad (3.42)$$

$$W_{t+1} \leftarrow W_t \rho_t \quad (3.43)$$

Where w is the weight used to correct the return in the following way

$$G \leftarrow \gamma W G + R_{t+1} \quad (3.44)$$

3.2.4 Temporal Difference algorithm

In RL, one of the aims is to define a value function able to estimate the return from a given state. Starting from the following formula:

$$V(S_t) \leftarrow V(S_t) + \alpha [G_t - V(S_t)] \quad (3.45)$$

It is possible to update the estimate, step after step. With this equation, becomes striking the limitation of Monte Carlo where it is required to wait for the end of the episode to sample data about all the trajectories to compute the return. It would be much more effective to be able to learn during the episode so in the following is explained how to obtain that. Recalling the definition of discounted return:

$$G_t = R_{t+1} + \gamma G_{t+1} \quad (3.46)$$

Now, starting from the formula of the value of a state at time t , it is possible to write the return in a recursive way and turn the value function into a recursive expression as well:

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t | S_t = s] \quad (3.47)$$

$$= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} | S_t = s] \quad (3.48)$$

$$= R_{t+1} + \gamma v_\pi(S_{t+1}) \quad (3.49)$$

Going back to equation 3.45, it is possible to rewrite the return at time t as the reward in the following time-step plus the estimate of the next state-value:

$$V(S_t) \leftarrow V(S_t) + \alpha [R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \quad (3.50)$$

Considering the value of the next state as it was the return until the end of the episode it is reached, in this way, it is not necessary to wait until the end of the episode to estimate the states but it is enough waiting for the next step. This is one of the core concepts of RL and it is called Temporal Difference (TD) algorithm [22]. The term in square brackets is called the TD error and is usually referred to as δ_t . The TD target is the sum between the future reward and the discounted value of the next state. TD updates the value of the state at time t to improve its own estimate of the next state. This concept is like what Dynamic Programming does in its update rule in fact, in that case, the value of the state was updated according to the value of all possible next states which requires the knowledge of the environment dynamics. In TD it is sufficient just the next state that is taken from the environment. Simply considering the current time step as $t + 1$ and the previous time step as t , it is enough to store the state from the previous time step

to compute the TD update [25]. At time t there is a state S and an action A is taken, obtaining a reward R and a new state S . Only when the data of the next state are available, it is possible to update the previous state value. The flow followed by TD(0) algorithm is very simple: a policy to follow is defined, the state is initialized and then at each time step an action is taken according to the policy, and the value of the state is updated through to the TD update rule. The only variable this algorithm requires to store is the value of the previous state. It is clear that TD introduces some great advantages:

- Unlike Dynamic Programming, it does not require a model of the environment, it learns directly from experience as MC;
- Unlike Monte Carlo, it can update the state at each time step using other estimates instead of waiting for the end of the episode;
- TD converges to the correct prediction usually faster than Monte Carlo thanks to the capability of updating the states at each time-step and just comparing each step to the following one, instead of waiting for the end and so updating all the encountered states with respect to the final set of rewards.

Now it is discussed how Temporal Difference can be implemented to do Policy Evaluation in a General Policy Iteration (GPI) algorithm. Remember that when it was shown how to implement Monte Carlo as an update rule in GPI, the algorithm did not perform a full policy evaluation step before improving the policy itself. To implement TD in GPI, it is necessary to slightly change the update rule explained above: instead of observing the stream of states and learning the value of each of them, the object of the update will be a state-action pair. This algorithm is called *SARSA* because in a state S_t the agent takes an action A_t obtaining a reward R_{t+1} and, since it is being used TD, the algorithm needs to know not only the next state S_{t+1} but also the next action A_{t+1} to update the previous state-action pair value. That is where the acronym comes from. Here is reported the new update equation:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha ((R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)) \quad (3.51)$$

This algorithm has been explained considering just a fixed policy to focus on policy evaluation but, thanks to GPI framework, it is possible to turn it into a

control algorithm that improves the policy at each timestep instead of waiting for the end of the episode. SARSA will perform much better than MC, especially in those tasks where the terminal state is required to end the episode. In fact, unlike Monte Carlo which could get stuck in the same episode forever, SARSA would recognize a bad policy, improve it and so reach the terminal state.

3.3 Q-learning algorithms

One of the first major online Reinforcement Learning algorithms is Q-learning. Very similar to SARSA, let's see its formula to point where it differs.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left((R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a') - Q(S_t, A_t)) \right) \quad (3.52)$$

Sarsa seems to take several features from the action value Bellman's equation, while Q-learning comes from the Bellman's optimality equation for action values. Thanks to this, Q-learning has the advantage of directly obtaining the optimal action value q_* instead of switching between policy evaluation and policy improvement as in Sarsa. The latter in fact, is the sample-based version of policy iteration where Bellman's equation is used for action values according to a fixed policy. On the other hand, Q-learning is the sample-based version of value iteration where the Bellman optimality equation is iteratively applied to improve value function.

One main difference between the two algorithms is that Q-learning updates the action-value only when it finds a better action while Sarsa updates the value every time it takes an exploratory action, this is because Sarsa's update is based on the next state estimate and not on its maximum.

As it has been discussed previously, the so-called off-policy algorithms exploit important sampling to estimate values according to the target policy while acting following the behavior policy. Q-learning does not use this approach. While in Sarsa the agent bootstraps off on the value of the action is going to choose, taken from its policy, Q-learning bootstraps off on the highest action value of the next state. This could be seen as sampling actions from an estimate of the optimal policy rather than on the behavioral policy which means that the agent is learning from the best possible action but then acts with another one and so can be defined

as off-policy. Thanks to this operating mode, Q-learning is able to learn always greedy while making its behavioral policy free to explore all state-action pairs. Furthermore, the reason for that there isn't any sampling ratio is because the agent is estimating action values following an unknown policy.

In literature is also present a variation of Sarsa called Expected Sarsa. The novelty consists in taking a weighted sum of the values of all possible next actions where the weights are the probability of taking that action in that state according to the policy. It is an expectation over next actions, that's from where comes the name. The update rule is nearly identical with respect to the one seen with Sarsa, with the only difference that TD error uses an estimate of the expectation of the next action value instead of sampling the next action value. That means that at every time step, the agent must average the next state's action values according to how likely they are under the policy. In this way the update of Expected Sarsa is much more stable because averaging the possible next action values, it relies on correct updates implying a much lower variance. A downside in terms of computations, especially with a huge number of possible next actions must be certainly taken into account. Finally, how can be Expected Sarsa be related to Q-learning? Since the estimate of the action value is computed considering the expectation over possible next action, the action the agent actually takes can belong to a different policy and so making Expected Sarsa an off-policy algorithm as much as Q-learning where both of them don't rely on importance sampling. Furthermore, if the target policy is greedy with respect to its expected action value estimates, only the highest value action is considered in the expectation and so it becomes like searching for the maximum among possible next actions as for Q-learning. It is now possible to conclude affirming that Expected Sarsa is a generalization of Q-learning.

3.3.1 Overview on models

As it has been depicted so far, there are methods like Temporal Difference which directly sample from the environment to gain experience about the world and choose accordingly how to act; but there are also methods like Dynamic Programming that use the complete knowledge of the environment to find the best control strategy for the agent. There's a third way which tries to pick just the benefits from the two extremes and it consists of building, and then relying, on a model of the world. A model is a representation of the environment, giving as output

the possible next state and reward if provided with a certain state and action. It is meant to plan and so to evaluate how good is a policy and then improve it. Models can be of *sample type* and they produce an outcome based on some probabilities underlying the true world. Alternatively, there are *distribution models* which specify the likelihood of every outcome. They take into account also joint probability and they can compute the exact expected outcome by summing over all outcomes weighted by their probability. All of this comes with the shortcoming of a larger computational memory needed while sample models are much more compact. As already mentioned, one of the main scopes of having a model of the environment relies upon planning that is improving a policy. Sampling experience from the environment allows one to update the action values and then behave greedily with respect to that. One practical approach to do that is Q-planning which follows the same procedure as Q-learning but instead of sampling from the world to update the value, it samples from the model. The huge potential of these kinds of *planning updates* is that they can be carried out in parallel with respect to the interaction of the agent with the real environment, speeding up the learning process and improving the target policy.

3.3.2 Function approximators

Up to now only tabular methods have been described where learned values are stored in tables for each possible state. Clearly, this approach is not suitable for real problems with a much larger amount of data. For this reason, it is now useful to take a step further. So far, the proposed value approximators always had the same form, for each state a value was stored and updated when needed. Here comes the idea of parametrized functions to approximate values thanks to a set of weights incorporated in the function that can be modified to tune its outcome. The following simple expression is considered:

$$\hat{v}(s, W) \doteq w_1 X + w_2 Y \quad (3.53)$$

With two simple weights to be stored, it is possible to represent the state values of several states in a form much more compact than it would be with tabular methods. In equation 3.53, X and Y are the vector constituting the feature describing the state of the environment. Note that the state-value approximation is denoted with \hat{v} . With this representation, the algorithm will no longer update

the value itself but the weight instead. The equation 3.53 reported above, it is a special case called *Linear Function Approximation* that can be defined in a more generalized way as:

$$\hat{v}(s, W) \doteq \sum w_i x_i(s) \quad (3.54)$$

$$= \langle w, x(s) \rangle \quad (3.55)$$

Where the vector x contains the so-called feature. Constructing the feature vector becomes a real challenge because it heavily impacts how the function can approximate values. But what should an approximation function guarantee to properly estimate values? Two keywords can be introduced to describe a good approximation function: *generalization* and *discrimination*. More broadly, generalization is the ability to use the knowledge related to specific situations to draw conclusions about a wider variety of situations. In the context of policy evaluation, it means that updates to the value estimates of one state affect the value of other states. In other words, visiting some states can be avoided if they are like some states that have already been visited. On the other hand, discrimination means making the values of two states to be different to consider them differently. It should be quite evident how both generalization and discrimination aim for opposite objectives, that's why a trade-off between the two is at the base of approximation function design.

Recalling that in RL an agent interacts with an environment obtaining data in the so-called online setting, it is not possible to treat data as it would be done with a fixed batch of data as in the case of Supervised Learning (SL), respectively called offline setting. This implies that the same methods used to obtain approximation functions in SL cannot be directly applied to RL, especially because they should deal with online generated data, that sometimes can also be correlated with each other as for TD, where the next value estimate is based on other estimate that are changing during the training.

Having pointed out what is function approximation and why it is useful to get rid of tabular methods, it is time to define what is the target to be optimized and that can be used as an indicator of the approximation accuracy. A well-known and widely used example of this indicator is *Mean Squared Value Error* (MSVE) defined as follows:

$$\bar{V}E = \sum_s \mu(s) [v_\pi(s) - \hat{v}(s, W)]^2 \quad (3.56)$$

Where the equation between brackets is the squared error that is the square of the difference between the value and its approximation, $\mu(s)$ is a probability distribution giving weight about how much error of a state should be considered and it is usually computed as the amount of time spent in that state by the agent when following the given policy (policy π in equation 3.56). Changing the weights is how the Value Error (VE) can be decreased or unfortunately increased.

3.4 Policy gradient methods

To frame value function estimation as a Supervised Learning problem, the Mean Squared Value Error has been introduced as a target to be minimized. According to equation 3.56, the variable to be minimized depends on a set of weights used to parametrize the function that tries to approximate the value of each state. It has been underlined the relevance of these weights and their role in achieving the lowest value possible of value error. An obvious question may now arise: how is it possible to change them in a way that minimizes the target? It will now be presented a method to solve this issue called *Gradient Descent* [6]. Before going through the working principle of this approach, it is essential to recall some concepts from calculus.

$$\frac{\partial f}{\partial w} \quad (3.57)$$

The expression 3.57 describes the derivative of a function f with respect to the variable w . The sign of the derivative of f at a particular w , indicates the direction towards the weight w should be changed to increase f while the magnitude of the derivative indicates the slope of the function f at point w . The function f to be minimized is usually parametrized by multiple variables which means that w is a vector. Here's where the concept of gradient is introduced to describe how f changes according to vector w .

$$w \doteq \begin{bmatrix} w_1 \\ w_2 \\ \dots \\ w_d \end{bmatrix} \quad \nabla f \doteq \begin{bmatrix} \frac{\partial f}{\partial w_1} \\ \frac{\partial f}{\partial w_2} \\ \dots \\ \frac{\partial f}{\partial w_d} \end{bmatrix} \quad (3.58)$$

The gradient is a vector of partial derivatives indicating how a local change of each component of the vector w affects the function f . The higher the magnitude, the steeper the function in that point w , but regardless of the slope, the gradient is mainly useful to get the direction towards which w should be changed. In the specific case considered, the function to be minimized could be the MSVE which is in fact a function of the weights because it contains the estimate of the state that is realized thanks to an approximation function that uses weights. To sum up, the idea behind the definition of Gradient Descent is that the weight should be moved in the direction of the negative of the gradient to reduce the objective. This concept is summarized by the following formula:

$$W_{t+1} \doteq W_t - \alpha \nabla J(W_t) \quad (3.59)$$

The formula expresses the concept of making small changes toward the direction of the gradient where the magnitude of the change is set by the step size parameter to avoid stepping too far. This update rule, in fact, guarantees only a decrease locally the function, reaching in this way a local minimum. Gradient Descent is guaranteed to converge at least to a local minimum.

Recalling equation 3.57 of the Mean Squared Value Error, it is time to compute the gradient of that since it will be used as target function to be minimized:

$$\nabla \sum_{s \in \mathcal{S}} \mu(s) [v_\pi(s) - \hat{v}(s, W)]^2 \quad (3.60)$$

According to the rules of calculus, the gradient can be pulled inside the sum and computed for each term of the VE, where only \hat{v} is a function of w . At this point expression 3.60 will look like:

$$\sum_{s \in \mathcal{S}} \mu(s) 2[v_\pi(s) - \hat{v}(s, W)] \nabla \hat{v}(s, W) \quad (3.61)$$

3.4.1 Stochastic gradient descent

Computing the gradient for the mean squared value error requires summing over all states which is generally not possible and it's also very likely that the distribution μ is unknown. It is so necessary to approximate the gradient. Also, without explicitly have μ , it is possible to sample state just following the policy π . That sample can be used to do an update to reduce the Value Error. Here's an example with the state s_1 :

$$w_2 \doteq w_1 + \alpha [v_\pi(s_1) - \hat{v}(S_1, W_1)] \nabla \hat{v}(s_1, W_1) \quad (3.62)$$

This updating approach is called stochastic gradient descent because it only uses a stochastic estimate of the gradient. It can be seen as a noisy approximation to the gradient that is much cheaper to compute but can nonetheless converge to a minimum. This sampling procedure is a very effective way to estimate the gradient but there's still a problem to be issued: the policy π from which the state is sampled is unknown. One possible solution is adopting the so-called *Gradient Monte Carlo Algorithm* where the value of the state under policy π is estimated through samples of the returns for each visited state. In this way, the expectation of the gradient is still equal to the gradient because the expectation of the return is the state value.

$$w \leftarrow w + \alpha [G_t - \hat{v}(S_t, W)] \nabla \hat{v}(S_t, W) \quad (3.63)$$

When dealing with large state space, learning the value of each of them could become computationally not convenient, that's where makes sense to use the strategy called *State Aggregation* where some states are considered equal. This method is an example of linear function approximation because there is one feature for each group of states. That being said, the gradient is equal to the feature vector. So, during the update, only the weight related to the active group of states will be modified.

In Gradient Monte Carlo, the return was used instead of the state value under π but any estimate of the value can be used. To generalize, equation 3.63 is rewritten with the variable U_t as estimate:

$$w \leftarrow w + \alpha [U_t - \hat{v}(S_t, W)] \nabla \hat{v}(S_t, W) \quad (3.64)$$

U_t can be an unbiased estimate of the true state value, as in the case of Gradient Monte Carlo where the return is used. In this way the converge to a local optimum is secured. Otherwise, U_t can be replaced with a bootstrap target such as the one-step TD target. This estimate of the return is defined biased because the TD target uses the current value estimate that is not the true value function and can be not accurate, so the convergence to a local minimum is not guaranteed.

$$U_t \doteq R_{t+1} + \gamma \hat{v}(S_{t+1}, w) \quad (3.65)$$

In this case, the approach is called Semi-Gradient because computing the gradient of the Value Error using the TD target as value estimate it is not possible to pull inside the VE the gradient because U_t depends itself on the weights and so the final expression would be different with respect to the TD update. Even if this strategy cannot be called Stochastic Gradient Descent, it shows convergence in many cases relevant to RL. Here's the update rule of the weight at each time step:

$$w \leftarrow w + \alpha [R + \gamma \hat{v}(S', w) - \hat{v}(S, w)] \nabla \hat{v}(S, w) \quad (3.66)$$

In conclusion, TD semi-gradient descent is not guaranteed to converge but if it does, it is much faster than Monte Carlo because it does not have to wait for the end of the episode and furthermore it has lower variance updates. These characteristics make TD very valuable because it performs well in early learning which usually is the most desired feature because there is never the chance to run for asymptotic performance.

Linear function approximation is a special case when talking about function approximation because it is a simple but effective way to estimate values. Recalling the definition of linear TD update just saw where the TD Error has been highlighted:

$$w \leftarrow w + \alpha \delta_t \nabla \hat{v}(S_t, w) \quad (3.67)$$

$$\delta_t \doteq R_{t+1} + \gamma \hat{v}(S_{t+1}, w) - \hat{v}(S_t, w) \quad (3.68)$$

In the linear case, the estimated state value is just the inner product between the feature vector and the weight vector:

$$\hat{v}(S_t, w) \doteq w^T x(S_t) \quad (3.69)$$

And so the gradient will be just the feature vector:

$$\nabla \hat{v}(S_t, w) = x(S_t) \quad (3.70)$$

Make the update rule to look like this:

$$w \leftarrow w + \alpha \delta_t x(S_t) \quad (3.71)$$

So the weight is updated in the direction of the feature vector times the TD error. The larger the feature, the larger the impact on the update. In the extended form, the update rule would be:

$$w_{t+1} \doteq w_t + \alpha [R_{t+1} + \gamma \hat{v}(S_{t+1}, w_t) - \hat{v}(S_t, w_t)] x_t \quad (3.72)$$

It is worth noticing that tabular TD is just a special case where the feature vector has only one component different from zero and so the update becomes a scalar computation. But why so much care on such trivial methods? Even if this approach seems far from reality because too simple mathematically speaking, if matched with the knowledge of some experts in the field of application who help in designing the features, can become a very powerful tool [6]. With some mathematical rearrangements, equation 3.72 can be rewritten as:

$$w_{t+1} = w_t + \alpha [R_{t+1} x_t - x_t (x_t - \gamma x_{t+1})^T w_t] \quad (3.73)$$

From this equation, two new parameters can be defined:

$$\mathbf{b} \doteq \mathbb{E}[R_{t+1} x_t] \quad (3.74)$$

$$\mathbf{A} \doteq \mathbb{E}[x_t (X_t - \gamma x_{t+1})^T] \quad (3.75)$$

The TD update can be rewritten again as the expected return plus a noise term which means that it can be well approximated by the expectation itself. The equation then looks like:

$$\mathbb{E}[\Delta w_t] = \alpha(\mathbf{b} - \mathbf{A}w_t) \quad (3.76)$$

The weights are said to converge when this expectation is equal to zero and that values if the weights are defined as:

$$w_{TD} = \mathbf{A}^{-1}\mathbf{b} \quad (3.77)$$

This solution is called *TD Fixed Point* and it can be proven that this is the point where TD converges. It is now reported the inequality that relates the minimum VE solution and the point found by TD.

$$\bar{V}E_{(w_{TD})} \leq \frac{1}{1-\gamma} \min_w \bar{V}E(w) \quad (3.78)$$

To conclude, the two values are not equal because of bootstrapping under function approximation. If the estimate of the next state is persistently inaccurate, then the update is always done towards an inaccurate target.

3.5 Deep Reinforcement Learning

In the previous paragraphs, it has been explained how function approximation is a key step in Reinforcement Learning but only the simplest way of approximation, i.e. linear approximation, has been presented. On the contrary, one of the most used methods nowadays is Neural Networks (NN). In the following a brief overview of NNs will be presented and how they work before seeing the several ways to apply them in a RL algorithm.

Figure 3.4 shows the overall structure of a NN. Every circle represents a *node* and the black solid lines are the connections between nodes. The nodes are organized into layers, in particular, the first one on the left where the input data enters the network is called the Input Layer, while the last layer on the right, which is usually made by just one node where the value to be approximated is delivered, is called Output Layer. All the layers in between these two are named Hidden Layers. The one in the figure is denoted as a Feed-forward NN because the stream of data always moves on through several layers, otherwise, it would be called a Recurrent Neural Network (RNN). When data is passed through a

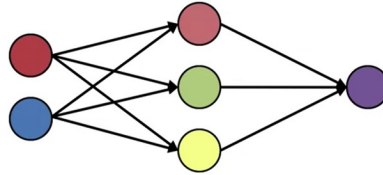


Fig. 3.4 Neural Network layout [6].

connection, a weight is applied so that the node can sum each of the weighted inputs and apply some activation function to that sum. The activation function is usually non-linear such as:

- Sigmoidal function $f(x) = \tanh(x)$
- Rectified Linear units or ReLu $f(x) = x \text{ if } x > 0; \quad 0 \text{ otherwise}$
- Threshold function $f(x) = 1 \text{ if } x > 0; \quad 0 \text{ otherwise}$

A NN is a parametrized function made of a collection of interconnected nodes where the output of a layer is the input of the following one if that is not the output layer.

To build a NN, some prior knowledge is useful to define some usually fixed parameters such as the number of layers, the number of nodes in each layer, and the activation functions. Furthermore, the weights of the nodes must be initialized and their initial value has a great relevance. In fact, every node will have different weights and so will pass to a non-linear function a different input, and the output of each node will represent a *feature* of the network. The weights become the indicator of the activation of each feature. Coming back to the architecture, when talking about NN with the term *depth* it means how many hidden layers compose the network. However, according to the *universal approximation property*, it is not necessary for NN to have several hidden layers, one is enough to approximate any continuous function given that it is sufficiently wide. Anyway, increasing

the depth allows the composition of features, for instance, starting with low-level features in the input layer and going to more and more combined features in order to accurately approximate complex functions. The depth can also be exploited by introducing the concept of a bottleneck layer where each successive layer contains fewer nodes than the previous one. The output layer will have the lowest amount of nodes and will contain the key details of the function needed for the prediction. Having defined the overall structure and functioning of a NN, it is time to talk about the so-called training of the network which is finding the parameters which minimize the loss function. Figure 3.5 proposes some notation:

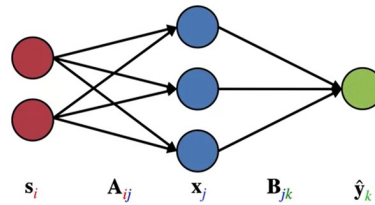


Fig. 3.5 Neural Network notation[6].

Where s_i are the inputs, x_j the features, \hat{y}_k the output and A_{ij} and B_{jk} two sets of weights. The mathematical explanation of how gradient descent can be applied to back-propagate the error from the output layer to the hidden ones, and so update the weights, is present in the literature but will not be reported here because too complex and out of the scope of this thesis. The authors' aim is to provide an overall understanding of the basic concepts of RL to better understand how it works and so choose with consciousness the best tools.

After going through some basic concepts of Neural Networks and having mentioned the relevance of features, it is time to move on and leave behind TD and value prediction with a fixed policy to talk about control with function approximation with Sarsa. Switching from state to action values will look like:

$$q_{\pi}(s, a) \approx \hat{q}(s, a, w) \doteq w^T x(s, a) \quad (3.79)$$

From equation 3.79 it is clear how features are now supposed to represent action as well. This can be accomplished by stacking features which means repeating the same set of state features for each action. At each update, only the features related to a certain action will be active. This stacking procedure is not only possible with linear approximators but also with NN where this is usually done generating multiple outputs, one for each action value as figure 3.6 shows. Just like in stacking, the weights for each action do not interfere with the others.

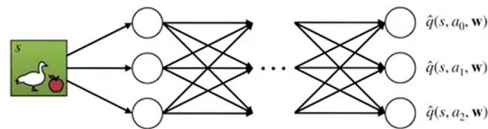


Fig. 3.6 Neural Network stacking[6].

Alternatively, if the goal is to generalize over action in the same way it is done over states, there would be as input both the states and the actions with one only node at the output with the approximate action value. This approach can be visualized in figure 3.7.

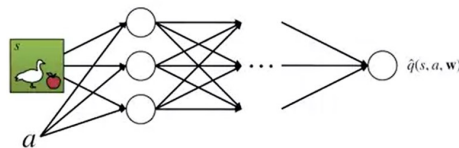


Fig. 3.7 Neural Network action generalization[6].

The update rule for Sarsa with function approximation will look like:

$$w \leftarrow w + \alpha (R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, w) - \hat{q}(S_t, A_t, w)) \nabla \hat{q}(S_t, A_t, w) \quad (3.80)$$

If instead Expected Sarsa is preferred, the equation becomes:

$$w \leftarrow w + \alpha \left(R_{t+1} + \gamma \sum_{a'} \pi(a' | S_{t+1}) \hat{q}(S_{t+1}, a', w) - \hat{q}(S_t, A_t, w) \right) \nabla \hat{q}(S_t, A_t, w) \quad (3.81)$$

Finally, it is easy to obtain from Expected Sarsa the update rule for Q-learning with function approximation:

$$w \leftarrow w + \alpha \left(R_{t+1} + \gamma \max_{a'} \hat{q}(S_{t+1}, a', w) - \hat{q}(S_t, A_t, w) \right) \nabla \hat{q}(S_t, A_t, w) \quad (3.82)$$

Action value estimates are also very useful when it is matter of promoting exploration.

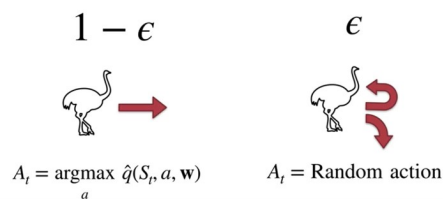


Fig. 3.8 Exploration with action values[6].

However, the ϵ Greedy policy is not a direct exploration method because it relies on randomness to discover better actions near states followed by the current policy which means that is not as systematic as exploration methods that rely on optimism.

So far, all the methods analyzed aimed to estimate state or action values in order to learn a good policy. However, a policy can be parametrized directly which means choosing an action based on the state of the environment without computing its value or an action value before picking the best action. Figure 3.9 is quite explicative in comparing the parametrized policy with a parametrized approximation function.

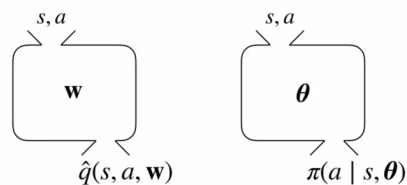


Fig. 3.9 Parametrized policy and function approximation[6].

The state and action are the inputs and, as w was used to indicate the weights added to the input of an approximation function, the letter θ is used to indicate the parameters of the parametrized policy that will output the probability of taking that action in that state. The policy π must be a legit probability distribution which means it has to satisfy the two following requirements:

- The probability of selecting an action has to be greater than or equal to zero;
- For each state, the sum of the probabilities over all actions must be equal to one.

The Soft-max Policy is an effective way to satisfy these statements:

$$\pi(a|s, \theta) \doteq \frac{e^{h(s,a,\theta)}}{\sum_{b \in \mathcal{A}} e^{h(s,b,\theta)}} \quad (3.83)$$

The function h is called the action preference and it should not be confused with the action value, it is just an indicator of how much the agent is prone to

select a certain action. The action preference is a parametrized function that can assume whatever kind of approximation (linear, NN, ...) because the constraints are respected by how π is built. In fact, the exponential function at the numerator guarantees the positivity of the probability for each action, while the sum at the denominator is meant to normalize the output of each action to make all the probabilities sum up to one.

The reader may wonder why it wasn't enough to estimate the action values and then search for the optimal policy, so now all the advantages and downsides of learning directly a policy will be evaluated. First of all, the agent can make its policy more greedy over time to allow higher exploration at the beginning when the estimates are not accurate. Then, as the learning progresses, the policy can converge to a greedy deterministic policy. Starting stochastically also allows to get out from possible situations where the agent is stuck in a state due to a poor function approximation. Acting randomly helps the agent to complete the task. Finally, directly learning a policy sometimes can turn out to be much easier than evaluating all the state-action pair values of a complex environment.

Now, as previously done for action-value-based methods, it is necessary to specify an objective and estimate its gradient. When directly learning a policy, choosing an objective is more straightforward than it was with value-based methods since the reward itself is the objective, to be maximized in the long-term in this case. Undiscounted or discounted rewards can both be used as objectives depending if the task is episodic or continuing. In the following is reported the formula of the *averagereward*, another type of reward not mentioned before because out of the interest for the author, but now useful to understand how to set the objective.

$$r(\pi) = \sum_s \mu(s) \sum_a \pi(a|s, \theta) \sum_{s', r} p(s', r|s, a) r \quad (3.84)$$

Now the goal is to define a policy that maximizes the reward. The idea is to estimate the gradient of the objective with respect to the policy parameters and adjust them based on the estimates. This approach is called *Policy Gradient Method*. Computing the gradient is not that easy, unfortunately, this is because by changing the policy the distribution μ changes too. To solve this issue is suggested to apply the *Policy Gradient Theorem* that turns the gradient of equation 3.84 into equation 3.85:

$$\nabla r(\pi) = \sum_s \mu(s) \sum_a \nabla \pi(a|s, \theta) q_\pi(s, a) \quad (3.85)$$

Where the gradient of the policy π tells how to adjust the parameters to increase the probability of a certain action in that state. On the other hand, the associated action value gives a sort of weight in the sum over the action probability gradient. Again, computing the sum over states is impractical so the gradient is going to be approximated thanks to a stochastic sample of it. The updates are done from the states observed while following policy π . The stochastic gradient descent update will look like this:

$$\theta_{t+1} \doteq \theta_t + \alpha \sum_a \nabla \pi(a|S_t, \theta_t) q_\pi(S_t, a) \quad (3.86)$$

Where S_t is the sampled state under policy π . With some mathematical computations it is possible to get rid of the sum over actions and turn it into an expectation over policy π , transforming equation 3.86 into:

$$\theta_{t+1} \doteq \theta_t + \alpha \frac{\nabla \pi(A_t|S_t, \theta_t)}{\pi(A_t|S_t, \theta_t)} q_\pi(S_t, a) \quad (3.87)$$

Or even better:

$$\theta_{t+1} \doteq \theta_t + \alpha \nabla \ln \pi(A_t|S_t, \theta_t) q_\pi(S_t, a) \quad (3.88)$$

Equation 3.88 should now look familiar to the reader because it is very close to many of the learning equations seen up to now. The policy is known and so is its parametrization therefore computing the stochastic gradient of the objective to proportionally adjust the parameter θ is now easy. Then, a step size parameter α is used to control the magnitude of the step in the direction of the gradient. To complete the update equation, it is finally necessary to compute the action value. Unfortunately, q_π is unknown and so must be approximated in a variety of ways, for instance in the following expression is adopted a TD algorithm.

$$\theta_{t+1} \doteq \theta_t + \alpha \nabla \ln \pi(A_t|S_t, \theta_t) [R_{t+1} - \bar{R} + \hat{v}(S_{t+1}, w)] \quad (3.89)$$

In equation 3.89 it is hidden the role of the well-known Reinforcement Learning algorithm named *Actor – Critic*. The parametrized function \hat{v} is a learned estimate of the value function and it represents the critic part of the Actor-Critic algorithm. To train the critic, any state-value learning algorithm can be adopted.

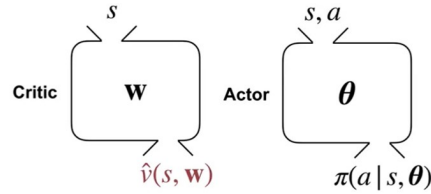


Fig. 3.10 Actor and critic.

On the other side, there's the parametrized policy which constitutes the actor part that uses the policy gradient updates shown in the equation above. Making a little step forward in the update formulation, it's possible to subtract from the next state value estimate the current value estimate. In this way, it is obtained the TD error inside the square brackets.

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha \nabla \ln \pi(A_t | S_t, \boldsymbol{\theta}_t) [R_{t+1} - \bar{R} + \hat{v}(S_{t+1}, w) - \hat{v}(S_t, w)] \quad (3.90)$$

It might be a little confusing but with a simple theoretical demonstration it can be proven that adding the estimate of the current state, which behaves as a kind of baseline, to the update rule does not change the value of the expectation of the update. This step is useful because it is now possible to rewrite the two update rules present on an Actor-Critic algorithm as a function of the TD error in order to highlight how this approach works.

$$w \leftarrow w + \alpha^w \delta \nabla \hat{v}(S, w) \quad (3.91)$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^\theta \delta \nabla \ln \pi(A | S, \boldsymbol{\theta}) \quad (3.92)$$

In Figure 3.11 it is possible to see a visual representation of the functioning of an Actor-Critic algorithm. At each time-step, an action is taken according to the policy, and the next state and reward are obtained from the environment.

With this data, the TD error is computed and the estimate of the return is updated. In the same way, the weights of the value functions and the parameters of the policy are updated using the policy gradient update. The TD error is used as an indicator of how good the action taken by the actor was. If the value is higher than what is expected by the critic, the policy parameters are changed in a way that increases the probability of taking that action in that state. On the other hand, if the critic is disappointed by the reward obtained from a certain state-action pair, then its probability is reduced. The actor and the critic are constantly learning by interaction: the actor is always changing the policy to exceed the critic's expectation while the critic is constantly updating the value function to evaluate the changing policy. This loop continues forever if not externally interrupted, that's why this approach is particularly suitable for continuing tasks.

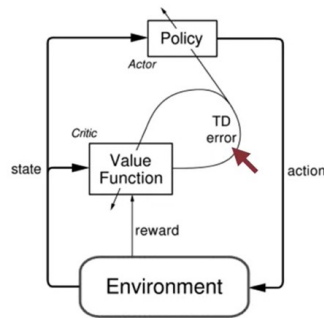


Fig. 3.11 Actor and critic loop[6].

3.6 DDPG, TD3 and SAC

All the RL fundamentals have been covered so far but, as the reader could guess, there are plenty of different algorithms living under the same Machine Learning branch. Dealing with episodic or continuing tasks, working with discrete rather than continuous spaces for action or even states, learning policy directly or estimating state-action value and then greedify, and much more different approaches have pushed during the last decade to the development of several

types of agents. In the following, some of the most promising, performant, and cutting-edge agents, suitable for the study case, are illustrated.

The first agent proposed is called DDPG, where the acronym stands for *Deep Deterministic Policy Gradient*. It is an actor-critic learning agent where the actor and critic networks are implemented using deep NN to better handle high dimensional state and action spaces. It searches for an optimal policy that maximizes the expected cumulative long-term reward. One feature worth remarking on is the type of the spaces: the state space can be both continuous and discrete but the action one must be continuous. This type of action space represents an additional complexity to the algorithm and how it handles data but it's necessary if it is wanted to improve the quality of the action-taking capability of the agent. DDPG is called a model-free agent because it interacts directly with the environment and does not rely on some simulated dynamics of the environment in a structure called model. When dealing with complex scenarios, trying to build an internal representation of how the environment works is almost unfeasible, that is why rather than planning through a model, the agent learns by trial and error. Obviously, as a key feature of every RL method, it is an online algorithm that learns while interacting, gaining a proper experience as a human being would do. Furthermore, it is an off-policy algorithm. The theory behind this concept has already been explained but, for the sake of completeness, it consists of behaving following a certain policy but estimating state values according to another policy called target. In particular, DDPG works with four networks as function approximators: an actor, a target actor, a critic, and a target critic. This configuration allows to update the parameters of the parametrized target policy and the weights of the target critic increasing the stability of optimization. The learning process of this agent is characterized by a kind of experience replay technique: the stream of data coming from the interaction of the agent with the world (state, action, reward, next state) is stored in a replay buffer. Then a mini-batch is sampled randomly from the buffer during training to increase the learning stability.

The direct evolution of DDPG is the agent named *Twin – Delayed Deep Deterministic (TD3) Policy Gradient*. Most of the underlying structure is common between TD3 and its predecessor because they are both model-free, online, off-policy reinforcement learning agents that search for an optimal policy that maximizes the expected cumulative long-term reward. Also, for this method, the state space can be both continuous and discrete while the action space is

only continuous. As an improvement of the DDPG, the TD3 brings mainly three modifications to the algorithm:

- It learns two Q-value functions (i.e. the critics) and uses the minimum value function estimate during policy updates;
- It updates the policy and targets less frequently than the Q functions;
- It adds noise to the target action.

To limit a possible overestimation of the value function, TD3 uses four networks for the critic part of the agent, two for the behavioral critic, and two for the target critic. Then, at each timestep, each critic computes its estimate of the state value but the agent only takes the minimum between the two behavioral critics and the minimum among the two target critics. This is an attempt to decrease a bit the critics' expectations on the value function. Both the couples of critics can have critics with different structures of the networks but they work better if the same architecture is shared. If this last case holds, the weights must be initialized in a different way and, as a mathematical rule, the number of target critics must always match the number of critics. The learning process of the agent is quite the same between the DDPG and the TD3 which still relies on a replay buffer and on a sample of it to compute the stochastic gradient and update the network parameters, both for the value function and the policy. The only additional step introduced by TD3 is the perturbation of the chosen action using a stochastic noise model to make the actor less likely to exploit actions with a high Q-value estimate, especially because this agent as DDPG adopts a deterministic policy.

The third and last agent presented is the latest evolution of DDPG and TD3 called *Soft Actor – Critic (SAC)*. Even if the core structure is shared with the two agents presented, this algorithm hides some huge novelties. Starting from what it has in common, it is an actor-critic agent working online with an off-policy approach in a model-free environment. Again, the state and action spaces are the same as before, but the first relevant difference is that now the actor adopts a stochastic policy to pick an action. Previously, with a deterministic policy, the agent related to each state one and one only action. With stochastic policy, things are quite different. The action generation starts as always with the reception of a state as input to the NN whose role this time is not approximating the most suitable action but the mean value and the standard deviation of a Gaussian distribution. If earlier the policy parameters update were meant to improve the

agent performance in action selection, this time the updates have the role to shape a Gaussian distribution that step after step becomes always narrower around its mean that usually is the action with the highest value. This approach implies that, during the training of the agent, the actor will never choose greedily with respect to his action. Of course, when testing the agent, the actor's policy can be switched to deterministic to let it exploit what it has learned during training at its best. Using a probability distribution for action selection provides the algorithm an intrinsic randomness because the action is sampled from the distribution. Being a Gaussian distribution unbounded it could lead, even if with very low likelihood, to the selection of some very poor actions, that's why once the action has been sampled from the curve, the tangent function and then a scaling function is applied to bound and re-scale the action. Figure 3.12 shows a diagram visually representing what has just been covered.

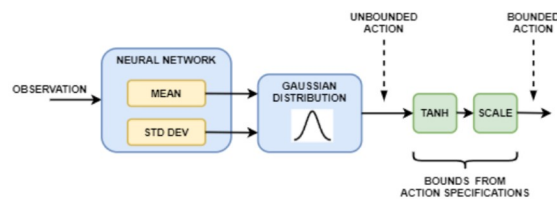


Fig. 3.12 How the stochastic policy is created [7].

Moving to the core of the agent, it still relies on two sets of two critics each for the behavioral and target part of the critic side. Concerning the actor, the duality between behavioral and target actor is abandoned in favor of a configuration that sees only one actor since the agent selection procedure contains implicit stochasticity. Anyway, the main novelty introduced by SAC has not already been mentioned. This RL agent is looking for an optimal policy that not only maximizes the expected cumulative long-term reward but also the entropy of the policy. In a more general way, entropy can be defined as "... the measurement of uncertainty over the value a random variable can assume" [26]. In this context, entropy is the measure of selecting an action in a certain state according to a

policy. It should be clear how this concept perfectly fits with a stochastic policy because there's no uncertainty around a deterministic policy. The search for a policy that maximizes both the reward and the entropy is formalized by the so-called "Max-ent formulation" reported here below:

$$\max_{\pi} \mathbb{E} \left[\sum_{t=0}^H r_t + \beta \mathcal{H}(\pi(\cdot|s_t)) \right] \quad (3.93)$$

Equation 3.93 aims to maximize, under policy π , the expectation of the cumulative long-term reward and the entropy function \mathcal{H} of the action to be selected in state S_t under policy π . The entropy is weighted by a trade-off factor β that will be updated during learning based on the obtained returns. Adding this term to the expected value to be maximized introduces, without doubt, a sub-optimal behavior in the short-term period because it enhances the randomness around the action, but this formulation should be seen as a long-sighted approach. In fact, this entropy not only makes the policy more robust in the case of change in the environment but also improves learning by promoting an exploratory behavior that permits the collection of way more data about the world in the long run.

Chapter 4

Energy Management System

4.1 Introduction

As has been shown so far, with the great potentialities of Hybrid Electric Vehicles also comes a much higher complexity than a conventional vehicle. This is the product of having two different propulsion systems, with two completely different energy sources, that propel the car. The primary energy source is generally the chemical energy stored in the fuel used by the Internal Combustion Engine. As previously explained, providing an ICE vehicle with an Electric motor always adds some more Degrees of Freedom (DoF) in terms of operating modes independent of the architecture or the position of the EM. These additional DoFs create the potential of a very efficient vehicle with the additional cost for a sophisticated high-level controller rather than just a low-level one, as it was for a conventional ICE vehicle [27]. This additional controller is called Energy Management System (EMS) and it is composed of two parts, as shown in figure 4.1[1][28]:

- Supervisory controller: decides the best operating mode considering the operating conditions of the vehicle and the powertrain;
- Energy Management System: decides the power split among the actuators based on the vehicle's operating conditions, the powertrain's ones, and the information coming from the supervisory controller.

The EMS represents the cutting-edge technology that administers the mutual interaction between the electric motor, the internal combustion engine and the

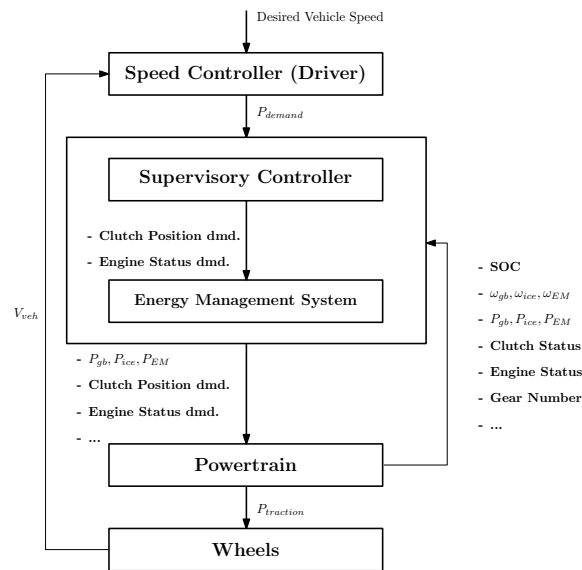


Fig. 4.1 The role of energy management system in a hybrid electric vehicle [1].

energy storage system. It decides when and how to enable certain modes to optimize energy consumption, enhance vehicle performance and reduce the pollutant emissions from the ICE and its after-treatment devices. How to satisfy the road request while optimizing the energy path along the driveline is the role of the EMS. Thanks to several sensors that measure various vehicle parameters while the vehicle is running, the EMS is able to compute the best working mode at each time instant through complicated algorithms. Exploiting the optimal operating area for the ICE, taking advantage of the strengths of an electric motor while considering the battery State of Charge (SoC), maximizing energy recovery and regenerative braking while ensuring driving performance and pleasure for the driver are all the objectives of the EMS.

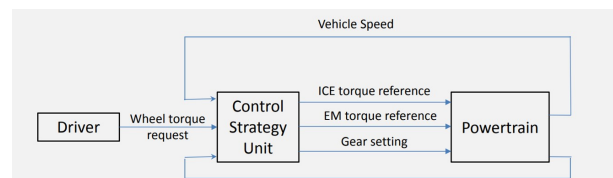


Fig. 4.2 Control loop on board[2].

The EMS output, and so the parameter it uses to control the powertrain behavior, is the instantaneous torque delivered and the torque split among the available resources.

It should be now clear that the optimization problem has a very large number of design variables (like powertrain architectures and components) and control parameters (like operating modes, transmission control policy, braking policy. . .) and so developing EMS algorithms necessary means facing a large-scale numerical problem. Mathematically speaking optimization is the process of minimizing (or maximizing) an objective function J subject to some constraints on the design variables or control parameters. Within this thesis, the focus will be on finding the best control strategies and so the authors will try to optimize the vehicle operation instant by instant with given architecture and powertrain elements specifications, so with fixed design variables.

4.2 Conventional strategies

4.2.1 Rule-based strategies

This category is the only solution for an EMS to handle the energy path without an explicit optimization of a sort of objective function (or cost function). These strategies only imply a set of heuristic methods based on engineering considerations on the best operating mode for the vehicle in certain particular conditions[29]. It is worth noting that this solution is quite far from being optimal and it is also very sensitive to the vehicle parameters but they represent a good solution because they are easy to implement and computationally light with respect to formal optimization problems[30][31]. Anyway, these methods are very commonly applied on marketed vehicles.

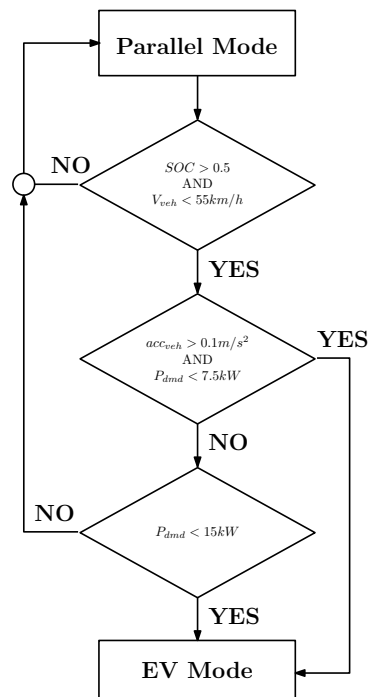


Fig. 4.3 An example of rule-based control[3]

In figure 4.3 it is represented an example of the logic behind the EMS based on heuristic methods. Rule-based strategies rely on a set of logical rules, usually “if-then” statements, that an algorithm exploits to properly split the power among the energy sources according to the current driving conditions. Based on some relevant parameters such as the battery SoC, the vehicle speed, the load required by the driver at the accelerator pedal and others. During the vehicle operation, the EMS gathers information from several sensors about the relevant parameters needed and then, following the pre-set rules, it determines the working mode for the powertrain and so the power allocation. The greatest advantage of this approach is its simplicity: straightforward to implement and based on engineers’ know-how about vehicle functioning. On top of that, this control method is applicable in real-time since the computation effort needed to make a decision is almost negligible and so the system can easily and quickly react to any change in the driving scenario. Being computationally cheap, this approach was applied and still is, to cost-effective HEV models because it does not need costly and extensive computational resources. However, the limitations of this solution are not of minor relevance, that is why a better solution has been deeply investigated and much effort has been put to apply it on vehicles with more sophisticated and performant algorithms. Among all, the lack of adaptability is probably the most striking. The

rules are simple and cover the majority of the common driving operations of a vehicle but they struggle to adapt to complex scenarios and transient conditions, providing as a result, poor exploitation of the energy sources. This is one of the first reasons that causes the algorithm to be sub-optimal, but also not being able to properly read the driving scenario or to predict the upcoming driver behavior is another one. They lack optimality if compared to other solutions where the energy management at each instant is based also on the prediction of the current driving conditions. Furthermore, tuning these empirical rules for a certain vehicle, and for certain scenarios, can be very demanding and even without a real optimal solution. In conclusion, these methods are certainly a way to handle energy on board a HEV but different, and hopefully better, ways must be found.

4.2.2 Optimization-Based strategies

The key feature of this category of approaches is that they all strive to find the best possible outcome for the task they are designed for, by minimizing or maximizing a certain objective function also called cost function. Dealing with the energy management of a Hybrid Electric Vehicle, the cost function is usually defined as dependent on the fuel consumption, battery SOC variation, pollutant emissions, after-treatment device temperature and other parameters, either by themselves or as a combination of them [2]. With the minimization of a cost function, the ultimate goal for this method is to achieve optimality in the exploitation of the energy stored on board by properly splitting the power between the electric motor and the internal combustion engine. The optimization can be global or applied online and under these two labels [32], different optimization methods can be identified: the Dynamic Programming (DP) [33] and Pontryagin's Minimum Principle (PMP)[34] are global optimization methods while Equivalent Consumption Minimization Strategy (ECMS)[35] or Model Predictive Control (MPC)[36] are online methods.

Global optimization methods

The two methods mentioned above belonging to this category are called global because by considering the entire driving cycle they try to find the optimal control trajectory that minimizes the cost function. This aspect gives them the chance to actually reach the optimal control strategy for a certain driving cycle because the

optimization process relies on the complete knowledge of the task, all the future actions are known in advance so it is possible to compute the best sequence of actions, and so the power split among the two onboard sources in this case, in order to achieve the minimization of the cost function. The two main approaches of this category are now briefly analyzed.

- Dynamic Programming

"The principle of optimality suggests that an optimal policy can be constructed in piecemeals, first constructing an optimal policy for the "tail subproblem" involving the last stage, then extending the optimal policy involving the last two and continuing in this manner until an optimal policy for the entire problem is found."
Bellman, 1949 Principle of Optimality [33].

Dynamic programming (DP)[33][23] is used to solve multi-stage decision problems with the aim of minimizing a cost function. The problem is structured by identifying the state variables $x(t)$, those representing the state of the system and its evolution in time, the control variables $u(t)$, those chosen to be the way through which the system behavior is controlled and the ones that the algorithm can use to reach optimality, the cost of a state $g(x(t), u(t))$ which represents the value to be minimized at each time instant. At the end of the task, all the instantaneous costs will be summed in the total cost that depends on a precise control trajectory and so the trajectory of the minimum total cost is called the optimal trajectory.

$$\text{Total cost } J(x_0, u_0, \dots, u_{N-1}) = g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, u_k) \quad (4.1)$$

$$\text{Optimal cost } J^{opt}(x_0) = \min_{u_k \in U_k(x_k)} J_k(x_0, u_0, \dots, u_{N-1}) \quad (4.2)$$

$$\text{Optimal control strategy } [u_0^*, \dots, u_{N-1}^*] = \arg \min_{u_k \in U_k(x_k)} J_k(x_0, u_0, \dots, u_{N-1}) \quad (4.3)$$

Now, it has been mentioned the so-called *tail sub – problem* which means solving the search for the optimal strategy starting from the last stage, the *tail*, and moving back until the initial state is reached. This method is particularly effective since the optimal cost for the considered sub-problem is found by minimizing the sum of the current state and the optimal cost-to-go which is the optimal cost computed in the previous computation stage. In

this way is possible to build the optimal control strategy iteratively solving a tail sub-problem, moving off one stage at a time. The whole optimization process can be subdivided into two main phases: the backward phase and the forward phase.

As just completely explained, the first phase is meant to solve the tail sub-problem, increasing the *length of the tail* step by step and storing the optimal value of the cost function and the optimal control sequence for each iteration and for each possible state of the system. For all x_N set:

$$J_N^{opt}(x_n) = g_N(x_N) \quad (4.4)$$

For each $k = N-1, N-2, \dots, 0$ evaluate iteratively:

$$J_k^{opt} = \min (g(x_k, u_k) + J_{k+1}^{opt}(f_k(x_k, u_k))) \quad \forall x_k \quad (4.5)$$

Then, for the forward phase, the algorithm starts from the initial state $x(0)$ of the system and uses the stored value of the optimal cost function in order to identify the related control trajectory and proceeding with the simulation until the last instant of the task. For each $k = 0, 1, \dots, N-1$ evaluate:

$$u_k(x_k) = \arg \min (g(x_k, u_k) + J_{k+1}^{opt}(f_k(x_k, u_k))) \quad (4.6)$$

and advance the simulation:

$$x_{k+1} = f_k(x_k, u_k) \quad (4.7)$$

When dealing with HEV and Energy Management System [37], the most used state variables are the battery SOC and the engine state while, for the control parameter, it is straightforward using the gear number and the power split factor α which is the ratio between the torque of the electric motor and the total torque required by the road. These must not be taken as rules but they are a good starting point to understand how to treat our specific case with Dynamic Programming.

- Pontryagin's Minimum Principle

Starting from the Euler-Lagrange equations including constraints on the control inputs, the minimum principle of Pontryagin (PMP) generalizes those concepts[34]. To dive a little bit the principle, let's start with the

definition of the Hamiltonian function H :

$$H(x, u, p) = F(x(t), u(t)) + p^T f(x, u) \quad (4.8)$$

where p is the co-state vector. The objective with this approach would be of minimizing a performance index J defined as:

$$J = \Phi(x(t_f)) + \int_{t_0}^{t_f} F(x(t), u(t)) \quad (4.9)$$

where the dynamics of the state depends on the state itself and on the control variables and $x(t_0) = x_0$ and $x(t_f) = x_f$. The optimal control strategy will be:

$$u^*(t) = \arg \min_{u \in U} (H(x, u, p)) \quad (4.10)$$

And the necessary condition for u to be the optimal control action is:

$$\dot{p} = - \left(\frac{\partial H}{\partial x} \right)^T \quad (4.11)$$

For an HEV application, the PMP optimization problem can be defined as follow:

$$J = \int_{t_0}^{t_f} \dot{m}_f(u) dt \quad \text{subject to} \quad SoC(t_0) = SoC(t_f) \quad (4.12)$$

What these two approaches have in common is in the limitation of offline applications only. This is due to two main reasons:

- They rely on the knowledge in advance of the whole driving cycle (more in general on the future information of the task itself) and so they are not applicable online, in driving scenarios never met before, because they could not be provided with future data if not based on predictions and so not compliant with the optimality principle.
- Especially for DP, the computational cost rises quickly with the increasing number of states and control variables. The more the state is accurately represented by many variables, the more reliable will be the optimal control trajectory identified. The same holds if the variables are highly discretized or even continuous. On one side, the solution will be always more accurate but, as a shortcoming, the number of computations will grow rapidly facing

the so-called *Curse of dimensionality*[2]. For this reason, the computational burden is too high to apply these solutions online.

Online optimization method

Also known as *Static optimization method*, they are the answer of the automotive industry to the limitations of the global optimization methods. Instead of requiring information on the whole task, from the beginning to the end, these methods try to optimize the power split by only relying on the short-term time horizon of the past and, if possible, of the future. In this way, optimization turns from a global to a local one.

- Equivalent Consumption Minimization Strategy (ECMS)

One of the most known approaches is the ECMS which aims to minimize instantaneous fuel consumption using powertrain information to estimate the energy flow. As it is possible to easily understand from the name, this method exploits the concept of an *Equivalent Fuel Consumption* where, thanks to an *equivalence factor*, the energy consumption of the battery is summed to the actual fuel consumption coming from the ICE. This sum represents the function the algorithm strives to locally minimize realizing a real-time optimization. In the following is reported the formula of the equivalent fuel consumption:

$$\dot{m}_{f,eq}(t) = \dot{m}_f(t) + \dot{m}_v(t) \quad (4.13)$$

where obviously, \dot{m}_f is the mass flow rate of the fuel and \dot{m}_v is the *virtual* fuel consumption coming from the electrical energy used can be defined as follow:

$$\dot{m}_v(t) = \frac{s(t) \cdot P_{batt}(t) \cdot p(SOC(t))}{Q_{lhv}} \quad (4.14)$$

where:

- $s(t)$ is the Equivalence Factor converting the battery energy into fuel energy
- $P_{batt}(t)$ is the instantaneous power provided or absorbed by the battery
- p is the *penalty function* which is a correction defined by the difference by the current $SOC(t)$ and the target value
- Q_{lhv} is the fuel lower heating value [MJ/kg]

The concept of considering the electrical energy expenses as fuel consumption is that, if the vehicle is propelled by the EM, solely or in collaboration with the engine, later during the cycle, that energy will be restored by the engine itself in a recharging phase so increasing the fuel consumption with respect to the one needed to satisfy the required load. This is because, most of the time an HEV operates in Charge-Sustaining (CS) mode and so the initial energy present in the battery should equal the one at the end of the cycle plus or minus a certain tolerance. This does not hold for Plug-In HEV which usually runs in the combination of Charge-Depleting (CD) and CS modes.

The performance of the logic behind the ECMS holds in its equivalence factor. That constant establishes the degree of conversion of the battery energy into fuel energy:

- if the factor is too high, the usage of the EM is discouraged and so the controller mainly relies on the engine not exploiting as it could the potentiality of an HEV;
- if the factor is too low, the cost given to the battery energy is low enough to often promote the usage of the EM leading to an advanced battery depletion.

From this reason comes the main limitation of the ECMS, where the equivalence factor should be properly tuned on the specific task and cannot fit different driving cycles, otherwise it would output very poor results. The ECMS is applicable online because it does not require future information to run over a driving cycle but, concerning the equivalence factor evaluation, it requires information about the vehicle mission profile. To overcome this issue, Adaptive-ECMS (AECMS) has been proposed as a solution to avoid the need for information about the driving scenario because the equivalence factor is updated according to the current driving conditions.

- Model Predictive Control (MPC)

At every time instant the current state and inputs are considered as the initial condition of the optimization problem. From that data, the prediction of the short-term future and then a finite-time open-loop optimization problem is solved online in the so-called rolling optimization phase. The first action of the optimal strategy evaluated at that time instant is applied to the system under control. Then the procedure is repeated at each time instant using

as a starting condition the new state of the system in a sort of feedback correction phase. MPC-EMS[38] is considered one of the most promising control strategies due to its reasonable computational cost and ease of control effect. It is characterized by the main following advantages:

- Thanks to a short-term prediction of the vehicle operating mode, it responds to the change in the power demand by improving the vehicle's performance. This benefit can be obtained by applying different procedures like the k-nearest neighbor method to identify the current driver model and so predicting future parameters or proposing a robust fuzzy MPC scheme.
- While optimizing the energy split in the short-term predictive range, MPC is also able to consider multi-objective functions. In fact, the search for an optimal control depends also on the definition of the cost function that can contain different goals such as fuel consumption minimization, battery degradation cost, reduction of engine start and stop events. . .
- Integrating in the optimal control strategy the system constraints is fundamental like making the ICE and EM work at the allowed speed and power. Anyway, the dynamic constraints also play a relevant aspect in the optimization process, the value of the SOC is one of the main constraints for an HEV, especially if a real fuel consumption reduction is targeted. These constraints are easily implementable in MPC.

4.3 Current AI-based control strategies

During the last twenty years, loads of step forward has been made in terms of applicability and performance when talking about EMSs for HEVs. From the first rule-based strategies, the optimality has constantly increased all the way to the recent introduction of Artificial Intelligence (AI) algorithm[39]. The huge study and advances carried out in the AI field, summed with the enhancement on the computational platforms provided to vehicles, has brought great interest in the application of Machine Learning (ML) techniques to improve the HEV energy management.

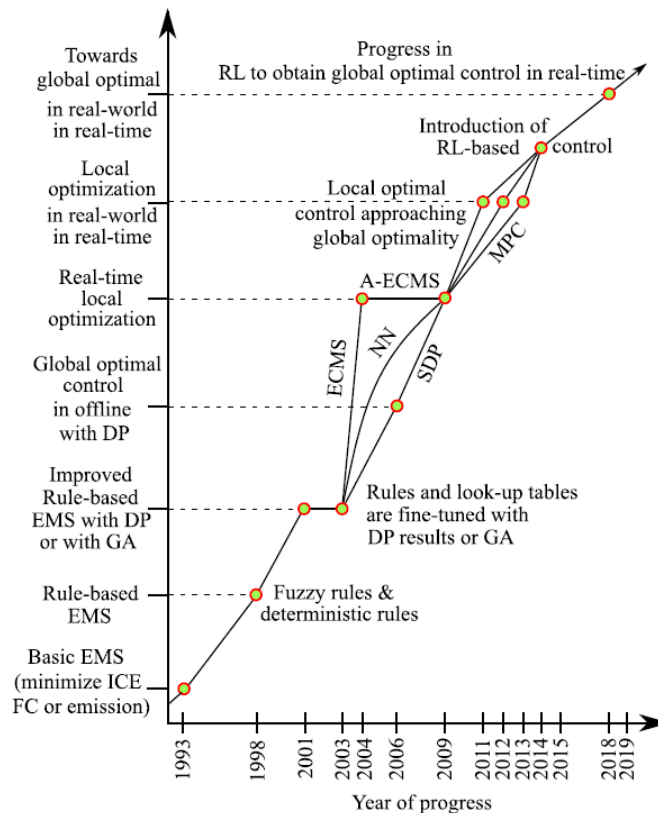


Fig. 4.4 Evolution of EMSs from 1993 to 2018 [8].

AI methods seem promising not only because they can be applied online since they don't require any data about the driving cycle, but also because they aim to reach global optimality rather than a local one as it was for ECMS or for MPC. They are able to process vast amounts of data and they can learn how to quickly adapt to changes in the environment where they are asked to operate, from urban congestion to highway cruising. Based on all the information gathered at every instant, AI approaches can identify the current driving scenario and accordingly the most suitable driving mode and the best power allocation. Any kind of ML has been applied to solve the energy management problem, from Supervised Learning, Unsupervised Learning and Reinforcement Learning.

Furthermore, AI-powered EMS can be not only *passive* in orchestrating the energy flow just by reading the current driving conditions but they can be proactive in the energy management by predicting future driving scenarios. By analyzing and combining different data such as traffic patterns, topography and weather forecasting (or current conditions) they can optimize the power allocation

and choose the most suitable operating mode to better deal with the upcoming events[40].

Integrating this kind of tool into HEV EMS has immense potential but it is still at the very early stage and plenty of work must be done to achieve a good accuracy of the models, create robust and reliable algorithms, handle the computational complexity and properly collect the huge amount of real-time data most of these sophisticated algorithms require. Anyway, the horizon has never been so promising for EMS of HEV where AI technology represents a huge leap forward towards always more efficient algorithms able not only to use the HEV potential more effectively but also to enhance the driving experience.

Chapter 5

Test case

5.1 Vehicle specification

As previously mentioned, the thesis work aims at developing a supervisory control for a Hybrid Electric Vehicle. For this reason, the selected vehicle used as the base to create its digital twin is a Mercedes E300de, a state-of-the-art diesel PHEV available in the European market. The powertrain layout is schematically shown in figure 5.1.

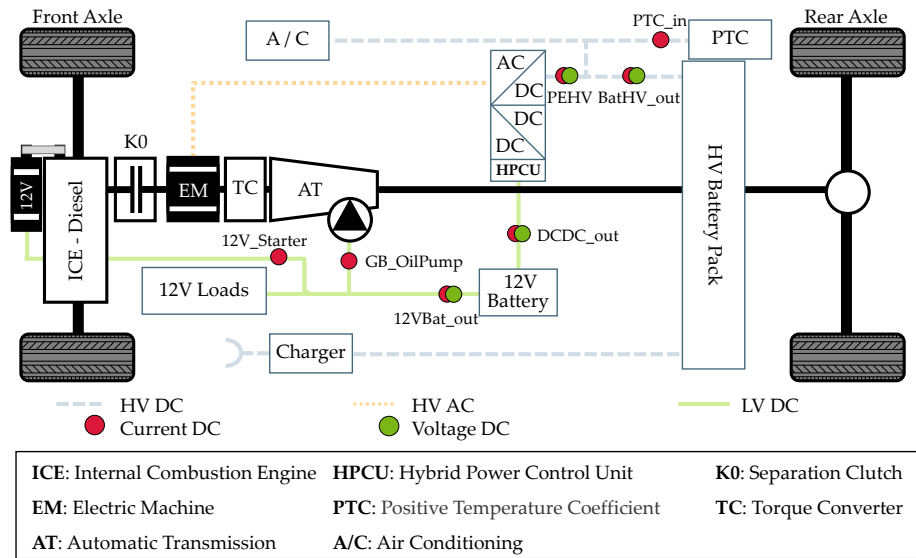


Fig. 5.1 Powertrain layout: a diesel engine is connected through an auxiliary clutch to an EM. Both the ICE and the EM are connected to the transmission by means of a torque converter.

Table 5.1 Vehicle and powertrain main specifications.

Vehicle	Curb Weight [kg]	2060
	Power [kW] @ 100 km/h	14.9
Transmission	Type	9-AT w/ Torque Converter
	Type	In-line 4 cylinders Turbo Diesel
ICE	Displacement [cm ³]	1950
	Max Power [kW] @ 3800 rpm	143
	Max Torque [Nm] @ 1600-2800 rpm	400
	Compression Ratio	15.5:1
EM	Type	PM Synchronous Motor
	Max Power [kW] @ 2000 rpm	90
	Max Torque [Nm] @ 1750 rpm	440
	Max Speed [rpm]	6000
HV Battery	Type	Li-NMC
	Rated Voltage [V]	365
	Capacity [kWh]/[Ah]	13.5/37
	Cooling System	Water Cooled

The propulsion system relies on a conventional 2.0-liter diesel ICE compliant with the Euro 6d-temp legislation coupled with a 90 kW Permanent Magnets (PM) synchronous EM through an auxiliary clutch (K0) realizing a P2 architecture. Both the ICE and EM drive the rear axle through a 9-speed Automatic Transmission (AT) with upstream a Torque Converter (TC). To feed the EM, a 13.5 kWh Li-Ion Nickel-Manganese-Cobalt-oxide (Li-NMC) HV battery is placed towards the back of the vehicle, guaranteeing a full-electric range of 54 km and a maximum speed in full-electric mode of 130 km/h. The Low Voltage (LV) loads, i.e. the 12V starter and the electrical oil pump for gearbox lubrication, and the 12V battery are connected thanks to a DC/DC converter to the HV battery that supplies them with electrical energy. All these features are summarized in table 5.1.

A PHEV is meant to be operated in two different modes:

- **Charge Depleting (CD):**
Usually enabled with high level of State of Charge (SOC) of the battery, it consists in propelling the vehicle with the EM only in the so-called full-electric mode and so depleting the charge present in the battery;
- **Charge Sustaining (CS):**
This operating mode is usually associated to an HEV with a small battery or to a PHEV that is out of the window to keep going in CD mode. When

the SOC hits a pre-set lower threshold, the ICE is switched on and a blend of both the ICE and the EM is adopted in order to maintain the SOC in a certain range.

Even if the combination of these two modes represents the best way to exploit all the potentialities of a PHEV, in this project only the CS mode will be considered to test the vehicle over standardized mission profiles.

To obtain the vehicle performance and all the possible powertrain operating conditions, it has been subjected to a massive experimental campaign where through reverse engineering procedures the various operating modes have been analyzed [41] [42]. The experimental measures have been carried out thanks to an All-Wheel Drive (AWD) dynamometer where the vehicle, equipped with Portable Emissions Measurement System (PEMS), was run on Real Driving Emission (RDE) scenarios. A Hybrid Power Control Unit (HPCU) connected to the HV inverter handles the energy flows between the EM and the HV battery pack and enables four possible operating modes according to the driver's power request and desire. A brief explanation for each of them is here reported:

- Hybrid Drive:
The ICE and EM work in cooperation to satisfy the road request, all the possible hybrid modes are available, from energy recovery to the electric boost meant to cover the transient response of the ICE;
- Electric Drive:
The vehicle is propelled relying only on the EM following what has been previously defined as CD mode. This modality is usually enabled when traveling at low speeds or in congested scenarios;
- E-Save Drive:
This mode sees again the ICE and the EM working in combination but this time ensuring the sustaining of the battery SoC;
- Charge Drive:
The road load is totally covered by the ICE which is asked to output an even higher power, running at a high-efficiency point, to charge the battery.

One final feature to point out is how the Hybrid Drive or Electric Drive is mechanically realized. Considering figure 5.1, the device in between the ICE and the EM is the clutch K0 that discriminates between the two modes. When

engaged, it couples the two propulsion systems that run synchronously to provide the necessary power. When the clutch is disengaged it relieves the powertrain from engine inertia and the EM drives the whole drivetrain in full electric mode avoiding the waste of dragging the ICE.

5.2 Mission profiles

Concerning regulatory driving cycles, i.e. WLTC and NEDC, they present a clear procedure to follow on the chassis dynamometer for type-approval purposes[43]. If the testing involves a PHEV, the guidelines defined in the UNECE Regulation 83 require two tests [44]:

- Condition A:
At the beginning of the test, the HV battery must be fully charged;
- Condition B:
At the beginning of the test, the HV battery must be fully discharged.

According to the guidelines of the legislation, Condition B has to be repeated both with a warm engine right after Condition A and a cold engine one day after Condition A. Table 5.2 shows the main characteristics of the most representative cycles used during the experimental campaign.

On the other hand, the RDE cycles shown in table 5.2, were performed to identify the control logic rather than to certify the vehicle performance over regulatory mission profiles. Some of the RDE cycles were carried out on the test bench, while others were run on public roads around the Italian city of Turin (these cycles are denoted with the symbol \mathcal{X} in table 5.2). All those RDE cycles allow to test the vehicle in a wide variety of driving conditions ranging from urban congested scenarios to motorway routes or even uphill and downhill rural roads.

By the way, the driving cycles database was not only used to evaluate the real vehicle performance and to run reverse engineering analysis on its powertrain operating conditions. Those cycles represent tasks and so episodes that can be used to train a RL agent. In the specific case considered, the environment will be made up of a virtual vehicle model running on a standardized cycle. As previously highlighted, RL is different from Supervised or Unsupervised Learning and it does not need a huge static database to learn but it directly interacts with the

Table 5.2 Characteristic values of the cycles performed during the experimental campaign.

Cycle	Time	Distance	Avg. Speed	Max Speed	Avg. Acc.	Max Acc.	Required Energy	Test Bench
	[s]	[km]	[km/h]	[km/h]	[m/s ²]	[m/s ²]	[Wh/km]	[–]
<i>NEDC</i>	1180	11	34	120	0.38	1.42	184	✓
<i>WLTC</i>	1800	23	47	131	0.41	1.84	222	✓
<i>RDE₁</i>	838	13	57	161	0.74	5.56	274	✗
<i>RDE₂</i>	4322	69	57	128	0.36	1.74	258	✗
<i>RDE₃</i>	4327	68	56	144	0.37	1.91	273	✗
<i>RDE₄</i>	6657	89	48	126	0.30	1.68	207	✓
<i>RDE₅</i>	6666	89	48	126	0.29	1.69	215	✓
<i>RDE₆</i>	5926	97	59	139	0.38	3.41	225	✗
<i>RDE₇</i>	5532	97	63	138	0.35	4.22	223	✗

world proposed. This means that it is enough to submit a cycle to the agent and let it run over the same cycle several times until a convergence of the reward is obtained. The agent training step will be deeply explained in section 6.3 but it is worth mentioning here that for that purpose only the WLTC is selected because representative of various real-world driving patterns. To what concerns the testing of the agent, the choice can be easily expanded to the other cycles of the database to assess the performance of the agent, especially when it is asked to interact in an environment that it has never seen before. This is the best practical test to evaluate the ability to generalize of the agent. Many more details on that in section 7.2.2.

5.3 Modelling approach

Nowadays, the majority of the analysis carried out at the production level of the vehicle component relies on simulation tools since an experimental approach implies an amount of time and costs no longer feasible. In this context, dealing with single components or limited-size subsystems for development purposes is quite usual: detailed simulation models, from 1D up to 3D-CFD, are often available and well spread among the community easing their usage and implementation. At the vehicle level, things are not that straightforward because even a 1D simulation model could result in excessive computational cost due to the system complexity and the length of the simulation itself. This is why, in these cases, energy analysis

at the vehicle level can better fit the task thanks to a look-up table approach to compute fuel efficiency and pollutant emissions [9].

The scope of this project is to compute the fuel consumption and the energy consumption of a digital twin of the vehicle under study. To do so, the vehicle must be modeled through mathematical equations in all of its components and evaluated over some standardized cycles following the vehicle-level energy analysis mentioned before. Since these methodologies are well-known in the field and this thesis work has not introduced any novelty in these terms, a brief overview will be presented as a reference to the approach adopted.

The vehicle is considered as a single-point mass lumped in the center of gravity of the vehicle. As a first and simplistic approach, the dynamics analysis is limited to just vehicle speed, acceleration, and power (or torque) demand. This means that the energetic approach of equation 5.1 is sufficient to properly describe the vehicle dynamics.

$$M_{eq} \frac{dv}{dt} = F_{pwt} + F_{brakes} + F_{roll} + F_{aero} + F_{grade} \quad (5.1)$$

$$M_{eq} = M_{veh} + \frac{2I_{wh,f} + 2I_{wh,r}}{r_{wh}^2} + \frac{I_{eng}}{r_{wh}^2} i_{fd}^2 i_{gb}^2 + \frac{I_{EM}}{r_{wh}^2} i_{fd}^2 i_{gb}^2 \quad (5.2)$$

Where F_{pwt} is the powertrain tractive force, F_{brakes} is the mechanical braking force, F_{roll} is the rolling resistance, F_{aero} is the aerodynamic resistance and F_{grade} is the resistant force due to the road slope [9].

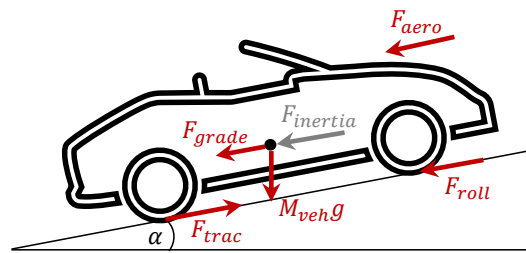


Fig. 5.2 Forces acting on a vehicle [3].

M_{eq} is the term taking into account all the real and equivalent masses of the vehicle. All the driveline components are part of the overall equivalent mass due to their inertia contribution. dv/dt represents the acceleration of the vehicle according to the second Newton's law.

The rolling resistant force can be simply modeled according to [16]:

$$F_{roll} = c_{roll} M_{veh} g \cos \alpha \quad (5.3)$$

Where g is the gravitational acceleration, α is the road slope angle and C_{roll} is the rolling resistance coefficient. This last term is the combination of different contributions coming from tire pressure, thread design, external temperature and, obviously, vehicle speed:

$$c_{roll} = c_0 + c_1 v + c_2 v^2 + c_3 v^3 \quad (5.4)$$

Where c_i are coefficients experimentally evaluated. Concerning the aerodynamic resistance can be expressed as:

$$F_{aero} = \frac{1}{2} \rho_{air} A_f C_d v^2 \quad (5.5)$$

Where ρ_{air} is the air density, A_f the vehicle frontal area and C_d the aerodynamic drag coefficient. The resistant force due to the road slope is modelled thanks to the following equation:

$$F_{grade} = M_{veh} g \sin \alpha \quad (5.6)$$

To limit the costs related to the estimation of the coefficients characterizing the vehicle, the simple but effective coast-down test is performed. The vehicle is brought to a certain speed in open space conditions and then free deceleration performances are measured. Properly neglecting the environmental conditions as possible bias of the measurements, the deceleration is considered as due only to the aerodynamic and rolling resistances and so, according to the instantaneous vehicle speed, the so-called Coast-Down coefficients [45] are estimated. It comes the following equation for the total drag force acting on the vehicle:

$$F_{roll+aero} = C_0 + C_1 v + C_2 v^2 \quad (5.7)$$

Where C_0 , C_1 and C_2 are the above-mentioned Coast-Down coefficients.

Considering the just seen equations of motion as the starting point for the vehicle virtual modelling, two different and well-known methodologies are de-

scribed in the following to evaluate the fuel consumption and other energy related parameters.

The first approach presented is called *Forward Dynamic Analysis* and it tries to replicate the dynamics of a real vehicle, including the driver. The longitudinal vehicle dynamics equations are solved starting from the driver's behavior. The vehicle speed profile described by the cycle is set as the target value that a Proportional-Integral-Derivative (PID) controller, representing the driver model, must reach acting on the accelerator or braked pedal and so generative a traction or braking torque. From this point the operating conditions of the vehicle are computed along with the engine speed and torque demand.

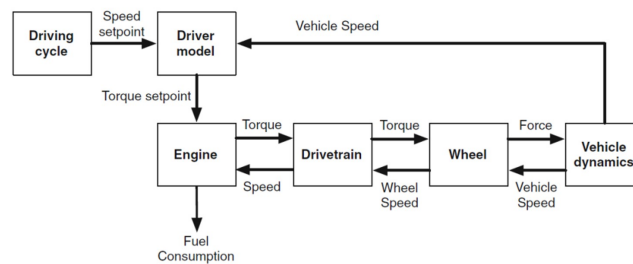


Fig. 5.3 Information flow in a forward simulator [9].

The other well-established methodology is called *Backward Kinematic Analysis* where the speed profile of the cycle is imposed on the vehicle, unlike the forward approach that uses that value as a target. The internal powertrain dynamics are neglected here because efficiency, power loss, and fuel consumption maps are used to model the components. Instead of using detailed 0D or 1D fluid-dynamics models, look-up tables are used in a quasi-static approach where the mission is divided into small intervals where speed, acceleration, and torque remain constant. Even if this methodology seems to be too simplified it is widely adopted because of its computational efficiency, furthermore, the fuel consumption and emissions predictions turn out to be acceptable in terms of accuracy, especially when a preliminary estimation is required.

Starting from the imposed speed and acceleration, it is possible to derive the engine, electric motor, and wheel speed thanks to a very simple kinematic



Fig. 5.4 Information flow in a backward kinematic approach [3].

relationship that neglects any kind of transient behavior and considers all the wheels in pure rolling conditions.

$$w_{wh} = \frac{v}{r_{wh}} \quad (5.8)$$

$$w_{gb} = w_{wh} i_{fd} i_{gb} \quad (5.9)$$

Where:

- w_{wh} is the wheel speed
- v is the vehicle speed
- w_{gb} is the inlet gearbox rational speed (electric motor side)
- r_{wh} is the effective wheel rolling radius
- i_{fd} is the final drive gear ratio
- i_{gb} is the gearbox gear ratio (depending on the actual gear)

Thanks to the equation of motion presented before, the power demand is computed. Once the kinematic relationship is established as shown in equations 5.8 and 5.9, the power flow is computed going backward along the driveline. For the sake of brevity, many of the mathematical steps are avoided and the final relationship is here shown.

$$P_{gb} = (f_0 + M_{veh}g \sin(\delta) + f_1 v + f_2 v^2 + M_{eq} a) v \eta_{gb}^{\text{sign}(-(f_0 + M_{veh}g \sin(\delta) + f_1 v + f_2 v^2 + M_{eq} a))} \quad (5.10)$$

Where P_{gb} is the power at the gearbox inlet which can be seen as the power request to the whole powertrain that the controller will split among the two on-board sources. C_0 , C_1 and C_2 are the coast-down coefficients, M_{veh} is the vehicle mass, M_{eq} is the equivalent mass of the vehicle considering all the driveline inertia

contributions as explained previously, α is the road slope, g is the gravitational force, a is the vehicle acceleration and η_{gb} is the gearbox efficiency computed thanks to a map interpolation.

Once the split between the ICE and the EM has been computed by the controller on the basis of the total power request, and the kinematic relation at the wheels has been back-propagated until the engine speed is computed, it is finally possible to evaluate the engine Brake Mean Effective Pressure (BMEP) according to the following formula:

$$BMEP = \frac{60 i P_{eng}}{nV} \quad (5.11)$$

Where P_{eng} is the fraction of the whole power demand requested to the engine, i is the number of revolutions per power stroke (i.e. 2 for four-stroke engines), n is the engine rotational speed in rad/s and V is the engine displacement. Knowing the BMEP, it is only necessary to consult the look-up tables and enter interpolating engine speed and mean effective pressure. In this way, the fuel consumption is obtained with a punctual computation based on a table.

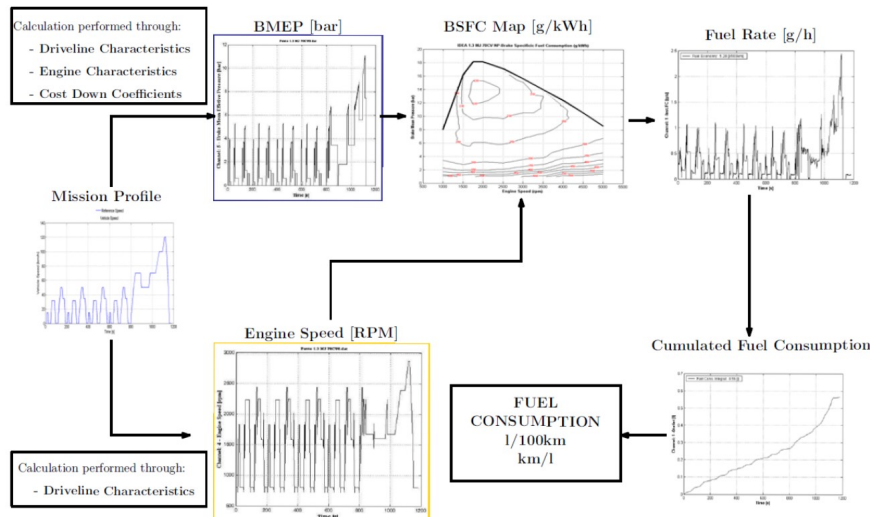


Fig. 5.5 Detailed information flow in a backward kinematic approach [1].

Seeking a simple model that allows most of the focus on the RL application, the backward kinematic approach has been selected and it will be adopted in this project work.

5.4 Simulation tools

AI seems to fit every kind of task, improving the human or machine capabilities to evaluate patterns, manage data and classify categories. That's why nowadays these kinds of algorithms are at the center of everyone's attention. However, apart from some fancy applications devoted to the public, their potentialities are not simple to be understood and applied. Furthermore, when dealing with Energy Management Systems, these mathematical tools represent a good opportunity to optimize the energy flow on an HEV where there is a higher number of degrees of freedom with respect to a conventional vehicle. In this context, the selection of the software environment is not a trivial issue. Not only it must be suitable for the creation of the simulation environment, i.e., the digital vehicle twin running on a standardized mission profile, but even more importantly for the implementation and training of a Reinforcement Learning algorithm. On the wide portfolio available, one of the most spread and well-known programming languages is MATLAB, short for Matrix Laboratory. Widely used in academia, research and industry, it is designed primarily for numerical computations aimed at mathematical modeling, algorithms development, data analysis and visualization. Thanks to its built-in toolboxes, which ease almost any kind of domain-specific task, and to its interactive and user-friendly environment, it has been largely adopted in engineering and scientific fields. When dealing with control systems simulations, experimental data analysis, physical phenomena modeling and complex systems algorithms development, MATLAB represents a guarantee to the result achievement. Despite all these good premises, it comes with some downsides, especially when compared to another powerful language that has gained popularity in scientific computing and data analysis: Python. Probably, one of the first benefits of switching from MATLAB to Python is due to the license cost. In fact, while MATLAB is a commercial software characterized by a costly license, Python is an open-source and free-to-use software, which makes it accessible to a much broader audience. Not with less relevance, the community and ecosystem around Python should be identified as another great advantage of this latter language. Especially when dealing with new tools like AI, relying on a larger community where thousands of users are trying to exploit them for the first time, makes a huge difference when developing new code. Furthermore, the rich ecosystem of libraries and infrastructure built around Python (some examples are NumPy, SciPy, Pandas and many others) provides similar functionalities to the toolboxes available in MATLAB. Moreover, in terms of flexibility and integration, Python is

unrivaled: being a general-purpose language, it can not only be applied to a wide variety of tasks but it also integrates well with other programming languages and tools, making it the preferred choice when involving various technologies and data sources. In contrast, MATLAB is very powerful with specialized tasks and therefore less versatile. Last but not least, libraries such as Tensorflow, PyTorch and scikit-learn have made Python a go-to choice. At the end of the story, even if MATLAB represents a robust and well-established environment, choosing Python can imply an improvement if seeking for a cost-effective solution with broader applicability and access to a vast array of libraries and add-on tools.

This thesis work aims to develop RL algorithms on Python. This software environment owes its success to the Objected Oriented Programming (OOP) which is a paradigm that focuses on modeling real-world entities using *objects*. Everything is an object in Python but, more precisely, an object is an instance of a class. Accordingly, a Python class can be seen as an object constructor, a sort of *blueprint* for creating objects. Defining a class means defining a set of possible objects that will inherit the same properties and methods, that are functions built-in in the object itself. Attributes and methods are said to be *encapsulated* in a class aiming at operating with the data provided to the class itself. The OOP promotes a modular design of classes which enhances code organization, reusing classes, modifying them and avoiding redundant code. This approach is also very useful to improve code readability since it creates a kind of abstraction that hides the complexity of the object's internal implementation. From outside it becomes easier to get what an object does rather than how it does it. In other words, OOP enables developers to mimic real-world entities through objects, making the code intuitive and self-explanatory. In this project, the main vehicle components used in a backward kinematic approach are modeled through classes.

As an example of how a Python class is built to mimic a real vehicle component, in the following the engine class will be analyzed in all its features. The part of the code defining that class, the subject of the discussion, is reported in Appendix A.1 to improve the explanation. From here on, only small parts of the code will be directly shown in the text while, for the sake of brevity, the longer ones are visible in Appendix A.

The `__init__` method is used to initialize the class properties according to the data given as input to the class. As can be seen from the first lines of code, those properties can be called with the method `self` which represents a way to call the class itself. The function `def` is used to define a method of the class, it

is possible to create a sort of built-in function that represents some aspects of the real-world entity the class is trying to model. Taking for instance the *Maxtrq* method, its function is to display the maximum torque allowed to the engine at a given rotational speed. In particular, the *LookUp1D* class, reported here below, is called in the output of the method to interpolate the values of a one-dimensional array, the engine torque limits array in this case.

```

1 class LookUp1D:
2     def __init__(self, x, y, var_info):
3         self.x = np.array(x)
4         self.y = np.array(y)
5         self.var_info = var_info
6
7         if len(x) != len(y):
8             raise("Not compatible dimension between
9                 x and y")
10
11     def interp(self, x_eval):
12         f = interp.interp1d(self.x, self.y)
13         y_eval = f(x_eval)
14         return y_eval

```

Similar are the subsequent methods, such as *FuelTrqMap*, where the only difference is that another class called *LookUp2D* is used to interpolate data over a two-dimensional map to extrapolate values, such as the fuel consumption in this case, as a function of two variables, torque and engine speed in this method.

Having explained how a class is built, it is now useful to see some examples of how it can be used. First of all, an object should be created, recalling that it is an instance of a class. This is done in the code here reported:

```

1 EN = Engine(Engine_data)
2 EN.Maxtrq(Engine_limits)
3 EN.Maxpwr(Engine_limits)
4 EN.Mintrq(Engine_limits)
5 EN.Minpwr(Engine_limits)
6 EN.FuelTrqMap(Engine_FuelTrqMap)
7 EN.FuelPwrMap(Engine_FuelPwrMap)
8 EN.BsfcTrqMap(Engine_BsfcTrqMap)

```

```

9 EN.BsfcPwrMap(Engine_BsfcPwrMap)
10 EN.EffTrqMap(Engine_EffTrqMap)

```

In the first line, an object called *EN* is created assigning to it all the properties and methods of the class *Engine* that takes as input the *Dataframe* (i.e. a data structure of the library *Pandas*) called *Engine_data*. Once the instance is created, all the methods are executed to create in the workspace of the software the structured data coming from their inputs and elaborated by the functions defined inside the method itself.

Once these steps are performed for each vehicle component, the class modeling the whole vehicle and its functioning is defined. Now, all the previously defined objects become the properties of the vehicle class as shown in the code below:

```

1 class VehicleModel:
2
3     def __init__(self, DC, WH, BT, EN, EM, FD, GB,
4                 Vh):
5         self.DrivingCycle = DC
6         self.Wheel = WH
7         self.Battery = BT
8         self.Engine = EN
9         self.Electricmotor = EM
10        self.Finaldrive = FD
11        self.Gearbox = GB
        self.Vehicle = Vh

```

Now, in the definitions of the dynamics of the system, the objects and their features are used to model the vehicle components' behavior, as the reader can see from the following lines of code:

```

1 def FuelConsumption(self, iceSpd, icePwr, ice_state)
2     :
3         fuel_rate = self.Engine.FuelPwrMap_tab.
4             interp(iceSpd, icePwr)
5         if ice_state == 0:

```

```
6         fuel_rate = 0
7     elif ice_state == 1 and iceSpd > self.
8         Engine.Cutofflimit and icePwr < 0:
9         fuel_rate = 0
10
11     co2_rate = fuel_rate * self.Engine.
12         CO2molarmass / self.Engine.Fuelmolarmass
13
14     #output
15     self.fuel_rate = fuel_rate
16     self.co2_rate = co2_rate
```

To better understand, an example of employing an object property is shown in the computation of the CO_2 rate where the molar mass of the carbon dioxide and of the fuel are recalled as properties of the engine object with the formula `self.Engine.CO2molarmass` and `self.Engine.Fuelmolarmass`. On the other side, for the computation of the fuel rate, the method `FuelPwrMap` of the engine object is used directly relying on its output map, as explained before, and interpolating the entering values of that map with the method `interp` of the class `LookUp2D`.

5.5 Model validation

Up to now, it was explained how the vehicle model was built up according to the theory behind a backward kinematic approach. As figure 5.6 shows, the vehicle has been modelled thanks to the connection between several *blocks*, each of them implemented as an *object*, a typical data structure in Python language.

As previously explained, the cycle is imposed on the vehicle and, moving backward over the driveline, all the physical quantities (i.e. speed, torque and power) are computed to finally evaluate the fuel consumption and the energy expenditure. However, the most important *block* of the vehicle model is still missing. The controller that, at any instant, allocates power to the ICE rather than to the EM has not been integrated yet. In this thesis work, the aim is to realize it through RL algorithms but, to make the results comparable, the model of the vehicle needs to be validated. For this purpose, a model based on the same real vehicle built on a MATLAB environment is available. In that model, the control

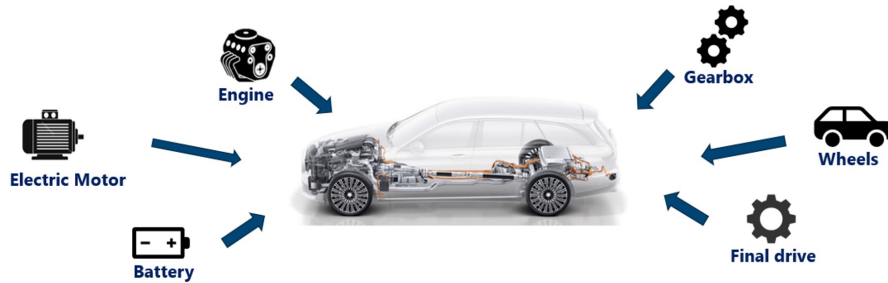


Fig. 5.6 The main vehicle's components are modeled through Python classes.

strategy is realized through the mathematical technique denoted as Dynamic Programming. It is particularly suitable for recursive algorithms where the goal is to find a *global optimum*. In the EMS framework, it is usually involved in setting a benchmark in terms of an optimal control strategy that, anyway, lacks real-time applicability due to the offline nature of DP. For this project, DP will serve not only as a target for the introduced RL agent but will also set a common control strategy between the MATLAB and Python models to carry out the validation of the latter.



Fig. 5.7 Dynamic Programming is used as the algorithm to evaluate the optimal control strategy.

Once the optimal control strategy identified by the DP is delivered to the Python model, some variables of interest are taken to validate the model with respect to the MATLAB baseline. Running back along the driveline, the first component selected to compare the two models is the power trend at the inlet of the final drive. In figure 5.8 is possible to see that power plotted against the speed profile imposed by the WLTC. Up to this stage of the driveline, the power trend of the two models is fully superimposable.

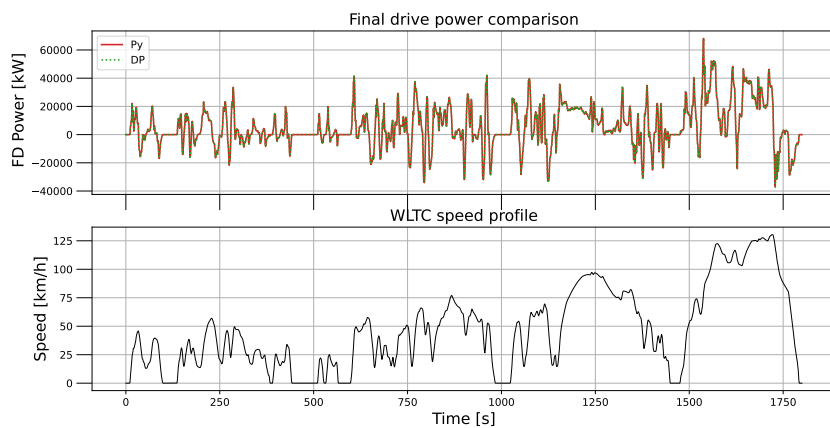


Fig. 5.8 Power at the final drive plotted against the speed profile.

Moving on, the same comparison is realized for the power level reported at the inlet of the gearbox, as shown in figure 5.9

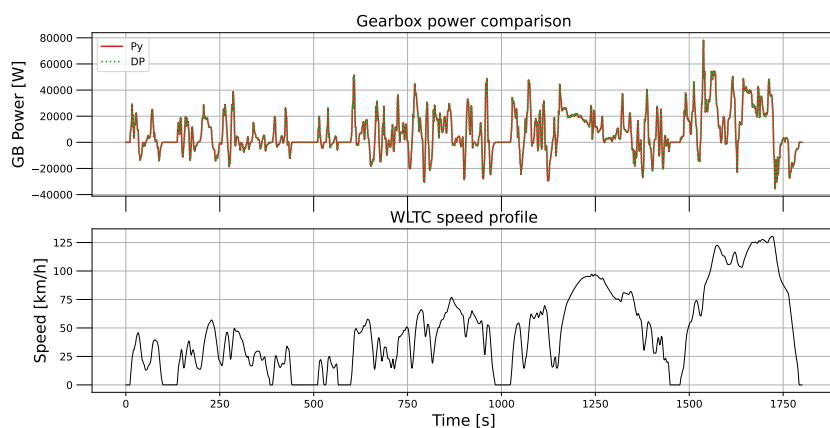


Fig. 5.9 Power at the gearbox plotted against the speed profile.

Despite the perfect resemblance between the two cases of the final drive power, which implies the correctness of the required torque and speed up to that point,

the power at the Gearbox is the first component where the two trends cannot be perfectly overlapped. Even if the graphs here above seem to match, a closer look at the power trend shows that some inaccuracy is introduced in the model of the gearbox. Figure 5.10 and 5.11 confirms what just stated.

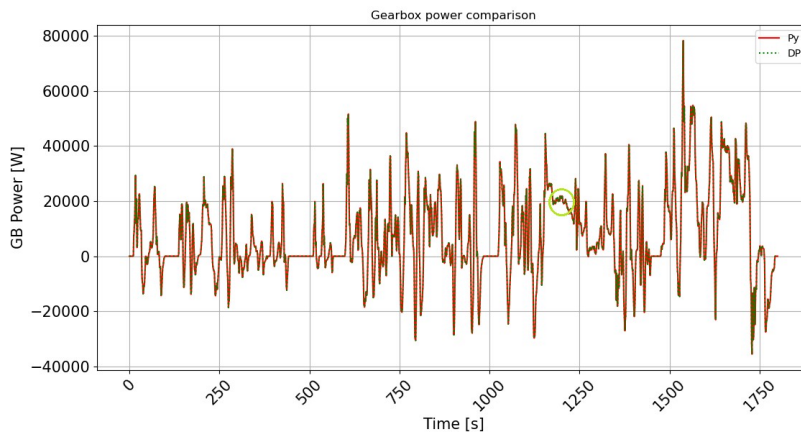


Fig. 5.10 Power at the gearbox where a discrepancy point is highlighted.

Figure 5.11 is the zoom of the graph reported in figure 5.10 inside the small green circle. It highlights the difference between the two models that might look almost negligible at this stage of the comparison but that must be considered as the possible source of future divergences that propagated from this point.

Obviously, the reason behind that divergence was investigated and identified as an issue with the gearbox efficiency which comparison is reported in figure 5.12 that shows the eta values at the gearbox related to the same instants of the cycle focused on the previous graph.

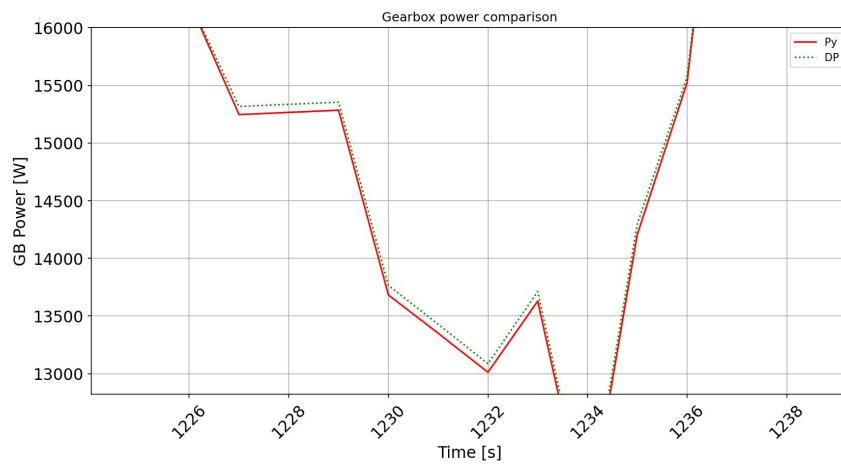


Fig. 5.11 Focus on the power at the gearbox where it is clear the difference between the model built in Python and the results of the DP from the MATLAB model.



Fig. 5.12 Focus on the gearbox efficiency.

This behavior is explained by the code implementation difference in the conversion of the Gearbox Efficiency map from the input to the output values. This correction is required because the values to enter the map are measured at

the input of the gearbox but in a backward approach, the entering values come from the output of the gearbox (as figure 5.13 tries to visually explain).

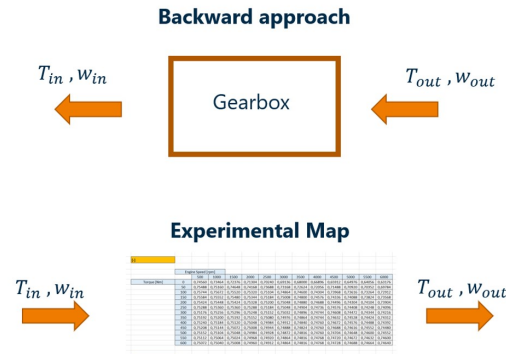


Fig. 5.13 Visual explanation of the inconsistency between the backward approach and the way the gearbox efficiency has been modeled.

The next variable used to assess the resemblance between the two models is the fuel rate. The two plots reported here below are overlapped, this is because the DP has as a control variable the Engine Torque and so its value is imposed. In other words, independently of the value of power measured at the gearbox inlet of the two models, the value of torque at the ICE is decided by the optimal control strategy found by DP and so it will not be affected by the deviation encountered at the gearbox. Therefore, the plot confirms that the model for the thermal source of the vehicle is accurate.

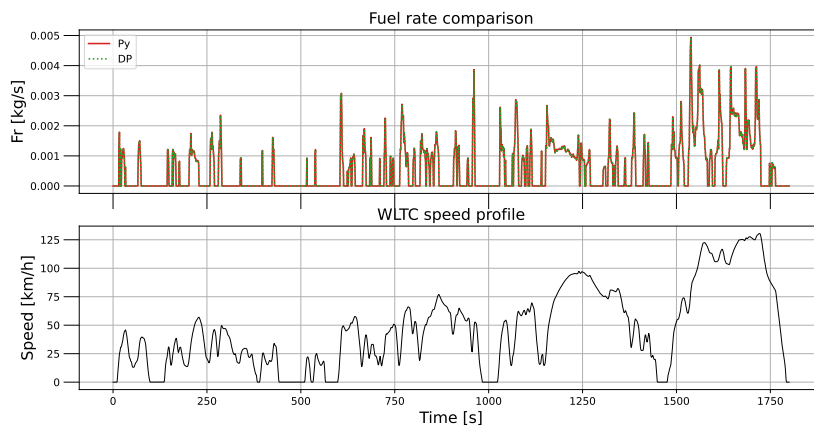


Fig. 5.14 Fuel rate plotted against the speed profile.

For the sake of completeness, cumulative fuel consumption is also involved in the comparison.

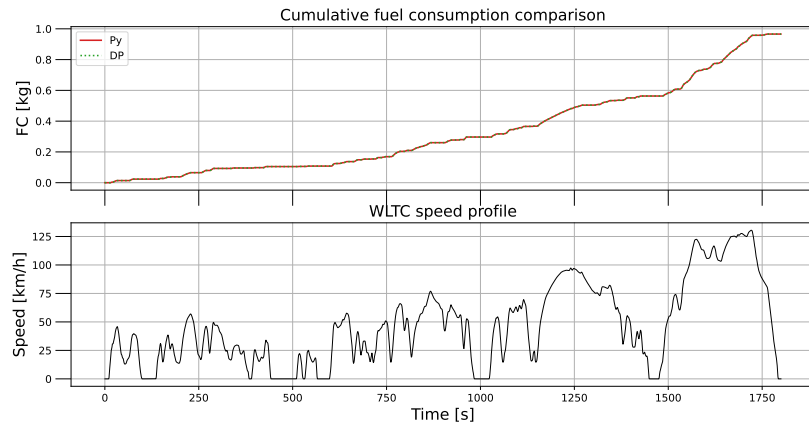


Fig. 5.15 Cumulative fuel consumption plotted against the speed profile.

Switching to the electric side of the on-board sources, the energy taken and restored in the battery is computed and the SOC is updated. Globally the trends are quite similar but not identical.

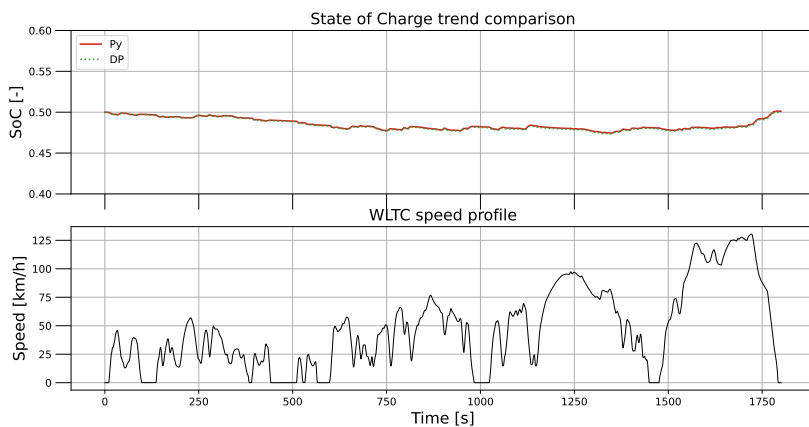


Fig. 5.16 SoC plotted against the speed profile.

Looking closer at the SOC fluctuations it is possible to notice the divergence better. This is due to the propagation of Gearbox Efficiency error since no other issues have been found. However, the magnitude of this inaccuracy is very little and, along the whole trend, the sign of the divergence is kept constant, hints that the source of this difference is always the same.

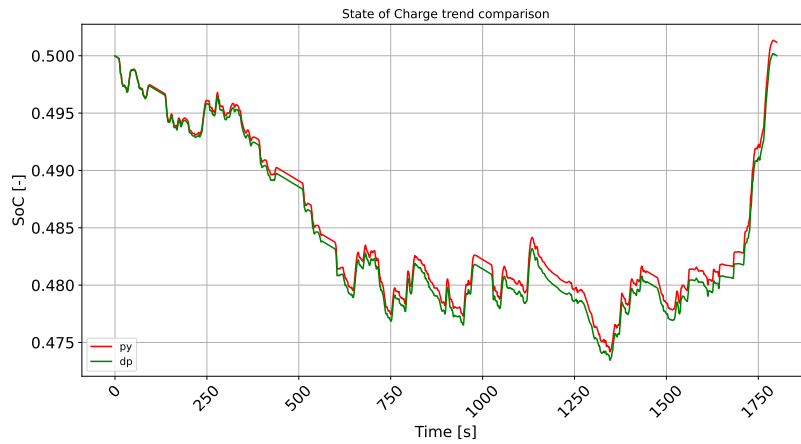


Fig. 5.17 A look closer to the SoC trend over the cycle.

At the end of the cycle, the vehicle model in python registers a value of the SOC that is 0.23% higher than the one of the MATLAB model. This very low difference can be neglected to the scope of this project and, from this point onward, the two models will be considered completely equivalent.

Chapter 6

EMS controller design

6.1 Agent selection

In section 3.6 three RL agents have been introduced: DDPG, TD3 and SAC. The DDPG is a well-known and largely employed agent thanks to its peculiar features such as the excellent behavior it shows with continuous action spaces, where traditional RL algorithms struggle due to the challenges posed by infinite possibilities. Moreover, the use of DNN allows the agent to handle high-dimensional state spaces, making it suitable for complex tasks such as the one this project is working with. Thanks to the off-policy updates, this agent turns out to be very sample efficient, good at learning and not prone to fall into suboptimal policies. In contrast, it can become unstable during learning, leading to an overestimation of the Q-value and a possible overfitting of the policy. In this context, more exploration is required. TD3 tries to overcome those shortcomings by using a twin-critics architecture, significantly reducing the overestimation bias present in single-critic structures. This results in more accurate and robust Q-value estimation. Furthermore, the introduction of a delayed policy update enhances the stability of optimization. To what concerns the exploration, even if a stochastic noise is used to perturbate the chosen action, this agent inherits the poor exploration strategy of the DDPG keeping unsolved the trade-off between exploration and exploitation. Moving to the last agent proposed, i.e., SAC, the introduction of a stochastic policy actor integrates exploration as a core feature of the agent structure itself, instead of relying on some additional noise, promoting the search for optimal actions. In addition, the entropy formulation creates a distribution over near-optimal solutions and so it not only improves the learning by collecting

more interesting exploratory data but also makes the policy more robust allowing the agent to find good actions even when the environment changes. Besides the addition of the entropy term reduces the total reward obtained in the short term, it makes learning more robust in the long run by collecting data according to a near-optimal policy distribution rather than following the exact policy. In conclusion, thanks to its promising characteristics and the results obtained by [46], the choice falls on the SAC agent.

6.2 Implementation

It should be now quite clear how powerful is the Reinforcement Learning paradigm in Machine Learning. The potentialities lying under the framework of an agent learning to make decisions by interacting with an environment are outstanding but the application of these tools is not as straightforward as someone could imagine. For this reason, the authors looked for some pre-built toolkits or libraries able to ease the implementation of these new algorithms. Overlooking the long research and comparison of different sources, the set of improved RL implementations called Stable Baselines 3 (SB3) was selected. This library provides a collection of high-quality and easy-to-use modules of RL algorithms. It is developed as part of the OpenAI Baselines project and it is meant to facilitate the application of this new AI algorithm branch. Also, it includes a great variety of agents, among which is obviously present SAC, and it represents a modular and flexible solution, able to support multiple Deep Learning frameworks such as PyTorch, TensorFlow and so on. . .

However, the agents provided by SB3 need to have a properly set environment to work. In fact, the more the environment these agents find is suitable for them, the easier will be their implementation and the quicker will start the training phase. Being the SB3 package based on the OpenAI framework, all its agents can be directly used with the Open AI's Gymnasium (or Gym) environments meant to train RL agents. Despite the wide range of environments available, the huge potential of this framework is allowing to set and use custom environments. As already stated, the environment in which this thesis works is the digital model of a real car running over a standardized mission profile. While conceptually, all that is needed is just converting the environment created into a Gym environment, this process turns out to be quite tricky because it has to comply with several rules to be "registered", and so recognized, as a true Gym environment. That's why the

first tough part of the SAC implementation from SB3 begins with the engineering around the *world* the agent will interact with. As already done for Python classes, the Gym environment definition starts with the initialization method reported in the following.

```

1 class Vehicle300de_P2Env(gym.Env):
2     """Custom Environment that follows gym
3         interface"""
4     def __init__(self, DC, WH, BT, EN, EM, FD, GB,
5                 Vh):
6         self.DrivingCycle = DC
7         self.Ts = self.DrivingCycle.Time[1]-self.
8             DrivingCycle.Time[0]
9         self.Wheel = WH
10        self.Battery = BT
11        self.Engine = EN
12        self.Electricmotor = EM
13        self.Finaldrive = FD
14        self.Gearbox = GB
15        self.Vehicle = Vh
16        '''self.Update = {"Batt soc": 0, "FD speed
17            ": 0, "GB speed": 0, "GB power": 0,
18                "ICE state": 0}'''
19        self.Output = {"SOC": np.zeros(1), "FC": np.
20            array([]), "EM_Power": np.array([]), "
21            ICE_Power": np.array([]), "GB_Power": np.
22            array([]), "FD_Power": np.array([]), "
23            BT_Power": np.array([]), "Action": np.
24            array([])}

```

As it is possible to see, the piece of code imported and pasted above defines all the classes needed to model the vehicle but, more importantly, this is also the time to set the observation and action space. The action taken by the agent consists of the normalized torque of the ICE and it has a continuous space ranging from 0 to 1. The observation space is made by five normalized continuous variables:

- The vehicle speed

- The powertrain demand (i.e. the power at the gearbox inlet)
- The powertrain speed (again at the gearbox inlet)
- The battery SoC
- The traveled distance over the total distance

The action and the observation spaces are chosen according to the author's know how of the subject.

Then, all the methods and functions describing the dynamics of the environment are reported in the class *Vehicle300de_P2Env* that fully represents the Gym environment where the SAC agent will explore. Once these preliminary steps are completed, two fundamental methods of a Gym environment must be defined: the step and reset methods. The first logical method to address is the reset method since this is what gets called for every new episode before any step is done. Its purpose is to set up the start of the environment as well as return the first observation from where the agent can start choosing actions. The code relative to the reset method is reported in Appendix A.2.

The function of the reset method is nothing but initializing all the parameters to the value they had at the beginning of the episode. Firstly, the time index is initialized to zero and then the method *setSimulationInput* is called to set all the variables coming from the standardized cycle to their initial value, according to the time index. The method is here reported:

```

1 def setSimulationInput(self, time_idx):
2     self.time = self.DrivingCycle.Time[time_idx]
3     self.time_perc = self.time/self.DrivingCycle.
4         Time[-1]
5     self.speed = self.DrivingCycle.Speed[self.time]
6     self.acc = self.DrivingCycle.Acc[self.time]
7     self.travel = self.DrivingCycle.dist_travel[
8         self.time]
9     self.travel_perc = self.travel/self.
10        DrivingCycle.dist_travel[-1]
11    self.gear = int(self.DrivingCycle.GearN[self.
12        time])
13    return self.time, self.time_perc, self.speed,
14        self.acc, self.travel, self.gear

```


In the reset method, the SOC and the ICE state are set to the initial conditions but they will be then the result of the action taken by the agent at each timestep. The observations related to the first time instant of this new episode are computed and then normalized to improve the stability during learning. To sum up, the only output of the reset method is the observation array and the entries *self.done* and *self.truncated* are set to false because the episode is just starting.

Subsequently, it is the turn of the definition of the core of the environment: the step method. It takes only one input, that is the action selected by the agent. That action is then used to update the environment's state and to evaluate the reward. The reward is defined inside this method (aside from some constant terms that were defined just after the initialization method) but how and why it is set in some ways is a matter of paragraph 6.5. When all the variables of interest are computed, the time index and all the cycle parameters are updated and the observation for the next state is evaluated. If the time index has reached the last time instant of the cycle, the *self.done* variable is set equal to true if the episode was not truncated before by some unfeasible condition. For the step method, the outputs are the observation array, the reward, and the status of the episode (i.e. done or truncated). The code relative to the step method is reported in Appendix A.3.

The last stage of the creation of a custom Gym environment is its registration in the Open AI framework. This is done very easily with just a couple of lines of code as shown in the code below:

```
1 # Register the environment
2 register(
3     id='Vehicle300de_P2Env-v0',
4     entry_point='Env:Vehicle300de_P2Env',
5 )
```

Some tools were used to check the compliance of the custom environment with the requirements to make it a real Gym environment. Once these verifications are carried out with positive results, it is sufficient to have a single line of code to generate the environment desired from the gym library.

```
1 env = gym.make(env_id, DC=DC, WH=WH, BT=BT, EN=EN,
                EM=EM, FD=FD, GB=GB, Vh=Vh)
```

With the environment defined and the required libraries imported, another line of code is enough to create the SAC agent model, as shown below:

```

1 model = SAC('MlpPolicy', env, verbose=1,
2           tensorboard_log=logdir, learning_rate=
3             LearningRate,
4             learning_starts=200, batch_size=BatchSize,
5             tau= Tau, gamma=Gamma, train_freq=
              TrainingFrequency,
              gradient_steps=1, target_update_interval=
              TargetUpdateInterval, policy_kwargs=
              policy_kwargs,
              ent_coef=EntropyCoefficient)

```

The environment registration, its generation and the RL agent model creation are an example of how applying SB3 implementations become simple when the custom environment is recognized as part of the Open AI Gym library. At this point of the code, all the necessary steps have been carried out and the training can start with just two commands: the first to launch the learning of the model over the environment and the second to save the model trained.

```

1 for i in range(21):
2     print(i+1)
3     #training
4     model.learn(total_timesteps=TIMESTEPS,
5                 log_interval=5, reset_num_timesteps=False,
6                 tb_log_name=agent_name, progress_bar=True)
7     model.save(f"{models_dir}/{TIMESTEPS*(i+1)}
8               /1800}")

```

The two functions are integrated into a for loop because the model trained after a certain number of episodes is saved before proceeding with the training. The training results and trials will be discussed in an upcoming section.

To complete the summary of how the algorithm was implemented, a quick overview of the testing procedure is now presented. Similarly to the training process, the first steps required are the registration of the environment and its creation through the method *gym.make*. After, instead of creating a new model,

the one trained that is the object of the test is loaded in the workspace as follows:

```
1 model = SAC.load(model_path, env=env)
```

Everything is already set to start the testing so the environment is reset and a while loop is established:

```
1 while not done:
2     action, _states = model.predict(obs,
3         deterministic=True)
4     obs, reward, done, truncated, info = env.step(
5         action)
6     episode_reward.append(reward)
```

Up to when the done variable is not set to true, the loop continues and the trained model is used to make predictions based on the observations. It is worth noting that during the testing phase, the stochastic nature of the actor is canceled in favor of a greedier deterministic policy. After that, a step forward in the cycle is done until the while loop does not break.

6.3 Training

The training is the most exciting as well as boring part: the code has been set to run hundreds of thousands of steps where the agent is asked to interact with the environment to learn, episode after episode, the best optimal control strategy. This may take a while. In the meantime, it is possible to observe the training trend in two different ways.

```

rollout/
  ep_len_mean | 1.8e+03
  ep_rew_mean | 1.67e+03
time/
  episodes    | 45
  fps         | 34
  time_elapsed | 258
  total_timesteps | 81045
train/
  actor_loss   | -82.6
  critic_loss  | 0.432
  ent_coef     | 0.0126
  ent_coef_loss | -0.278
  learning_rate | 0.0003
  n_updates   | 80844

```

Fig. 6.1 One of the logger displayed on Python console during training.

Every time a model is saved, a logger appears in the Python console and it provides some information about how the agent is doing. A random logger is shown in figure 6.1. It reports some parameters related to the training to let the user better understand what's going on. The mean episode length is constant in the task addressed in this project because, apart from some possible deviation of the SOC out of the allowed boundaries (i.e. between 0.2 and 0.8), the episode consists in the standardized cycle defined by WLTP with a duration of 1800 seconds. On the other hand, the mean value of the cumulative reward of each episode is an indicator of how well the agent is performing. Keeping in mind that the algorithm is working with a normalized reward, scoring 1670 as the mean reward value means that the agent is close to converge to a saturation point. In fact, being the reward normalized, the timestep set equal to one second and the model trained over the WLTC that lasts 1800 seconds, the maximum mean reward cannot overcome the value of 1800. Other useful parameters are the number of episodes and the total timesteps elapsed when the logger is visualized. The actor loss and the critic loss are values representing how good the estimate of the expected reward or the value action was. The entropy coefficient is instead the β factor at that time instant. Remember that the entropy is self-adjusted by the agent during training according to some internal parameters depicting how good the agent is doing during training.

The other way to watch what's happening during training is by plotting the values just described. The tool used to log model performance is a tool of TensorFlow called *Tensorboard*. With this visual instrument is possible not only

to assess the agent's behavior but also to compare several models. Some of the diagrams plotted on Tensorboard are shown here.

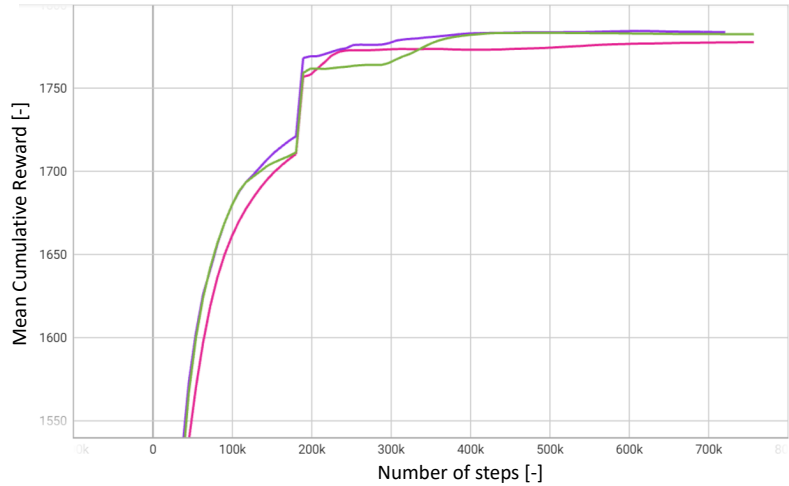


Fig. 6.2 Mean reward trend of different agents visualized through Tensorboard against the number of steps performed.

This plot depicts the trend of the cumulative mean reward for three different models over the number of steps run by the agent. More insights about how the hyperparameters of the agent impact the training will be given in section 6.4. By now, it is enough to say that being able to visualize this kind of value really helps in the examination of the model, especially when the model is unstable and a convergence is not reached.

Run	Value	Step	Time	Relative
● SAC_200250_0	1775	531,295	9/2/23, 3:55 AM	4.182 hr
● SAC_B_0	1784	531,295	8/30/23, 3:55 PM	4.751 hr
● SAC_DR3_300_0	1783	531,295	9/4/23, 3:28 AM	4.262 hr

Fig. 6.3 Table with additional information regarding the parameter plotted for several agents.

In addition, figure 6.3 shows the tab that appears when moving the pointer along the lines in the diagram. It provides some details about the run, in particular the name of the models, the value of the mean reward at that timestep, the time when the simulation was carried out, and the duration of the simulation up to the selected point.

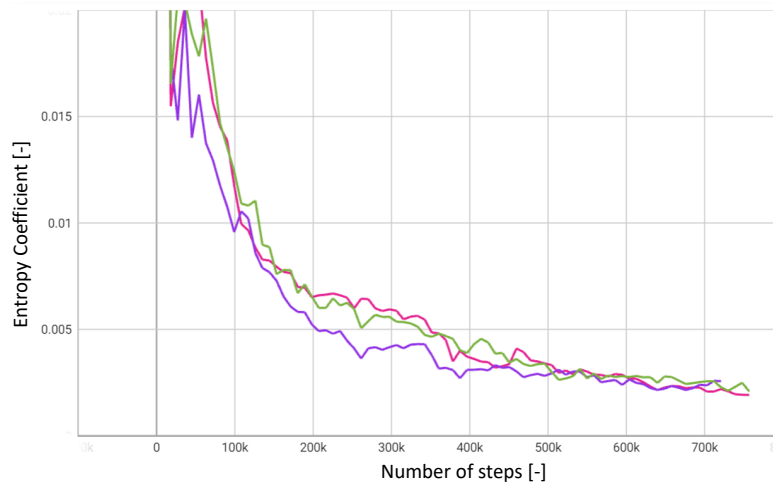


Fig. 6.4 Entropy coefficient adopted by the agent during training as a function of the steps performed.

As already explained, the length of the episode is constant because the vehicle always runs on the WLTC but the diagram showing the episode length is more useful than the reader might guess because it is an additional indicator that everything works properly. In fact, during training, due to some bugs in the code, the mean length started fluctuating, a phenomenon that is not compliant with the task.

The entropy coefficient graph reported in figure 6.4 is remarkable because it shows how the agent handles entropy during the training and, as trivial, the lower the value the higher the reward.

6.4 Neural Network manual tuning

To gain deeper insights into the training dynamics of the agent, manual parameter tuning was first assessed. Even if this approach is not feasible to achieve an actual optimization of the SAC agent's hyperparameters, it turns out to be useful for the

authors to test the sensitivity of the agent and to firsthand experience how some changes in the agent features affect the training, paving the way for more efficient approaches in future developments.

Before going through the several simulations run to understand the impact of some hyperparameters, a quick glance at the code is suggested.

```
1 policy_kwargs = dict(net_arch=dict(qf=[256, 256],
2 pi=[256, 256]))
3 LearningRate = 0.0003
4 BatchSize = 256
5 Tau = 0.005
6 Gamma = 0.99
7 TrainingFrequency = 1
8 TargetUpdateInterval = 1
9 EntropyCoefficient = 'auto'
10 model = SAC('MlpPolicy', env, verbose=1,
11 tensorboard_log=logdir, learning_rate=
12 LearningRate,
13 learning_starts=200, batch_size=
    BatchSize, tau=Tau, gamma=Gamma,
    train_freq=TrainingFrequency,
    gradient_steps=1,
    target_update_interval=
    TargetUpdateInterval, policy_kwargs=
    policy_kwargs,
    ent_coef=EntropyCoefficient)
```

Thanks to this simple notation allowed by SB3, the hyperparameter of the agent can be modified as desired. The values reported in the code are the default ones. In the following, a review of the most relevant tuning parameter is proposed with the aid of the previously mentioned visualization tools allowing for a better understanding.

Since the very first runs of the learning process, the model behaved very well with no relevant issues like huge divergence or very fluctuating behavior. For the sake of clarity, only the results of a properly working environment, in the largest sense of its meaning, are presented in this project work and all the interventions

needed to fix bugs in the code and modeling errors in the environment are left to the development phase of this thesis.

First, it is useful to define the baseline and the trend of the mean reward when the agent model has all its hyperparameters set to their default value. Keeping in mind that, as mentioned before, the maximum reward is 1800, the default setting performs quite well according to figure 6.5.

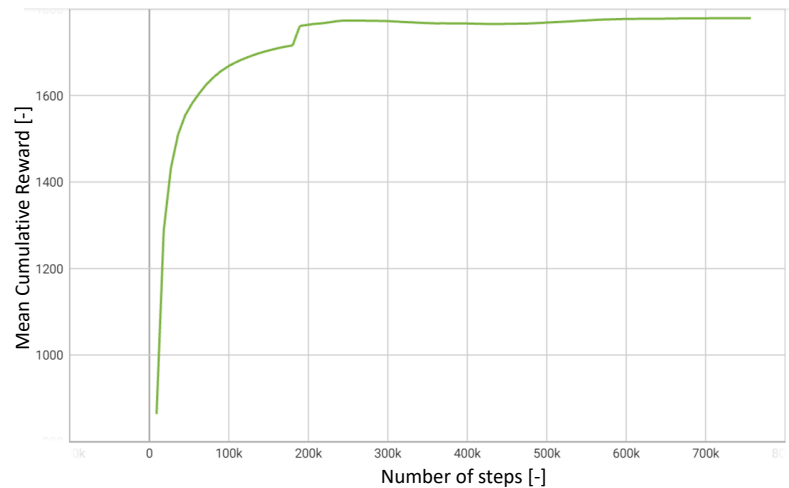


Fig. 6.5 Cumulative mean reward trend with default parameter in all its behavior plotted against the number of steps performed by the agent.

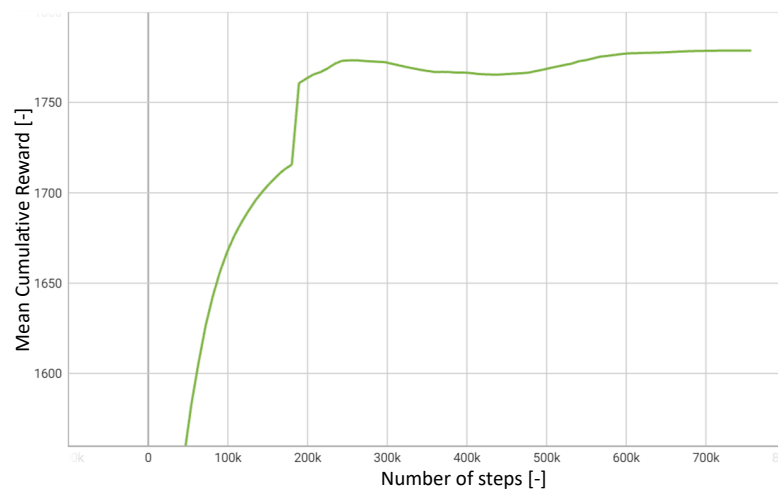


Fig. 6.6 Cumulative mean reward trend over the number of steps with default parameter in a zoomed view to catch the behavior where it will be compared with other models.

The most relevant aspect is that the reward converges to a value that is very close to 1800 indeed. The first part of the training is characterized by a sharp rise

which brings the reward to hits the value of 1600 in only 35 cycles. Right after this point, the reward seems to gradually start its convergence but a sudden peak is registered after about 100 cycles from the beginning that brings the reward around the saturation value. This steep jump can be explained by the agent that exploring is able to “escape” a local minimum and reach for the real global optimal policy. After only 105 cycles, the value plateaus in a value range included between 1765 and 1779. It is worth mentioning what the agent is doing during all the remaining cycles since it scores almost the same value. As discussed in section 3.6, the algorithm adopted follows the max-entropy formulation where the value to be maximized is not only the long-term expected cumulative reward but also the entropy of an action in a state. This implies that the mean value of the reward collected throughout the training contains the entropy term that is lowered, or even canceled, only close to the end of the training. In other words, even when the trend seems to be steady around the same figure and no additional runs are believed to be required, the agent is still exploring and learning. Only the values registered at the end of the training represent the real reward obtained by the agent. To conclude this overview, the trend followed by the reward in the default configuration will not experience substantial change due to some hyperparameter tuning which is coherent with the SAC property of being slightly affected by NN parameters modifications.

The first parameter to be modified is the learning rate, that is the rate at which the neural network parameters are updated during training. It determines how much the model’s weights are adjusted in response to the computed gradients by controlling the size of the step taken during the update. A too-high learning rate can lead to oscillations and divergence in training, while setting it too low may result in slow convergence or getting stuck in local minima. The learning rate is swept from 0,0003 to 0,001 and so a faster, but potentially more unstable, learning is expected.

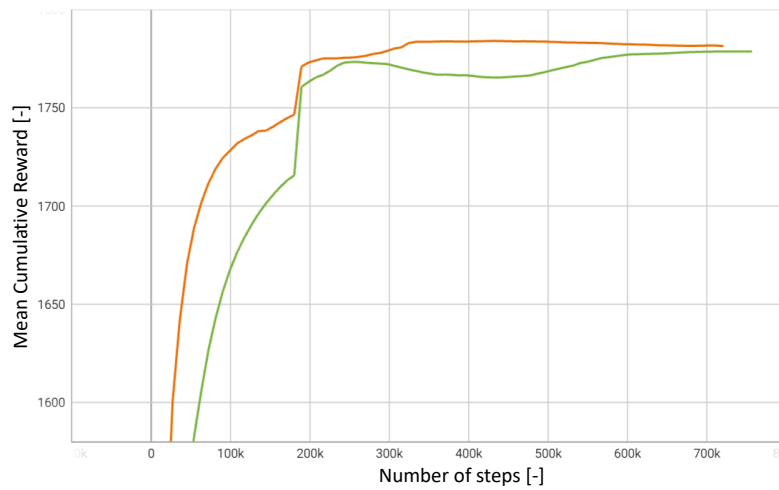


Fig. 6.7 Cumulative mean reward trends of an agent with default parameter, i.e., 0,0003, (green line) and another one with learning rate equal to 0,001 (orange line) plotted against the number of steps

The trend with the higher learning rate, the orange one, is much faster than the default one. The reward rapidly grows and after just 15 cycles it exceeds the value of 1600. Surprisingly, it turns out to be even more stable than the default setting and it saturates to a higher mean reward value around 1782.

The second sweep involves the coefficient γ , the well-known discount factor typical of continuous tasks. The role of this coefficient is crucial in controlling the balance between immediate and future rewards. It is a real number ranging from zero to 1:

- γ close to zero makes the agent short-sighted and mainly focuses the attention on maximizing the immediate rewards;
- γ close to one characterizes an agent valuing more the long-term performance of the algorithm.

In the charts reported in figure 6.8, gamma is decreased to 0.9 from the original 0.99 so it is given less relevance to future consequences that will probably translate into a lower final value and a slower convergence.

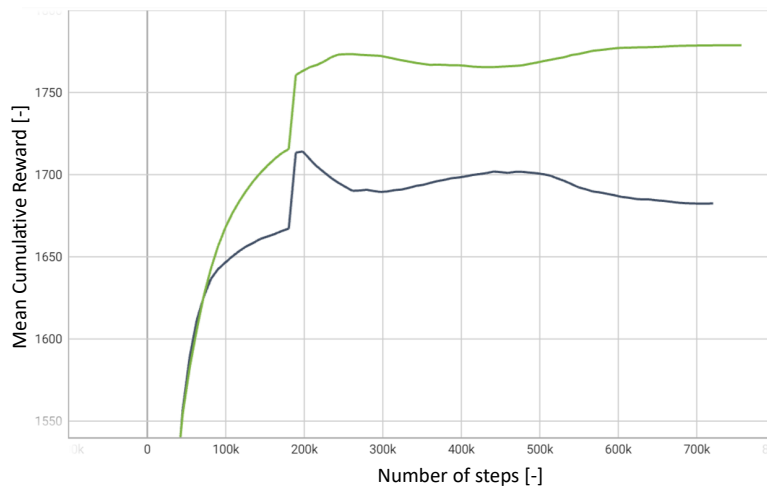


Fig. 6.8 Cumulative mean reward trends of an agent with default parameter, i.e., 0.99, (green line) and another one with gamma equal to 0.9 (blue line)

As expected, lowering gamma has brought very poor performance: not only the overall mean reward values scored are much lower with respect to the baseline, but also the trend depicts a very unstable learning that never reaches a steady behavior. Moreover, the value of 0.99 seems to be very well balanced because increasing it from its default value to 0.999 outputs worse results, with a very fluctuating value of the mean reward as it is shown in figure 6.9.

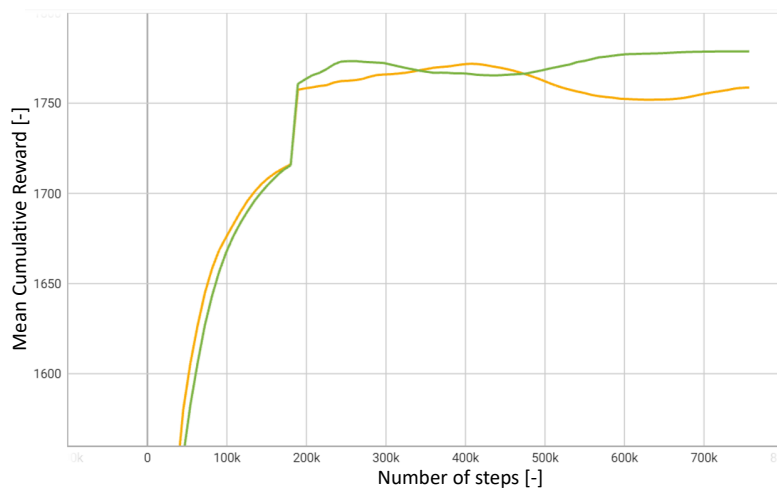


Fig. 6.9 Cumulative mean reward trends of an agent with default parameter, i.e., 0.99, (green line) and another one with gamma equal to 0.999 (yellow line)

Moving on, now it is time to double the batch size, from 256 to 512. Recalling some theory, the batch size is a hyperparameter that determines the number of experiences (i.e. state, action, reward and next state) collected from the environment and used in each training iteration. Its value is one of the most important trade-offs between computation efficiency and learning stability. More robust and consistent training is expected with a larger batch size because it averages over a larger set of data while a small batch size may introduce more noise in learning but it certainly allows to speed up convergence since it requires less memory.

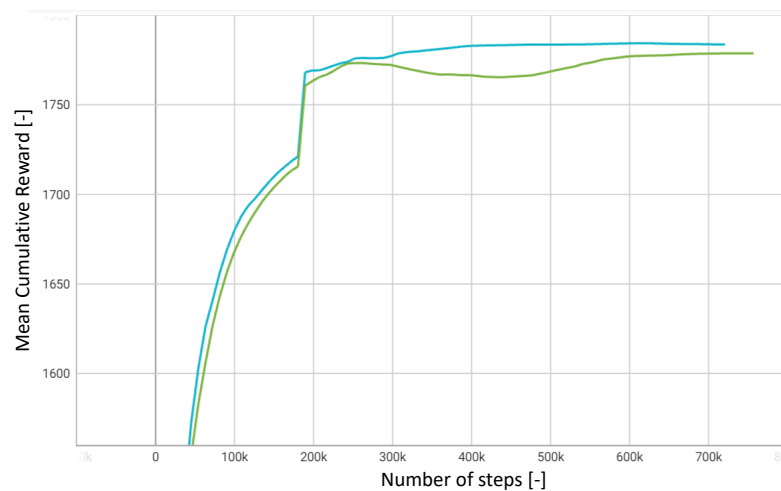
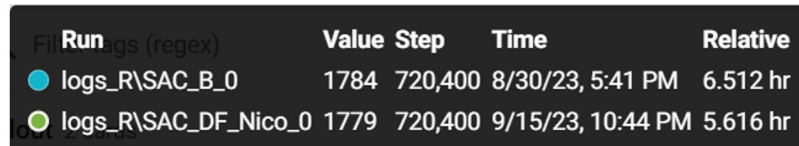


Fig. 6.10 Cumulative mean reward trends of an agent with default parameter, i.e., 256, (green line) and another one with a batch size of 512 (cyan line)

As forecasted, the trend is more stable and shows higher performance than the baseline as visible in figure 6.10. Below is also reported a log window with some additional data and two values that are worth noticing: the higher final value and the much greater simulation time implied by the larger batch size. A batch with double the size increases the training duration by approximately 16%.

A screenshot of a terminal window with a dark background and light text. It displays a table with four columns: 'Run', 'Value Step', 'Time', and 'Relative'. The first row has a cyan circle icon, and the second row has a green circle icon. The table data is as follows:

Run	Value Step	Time	Relative
logs_R\SAC_B_0	1784 720,400	8/30/23, 5:41 PM	6.512 hr
logs_R\SAC_DF_Nico_0	1779 720,400	9/15/23, 10:44 PM	5.616 hr

Fig. 6.11 Table with additional data regarding the comparison between an agent with default parameter (green circle) and another one with a batch size of 512 (cyan circle)

The next configuration evaluated sees two parameters changed at once: the τ coefficient and the target update interval. In SAC and other deep RL algorithms, target networks are used to stabilize training and improve convergence. These target networks are slowly updated over time to provide more stable target values for the Q-learning objective. τ is also sometimes referred to as the *soft update coefficient* because it smoothens the affection of the parameters of the main policy network (actor) or of the main Q-network (critic) to the update of the target network parameters, both for the target critic and actor. In the next run, the τ is raised from 0,005 to 1 which means that the parameters of the behavioral network are copied to the target network at each update. In addition, the target update interval is increased from 1 step to 1801 and so the target network is updated only at the end of each episode. In this way, the fluctuating behavior that would derive from setting τ equal to 1 is overcompensated with no other actions taken.

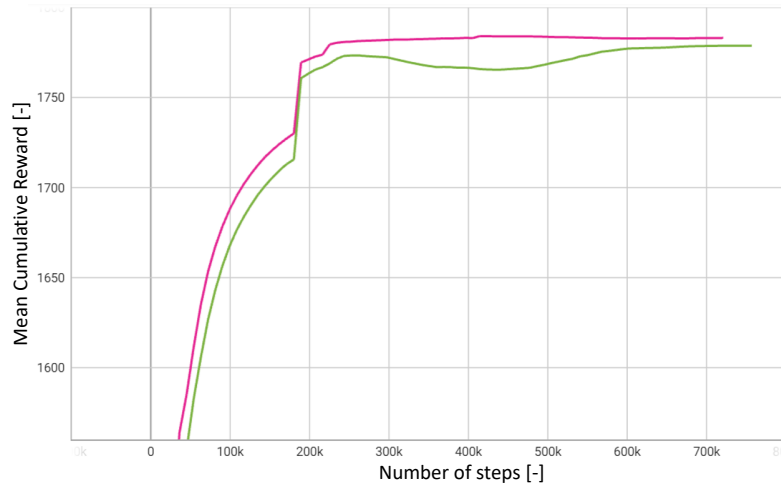


Fig. 6.12 Cumulative mean reward trends of an agent with default parameter, i.e., 0.005 for τ and 1 for the target update frequency, (green line) and another one with the τ equal to 1 and the target update equal to 1801 (purple line)

This configuration significantly improves the agent performance, converging more rapidly and to higher values of the mean reward showing that letting the behavior policy improve over a cycle and then copying its parameters to the target policy is a winning approach. Then, to slightly improve the computational efficiency with respect to the default setting, the target update interval is increased from 1 to 10 and no other parameters are modified. The results are nothing unexpected: a little faster training with the same result as it's possible to see in figure 6.13.

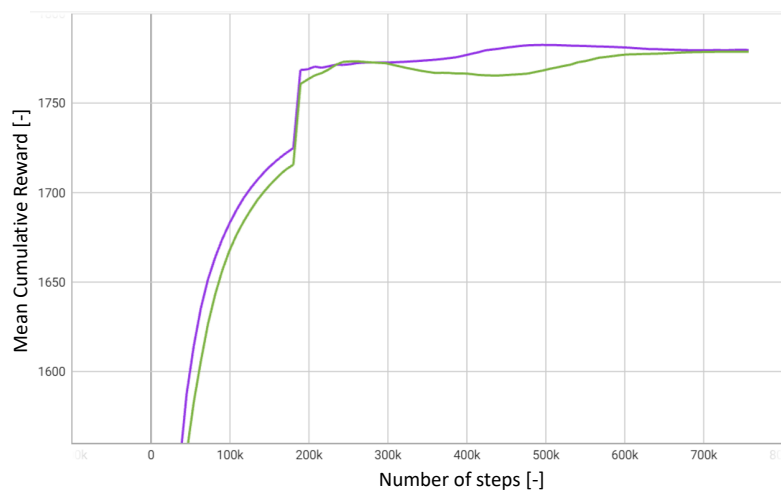


Fig. 6.13 Cumulative mean reward trends of an agent with default parameter, i.e., 1, (green line) and another one with the target update equal to 10 (violet line)

As last comparison is left the one between parallel and series training. In parallel training, multiple instances of the same episode are run simultaneously, each with its own agent. Even if these agents come from the same central system that handles all the data and updates all the agents in the same way, they interact with their respective environments independently and in parallel. After each timestep, all the data collected from the parallel training are used to update the target parameters of all the agents. In series training a single agent interacts with the environment sequentially. It completes an episode or batch of episodes before moving on to the next batch. This implies a faster convergence because the data is always exchanged between the same agent and environment but it also means dealing with longer simulation time of the process since all the episodes must be run in sequence.

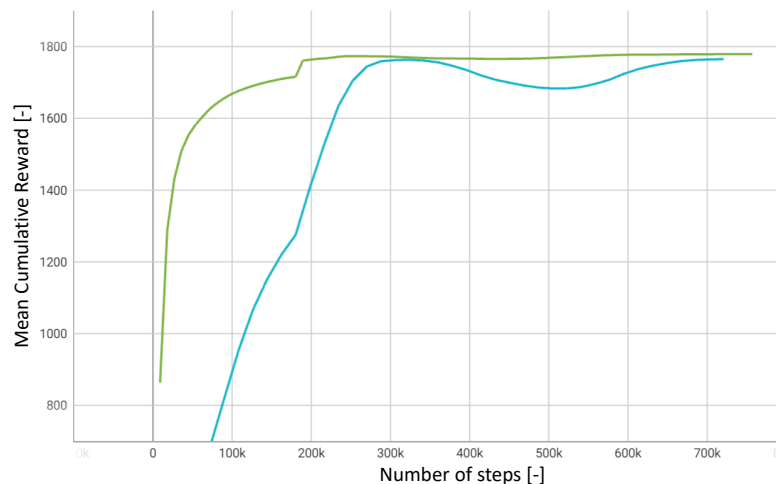


Fig. 6.14 Cumulative mean reward trends of two agents with default parameters but one trained in series (green line) while the other one in parallel (cyan line)

The blue line represents the parallel training and it turns out to be very unstable according to the fact that it averages the data collected from different instances of the same episode, but it eventually converges to the same mean reward value.

6.5 Reward

Before discussing how the reward should be set for the specific case treated in the framework of this thesis, a little recap could be useful to better address the reward definition task. The agent is asked to learn, but what does it mean for a machine

to learn? To answer try to think about what a policy is: a function made up of logic and tunable parameters. The goal of the agent is to set those parameters in a way that produces an optimal policy. In other words, it tries to map state to actions to maximize the long-term reward. It should be clear how relevant is the role of the reward in this context. Crafting a reward function is not as trivial as the reader might think because it should be able to make the algorithm understand when the policy is improving and, eventually, converging to the desired result avoiding any kind of shortcut. The reward is a function that produces a scalar number that represents the *goodness* of an agent being in a particular state and taking a particular action.

$$reward = function(state, action) \quad (6.1)$$

The great potential in RL when dealing with a reward function is that there are no restrictions on its creation. The agent can receive sparse rewards or rewards every timestep or even rewards coming only at the very end of an episode. Rewards can be the product of complex non-linear functions or just be computed using thousands of parameters. This great freedom in the definition comes with the huge shortcoming of getting stuck around the poor behavior of the agent. In fact, if the agent finds itself exploring tons of new actions and visiting thousands of new states without getting any reward, no learning is taking place. Hoping that the agent will casually take the right sequence of actions to get a reward in an environment with several degrees of freedom is worth of the best of the fairy tales. It should be striking how much difference makes shaping the reward function in one way rather than another.

On this premise, it is obvious that a well-developed know-how of the subject is required to define the reward. For this reason, the path highlighted by the previous work with DP carried out by the authors is followed as a guide in the definition of a reward aimed at optimizing the energy flows of an HEV. After some discussions and trials, two rewards are designed.

- Case 1

Considering that making an HEV work in CS mode means guaranteeing that the charge level in the battery stays in the neighborhood of the initial SOC, the first reward is thought to be a simple minimization of the fuel consumption with a penalty proportional to the deviation of the SOC from

the target value.

$$r_{case1} = k_1 - k_2 \dot{m}_f \Delta t - k_3 (SoC - SoC_{trg})^2 \quad (6.2)$$

where k_1 , k_2 and k_3 are constants which weigh the influence of the relative term, \dot{m}_f is the fuel rate, SoC and SoC_{trg} are the actual and the target SoC , respectively. This formulation exploits a so-called continuous penalization of the SoC . As it is possible to notice, the fuel consumption and SoC terms have a negative sign. This is easily explained recalling that an RL agent always wants to maximize the reward while the EMS's ultimate objective is to minimize the fuel consumption. As will be discussed in chapter 7.1, the constant coefficients k_1 , k_2 , k_3 are mutually changed to find the best-performing set of values starting from a default configuration coming from the work with DP. This function turns out to be very simple in training, it introduces a penalty to the SoC over the entire duration of the cycle, limiting the potentialities of a Hybrid configuration.

- Case 2

This case aims to give more freedom to the SoC to fluctuate between some boundaries and to tighten those limits while approaching the end of the cycle. Its definition looks like:

$$P_{SoC} = \begin{cases} (SoC - SoC_{trg})^2 & \text{if } SoC < SoC_l \\ 0 & \text{if } SoC_l \leq SoC \leq SoC_h \\ (SoC - SoC_{trg})^2 & \text{if } SoC > SoC_h \end{cases} \quad (6.3)$$

$$r_{case1} = k_1 - (k_2 \dot{m}_f + k_3 P_{SoC}) \Delta t \quad (6.4)$$

As shown in figure 6.15, for the first three-quarters of the cycle, the SoC is penalized proportionally to the square of the distance from the target value only if it overcomes one of the boundaries set at 0.55 and 0.45. Inside the window, the agent has complete authority in action selection. For the last quarter of the cycle, the boundaries converge to the target value to force the SoC to close the cycle near the desired value. This formulation comes from the DP boundary lines method described in [47] and it allows the agent to freely choose the action that maximizes the reward as long as the boundary lines are not exceeded.

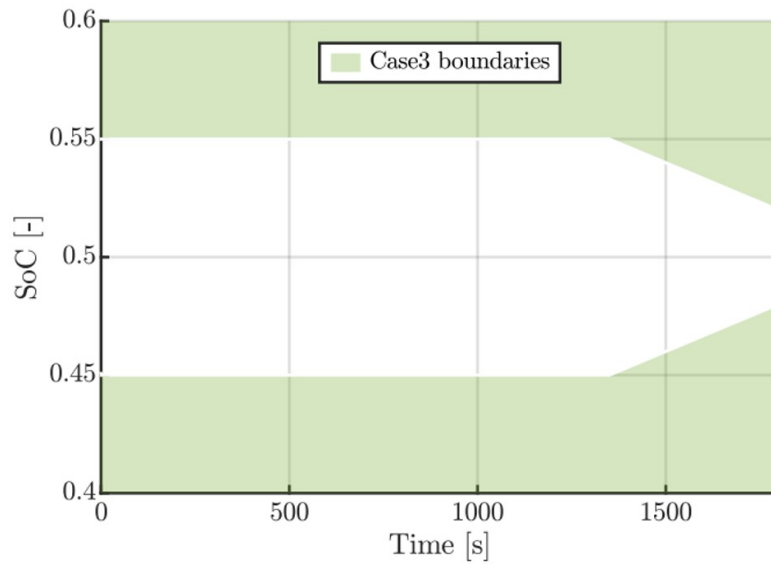


Fig. 6.15 SoC boundary lines for Case2 reward.

Chapter 7

Results

The proposed Reinforcement Learning-based Energy Management System was mainly tested on the same cycle used for training: WLTC. Even if the trained agent was then tested over two additional cycles to assess how bad or good was the generalization capability of the algorithm, in the framework of this thesis, the majority of the final analysis focused on the comparison between two reward formulations explained in section 6.5. This project presented a new methodology so plenty of different tests and improvements are still on the way. The author preferred to focus on a careful comparison of the above-mentioned rewards and their performance with respect to the benchmark set by Dynamic Programming. DP allows to obtain the optimal solution, that is why it represents a target to aim to but not reachable. In contrast, it needs apriori knowledge of the driving cycle that, in addition to the unfeasible computational effort, makes it not suitable for online application. On the other hand, the RL agent achieves a sub-optimal performance but with the possibility of being applied on board. In the following, a comparison between the two rewards in terms of SoC and fuel consumption is proposed and, once the best solution is identified, it is the moment for the performance assessment with respect to global optimum.

7.1 Comparative analysis of the rewards

The first reward was defined in section 6.5 but nothing was mentioned about the experimental coefficients k_1 , k_2 and k_3 . According to previous works of the authors with the same model, a genetic algorithm was used to better assign a

value to those parameters. The best combination proposed was adopted as default configuration and it featured the following values:

$$\begin{cases} k_1 = 0 \\ k_2 = 200 \\ k_3 = 400 \end{cases} \quad (7.1)$$

Looking at these values, it is striking how huge relevance is given to the SoC deviation. The goal of the agent is to score the highest reward possible which means minimizing the fuel consumption and the SoC deviation since both terms have a negative sign in the reward formulation. This rule is applied during the whole cycle which implies that the agent should be very careful in using the energy stored in the battery if it does not want to score very poorly. With these values, a low exploitation of the electric energy, and so little fluctuations in the SoC, is expected in the upcoming plots. Figure 7.1 shows the results of the reward 1 with default coefficients against the optimal control strategy identified by DP. As a reference, the final values obtained by DP are shown in table 7.1.

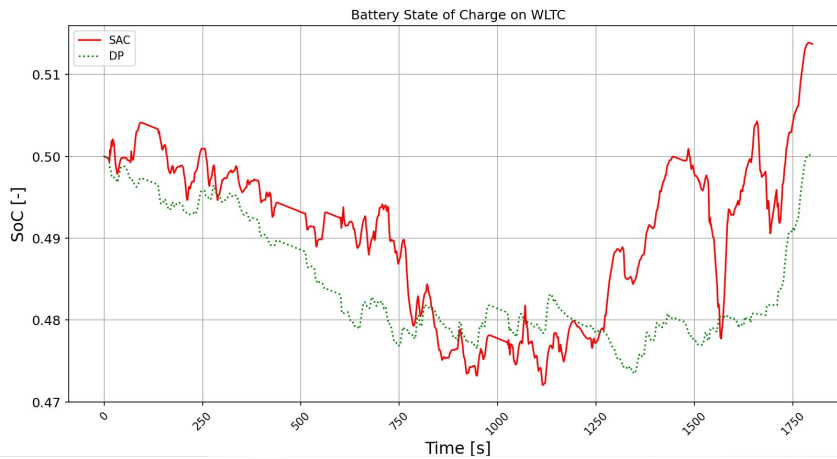


Fig. 7.1 SoC trend of the SAC agent trained with reward 1 with default parameters: $k_1 = 0$, $k_2 = 200$ and $k_3 = 400$.

Table 7.1 Dynamic Programming optimal control strategy results over WLTC.

	SoC	FC	FC
	[–]	[kg]	[l/100km]
<i>DP</i>	0.5	0.9658	4.9349

The final value of the SoC reached by the model guided by the SAC control strategy is equal to 0.5137 while the fuel consumption, shown in figure 7.2, is of 1080,5 g. As expected, the deviation from the target value of SoC (i.e. 0.5) is very little and this is due to the reward formulation that continuously penalizes the agent when charging or discharging the battery. For almost three quarters of the cycle, the agent follows the trend resulted from Dynamic Programming. In the last quarter, probably due to the distance from the reference value of SoC, the SAC agents decided to quickly recharge the battery. It should be remembered that the last section of the WLTC is the high-speed high-power demand part of the cycle, representing a possible highway scenario. To catch the relationship between the agent's behavior and the required speed, in figure 7.3 the two plots are aligned.

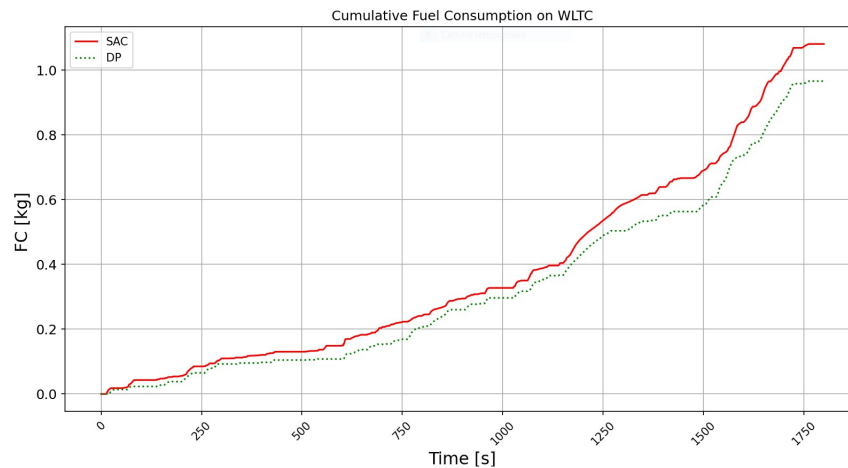


Fig. 7.2 FC trend of the SAC agent trained with reward 1 with default parameters: $k_1 = 0$, $k_2 = 200$ and $k_3 = 400$.

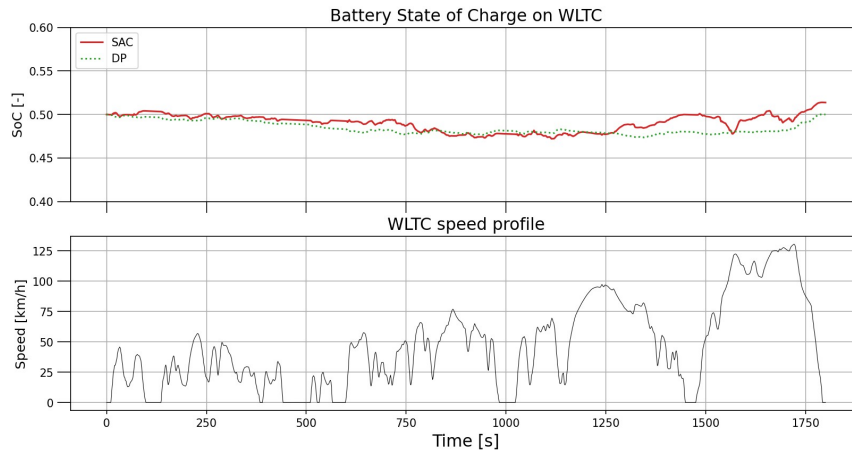


Fig. 7.3 SoC trend of the SAC agent trained with reward 1 with default parameters: $k_1 = 0$, $k_2 = 200$ and $k_3 = 400$; plotted against the speed profile of WLTC.

A peculiar characteristic of the SoC trend achieved by the SAC that will appear in all the different reward configurations presented in the following is the sudden rise of the level of energy in the battery at the very end of the cycle. A careful look at figure 7.1 points out that the same behavior also characterizes the DP result, even if in that case the spike is handled in a way that makes the vehicle to reach the target SoC at the end of the cycle. This sharp increase can be easily explained by looking at the speed profile: the cycle ends with a dip of speed and being the vehicle under test a HEV, it can exploit regenerative braking to store electric energy in the battery. While DP is aware of all the task's aspects from the beginning of the mission and can so plan what action to take according to the speed profile, the SAC agent does not have an idea of the upcoming road request and so it is forced to recharge during braking even if the SoC level is already about the target value. The reader might wonder why regenerative braking cannot be avoided if not necessary and the answer is that, for sake of simplicity, that mode was imposed every time the vehicle is required to slow down.

In the attempt of reducing the weight given to the SoC deviation in the reward formulation, the coefficient k_3 is lowered from 400 to 200 becoming equal to k_2 . The agent is tested again on WLTC with the new reward configuration and the output is reported in figure 7.4.

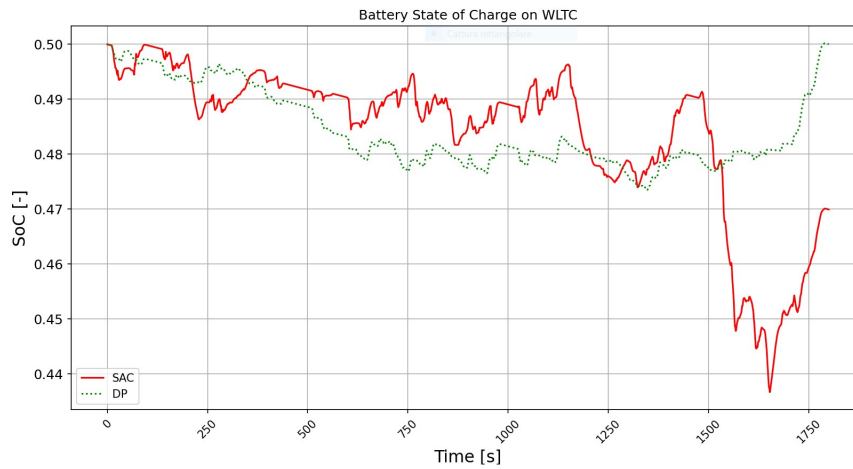


Fig. 7.4 SoC trend of the SAC agent trained with reward 1 with modified parameters: $k_1 = 0$, $k_2 = 200$ and $k_3 = 200$.

The SAC agent makes the vehicle complete the cycle with a SoC of 0.4698 that is quite below the target value of 0.5. This result is not surprising since the penalty given to the deviation from the SoC has been reduced: the agent is more likely to exploit the EM to propel the vehicle in order to minimize the fuel consumption because it is not receiving a reward as bad as it was with default parameters. In accordance with a higher use of the electrical energy stored in the battery, the fuel consumed during the cycle is 979 g, much closer to the DP value as shown in figure 7.5.

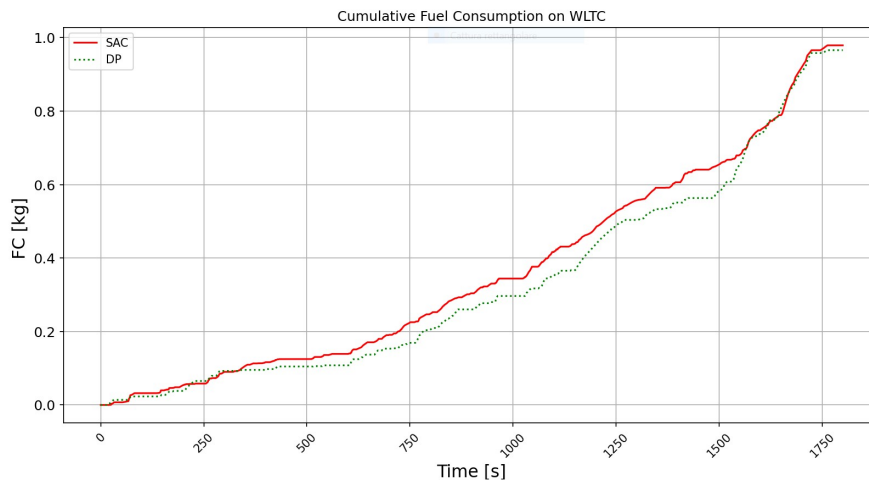


Fig. 7.5 FC trend of the SAC agent trained with reward 1 with modified parameters: $k_1 = 0$, $k_2 = 200$ and $k_3 = 200$.

Based on the result obtained, the value of coefficient k_3 is increased to 300. The goal of the authors in this framework is to minimize as much as possible the fuel consumption while guaranteeing a charge sustainability over the cycle that possibly terminates slightly below the target to take advantage of the little tolerance permitted by the legislation.

The trend of the SoC obtained in figure 7.6 is not trivial to be analyzed and understood. The weight given to the electrical side of the powertrain is higher than before and, as a matter of fact, the final SoC is higher than the target value. Unexpectedly, this model seems to be even more conservative than the default reward configuration where the weight assigned to the battery usage was even heavier. Not only this last test concludes with a higher level of the charge, but overall, it also exploits less the hybrid nature of the powertrain keeping the SoC between 0.49 and 0.50 for more than half of the mission. The fuel consumption almost resembles the result of the default case with a value of 1078,7 g.

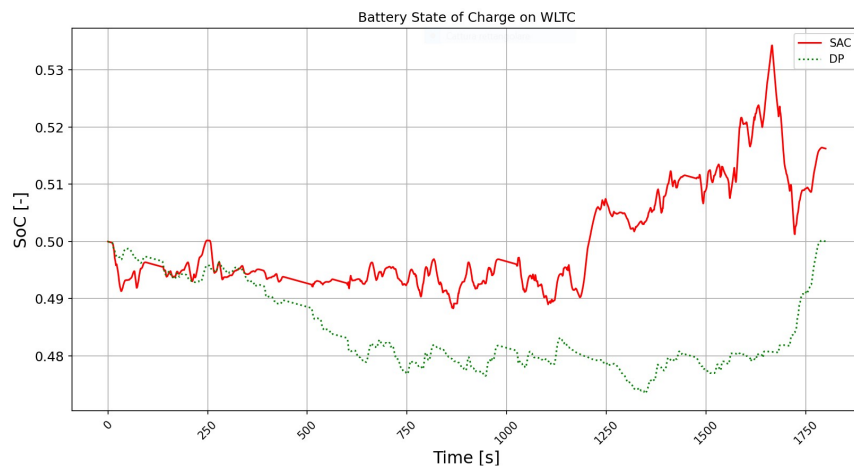


Fig. 7.6 SoC trend of the SAC agent trained with reward 1 with modified parameters: $k_1 = 0$, $k_2 = 200$ and $k_3 = 300$.

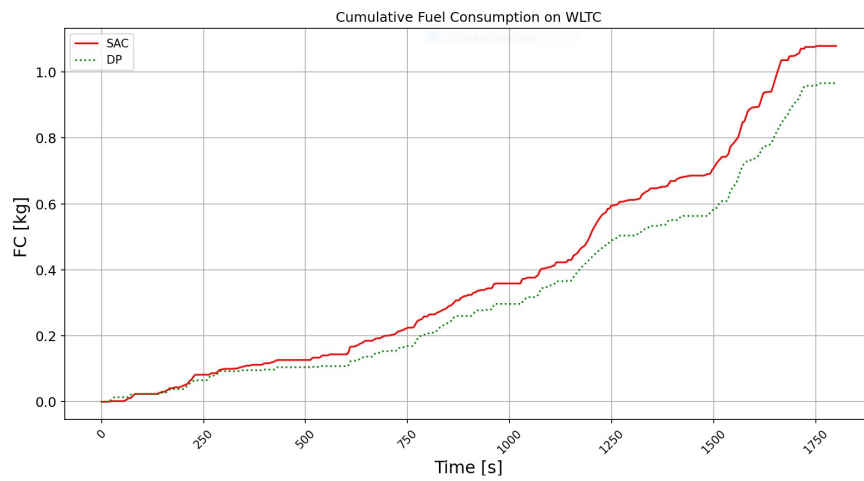


Fig. 7.7 FC trend of the SAC agent trained with reward 1 with modified parameters: $k_1 = 0$, $k_2 = 200$ and $k_3 = 300$.

Trying to reach perfect charge sustainability, the coefficient k_3 is lowered again, this time it is set equal to 250. The behavior is forecasted to lay in between the ones obtained for k_3 equal to 200 and 300. The result is depicted in figure 7.8.

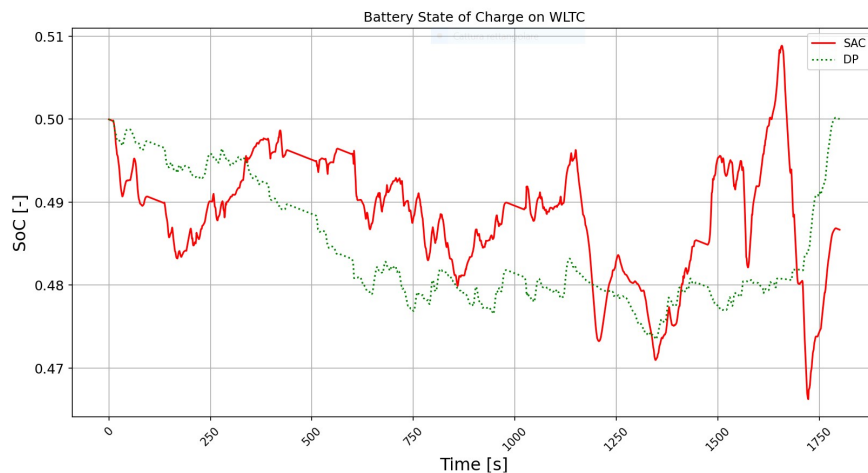


Fig. 7.8 SoC trend of the SAC agent trained with reward 1 with modified parameters: $k_1 = 0$, $k_2 = 200$ and $k_3 = 250$.

The vehicle completes the mission profile with a SoC of 0.4866, showing that the agent is very sensitive to the weight of the SoC deviation. Slightly better than the attempt with k_3 equal to 200, the final result is not satisfactory yet, that's why the k_3 coefficient is set to 275, trying to get closer to the behavior of the reward with k_3 equal to 300 but avoiding overcoming the target value. Figure 7.9 presents the result.

With a final value of 0.4959, it seems that the right value for k_3 has been found. Aside from a weird depletion at $t = 1250s$, the trend is quite conservative and it keeps the same features of the previous diagrams. On the other hand, fuel consumption is not as low as it could be based on battery exploitation. Figure 7.10 shows the cumulative trend that sums up to 1068,5 grams.

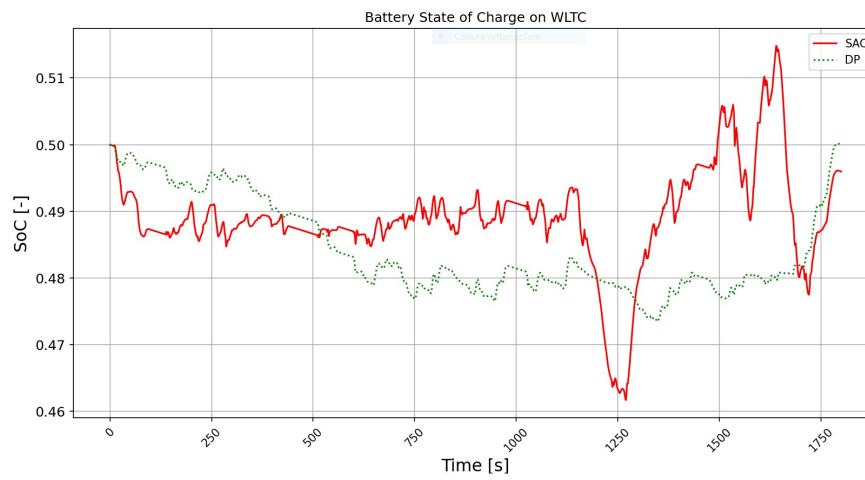


Fig. 7.9 SoC trend of the SAC agent trained with reward 1 with modified parameters: $k_1 = 0$, $k_2 = 200$ and $k_3 = 275$.

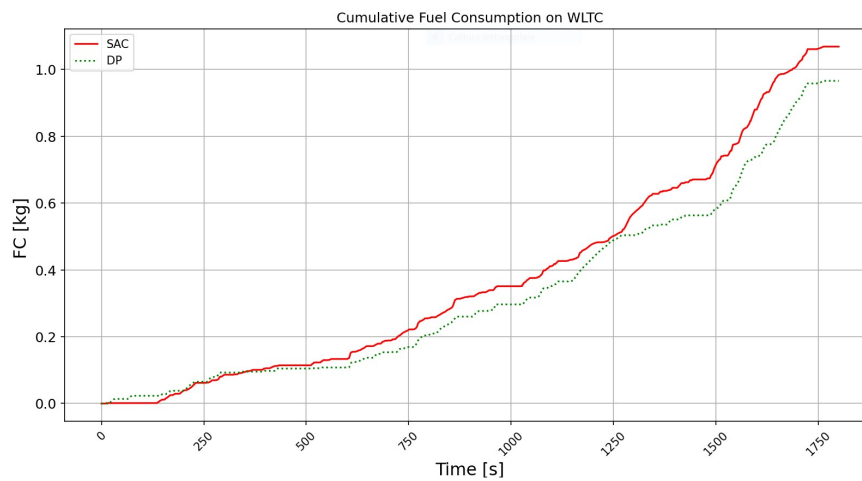


Fig. 7.10 FC trend of the SAC agent trained with reward 1 with modified parameters: $k_1 = 0$, $k_2 = 200$ and $k_3 = 275$.

Even if not a real best solution has been identified among the various configurations of reward tested, the results were all quite satisfactory with no signs of

divergence or unstable behavior. Anyway, the weakness of this reward stands in its continuous formulation which prevents the vehicle from really exploiting the additional degree of freedom introduced by the EM and the battery pack. With the objective of leaving the agent more room to act, reward 2 is applied. As explained in section 6.5, the agent is penalized proportionally to the SoC distance from the target value only when outside some pre-set boundaries. In between them, the agent is free to charge and discharge the battery according to its preference. The boundaries are defined by two straight lines positioned at SoC levels of 0.55 and 0.45 for three quarters (i.e. up to $t = 1350s$) of the cycle. During the last quarter, the two lines start to converge narrowing the window available to the agent for fluctuations. At the end of the cycle, the gap between the two thresholds is 0.02, symmetrically defined around the target value. The aim of the authors is to promote the agent to use the electrical energy as long as the SoC remains in the *penalty free* area. The first test is performed with the same default parameters used in reward 1.

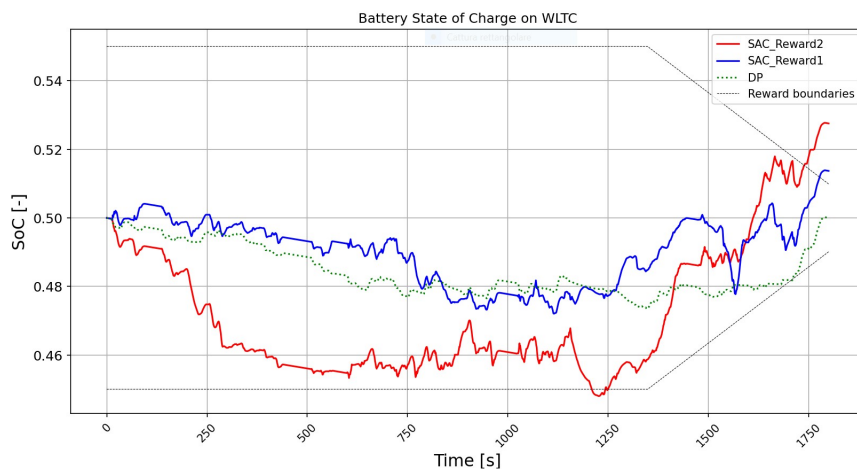


Fig. 7.11 SoC trend of the SAC agent trained with reward 2 with default parameters: $k_1 = 0$, $k_2 = 200$ and $k_3 = 400$; compared with the SoC trend of reward 1 with equal parameters.

In accordance with the SoC trend evaluated with reward 1 with default parameters, the agent behaves in a conservative way, making the vehicle complete the cycle with a charge level in the battery higher than the target. On the other hand, reward 2 allows the agent to use more of the electrical potential of the powertrain. As it appears clearly in figure 7.11, the agent significantly discharges the battery

in the first 250 seconds and then keeps the SoC almost constant around 0.46 to not face the poor reward out of the boundaries. In the last quarter of the cycle, it then modestly increases again the charge level overcoming the threshold up to the final value of 0.5275.

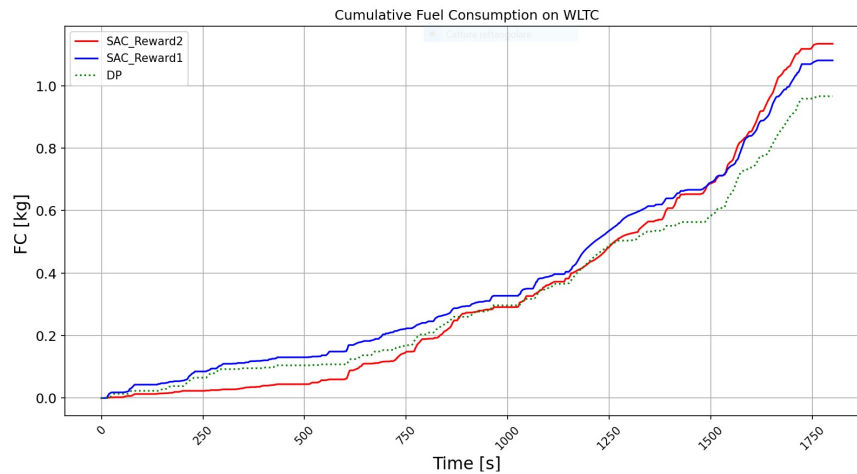


Fig. 7.12 FC trend of the SAC agent trained with reward 2 with default parameters: $k_1 = 0$, $k_2 = 200$ and $k_3 = 400$; compared with the FC trend of reward 1 with equal parameters.

Especially due to the last quarter of the mission, the cumulative fuel consumption experienced by reward 2 is even higher than the one of reward 1, with a final value of 1133,7 grams.

The same sweep of the k_3 coefficient from a value of 400 to the one of 200 performed with reward 1 is proposed again with reward 2. Figure 7.13 shows the output. Apart from a weird fall of the SoC of reward 2 close to the end of the cycle, the trend looks good because the agent can exploit more the battery but still reaches a value close to the target at the end of the driving cycle, precisely at 0.5010. With a fuel consumption of just 1054,4 g, and so a difference of slightly more than 9% with respect to the result obtained by DP, this value of k_3 seems to be much more suitable for reward 2 than it was for reward 1, where the agent carelessly discharged the battery.

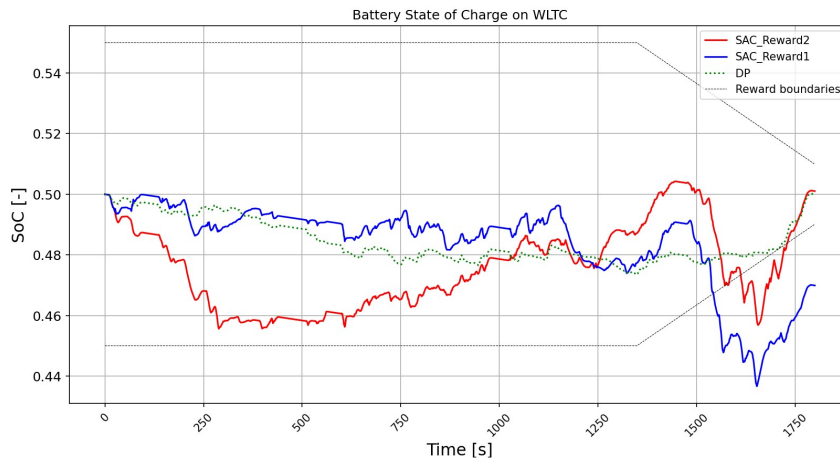


Fig. 7.13 SoC trend of the SAC agent trained with reward 2 with modified parameters: $k_1 = 0$, $k_2 = 200$ and $k_3 = 200$; compared with the SoC trend of reward 1 with equal parameters.

In order to proceed with the comparison between the two rewards, the sweeps already carried out for reward 1 are now proposed also for reward 2. The next k_3 value evaluated is 300. This reward configuration for reward 1 was quite conservative, closer to the default one. The same is expected for reward 2.

Unexpectedly, the trend obtained with reward 2 is very close to the one obtained by the same reward with k_3 coefficient equal to 200, as shown in figure 7.15. This time the agent is a little more conservative during the first half of the cycle, in fact it never makes the SoC overcome 0.5 or fall under 0.46. Only at the end of the cycle, the lower boundary is trespassed. This configuration is maybe one of the best Charge Sustaining example seen up to now, closing the cycle with a SoC value of 0.4996 that implies a fuel consumption of 1049,5 g, one of the lowest ever neglecting the ones obtained with a SoC too much below the target.

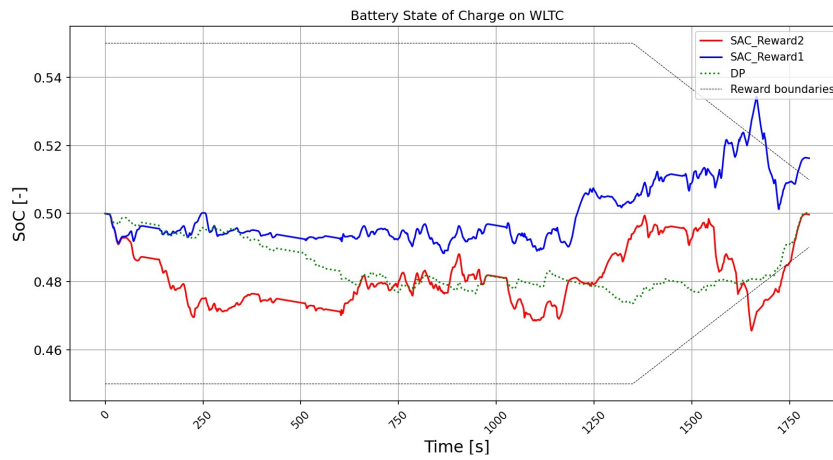


Fig. 7.14 SoC trend of the SAC agent trained with reward 2 with modified parameters: $k_1 = 0$, $k_2 = 200$ and $k_3 = 300$; compared with the SoC trend of reward 1 with equal parameters.

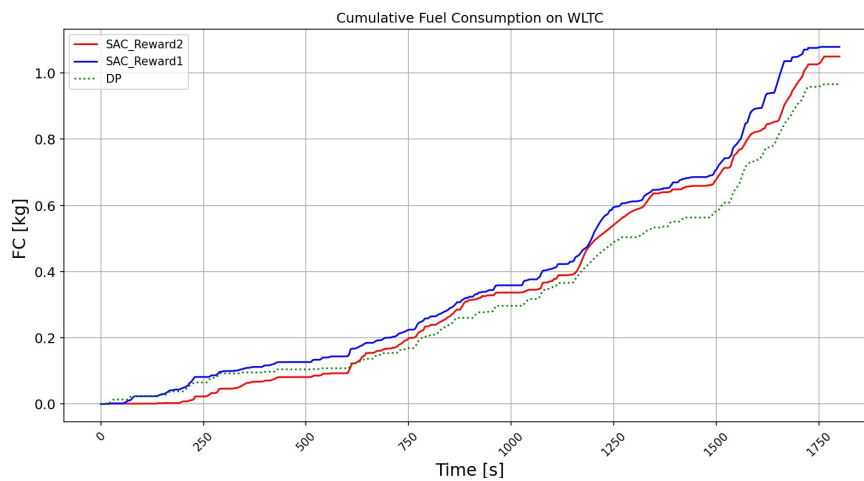


Fig. 7.15 FC trend of the SAC agent trained with reward 2 with modified parameters: $k_1 = 0$, $k_2 = 200$ and $k_3 = 300$; compared with the FC trend of reward 1 with equal parameters.

For the reward 2 case, the range 200-300 for the coefficient k_3 value seems to be much less sensitive than it was for the case of reward 1. Anyway, being very close to the possible best reward 2 set up, the k_3 values of 250 and 275 are investigated too.

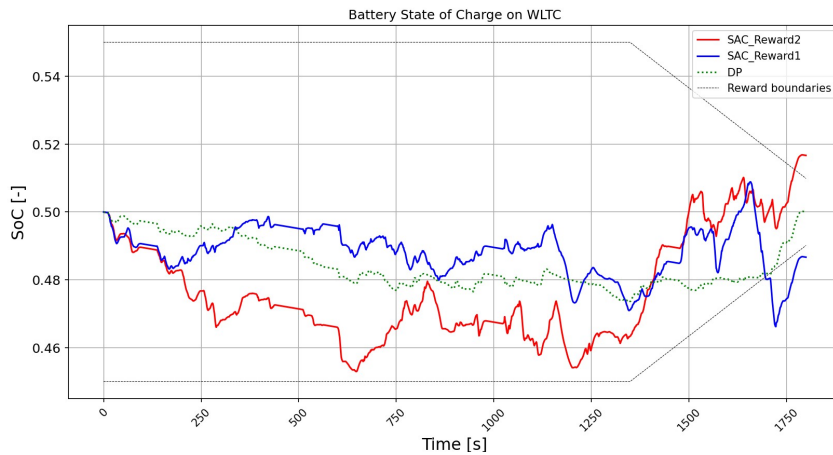


Fig. 7.16 SoC trend of the SAC agent trained with reward 2 with modified parameters: $k_1 = 0$, $k_2 = 200$ and $k_3 = 250$; compared with the SoC trend of reward 1 with equal parameters.

The result is far from being coherent with the previous simulation. With the value of 250 for the parameter k_3 , the agent behaves in a much less comprehensible way, showing this way to be less consistent than it was with reward 1. The SoC fluctuates much more than in previous runs and it finishes the cycle out of the boundaries.

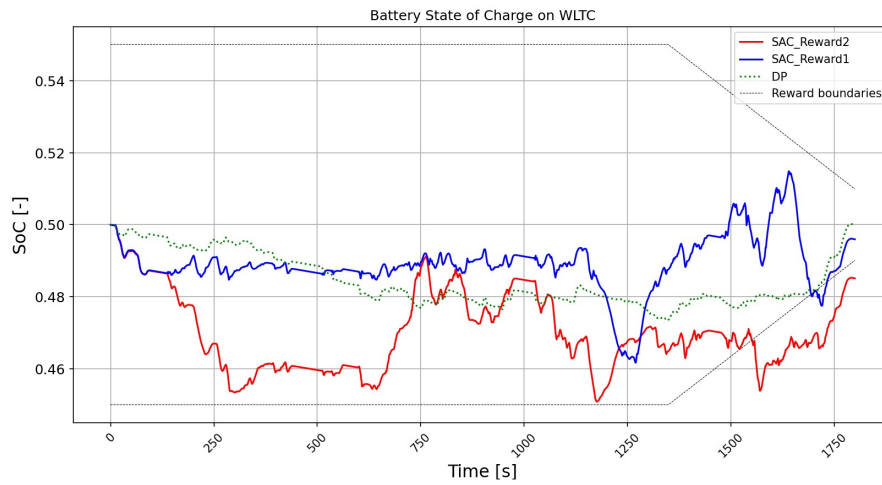


Fig. 7.17 SoC trend of the SAC agent trained with reward 2 with modified parameters: $k_1 = 0$, $k_2 = 200$ and $k_3 = 275$; compared with the SoC trend of reward 1 with equal parameters.

Figure 7.17 confirms that the agent trained with reward 2 is more unstable than predicted. With a slight change in the value of k_3 , from 250 to 275, the SoC trend goes from a fluctuating behavior ending far over the target, to a trend even more variable that ends way under the target. These last two tests put some doubt on the goodness of reward 2 but still, it was able to train the agent well in some conditions and so it is considered in the selection process of the best-performing agent among the ones tested.

So far, two kinds of reward with 5 configurations each have been proposed and evaluated in terms of fuel consumption minimization in a charge sustaining condition. To assess the potential of the SAC algorithm in this first application and compare its effectiveness with the result of Dynamic Programming, the authors looked for the reward configuration which trained the best agent. Figure 7.18 is meant to simplify the selection procedure allowing to compare the multiple cases with just a glance.

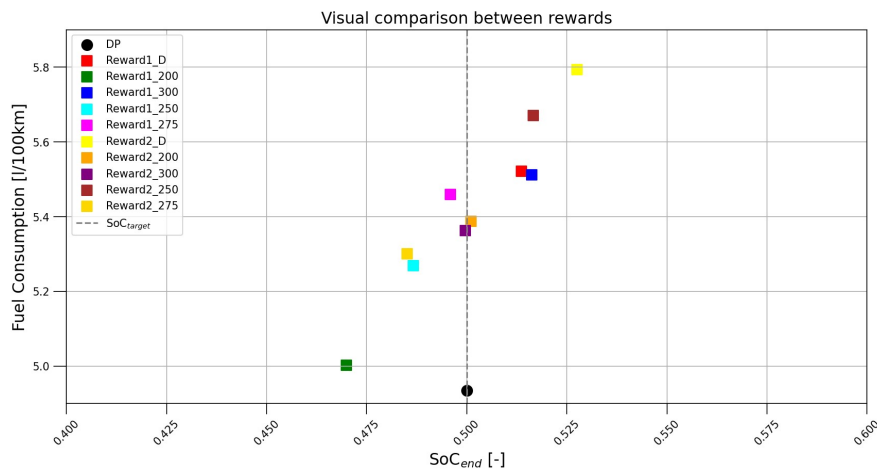


Fig. 7.18 Comparison between SAC agent with 2 different rewards with 5 configurations each and DP optimization: trade-off between fuel consumption and final SoC.

The diagram is thought to plot the performance of the agent in terms of trade-off between a certain reward final SoC and fuel consumption. The reader might remember that the SoC target, highlighted in the graph with a gray dashed line, is 0.5 which implies that every test ending with a value lower than this target, aside from some tolerance, cannot be considered because breaking the charge sustaining requirement. The agent trained with reward 2 with k_3 coefficient equal to 300 is selected as the most desirable thanks to its very low fuel consumption (i.e. 1049,5 grams) and a SoC of 0.4996, that can be approximated to 0.5 with very low numerical deviation.

7.2 Agent performance assessment

The agent's performance assessment will be carried out firstly on the same cycle used for training, i.e., WLTC, and then on two additional cycles in order to evaluate its generalization capabilities.

7.2.1 Agent testing on WLTC

Once the best agent model has been selected, a closer comparison with the benchmark set by DP is useful to evaluate the agent behavior. Before going through that, a little step back must be taken. The agent, with all default hyperparameters, has been trained with the case 2 reward (and $k_3=300$) which gives complete freedom to the agent to use the battery if the SoC remains inside the boundaries. Out of those thresholds, a penalty for the SoC deviation is applied. This setup involves greater use of the battery aimed to rely as little as possible on the ICE in the minimization of the fuel consumption that represents the term of continuous penalty to the agent according to reward 2 formulation explained in section 6.5. As previously explained in section 3.5 the agent starts the interaction with the environment at the beginning of the task with no a priori knowledge about it and so it explores to learn about the world. This learning phase, analyzed in section 6.3 thanks to the plot of the mean cumulative reward value over the number of steps performed, can be observed also through the vehicle's quantities and not only with the algorithm parameters. The greatest example of this is represented by the SoC trend over training shown in figure 7.19.

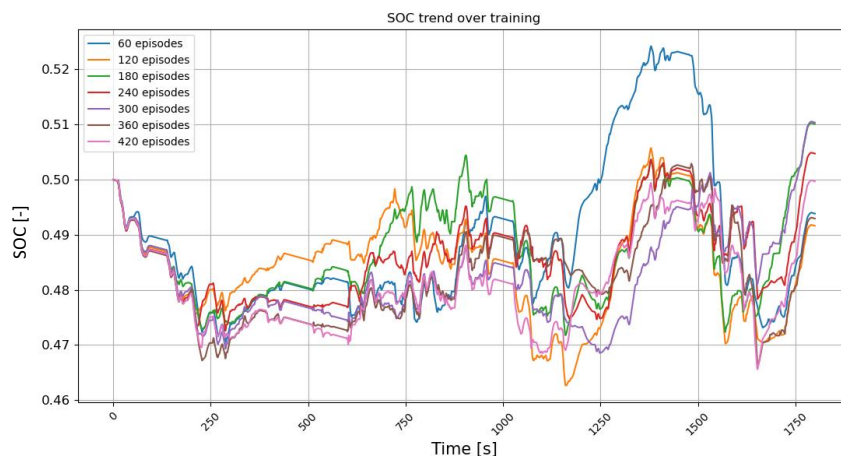


Fig. 7.19 SoC behavior during SAC agent training over the WLTC with reward 2 with modified parameters: $k_1 = 0$, $k_2 = 200$ and $k_3 = 300$.

In figure 7.19 the charge level of the battery along the WLTC is displayed every 60 episodes of training. To better catch the meaning of this plot, in figure 7.20 is reported the trend of the value of the cumulative mean reward for the agent under investigation in this paragraph.

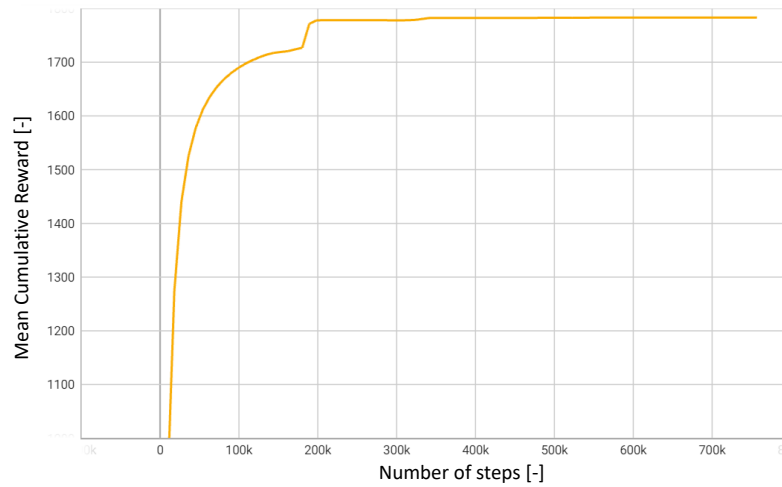


Fig. 7.20 Trend of the mean value of the selected best performing reward in all its behavior.

In figure 7.20 the reward value is plotted against the number of steps instead of the number of episodes but it is just enough to remember that 1800 steps constitute one cycle and so one episode. After the first 60 episodes (i.e. 108000 steps), the agent is still in its explorative part of the learning but the cumulative mean reward value is already approaching 1700. That is why, looking at figure 7.19, the SoC trend after 60 episodes is the only one that seems to be fluctuating. Of course, a closer look at figure 7.19 reveals that the agent continues to explore because the trends are not overlapped but, as figure 7.20 confirms, from 200000 steps on (equal to say 120 episodes) the reward is going to convergence and it is gradually reducing the relevance of the entropy term as explained in section 3.6.

It is now time to evaluate the performance achieved with this SAC algorithm. In figure 7.21 and figure 7.22 are reported the trends of the SoC and fuel consumption with respect to the time.

It is striking that the difference between the SAC agent and the DP behavior stands in the higher exploitation of the electrical capabilities of the vehicle from the first of them. Being free to act inside the boundaries, the agent quickly discharges the battery at the beginning of the cycle and then, probably aware of the penalty received out of the limits, keeps the charge level quite steady around the value of 0.48. The fluctuations at the end of the cycle characterized all the simulations and can be seen as a peculiar feature of the agent rather than as an unstable behavior.

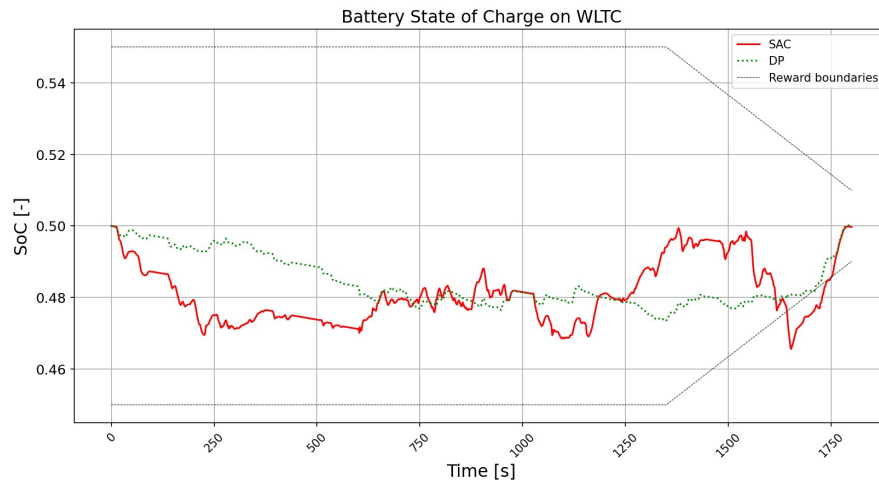


Fig. 7.21 SoC trend performed by the SAC agent trained with the selected reward configuration.

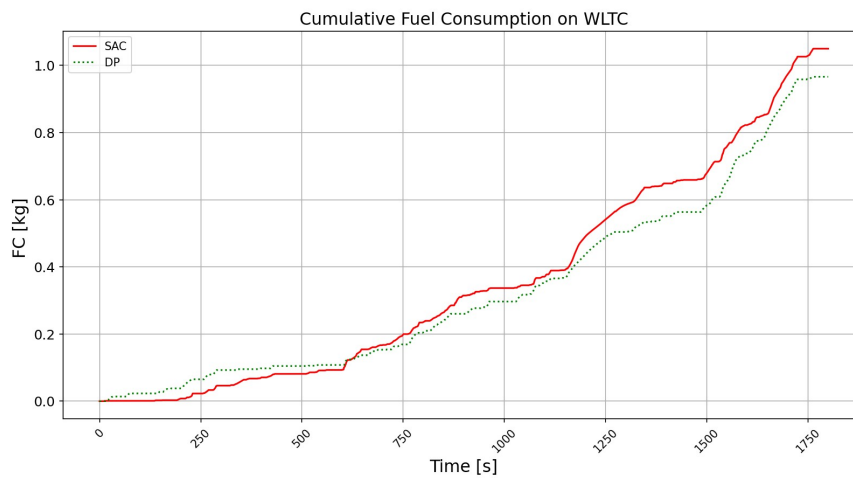


Fig. 7.22 FC trend performed by the SAC agent trained with the selected reward configuration.

The resulting SoC trend is reflected on the fuel consumption plot where the SAC agent relies much more on the EM to propel the vehicle in the first part of

the cycle, outperforming the DP values. The trend is reversed around $t = 600s$, where the will of the RL agent to keep the SoC around the same value, forces it to use more of the ICE increasing the fuel consumption that will never fall again under the DP value. In this configuration, the proposed algorithm is able to conclude the WLTC in a perfectly charge-sustaining condition with an increase in the fuel consumption of just 8.6% with respect to the DP result.

Table 7.2 Results comparison between Dynamic Programming optimal control strategy and SAC control strategy on WLTC.

	SoC	FC	FC
	[–]	[kg]	[l/100km]
<i>DP</i>	0.5	0.9658	4.9349
<i>SAC</i>	0.4996	1.0495	5.3623

7.2.2 Generalization over different mission profiles

One of the most important aspects of training a RL agent, or a NN more in general, is achieving good performance not only on the task used for training itself but also on tasks the agent has never experienced before. A common problem in ML is the so-called *overfitting* phenomenon that occurs when a model becomes too specialized in capturing noise and specific patterns in the training data, rather than learning the underlying patterns. In other words, it learns the training data too well to the point that it starts to perform poorly on new, never-seen tasks. Finding the right balance between fitting the training data and generalizing to new data was not among the objectives of this thesis since it represents one of the biggest challenges when dealing with AI algorithms. Nonetheless, a brief glance at the performance the selected agent is able to obtain on new mission profiles is proposed in this final paragraph. In particular, two additional tests are carried out on two standardized driving cycles coming from American legislation: Federal Test Procedure 75 (FTP75) and Highway Fuel Economy Test (HFET)[48]. FTP75 is one of the main standardized test cycles developed by the Environmental Protection Agency (EPA) for regulatory purposes, designed to simulate urban driving conditions. HFET is another standardized driving cycle used for evaluating the vehicle’s fuel efficiency and emissions in highway or steady-state driving conditions. According to the authors, the choice was thought to test the agent in two opposite scenarios the agent had never seen before and compare its performance again with respect to the optimal control strategy identified by DP. In table 7.3 are presented the

benchmark values set by DP on the two cycles in terms of final SoC and fuel consumption.

Table 7.3 Dynamic Programming optimal control strategy results for the two additional cycles.

	SoC	FC	FC
	$[-]$	$[kg]$	$[l/100km]$
<i>FTP75</i>	0.5	0.7815	5.2569
<i>HFET</i>	0.5	0.5706	4.1172

As already done in the tests for the rewards evaluation, the SoC and fuel consumption trends are presented in the following to evaluate the goodness of the agent over the new cycles. The performance assessment starts with FTP75.

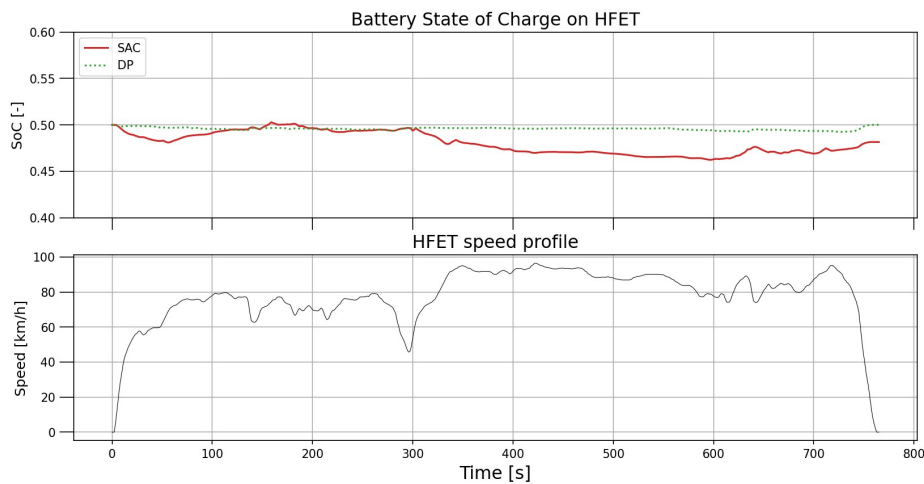


Fig. 7.23 SoC trend performed by the SAC agent over the FTP75 standardized cycle plotted against the speed profile.

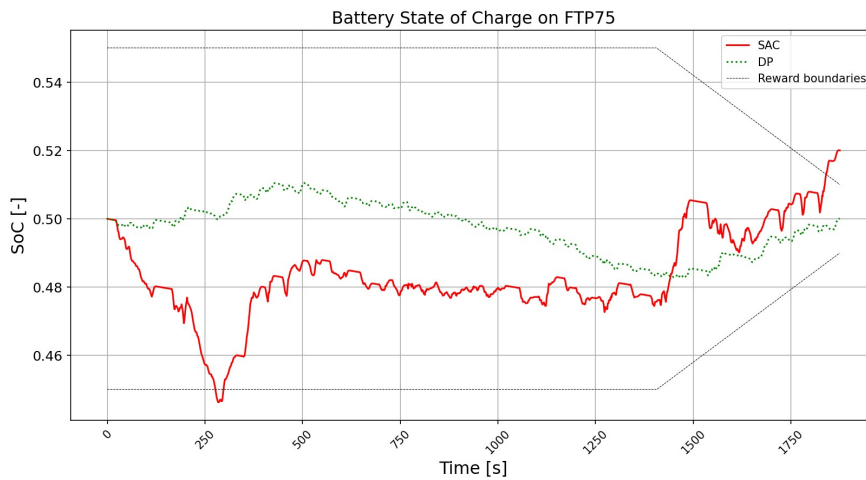


Fig. 7.24 SoC trend performed by the SAC agent over the FTP75 standardized cycle.

Looking at figure 7.24 it's possible to see almost the same behavior the agent showed in figure 7.21. At the beginning of both cycles, the agent discharges the battery and then it tries to keep the SoC fluctuating around the same charge level. This trend is more severe on the FTP75 where the initial depletion makes the SoC fall under the value of 0.46 to then take it back to 0.48 and keep it steady until the last quarter of the mission. To understand the behavior of the agent in the last part of the cycle, it's necessary to look at figure 7.23. Differently from the WLTC where the last quarter simulates a highway or extra-urban scenario, in FTP75, the environment represents just an urban scenario. This being said, the agent performs well at the end of the FTP75, probably because it has learned that the SoC boundaries are narrowing as the task completion is approached. It recharges slightly the battery, between $t = 1400s$ and $t = 1500s$, to bring the charge level around the target value (i.e. 0.5) and tries to keep it until the end. In this last part, the agent also collects energy from the steep decelerations imposed by the cycle, that it carries out in regenerative braking mode as imposed by the rules set at the boundaries by the authors, making the SoC to overcome the upper penalty threshold.

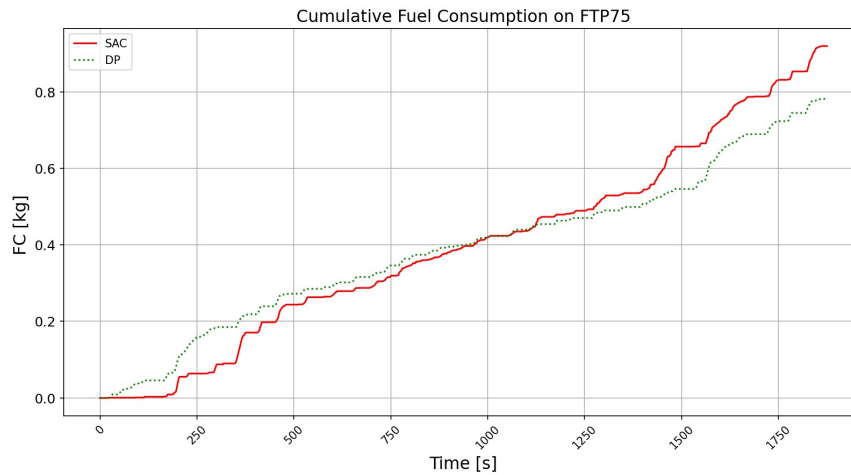


Fig. 7.25 FC trend performed by the SAC agent over the FTP75 standardized cycle.

For the sake of completeness, the cumulative fuel consumption is reported in figure 7.25 where the agent scores a value of 919.2 g which translates to 6.1826 l/100 km. Being the final SoC for the SAC agent equal to 0.52, it's clear that the fuel consumption is higher than the one obtained by DP and reported in table 7.3, equal to 5.2569 l/100km.

Table 7.4 Results comparison between Dynamic Programming optimal control strategy and SAC control strategy on FTP75.

	SoC	FC	FC
	[—]	[kg]	[l/100km]
<i>DP</i>	0.5	0.7815	5.2569
<i>SAC</i>	0.52	0.9192	6.1826

Moving to HFET cycle, it's now time to evaluate the behavior of the agent in an unknown almost steady-state scenario.

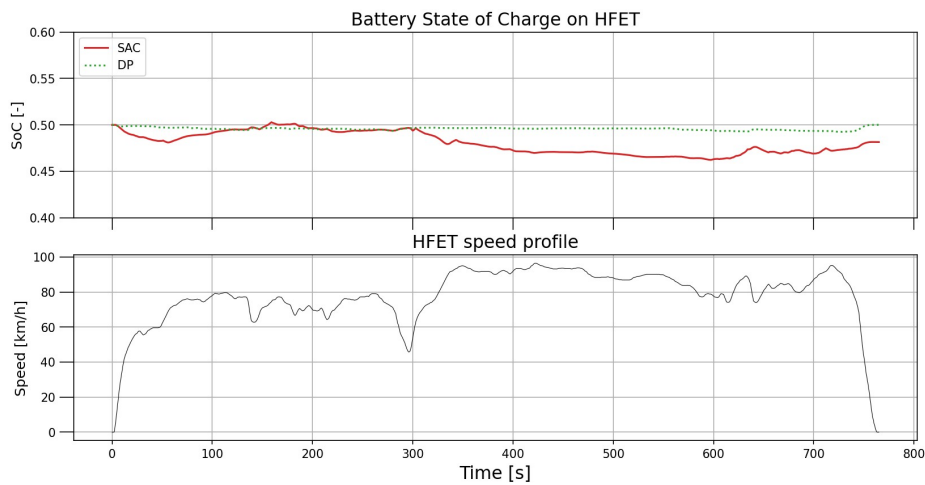


Fig. 7.26 SoC trend performed by the SAC agent over the HFET standardized cycle, plotted against the speed profile.

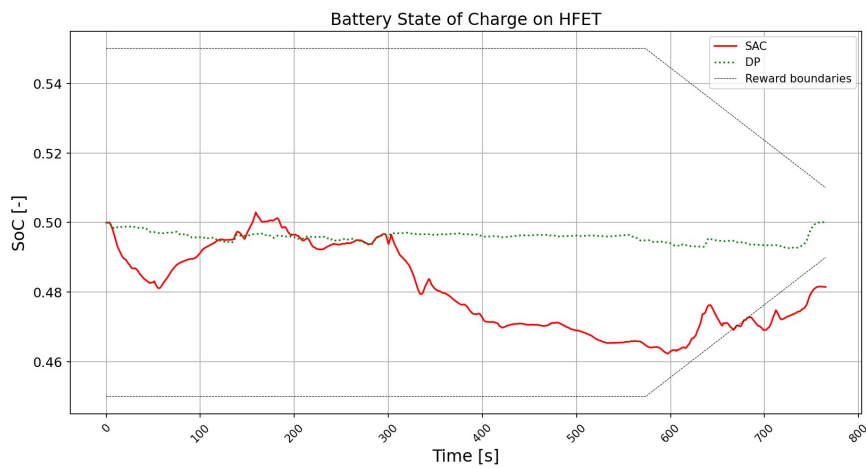


Fig. 7.27 SoC trend performed by the SAC agent over the HFET standardized cycle.

As figure 7.26 depicts, the speed profile is less fluctuating than FTP75 or WLTC, it involves higher speeds so it's not trivial for the agent deciding what to do, if exploiting the EM power to steadily propel the vehicle, or just to cover

some little transients dynamic or, on the contrary, to shut down torque delivery of the EM and rather using it as a generator run by the torque in excess provided by the ICE. Figure 7.27 shows that, in this case, the agent is not able to handle the power request while ensuring a charge-sustaining condition. At the beginning, it strives to keep the charge level around the target value but after only 300 seconds, the SoC starts to diverge and will never rise again at levels close to the DP ones. According to this battery charge-depleting attitude, lower fuel consumption is expected for the SAC agent that closes the cycle with an SoC level of 0.4814.

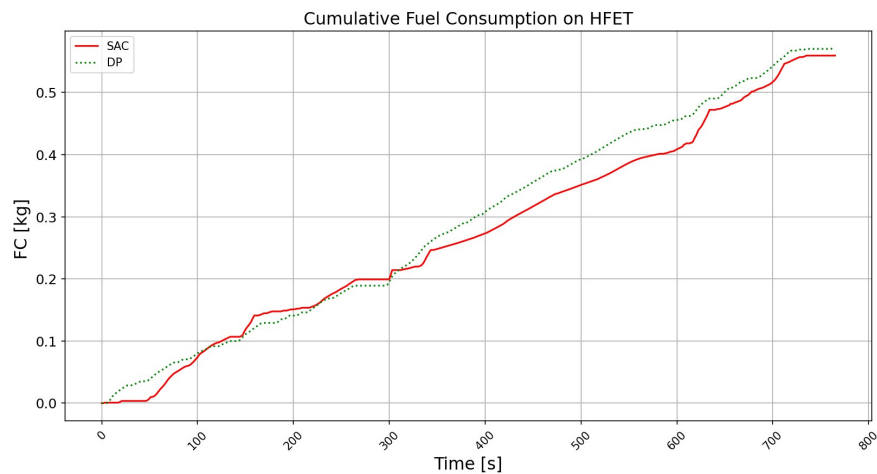


Fig. 7.28 FC trend performed by the SAC agent over the HFET standardized cycle.

Figure 7.28 confirms what was expected: the SAC agent realizes a fuel consumption of 559.3 g, which translates into 4.04 l/100km.

Table 7.5 Results comparison between Dynamic Programming optimal control strategy and SAC control strategy on HFET.

	SoC	FC	FC
	[—]	[kg]	[l/100km]
<i>DP</i>	0.5	0.5706	4.1172
<i>SAC</i>	0.4814	0.5593	4.0358

To sum up, the agent selected at the end of section 7.1 guarantees satisfying performance even in tasks never experienced before. The cycles sample used to assess the generalization capability of the agent is too small to make precise statements about it, but it shows that at least the agent doesn't fall in unacceptable fluctuating behavior of the SoC and it's able to stay in the neighborhood of the DP results, even if better results must, and can, be obtained.

Chapter 8

Conclusions and Future Work

In the strive of the world to pursue carbon neutrality, Hybrid Electric Vehicles (HEV) rose as an effective solution thanks to their capability to combine the benefits of an electric and conventional powertrain. Nonetheless, the additional degree of freedom, key to their efficiency, implies a greater complexity that must be handled by an additional control level: the Energy Management System (EMS). During the past years, several approaches have been investigated to identify the optimal control strategies but nowadays, Artificial Intelligence (AI) algorithms have gained popularity due to their ability to process vast amounts of data.

In this framework, this thesis project has presented a new methodology to develop, test and tune an AI-based EMS. After having explained the theory and the potential behind the concept of an agent obtaining rewards from its interaction with an environment characterizing the branch of Machine Learning (ML) called Reinforcement Learning (RL), the workflow directed to the development of a virtual simulation platform. The vehicle digital twin to design and evaluate the innovative EMS was developed based on some experimental data gathered from a commercially available plug-in HEV (pHEV) with reverse engineering techniques. Once the virtual test rig has been validated against another simulation model, the Soft-Actor Critic agent was selected as one of the most promising RL algorithms currently used and trained over the Worldwide Harmonized Light-duty vehicles Test Cycle (WLTC). This agent turned out to quickly learn from the environment, taking the trained model to convergence in few hundreds episodes. The SAC agent was then tested with two different rewards set in various configurations, showing results very close to the solution provided by DP even with the less performant configurations. The best SAC model was able to operate the vehicle

over the WLTC in a perfect Charge Sustaining (CS) mode while increasing the fuel consumption of only 8.6% with respect to the benchmark set by DP.

Future work will be aimed at achieving the targets discussed here below:

- The Python environment where the vehicle digital twin has been modelled offers a rich ecosystem of libraries and infrastructure. In this context, several tools are available to carry out an optimization of all the Neural Networks (NN) parameters in order to obtain the most performant algorithm for the specific study case;
- Expanding the batch of possible rewards to better guide the RL agent in the learning while acting;
- Increasing the complexity of the vehicle model, introducing transient dynamics and a driver loop to better represent the real driving conditions. This step can be performed either before or after the training. In the first it would train an agent to make it aware of the real environment dynamics, while in the second it would be useful to assess the generalization capability of the agent;
- Training the agent over different cycles, maybe even through a parallel training, to give it much more comprehension of the world and increasing the chance of obtaining an agent suitable for every possible driving condition.

Acknowledgements

With the completion of this thesis project, another significant chapter of my life comes to an end. It's worthless to stress how much I have grown during this further step in my career, both professionally and personally. Therefore, I feel obliged to express my heartfelt gratitude to those who supported me throughout these years.

My first and foremost thought is for my parents, brother, and sister whose tremendous support helped me get through every challenge. I feel blessed to have been raised with a boundless curiosity about what surrounds me and the endless hunger to learn something new every day.

I wish to thank Prof. Federico Millo and Luciano Rolando, who welcomed me into their research team and provided me with the opportunity to carry out my thesis under their supervision even when it was not taken for granted. I cannot but thank Luca and, particularly, Luigi, my academic tutors, who were able to cope with my expansive and hectic attitude, guiding me day by day in the development of my thesis. I could not have asked for better mentors. My appreciation also goes to Prof. Moein Mehrtash, who accepted me at McMaster University, gifting me with one of the most rewarding experiences of my life.

Last but certainly not least, I would like to extend my special thanks to all my friends now scattered everywhere: from lifelong companions to those who briefly crossed my path. While they may not have directly contributed to my accomplishments, they have made life what it's meant to be: a journey.

Appendix A

Python Code

A.1 Engine Class (extract)

```
1 class Engine:
2     def __init__(self, engine=pd.DataFrame):
3         if not engine.empty:
4             self.Fuellhvh = engine["Fuel_LHV"][1]
5             self.Fueldensityvalue = engine["Fuel_
6                 density_value"][1]
7             self.Idlespeed = engine["Idle_speed"
8                 ][1]
9             self.Maxspeed = engine["Max_speed"][1]
10            self.Bore = engine["Bore"][1]
11            self.Stroke = engine["Stroke"][1]
12            self.Ncylinder = engine["N_cylinder"
13                ][1]
14            self.Inertia = engine["Inertia"][1]
15            self.Displacement = engine["
16                Displacement"][1]
17            self.CO2molar mass = engine["CO2_molar_
18                mass"][1]
19            self.Fuelmolar mass = engine["Fuel_molar_
20                mass"][1]
21            self.Cutofflimit = engine["Cut-off_
22                limit"][1]
```

```
16         else:
17             self.Fuellhvh = []
18             self.Fueldensityvalue = []
19             self.Idlespeed = []
20             self.Maxspeed = []
21             self.Bore = []
22             self.Stroke = []
23             self.Ncylinder = []
24             self.Inertia = []
25             self.Displacement = []
26             self.CO2molarmass = []
27             self.Fuelmolarmass = []
28             self.Cutofflimit = []
29
30     def Maxtrq(self, engine_limits = pd.DataFrame):
31         var_info_dict = {"Speed": engine_limits["Speed"][0], "MaxTrq": engine_limits["MaxTorque"][0]}
32         self.Maxtrq_tab = LookUp1D(engine_limits["Speed"][1:], engine_limits["MaxTorque"][1:], var_info_dict)
33
34
35     def FuelTrqMap(self, engine_fueltrqmap = pd.DataFrame):
36         engine_fueltrqmap = np.array(engine_fueltrqmap)
37         engine_fueltrqmap_x = engine_fueltrqmap[1, 2:]
38         engine_fueltrqmap_y = engine_fueltrqmap[2:, 1]
39         engine_fueltrqmap_z = engine_fueltrqmap[2:, 2:]
40         var_info_dict = {"Speed": engine_fueltrqmap[2, 0], "Torque": engine_fueltrqmap[0, 1], "FuelRate": engine_fueltrqmap[0, 0]}
```

```
41     self.FuelTrqMap_tab = LookUp2D(
        engine_fueltrqmap_x, engine_fueltrqmap_y,
        engine_fueltrqmap_z, var_info_dict)
```

A.2 Reset Method

```
1 def reset(self, seed=None, options=None):
2
3     #Method to reset environment to initial state
4     #and output initial observation
5     self.time_idx = 0
6     self.setSimulationInput(self.time_idx)
7
8     self.ice_state = 0
9     self.SOC = 0.5
10
11     self.RoadToGb(self.ice_state, self.speed, self.
12     acc, self.gear)
13     gbPwr_in = self.gbPwr_in
14     gbSpd_in = self.gbSpd_in
15     fdSpd = self.fdSpd
16
17     self.state = {"Speed": self.speed, "Power_
18     Demand": self.gbPwr_in,
19     "Gearbox_speed": self.gbSpd_in,
20     "SOC": self.SOC,
21     "Traveled_distance": self.
22     travel}
23
24     InputState = np.array([self.speed, self.
25     gbPwr_in, self.gbSpd_in, self.SOC,
26     self.travel]).astype(np
27     .float32)
```

```

24     self.Normalization = StateNormalization(self)
25
26     self.observation = self.Normalization.Normalize
        (InputState, 1, 0).astype(np.float32)
27
28
29     self.Output = {"SOC": np.zeros(1), "FC": np.
        array([]), "EM_Power": np.array([]), "
        ICE_Power": np.array([]), "GB_Power": np.
        array([]), "FD_Power": np.array([]), "
        BT_Power": np.array([]), "Action": np.array
        ([]) }
30     self.Output["SOC"][0] = self.SOC
31
32     #The episode has just started so
33     self.done = False
34     self.truncated = False
35
36     if seed is not None:
37         random.seed(seed)
38
39     self.info = dict()
40
41     return self.observation, self.info

```

A.3 Step Method

```

1 def step(self, action):
2     # Method that apply system dynamics and
        simulates the environment with the given
3     # action for one step.
4
5     if action > 0.075:
6         self.ice_state = 1
7     else:
8         self.ice_state = 0

```

```
9
10
11     self.RoadToGb(self.ice_state, self.speed, self.
12         acc, self.gear)
13     fdPwr = self.fdPwr
14     gbPwr_in = self.gbPwr_in
15     gbSpd_in = self.gbSpd_in
16     fdSpd = self.fdSpd
17     ice_state = self.ice_state
18     u = action
19
20     self.Powersplit(u, gbPwr_in, gbSpd_in,
21         ice_state, self.acc, self.speed)
22     ice_speed = self.iceSpd
23     ice_Pwr = self.icePwr
24     em_speed = self.emSpd
25     em_Pwr = self.emPwr
26
27     self.FuelConsumption(ice_speed, ice_Pwr, self.
28         ice_state)
29     FC = self.fuel_rate
30
31     self.ElectricalUnits(em_speed, em_Pwr, self.SOC
32         )
33     btPwr = self.battPwr
34     self.SOC = self.batt_soc[0]
35
36     self.Output["SOC"] = np.append(self.Output["SOC
37         "], self.SOC)
38     self.Output["FC"] = np.append(self.Output["FC"
39         ], FC)
40     self.Output["EM_Power"] = np.append(self.Output
41         ["EM_Power"], em_Pwr)
42     self.Output["ICE_Power"] = np.append(self.
43         Output["ICE_Power"], ice_Pwr)
44     self.Output["GB_Power"] = np.append(self.Output
45         ["GB_Power"], gbPwr_in)
```

```
37     self.Output["FD_Power"] = np.append(self.Output
38         ["FD_Power"], fdPwr)
39     self.Output["BT_Power"] = np.append(self.Output
40         ["BT_Power"], btPwr)
41     self.Output["Action"] = np.append(self.Output["
42         Action"], action)
43
44     #Reward
45     '''Reward 1'''
46     #self.reward = float(self.k1 + (self.k2 * FC *
47         self.Ts) + \
48         #
49         (self.k3 * self.soc_gain * (self
50         .SOC - self.soc_target)**2))
51     '''Reward 3'''
52
53     if self.time_idx < 1350:
54         if self.SOC < 0.45:
55             self.P_soc = (self.SOC - self.
56                 soc_target)**2
57         elif self.SOC > 0.55:
58             self.P_soc = (self.SOC - self.
59                 soc_target) ** 2
60         else:
61             self.P_soc = 0
62     else:
63         if self.SOC > (0.67 - (self.time_idx * 8.89
64             e-5)):
65             self.P_soc = (self.SOC - self.
66                 soc_target) ** 2
67         elif self.SOC < (0.33 + (self.time_idx *
68             8.89e-5)):
69             self.P_soc = (self.SOC - self.
70                 soc_target) ** 2
71         else:
72             self.P_soc = 0
```



```
62     self.reward = float(self.k1 + (self.k2 * FC +
63         self.k3 * self.soc_gain * self.P_soc) * self.
64         Ts)
65
66     # timestep update
67     if self.time_idx < (len(self.DrivingCycle.Time)
68         - 1):
69         self.time_idx = self.time_idx + self.Ts
70
71         self.setSimulationInput(self.time_idx)
72
73         self.RoadToGb(self.ice_state, self.speed,
74             self.acc, self.gear)
75         gbPwr_in = self.gbPwr_in
76         gbSpd_in = self.gbSpd_in
77         fdSpd = self.fdSpd
78
79         self.state = {"Speed": self.speed, "Power_
80             Demand": self.gbPwr_in,
81                 "Gearbox_speed": self.
82                 gbSpd_in, "SOC": self.SOC
83                 ,
84                 "Traveled_distance": self.
85                 travel}
```

```
85     elif self.time_idx == (len(self.DrivingCycle.  
86         Time) - 1):  
87         self.done = True  
88         print(f"done_{self.time_idx}")  
89     elif self.SOC <= 0.3 and self.SOC >= 0.7:  
90         self.truncated = True  
91         print(f"truncated_{self.time_idx}")  
92  
93     self.info = {}  
94  
95     return self.observation, self.reward, self.done  
96         , self.truncated, self.info
```

References

- [1] Luciano Rolando. *An Innovative Methodology for the Development of HEVs Energy Management System*. PhD thesis, Politecnico di Torino, 2012.
- [2] E. Spessa. Energy management for hybrid and electric vehicles, 2021. -.
- [3] Luca Pulvirenti. *Development of Innovative Methodologies to Support the Design of Connected and Electrified Vehicles*. PhD thesis, Politecnico di Torino, 2023.
- [4] e nsight. Mixed or complex hybrid. <https://www.e-nsight.com/2020/04/10/mixed-or-complex-hybrid/>, 2020, online documentation.
- [5] S. Vaschetto and L. Rolando. Hybrid propulsion system, 2021. -.
- [6] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.
- [7] MathWorks. Soft actor-critic (sac) agents. <https://it.mathworks.com/help/reinforcement-learning/ug/sac-agents.html>, 2023, online documentation.
- [8] Atriya Biswas and Ali Emadi. Energy management systems for electrified powertrains: State-of-the-art review and future trends. *IEEE Transactions on Vehicular Technology*, 68(7):6453–6467, 2019.
- [9] Simona Onori, Lorenzo Serrao, and Giorgio Rizzoni. *Hybrid electric vehicles: Energy management strategies*. Springer London, 2016.
- [10] UNECE. European green deal: commission proposes transformation of eu economy and society to meet climate ambitions,” accessed october 2021. https://ec.europa.eu/commission/presscorner/detail/en/IP_21_3541., 2021, online documentation.
- [11] IEA. Largest end uses of energy by sector in selected iea countries, 2019. <https://www.iea.org/data-and-statistics/charts/largest-end-uses-of-energy-by-sector-in-selected-iea-countries-2019>., 2022, online documentation.
- [12] ACEA. Electric vehicles: tax benefits and purchase incentives. https://www.acea.auto/files/Electric_vehicles-Tax_benefits_purchase_incentives_European_Union_2020.pdf., 2021, online documentation.

-
- [13] EEA. New registrations of electric vehicles in europe. <https://www.eea.europa.eu/ims/new-registrations-of-electric-vehicles>., 2022, online documentation.
- [14] IEA. Global electric car sales have continued their strong growth in 2022 after breaking records last year. <https://rb.gy/3wf5c>., 2022, online documentation.
- [15] Lars-Henrik Björnsson and Sten Karlsson. Electrification of the two-car household: Phev or bev? *Transportation Research Part C: Emerging Technologies*, 85:363–376, 2017.
- [16] Antonio Sciarretta and Lino Guzzella. Control of hybrid electric vehicles. *IEEE Control Systems Magazine*, pages 60–70, April 2007.
- [17] Dai-Duong Tran, Majid Vafaiepour, Mohamed El Baghdadi, Ricardo Barroero, Joeri Van Mierlo, and Omar Hegazy. Thorough state-of-the-art analysis of electric and hybrid vehicle powertrains: Topologies and integrated energy management strategies. *Renewable and Sustainable Energy Reviews*, 119:109596, 2020.
- [18] Forbes. What is the artificial intelligence revolution and why does it matter to your business? <https://shorturl.at/qvxSY>., 2022, online documentation.
- [19] Tom M Mitchell. *Machine learning*, 1997.
- [20] Yuxi Li. Reinforcement learning applications. *arXiv preprint arXiv:1908.06973*, 2019.
- [21] Iqbal H Sarker. Machine learning: Algorithms, real-world applications and research directions. *SN computer science*, 2(3):160, 2021.
- [22] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.
- [23] Dimitri Bertsekas. *Dynamic Programming and Optimal Control*. Athena Scientific, Belmont, MA, 1995.
- [24] Nicholas Metropolis and Stanislaw Ulam. The monte carlo method. *Journal of the American statistical association*, 44(247):335–341, 1949.
- [25] Bradley Efron. Bootstrap methods: another look at the jackknife. In *Breakthroughs in statistics: Methodology and distribution*, pages 569–593. Springer, 1992.
- [26] Claude Elwood Shannon. A mathematical theory of communication. *The Bell system technical journal*, 27(3):379–423, 1948.
- [27] Federico Millo, Luciano Rolando, and Emanuele Servetto. Development of a control strategy for complex light-duty diesel-hybrid powertrains. Technical report, SAE Technical Paper, 2011.

- [28] Joseph M Morbitzer. *High-level modeling, supervisory control strategy development, and validation for a proposed power-split hybrid-electric vehicle design*. PhD thesis, The Ohio State University, 2005.
- [29] Theo Hofman, Maarten Steinbuch, Roell Van Druten, and Alex Serrarens. Rule-based energy management strategies for hybrid vehicle drivetrains: a fundamental approach in reducing computation time. *IFAC Proceedings Volumes*, 39(16):740–745, 2006. 4th IFAC Symposium on Mechatronic Systems.
- [30] N. Jalil, NA Kheir, and Mutasim Salman. A rule-based energy management strategy for a series hybrid vehicle. *Proceedings of the 1997 American Control Conference*, 1, 1997.
- [31] Mutasim Salman, NJ Schouten, and NA Kheir. Control strategies for parallel hybrid vehicles. *Proceedings of the 2000 American Control Conference.*, 1(6):524–528, 2000.
- [32] Chan-Chiao Lin, Huei Peng, JW Grizzle, and Jun-Mo Kang. Power management strategy for a parallel hybrid electric truck. *IEEE Transactions on Control Systems Technology*, 11(6):839–849, 2003.
- [33] Richard Bellman. *Dynamic Programming*. Dover Publications, 1957.
- [34] H. Kaufman. The mathematical theory of optimal processes, by I. S. Pontryagin, V. G. Boltyanskii, R. V. Gamkrelidze, and E. F. Mishchenko. Authorized translation from the Russian. Translator: K. N. Trirogoff, Editor: L. W. Neustadt. Interscience Publishers (Division of John Wiley and Sons, Inc., New York) 1962. viii 360 pages. *Canadian Mathematical Bulletin*, 7(3):500–500, 1964.
- [35] Gino Paganelli, Thierry-Marie Guerra, Sebastien Delprat, Jean-Jacques Santin, M. Delhom, and E. Combes. Simulation and assessment of power control strategies for a parallel hybrid car. *Int. J. of Automobile Engineering*, 214:705–717, 07 2000.
- [36] Basil Kouvaritakis and Mark Cannon. *Model Predictive Control: Classical, Robust and Stochastic*. Springer, 01 2016.
- [37] A. Brahma, Yann Guezennec, and Giorgio Rizzoni. Optimal energy management in series hybrid electric vehicles. In *American Control Conference, 2000. Proceedings of the 2000*, volume 1, pages 60 – 64 vol.1, 10 2000.
- [38] Xueqin Lü, Siwei Li, XiangHuan He, Chengzhi Xie, Songjie He, Yuzhe Xu, Jian Fang, Min Zhang, and Xingwu Yang. Hybrid electric vehicles: A review of energy management strategies based on model predictive control. *Journal of Energy Storage*, 56:106112, 2022.
- [39] Fengqi Zhang, Lihua Wang, Serdar Coskun, Hui Pang, Yahui Cui, and Junqiang Xi. Energy management strategies for hybrid electric vehicles: Review, classification, comparison, and outlook. *Energies*, 13(13):3352, 2020.

- [40] Yang Zhou, Alexandre Ravey, and Marie-Cécile Péra. A survey on driving prediction techniques for predictive energy management of plug-in hybrid electric vehicles. *Journal of Power Sources*, 412:480–495, 2019.
- [41] Giuseppe DiPierro, Enrico Galvagno, Gianluca Mari, Federico Millo, Mauro Velardocchia, and Alessandro Perazzo. A reverse-engineering method for powertrain parameters characterization applied to a p2 plug-in hybrid electric vehicle with automatic transmission. *SAE International Journal of Advances and Current Practices in Mobility*, 3(2020-37-0021):715–730, 2020.
- [42] Federico Millo, Luciano Rolando, Luca Pulvirenti, and Giuseppe Di Pierro. A methodology for the reverse engineering of the energy management strategy of a plug-in hybrid electric vehicle for virtual test rig development. *SAE International Journal of Electrified Vehicles*, 11(14-11-01-0009):113–132, 2021.
- [43] 2015 UNECE: Geneva, Switzerland. 83–revision 5. uniform provisions concerning the approval of vehicles with regard to the emission of pollutants according to engine fuel requirements; unece: Geneva, switzerland, 2015. <https://tinyurl.com/cxmdcdzy>.
- [44] 2015 UNECE: Geneva, Switzerland. Unece global technical regulation no. 15 worldwide harmonized light vehicles test procedure. <https://tinyurl.com/859eyr6c>.
- [45] Federico Millo, Luciano Rolando, and Maurizio Andreatta. Numerical simulation for vehicle powertrain development. In Jan Awrejcewicz, editor, *Numerical Analysis*, chapter 24. IntechOpen, Rijeka, 2011.
- [46] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pages 1861–1870. PMLR, 2018.
- [47] Olle Sundström, Daniel Ambühl, and Lino Guzzella. On implementation of dynamic programming for optimal control problems with final state constraints. *Oil & Gas Science and Technology—Revue de l’Institut Français du Pétrole*, 65(1):91–102, 2010.
- [48] United States Government. United states environmental protection agency. <https://www.epa.gov/>, 2023, online documentation.